Magischer Würfel

Warum ein Ahnungsloser ein Lösungsprogramm für den magischen Würfel schreibt

Dieser Bericht handelt davon, warum ich, der noch nie erfolgreich einen magischen Würfel gelöst hat, ein funktionierendes Programm zur Lösung zustande bringen will. Und warum ich damit meine kostbare Lebenszeit verschwendet.

Es wird beschrieben, in welchen Schritten diese Software entsteht und welche Überlegungen und Recherchen zu diesem Ergebnis führen. Am Ende steht ein fast sinnloses Stück Code, dessen Aufgabe seit Jahrzehnten schon von anderen Implementierungen erledigt wird. Das ist also in etwa so, als wenn jemand ein Lego-Modell baut, was tausende Andere vor ihm auch schon mal gebaut haben. Also etwas, was jeden Tag ständig passiert. Ich mach das auch. Das Leben ist schon verrückt.

Subtiles Mobbing als Motivation

Vor den Eingängen unserer Büros befindet sich ein breiter, heller Gang mit runden Tischen und einem grandiosen Fensterblick auf die grau betonierte Fläche unseres für Führungskräfte reservierten Parkplatzes.

Nicht, dass diese Ecke ein besonders attraktiver Ort für unsere Mittagspause ist, aber es ist der Ort, den man mit dem geringsten physischen Aufwand vom Büro aus erreichen kann. Also saßen an diesem Tage mal wieder einige Kollegen zusammen. Zwei davon spielten mit Zauberwürfeln und

zeigten einem Kollegen und mir, was man mit Fleiß, Ausdauer und Härte gegen sich selbst alles erreichen kann: Unser jüngster Neuzugang drehte den Würfel einhändig mit seiner linken Hand und war damit um eine Größenordnung schneller als der zweite Kollege, der auch nicht gerade auf den Kopf gefallen ist. Er begründete das damit, dass er ein Handicap bräuchte, weil er sonst noch viel schneller wäre.

Dann erklärte er mir kurz etwas in einer mir unbekannten Sprache und gab mir seinen verdrehten Würfel. Dass ich beide (ihn und den Würfel) lediglich fragend anschaute entsprach offensichtlich nicht seiner Erwartungshaltung. Als er dann noch berichtete, dass er ein Lösungsprogramm in C++ geschrieben hat, fühlte ich mich wie jemand, den man gerade mit einer Dampfwalze in den Boden integriert hat.

OK. Das geht gar nicht. *Ich* bin der Architekt in unserer Abteilung. Vier Wochen Urlaub und der Ehrgeiz, die Java-Kenntnisse wieder aufzufrischen sind die Zutaten, um die Schmach wenigstens ansatzweise zu tilgen. Nehmt das!

Die Vorbereitung

In unserem Unternehmen wird der Begriff „Voruntersuchung" gern verwendet. Im Wikipedia, des offiziellen Gehirnextenders der Digital Natives, ist „Voruntersuchung" als Teil eines Vorgehensmodells beim Projektmanagement folgendermaßen beschrieben:

Die Voruntersuchung (auch Pilotstudie genannt, engl. feasibility study) ist ein Problemlösungszyklus im Kleinen. In dieser Phase gilt es, das Problem genau zu definieren (Was soll geändert werden? Welches sind die Ziele?), die Chancen und Risiken eines Veränderungsprozesses einzuschätzen (Cui bono? Mit welcher Unterstützung und welchem Widerstand ist zu rechnen?) und die zur Verfügung stehenden Ressourcen zu überprüfen (Ist genügend Managementkapazität und Budget vorhanden?) In der Voruntersuchung sollte auch eine grobe Idee einer möglichen Lösung entwickelt werden. Am Ende der Voruntersuchung steht die Entscheidung, den Prozess fortzusetzen oder abzubrechen. Go/No Go-Entscheidung).[1]

Das Ergebnis in meinem Fall: Ja, mach ich!

Im Internetz gibt's alle möglichen Anleitungen, wie ein verdrehter Würfel wieder in die Spur gebracht wird[2]. Das in ein Stück Software zu gießen, sollte ja für mich ein No-Brainer sein. Die obligatorischen Ergebnisse einer Voruntersuchung sind damit erfolgreich erreicht: Die Aufgabe zu unter- und sich selbst zu überschätzen. Ich sollte ins Management wechseln.

---

[1] https://de.wikipedia.org/wiki/Vorgehensmodell#Voruntersuchung
[2] Das hier war meine Bibel: https://speedcube.de

Hard- und Software

Da auch ich ein Handicap brauche will ich mich mit einer Sprache zu beschäftigen, mit der ich Anfang dieses Jahrtausends das letzte Mal zu tun hatte: **Java**. Das bedeutet zusätzlich auch noch mit einer ungewohnten IDE zu arbeiten: **IntelliJ IDEA**. Auch das ist mir noch zu einfach, also wechsle ich während der Arbeit zwischen **PC** und **Mac** hin und her. Ja, die Art Mac, bei der die für Programmierer lebenswichtigen Zeichen [,], |, { und } nicht nur woanders auf der Tastatur liegen, sondern bei Apple in weißer Farbe auf weiße Tasten gedruckt sind. Das trainiert das Gedächtnis, was in meinem Alter mittlerweile eine gute Idee ist. Den Backslash habe ich übrigens immer noch nicht gefunden[3].

Apple hat schon immer viel Wert auf Design gelegt. Ich denke, dass irgendwann aus ästhetischen Gründen ganz auf jeden Aufdruck verzichtet wird. Zen-Buddhisten werden dann lächelnd auf ihre komplett weißen Tasten starren und beim Meditieren über den Sinn von Werden, Sein und Vergehen nicht mehr abgelenkt.

Mein Plan sieht aber anders aus. Warum ich nicht einfach eine PC-Tastatur anschließe? Apple-User glauben, dass der iMac einen tödlichen Stromstoß durch das USB-Kabel schickt, sobald er PC-Peripherie entdeckt. Ich glaube nicht daran, bin aber in diesem Gebiet eher Agnostiker als Atheist, d.h. dass bei diesem möglichen Schadensbild die Nutzung von Originalhardware pragmatisch gesehen das kleinere Übel darstellt und es das Risiko nicht wert ist.

Sogar ein Versionsverwaltungssystem setze ich für dieses kleine Hobbyprojekt ein. Git ist einfach, zuverlässig und in IntelliJ Out-Of-The-Box verwendbar. Wenn man ohnehin nicht genau weiß was man tut, ist eine Versionshistorie nichts Falsches. Also mehrmals am Tag ein „Commit" erspart Kummer und Sorgen.

Die Designziele

Jeder gute Architekt sollte die Designziele seiner Software vorher festlegen. Diese Ziele werden auch gerne „Qualitäten" genannt. Ist so ein Architektending. Normale Programmierer schreiben ihren Code halt und gucken, dass er läuft. Geht auch irgendwie, funktioniert aber nur gut bei Einzelkämpfern und anderen Superhelden.

