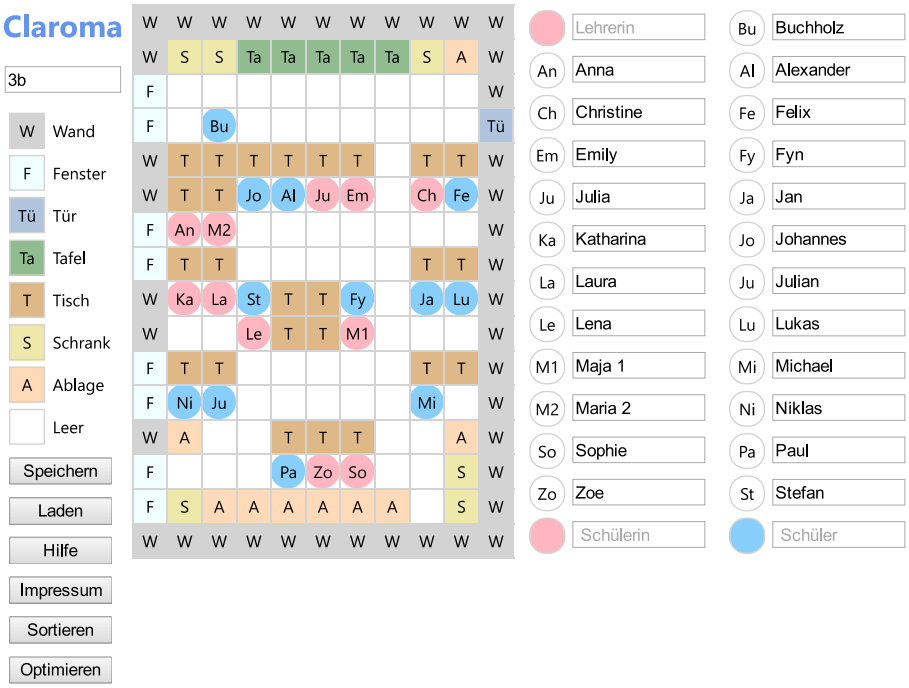


Claroma

Intelligente Sitzplanerstellung

Jörg J. Buchholz
Florian Buchholz

24. August 2016

Inhaltsverzeichnis

Teil I

Bedienungsanleitung

1 Einführung

Claroma (**Clas**s**ro**om **Ma**nagement) ist ein Programm zur manuellen und halbautomatischen Sitzplanerstellung in einer Schulklasse. Lehrkräfte können eine Liste ihrer Schülerinnen und Schüler anlegen und diese zusammen mit Tischen und anderen Gegenständen in einem Raster grafisch anordnen. In der anschließenden Optimierungsphase ist es dann sehr leicht, Schülerinnen und Schüler manuell „umzusetzen", um persönlichen Wunschbedingungen der Form

- Elias möchte neben Julian sitzen.
- Lena darf auf gar keinen Fall neben Sarah sitzen.
- Marcel sollte möglichst in Türnähe sitzen.
- Laura muss möglichst weit hinten sitzen.

gerecht zu werden.

Wenn sie mit ihrem manuell erstellten Sitzplan noch nicht ganz zufrieden ist, kann die Lehrkraft ihre Wünsche im Programm abbilden und den Sitzplan vom programminternen Optimierer nachoptimieren lassen, der manchmal tatsächlich eine Sitzanordnung findet, die die vorgegebenen Randbedingungen noch besser erfüllt.

Datenschutz Claroma speichert keine Daten im Internet. Alle Informationen (Pläne, Namen, ...) werden ausschließlich auf dem eigenen Rechner gehalten und auf Wunsch dort abgespeichert oder ausgedruckt.

2 Anwendung des Programms

In diesem Kapitel wollen wir die typische Vorgehensweise beim Arbeiten mit Claroma kennenlernen.

2.1 Die Oberfläche

In Abbildung 2.1 ist exemplarisch eine typische Arbeitsoberfläche dargestellt, auf der wir den Sitzplan erstellen.

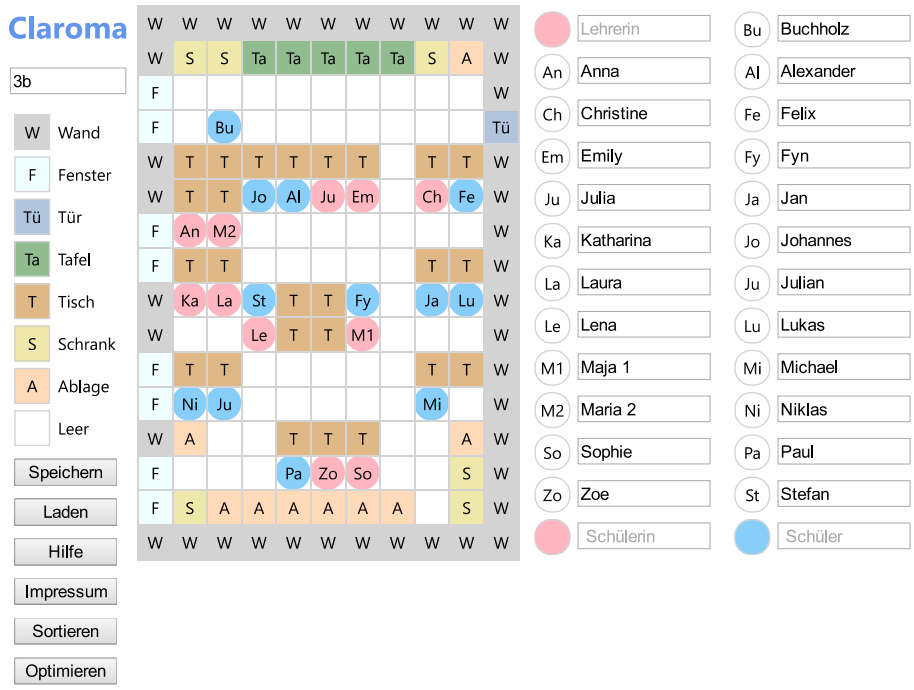


Abbildung 2.1: Sitzplanerstellung mit Claroma

In der linken oberen Ecke der Oberfläche finden wir den Programmnamen.

Direkt darunter gibt es ein Feld, in das wir den Namen der Klasse eintragen können. Dies ist insbesondere dann sinnvoll, wenn wir den Plan später abspeichern wollen, da der Klassenname dann – zur Unterscheidung – als Teil des Dateinamens verwendet wird.

Alsdann folgt eine Liste der Gegenstände, die wir durch Anklicken aufnehmen und durch erneutes Klicken im rechts daneben liegenden Raum beliebig oft absetzen können. Die ersten sieben Einträge (Wand ... Ablage) sind reale Objekte, die wir frei im Raum positionieren können. Dabei können wir jedes Objekt auch auf ein schon vorhandenes Objekt ablegen, das dadurch automatisch entfernt wird. Auf diese Weise können wir beispielsweise einfach ein Fenster oder eine Tür direkt an passender Stelle in eine Wand einbauen.

Der letzte Eintrag (Leer) dient zum Löschen. Wenn wir ihn anklicken, nehmen wir quasi einen Löschstempel auf, mit dem wir im Raum vorhandene Gegenstände (oder Schülerinnen und Schüler) durch Anklicken wieder entfernen können.

Durch Anklicken der Schaltfläche Speichern gelangen wir auf eine Seite, auf der uns der aus Klassenname und aktuellem Datum zusammengesetzte Name der Datei angezeigt wird, unter der der momentane Plan gerade abgespeichert wird.

Die Schaltfläche Laden führt uns auf eine Seite, auf der wir sowohl einen neuen, leeren Raum mit gewünschten Abmaßen (Standardwert: 20×20 Kästchen) erstellen, als auch einen vorher abgespeicherten Plan laden können. In beiden Fällen wird der gerade bearbeitete Plan unwiderruflich und ohne Nachfrage gelöscht.

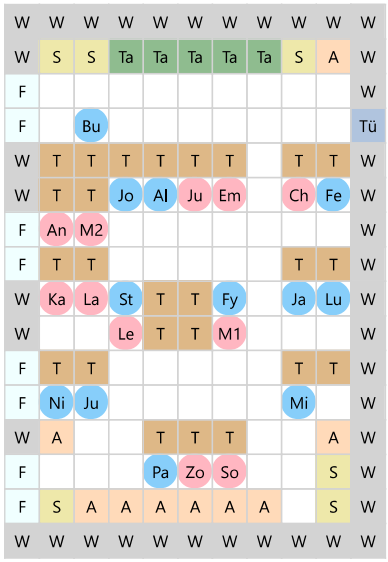
Die Schaltfläche Hilfe öffnet dieses Dokument.

Jede Website braucht nach § 5 des Telemediengesetzes eine Impressum-Seite, auf der die Kontaktdaten der Verantwortlichen (Telefonnummer, E-Mail-Adresse, ...) aufgelistet sind.

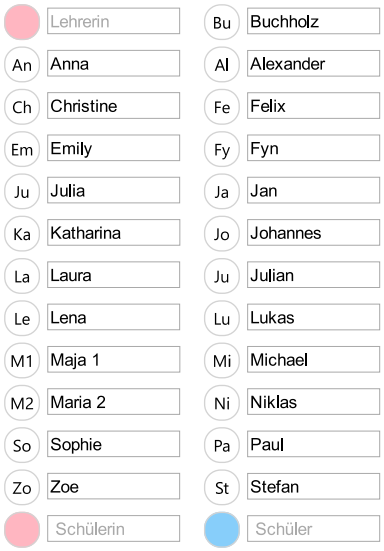
Das Anklicken der Schaltfläche Sortieren sortiert die Listen der Schülerinnen und Schüler alphabetisch. Die Sortierreihenfolge lautet dabei:

1. Sonderzeichen (*, #, ...)
2. Zahlen
3. Großbuchstaben
4. Kleinbuchstaben

Durch Anklicken der gleichnamigen Schaltfläche gelangen wir zur Seite Optimieren, auf der wir angeben können, welche Schülerinnen und Schüler gerne (oder nicht gerne) nebeneinander sitzen möchten bzw. sollten. Ein Optimierungsprogramm versucht dann, durch Umsetzen einzelner Schülerinnen und Schüler, diese Wünsche zu erfüllen.



In der Mitte von Abbildung 2.1 ist der Raum dargestellt, in dem wir Gegenstände (Tisch, Tafel, ...), Schülerinnen und Schüler positionieren können. Dazu klicken wir erst auf die Objekte, die wir einfügen wollen und dann an der passenden Stelle in den Raum. Neue Objekte überschreiben dabei alte Objekte. Nachdem wir auf Leer geklickt haben, können wir mit einem Löschstempel Objekte wieder aus dem Raum entfernen. Wenn wir im Raum auf schon gesetzte Schülerinnen und Schüler klicken, werden sie beweglich – sie kleben quasi am Mauszeiger – und wir können sie sehr einfach durch erneutes Klicken an einer anderen Position wieder absetzen.



Auf der rechten Seite von Abbildung 2.1 können wir eine Lehrerin und/oder einen Lehrer und die Schülerinnen und Schüler namentlich in eine Liste eintragen. Während wir den Namen einer Schülerin oder eines Schülers eintragen, wird unter dem aktuellen Eintrag sofort ein neues leeres Feld ergänzt.
Links neben dem Namen gibt es ein kreisförmiges Symbol (rot für weiblich, blau für männlich), in das automatisch eine Abkürzung des Namens eingetragen wird. Wir können einen Menschen entweder nur mit seinem Vornamen oder mit seinem Vor- und Nachnamen benennen. Vom Vornamen werden die ersten beiden Buchstaben in das Symbol kopiert; wenn wir einen Vor- und einen Nachnamen verwenden, werden beide Anfangsbuchstaben ins Symbol übertragen.

So können wir Maja Petersen und Maria Paulsen, die ja beide die Abkürzung „Ma" bzw. „MP" hätten, beispielsweise auch durch eine „1" und „2" im „Nachnamen" unterscheiden.

Wenn wir ein Symbol anklicken und es durch erneutes Klicken im Raum absetzen, wird das Symbol in der Liste farblos. Wenn wir ein farbloses Symbol in der Liste anklicken, wird das entsprechende Symbol wieder aus dem Raum entfernt. Auf diese Weise können wir sehr schnell alle Menschen aus dem Raum entfernen, indem wir in der Liste der Reihe nach alle Symbole anklicken.

2.2 Drucken

Um einen fertigen Plan auszudrucken, verwenden wir die Druckvorschau unseres Browsers, die wir in modernen Browsern rechts oben hinter drei Strichen oder einem Zahnrad finden. In manchen Browsern müssen wir zur Anzeige eines Druckmenüeintrags die Alt-Taste drücken. Wenn wir einen „normalen" Raumplan ausdrucken wollen, ist es sinnvoll, in der Druckvorschau als Layout das Querformat auszuwählen, damit die Objekte nicht seitlich gestaucht werden. Falls dies doch der Fall sein sollte, erlauben es manche Browser (beispielsweise Firefox), den Seiteninhalt zu skalieren und damit zum Beispiel automatisch auf die Seitengröße anzupassen.

Außerdem sollten wir die Option „Hintergrundfarben und -bilder drucken" auswählen, die sich manchmal hinter „Weitere Einstellungen" oder „Seite einrichten" versteckt. Nur durch diese Option werden die Planobjekte (Gegenstände und Menschen) farbig gedruckt. Durch die auf die Planobjekte aufgedruckten Abkürzungen ist der Plan zwar auch ohne Farbe noch lesbar; mit Farbe ist er aber natürlich übersichtlicher.

Wenn wir unseren Browser selbst nicht zu einem vernünftigen Ausdruck überreden können, bleibt uns immer noch die Möglichkeit, das Browserfenster auf den ganzen Bildschirm zu vergrößern, die Skalierung so anzupassen, dass der gesamte Plan sichtbar ist (probieren Sie mal die Strg-Taste zu drücken und gleichzeitig das Mausrad zu drehen), mit dem Snipping Tool (oder durch Drücken der Druck-Taste auf der Tastatur) ein Bildschirmfoto zu erzeugen und dieses dann beispielsweise in eine leere Word-Seite einzufügen, die wir dann „normal" ausdrucken können.

2.3 Optimieren

Nach dem Drücken der Optimieren-Schaltfläche[1] auf der Hauptseite landen wir auf der Optimierungsseite (Abbildung 2.2), auf der wir Randbedingungen wie

- Stefan möchte neben Maria sitzen.
- Stefan sollte Fyn nicht gegenüber sitzen.
- Stefan sollte während der Optimierung nicht verschoben werden.

vorgeben können. Ein Optimierungsprogramm versucht dann, diese Wünsche möglichst alle zu erfüllen, bzw. einen optimalen Kompromiss zu finden.

[1] **Vor** dem Optimieren sollten wir unbedingt den aktuellen Sitzplan abspeichern, damit wir ihn gegebenenfalls wieder laden können, wenn uns das Ergebnis der Optimierung – wider Erwarten – doch nicht gefällt.

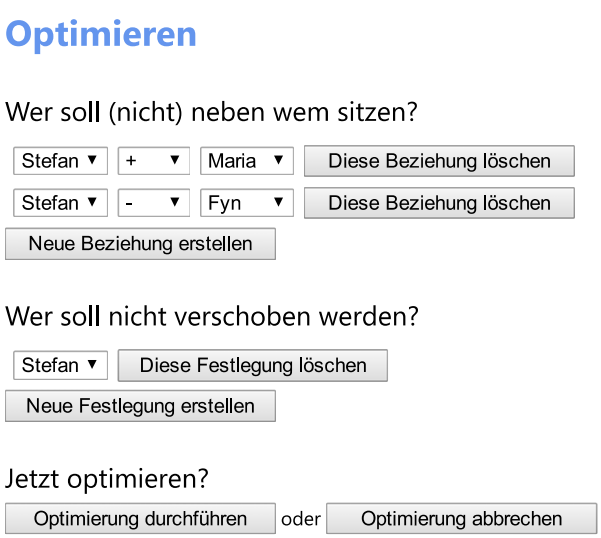


Abbildung 2.2: Optimierungsseite

Unter der Überschrift Wer soll (nicht) neben wem sitzen? können wir, nach Drücken der Neue Beziehung erstellen-Schaltfläche, beispielsweise Stefan und Maria mit einem Pluszeichen verknüpfen, um dem Optimierer mitzuteilen, dass die beiden gerne nebeneinander sitzen möchten. Entsprechend verknüpfen wir Stefan und Fyn mit einem Minuszeichen, um zu fordern, dass diese möglichst weit voneinander entfernt sitzen sollen.

Manchmal gibt es keine Möglichkeit, alle Forderungen gleichzeitig zu erfüllen. Wenn beispielsweise Stefan und Fyn beide gerne neben Maria sitzen wollen, neben Maria aber nur ein Platz frei ist, können wir die Dringlichkeit eines Wunsches durch ein doppeltes (oder sogar dreifaches) Pluszeichen ausdrücken. Auf diese Weise wird dann beispielsweise Stefan durch ein doppeltes Pluszeichen mit größerer Wahrscheinlichkeit direkt neben Maria platziert und Fyn durch ein einfaches Pluszeichen auf den dann nächstliegenden Platz gesetzt.

Durch ein o kennzeichnen wir eine zu ignorierende Beziehung. Sie wird beim nächsten Optimierungslauf[2] automatisch gelöscht. Alternativ können wir eine Beziehung auch unmittelbar mit der Schaltfläche Diese Beziehung löschen entfernen.

Unter der Überschrift Wer soll nicht verschoben werden? markieren wir alle Lehrkräfte[3], Schülerinnen und Schüler, die während der Optimierung nicht verschoben werden sollen. Sie können natürlich trotzdem in Beziehungen auftreten, solange ihr Beziehungspartner[4] verschoben werden darf.

[2] Auch unsinnige Beziehungen, bei denen ein Partner mit sich selbst verknüpft wird, werden vom Optimierer automatisch gelöscht.

[3] Der Optimierer verwendet nur die im Sitzplan gesetzten Plätze. Alternativ können wir also die Lehrkräfte vor der Optimierung einfach aus dem Sitzplan entfernen, um zu verhindern, dass der Optimierer sie „versetzt" und ihren Platz mit einer Schülerin oder einem Schüler besetzt.

[4] Natürlich wäre es ebenso sinnlos, eine Beziehung zwischen zwei Partnern zu definieren, die beide nicht verschoben werden dürfen. Der Optimierer ignoriert eine solche Beziehung, löscht sie aber nicht.

Das Drücken der Optimierung durchführen-Schaltfläche unter der Überschrift Jetzt optimieren? startet dann den Optimierer. Alternativ schließt die Schaltfläche Optimierung abbrechen die Optimierungsseite und verwirft alle neu definierten Forderungen.

2.3.1 Nicht-personenbezogene Wünsche

Manchmal gibt es auch nicht-personenbezogene Wünsche wie

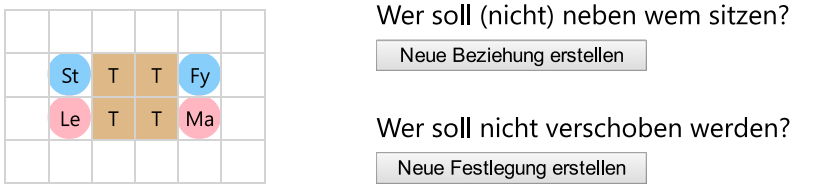
- Marcel sollte möglichst in Türnähe sitzen.
- Laura muss möglichst weit hinten sitzen.

Da aber in der Beziehungsliste nur Personen (aber keine Gegenstände oder abstrakte Eigenschaften wie vorne, links oder in der Mitte) auftreten, können wir in diesen Fällen einfach virtuelle Schülerinnen oder Schüler definieren, denen wir möglichst sinnvolle Namen wie Tafel, Fenster oder hinten links geben, die wir an geeigneter Stelle als nicht verschiebbar im Sitzplan platzieren, um dann Beziehungen zwischen ihnen und den Personen zu definieren.

Wir können sogar beispielsweise mehrere „virtuelle Fenster" (F 1, F 2, ...) einführen und jeweils vor jedes reale Fenster fest im Sitzplan platzieren. Wenn wir dann eine Person in gleichrangigen Beziehungen mit jedem Fenster verknüpfen, kann sich der Optimierer sogar aussuchen, vor welches Fenster er die Person platziert.

2.3.2 Beispiel

Das in Abbildung 2.3 dargestellte sehr einfache[5] Beispiel soll die Vorgehensweise beim Optimieren deutlich machen.



(a) Stefan, Fyn, Lena und Maria sitzen an einem Vierertisch.

(b) Noch sind keine Forderungen gestellt.

Abbildung 2.3: Vor der Optimierung

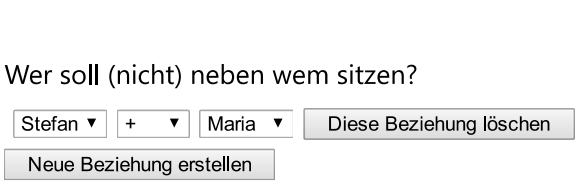
Wir haben in Abbildung 2.3a vier Schülerinnen und Schüler (Stefan, Fyn, Lena und Maria) ohne besondere Berücksichtigung von Zu- oder Abneigungen an einem Vierertisch gesetzt. Damit haben wir festgelegt, wo im Raum die vier Sitzplätze sein sollen;

[5] In der Praxis werden auch umfangreiche Pläne mit mehr als 30 Schülerinnen und Schüler innerhalb weniger Sekunden optimiert.