Lasst uns mal schauen, was mir so eingefallen ist:

1. *Der Code sollte portabel sein*
   Also nicht speziell für ein Betriebssystem oder eine Plattform. Wäre nett, wenn man das

[3] Nach intensiver Recherche war ich dann doch noch erfolgreich: fn + Umschalt + 7

Ganze z.B. auch in einem Microservice (was immer das auch genau ist) einsetzen könnte. Vielleicht will man ja mal einen Lego-Roboter damit steuern.

2. *Der Code soll langlebig sein*
   Schick ist heutzutage, schnell mal ein aktuelles Framework einzusetzen, bei dem eine neue Version schneller entsteht, als man „object relational mapping" sagen kann. Leider jagt man anschließend allzu oft diesen neuen Versionen hinterher und muss seinen Code ständig anpassen, um nicht aus der Wartung zu fallen oder Sicherheitslöcher zu haben, die größer als das Saarland sind. Das will ich nicht. Deswegen: Nur das verwenden, was Java mitliefert. So hält's länger.
3. *Der Code soll komplett gemäß Clean-Code-Regeln[4] geschrieben sein*
   Ist natürlich nur Spaß. Das geht nicht. Aber ich wollte es zumindest versuchen.

Der dritte Punkt füllt bereits ganze Bücher und Websites, so dass man eigentlich schon allein dadurch ausgelastet ist. Da fällt z.B. auch eine möglichst vollständige Testabdeckung durch Unit- und Integrationstests drunter. Echt lästig. Aber was tut man nicht alles für Geld.

Ich habe schon oft erlebt, dass man Lösungen einsetzt, für die man kein passendes Problem hat. Unabhängig von Ausbildung oder Stellung in der Hierarchie ist im Grunde fast Jeder anfällig für verschieden schwere Grade der Technologitis. „Wir müssen ins Internet", „Wir brauchen jetzt Microservices", „Wir gehen in die Cloud" – die Lösung existiert immer vor dem Problem. Das finde ich großartig. So mache ich es auch. Kein Mensch braucht ein weiteres Programm zur Lösung des Zauberwürfels[5], aber es ist cool und man fühlt sich gleich um Jahre jünger.

Was geht in meinem Kopf vor?

Eigentlich will das keiner wissen. Aus ästhetischen Gründen filtere ich das deswegen hier und reduziere es auf die Gedanken, die sich auf das Vorgehen beim Softwareentwickeln beziehen. Ist schon schlimm genug.

Digitalisierung der Welt

Beim Magischen Würfel haben wir ein reales Objekt aus der analogen Welt, welches wir in einer digitalen Software verwenden wollen. Im Grunde kann man dies als die grundlegende Herausforderung aller Softwareentwickler ansehen: Etwas aus der echten Welt in einer digitalen Form abbilden, um es mit entsprechenden Algorithmen zu bearbeiten. Ob es sich um Kontenblätter,

[4] https://clean-code-developer.de
[5] Allein auf GitHub gibt es 161 Repositories mit dem Namen CubeSolver...

Steuererklärungen, Bibliotheken, Autorennstrecken oder unser Universum handelt: Es muss erst in digitale Entsprechungen übersetzt werden, damit ein Computer damit etwas anfangen kann.

Deswegen gibt es sogar so tolle Ideen, wie auf einem PC-Drucker per Fakturierungssoftware erstellte Papierdokumente auf einen Scanner zu legen und per Software in digitalen Text umzuwandeln, um per künstlicher Intelligenz darin steuerrelevante Informationen zu erkennen, die man dann in einer PC-Software verbuchen kann, um es am Ende auf Papier auszudrucken. Das nenne ich mal effiziente CO^2-Erzeugung. In etwa so, wie einen Plastikwürfel in Software zu übersetzen und ihn von einem PC lösen zu lassen. Also das, was ich jetzt vorhabe.

In der objektorientierten Welt hat man dieses Vorgehen quasi in die Sprachen eingebaut. Ein Würfel ist ein Objekt, also mache ich eine Klasse „Würfel" mit den Eigenschaften eines Würfels in digitaler Form. Wenn ich davon eine Variable (Instanz) erzeuge, habe ich die digitale Entsprechung des Würfels. Wir erschaffen sozusagen für den ganzen Krempel eine eigene Matrix[6], ein Modell der analogen Welt. Weil ich nachmittags noch was anderes vorhabe, beschränke ich mich dabei allerdings auf den Teil des Universums, der was mit einem Magischen Würfel zu tun hat.

Ein Würfel in Zahlen

Ich muss also digitale Modelle der Dinge erzeugen, die für die Lösung des Problems notwendig sind. Daher kommt wohl auch der Ausdruck „Modellieren", der bei der Softwareentwicklung in der Entwurfsphase verwendet wird. Der Würfel erscheint zwar auf den ersten Blick ein einfaches Ding, aber es ist trotzdem eine gute Idee, ihn in weitere Einzelteile zu zerlegen. In der analogen Welt ist es schwierig, ihn danach wieder in einen brauchbaren Zustand zu versetzen, in der digitalen ist es umgekehrt.

Irgendwann brauchen wir außerdem noch Objekte, die den Würfel drehen oder analysieren und Lösungen finden. Und schon wird es unangenehm komplex. Für mich hat es sich bisher immer bewährt, dabei vom Großen ins Kleinere zu denken. ...

Wo findet man das alles?

Die Ergebnisse, von denen die folgenden Abschnitte handeln, habe ich zur allgemeinen Belustigung auf GitHub gestellt:

https://github.com/proggi64/CubeSolver

[6] Das ist eine Anspielung auf den Kinofilm „Matrix". Für meine Generation ist das ein Klassiker. Für die YouTube-Generation muss das aber erklärt werden. Schaut mal im Wikipedia nach...

Die Klassen des Würfelmodells

Meine erste spontane Idee ist, dass ich wohl zunächst ein Modell eines Würfels brauche. Da es inzwischen auch Würfel mit mehr als 3x3 Elementen pro Seite gibt, wäre es schick, das Modell entsprechend flexibel zu gestalten.

Cube

Der Name der Klasse „**Cube**", der das Würfelmodell abbildet, erscheint mir naheliegend. Der Würfel besteht aus sechs Flächen (Array der Klasse **CubeFace**), die wiederum aus einzelnen Farbelementen bestehen (Enumeration **CubeColor**).

Also:

```
public class Cube {
   private final int _dimension;
   private final CubeFace[] _faces = new CubeFace[6];
```

Ich war so frei, die Anzahl der Würfelseiten auf fest codierte sechs festzulegen. Mathematiker und Astrophysiker mögen mir das verzeihen, da es in deren Weltbild eine unzulässige Vereinfachung sein dürfte, von einer dreidimensionalen Welt auszugehen. Da ich aber schon genug mit drei Dimensionen zu kämpfen habe, müssen wir da jetzt einfach mal alle durch.