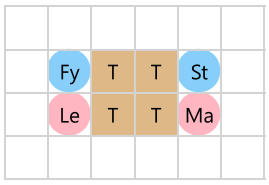
die Verteilung der Schülerinnen und Schüler auf die Plätze wollen wir jetzt aber dem Optimierer überlassen.

Nach dem Aufruf der Optimierungsseite sehen wir in Abbildung 2.3b, dass noch keine Wünsche definiert sind.

Um nun den Optimierer anzuweisen, Stefan und Maria möglichst nebeneinander zu setzen, wählen wir auf der Optimierungsseite (Abbildung 2.4a) in der linken Liste[6] Stefan und in der rechten Liste Maria aus und verbinden beide mit einem Pluszeichen.



(a) Stefan und Maria möchten nebeneinander sitzen.

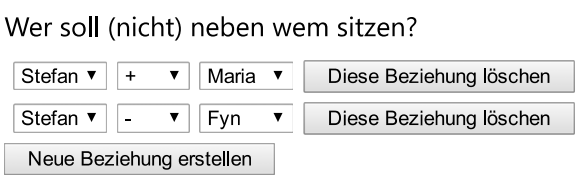


(b) Stefan und Maria sitzen jetzt nebeneinander.

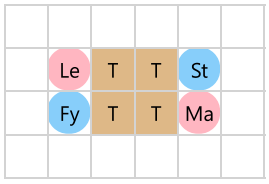
Abbildung 2.4: Pluszeichen: Platzierung möglichst nahe zusammen

Nach der Optimierung hat dann in der Tat Stefan seinen Platz mit Fyn getauscht und sitzt wunschgemäß neben Maria (Abbildung 2.4b).

Im nächsten Schritt[7] soll jetzt verhindert werden, dass Stefan und Fyn einander direkt gegenüber sitzen. Dazu wird eine zweite Beziehung erstellt, in der Stefan und Fyn mit einem Minuszeichen verknüpft werden (Abbildung 2.5a).



(a) Stefan und Fyn sollen einander nicht gegenüber sitzen.



(b) Stefan und Fyn sitzen einander jetzt nicht mehr gegenüber.

Abbildung 2.5: Minuszeichen: Platzierung möglichst weit voneinander entfernt

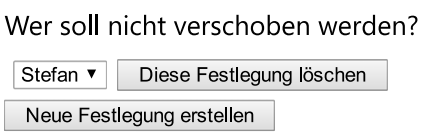
Der Optimierer versucht dann, die beiden möglichst weit voneinander entfernt zu platzieren, was in diesem trivialen Beispiel an einem Vierertisch natürlich nur dazu führt,

[6] Die Reihenfolge in der Liste ist dabei nicht alphabetisch, sondern spiegelt die Anordnung der Schülerinnen und Schüler im Raum wieder: Wie beim Lesen eines Textes wird der Raum zeilenweise von oben nach unten durchsucht.

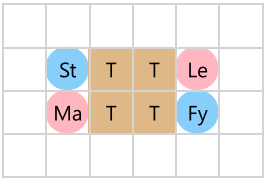
[7] Natürlich können wir alle Wünsche auch auf einen Schlag definieren und abarbeiten lassen.

dass Fyn seinen Platz mit Lena tauscht und damit wunschgemäß maximal weit von Stefan entfernt sitzt (Abbildung 2.5b).

Wenn wir, wie in Abbildung 2.6 dargestellt, sicherstellen möchten, dass Stefan seinen Platz während der Optimierung behält, können wir dies in der entsprechenden Liste auswählen (Abbildung 2.6a).



(a) Stefan soll nicht verschoben werden.



(b) Stefan bleibt an seinem ursprünglichen Platz sitzen.

Abbildung 2.6: Verschiebungsunterdrückung

Wenn wir jetzt wieder mit der Ursprungsanordnung gemäß Abbildung 2.3a beginnen und zusätzlich zu Stefans Fixierung die beiden Bedingungen gemäß Abbildung 2.5a stellen, rückt der Optimierer Maria an Stefan heran und entfernt Fyn möglichst weit von Stefan, ohne – bezogen auf Abbildung 2.3a – Stefan selbst zu bewegen (Abbildung 2.6b).

Teil II

Unter der Haube

3 Blockschaltbild

Abbildung 3.1 zeigt das Claroma-Blockschaltbild ohne die Optimierung.

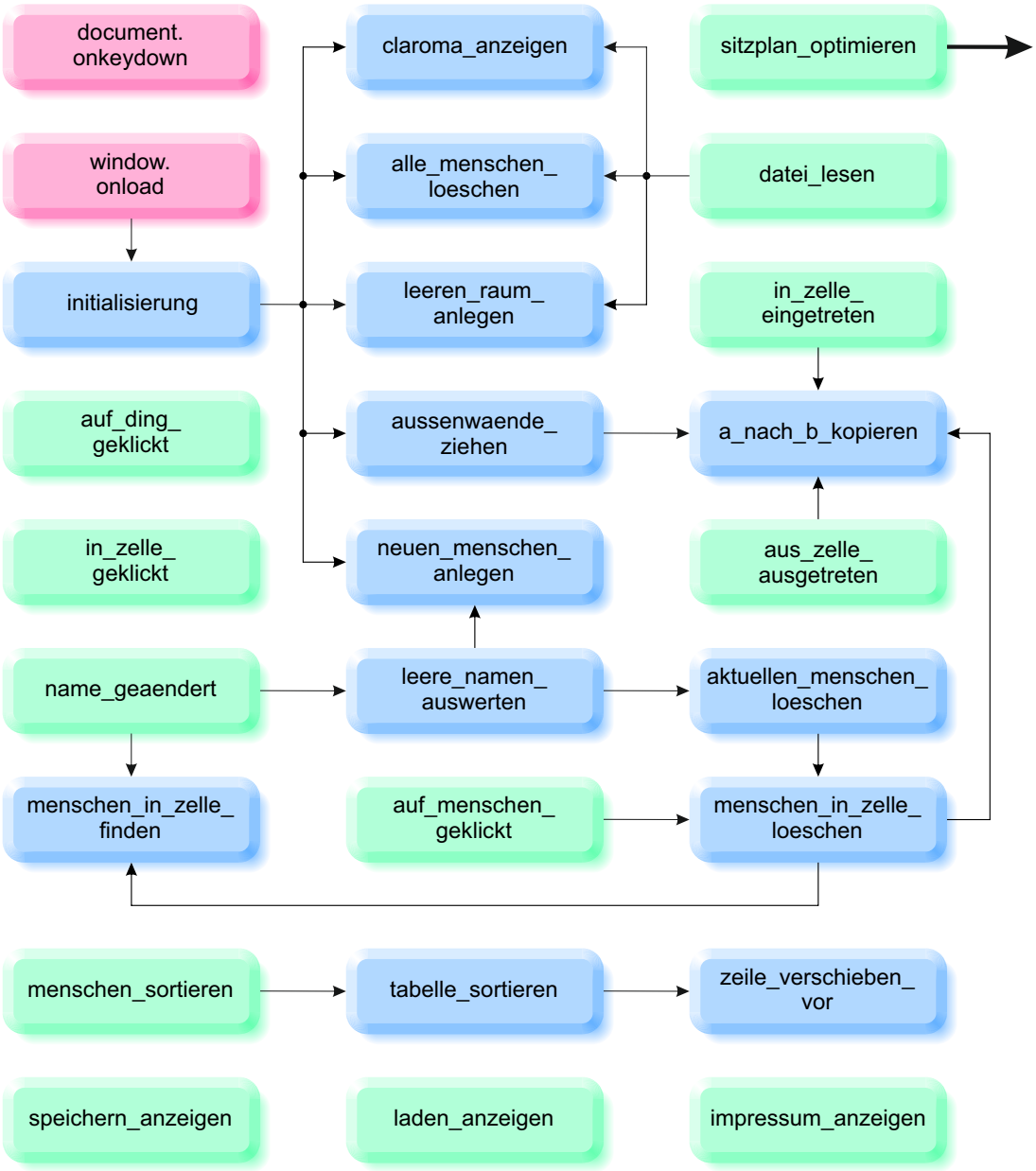


Abbildung 3.1: Blockschaltbild ohne Optimierung

Dabei sind die roten Blöcke globale Event-Handler, die immer dann aufgerufen werden, wenn ein externes Ereignis eintritt. Die grünen Blöcke werden aufgerufen, wenn die Lehrkraft die Maus bewegt oder eine Schaltfläche anklickt. Die blauen Blöcke sind Hilfsunterprogramme, die von den roten und grünen (und von den anderen blauen) aufgerufen werden. Im Folgenden geben wir einen kurzen Überblick über das Zusammenspiel der Unterprogramme.

`document.onkeydown` wird immer dann aufgerufen, wenn die Lehrkraft eine Taste auf der Tastatur drückt. Es dient dazu, zu verhindern, dass eine versehentlich gedrückte Zurück-Taste die Claroma-Seite verlässt.

`window.onload` wartet, bis die Hauptseite vollständig geladen ist und ruft dann das Unterprogramm `initialisierung` auf. Dort entfernen wir alle möglicherweise schon in den Namenslisten vorhandenen Menschen (`alle_menschen_loeschen`), legen einen neuen, leeren Raum (`leeren_raum_anlegen`) mit Außenwänden an (`aussenwaende_ziehen`), tragen jeweils eine Lehrerin, einen Lehrer, eine Schülerin und einen Schüler ohne Namen ein (`neuen_menschen_anlegen`) und machen die Hauptseite sichtbar (`claroma_anzeigen`). Beim Ziehen der Außenwände verwenden wir das Unterprogramm `a_nach_b_kopieren`, um Wandobjekte in den Raum zu kopieren.

Wenn die Lehrkraft auf einen der vorgegebenen Gegenstände (Tisch, Tür, ...) in der linken Spalte klickt (`auf_ding_geklickt`), machen wir diesen zum aktuellen Objekt. Wenn die Lehrkraft in eine Zelle im Raum klickt (`in_zelle_geklickt`), tauschen wir das dort möglicherweise vorhandene Objekt mit dem aktuellen. Beim Klicken auf einen Menschen in der rechten Spalte (`auf_menschen_geklickt`) löschen wir mit `menschen_in_zelle_loeschen` (das wiederum mit `menschen_in_zelle_finden` den Menschen im Raum sucht und dann mit `a_nach_b_kopieren` durch einen leeren Eintrag ersetzt) den Menschen im Raum. Beim Eintreten der Maus in eine neue Zelle (`in_zelle_eingetreten`) und beim Verlassen der Zelle (`aus_zelle_ausgetreten`) kopieren wir den aktuellen Zellinhalt mit `a_nach_b_kopieren` in einen Zwischenspeicher beziehungsweise wieder zurück in die Zelle.

Wenn die Lehrkraft den Namen eines Menschen verändert, wird `name_geaendert` aufgerufen. Im Unterprogramm untersuchen wir mit Hilfe von `menschen_in_zelle_finden`, ob der Mensch schon in den Raum gesetzt wurde, ändern gegebenenfalls seinen Aufdruck und werten mittels `leere_namen_auswerten` aus, ob wir in der Namensliste einen neuen Eintrag hinzufügen (`neuen_menschen_anlegen`) oder einen leeren Namenseintrag löschen müssen (`aktuellen_menschen_loeschen`). Beim Löschen müssen wir sowohl den Eintrag in der Liste als auch gegebenenfalls das Symbol in der Raumzelle entfernen (`menschen_in_zelle_loeschen`).

Beim Anklicken der Schaltfläche Sortieren wird das Unterprogramm `menschen_sortieren` aufgerufen, das seinerseits für beide Tabellen `tabelle_sortieren` aufruft. Dabei haben wir das Verschieben einer Tabellenzeile auf Grund der gewöhnungsbedürftigen Syntax nach `zeile_verschieben` ausgelagert.

Die Unterprogramme `speichern_anzeigen`, `laden_anzeigen` und `impressum_anzeigen`

werden beim Anklicken der „gleichnamigen" Schaltflächen aufgerufen und machen einfach die entsprechenden Seiten sichtbar.

Wenn die Lehrkraft das Hochladen einer Datei angefordert hat, wird `datei_lesen` aufgerufen. Es liest die in der Datei gespeicherten Informationen, löscht alle momentan vorhandenen Einträge in der Liste (`alle_menschen_loeschen`), legt einen leeren Raum (mit den in der Datei geforderten Maßen) an (`leeren_raum_anlegen`) und zeigt das Hauptfenster an (`claroma_anzeigen`).

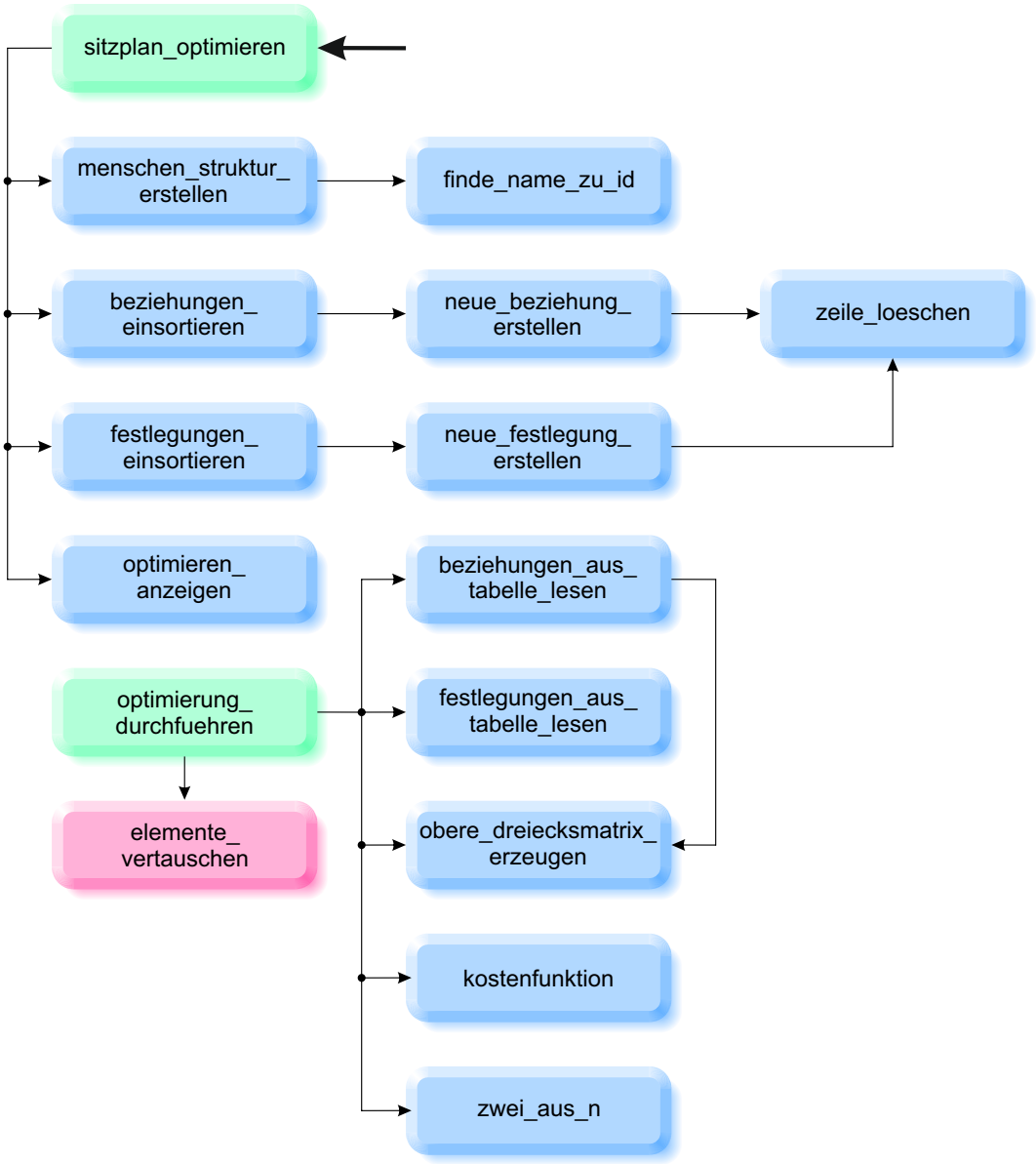


Abbildung 3.2: Blockschaltbild der Optimierung

Nach dem Drücken der Schaltfläche Optimieren wird das Unterprogramm `sitzplan_optimieren` aufgerufen, in dem ein Optimierer versucht, die Sitzplätze der Schülerinnen

und Schüler zu variieren, um die von der Lehrkraft vorgegebenen Wünsche bestmöglich zu erfüllen. Das entsprechende Blockschaltbild ist in Abbildung 3.2 dargestellt.

Das Unterprogramm `sitzplan_optimieren` ruft vier weitere Unterprogramme auf:

- `menschen_struktur_erstellen` erzeugt ein Strukturfeld aller im Plan gesetzten Lehrkräfte, Schülerinnen und Schüler, da nur diese bei der Optimierung berücksichtigt werden. Dabei verwenden wir das Unterprogramm `finde_name_zu_id`, um die in den Tabellen der Hauptseite definierten Namen der Menschen aus ihren IDs zu ermitteln.
- `beziehungen_einsortieren` sortiert die gegebenenfalls aus einer Datei gelesenen Beziehungen in die entsprechende Tabelle auf der Optimierungsseite ein. Das Erzeugen einer neuen Beziehungszeile in der Tabelle haben wir dabei in das Unterprogramm `neue_beziehung_erstellen` ausgelagert, das wiederum das Löschen einer Zeile mit Hilfe von `zeile_loeschen` als onclick-Ereignis der Schaltfläche Diese Beziehung löschen beinhaltet.
- `festlegungen_einsortieren` sortiert die gegebenenfalls aus einer Datei gelesenen Festlegungen in die entsprechende Tabelle auf der Optimierungsseite ein. Das Erzeugen einer neuen Festlegungszeile in der Tabelle haben wir dabei in das Unterprogramm `neue_festlegung_erstellen` ausgelagert, das wiederum das Löschen einer Zeile mit Hilfe von `zeile_loeschen` als onclick-Ereignis der Schaltfläche Diese Festlegung löschen beinhaltet.
- `optimieren_anzeigen` versteckt die Hauptseite und macht die Optimierungsseite sichtbar.

Durch Drücken der Schaltfläche Optimierung durchführen auf der Optimierungsseite wird das Unterprogramm `optimierung_durchfuehren` aufgerufen. Es ruft dann selbst sechs weitere Unterprogramme auf:

- `beziehungen_aus_tabelle_lesen` liest die von der Lehrkraft in der Tabelle auf der Optimierungsseite angegebenen Beziehungen und sortiert sie in eine obere Dreiecksmatrix ein. Dazu wird vorher das Unterprogramm `obere_dreiecksmatrix_erzeugen` aufgerufen.
- `festlegungen_aus_tabelle_lesen` liest die von der Lehrkraft in der Tabelle auf der Optimierungsseite angegebenen Festlegungen und sortiert sie in ein eindimensionales Feld ein. Als komplementäre Menge ermitteln wir daraus die Menschen, die während der Optimierung verschoben werden dürfen.
- `obere_dreiecksmatrix_erzeugen` gibt eine obere Dreiecksmatrix zurück, in der alle nordöstlichen Elemente eines Feldes mit einer numerischen `0` und die übrigen Elemente mit dem Literal `null`, das das Fehlen eines Elementes symbolisiert, gefüllt sind.
- `kostenfunktion` berechnet aus den Beziehungen und dem aktuellen Sitzplan dessen Kostenwert. Je besser ein Sitzplan die Wünsche der Lehrkraft erfüllt, desto kleiner ist sein Kostenwert.

- `zwei_aus_n` gibt ein zweidimensionales Feld (mit zwei Spalten) zurück, in dem alle Kombinationen (ohne Wiederholung) aufgelistet sind, die sich ergeben, wenn man jeweils zwei Elemente aus einer Urne zieht.
- `elemente_vertauschen` erweitert den Prototyp Array um die Methode, zwei Elemente des Feldes zu vertauschen.

4 claroma.html

In den folgenden Kapitel werden wir beschreiben, wie wir die Werkzeuge HTML5 [1], CSS3 [2] und JavaScript [3] nutzen, um der Seite `claroma.html` mit den in `claroma.js` zusammengefassten Programmen Leben einzuhauchen.

Eine Webseite hat in HTML5 den folgenden Aufbau:

```
<!DOCTYPE html>

<html>

<head> ... </head>

<body> ... </body>

</html>
```

Die Dokumententypdeklaration

```
<!DOCTYPE html>
```

sagt dem Browser, dass es sich um eine HTML5-Seite und nicht eine ältere Version handelt

Die eigentliche Seite beginnt dann mit dem öffnenden `<html>`-Tag und endet mit dem schließenden `</html>`-Tag. Im `<head>` werden die Style Sheet-Datei `claroma.css`, die JavaScript-Datei `claroma.js`, in der sich das eigentliche Programm befindet und die Bibliothek `FileSaver.js` geladen. Im `<body>` befinden sich die Beschreibung der Seitenelemente.

4.1 <head>

```
<head>
```

Als Zeichensatz verwenden wir – wie im Internet üblich – Unicode in 8-Bit-Kodierung, mit der wir beispielsweise Umlaute direkt verwenden können [4]:

```
  <meta
    http-equiv="Content-Type"
    content="text/html; charset=utf-8" />
```

In der nächsten Zeile verweisen wir auf die CSS-Datei claroma.css, die die Formatierung der Elemente definiert:

```
<link
   href="claroma.css"
   rel="stylesheet" />
```

Den JavaScript-Code des Programmes haben wir in die Datei claroma.js ausgelagert:

```
  <script src="claroma.js"></script>
```

Um Dateien problemlos in allen Browsern abspeichern zu können, verwenden wir eine externe Bibliothek [5]:

```
  <script src="FileSaver.js-master/FileSaver.js"></script>
```

Abschließend definieren wir noch den Seitentitel, der im Reiter der Registerkarte im Browser angezeigt wird:

```
  <title>Claroma</title>
</head>
```

4.2 <body>

Im Seitenkörper

```
<body>
```

definieren wir alle Seiten und Objekte, die wir im Rahmen von Claroma darstellen wollen. Dabei verwenden wir ein jQuery Mobile [6] nachempfundenes[1] Verfahren, in dem wir alle Seiten des Programmes gemeinsam in einer einzigen HTML-Seite beschreiben und jeweils nur eine Seite sichtbar machen:

```
  <div id="seite_claroma"> ... </div>
  <div id="seite_speichern"> ... </div>
  <div id="seite_laden"> ... </div>
  <div id="seite_impressum"> ... </div>
  <div id="seite_optimieren"> ... </div>
</body>
```

4.2.1 seite_claroma

Die in einem gekapselte Hauptseite

```
<div id="seite_claroma">
```

[1] Wir verzichten ganz bewusst auf die Einbindung von jQuery Mobile, um das Programm schlank, netzunabhängig und problemlos zu halten. Auf einem modernen Android-Tablet ist es trotzdem gut bedienbar.

ist bei Programmstart sichtbar und beinhaltet alle in Abbildung 2.1 sichtbaren Objekte. Zur Formatierung verwenden wir eine klassische HTML-Tabelle

```
<table>
```

deren einzige Zeile

```
  <tr>
```

drei Spalten enthält. In der linken Spalte

```
    <td class="oben">
```

geben wir als erstes den Namen des Programms als Überschrift[2] aus

```
      <h1>Claroma</h1>
      <br />
```

und fügen ein Eingabefeld für den Klassennamen ein:

```
      <input id="klassen_name"
             type="text"
             placeholder=" Klasse"
             class="klassen_name" />
```

Gegenstände

Darunter stellen wir die im Klassenraum zu verteilenden Gegenstände in Form einer zweispaltigen Tabelle dar:

```
      <table id="table_dinge">
```

Die erste Spalte ist dabei jeweils das durch seine CSS-Klasse farbig angelegte Symbol des Gegenstands mit seiner Abkürzung:

```
        <tr>
          <td id="wand"
              class="feld wand"
              onclick="auf_ding_geklickt(this)">
            W
          </td>
```

Durch die `onclick`-Eigenschaft der Zelle definieren wir die JavaScript-Funktion, die aufgerufen wird, wenn die Lehrkraft auf das Symbol klickt. In der zweiten Spalte geben wir den Langnamen des Gegenstands aus:

```
          <td>Wand</td>
        </tr>
```

Das gleiche Verfahren wiederholen wir für die anderen Gegenstände einschließlich des „Löschstempels”:

[2] Die CSS-Klassendefinitionen aller Objekte finden wir natürlich in `claroma.css`.

```
          <tr>
            <td id="fenster"
                class="feld fenster"
                onclick="auf_ding_geklickt(this)">
              F
            </td>
            <td>Fenster</td>
          </tr>
          <tr>
            <td id="tuer"
                class="feld tuer"
                onclick="auf_ding_geklickt(this)">
              Tü
            </td>
            <td>Tür</td>
          </tr>
          <tr>
            <td id="tafel"
                class="feld tafel"
                onclick="auf_ding_geklickt(this)">
              Ta
            </td>
            <td>Tafel</td>
          </tr>
          <tr>
            <td id="tisch"
                class="feld tisch"
                onclick="auf_ding_geklickt(this)">
              T
            </td>
            <td>Tisch</td>
          </tr>
          <tr>
            <td id="schrank"
                class="feld schrank"
                onclick="auf_ding_geklickt(this)">
              S
            </td>
            <td>Schrank</td>
          </tr>
          <tr>
            <td id="ablage"
                class="feld ablage"
                onclick="auf_ding_geklickt(this)">
              A
            </td>
```

```
            <td>Ablage</td>
          </tr>
          <tr>
            <td id="leer"
                class="feld"
                onclick="auf_ding_geklickt(this)"></td>
            <td>Leer</td>
          </tr>
        </table>
```

Schaltflächen

Unterhalb der zu setzenden Gegenstände ordnen wir die Schaltflächen an (Abbildung 2.1):

```
        <button onclick="speichern_anzeigen()">
          Speichern
        </button>
        <br />
        <button onclick="laden_anzeigen()">
          Laden
        </button>
        <br />
        <button onclick="window.open('claroma.pdf');">
          Hilfe
        </button>
        <br />
        <button onclick="impressum_anzeigen()">
          Impressum
        </button>
        <br />
        <button onclick="menschen_sortieren()">
          Sortieren
        </button>
      </td>
```

Die Hilfe-Schaltfläche ruft beim Auswählen mit dem Befehl `window.open` direkt dieses Dokument auf; die anderen Schaltflächen verzweigen zu den entsprechenden JavaScript-Funktionen `speichern_anzeigen`, `laden_anzeigen`, `impressum_anzeigen` und `menschen_sortieren`.

Raum

Damit haben wir die Beschreibung der linken Spalte der äußeren Tabelle abgeschlossen und wir können uns der mittleren Spalte widmen, in der gemäß Abbildung 2.1 der eigentliche Klassenraum dargestellt wird. Da wir unterschiedlich große Räume zulassen wollen, können wir den Raum hier nicht einmalig statisch definieren, sondern müssen ihn später in `claroma.js` je nach Wunsch der Lehrkraft dynamisch erzeugen. Hier definieren wir daher nur eine leere Tabelle:

```
      <td class="oben">
        <table id="table_raum"></table>
      </td>
```

Menschen

Wie in Abbildung 2.1 zu sehen, listen wir die Menschen (Lehrkräfte, Schülerinnen und Schüler) in der rechten Spalte der äußeren Tabelle auf. Um dem Nutzerwunsch nach geschlechtsspezifischer Aufteilung Genüge zu tun, wischen wir schweren Herzens sämtliche Gender-Bedenken beiseite und erzeugen eine weibliche (Lehrerin, Schülerinnen) und eine männliche (Lehrer, Schüler) Tabelle nebeneinander:

```
      <td class="oben rechts">
        <table id="table_rosa"></table>
      </td>
      <td class="oben rechts">
        <table id="table_blau"></table>
      </td>
    </tr>
  </table>
</div>
```

Auch diese Tabellen werden wir in `claroma.js` dynamisch verwalten, da sich die Anzahl der Menschen ja während des Programmlaufs immer wieder ändert.

4.2.2 `seite_speichern`

Wie schon erwähnt, kapseln wir einzelne Seiten des Programmes in `<div>`-Abschnitte, die wir einzeln sichtbar bzw. unsichtbar machen können. Wenn die Lehrkraft die Speichern-Schaltfläche ausgewählt hat, verstecken wir die Hauptseite `seite_claroma` und zeigen `seite_speichern`:

```
<div id="seite_speichern">
```

Dort schreiben wir als erstes eine Überschrift

```
  <h1>Speichern</h1>
```

und dann einen Informationstext mit einem eingebetteten Tag, das wir in `speichern_anzeigen` mit Leben füllen:

```
  <p>
    Der aktuelle Zustand von Claroma wird gerade
    in Ihrem Download-Verzeichnis unter dem Namen:
  </p>
  <p id="speicher_datei_name"></p>
  <p>
    gespeichert.
  </p>
```

Anschließend geben wir der Lehrkraft die Möglichkeit, wieder auf die Hauptseite zurückzukehren:

```
  <p>
    <button onclick="claroma_anzeigen()">
      Zurück
    </button>
  </p>
</div>
```

Da wir ja nicht wirklich zu einer anderen HTML-Seite gewechselt sind, sondern nur das der Hauptseite unsichtbar gemacht haben, müssen wir nach Klicken der Zurück-Schaltfläche in claroma_anzeigen auch nur die aktuelle Seite (seite_speichern) verstecken und die Hauptseite (seite_claroma) wieder zum Vorschein bringen.

4.2.3 seite_laden

Auf der Seite zum Laden bzw. Anlegen eines Klassenplanes

```
<div id="seite_laden">
```

geben wir der Lehrkraft nach der Überschrift

```
  <h1>Laden</h1>
```

drei Möglichkeiten. Erstens kann sie eine vorhandene Datei von ihrer Festplatte laden, die sie oder eine andere Lehrkraft dort vorher abgespeichert hat:

```
  <p>
  Sie können entweder eine vorher abgespeicherte Datei laden:
  </p>
  <p>
    <input type="file"
           onchange="datei_lesen(event)" />
  </p>
```

Um der Seite den lesenden Zugriff auf Dateien von der lokalen Festplatte zu ermöglichen, verwenden wir das <input>-Tag vom Typ file. Wenn die Lehrkraft eine lokale Datei ausgewählt hat, wird automatisch die Funktion datei_lesen aufgerufen, um die Datei zu verarbeiten.

Alternativ zum Laden einer Datei geben wir der Lehrkraft die Möglichkeit, einen ganz neuen Raum beliebiger Dimensionen anzulegen:

```
  <p>
    Oder Sie erzeugen einen neuen leeren Raum
    mit eine Tiefe von
    <input id="input_n_raum_zeilen"
           type="number"
```

```
            min="1"
            max="50"
            step="1"
            value="20" />
    und einer Breite von
    <input id="input_n_raum_spalten"
            type="number"
            min="1"
            max="50"
            step="1"
            value="20" />
    Kästchen:
  </p>
```

„beliebig" bedeutet dabei, dass die Abmessungen zwischen 1 (Was für einen Sinn ergibt das eigentlich?) und 50 Kästchen[3] variieren dürfen.

Die zugehörige Schaltfläche ruft die Initialisierungsroutine `initialisierung` auf, die die veränderten Raummaße aus den `<input>`-Feldern liest und einen neuen, leeren Raum anlegt:

```
  <p>
    <button onclick="initialisierung()">
      Neu
    </button>
  </p>
```

Aus diesem Grund ist es sinnvoll, die Lehrkraft zu warnen, dass ihre bisherige Arbeit beim Laden oder Neuanlegen eines Raumes verloren geht:

```
  In beiden Fällen wird der aktuelle Raum gelöscht!
```

Als dritte Alternative ermöglichen wir der Lehrkraft, ohne Veränderung zum bisherigen Raum zurückzukehren:

```
  <p>
    Oder Sie gehen zurück,
    ohne einen Raum zu laden oder neu zu erzeugen:
  </p>
  <p>
    <button onclick="claroma_anzeigen()">
      Zurück
    </button>
```

[3] Schultische haben üblicherweise eine Breite von 65 cm (als Doppeltisch mit 130 cm) bis 75 cm und eine Tiefe von 50 cm bis 65 cm. Gehen wir also näherungsweise von einem quadratischen Tisch der Breite 70 cm aus, bedeutet der Standardwert von 20 Kästchen (18×18 Kästchen Nutzfläche innerhalb der Wände), dass wir über eine Innenraumfläche von 12.6×12.6 m reden, was für die Modellierung der meisten Klassenräume ausreichen sollte. Mit einem Maximalwert von 50 Kästchen (33.6×33.6 m) können wir dann auch mittelgroße Turnhallen und Aulen abbilden.

```
  </p>
</div>
```

4.2.4 seite_impressum

Auf der Impressumsseite

```
<div id="seite_impressum">
```

die laut § 5 des Telemediengesetzes in jeder Website verfügbar sein muss, listen wir nach der Überschrift

```
  <h2>Impressum</h2>
```

unsere Kontaktdaten auf:

```
  <p>
    Angaben gemäß § 5 TMG:
  </p>
  <p>
    Jörg J. Buchholz<br />
    Florian Buchholz
  </p>
  <p>
    Hochschule Bremen<br />
    Neustadtswall 30<br />
    28199 Bremen
  </p>
  Telefon: 0421 5905 3544<br />
  Telefax: 0421 5905 3505<br />
  E-Mail: <a href="mailto:buchholz@hs-bremen.de">
    buchholz@hs-bremen.de</a>
```

Zusätzlich nutzen wir die Impressumsseite, um ein paar Worte über den Datenschutz

```
  <h2>Datenschutz</h2>
```

von Claroma zu verlieren.

```
  <p>
    Die Nutzung unserer Webseite
    ist ohne Angabe personenbezogener Daten möglich.
  </p>
  <p>
    Wir weisen darauf hin,
    dass die Datenübertragung im Internet
    (z.B. mit dem Browser) Sicherheitslücken aufweisen kann.
    Ein lückenloser Schutz der Daten
```

```
    vor dem Zugriff durch Dritte ist nicht möglich.
  </p>
  <p>
    Der Nutzung von im Rahmen der Impressumspflicht
    veröffentlichten Kontaktdaten
    durch Dritte zur Übersendung von nicht
    ausdrücklich angeforderter Werbung und
    Informationsmaterialien wird
    hiermit ausdrücklich widersprochen.
    Die Betreiber der Seiten behalten sich
    ausdrücklich rechtliche Schritte
    im Falle der unverlangten Zusendung von Werbeinformationen,
    etwa durch Spam-Mails, vor.
  </p>
  <p>
    Quellenangaben:
    <a
href="http://www.e-recht24.de/muster-datenschutzerklaerung.html"
        target="_blank">eRecht24</a>
  </p>
```

Die obligatorische Zurück-Schaltfläche versteckt – wie üblich – diese Seite und bringt die Hauptseite zum Vorschein:

```
  <p>
    <button onclick="claroma_anzeigen()">
      Zurück
    </button>
  </p>
</div>
```

4.2.5 seite_optimieren

Auf der Optimierungsseite

```
<div id="seite_optimieren">
```

beginnen wir sinnvollerweise mit einer passenden Überschrift

```
  <h2>Optimieren</h2>
```

und erstellen dann zwei Tabellen mit jeweils einer Unterüberschrift und jeweils einer Schaltfläche zum Anhängen einer neuen Tabellenzeile. Unter der Unterüberschrift

```
  <h3>Wer soll (nicht) neben wem sitzen?</h3>
```

trägt die Lehrkraft ihre Wünsche bezüglich der Nähe bzw. Entfernung einzelner Schülerinnen und Schüler in die erste Tabelle ein:

```
  <table id="table_beziehungen"></table>
```

Unter der Tabelle gibt es besagte Schaltfläche zum Einfügen einer neuen Beziehung, die auf das Unterprogramm neue_beziehung_erstellen verweist:

```
  <button class="button_lang"
          onclick="neue_beziehung_erstellen()">
    Neue Beziehung erstellen
  </button>
```

Nach der Unterüberschrift

```
  <h3>Wer soll nicht verschoben werden?</h3>
```

kann die Lehrkraft Schülerinnen und Schüler in die Tabelle

```
  <table id="table_festlegungen"></table>
```

eintragen, deren Sitzplatz während der Optimierung nicht verändert werden soll. Darunter gibt es wieder eine Schaltfläche zum Einfügen einer neuen Festlegung:

```
  <button class="button_lang"
          onclick="neue_festlegung_erstellen()">
    Neue Festlegung erstellen
  </button>
```

Abschließend gibt es die Unterüberschrift

```
  <h3>Jetzt optimieren?</h3>
```

gefolgt von den beiden Schaltflächenalternativen, die Optimierung durch Sprung zu optimierung_durchfuehren tatsächlich durchzuführen

```
  <button class="button_lang"
          onclick="optimierung_durchfuehren()">
    Optimierung durchführen
  </button>
```

oder zur Hauptseite zurückzukehren:

```
  oder
  <button class="button_lang"
          onclick="claroma_anzeigen()">
    Optimierung abbrechen
  </button>
</div>
</body>
</html>
```

5 claroma.js

In diesem Kapitel wollen wir die einzelnen Funktionen von `claroma.js` genauer analysieren.

5.1 window.onload

Das `onload`-Ereignis eines Objektes wird ausgelöst, wenn das Objekt vollständig geladen ist. Da wir schon während der Initialisierung Tabelleninhalte verändern möchten, müssen wir damit warten, bis die Hauptseite komplett geladen ist. Wir definieren daher eine anonyme Funktion, die genau dann ausgeführt wird, wenn wir auf alle Inhalte der Hauptseite zugreifen können

```
window.onload = function () {
```

und rufen erst dann `initialisierung` auf:

```
    initialisierung()
}
```

5.2 initialisierung

Während der Initialisierung

```
function initialisierung() {
```

erzeugen wir einen leeren Raum mit Außenwänden und jeweils eine Lehrkraft, eine Schülerin und einen Schüler ohne Namen (Abbildung 5.1)

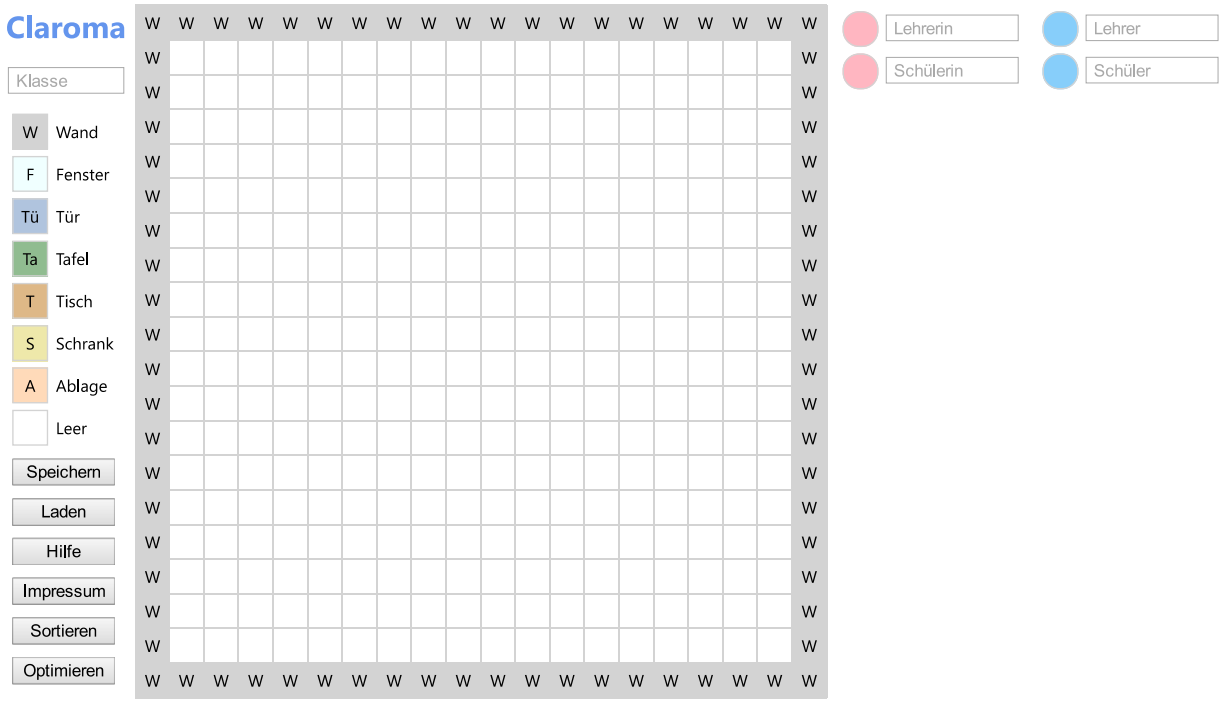


Abbildung 5.1: Nach der Initialisierung

Zur Raumerzeugung definieren wir als erstes die Anzahl der Zeilen (Tiefe) und der Spalten (Breite) des Klassenraumes:

```
  n_raum_zeilen = input_n_raum_zeilen.value
  n_raum_spalten = input_n_raum_spalten.value
```

Dabei greifen wir auf die in `seite_laden` von der Lehrkraft eingegebenen Werte zurück, um auch beim erneuten Erstellen eines Raumes einfach `initialisierung` aufrufen zu können. Der in `seite_laden` vorbesetzte Standardwert beträgt 20×20 Kästchen.