Die Membervariable **_dimension** gibt die Anzahl der Elemente einer Fläche in X- und Y-Richtung an, also beim Standardwürfel je drei. Dies wird auch vom Standardkonstruktor vorgegeben. Die Verwegenen unter uns dürfen auch den Konstruktor verwenden, der die freie Angabe dieses Wertes zulässt. Diese Variante ist eine Folge der chronischen Persönlichkeitsstörung, die fast alle Programmierer haben: Das „Man-könnte-es-ja-mal-brauchen"-Syndrom. Auch YAGNI[7] hilft dagegen bisher kaum. Ich bin ein hoffnungsloser Fall. Außerdem habe ich erwogen, dass ich es zunächst mit dem Wert 1 probieren werde. Das vereinfacht die Sache zwar ungemein, nimmt ihr aber auch ein Stück weit seinen Reiz.

CubeColor

CubeColor ist lediglich eine Enumeration. Jede der Flächen einer Seite hat einen dieser Werte:

[7] You Ain't Gonna Need It – Du wirst es nicht brauchen – Eines der Clean Code Prinzipien.

```
public enum CubeColor {
    White,
    Orange,
    Green,
    Red,
    Blue,
    Yellow
}
```

Ich habe mich bei den Originalfarben des Rubik's Cube bedient. Ich hoffe, ich werde dafür nicht verklagt. Ich weiß nicht, wie weit in den deutschen Gefängnissen inzwischen der Breitbandausbau fortgeschritten ist, aber das Risiko gehe ich trotzdem ein. Der Würfel wird schließlich nicht von Apple verkauft, das gibt mir Hoffnung, dass es ohne Rechtstreit geht.

Die numerischen Werte der Enumerationswerte entsprechen den Indexwerten der Würfelseiten mit den entsprechenden Farben bei der Standardausrichtung. Dazu gibt's weiter unten mehr.

CubeFace

Die Klasse **CubeFace** ist ein einfacher Datencontainer für **CubeColor**-Werte. Jede Würfelseite ist eine Matrix aus n x n Farbfeldern, im Standardfall also 3 x 3. Für die intellektuellen Superhelden unter uns können es auch mehr sein:

```
public class CubeFace {
    private final int _dimension;
    private final CubeColor[][] _fields;
```

Um allen Elementen einer Seite mit einem Rutsch eine einheitliche Farbe geben zu können, bietet **CubeFace** die Methode **setFaceColor** an. Die setzt alle Elemente von **_fields** auf dieselbe Farbe.

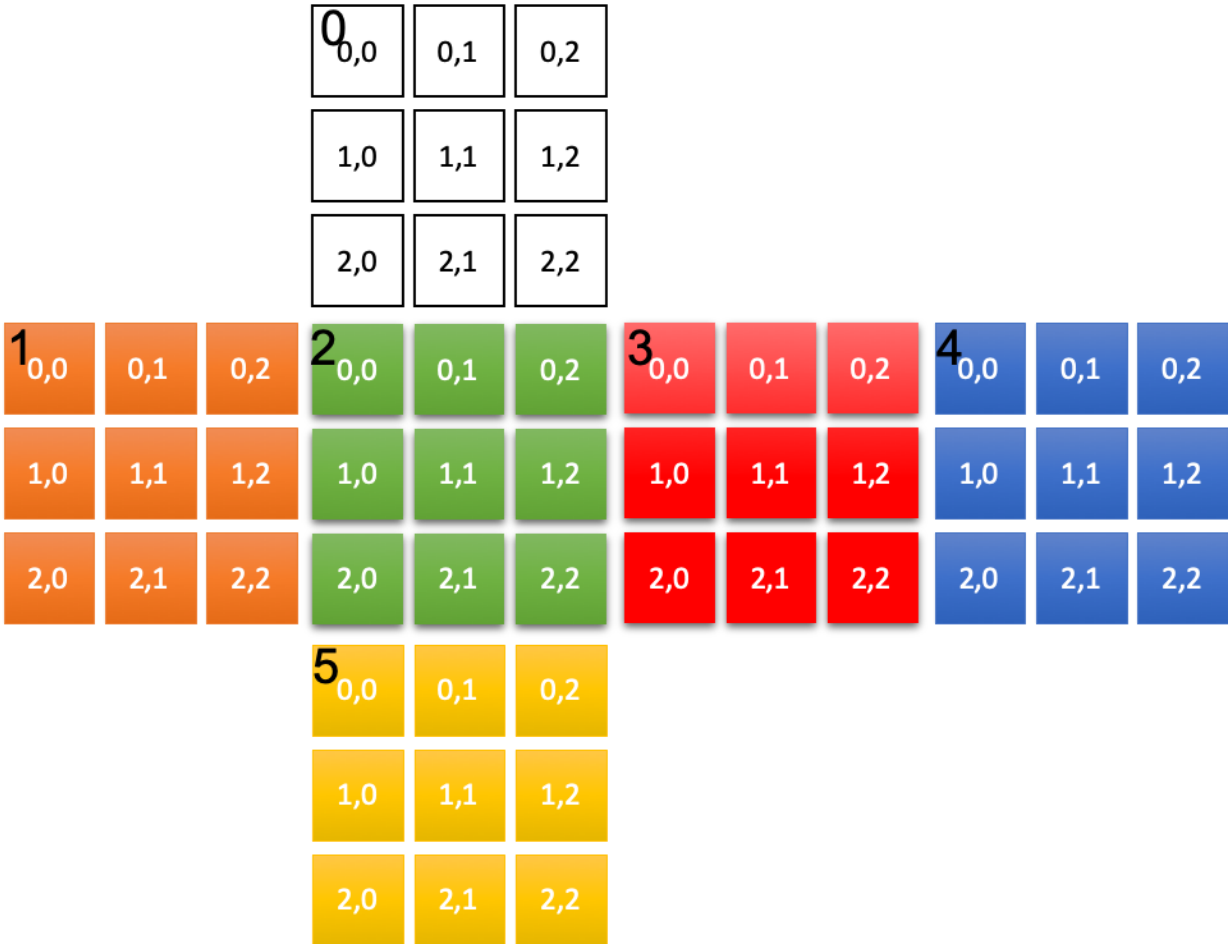
Die einzelnen Felder kann man mit den Methoden **setField** und **getField** ansprechen. Dabei habe ich mich entschieden, der Matrix zwei Koordinaten zu geben: Row und Column. Was uns zum nächsten Abschnitt bringt: Wie sind die Würfelseiten eigentlich ausgerichtet, damit man ein Koordinatensystem einsetzen kann?

Das Koordinatensystem

Die Ausrichtung muss für das Modell so festgelegt werden, dass man bei der Angabe von Koordinaten immer dieselbe Position auf dem Würfel meint. Üblich[8] ist es, dass man den Würfel von einer Seite frontal betrachtet und diese Seite als „Front" bezeichnet. Das Koordinatensystem des

[8] Diese Bezeichnungen (z.B. Front") habe ich von den einschlägigen Seiten aus dem Netz der Netze

Würfels der Klasse **Cube** ist von mir in einem Akt purer selbstverliebter Willkür folgendermaßen festgelegt worden:



Jede Seite (**CubeFace**) hat den in schwarz in der oberen Ecke stehenden Index. Dann folgen die Koordinaten **Row** und **Column** für jedes Feld. Alle Angaben von Koordinaten verwenden immer diese Werte. Ich halte es für eine gute Idee, die Seiten-Indexe in Form einer **CubeColor** anzugeben, die sich mit der **Enum**-Methode **ordinal()** notfalls in einen **int**-Index wandeln lässt.

Solange man nichts bewegt findet man sich damit schon ganz gut zurecht. Fängt man aber an etwas zu verdrehen, fangen die Probleme an und es wird knifflig. Wenn man einen total verdrehten Würfel vor sich hat, wo ist dann der Ursprung dieser Koordinaten?

Mir ist dazu eingefallen, dass die einzigen Flächen, die sich bei einem 3x3-Cube nicht gegeneinander bewegen lassen, die mittleren sind. Im Bild oben also alle mit der Position Row = 1 und Column = 1 (1,1). Weiß und Gelb, Orange und Rot, sowie Grün und Blau liegen sich immer gegenüber. Auch die Nachbarn sind immer dieselben. Das liegt daran, dass diese Teile beim physischen Würfel an einem festen Kreuz angebracht sind und sich nur um sich selbst drehen lassen.

Damit liefert uns dieses Wunder der Mechanik auch den festen Bezugspunkt des Koordinatensystems: Die weiße Mitte bleibt auf dem **CubeFace** 0, Orange auf 1 und so weiter. Schaut man auf die Front 2 (**CubeColor.Green**), hat man also immer den grünen Mittelpunkt vor sich.

Bewegung

Aktuell bildet das zuvor beschriebene Modell einen Würfel ab, dessen einzelne Teile alle mit Superkleber miteinander verschweißt sind. Damit kann man auch seinen Spaß haben, ist dann aber eher in der Beobachterposition und braucht noch eine weitere Person mit einer gewissen Frustrationstoleranz dazu.

Ich habe mich entschieden, das Prinzip „Separation of concerns“[9] auf Modell und Bewegung anzuwenden und beides getrennt zu betrachten. Deswegen gibt es die Klasse **CubeFaceRotator**. Sie ist für die Rotation von Oberflächen und auch mittleren Schichten zuständig. Die öffentliche Methode **rotateFace**() dreht eine Seite mit Null bis n Schichten, **rotateLayers**() kann auch mittlere Schichten drehen, ohne die Flächen mit zu drehen.

Intern ist das Ganze etwas umfangreicher.

Was uns bewegt

Dreht man die Fläche eines Würfels, dann sind immer insgesamt fünf **CubeFace**-Objekte betroffen. Das ist wohl auch das Geheimnis des Zauberwürfels, welches ihn so hinterhältig macht. Spätestens nach der dritten zufälligen Drehung fangen Leute wie ich an, die Sache durch Ablösen der Farbaufkleber und Neubekleben zu lösen.

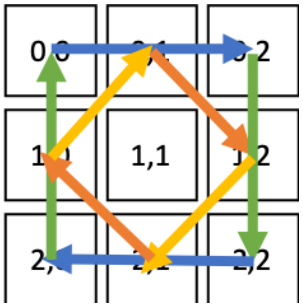
Die neueste Generation der Würfel hat deswegen keine Aufkleber mehr, sondern fest verbundene Plastik-Farbflächen. Das ist nicht verbraucherfreundlich. Deswegen gehen wir zur nächsten Eskalationsstufe und machen das mit Software. Da startet man einfach das Programm neu. Oder lässt sich die Lösung anzeigen. Wir lachen dir in’s Gesicht, Ernö Rubik!

Eine Fläche rotieren

Eine Fläche zu rotieren bedeutet, die Farbwerte des **CubeFace**-Objekts zu verschieben. Das folgende Bild ist eine Rotation im Uhrzeigersinn:

---

[9] „Trennung der Aufgaben“. Eine Klasse sollte immer nur eine Aufgabe haben. Wobei der Umfang der Aufgabe immer auch Interpretationsspielraum lässt.



Die private Methode **rotateTopClockwise**() macht das mit folgendem Code:

```
private void rotateTopClockwise(CubeFace face) {
    final CubeFace sourceFace = new CubeFace(face);
    int maxIndex = _cube.getDimension() - 1;

    int targetColumnIndex = maxIndex;
    for (int sourceRowIndex = 0; sourceRowIndex <= maxIndex; sourceRowIndex++) {
        int targetRowIndex = 0;
        for (int sourceColumnIndex = 0; sourceColumnIndex <= maxIndex; sourceColumnIndex++) {
            face.setField(targetRowIndex, targetColumnIndex,
                    sourceFace.getField(sourceRowIndex, sourceColumnIndex));
            targetRowIndex++;
        }
        targetColumnIndex--;
    }
}
```

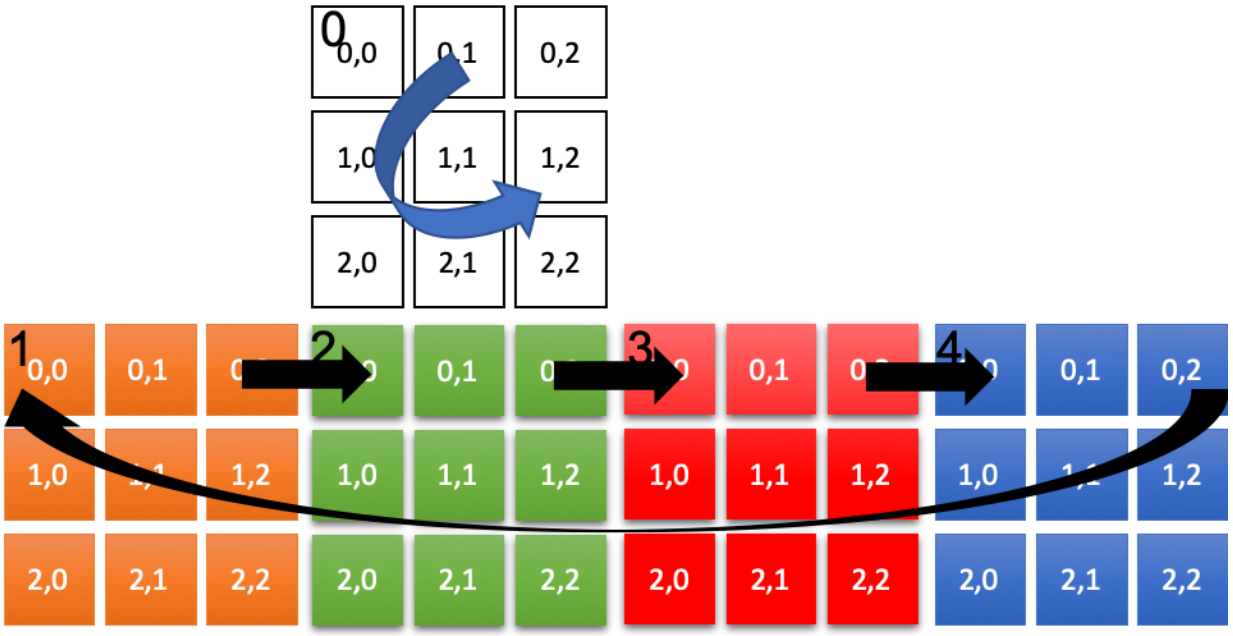
Bei Verschiebeaktionen ist es meistens eine gute Idee, wenn Quelle und Ziel nicht identisch sind. Deswegen habe ich erst eine Kopie namens *sourceFace* angelegt und bediene mich dieser als Datenquelle. Die Schleife ist so angelegt, dass sie keinen Mist baut, d.h. sie erzeugt Quell- und Zielkoordinaten korrekt. Dann wird einfach der Wert aus den Quellkoordinaten in die Zielkoordinaten der Würfelfläche kopiert.

Die Methode **rotateTopCounterclockwise**() macht das Ganze anders herum. Womit wir die Drehmöglichkeiten für die Oberseite eines Würfels bereits abgedeckt hätten.

Ich habe im Internet auch gleich mal geschaut, ob es für Java Matrizenklassen gibt, die sowas können. Und siehe da: Nö. Zumindest nichts, was Out-Of-The-Box dabei wäre. Da ich mich nicht von externer Raketentechnik abhängig machen will, habe ich das also selbst geschrieben.

Die Seitenteile verschieben

Kommen wir zu den vier Seitenflächen. Hier schiebt sich eine komplette Reihe jeweils auf die nächste Nachbarfläche, bis alle vier Seiten rotiert sind:



Die Methoden dafür nennen sich **shiftSideLayerClockwise**() und **shiftSideLayerCounterclockwise**(). Sie sind etwas komplexer, weil die betroffenen Koordinaten leider bei jeder der Oberflächen ganz andere sind. Deswegen werden diese vor der jeweiligen Aktion „normalisiert". Das bedeutet, dass für den Algorithmus jede Seitenfläche erst so gedreht wird, dass die zu verschiebende Reihe die Row 0 ist. Anschließend muss das Ganze wieder zurückgedreht werden (denormalisiert). Für diesen lästigen Mist musste ich einige Tabellen erfassen, aus denen die für die jeweilige Situation notwendigen Indexwerte für Nachbarflächen, Drehrichtungen und Anzahl der Drehungen herausgelesen werden. Ich hasse diesen Scheiß.

RotationDirection

Für die Drehrichtung habe ich die Bezeichnungen Clockwise und Counterclockwise in eine eigene Enumeration gepackt. Man kann das auch ganz klassisch mit Werten wie 1 und 0 oder **True** und **False** machen. Kann dann halt niemand mehr lesen.

```
public enum RotationDirection {
    Counterclockwise,
    Clockwise,
    None
}
```

Bei allen Angaben einer Drehrichtung verwende ich diese Enumeration. Bei der Drehung einer **CubeFace** betrachtet diese immer frontal, sodass die Drehrichtung in diesem Fall klar sein dürfte. Beim Drehen eines ganzen Würfels hilft einem das zunächst nicht, aber das kommt später dran.

Eine eigene Klasse für Bewegung

Die Klasse heißt, wie schon erwähnt, **CubeFaceRotator**. Sie kann von einem Cube ein Face rotieren. Linksrum (**Counterclockwise**) und rechtsrum (**Clockwise**). Das ist schön, weil wir dadurch dem Sinn des Zauberwürfels erstmals etwas näherkommen. Ich hätte die Methode auch als Methode des Würfels implementieren können. Habe ich aber nicht gemacht, weil ich es irgendwie cool finde, Modell und Logik voneinander zu trennen.

```
public void rotateFace(
  final RotationDirection direction,
  final CubeColor face,
  final int countOfLayers)
```

Die Methode **rotateFace** kann auch noch weitere Ebenen des Würfels beim Drehen mitnehmen. Das kann man über die countOfLayers mitgeben. Standard sollte 1 sein. Mit mehr als 380 Zeilen (inklusive JavaDoc-Dokumentation) ist **CubeFaceRotator** tatsächlich die größte Klasse des Projekts. Damit kann ich also leider nicht angeben. Ich kenne einzelne C++-Methoden aus meiner beruflichen Praxis, die weitgehend unkommentiert locker auf das Zehnfache kommen. Ich muss also noch ein wenig an mir arbeiten.

Der Würfel, den die Methode drehen soll, wird dem Konstruktor der Klasse mitgegeben. Ich habe bei anderen Klassen häufiger mal eine statische Methode gebaut, die nur intern eine Instanz ihrer Klasse anlegt. Das Kriterium, wann ich das Eine oder das Andere mache ist, ob man für dasselbe Objekt eine Methode häufiger oder i.d.R. nur einmal aufruft. Einer statischen Methode ständig dasselbe Argument mitgeben zu müssen, kann auf die Dauer ermüden.

Die zweite Methode der Klasse ist diese hier, die sogar von der ersten intern gerufen wird:

```
public void rotateLayers(
      final RotationDirection direction,
      final CubeColor face,
      final int startRow,
      final int countOfLayers
```

Sie hat den zusätzlichen Parameter **startRow**. Das ermöglicht es z.B. nur den mittleren Layer des Würfels zu drehen, was echte SpeedCuber häufig einsetzen. Dann gibt man bei einem 3x3-Würfel als **startRow** 1 statt 0 an.

Record and Play

Mit den bisherigen Klassen kann man Java-Code schreiben, der einen virtuellen Würfel abbildet und drehen kann. Das ist als Benutzerschnittstelle für echt hartgesottene Nerds ein Traum, kann aber bei Normalsterblichen zu Frustrationen führen. Wenn man weiterdenkt, braucht man also irgendwas, mit dem man die Würfelbewegungen festhalten und abspielen kann. Ist dann auch für die späteren Lösungsalgorithmen noch vorteilhaft, wie sich später herausstellen wird.

Ich brauche also eine Klasse, die eine Drehung speichert, sowie eine Collection, die mehrere solcher Datensätze halten kann: **CubeFaceRotationRecord** und **CubeFaceRotationRecords**. Ein Player für diese Daten wäre auch noch angebracht: **CubeFaceRotationPlayer**.

CubeFaceRotationRecord

Welche Daten für eine Drehung notwendig sind steht praktischerweise schon in der Parameterliste der Methode **CubeFaceRotator.rotateFace**:

```
public class CubeFaceRotationRecord {
    private final CubeColor _face;
    private RotationDirection _direction;
    private int _startRow;
    private int _countOfLayers;
```

OK, **_startRow** fehlt in **rotateFace**. Soll uns hier aber nicht ablenken, wir nehmen einfach mal an, dass das immer 0 ist. Da ich ein Fan davon bin, Modell und Logik voneinander zu trennen, hat die Klasse lediglich Konstruktoren und set/get-Methoden, um auf die Daten zuzugreifen. Das Ding kann also eigentlich fast nix.

CubeFaceRotationRecords

Noch einfacher ist die Collection, die als eine Art Aufzeichnungsband[10] für die Drehungen dient:

```
public class CubeFaceRotationRecords extends ArrayList<CubeFaceRotationRecord> {
}
```

Da muss man nichts weiter erklären.

[10] Für die Jüngeren unter uns: Früher hat man mal Daten auf Magnetbändern gespeichert. Da ging auf kilometerlange Bänder ein winziger Bruchteil einer heutigen Micro-SD-Karte.

CubeFaceRotationPlayer

Wenn wir Drehungen ausführen wollen, benötigen wir

1. Einen **Cube**
2. Einen **CubeFaceRotator**

Der **CubeFaceRotator** enthält als Member den zu drehenden **Cube**. Der Konstruktor braucht also nur den **CubeFaceRotator**:

```
public CubeFaceRotationPlayer(CubeFaceRotator rotator) {
    _rotator = rotator;
}
```

Die einzige Methode der Klasse **CubeFaceRotationPlayer** heißt **play**:

```
public void play(CubeFaceRotationRecords records)
```

Darin läuft eine Schleife über alle **CubeFaceRotationRecord**-Einträge. Anschließend ist der Status des im **CubeFaceRotator** referenzierten **Cube**s entsprechend geändert.

Mit diesen Klassen kann ich jetzt Sequenzen von Bewegungen abspielen, was später ziemlich nützlich ist, wenn die Lösungsalgorithmen ins Spiel kommen.

Damit hätten wir jetzt das Modell und die Bewegungslogik des Modells.

Alles durcheinander

Bevor ich mich an die Lösung mache, beschäftige ich mich erst noch kurz mit dem Gegenteil. Um Unordnung zu schaffen brauche ich noch keine Hilfe im Internet. Ich würde mich da sogar ganz unbescheiden als eine Art Experte bezeichnen. Nicht nur zum Testen ist eine Klasse für das zufällige Verdrehen des Würfels nützlich. Ich benötige nur noch einen Zufallsgenerator, der mir die Daten von **CubeFaceRotationRecord**-Objekten erzeugt. Das hat gegenüber dem direkten zufälligen Verdrehen des Würfelmodells den Vorteil, dass der Weg zum Durcheinander aufgezeichnet ist. Bei Tests kann ich das dann als SpeedCube-Notation ausgeben. Falls es sich bei einem Fehler um einen der üblichen „bedauerlichen Einzelfälle" handelt, kann man den dann präzise Wiederholen.

Die Klasse heißt **CubeScrambler** und hat u.a. die folgende Methode:

```
public static CubeFaceRotationRecords scrambleCube(int depth, int dimension) {
    Random generator = new Random(System.nanoTime());
    CubeFaceRotationRecords records = new CubeFaceRotationRecords();

    for (int i = 0; i < depth; i++) {
        RotationDirection direction = generator.nextInt(2) == 0 ?
                RotationDirection.Clockwise : RotationDirection.Counterclockwise;
        int face = generator.nextInt(CubeColor.values().length);
        int countOfLayers = dimension <= 3 ? 1 : generator.nextInt(dimension / 2) + 1;

        CubeFaceRotationRecord record =
                new CubeFaceRotationRecord(CubeColor.values()[face], direction, countOfLayers);
        records.add(record);
    }
    return records;
}
```

Die zweite Methode verdreht den Würfel sofort:

```
public static void scrambleCube(Cube cube, int depth) {
    CubeFaceRotationPlayer player =
            new CubeFaceRotationPlayer(new CubeFaceRotator(cube));
    player.play(scrambleCube(depth, cube.getDimension()));
}
```

Praktisch, dass ich für diese Variante einfach die andere aufrufen kann. Der Parameter *depth* gibt an, wie viele Verdrehungen die Methode vornehmen soll. Ein Wert von 0 sorgt beim Lösen für maximale Entspannung. 1 ist auch noch akzeptabel. Leute wie ich sollten es sich genauer überlegen, Werte darüber anzugeben.

Warum habe ich nicht einfach die Farbflächen zufällig auf dem Würfel verteilt und habe stattdessen zufällige Drehungen verwendet? Die Antwort liegt in der maximalen Frustration, die ein unlösbarer Würfel verursacht. Die Gesetze der Mechanik sagen mir, dass ein per Drehungen durcheinander gebrachter Würfel mindestens durch Umkehr der Drehungen wieder lösbar sein müsste. Das beruhigt schon ein bisschen. Das freie Verteilen von Farbflächen dürfte dagegen in fast allen Fällen zu einem unlösbaren Problem führen, sofern man nicht ganz viele komplizierte Regeln einhält, z.B. dass der Endzustand durch Verdrehen eines gelösten Würfels entstehen muss.

Lösen

Es soll jemanden geben, der Würfel jongliert und sie dabei auch noch löst. Da diese Krankheit offensichtlich sehr selten ist, dürfte die Pharmaindustrie nichts Passendes anzubieten haben. Der Mensch muss also wohl bis zu seinem Lebensende damit leben. Mein Code verzichtet zunächst auf

den zirkusreifen Teil und soll einfach nur den verdrehten Würfel wieder in Ordnung bringen. Wer will, kann sich ja einen eigenen Branch abziehen und das mit dem Jonglieren ergänzen.

Weil ich mich beim Lösen eines Würfels etwa so intelligent anstelle wie ein Eimer Sand, muss ich mich im Internet umschauen, ob mir damit jemand helfen kann. Das unterscheidet uns dann eben doch vom Sand: Wir wissen, wie man googelt. Und siehe da: Speedcube.de[11] bietet eine Lösungsvorlage, bei der sogar ein Eimer Sand eine reelle Chance auf Erfolg hat: die Layer-by-Layer-Lösung für Anfänger. Das scheint mir machbar.

Erstmal eine Sprache für Bewegungen

Mit der Klassen **CubeRotationRecords** und **CubeRotationRecordPlayer** sind die Grundlagen geschaffen, Sequenzen von Bewegungen abzuspielen. Sowas ist die Basis, wenn man Lösungen aus dem Internet schamlos abkupfern will, denn die beschreiben immer solche Sequenzen. Speedcube.de widmet eine Seite nur der Notation solcher Bewegungssequenzen[12]. Ich muss also noch was programmieren, was diese Notation für meine Klassen übersetzt.

Eingabe

Ich bin nicht besonders fleißig. Zumindest mache ich mir ungern vermeidbare Arbeit, die keinen Spaß macht. Deswegen möchte ich die Sequenzen, die ich auf der SpeedCube-Seite sehe, genauso verwenden, wie sie dort stehen. Meine Idee: Eine Klasse zu haben, die aus einem String in dieser Syntax eine gefüllte **CubeRotationRecords**-Collection macht. Ich hätte die Klasse sicher auch Zitronenfalter nennen können, aber aus naheliegenden Gründen habe ich mich dann für **SpeedCubeNotationInterpreter** entschieden. Mit entscheidend für diese Namenswahl war folgende Methode:

```
public void addMoves(final String moves)
```

Der Konstruktor bekommt die **CubeRotationRecords**-Instanz als Parameter, in der alle mit **addMoves** hinzugefügten Bewegungen landen:

```
public SpeedCubeNotationInterpreter(final CubeFaceRotationRecords records)
```

Die hier erwähnte Klasse mit der Methode kommt in allen Lösungsklassen zum Einsatz, um die Bewegungen des Würfels aufzuzeichnen.

[11] https://speedcube.de/lbl.php
[12] https://speedcube.de/notation.php

... und Ausgabe

Das Programm soll die Lösung nicht für sich behalten. Das wäre irgendwie egoistisch. Die Darstellung des gelösten Würfels allein reicht dafür nicht. Und jeden einzelnen Schritt durch Darstellung des jeweils ganzen Würfels ist auch irgendwie blöd. Die SpeedCube-Notation des Lösungswegs dagegen ist eine ordentliche Alternative, so dass eine Klasse für die Rückübersetzung der **CubeRotationRecords** eine gute Idee zu sein scheint. Diese Klasse habe ich programmiert und sie heißt **SpeedCubeNotationWriter**. Sie besitzt genau eine öffentliche Methode:

```
static String write(final CubeFaceRotationRecords records)
```

Also: **CubeFaceRotationRecords**-Objekt 'rein, **String** mit Anweisungen in SpeedCube-Notation 'raus. Das kann ich an vielen Stellen mal gebrauchen.

An dieser Klasse kann man im Code auch eine Vorgehensweise erkennen, die ich bei den Lösungsklassen praktisch überall eingesetzt habe: Eine öffentliche statische Methode, die intern eine Instanz der eigenen Klasse anlegt und dann darin weiter arbeitet, indem sie private Methoden der Instanz aufruft. Warum mache ich so etwas Schräges? Weil ich's kann.

Außerdem hat es aus meiner Sicht auch Vorteile, wenn man die Klasseninstanz nur für einen Aufruf von außen benötigt. Intern werden Membervariablen verwendet, anstatt alles per Parameter weitergeben zu müssen. Nur wenn die Methoden der Klasse mehrfach mit denselben Daten, oder mehrere verschiedene Methoden für dieselben Daten benötigt werden, verwende ich öffentliche Konstruktoren.

Die Methode **write** ist zurzeit nicht sonderlich schlau. Sie schreibt stur alle Bewegungen aus der in Das Koordinatensystem auf Seite 7 beschriebenen Sicht auf den Würfel. Das kann mitunter zu Verknotungen der oberen Extremitäten führen, was mir aber aktuell völlig Schnuppe ist, denn ich mach sowas bestimmt nicht selbst. Wenn die Leute mal mit Mistgabeln und Fackeln vor meiner Haustür stehen, kann ich mir ja noch was überlegen.

Wir brauchen eine Strategie

Anders als manch jung-dynamische Führungskraft haben wir Alten noch gelernt, dass für hinreichend komplexe Aufgaben auch mal ein Plan ganz hilfreich sein kann. Ich habe mir den für die Lösung eines Würfels mangels eigener Intelligenz aus dem World-Wide-Web ge-cloud[13]. Die Lösung, die auch noch extra für blutige Anfänger gedacht ist, heißt „Layer-by-Layer"-Algorithmus

[13] Was für ein Schenkelklopfer! Echt zum Wegschmeißen...

(„LbL“). Auf der Seite speedcube.de[14] ist die Anleitung komplett so beschrieben, dass man mit etwas Hirnschmalz ein Programm daraus machen kann.

Schrittweise aus dem Chaos

Es gibt gute Gründe, die Lösungsalgorithmen in getrennten Klassen zu halten und nicht mit in die Cube-Klasse hinein zu packen. Sie fallen mir nur momentan nicht ein. Ich geh‘ erstmal einen Kaffee trinken.

Doch, jetzt hab‘ ich’s wieder:

1. Der Algorithmus ist nur für 3x3-Würfel gültig, aber das Modell kann auch n x n.
2. Es gibt auch andere Algorithmen für 3x3, das Modell gilt aber für alle.
3. Klassen sollten nicht zu Gott-Klassen werden und Alles können.

Außerdem sollte man beim Layer-by-Layer-Algorithmus schrittweise vorgehen. Das steht so im Internet. Und was dort steht, ist immer wahr. Als ich das gelesen habe, hat eine innere Stimme zu mir gesagt, dass ich für jeden beschriebenen Schritt auch eine eigene Klasse schreiben soll. Ich folge meiner inneren Stimme, es sei denn, sie wird von der Stimme meiner Frau überstimmt. Im Fall der Würfellösung hat sich meine Frau aber herausgehalten und folgende Klassen sind herausgekommen:

1. **WhiteCrossStep**
   Erzeugt ein weißes Kreuz mitsamt den richtigen Kantenfarben.
2. **WhiteCornersStep**
   Bringt die vier weißen Ecken in die richtige Positionen und Orientierungen, so dass die gesamte weiße Fläche vollständig ist.
3. **SecondLayerStep**
   Bringt die vier Kanten der mittleren Schicht in die richtigen Positionen und Orientierungen, so dass zwei Schichten gelöst sind.
4. **YellowCrossStep**
   Bringt die vier gelben Kantensteine an die richtigen Positionen, aber eventuell falsch orientiert.
5. **YellowCrossLinearEdgesStep**
   Bringt ggfs. zwei gegenüberliegende gelbe Randsteine an den richtigen Positionen in einen Zustand, dass zwei gelbe Randsteine in einem Winkel positioniert sind.
6. **YellowCrossAngledEdgesStep**
   Bringt die gelben Randsteine in die richtigen Positionen und Orientierungen, so dass ein gelbes Kreuz entsteht.

[14] https://speedcube.de/index.php

7. **YellowCornersPositionStep**
   Bringt die gelben Ecksteine an die richtigen Positionen, aber eventuell falsch orientiert.
8. **YellowCornersOrientationStep**
   Dreht die gelben Ecksteine in die richtigen Orientierungen, so dass der Würfel gelöst ist.

Die Namen der Klassen folgen einem einfachen Schema: Was ist nach dem Schritt richtig angeordnet plus das englische Wort für „Schritt". Ich bin ganz aus dem Häuschen, denn das sieht fast so aus wie ein Plan. Jetzt fehlt nur noch der Code, der das Richtige tut. Ganz bestimmt ein Kinderspiel. Jeder Chef würde spätestens jetzt fragen, ob wir schon fertig sind.

Ein Schritt in die richtige Richtung...

Mir ist beim Klauen der Lösung aufgefallen, dass die ersten drei Schritte einem gemeinsamen Muster folgen:

1. Suche ein falsch positioniertes Teil für eine Seitenfarbe
2. Finde für diese Position die passende Lösung
3. Wende diese an und gehe zur nächsten Seitenfarbe

Das Ganze muss man für die vier Seiten (Orange, Grün, Rot und Blau) genau vier Mal tun. Die Lösungen sind für die drei verschiedenen Schritte natürlich unterschiedlich, aber dieses Muster ist für alle dasselbe. Deswegen habe ich mich der Kunst der objektorientierten Programmierung hingegeben und eine Basisklasse für diese drei Schritte programmiert:

```
abstract class AbstractSolutionStep
```

Die oben beschrieben Schleife über die vier zu lösenden Würfelteile stecken schon komplett ausprogrammiert in folgender Methode:

```
private static final CubeColor[] faceSteps = {
  CubeColor.Green, CubeColor.Orange, CubeColor.Blue, CubeColor.Red
};
```

```
void solve() {
   SpeedCubeNotationInterpreter interpreter =
     new SpeedCubeNotationInterpreter(_records);
   StringBuilder cubeRotations = new StringBuilder();

   for (int i = 0; i < 4; i++) {
      Cube steppedCube = CubeFactory.create(_cube, _records);

      PartPosition position = findPosition(steppedCube, faceSteps[i]);
      String solutionMoves = cubeRotations + findSolutionFor(position);
      interpreter.addMoves(solutionMoves);

      // Go to the next front face: The y-rotations are only interpreted by each
      // addMoves call, so we have to append one "y" per rotation.
      _orientation.rotate('y', RotationDirection.Clockwise, 1);
      cubeRotations.append("y ");
   }
}
```

Was bei der jeweiligen Suche nach der passenden Lösung eines Würfelteils echt praktisch ist, sind die relativ wenigen Möglichkeit, wo und in welcher Orientierung die Teile sich auf dem Würfel verstecken. Bei den Randteilen sind das nämlich ganze 12 Positionen mit jeweils zwei Orientierungen: Es gibt also ungefähr 24 verschiedene Lösungen. Eine davon ist ziemlich kurz, nämlich wenn das Teil schon richtig steht. Die anderen findet man u.a. auf der SpeedCube-Seite zum abtippen.

Lösungssuche

Jede Lösung passt also zu einer Position eines Würfelteils. Da bietet es sich an, eine Klasse dafür zu programmieren, die Position und passende Lösung (**Solution**) enthält:

```
class Solution {
   final PartPosition position;
   final String moves;

   Solution(PartPosition position, String moves) {
      this.position = position;
      this.moves = moves;
   }
}
```

Die Klasse **PartPosition** enthält die Koordinaten und Ausrichtung des Teils;

```
class PartPosition {
   private int _face;
```

```
private int _row;
private int _column;
```

Für eine Kante muss man lediglich festlegen, welches die **_face**-Farbe sein soll. Das ist bei den weißen Kantensteinen z.B. immer die Seitenfarbe (weiß weiß man ja).

Die Methode **findPosition**, die in **solve** aufgerufen wird, liefert ein Objekt der Klasse **Position** zurück. In der jeweils abgeleiteten **Step**-Klasse habe ich in einem **Solution**-Array die Bewegungen für jede mögliche **Position** in SpeedCube-Syntax erfasst:

```
private static final Solution[] solutions = {
        // main color found at the up face
        new Solution(new PartPosition(up, 2, 1),
                noMove),
        new Solution(new PartPosition(up, 1, 2),
                upRightToFrontRight +
                frontRightToFrontUp + turnEdge),
        new Solution(new PartPosition(up, 0, 1),
                upBackToDownBack +
                downBackToDownFront +
                downFrontToUpFront),
        new Solution(new PartPosition(up, 1, 0),
                upLeftToFrontLeft +
                frontLeftToFrontUp + turnEdge),
...
```

Für die Strings habe ich mir erlaubt, Konstanten zu definieren, z.B.:

```
private static final String backRightToFrontDown = "B' D2 B ";
```

Viele Sequenzen setzen sich aus wiederkehrenden Teilen zusammen, so dass man diese einfach nur neu zusammensetzen muss.

So sieht das fast einfach aus. Der große Trick ist dabei, dass die 24 Lösungen für alle Farben verwendet werden können, sofern man den Würfel passend dreht und die gesuchten Farben dann als angepasste Parameter übergibt. Mit dem SpeedCube-Kommando „y" wird das in der abstrakten Basisklasse in der Methode **solve()** getan. Im Array **faceSteps** steht noch die jeweilige Farbe, die bei dem Schritt jeweils dran ist. Die Methode **findSolutionFor**() sucht dann nur noch das Array nach der Position durch und liefert die auszuführende Drehsequenz.

Im Grunde läuft das bei fast allen Schritten auf ähnliche Art und Weise, variiert allerdings so, dass ich die Algorithmen nicht in sinnvollen Basisklassen implementieren konnte oder wollte. Die Lösungsklassen sind recht übersichtlich geblieben. Unittests für jeden Schritt sind nicht ganz

unwichtig, denn ich lerne immer wieder auf die harte Tour, dass Code nicht dadurch korrekt läuft, indem man nur fest daran glaubt.

Der letzte Schritt ist der schwerste

Die Klasse **YellowCornersOrientationStep** ist nicht nur die mit dem längsten Namen. Sie repräsentiert auch den letzten Schritt zur Gesamtlösung und löste bei mir den Wunsch aus, mich in die professionellen Hände eines fähigen Therapeuten zu begeben. Dabei sah es auf der SpeedCube-Seite zunächst nach dem einfachsten Schritt aus: „R' D' R D" (Rechts gegen den Uhrzeigersinn, unten gegen den Uhrzeigersinn, rechts im Uhrzeigersinn und unten im Uhrzeigersinn). Einfach für jede gelbe Ecke wiederholen, bis es stimmt.

Anders als bei den vorangegangenen Schritten muss hier

- der Status des Würfels während der Bewegungen geprüft
- eine Seite des Würfels abhängig vom Status gedreht werden

Die zu lösende Ecke muss vom Betrachter aus gesehen stets rechts unten auf der gelben Fläche sein. Bei der ersten Ecke wird dafür der ganze Würfel gedreht. Die weiteren aber erfordern es, dass nur die gelbe Fläche gedreht wird.

Das ist der Grund für schwerste mentale Ausfallerscheinungen während der Tests. Immer wieder bin ich das Mapping der Koordinaten und Eckenfarben durchgegangen und rätselte, warum es diesmal