Als nächstes löschen wir in `alle_menschen_loeschen` alle möglicherweise schon eingegebenen Lehrkräfte, Schülerinnen und Schüler

```
  alle_menschen_loeschen()
```

legen in `leeren_raum_anlegen` einen leeren Raum der gewünschten Tiefe und Breite an

```
  leeren_raum_anlegen()
```

und setzen Objekte vom Typ Wand in die äußeren Zellen des Raumes (oben, unten, links und rechts):

```
  aussenwaende_ziehen()
```

In den nächsten Schritten legen wir leere Eingabefelder für jeweils eine Lehrerin, einen Lehrer, eine Schülerin und einen Schüler ein. Wir beginnen mit der Lehrerin, indem wir in `neuen_menschen_anlegen` einen neuen weiblichen Menschen anlegen:

```
  neuen_menschen_anlegen('rosa')
```

und tragen in seinem Eingabefeld den passenden Platzhaltertext ein:

```
  table_rosa.rows[0].cells[1].children[0].setAttribute
    ('placeholder', ' Lehrerin')
```

Auf die gleiche Weise initialisieren wir einen potenziellen Lehrer:

```
  neuen_menschen_anlegen('blau')
  table_blau.rows[0].cells[1].children[0].setAttribute
    ('placeholder', ' Lehrer')
```

Für die Schülerinnen und Schüler erzeugen wir nur jeweils ein neues Eingabefeld, das in `neuen_menschen_anlegen` automatisch mit dem richtigen Platzhalter besetzt wird:

```
  neuen_menschen_anlegen('rosa')
  neuen_menschen_anlegen('blau')
```

In vielen Fällen wird die Lehrkraft zu Beginn gleich Tische im Raum verteilen wollen. Aus diesem Grund hängen wir ihr als erstes einen Tisch an den Mauszeiger:

```
  aktuelles_objekt = tisch
```

Wenn der Mauszeiger in `in_zelle_eingetreten` in eine neue Zelle eintritt, müssen wir das in der Zelle schon vorhandene Objekt zwischenspeichern. Dazu erzeugen wir hier schon einmal einen Zwischenspeicher als globale Variable:

```
  objekt_puffer = document.createElement('TD')
```

Als letztes initialisieren wir noch die für das Abspeichern der Optimierungsdaten benötigten Felder:

```
  beziehungen = []
  festlegungen = []
```

Abschließend stellen wir mit `claroma_anzeigen` die Hauptseite `seite_claroma` (wieder) dar. Dies ist beispielsweise dann nötig, wenn wir vorher auf `seite_laden` einen neuen Plan definiert hatten:

```
  claroma_anzeigen()
}
```

5.3 alle_menschen_loeschen

Während der Initialisierung müssen wir alle möglicherweise schon vorhandenen Lehrkräfte, Schülerinnen und Schüler aus beiden Namenslisten entfernen.

```
function alle_menschen_loeschen() {
```

Dazu ermitteln wir die Anzahl der weiblichen Listeneinträge

```
  var n_rosa = table_rosa.rows.length
```

und starten eine Schleife über alle Einträge

```
  for (var i_rosa = 0; i_rosa < n_rosa; i_rosa++) {
```

in der wir jeden einzelnen Eintrag löschen:

```
    table_rosa.deleteRow(0)
  }
```

Das Gleiche wiederholen wir für die Einträge der männlichen Liste:

```
  var n_blau = table_blau.rows.length
  for (var i_blau = 0; i_blau < n_blau; i_blau++) {
    table_blau.deleteRow(0)
  }
}
```

5.4 leeren_raum_anlegen

Zum Anlegen eines neuen, leeren Raumes

```
function leeren_raum_anlegen() {
```

entkernen wir als erstes einen möglicherweise schon vorhandenen Raum. Dazu beschaffen wir uns die Anzahl der Zeilen (Tiefe) des bisherigen Raumes

```
  var n_zeilen_alt = table_raum.rows.length
```

und löschen in einer Schleife jede einzelne Zeile:

```
  for (var i_zeile = 0; i_zeile < n_zeilen_alt; i_zeile++) {
    table_raum.deleteRow(0)
  }
```

Alsdann legen wir jede einzelne Zelle des neuen Raumes mit ihren gewünschten Eigenschaften neu an. Dazu starten wir eine Schleife über alle Zeilen des neuen Raumes, deren Anzahl wir in initialisierung ermittelt und in der globalen Variablen n_raum_zeilen abgespeichert hatten:

```
  for (var i_zeile = 0; i_zeile < n_raum_zeilen; i_zeile++) {
```

In der Schleife fügen wir dem Raum eine neue Zeile hinzu

```
    var zeile = table_raum.insertRow(i_zeile)
```

und beginnen eine Schleife über alle gewünschten (n_raum_spalten) Zellen der aktuellen Zeile:

```
    for (var i_spalte = 0; i_spalte < n_raum_spalten; i_spalte++)
    {
```

Wir fügen der aktuellen Zeile jeweils eine neue Zelle hinzu

```
      var zelle = zeile.insertCell(i_spalte)
```

geben ihr die Eigenschaften (claroma.css) der CSS-Klasse feld

```
      zelle.setAttribute('class', 'feld')
```

und setzen die Eigenschaft, in der wir später die ID des in der Zelle abgelegten Objektes halten werden, auf ihren Anfangswert:

```
      zelle.setAttribute('data-id', 'leer')
```

Außerdem definieren wir die Unterprogramme, die aufgerufen werden, wenn der Mauszeiger in eine Zelle eintritt (in_zelle_eingetreten)

```
      zelle.setAttribute('onmouseover',
        'in_zelle_eingetreten(this)')
```

die Zelle wieder verlässt (aus_zelle_ausgetreten)

```
      zelle.setAttribute('onmouseout',
        'aus_zelle_ausgetreten(this)')
```

oder wenn die Lehrkraft in die Zelle klickt (in_zelle_geklickt):

```
      zelle.setAttribute('onclick',
        'in_zelle_geklickt(this)')
    }
  }
}
```

5.5 neuen_menschen_anlegen

Das Unterprogramm

```
function neuen_menschen_anlegen(
  farbe, id, klasse, aufdruck, name) {
```

nutzen wir in initialisierung, um leere Listeneinträge für die Symbole und die Namen der Lehrkräfte, Schülerinnen und Schüler anzulegen und in datei_lesen, um die aus der Datei gelesenen Menschen in die Liste einzutragen. Dazu ermitteln wir zuerst die Tabelle, die zu der als Parameter übergebenen Farbe passt

```
  tabelle = document.getElementById('table_' + farbe)
```

bestimmen ihre Länge

```
  var n_menschen = tabelle.rows.length
```

hängen an das Ende eine neue Zeile an:

```
  var neue_zeile = tabelle.insertRow(n_menschen)
```

Symbol

In der neuen Zeile erzeugen wir als erstes eine Zelle für das Symbol des neuen Menschen:

```
  var neuer_mensch = neue_zeile.insertCell(0)
```

Als nächstes bestimmen wir die Eigenschaften des Symbols. Dabei untersuchen wir jeweils, ob eine gewünschte Eigenschaft über die Parameterliste übergeben wurde (was beim Lesen aus einer Datei der Fall ist) oder ob wir einen ganz neuen Menschen anlegen müssen. Wenn also keine ID über die Parameterliste übergeben wurde

```
  if (id === undefined) {
```

erzeugen wir eine sehr große[1] ganzzahlige Zufallszahl

```
  var zuf = Math.floor(Math.random() * 9007199254740991)
```

und basteln uns daraus eine (mit sehr großer Wahrscheinlichkeit) eindeutige ID:

```
  neuer_mensch.id = 'm' + zuf
```

Wenn hingegen eine ID über die Parameterliste übergeben wurde

```
  } else {
```

verwenden wir sie:

```
    neuer_mensch.id = id
  }
```

Das gleiche Konzept benutzen wir bei der Definition der CSS-Klasse des neuen Symbols. Wenn es keine Vorgabe über die Parameterliste gibt, definieren wir die Klasse entsprechend des vorgegebenen Geschlechts (`farbe`):

```
  if (klasse === undefined) {
    neuer_mensch.setAttribute('class', 'feld mensch ' + farbe)
```

Anderenfalls übernehmen wir direkt die vorgegebene Klasse aus der Parameterliste:

```
  } else {
    neuer_mensch.setAttribute('class', klasse)
  }
```

[1] Die Zahl 9007199254740991 ist dabei die größte im IEEE-Format darstellbare positive Ganzzahl ($2^{53} - 1 = 9007199254740991$). Die entsprechende, in JavaScript dafür vorgesehene Konstante `Number.MAX_SAFE_INTEGER` wird leider (mal wieder) vom Internet Explorer nicht erkannt.

Ein ganz neuer Mensch hat noch keinen Namen. Demzufolge hat sein Symbol auch noch keine Namensabkürzung als Aufdruck. Wenn in der Parameterliste aber ein Symbolaufdruck vorgegeben ist

```
  if (aufdruck !== undefined) {
```

verwenden wir diesen:

```
    neuer_mensch.innerHTML = aufdruck
  }
```

Abschließend definieren wir das Unterprogramm `auf_menschen_geklickt`, das aufgerufen wird, wenn die Lehrkraft auf das Symbol klickt:

```
  neuer_mensch.setAttribute('onclick',
    'auf_menschen_geklickt(this)')
```

Name

Neben seinem Symbol hat ein Mensch natürlich auch einen Namen. Um diesen eintragen zu können, erzeugen wir ein Eingabefeld

```
  var neuer_name = document.createElement('input')
```

der Sorte Text:

```
  neuer_name.setAttribute('type', 'text')
```

Als nächstes wählen wir den Platzhalter[2] des Textfeldes entsprechend der vorgegebenen Tabellenfarbe

```
  if (farbe === 'rosa') {
    neuer_name.setAttribute('placeholder', ' Schülerin')
  } else {
    neuer_name.setAttribute('placeholder', ' Schüler')
  }
```

und definieren seine CSS-Klasse:

```
  neuer_name.setAttribute('class', 'mensch_name')
```

Wir möchten gerne, dass der Aufdruck des Symbols mit den ersten Namensbuchstaben gefüllt wird, schon während die Lehrkraft einen neuen Namen eintippt. Dazu rufen wir bei jeder Veränderung (Ereignis `oninput`) des Textfeldinhaltes das Unterprogramm `name_geaendert` auf und übergeben ihm sowohl das aktuelle Textfeld selbst (`this`) als auch die ID des Symbols des neuen Menschen:

```
  neuer_name.setAttribute('oninput',
    'name_geaendert(this, ' + neuer_mensch.id + ')')
```

[2]Bei Lehrkräften überschreiben wir in `initialisierung` die Platzhalter „Schülerin" und „Schüler" mit den entsprechenden Platzhaltern „Lehrerin" und „Lehrer".

Wenn – beim Einlesen eines Plans aus einer Datei – in der Parameterliste ein Name übertragen wird, tragen wir diesen Namen in das Textfeld ein:

```
  if (name !== undefined) {
    neuer_name.value = name
  }
```

Jetzt müssen wir nur noch rechts neben dem Symbol eine weitere Tabellenzelle für das Textfeld anlegen

```
  var neue_zelle = neue_zeile.insertCell(1)
```

und das Textfeld in die neue Zelle einfügen:

```
 neue_zelle.appendChild(neuer_name)
}
```

5.6 a_nach_b_kopieren

Das Unterprogramm

```
function a_nach_b_kopieren(a, b) {
```

dient dazu, die Symbole von Gegenständen und Menschen zu kopieren, um sie auf diese Weise beispielsweise in die Zellen des Raumes abzulegen. Dabei untersuchen wir als erstes, ob die Quelle (das Original) eine `data-id`-Eigenschaft besitzt, was beispielsweise bei allen Raumzellen der Fall ist. Diese besitzen nämlich statt einer eigenen[3] ID nur eine `data-id`. Wenn die Quelle aber keine `data-id` besitzt

```
  if (a.getAttribute('data-id') === null) {
```

handelt es sich um ein Symbol eines Gegenstandes oder eines Menschen selbst, das eine eigene ID besitzt. Diese ID kopieren wir dann in die `data-id` des Ziels (der Kopie):

```
    b.setAttribute('data-id', a.id)
```

Wenn es sich aber um eine Raumzelle (oder den Zwischenspeicher) mit einer eigenen `data-id` handelt

```
  } else {
```

dann kopieren wir diese `data-id`:

```
    b.setAttribute('data-id', a.getAttribute('data-id'))
  }
```

[3] Da zwei Objekte nicht die gleiche ID haben dürfen, können wir einer Raumzelle, die beispielsweise einen Tisch repräsentieren soll, nicht einfach auch die ID des Tisches geben. Statt dessen speichern wir die ID des Tisches in der `data-id`-Eigenschaft der Zelle ab.

Des Weiteren kopieren wir die CSS-Klasse

```
  b.setAttribute('class', a.getAttribute('class'))
```

und den Symbolaufdruck von der Quelle ins Ziel:

```
  b.innerHTML = a.innerHTML
}
```

5.7 aussenwaende_ziehen

Im Unterprogramm

```
function aussenwaende_ziehen() {
```

das wir während der initialisierung aufrufen, nehmen wir der Lehrkraft ein bisschen lästige Arbeit ab und kopieren Wandsymbole in alle Randzellen des Raumes, so dass er wie in Abbildung 5.1 dargestellt aussieht.

Für die obere und die untere Wand lassen wir eine Schleife über alle Spalten des Raumes laufen:

```
  for (i_spalte = 0; i_spalte < n_raum_spalten; i_spalte++) {
```

Für die obere Wand kopieren wir dann ein Wandsymbol in jede Zelle der ersten Zeile:

```
    a_nach_b_kopieren(wand,
      table_raum.rows[0].cells[i_spalte])
```

Für die untere Wand belegen wir entsprechend jede Zelle der letzten Zeile mit einem Wandobjekt:

```
    a_nach_b_kopieren(wand,
      table_raum.rows[n_raum_zeilen - 1].cells[i_spalte])
  }
```

Ganz analog erzeugen wir die linke und die rechte Wand, indem wir in einer Schleife über alle Zeilen die ersten und letzten Zellen ebenfalls als Wand deklarieren:

```
  for (i_zeile = 1; i_zeile < n_raum_zeilen - 1; i_zeile++) {
    a_nach_b_kopieren(wand,
      table_raum.rows[i_zeile].cells[0])
    a_nach_b_kopieren(wand,
      table_raum.rows[i_zeile].cells[n_raum_spalten - 1])
  }
}
```

5.8 name_geaendert

Wann immer die Lehrkraft den Namen eines Menschen verändert, wird das in neuen_menschen_anlegen definierte oninput-Ereignis ausgelöst und das Unterprogramm

```
function name_geaendert(mensch_name, mensch) {
```

aufgerufen, das automatisch die Anfangsbuchstaben des Namens in das Symbol des Menschen einträgt. Dabei untersuchen wir als erstes, ob der Name aus Vorname und Nachname besteht und daher ein Leerzeichen beinhaltet:

```
  var leerzeichen_index = mensch_name.value.indexOf(' ')
```

Wenn dies der Fall ist

```
  if (leerzeichen_index > 0) {
```

basteln wir die Namensabkürzung aus dem ersten Buchstaben des Vornamens und dem ersten Buchstaben des Nachnamens (gleich hinter dem Leerzeichen) zusammen:

```
    mensch.innerHTML =
      mensch_name.value.substring(0, 1) +
      mensch_name.value.substring(
      leerzeichen_index + 1, leerzeichen_index + 2)
```

Wenn es kein Leerzeichen gibt, wenn der Name also nur aus einem Vornamen oder nur aus einem Nachnamen besteht, nehmen wir einfach die ersten beiden Buchstaben des Namens als Abkürzung:

```
  } else {
    mensch.innerHTML = mensch_name.value.substring(0, 2)
  }
```

Wenn die Lehrkraft das Symbol des Menschen vorher schon im Raum angeordnet hat, müssen auch in diesem die Anfangsbuchstaben anpassen, wenn sich der Name ändert. Dazu suchen wir mit dem Unterprogramm menschen_in_zelle_finden die Zelle, in der sich das abgelegte Symbol befindet:

```
  var mensch_in_zelle = menschen_in_zelle_finden(mensch.id)
```

Wenn es eine solche Zelle gibt

```
  if (mensch_in_zelle !== null) {
```

aktualisieren wir auch dort den Aufdruck des Symbols:

```
    mensch_in_zelle.innerHTML = mensch.innerHTML
  }
```

Zusätzlich rufen wir noch bei jeder Namensänderung für beide Namenslisten das Unterprogramm leere_namen_auswerten auf, das dafür sorgt, dass immer genau ein leeres Namensfeld vorhanden ist:

```
  leere_namen_auswerten('rosa', mensch)
  leere_namen_auswerten('blau', mensch)
}
```

5.9 leere_namen_auswerten

Im Unterprogramm

```
function leere_namen_auswerten(farbe, mensch) {
```

löschen wir einen Namenslisteneintrag, bei dem gerade der letzte Buchstabe gelöscht wurde, beziehungsweise erstellen einen neuen, leeren Eintrag, wenn es gerade kein leeres Feld gibt. Dazu entscheiden wir anhand der über die Parameterliste hereinkommenden Farbe als erstes, um welche Tabelle es gerade geht

```
  var tabelle = document.getElementById('table_' + farbe)
```

und bestimmen die Anzahl der in der Liste vorhandenen Einträge (einschließlich der möglicherweise leeren Einträge):

```
  var n_mensch = tabelle.rows.length
```

Im nächsten Schritt wollen wir dann die Anzahl der leeren Einträge ermitteln. Dazu initialisieren wir einen entsprechenden Zähler

```
  var n_leere_menschen = 0
```

und beginnen eine Schleife über alle Listeneinträge:

```
  for (var i_mensch = 1; i_mensch < n_mensch; i_mensch++) {
```

In der Schleife untersuchen wir, ob der aktuelle Namenseintrag (`cells[1]`) leer ist:

```
    if (tabelle.rows[i_mensch].cells[1].children[0].value
      == '') {
```

Wenn dies der Fall ist, erhöhen wir den entsprechenden Zähler:

```
      n_leere_menschen++
    }
  }
```

Wir möchten nun dafür sorgen, dass es immer genau einen leeren Eintrag am Ende der Liste gibt. Wenn die Lehrkraft aber gerade den ersten Buchstaben in dieses leere Namensfeld eingetragen hat, gibt es in der Liste keinen leeren Eintrag mehr. In diesem Fall

```
  if (n_leere_menschen == 0) {
```

erzeugen wir mittels neuen_menschen_anlegen einen neuen, leeren Eintrag am Ende der Liste:

```
    neuen_menschen_anlegen(farbe)
  }
```

Wenn die Lehrkraft hingegen gerade den letzten Buchstaben eines vorhandenen Namens gelöscht hat, gibt es plötzlich zwei leere Einträge. In diesem Fall

```
  else if (n_leere_menschen > 1) {
```

löschen wir einfach den gerade leer gewordenen Namenseintrag:

```
    aktuellen_menschen_loeschen(mensch)
  }
}
```

5.10 auf_ding_geklickt

In seite_claroma haben wir definiert, dass das Unterprogramm

```
function auf_ding_geklickt(ding) {
```

aufgerufen wird, wenn die Lehrkraft auf einen Gegenstand klickt. Wenn dies der Fall ist, hängen wir das Symbol des Gegenstandes an den Mauszeiger:

```
  aktuelles_objekt = ding
}
```

5.11 auf_menschen_geklickt

In neuen_menschen_anlegen haben wir definiert, dass das Unterprogramm

```
function auf_menschen_geklickt(mensch) {
```

aufgerufen wird, wenn die Lehrkraft auf das Symbol eines Menschen klickt. Hier untersuchen wir als erstes, ob es sich um ein leeres (schon in den Raum gesetztes) Symbol in einer Menschenliste handelt:

```
  if (mensch.getAttribute('class') === 'feld mensch leer') {
```

In diesem Fall wollen wir das Symbol aus dem Raum wieder zurück in die Liste holen. Dazu suchen wir mit menschen_in_zelle_finden das Symbol im Raum

```
    mensch_in_zelle = menschen_in_zelle_finden(mensch.id)
```

und kopieren seine CSS-Klasse (und damit seine Farbe) zurück in die Liste:

```
    mensch.setAttribute('class',
      mensch_in_zelle.getAttribute('class'))
```

Abschließend löschen wir mit menschen_in_zelle_loeschen das Symbol aus dem Raum:

```
    menschen_in_zelle_loeschen(mensch.id)
  }
```

In beiden Fällen hängen wir das aktuelle Menschensymbol an den Mauszeiger:

```
  aktuelles_objekt = mensch
}
```

5.12 in_zelle_eingetreten

Wir möchten, dass das gerade ausgewählte Symbol (egal, ob Gegenstand oder Mensch) im Raum am Mauszeiger hängt und durch Klicken in einer Zelle abgelegt werden kann. Dazu verwenden wir die drei Unterprogramme

- in_zelle_eingetreten (Zwischenspeichern des Symbols, das sich momentan in der Zelle befindet),
- in_zelle_geklickt (Ablegen des aktuellen Symbols in der Zelle)
- aus_zelle_ausgetreten (Restaurieren des zwischengespeicherten Symbols)

Gemäß leeren_raum_anlegen wird jedes Mal, wenn der Mauszeiger in eine neue Raumzelle eintritt, das Unterprogramm

```
function in_zelle_eingetreten(zelle) {
```

aufgerufen. In diesem initialisieren wir die globale Variable gesetzt, die später gegebenenfalls in in_zelle_geklickt gesetzt und in aus_zelle_ausgetreten interpretiert wird, auf false:

```
  gesetzt = false
```

Dann kopieren wir das Symbol, das sich momentan in der Zelle befindet, in einen Zwischenspeicher

```
  a_nach_b_kopieren(zelle, objekt_puffer)
```

und das aktuell ausgewählte Symbol, das ja am Mauszeiger hängt, in die Zelle:

```
  a_nach_b_kopieren(aktuelles_objekt, zelle)
}
```

5.13 in_zelle_geklickt

Das Unterprogramm

```
function in_zelle_geklickt(zelle) {
```

haben wir in leeren_raum_anlegen als das Unterprogramm definiert, das aufgerufen wird, wenn die Lehrkraft in eine Zelle des Raumes klickt. In der Zelle selbst müssen wir jetzt eigentlich nichts mehr erledigen, da wir in in_zelle_eingetreten ja schon das aktuelle Symbol in die Zelle kopiert haben. Durch

```
  gesetzt = true
```

signalisieren wir lediglich dem beim Verlassen der Zelle aufzurufenden aus_zelle_ausgetreten, dass es das neue, in der Zelle abgelegt Symbol auch dort belassen und nicht wieder durch die alte, zwischengespeicherte Version ersetzen soll.

Wenn es hier nur um Gegenstände gehen würde, wären wir jetzt mit dem Unterprogramm fertig. Da Menschensymbole aber sowohl in ihrer Liste als auch in den Zellen des Raumes auftreten können, müssen wir diese gesondert betrachten. Wenn nämlich beim Klicken in die Zelle ein Symbol vom Typ Mensch am Mauszeiger hängt

```
  if (aktuelles_objekt.getAttribute('class').
    indexOf('feld mensch') >= 0) {
```

dann entfernen wir das Symbol (die Farbe) aus der Liste

```
    aktuelles_objekt.setAttribute('class', 'feld mensch leer')
```

weil das Symbol ja praktisch aus der Liste in die Zelle bewegt wurde. Außerdem hängen wir das Löschsymbol an den Mauszeiger

```
    aktuelles_objekt = leer
  }
```

damit beim nächsten Klicken auf ein Menschensymbol der Mensch aus der Zelle aufgenommen und an den Mauszeiger gehängt wird, was sehr praktisch ist, wenn wir Menschen von einem Platz auf einen neuen Platz umsetzen wollen.

Wenn in diesem Fall also ein Menschensymbol in der Zelle lag und daher beim Betreten der Zelle in den Zwischenspeicher kopiert wurde

```
  if (objekt_puffer.getAttribute('class').
    indexOf('feld mensch') >= 0) {
```

besorgen wir uns aus dem Zwischenspeicher die ID des Menschen, die ja in die data-id der Zelle kopiert wurde

```
    var data_id = objekt_puffer.getAttribute('data-id')
```

und über seine ID den Listeneintrag des Menschen selbst:

```
    var mensch = document.getElementById(data_id)
```

Da wir durch das Klicken den Menschen ja aus der Zelle des Raumes entfernt haben, bewegen wir ihn praktisch wieder zurück auf seinen Listenplatz, indem wir den Listeneintrag wieder mit seiner CSS-Klasse (Originalfarbe) füllen:

```
    mensch.setAttribute('class',
      objekt_puffer.getAttribute('class'))
```

Außerdem hängen wir den gerade aus der Zelle entfernten Menschen an den Mauszeiger

```
    aktuelles_objekt = mensch
  }
}
```

um ihn im nächsten Schritt an anderer Stelle wieder absetzen zu können.

5.14 aus_zelle_ausgetreten

Das Unterprogramm

```
function aus_zelle_ausgetreten(zelle) {
```

wird laut `leeren_raum_anlegen` immer dann aufgerufen, wenn der Mauszeiger eine Zelle wieder verlässt. Wenn die Lehrkraft vorher in die Zelle hinein geklickt hat, gibt es hier überhaupt nichts zu tun, da wir alles Notwendige schon in `in_zelle_geklickt` erledigt haben.

Wenn hingegen nicht in die Zelle geklickt wurde

```
  if (!gesetzt) {
```

müssen wir das in `in_zelle_eingetreten` durchgeführte Ersetzen des Zellinhaltes durch das aktuelle Symbol wieder rückgängig machen. Dazu kopieren wir einfach mittels `a_nach_b_kopieren` den im Unterprogramm `in_zelle_eingetreten` zwischengespeicherten Originalzellinhalt wieder in die Zelle zurück:

```
    a_nach_b_kopieren(objekt_puffer, zelle)
  }
}
```

5.15 aktuellen_menschen_loeschen

Wenn die Lehrkraft den letzten Buchstaben eines Menschen gelöscht hat, wird in `leere_namen_auswerten` das Unterprogramm

```
function aktuellen_menschen_loeschen(mensch) {
```

aufgerufen, das sowohl seinen Listeneintrag als auch das Symbol des Menschen aus dem Raum entfernt. Zum Löschen des Symbols rufen wir das Unterprogramm menschen_in_zelle_loeschen mit der ID des zu löschenden Menschen auf:

```
  menschen_in_zelle_loeschen(mensch.id)
```

Zum Entfernen des Listeneintrags löschen wir in der Tabelle des Menschen die Zeile, in dessen Zelle sich das Symbol des Menschen befindet:

```
  mensch.parentNode.parentNode.deleteRow(
    mensch.parentNode.rowIndex)
}
```

5.16 menschen_in_zelle_loeschen

Um das Symbol eines Menschen aus einer Zelle des Raumes zu löschen

```
function menschen_in_zelle_loeschen(mensch_id) {
```

untersuchen wir mittels menschen_in_zelle_finden, ob sich ein Mensch mit der gesuchten ID in irgendeiner Zelle befindet:

```
  var mensch_in_zelle = menschen_in_zelle_finden(mensch_id)
```

Wenn dies der Fall ist

```
  if (mensch_in_zelle !== null) {
```

kopieren wir einfach das leere Löschsymbol in die entsprechende Zelle:

```
    a_nach_b_kopieren(leer, mensch_in_zelle)
  }
}
```

5.17 menschen_in_zelle_finden

Das Unterprogramm

```
function menschen_in_zelle_finden(mensch_id) {
```

findet eine Zelle im Raum, in der sich das Symbol eines Menschen mit der bestimmten ID befindet. Dazu verwenden wir den Befehl querySelector, mit dem wir alle Objekte des aktuellen Dokumentes finden, deren data-id die ID des gesuchten Menschen beinhaltet:

```
  return document.querySelector('[data-id=' + mensch_id + ']')
}
```

Da jeder Mensch nur maximal einmal im Raum gesetzt worden sein kann, hat die zurückgegebene „Liste" immer maximal eine Länge von eins.

5.18 speichern_anzeigen

Das Unterprogramm

```
function speichern_anzeigen() {
```

wird ausgeführt, wenn die Lehrkraft durch Anklicken der Speichern-Schaltfläche die Seite seite_speichern öffnet. Auf dieser Seite sammeln wir alle Informationen über den aktuellen Sitzplan und geben der Lehrkraft die Möglichkeit, den Plan in eine Datei auf ihrer Festplatte abzuspeichern, um sie später wieder zu laden.

Als erstes deklarieren wir das Speicherobjekt claroma, das wir im Folgenden mit den abzuspeichernden Informationen füllen werden:

```
  var claroma = {}
```

Im Objekt speichern wir nun die aktuelle Programmversion

```
  claroma.version = 2.0
```

und den Namen der Klasse, den wir aus dem entsprechenden Eingabefeld lesen:

```
  claroma.klassen_name = klassen_name.value
```

Für das Datum holen wir uns das aktuelle Datumsobjekt

```
  var datum = new Date()
```

und basteln uns ein eigenes Format[4] der Form Jahr_Monat_Tag_Stunde_Minute_Sekunde:

```
  claroma.datum =
    datum.getFullYear() + '_' +
    parseInt(datum.getMonth() + 1) + '_' +
    datum.getDate() + '_' +
    datum.getHours() + '_' +
    datum.getMinutes() + '_' +
    datum.getSeconds()
```

Menschen

Als nächstes wollen wir die Daten der Menschen in das Speicherobjekt ablegen. Dazu beschaffen wir uns eine Liste aller Zeilen der „weiblichen" Tabelle

```
  var rosa_zeilen = table_rosa.rows
```

und bestimmen ihre Länge:

```
  var n_rosa = rosa_zeilen.length
```

[4] Da JavaScript als nullbasierte Sprache bekloppterweise den Januar als Monat 0 ausgibt, müssen wir dabei zu jedem Monat noch eine 1 hinzuaddieren.

Für die Tabellendaten[5] erzeugen wir ein leeres Feld

```
  var array_rosa = []
```

und beginnen eine Schleife über alle Zeilen:

```
  for (var i_zeile = 0; i_zeile < n_rosa; i_zeile++) {
```

Für jede Zeile erzeugen wir ein eigenes leeres Objekt

```
    var rosa = {}
```

in das wir für jeden Menschen seine ID

```
    rosa.id = rosa_zeilen[i_zeile].cells[0].id
```

seine CSS-Klasse

```
    rosa.class = rosa_zeilen[i_zeile].cells[0]
      .getAttribute('class')
```

seinen Aufdruck

```
    rosa.aufdruck = rosa_zeilen[i_zeile].cells[0].innerHTML
```

und seinen Namen einfügen:

```
    rosa.name = rosa_zeilen[i_zeile].cells[1].children[0].value
```

Jedes neue gefüllte Objekt hängen wir dann an das wachsende Feld an:

```
    array_rosa.push(rosa)
  }
```

Schließlich übertragen wir die Anzahl der Namen

```
  claroma.n_rosa = n_rosa
```

und die im Feld abgespeicherten Daten der Menschen in das Speicherobjekt:

```
  claroma.array_rosa = array_rosa
```

Die gleichen Schritte wiederholen wir für die „männliche” Tabelle:

```
  var blau_zeilen = table_blau.rows
  var n_blau = blau_zeilen.length
  var array_blau = []
  for (var i_zeile = 0; i_zeile < n_blau; i_zeile++) {
    var blau = {}
    blau.id = blau_zeilen[i_zeile].cells[0].id
```

[5] Leider gibt es in JavaScript keine direkte Möglichkeit, die Inhalte einer HTML-Tabelle zu serialisieren. Wir müssen sie daher erst selbst in eigene Objekte umkopieren und diese dann in eine JSON-Zeichenkette umwandeln lassen.

```
    blau.class = blau_zeilen[i_zeile].cells[0]
      .getAttribute('class')
    blau.aufdruck = blau_zeilen[i_zeile].cells[0].innerHTML
    blau.name = blau_zeilen[i_zeile].cells[1].children[0].value
    array_blau.push(blau)
  }
  claroma.n_blau = n_blau
  claroma.array_blau = array_blau
```

Raum

Zum Abspeichern der Raumdaten gehen wir genauso vor wie beim Abspeichern der Menschendaten; mit dem kleinen Unterschied, dass wir jetzt auch beliebig viele Zellen in einer Zeile haben können. Wir speichern also alle Zeilen der Raumtabelle zwischen

```
  var raum_zeilen = table_raum.rows
```

und erzeugen ein leeres Feld:

```
  var array_raum = []
```

Die äußere Schleife geht wieder über alle Zeilen:

```
  for (var i_zeile = 0; i_zeile < n_raum_zeilen; i_zeile++) {
```

Jede Zeile besteht jetzt aber aus beliebig vielen Zellen, so dass wir ein weiteres Feld für jede Zeile anlegen müssen:

```
    var array_zeile = []
```

Jetzt beginnen wir die innere Schleife über alle Zellen (Spalten):

```
    for (var i_spalte = 0; i_spalte < n_raum_spalten; i_spalte++)
      {
```

Jede Zelle ist wieder ein eigenes Objekt

```
      var zelle = {}
```

in dem wir die `data-id`

```
      zelle.data_id = raum_zeilen[i_zeile].cells[i_spalte]
        .getAttribute('data-id')
```

und die CSS-Klasse der Tabellenzelle ablegen:

```
      zelle.class = raum_zeilen[i_zeile].cells[i_spalte]
        .getAttribute('class')
```

Schließlich müssen wir nur noch die Zelle an die Zeile

```
      array_zeile.push(zelle)
    }
```

und die Zeile an das Feld anhängen:

```
    array_raum.push(array_zeile)
  }
```

Damit können wir jetzt auch die Raumdaten im Speicherobjekt ablegen:

```
  claroma.n_raum_zeilen = n_raum_zeilen
  claroma.n_raum_spalten = n_raum_spalten
  claroma.array_raum = array_raum
```

Beziehungen und Festlegungen

Schließlich übertragen wir noch die gegebenenfalls auf der Optimierungsseite definierten Beziehungen und Festlegungen ins Speicherobjekt:

```
  claroma.beziehungen = beziehungen
  claroma.festlegungen = festlegungen
```

Speichern

Normalerweise sollte ein JavaScript-Programm aus Sicherheitsgründen selbst keine Dateien auf der Festplatte eines Nutzers speichern. Da wir der Lehrkraft aber genau diese Möglichkeit geben möchten, den gerade erstellten Plan auf dem eigenen Rechner zu speichern, verwenden wir dazu eine Bibliothek [5], die wir in `<head>` eingebunden haben und die in allen modernen Browsern zu funktionieren scheint.

Als erstes setzen wir den Dateinamen, unter dem wir den Plan abspeichern wollen, aus dem von der Lehrkraft eingegebenen Klassennamen[6] und dem aktuellen Datum in unserem eigenen Format zusammen

```
  var datei_name =
    claroma.klassen_name + '_' + claroma.datum + '.json'
```

und verwenden ihn auf der Seite `seite_speichern`, um die Lehrkraft darüber zu informieren, wo und unter welchem Namen sie die abgespeicherte Datei zum späteren Laden finden kann:

```
  speicher_datei_name.innerHTML = datei_name
```

Als nächstes serialisieren wir das Speicherobjekt `claroma` in eine JSON-Zeichenkette:

```
  zustand = JSON.stringify(claroma, null, '\t')
```

Dabei bewirkt der Parameter `'\t'`, dass die Daten später in der Datei (`claroma.json`) hübsch formatiert[7] dargestellt werden und damit leichter les- und interpretierbar sind.

[6] Wenn die Lehrkraft keinen Klassennamen eingetragen hat, fehlt dieser im Dateinamen natürlich. Der Dateiname beginnt dann mit dem Unterstrich, der normalerweise den Klassennamen und das Datum verbindet.

[7] Durch die zusätzlich eingebauten Tabulatoren und Zeilenumbrüche wird die Datei zwar insgesamt etwas länger; bei einer Nominallänge von etwa 30 kB lohnt es sich aber wohl nicht, darüber lange nachzudenken.

Die saveAs-Methode der Bibliothek erwartet die abzuspeichernden Daten in Form eines BLOBs (Binary Large Object). Wir kapseln daher die Planzustandszeichenkette durch eckige Klammern in einem Array und erzeugen daraus das entsprechende BLOB mit dem passenden MIME-Typ und Zeichensatz:

```
  var blob_zustand = new Blob([zustand],
    { type: "application/json;charset=utf-8" })
```

Nach dieser Vorarbeit können wir den aktuellen Planzustand mit Hilfe von [5] sehr einfach direkt[8] auf die Festplatte der Lehrkraft schreiben:

```
  saveAs(blob_zustand, datei_name)
```

Schließlich müssen wir jetzt noch die Hauptseite verstecken und die Seite seite_speichern sichtbar machen:

```
  seite_claroma.style.display = 'none'
  seite_speichern.style.display = 'block'
}
```

5.19 datei_lesen

Wenn die Lehrkraft die Laden-Schaltfläche anklickt, wird die Seite seite_laden geöffnet. Wenn die Lehrkraft dort mit Hilfe der Schaltfläche Datei auswählen eine Datei zum Öffnen ausgewählt hat, wird das Unterprogramm

```
function datei_lesen(evt) {
```

ausgeführt. Dabei erhält das Unterprogramm den Namen der (gegebenenfalls mehreren) ausgewählten Dateien über seine Parameterliste in der Ereignisvariablen evt. Die Dateinamen speichern wir zunächst zwischen

```
  var dateien = evt.target.files
```

und wählen dann die erste Datei aus:

```
  var datei = dateien[0]
```

Zum Einlesen der Datei definieren wir einen Dateileser

```
  var leser = new FileReader()
```

und eine anonyme Funktion, die aufgerufen wird, wenn der Dateileser die Datei komplett gelesen hat:

```
  leser.onload = function () {
```

[8] Die einzelnen Browser entscheiden dabei, ob sie die Datei sofort ohne Nachfrage ins Downloadverzeichnis schreiben oder ob sie dem Nutzer noch die Möglichkeit geben, den Dateinamen zu ändern.

Wenn dies der Fall ist, wandeln (deserialisieren) wir die in der Datei gespeicherte Zeichenkette wieder in das JavaScript-Objekt `claroma` um:

```
    var claroma = JSON.parse(this.result)
```

Wir entnehmen `claroma` den Namen der Klasse

```
    klassen_name.value = claroma.klassen_name
```

und die Größe (Zeilen und Spalten) des Raumes:

```
    n_raum_zeilen = claroma.n_raum_zeilen
    n_raum_spalten = claroma.n_raum_spalten
```

Als nächstes löschen wir im aktuellen Plan mit `alle_menschen_loeschen` alle Menschen

```
    alle_menschen_loeschen()
```

und legen mittels `leeren_raum_anlegen` einen neuen, leeren Raum an:

```
    leeren_raum_anlegen()
```

Die Größe des neuen Raumes erhält das Unterprogramm dabei aus den globalen Variablen `n_raum_zeilen` und `n_raum_spalten`, die wir ja gerade aus der Datei gelesen haben. Jetzt können wir den neuen Raum anzeigen

```
    claroma_anzeigen()
```

und im Folgenden mit den aus der Datei gelesenen Daten füllen. Dazu bestimmen wir als erstes die Anzahl der Lehrerinnen und Schülerinnen

```
    var n_rosa = claroma.n_rosa
```

und starten dann eine Schleife über alle Einträge der „weiblichen” Liste:

```
    for (var i_rosa = 0; i_rosa < n_rosa; i_rosa++) {
```

Für jeden aus der Datei gelesenen Menschen erzeugen wir mittels `neuen_menschen_anlegen` einen neuen Listeneintrag mit seinen ebenfalls aus der Datei gelesenen Eigenschaften:

```
      neuen_menschen_anlegen(
        'rosa',
        claroma.array_rosa[i_rosa].id,
        claroma.array_rosa[i_rosa].class,
        claroma.array_rosa[i_rosa].aufdruck,
        claroma.array_rosa[i_rosa].name)
    }
```

Natürlich erzeugen wir auf die gleiche Weise auch die Einträge der „männlichen” Liste:

```
    var n_blau = claroma.n_blau
    for (var i_blau = 0; i_blau < n_blau; i_blau++) {
      neuen_menschen_anlegen(
        'blau',
        claroma.array_blau[i_blau].id,
        claroma.array_blau[i_blau].class,
        claroma.array_blau[i_blau].aufdruck,
        claroma.array_blau[i_blau].name)
    }
```

Da die ersten beiden Einträge der Listen für die Lehrkräfte vorgesehen sind, müssen wir in diesen explizit die richtigen Platzhalter setzen:

```
    table_rosa.rows[0].cells[1].children[0].setAttribute(
      'placeholder', 'Lehrerin')
    table_blau.rows[0].cells[1].children[0].setAttribute(
      'placeholder', 'Lehrer')
```

In den nächsten Schritten wollen wir den Raum mit den eingelesenen Daten füllen. Dazu speichern wir das eingelesene Raumfeld um

```
    var array_raum = claroma.array_raum
```

und starten eine Doppelschleife über alle Zeilen und Spalten des eingelesenen Raumfeldes:

```
    for (var i_zeile = 0; i_zeile < n_raum_zeilen; i_zeile++) {
      for (var i_spalte = 0;i_spalte<n_raum_spalten;i_spalte++) {
```

Innerhalb der Schleife entnehmen wir jeweils eine einzelne Zelle des eingelesenen Raumes

```
        var zelle = array_raum[i_zeile][i_spalte]
```

und suchen das Objekt (Gegenstand oder Mensch), auf das die `data_id` der eingelesenen Zelle zeigt:

```
        var ding_mensch = document.getElementById(zelle.data_id)
```

Dessen Eigenschaften kopieren wir dann mit `a_nach_b_kopieren` in die Zelle des Raumes

```
        a_nach_b_kopieren(ding_mensch,
          table_raum.rows[i_zeile].cells[i_spalte])
```

Dabei gibt es ein kleines Problem: Menschen, die schon in den Raum gesetzt wurden, besitzen in ihrer Liste ein leeres Symbol, das dann im gerade durchgeführten Schritt auch leer in den Raum kopiert wurde. Im Raum muss das Symbol aber mit der richtigen CSS-Klasse (farbig) ausgefüllt sein. Deshalb kopieren wir nachträglich noch in alle[9] Zellen ihre eingelesenen CSS-Klassen:

[9] Eigentlich wäre das Einfärben nur für die Menschen und nicht für die Gegenstände, deren Liste ja unverändert bleibt, nötig. Die Unterscheidung zwischen Mensch und Gegenstand wäre hier aber umständlicher als das pauschale Kopieren der CSS-Klassen aller Objekte.

```
        table_raum.rows[i_zeile].cells[i_spalte]
        .setAttribute('class', zelle.class)
      }
    }
```

Schließlich kopieren wir die für die Optimierung notwendigen Beziehungen und Festlegungen in ihre entsprechenden Objekte:

```
    beziehungen = claroma.beziehungen
    festlegungen = claroma.festlegungen
  }
```

Als letztes teilen wir dem Dateileser dann noch mit, aus welcher Datei er den vorher abgespeicherten Sitzplan lesen soll

```
  leser.readAsText(datei)
```

und sorgen dafür, dass beim nächsten Lesen gegebenenfalls wieder die gleiche Datei ausgewählt werden kann:

```
  input_datei_lesen.value = ''
}
```

5.20 menschen_sortieren

In Kapitel 4.2.1 haben wir definiert, dass das Unterprogramm

```
function menschen_sortieren() {
```

aufgerufen wird, wenn die Lehrkraft auf die Schaltfläche Sortieren klickt. Da wir in diesem Fall beide Menschenlisten sortieren wollen, rufen wir das Unterprogramm `tabelle_sortieren` für beide Listen auf:

```
  tabelle_sortieren('rosa')
  tabelle_sortieren('blau')
}
```

5.21 tabelle_sortieren

Im Unterprogramm

```
function tabelle_sortieren(farbe) {
```

erhalten wir die Angabe, welche Liste wir sortieren wollen, über die Parameterliste und ermitteln als erstes die entsprechende Tabelle

```
  var tabelle = document.getElementById('table_' + farbe)
```

und deren Anzahl von Einträgen:

```
  var n_zeilen = tabelle.rows.length
```

Als Sortierverfahren wählen wir ein leicht zu implementierendes Insertationsort [7], das keinen zusätzlichen Speicherplatz benötigt und das sehr effizient ist, wenn die meisten Einträge schon sortiert sind[10]. Wir beginnen dazu eine äußere Schleife über „alle" Einträge[11]

```
  for (var i_kandidat = 2; i_kandidat < n_zeilen; i_kandidat++) {
```

und vergleichen in einer inneren Schleife den aktuellen Kandidaten mit allen darüber liegenden Einträgen:

```
    for (var i_gegner = 1; i_gegner < i_kandidat; i_gegner++) {
```

Dazu beschaffen wir uns die Tabellenzeile des aktuellen Kandidaten

```
      kandidat_zeile = tabelle.rows[i_kandidat]
```

und seinen (rechts neben dem Symbol) eingetragenen Namen:

```
      kandidat_text = kandidat_zeile.cells[1].children[0].value
```

Um „sicherzustellen", dass ein leerer Eintrag immer ganz an den Schluss der Liste sortiert wird, ersetzen wir ihn – nur für den Sortiervergleich – durch einen Ausdruck, der mit sehr großer Wahrscheinlichkeit[12] lexikografisch hinter allen „normalen" Namen liegt:

```
      if (kandidat_text === "") {
        kandidat_text = "üüüüüüüüü"
      }
```

Jetzt müssen wir nur noch den Text des aktuellen Vergleichspartners ermitteln

```
      gegner_zeile = tabelle.rows[i_gegner]
      gegner_text = gegner_zeile.cells[1].children[0].value
```

und den Vergleich durchführen. Wenn also der Name des aktuellen Kandidaten lexikografisch vor dem des aktuellen Vergleichspartners liegt

```
      if (kandidat_text < gegner_text) {
```

[10] Aus Wikipedia: Das Vorgehen ist mit der Sortierung eines Spielkartenblatts vergleichbar. Am Anfang liegen die Karten des Blatts verdeckt auf dem Tisch. Die Karten werden nacheinander aufgedeckt und an der korrekten Position in das Blatt, das in der Hand gehalten wird, eingefügt. Um die Einfügestelle für eine neue Karte zu finden, wird diese sukzessive (von links nach rechts) mit den bereits einsortierten Karten des Blattes verglichen. Zu jedem Zeitpunkt sind die Karten in der Hand sortiert und bestehen aus den zuerst vom Tisch entnommenen Karten.

[11] Tatsächlich beginnen wir mit dem zweiten Eintrag.

[12] Überlegen Sie mal, wie Sie diesen Algorithmus austricksen können ...

dann[13] sortieren wir den aktuellen Kandidaten mittels zeile_verschieben vor seinem aktuellen Vergleichspartner ein:

```
            zeile_verschieben_vor(kandidat_zeile, i_gegner)
          }
        }
      }
    }
```

5.22 zeile_verschieben

Das Verschieben einer Tabellenzeile (zeile) an die Stelle mit dem neuen Index zeilenindex im Unterprogramm

```
function zeile_verschieben_vor(zeile, zeilen_index) {
```

hat eine ziemlich gewöhnungsbedürftige, kontraintuitive Syntax. Sie resultiert aus der Tatsache, dass HTML-Tabellenzeilen gleichzeitig auch Knoten (nodes) darstellen, über die sie beim Einfügen aus JavaScript heraus angesprochen werden müssen:

```
  zeile.parentNode.insertBefore(zeile,
    zeile.parentNode.childNodes[zeilen_index])
}
```

5.23 laden_anzeigen

Die folgenden drei Unterprogramme dienen dazu, beim Wechsel von einer „Seite" auf eine andere „Seite" die jeweils andere(n) Seiten zu verstecken. Wenn also die Lehrkraft die Schaltfläche Laden angeklickt hat und seite_laden angezeigt werden soll

```
function laden_anzeigen() {
```

verstecken wir die Hauptseite, von der wir ja gerade kommen

```
  seite_claroma.style.display = 'none'
```

und machen die gewünschte Seite sichtbar:

```
  seite_laden.style.display = 'block'
}
```

[13] Wenn der aktuelle Kandidat lexikografisch hinter allen seinen Vergleichspartnern (Vorgängern) liegt, müssen wir gar nichts verschieben; der Kandidat befindet sich dann ja schon an der richtigen Stelle.

5.24 impressum_anzeigen

Entsprechend verfahren wir nach dem Anklicken der Schaltfläche Impressum:

```
function impressum_anzeigen() {
  seite_claroma.style.display = 'none'
  seite_impressum.style.display = 'block'
}
```

5.25 claroma_anzeigen

Wenn wir von einer „Unterseite" zurück auf die Hauptseite möchten, zeigen wir die Hauptseite wieder an

```
  seite_claroma.style.display = 'block'
```

und verstecken alle übrigen Seiten

```
  seite_speichern.style.display = 'none'
  seite_laden.style.display = 'none'
  seite_impressum.style.display = 'none'
}
```

5.26 document.onkeydown

In modernen Browsern gelangen wir durch Drücken der Zurück-Taste (Backspace) auf der Tastatur auf die vorherige Seite zurück. Während dieses Verhalten in vielen Fällen durchaus sinnvoll sein kann, ist es fatal, wenn wir uns beispielsweise in einem Texteingabefeld befinden und dessen Inhalt durch wiederholtes Drücken der Zurück-Taste löschen wollen. Wenn dabei sehr vorsichtig sind und nach dem Löschen des letzten Buchstabens sofort rechtzeitig den Finger von der Zurück-Taste nehmen, geht alles gut. Wenn wir allerdings die Zurück-Taste dabei auch nur ein einziges Mal zu oft drücken, wechselt der Browser auf die vorherige Seite und alle Eingaben, die wir vorher auf der Claroma-Seite gemacht haben, sind verloren. Um dies auf jeden Fall zu verhindern, fangen wir das Drücken der Zurück-Taste ab und lassen es nur zu, wenn wir uns in einem Eingabefeld befinden. Dazu definieren wir ein Unterprogramm

```
document.onkeydown = function (event) {
```

das bei jedem Tastendruck aufgerufen wird. Im Unterprogramm ermitteln[14] wir als erstes das Objekt, in dem die Zurück-Taste gedrückt wurde:

[14] Natürlich muss der Internet Explorer hier mal wieder sein eigenes Süppchen kochen, weshalb wir auch seine Spezialsyntax (`srcElement`) berücksichtigen.

```
  var objekt = event.srcElement || event.target
```

Wenn wir uns nun zum Zeitpunkt des Tastendruckes in einem Eingabefeld befunden haben

```
  if (objekt.tagName == 'INPUT') {
```

lassen wir den Tastendruck durch:

```
    return true
  }
```

Wenn hingegen gerade kein Eingabefeld aktiv war und trotzdem die Zurück-Taste gedrückt wurde

```
  else if (event.keyCode == 8) {
```

unterbinden wir das Weiterleiten des Tastendruckes:

```
    return false
  }
}
```

5.27 sitzplan_optimieren

Im Rahmen des Unterprogrammes

```
function sitzplan_optimieren() {
```

versucht ein Optimierer, die Sitzplätze der Menschen gezielt zu variieren, um die von der Lehrkraft vorgegebenen Wünsche bestmöglich zu erfüllen.

Ein paar allgemeine Optimierungsgedanken: Diskrete Objekte in ihrer Anordnung solange zu variieren, bis ein von der Anordnung abhängiges Kostenfunktional minimal geworden ist, stellt ein in der kombinatorischen Optimierungstheorie hinreichend beleuchtetes Problem dar (*Bin packing*, *Traveling salesman*, *Wedding planner*, ...).

Nur für eine sehr kleine Anzahl von Objekten lässt sich das Problem durch Ausprobieren (*Brute force*) aller Alternativen lösen. Schon bei zehn Objekten gibt es

$$10! = 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 3\,628\,800$$

Permutationsmöglichkeiten, für die alle die Kostenfunktion berechnet werden muss; bei einer Klasse mit 20 Schülerinnen und Schüler explodiert die Anzahl der Sitzplananalysen in nicht durchführbare, astronomische Dimensionen:

$$20! = 20 \cdot 19 \cdot \cdots \cdot 3 \cdot 2 \cdot 1 = 2\,432\,902\,008\,176\,640\,000$$

In der Praxis wird daher immer ein Optimierer verwendet, der von einen Anfangszustand ausgehend versucht, sich möglichst intelligent schrittweise auf ein Optimum hin zu bewegen. Dabei stehen mittlerweile eine große Vielzahl von Optimierungsalgorithmen zur Verfügung (*Linear programming*, *Simulated annealing*, *Genetic algorithm*, *Ant colony optimization*, ...), die zwar nicht zwingend jedes globale Optimum finden, aber in akzeptabler Zeit akzeptable Näherungslösungen finden („Nicht optimal aber gut genug").

In unserem Fall verwenden wir zwei ineinander geschachtelte Schleifen, die beide jeweils alle Tauschkombinationen zweier Menschen bewerten: In der äußeren Schleife führen wir also eine erste Vertauschung zweier Menschen durch. Dann führen wir in der inneren Schleife eine Vertauschung durch und berechnen die Kostenfunktion für den aktuellen Sitzplan. Wurde jetzt eine Verbesserung erzielt, beginnen wir die äußere Schleife sofort aufs Neue; ohne Verbesserung führen wir die nächste innere Vertauschung durch, bis alle inneren Vertauschungen analysiert wurden. Wenn bis jetzt keine Verbesserung erzielt wurde, führen wir die nächste Vertauschung in der äußeren Schleife durch und durchlaufen wieder die komplette innere Schleife. Wenn dann schließlich alle Vertauschungen der äußeren Schleife (mit jeweils allen Vertauschungen der inneren Schleife) zu keinen Verbesserungen geführt haben, beenden wir die Optimierung (möglicherweise ohne den wirklich optimalen Sitzplan gefunden zu haben).

Die Anzahl der Zweiertauschkombinationen ist signifikant kleiner als die der vorher betrachteten Permutationen. Bei 20 Objekten gibt es beispielsweise

$$\binom{20}{2} = \frac{20 \cdot 19}{2} = 190$$

Zweiertauschkombinationen. Selbst ein kompletter Durchlauf der Doppelschleife (also alle Zweiertauschkombinationen der äußeren Schleife multipliziert mit jeweils allen Zweiertauschkombinationen der inneren Schleife) benötigt dann nur

$$190 \cdot 190 = 36\,100$$

Auswertungen der Kostenfunktion, was auf heutiger Hardware nur einen kurzen Augenblick dauert. Natürlich findet auch dieser Algorithmus nicht in jedem Fall das globale Optimum. Unzählige Versuche mit realistischen Anordnungen und Beziehungsforderungen bescheinigen ihm aber eine sehr hohe Erfolgsquote.

Das Unterprogramm selbst bereitet erst einmal nur die Optimierungsseite vor. Dazu leert es die beiden Tabellen auf der Seite

```
table_beziehungen.innerHTML = ''
table_festlegungen.innerHTML = ''
```

und ruft das Unterprogramm `menschen_struktur_erstellen` auf, in dem alle im Plan gesetzten Menschen in ein Strukturfeld einsortiert werden:

```
  menschen_struktur_erstellen()
```

Als nächstes werden die beiden Unterprogramme `festlegungen_einsortieren` und `beziehungen_einsortieren` aufgerufen, in denen die aktuellen Festlegungen und Beziehungen in die Tabellen[15] einsortiert werden:

```
  festlegungen_einsortieren()
  beziehungen_einsortieren()
```

Schließlich verstecken wir die Hauptseite und stellen die Optimierungsseite mit dem Unterprogramm `optimieren_anzeigen` dar:

```
  optimieren_anzeigen()
}
```

5.28 menschen_struktur_erstellen

Im Unterprogramm

```
function menschen_struktur_erstellen() {
```

bauen wir ein Strukturfeld auf, das alle im Sitzplan gesetzten[16] Menschen mit ihren für die Optimierung notwendigen Informationen beinhaltet. Dazu initialisieren wir ein leeres Feld

```
  menschen = []
```

und füllen es in einer doppelten Schleife über alle Plätze des Sitzplans:

```
  for (var i_zeile = 0; i_zeile < n_raum_zeilen; i_zeile++) {
  for (var i_spalte = 0; i_spalte < n_raum_spalten; i_spalte++) {
```

In der inneren Schleife speichern wir die aktuelle Zelle des Sitzplans zwischen

```
    var zelle = table_raum.rows[i_zeile].cells[i_spalte]
```

und untersuchen, ob sich in der aktuellen Zelle ein Mensch befindet:

```
      if (zelle.getAttribute('class').
        indexOf('feld mensch') >= 0) {
```

Wenn dies der Fall ist, legen wir eine neue Struktur an

```
        var mensch = {}
```

[15] Da es in JavaScript nicht möglich ist, HTML-Tabellen direkt zum Abspeichern zu serialisieren, halten wir die entsprechenden Informationen in den globalen Variablen `festlegungen` und `beziehungen` und befüllen die Tabellen nur zur Interaktion mit der Lehrkraft.

[16] Es werden nur die im Sitzplan vergebenen Plätze optimiert. Menschen, die zwar in einer Liste angelegt aber noch nicht im Plan gesetzt sind, werden bei der Optimierung nicht angefasst.

und füllen sie mit den für die Optimierung notwendigen Daten der aktuellen Zelle:

```
        mensch.class = zelle.getAttribute('class')
        mensch.data_id = zelle.getAttribute('data-id')
        mensch.aufdruck = zelle.innerHTML
        mensch.raum_zeilen_index = i_zeile
        mensch.raum_spalten_index = i_spalte
        mensch.name = finde_name_zu_id(mensch.data_id)
```

Schließlich hängen wir die gerade erzeugte Struktur an das Feld an:

```
        menschen.push(mensch)
      }
    }
  }
```

Wenn wir alle Menschen ins Feld einsortiert haben, speichern wir seine Länge in einer globalen Variable ab:

```
  n_menschen = menschen.length
}
```

5.29 beziehungen_einsortieren

Im Unterprogramm

```
function beziehungen_einsortieren() {
```

sortieren wir die in der globalen Variablen `beziehungen` definierten Beziehungen zwischen zwei Menschen in die entsprechende Tabelle auf der Optimierungsseite ein. Dazu gehen wir in einer Schleife durch alle Beziehungen in der globalen Variablen

```
  for (var i_beziehung = 0; i_beziehung < beziehungen.length;
    i_beziehung++) {
```

und initialisieren zwei Indexvariablen:

```
    var i_1 = -1
    var i_2 = -1
```

In einer Schleife über alle zu optimierenden Menschen

```
    for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

untersuchen wir, ob der aktuelle Mensch auf der linken Seite der aktuellen Beziehung auftritt:

```
      if (beziehungen[i_beziehung][0] ==
        menschen[i_mensch].data_id) {
```

Wenn dies der Fall ist, speichern wir den Index des aktuellen Menschen in der ersten Indexvariablen:

```
        i_1 = i_mensch
      }
```

Wenn der aktuelle Mensch auf der rechten Seite einer Beziehung auftritt, speichern wir seinen Index in der zweiten Indexvariablen:

```
      if (beziehungen[i_beziehung][2] ==
        menschen[i_mensch].data_id) {
        i_2 = i_mensch
      }
    }
```

Wenn dann für die aktuelle Beziehung beide Menschen gefunden wurden

```
    if (i_1 > -1 && i_2 > -1) {
```

hängen wir mittels des Unterprogramm `neue_beziehung_erstellen` eine neue leere Beziehungszeile an die Beziehungstabelle an

```
      neue_beziehung_erstellen()
```

und wählen in der linken Auswahlliste der neuen Beziehungszeile den ersten gefundenen Menschen aus:

```
      table_beziehungen.rows
        [table_beziehungen.rows.length - 1].
        cells[0].children[0].value =
        menschen[i_1].data_id
```

In gleicher Weise verwenden wir die aktuelle Beziehung, um in der neuen Beziehungszeile das passende Beziehungszeichen auszuwählen

```
      table_beziehungen.rows
        [table_beziehungen.rows.length - 1].
        cells[1].children[0].value =
        beziehungen[i_beziehung][1]
```

und in der rechten Auswahlliste der neuen Beziehungszeile den zweiten gefundenen Menschen auszuwählen:

```
      table_beziehungen.rows
        [table_beziehungen.rows.length - 1].
        cells[2].children[0].value =
        menschen[i_2].data_id
    }
  }
}
```

5.30 festlegungen_einsortieren

Im Unterprogramm

```
function festlegungen_einsortieren() {
```

sortieren wir die in der globalen Variablen `festlegungen` definierten Festlegungen (Menschen, die während der Optimierung nicht verschoben werden sollen) in die entsprechende Tabelle auf der Optimierungsseite ein. Dazu gehen wir in einer Schleife durch alle Festlegungen in der globalen Variablen

```
for (var i_festlegung = 0; i_festlegung < festlegungen.length;
  i_festlegung++) {
```

und starten für jede Festlegung eine weitere Schleife über alle im Sitzplan gesetzten Menschen:

```
    for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

Wenn dann die ID des aktuellen Menschen in der Festlegungsliste auftaucht

```
      if (festlegungen[i_festlegung] ==
        menschen[i_mensch].data_id) {
```

hängen wir mittels des Unterprogramm `neue_festlegung_erstellen` eine neue leere Festlegungszeile an die Festlegungstabelle an

```
        neue_festlegung_erstellen()
```

und wählen in der Auswahlliste der neuen Festlegungszeile den aktuellen Menschen aus:

```
        table_festlegungen.rows
          [table_festlegungen.rows.length - 1].
          cells[0].children[0].value =
          menschen[i_mensch].data_id
      }
    }
  }
}
```

5.31 neue_beziehung_erstellen

Das Unterprogramm

```
function neue_beziehung_erstellen() {
```

wird von `beziehungen_aus_tabelle_lesen` aufgerufen und hängt eine neue leere Beziehungszeile an die Tabelle auf der Optimierungsseite an. Dazu hängen wir eine neue Tabellenzeile an die Tabelle an

```
  var neue_beziehung =
    table_beziehungen.insertRow(table_beziehungen.length)
```

und füllen sie mit vier Zellen:

```
  var zelle_links = neue_beziehung.insertCell(0)
  var zelle_mitte = neue_beziehung.insertCell(1)
  var zelle_rechts = neue_beziehung.insertCell(2)
  var zelle_loeschen = neue_beziehung.insertCell(3)
```

Für die erste Zelle erzeugen wir eine Auswahlliste

```
  var mensch_links = document.createElement('select')
```

und füllen die Auswahlliste mit den Namen und IDs aller zu optimierenden Menschen, indem wir in einer Schleife über alle Menschen

```
  for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

einen neuen Eintrag mit dem Namen (angezeigter Text) und der ID (zurückgegebener Wert) des aktuellen Menschen erzeugen

```
    var option = new Option(
      menschen[i_mensch].name,
      menschen[i_mensch].data_id)
```

und diesen Eintrag an die Auswahlliste anhängen:

```
    mensch_links.options.add(option)
  }
```

Jetzt müssen wir nur noch die Auswahlliste in die erste Zelle einfügen:

```
  zelle_links.appendChild(mensch_links)
```

Auch für die zweite Zelle erzeugen wir eine Auswahlliste

```
  var relation = document.createElement('select')
```

deren Einträge die Zeichenketten '+++' ... '---' sind:

```
  var option = new Option('+++', 3)
  relation.add(option)
  var option = new Option('++', 2)
  relation.add(option)
  var option = new Option('+', 1)
  relation.add(option)
  var option = new Option('o', 0, 'true', 'true')
  relation.add(option)
  var option = new Option('-', -1)
  relation.add(option)
  var option = new Option('--', -2)
```

```
  relation.add(option)
  var option = new Option('---', -3)
  relation.add(option)
```

Als Rückgabewerte geben wir dabei die Ganzzahlen `3` bis `-3` vor und wählen den neutralen Operator (`'o'`) als Defaultwert aus.

Die Auswahlliste fügen wir dann in die zweite Zelle ein:

```
  zelle_mitte.appendChild(relation)
```

Jetzt fehlt noch der zweite (rechte) Mensch der Beziehung. Auch für diesen erzeugen wir eine Auswahlliste:

```
  var mensch_rechts = document.createElement('select')
```

Da die Elemente der rechten Auswahlliste aber die gleichen wie die der linken sind, kopieren wir einfach den „Inhalt" der linken Liste nach rechts:

```
  mensch_rechts.innerHTML = mensch_links.innerHTML
```

Natürlich müssen wir auch noch die rechte Liste in die Tabellenzelle einfügen:

```
  zelle_rechts.appendChild(mensch_rechts)
```

Rechts neben jeder Beziehungszeile soll es eine Schaltfläche geben, mit der die Beziehung gelöscht werden kann. Dazu erzeugen wir eine neue Schaltfläche

```
  var button_loeschen = document.createElement('button')
```

legen ihre Beschriftung

```
  button_loeschen.innerHTML = 'Diese Beziehung löschen'
```

und das Unterprogramm (`zeile_loeschen`) fest, das aufgerufen wird, wenn die Lehrkraft auf die Schaltfläche klickt

```
  button_loeschen.setAttribute('onclick', 'zeile_loeschen(this)')
```

und legen die CSS-Klasse der Schaltfläche fest:

```
  button_loeschen.setAttribute('class', 'button_lang')
```

Schließlich fügen wir die Schaltfläche in die vierte Zelle der Tabellenzeile ein:

```
  zelle_loeschen.appendChild(button_loeschen)
}
```

5.32 neue_festlegung_erstellen

Im Unterprogramm

```
function neue_festlegung_erstellen() {
```

erzeugen wir eine neue leere Festlegungszeile in der Tabelle auf der Optimierungsseite. Dazu hängen wir eine neue Tabellenzeile an die Tabelle an

```
  var neue_festlegung =
    table_festlegungen.insertRow(table_festlegungen.length)
```

und fügen in der neuen Zeile zwei neue Zellen ein:

```
  var zelle_mensch = neue_festlegung.insertCell(0)
  var zelle_loeschen = neue_festlegung.insertCell(1)
```

Die erste Zelle soll den festzulegenden Menschen beinhalten. Dazu erzeugen wir eine Auswahlliste

```
  var mensch = document.createElement('select')
```

und füllen sie in einer Schleife über alle zu optimierenden Menschen

```
  for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

jeweils mit dem Namen und der ID des Menschen:

```
    var option = new Option(
      menschen[i_mensch].name,
      menschen[i_mensch].data_id)
```

Schließlich hängen wir den aktuellen Eintrag an die Auswahlliste

```
    mensch.options.add(option)
  }
```

und fügen die Auswahlliste in die erste Zelle ein:

```
  zelle_mensch.appendChild(mensch)}
```

In der zweiten Zelle einer jeden Festlegung wollen wir eine Schaltfläche zum Löschen der Festlegung vorsehen. Dazu erzeugen wir eine Schaltfläche

```
  var button_loeschen = document.createElement('button')
```

definieren ihren Aufdruck

```
  button_loeschen.innerHTML = 'Diese Festlegung löschen'
```

das aufzurufende Unterprogramm `zeile_loeschen`

```
  button_loeschen.setAttribute('onclick', 'zeile_loeschen(this)')
```

und die CSS-Klasse der Schaltfläche

```
  button_loeschen.setAttribute('class', 'button_lang')
```

und bauen die Schaltfläche in die Tabellenzelle ein:

```
  zelle_loeschen.appendChild(button_loeschen)
```

5.33 zeile_loeschen

Das Unterprogramm

```
function zeile_loeschen(sender) {
```

dient dazu, eine Zeile in einer der Tabellen der Optimierungsseite zu löschen. Da das Objekt, das dieses Unterprogramm aufruft, aber die Schaltfläche in der Zelle in der Zeile in der Tabelle ist, müssen wir uns erst einmal mit einem dreifachen `parentNode` bis hoch zur Tabelle hangeln, bevor wir die Zeile selbst (zweifaches `parentNode`) löschen können:

```
  sender.parentNode.parentNode.parentNode.deleteRow(
    sender.parentNode.parentNode.rowIndex)
}
```

5.34 optimieren_anzeigen

Das Unterprogramm

```
function optimieren_anzeigen() {
```

wird von `sitzplan_optimieren` aufgerufen, versteckt die Hauptseite

```
  seite_claroma.style.display = 'none'
```

und macht die Optimierungsseite sichtbar:

```
  seite_optimieren.style.display = 'block'
}
```

5.35 finde_name_zu_id

Das Unterprogramm

```
function finde_name_zu_id(id) {
```

wird von menschen_struktur_erstellen aufgerufen, um den zu der ID passenden Namen herauszufinden. Dazu suchen wir als erstes das Objekt mit der fraglichen ID in einer der beiden Menschentabellen auf der Hauptseite:

```
  var mensch_in_tabelle = document.getElementById(id)
```

Als nächstes besorgen wir uns den Zeilenindex des Menschenobjektes in seiner Tabelle:

```
  var tabellen_zeilen_index =
    mensch_in_tabelle.parentNode.rowIndex
```

Die Tabelle selbst erhalten wir über ein dreifaches `parentNode` (Menschenobjekt in Zelle in Zeile in Tabelle):

```
  var tabelle =
    document.getElementById(
    mensch_in_tabelle.parentNode.parentNode.parentNode.id)
```

Jetzt[17] können wir mit Hilfe des vorher bestimmten Zeilenindexes wieder in die Tabelle heruntersteigen bis zur Zeile mit dem Namen in der zweiten Zelle

```
  var name = tabelle.
    rows[tabellen_zeilen_index].cells[1].children[0].value
```

und den Namen zurückliefern:

```
  return name
}
```

5.36 optimierung_durchfuehren

Das Unterprogramm

```
function optimierung_durchfuehren() {
```

ist das Herzstück der Optimierung. Als erstes lesen wir im Unterprogramm beziehungen_aus_tabelle_lesen die von der Lehrkraft definierten Beziehungen aus der entsprechenden Tabelle auf der Optimierungsseite und sortieren sie in die globale Variable `beziehungen` ein:

```
  beziehungen_aus_tabelle_lesen()
```

Dann initialisieren wir das Anordnungsfeld `platz_mensch_index`, das angibt, auf welchem Platz momentan gerade welcher Mensch sitzt:

```
  var platz_mensch_index = []
```

[17] Natürlich hätten wir das ganze auch in einer einzigen Zeile schreiben können; diese wäre dann aber „etwas" unübersichtlich geworden ...

Eine drei im zweiten Element des Feldes bedeutet dabei beispielsweise, dass der dritte Mensch (also der Mensch mit dem Index drei) auf dem zweiten Platz (also dem Platz mit dem Index zwei) sitzt. Da es genauso viele Menschen wie Plätze gibt und wir definieren, dass anfänglich der erste Mensch auf dem ersten Platz, der zweite Mensch auf dem zweiten Platz, ... sitzt, füllen wir das Feld in einer Schleife über alle Menschen

```
  for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

mit so vielen natürlichen Zahlen, wie es Menschen gibt:

```
    platz_mensch_index[i_mensch] = i_mensch
  }
```

Durch den Aufruf des Unterprogrammes `festlegungen_aus_tabelle_lesen` lesen wir die von der Lehrkraft definierten Festlegungen aus der entsprechenden Tabelle auf der Optimierungsseite und sortieren die Informationen in die globalen Variablen `festlegungen`, `array_festlegungen` und `array_frei` ein:

```
  festlegungen_aus_tabelle_lesen()
```

Für die Berechnung der Sitzplatzabstände erzeugen wir mit dem Unterprogramm `obere_dreiecksmatrix_erzeugen` eine obere Dreiecksmatrix[18] mit so vielen Zeilen bzw. Spalten wie es Menschen gibt

```
  var array_entfernung = obere_dreiecksmatrix_erzeugen(n_menschen)
```

in die wir in einer Doppelschleife über alle[19] Zeilen und Spalten

```
  for (var i_zeile = 0; i_zeile < n_menschen; i_zeile++) {
    for (var i_spalte = i_zeile + 1;
      i_spalte < n_menschen; i_spalte++) {
```

die Abstände aller Menschen zueinander einsortieren:

```
      array_entfernung[i_zeile][i_spalte] =
        Math.abs(menschen[i_spalte].raum_zeilen_index -
        menschen[i_zeile].raum_zeilen_index) +
        Math.abs(menschen[i_spalte].raum_spalten_index -
        menschen[i_zeile].raum_spalten_index)
    }
  }
```

Dabei verwenden wir die Eigenschaften `raum_zeilen_index` und `raum_spalten_index` der Menschen, um die Betragssummennorm

$$|\Delta x| + |\Delta y|$$

[18] Abstände sind „gegenseitig": Platz A ist von B genau so weit entfernt wie B von A. Zur Abstandsspeicherung reicht also eine „halbe" Matrix.

[19] Natürlich füllen wir tatsächlich nur die nordöstlichen Elemente des Feldes, indem wir den Spaltenindex erst ab `i_zeile + 1` loslaufen lassen.

des Abstandsdifferenzvektors zweier Menschen zu berechnen. Man könnte an dieser Stelle ganze wissenschaftliche Arbeiten darüber schreiben, ob hier vielleicht die klassische euklidische Norm

$$\sqrt{(\Delta x)^2 + (\Delta y)^2}$$

oder eine Maximumsnorm

$$\max(\Delta x,\, \Delta y)$$

die diagonale Entfernungen genauso wie achsenparallele Entfernungen gewichtet, doch sinnvoller wären. In der Praxis unterscheiden sich die Optimierungsergebnisse nicht besonders stark. Den Autoren war es wichtig, dass es, wie in Abbildung 2.5a gefordert, möglich ist, Diagonalanordnungen gegenüber Situationen zu favorisieren, in denen sich die Schülerinnen und Schüler direkt gegenüber sitzen. Damit scheidet die Maximumsnorm aus. Und wenn man dann noch gerade in der Entwicklungsphase lieber Ganzzahlen statt der bei der euklidischen Norm entstehenden Fließkommazahlen überprüfen möchte, erscheint die Betragssummennorm ziemlich attraktiv.

Als weiteren Schritt der Initialisierung kopieren[20] wir das anfängliche Anordnungsfeld in die Variable `platz_mensch_index_min`, die während der Optimierung immer die aktuell beste Sitzanordnung repräsentiert:

```
var platz_mensch_index_min = platz_mensch_index.slice(0)
```

Durch einen erstmaligen Aufruf des Unterprogramms `kostenfunktion` berechnen wir die Kosten, die der Anfangssitzplan besitzt:

```
var kosten_min = kostenfunktion(
  array_entfernung, array_beziehungen, platz_mensch_index_min)
```

Die Variable `kosten_min` aktualisieren wir später während der Optimierung jedes Mal, wenn wir einen besseren Sitzplan gefunden haben.

Da wir in den Optimierungsschleifen immer wieder alle Vertauschungen zweier Menschen untersuchen wollen (eins mit zwei, eins mit drei, ...), erzeugen wir uns mit Hilfe des Unterprogrammes `zwei_aus_n` einmalig ein Feld, das alle[21] Zweierkombinationen (ohne

[20] Dabei verwenden wir den Befehl `slice`, der eine echte Kopie des Feldes erzeugt und nicht nur seine Adresse kopiert.

[21] Im Beispiel gehen wir davon aus, dass es keine Festlegungen gibt, dass also alle n Menschen frei verschiebbar sind und dass die Urne daher die Zahlen $1 \ldots n$ beinhaltet.

Wiederholung)

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & : \\ 1 & n \\ 2 & 3 \\ 2 & 4 \\ 2 & : \\ 2 & n \\ 3 & 4 \\ 3 & 5 \\ : & : \\ n-1 & n \end{bmatrix}$$

der frei beweglichen Menschenindizes beinhaltet:

```
var kombination = zwei_aus_n(array_frei)
```

Der nächste Schritt ist etwas schwerer zu verstehen: Wir möchten, dass bei jedem erstmaligen Aufruf der inneren Optimierungsschleife (nach jedem Weiterzählen der äußeren Schleife) der Sitzplan ohne Vertauschung in der inneren[22] Schleife untersucht wird. Dazu ergänzen wir als erstes Element des Feldes ein Dummy-Element, das gar keine Vertauschung durchführt (im Beispiel das Element [1 1]):

```
kombination.unshift([array_frei[0], array_frei[0]])
```

Die Anzahl der Kombinationen könnten wir natürlich auch berechnen; wir ermitteln sie aber (vermutlich) effizienter als Zeilenanzahl des (erweiterten) Feldes:

```
var n_kombination = kombination.length
```

Jetzt kann die doppelte Optimierungsschleife beginnen. Wir initialisieren den Zähler der äußeren Schleife

```
var i_1 = 0
```

und starten die äußere Schleife über alle Tauschkombinationen:

```
while (i_1 < n_kombination) {
```

Innerhalb der äußeren Schleife führen wir in jedem Schritt eine Tauschaktion zweier Menschen durch:

```
  var platz_mensch_index_1 =
    platz_mensch_index_min.elemente_vertauschen(
    kombination[i_1][0],
    kombination[i_1][1])
```

[22] Mit der Tatsache, dass damit auch in der äußeren Schleife einmalig ein unnötiger Aufruf der Kostenfunktion stattfindet, leben wir einfach ...

Bevor wir die innere Schleife starten, initialisieren wir ihren Zähler:

```
    var i_2 = 0
```

Auch die innere Schleife durchläuft alle Kombinationen:

```
    while (i_2 < n_kombination) {
```

Den Fall, dass wir in der inneren Schleife gerade die Vertauschung der äußeren Schleife wieder rückgängig machen würden, fangen wir ab:

```
      if (i_2 != i_1) {
```

Jetzt führen wir die innere Vertauschung durch

```
        var platz_mensch_index_2 =
          platz_mensch_index_1.elemente_vertauschen(
          kombination[i_2][0],
          kombination[i_2][1])
```

und berechnen die Kosten des aktuellen Sitzplans:

```
        var kosten_neu = kostenfunktion(
          array_entfernung,
          array_beziehungen,
          platz_mensch_index_2)
```

Wenn nun der aktuelle Plan besser[23] ist als der bislang beste

```
        if (kosten_neu < kosten_min) {
```

machen wir den aktuellen zum bislang besten Plan:

```
          kosten_min = kosten_neu
          platz_mensch_index_min = platz_mensch_index_2.slice(0)
```

Wenn wir einen neuen Favoriten gefunden haben, brechen wir die innere Schleife ab

```
          i_2 = n_kombination
```

und starten die äußere Schleife wieder neu:

```
          i_1 = -1
        }
      }
```

[23] Auch hier kann man lange darüber diskutieren, ob nicht ein „größer gleich" sicherer zum Ziel führen würde, da der Optimierer dann nicht so leicht in einem lokalen Nebenoptimum hängen bleiben würde. Das „größer gleich" würde aber die Wahrscheinlichkeit von Grenzschwingungen stark erhöhen, bei denen der Optimierer immer wieder zwischen zwei oder mehr gleich guten Sitzplänen hin- und herspringt. Wir bräuchten dann weitere Abbruchkriterien. Mit der „größer"-Bedingung ist sichergestellt, dass der Optimierer definitiv selbstständig abbricht, da er irgendwann zwar möglicherweise noch gleich gute Alternativpläne findet aber eben keine besseren mehr.

Da wir eine `while`-Schleife verwenden, müssen wir das Hochzählen der Schleifenzähler natürlich selbst erledigen:

```
      i_2 += 1
    }
    i_1 += 1
  }
```

Wenn wir jetzt den „optimalen" Plan gefunden haben, bilden wir ihn in der Raumtabelle grafisch ab. Dazu starten wir eine Schleife über alle Plätze

```
  for (var i_platz = 0; i_platz < n_menschen; i_platz++) {
```

und ermitteln als erstes, welcher Mensch auf dem aktuellen Platz des optimalen Plans sitzen soll:

```
    var i_mensch = platz_mensch_index_min[i_platz]
```

Als nächstes besorgen wir uns den Zeilen- und den Spaltenindex des aktuellen Platzes

```
    var i_zeile = menschen[i_platz].raum_zeilen_index
    var i_spalte = menschen[i_platz].raum_spalten_index
```

und lassen uns die entsprechende Zelle in der Raumtabelle zurückgeben:

```
    var zelle = table_raum.rows[i_zeile].cells[i_spalte]
```

Diese füllen wir dann mit den Eigenschaften des gerade ermittelten Menschen, also seiner ID

```
    zelle.setAttribute('data-id', menschen[i_mensch].data_id)
```

seiner CSS-Klasse

```
    zelle.setAttribute('class', menschen[i_mensch].class)
```

und seinem Namenskürzel:

```
    zelle.innerHTML = menschen[i_mensch].aufdruck
  }
```

Nachdem wir nun also die Raumtabelle im Hintergrund mit dem optimierten Plan aktualisiert haben, machen wir sie abschließend sichtbar:

```
  claroma_anzeigen()
}
```

5.37 beziehungen_aus_tabelle_lesen

Da es während der Optimierung viel effizienter ist, Informationen aus numerischen Feldern zu lesen als aus HTML-Tabellen, sortieren wir im Unterprogramm

```
function beziehungen_aus_tabelle_lesen() {
```

die von der Lehrkraft auf der Optimierungsseite definierten Beziehungen in numerische Felder ein. Dazu erzeugen wir eine obere Dreiecksmatrix[24], in die wir gleich die Beziehungen in numerischer Form einsortieren:

```
  array_beziehungen = obere_dreiecksmatrix_erzeugen(n_menschen)
```

Außerdem initialisieren wir ein serialisierbares Feld, in das wir die Beziehungen ebenfalls einsortieren werden und das wir in Abschnitt 5.18 zum Abspeichern der Beziehungen in einer Datei verwenden:

```
  beziehungen = []
```

Jetzt gehen wir in einer Schleife durch alle Zeilen der Beziehungstabelle:

```
    for (var i_zeile = 0;
      i_zeile < table_beziehungen.rows.length;
      i_zeile++) {
```

In der Schleife besorgen wir uns als erstes die aktuelle Tabellenzeile

```
    var zeile = table_beziehungen.rows[i_zeile]
```

und entnehmen dort die erste (linke) Auswahlliste:

```
    var mensch_1 = zeile.cells[0].children[0]
```

In der Liste lesen wir die ID des ausgewählten Menschen

```
    var mensch_1_id = mensch_1.value
```

und seinen Listenindex:

```
    var mensch_1_index = mensch_1.selectedIndex
```

In der zweiten Zelle der Zeile finden wir die Beziehung zwischen beiden Menschen als Zeichenkette, die wir in einen numerischen Wert umwandeln, um sie für die Kostenberechnungen verwenden zu können:

```
    var beziehung = parseInt(zeile.cells[1].children[0].value)
```

Die Informationen des zweiten (rechten) Menschen verarbeiten wir analog zum obigen Vorgehen beim ersten Menschen:

```
    var mensch_2 = zeile.cells[2].children[0]
    var mensch_2_id = mensch_2.value
    var mensch_2_index = mensch_2.selectedIndex
```

[24] Da alle Beziehungen auf Gegenseitigkeit beruhen, also ein Fall wie „Tim mag Maria, Maria mag Tim aber nicht" bei einer Sitzplatzoptimierung natürlich keine Berücksichtigung finden kann, reicht auch hier eine Dreiecksmatrix.

Nullbeziehungen, also Beziehungen die weder (positive) Sympathie noch (negative) Antipathie ausdrücken oder Beziehungen eines Menschen mit sich selbst, sind für die Optimierung irrelevant sind. Wir filtern sie daher an dieser Stelle[25] komplett heraus:

```
    if (beziehung != 0 && mensch_1_id != mensch_2_id) {
```

Wenn es sich aber um eine echte Beziehung handelt, fügen wir sie als neue Zeile im Beziehungsfeld ein:

```
      beziehungen.push([mensch_1_id, beziehung, mensch_2_id])
```

Um sicherzustellen, dass wir nur die nordöstliche Dreiecksmatrix[26] füllen, sorgen wir dafür, dass der Spaltenindex des Elementes, in das wir die Beziehung eintragen, immer[27] größer als sein Zeilenindex ist. Wenn also der Index des ersten Menschen größer als der des zweiten ist

```
      if (mensch_1_index > mensch_2_index) {
```

verwenden wir den größeren Index des ersten Menschen als Spalten- und den kleineren Index des zweiten Menschen als Zeilenindex:

```
        array_beziehungen[mensch_2_index][mensch_1_index] =
          beziehung
```

Wenn hingegen der Index des ersten Menschen kleiner als der des zweiten ist

```
      } else {
```

verwenden wir den kleineren Index des ersten Menschen als Zeilen- und den größeren Index des zweiten Menschen als Spaltenindex:

```
        array_beziehungen[mensch_1_index][mensch_2_index] =
          beziehung
      }
    }
  }
}
```

5.38 festlegungen_aus_tabelle_lesen

Das Umspeichern der Informationen der HTML-Tabelle der Festlegungen in numerische Felder in

[25] Wir löschen die Beziehungen nicht sofort aus der HTML-Tabelle; beim nächsten Aufruf der Optimierungsseite sind sie aber verschwunden.

[26] Im JavaScript-Sprachumfang gibt es natürlich nur rechteckige Felder.

[27] Da in der HTML-Liste alle Menschen sowohl auf der linken als auch auf der rechten Seine einer Beziehung auswählbar sind, könnte die Lehrkraft ja sowohl „Leonie mag Mia" als auch „Mia mag Leonie" auswählen. Wir wollen aber beide Informationen an die gleiche Stelle in der Beziehungsmatrix eintragen.

```
function festlegungen_aus_tabelle_lesen() {
```

verläuft ähnlich wie in beziehungen_aus_tabelle_lesen. Wir initialisieren die numerischen Felder

```
  var array_festlegungen = []
  festlegungen = []
```

und beginnen eine Schleife über alle Zeilen der Festlegungstabelle:

```
  for (var i_zeile = 0;
    i_zeile < table_festlegungen.rows.length; i_zeile++) {
```

Wir lesen die aktuelle Zeile

```
    var zeile = table_festlegungen.rows[i_zeile]
```

und darin die aktuelle Auswahlliste der Menschen

```
    var mensch = zeile.cells[0].children[0]
```

In der Liste finden wir die ID des ausgewählten Menschen

```
    var mensch_id = mensch.value
```

und seinen Listenindex

```
    var mensch_index = mensch.selectedIndex
```

Diese Informationen speichern wir anschließend in die entsprechenden numerischen Felder ab:

```
    festlegungen.push(mensch_id)
    array_festlegungen.push(mensch_index
  }
```

Für die Optimierung brauchen wir allerdings keine Liste der festgelegten Menschen, sondern ganz im Gegenteil die Liste der frei verschiebbaren Menschen. Dazu initialisieren wir ein weiteres Feld

```
  array_frei = []
```

und gehen durch alle im Plan gesetzten Menschen

```
  for (var i_mensch = 0; i_mensch < n_menschen; i_mensch++) {
```

Jetzt ermitteln wir die Indizes der im Festlegungsfeld gerade nicht auffindbaren Menschen, für die die Suchmethode `indexOf` eine `-1` zurückliefert

```
    if (array_festlegungen.indexOf(i_mensch) === -1) {
```

und speichern sie im Feld der optimierbaren Menschen ab:

```
      array_frei.push(i_mensch)
    }
  }
}
```

5.39 obere_dreiecksmatrix_erzeugen

Im Unterprogramm

```
function obere_dreiecksmatrix_erzeugen(n_zeilen) {
```

wollen wir eine „leere" obere Dreiecksmatrix erzeugen, in der alle nordöstlichen Elemente (rechts oberhalb der Hauptdiagonalen) eines Feldes mit einer numerischen `0` und die übrigen Elemente mit dem Literal `null`, das das Fehlen eines Elementes symbolisiert, gefüllt sind:

$$\begin{bmatrix} \texttt{null} & 0 & 0 & 0 \\ \texttt{null} & \texttt{null} & 0 & 0 \\ \texttt{null} & \texttt{null} & \texttt{null} & 0 \\ \texttt{null} & \texttt{null} & \texttt{null} & \texttt{null} \end{bmatrix}$$

Wir initialisieren das Feld

```
  var feld = []
```

und beginnen eine Schleife über alle[28] Zeilen:

```
  for (var i_zeile = 0; i_zeile < n_zeilen; i_zeile++) {
```

Für jede Zeile initialisieren wir ein weiteres Feld

```
    var feld_zeile = []
```

und durchlaufen alle Spalten, deren Index kleiner ist als der (um eins vergrößerte)[29] aktuelle Zeilenindex:

```
    for (var i_spalte = 0; i_spalte < i_zeile + 1; i_spalte++) {
```

Diese südwestlichen Elemente (einschließlich der Hauptdiagonalen) besetzen wir mit dem Literal `null`, das daran erinnert, dass dieser Teil der Matrix nicht verwendet wird:

```
      feld_zeile[i_spalte] = null
    }
```

Alle Elemente (rechts) oberhalb der Hauptdiagonalen

```
    for (var i_spalte = i_zeile + 1; i_spalte < n_zeilen;
      i_spalte++) {
```

füllen wir mit numerischen Nullen:

```
      feld_zeile[i_spalte] = 0
    }
```

[28] Die Anzahl der Zeilen der quadratischen Matrix erhalten wir dabei über die Parameterliste.
[29] Wir möchten auch die Hauptdiagonale selbst mit `null` vorbesetzen.

Die fertige Zeile hängen wir dann an die wachsende Matrix an:

```
        feld.push(feld_zeile)
    }
```

Wenn die Matrix schließlich komplett gefüllt ist, geben wir sie an das aufrufende Programm zurück:

```
    return feld
}
```

5.40 elemente_vertauschen

Während der Optimierung möchten wir häufig zwei Elemente eines Feldes miteinander vertauschen. JavaScript kennt nativ keine Möglichkeit dazu, macht es uns aber sehr einfach, eine neue[30] entsprechende Methode für das vorhandene Array.prototype-Objekt zu schreiben, die dann für jedes Feld zur Verfügung steht. In der Definition der neuen Methode

```
Array.prototype.elemente_vertauschen = function (i, j) {
```

die das i-te Element des Feldes mit seinem j-ten Element vertauschen soll, puffern wir als erstes das Feld selbst als echte Kopie[31]

```
    var that = this.slice(0)
```

und ersetzen dann in der Kopie das i-te Element durch das j-te Element des Originals

```
    that[i] = this[j]
```

und das j-te Element der Kopie durch das i-te Element des Originals:

```
    that[j] = this[i]
```

Schließlich geben wir die modifizierte Kopie zurück:

```
    return that
}
```

5.41 kostenfunktion

Das Unterprogramm

[30] In einem größeren Projekt, an dem möglicherweise sogar mehrere Programmierer unabhängig voneinander arbeiten, ist diese quick-and-dirty-Erweiterung des Standard-Array-Prototypen vermutlich keine besonders elegante Lösung.

[31] Auch hier gibt es sicherlich weniger speicherplatzintensive Methoden.

```
function kostenfunktion(
  array_entfernung,
  array_beziehungen,
  platz_mensch_index) {
```

wird in jedem Durchlauf der Optimierungsschleife in optimierung_durchfuehren aufgerufen und bekommt über seine Parameterliste die Entfernungen der im Plan gesetzten Menschen zueinander (array_entfernung), die Wünsche der Lehrkraft hinsichtlich der Beziehungen der Menschen (array_beziehungen) und den aktuellen Sitzplan (platz_mensch_index) übergeben. Es berechnet daraus, wie gut der aktuelle Sitzplan die Wünsche der Lehrkraft erfüllt; je niedriger der Wert der zurückgelieferten Variable kosten ist, desto besser ist der aktuelle Plan.

Im Unterprogramm bestimmen wir die Anzahl der Sitzplätze

```
  var n_platz = platz_mensch_index.length
```

und initialisieren die Rückgabevariable:

```
  var kosten = 0
```

Bei der Berechnung der Kosten eines Sitzplans berechnen wir für jede Kombination zweier Plätze die jeweiligen Einzelkosten und addieren diese dann zu den Gesamtkosten auf. Die Einzelkosten eines Platzpaares berechnen wir aus dem Produkt der Entfernung der Plätze zueinander mit der Wunschstärke, dass diese Plätze möglichst nahe beieinander liegen sollten. Ein Platzpaar verursacht also genau dann besonders große Kosten, wenn die beiden Plätze weit auseinander liegen, die Lehrkraft aber angegeben hat, dass die Menschen, die auf den Plätzen sitzen, eigentlich möglichst nahe beisammen sitzen sollen. Entsprechend produzieren Plätze, von denen die Lehrkraft gefordert hat, dass sie möglichst weit voneinander entfernt liegen sollen, sogar negative Einzelkosten. Sie reduzieren damit die Gesamtkosten genau dann, wenn die beiden Plätze tatsächlich weit voneinander entfernt liegen. Die Gesamtkosten können damit durchaus negativ werden, wenn die Lehrkraft überwiegend Antipathien (mit negativem Vorzeichen in der Beziehungsmatrix) definiert hat; auch dann ist ein Plan besser, wenn sein Wert (nicht sein Betrag) möglichst klein ist ($-42 < -41$).

Es gibt jetzt ein kleines Problem: Die Beziehungen, also die Wünsche der Lehrkraft bezüglich der Nähe oder Ferne zweier Menschen sind eben tatsächlich als Beziehungen zwischen Menschen festgelegt. Für die Kostenberechnung benötigen wir aber Wünsche hinsichtlich der Plätze, da wir ja auch die Entfernungen der Plätze (und nicht der Menschen) einmalig berechnet haben. Bei jedem Plan ändern sich aber natürlich die Plätze, auf denen die jeweiligen Menschen sitzen. Wir müssen daher die Beziehungen zwischen den Menschen unter Verwendung der aktuellen Sitzplatzbelegung (platz_mensch_index) auf Beziehungen zwischen den Plätzen umrechnen. Dazu untersuchen wir in einer Doppelschleife die Kombinationen aller Plätze miteinander:

```
  for (var i_platz_1 = 0; i_platz_1 < n_platz; i_platz_1++) {
    for (var i_platz_2 = i_platz_1 + 1; i_platz_2 < n_platz;
      i_platz_2++) {
```

In der Schleife treffen wir als erstes eine Fallunterscheidung: Wir können zwar sicher sein, dass durch die Wahl des Startwertes der inneren Schleife der Platzindex `i_platz_2` immer größer als `i_platz_1` ist; über die zugehörigen Menschenindizes, die sich durch Einsetzen der Platzindizes in den aktuellen Sitzplan (`platz_mensch_index`) ergeben, können wir diese Aussage aber nicht treffen. Da wir aber die Menschenindizes als Zeilen- und Spaltenindizes der Beziehungsmatrix verwenden werden, die wir ja nur als obere Dreiecksmatrix befüllt haben, müssen wir sicherstellen, dass auch der Menschenindex, den wir als Spaltenindex in die Dreiecksmatrix verwenden, immer größer ist als der Menschenindex, den wir als Zeilenindex verwenden. Wir vergleichen also die beiden Menschenindizes miteinander: Wenn der Menschenindex des ersten Platzes größer als der Menschenindex des zweiten Platzes ist

```
      if (
        platz_mensch_index[i_platz_1] >
        platz_mensch_index[i_platz_2]) {
```

dann verwenden wir den größeren Menschenindex (`platz_mensch_index[i_platz_1]`) des ersten Platzes (`i_platz_1`) als Zeilenindex in die (Menschen-)Beziehungsmatrix (`array_beziehungen`), um den (Platz-)Beziehungswert (`beziehung_platz`) der beiden gerade betrachteten Plätze zueinander zu berechnen:

```
        var beziehung_platz = array_beziehungen
          [platz_mensch_index[i_platz_2]]
          [platz_mensch_index[i_platz_1]]
```

Wenn hingegen der zweite Platz einen größeren Menschenindex besitzt

```
      } else {
```

verwenden wir diesen (`platz_mensch_index[i_platz_2]`) als Spaltenindex in die Beziehungsmatrix:

```
        var beziehung_platz = array_beziehungen
          [platz_mensch_index[i_platz_1]]
          [platz_mensch_index[i_platz_2]]
      }
```

Bei jedem Schleifendurchlauf berechnen wir nun die neuen Einzelkosten als Produkt aus dem Beziehungswert (`beziehung_platz`) der aktuellen Plätze zueinander und ihrer Entfernung (`array_entfernung[i_platz_1][i_platz_2]`) voneinander und addieren sie zur Gesamtkostensumme hinzu:

```
      kosten +=
        beziehung_platz * array_entfernung[i_platz_1][i_platz_2]
    }
  }
```

Nachdem wir dann alle Einzelkosten aufsummiert haben, geben wir den Gesamtkostenwert des aktuellen Sitzplans zurück:

```
  return kosten
}
```

5.42 zwei_aus_n

Im Unterprogramm

```
function zwei_aus_n(urne) {
```

erzeugen wir ein $m \times 2$-Feld, in dem wir alle Zweierkombinationen der im eindimensionalen Feld `urne` vorhandenen Elemente ablegen. Wenn in `urne` also beispielsweise die Zahlen

$$\texttt{urne} = \begin{bmatrix} 2 \\ 5 \\ 6 \\ 9 \end{bmatrix}$$

vorhanden sind, so beinhaltet das Ergebnisfeld `erg` die folgenden Kombinationen:

$$\texttt{erg} = \begin{bmatrix} 2 & 5 \\ 2 & 6 \\ 2 & 9 \\ 5 & 6 \\ 5 & 9 \\ 6 & 9 \end{bmatrix}$$

Die Anzahl m der Zweierkombinationen ergibt sich dabei aus der Anzahl n der in der Urne vorhandenen Elemente natürlich über den Binomialkoeffizienten zu:

$$m = \binom{n}{2} = \frac{n \cdot (n-1)}{2}$$

Als erstes bestimmen wir im Unterprogramm die Anzahl der Zahlen in der Urne

```
  var n_urne = urne.length
```

und initialisieren das Ergebnisfeld:

```
  var erg = []
```

Dann beginnen wir die klassische Doppelschleife, in der die äußere Schleife über alle (bis auf das letzte) Elemente läuft und die innere Schleife nur die Elemente durchläuft, die in der äußeren Schleife noch nicht abgearbeitet wurden:

```
  for (var i_1 = 0; i_1 < n_urne - 1; i_1++) {
    for (var i_2 = i_1 + 1; i_2 < n_urne; i_2++) {
```

Im Inneren der Schleife fassen wir dann die beiden aktuell betrachteten Elemente in einer Zeile zusammen und hängen sie an an das Ergebnisfeld an:

```
      erg.push([urne[i_1], urne[i_2]])
    }
  }
```

Schließlich geben wir das gefüllte Ergebnisfeld zurück:

```
  return erg
}
```

6 claroma.css

In einer CSS-Datei beschreiben wir die Formatierung von HTML-Elementen; wir bestimmen hier also, wie unsere Seite aussieht. Grundsätzlich verwenden wir eine kleine[1] serifenlose Schrift und einen Mauszeiger, der deutlich macht, dass wir in einer Tabelle arbeiten:

```
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  font-size: small;
  cursor: cell;
}
```

Eine Überschrift erster Ordnung (die wir nur auf der Hauptseite verwenden) möchten wir groß, blau und ohne Randabstände:

```
h1 {
  font-size: x-large;
  color: cornflowerblue;
  margin-top: 0;
  margin-bottom: 0;
}
```

Überschriften zweiter Ordnung sollen zusätzlich noch einen oberen Randabstand bekommen:

```
h2 {
  font-size: x-large;
  color: cornflowerblue;
  margin-top: 20px;
  margin-bottom: 0;
}
```

Der Rand zwischen allen Tabellenzellen (außer in der Raumtabelle) soll 5 Pixel betragen:

```
table {
  border-spacing: 5px;
}
```

Alle Schaltflächen sollen eine einheitliche Breite von 80 Pixeln und eine Höhe von 5 Pixeln besitzen:

[1] Wir möchten gerne, dass die Standardseite (20×20 Felder) auch auf einem kleinen Bildschirm mit einer Auflösung von 1024×768 Pixeln noch ohne Blättern dargestellt werden kann.

```
button {
  width: 80px;
  margin: 5px;
}
```

In der Raumtabelle sollen die Rahmen benachbarter Zellen zusammenfallen:

```
#table_raum {
  border-collapse: collapse;
}
```

Die Namensfelder sollen eine Breite von 100 Pixeln besitzen:

```
.mensch_name {
    width: 100px;
}
```

Das Feld für den Klassennamen passen wir händisch so an, dass es mittig unter dem Titel steht:

```
.klassen_name {
  width: 87px;
  margin-top: 0;
  margin-bottom: 10px;
  margin-left: 2px;
}
```

Die Standardraumtabellenzelle soll eine Größe von 26×26 Pixeln, einen schmalen hellgrauen Rand, keinen zusätzlichen Randabstand und zentrierten Text besitzen:

```
.feld {
  width: 26px;
  height: 26px;
  border: 1px lightgray solid;
  padding: 0;
  text-align: center;
}
```

Das Symbol eines Menschen hat die gleiche Größe, besteht aber aus einem Kreis:

```
.mensch {
  width: 26px;
  height: 26px;
  border-radius: 13px;
}
```

Da wir die drei in `seite_claroma` beschriebenen (unterschiedlich langen) Untertabellen an ihren Oberkanten ausrichten möchten, definieren wir dafür eine weitere CSS-Klasse:

```
.oben {
  vertical-align: top;
}
```

Jetzt müssen wir nur noch für die unterschiedlichen Objekte jeweils eine eigene Klasse festlegen, die wir ja im Programm als Erkennungsmerkmal verwenden. Natürlich bekommt dabei jedes Objekt seine eigene Farbe:

```
.rosa {
  background-color: lightpink;
}

.blau {
  background-color: lightskyblue;
}

.leer {
  background-color: white;
}

.wand {
  background-color: lightgray;
}

.fenster {
  background-color: azure;
}

.tuer {
  background-color: lightsteelblue;
}

.tafel {
  background-color: darkseagreen;
}

.tisch {
  background-color: burlywood;
}

.schrank {
  background-color: palegoldenrod;
}

.ablage {
  background-color: peachpuff;
}
```

A claroma.json

Im Folgenden ist exemplarisch[1] die JSON-Datei dargestellt, die Claroma erzeugt, wenn wir sofort nach der Initialisierung (Abbildung 5.1) des Programmes die Speicherschaltfläche drücken:


```
{
  "version": 2,
  "klassen_name": "",
  "datum": "2015_10_19_18_4_38",
  "n_rosa": 2,
  "array_rosa": [
    {
      "id": "m4395349351333887",
      "class": "feld mensch rosa",
      "aufdruck": "",
      "name": ""
    },
    {
      "id": "m7485032316796927",
      "class": "feld mensch rosa",
      "aufdruck": "",
      "name": ""
    }
  ],
  "n_blau": 2,
  "array_blau": [
    {
      "id": "m3581373946265599",
      "class": "feld mensch blau",
      "aufdruck": "",
      "name": ""
    },
    {
      "id": "m8331121439801343",
      "class": "feld mensch blau",
      "aufdruck": "",
      "name": ""
```


[1] Das Datum entspricht dabei natürlich dem aktuellen Datum und die ID-Ziffern sind zufällig. Die dreifachen Doppelpunkt bedeuten, dass dort redundante Zeilen ausgelassen wurden.

```
    }
  ],
  "n_raum_zeilen": "20",
  "n_raum_spalten": "20",
  "array_raum": [
    [
      {
        "data_id": "wand",
        "class": "feld wand"
      },
      {
        "data_id": "wand",
        "class": "feld wand"
      },
:
:
:
      {
        "data_id": "wand",
        "class": "feld wand"
      }
    ],
    [
      {
        "data_id": "wand",
        "class": "feld wand"
      },
      {
        "data_id": "leer",
        "class": "feld"
      },
:
:
:
      {
        "data_id": "wand",
        "class": "feld wand"
      }
    ]
  ],
  "beziehungen": [],
  "festlegungen": []
}
```

Literaturverzeichnis

[1] Wikipedia. (2015) HTML5. [Online]. Available: http://de.wikipedia.org/wiki/HTML5

[2] ——. (2015) Cascading Style Sheets. [Online]. Available: http://de.wikipedia.org/wiki/Cascading_Style_Sheets

[3] ——. (2015) JavaScript. [Online]. Available: http://de.wikipedia.org/wiki/JavaScript

[4] ——. (2015) UTF-8. [Online]. Available: http://de.wikipedia.org/wiki/UTF-8

[5] Eli Grey. (2015) Filesaver.js. [Online]. Available: https://github.com/eligrey/FileSaver.js

[6] Wikipedia. (2015) jQuery Mobile. [Online]. Available: https://www.wikiwand.com/en/JQuery_Mobile

[7] ——. (2015) Insertationsort. [Online]. Available: https://de.wikipedia.org/wiki/Insertionsort

[8] ——. (2015) HSV-Farbraum. [Online]. Available: https://de.wikipedia.org/wiki/HSV-Farbraum

[9] ——. (2015) Data-URL. [Online]. Available: https://de.wikipedia.org/wiki/Data-URL

[10] The LyX Team. (2015) LyX - The Document Processor. [Online]. Available: http://www.lyx.org/

[11] Mozilla Developer Network. (2015) Number.MAX_SAFE_INTEGER. [Online]. Available: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER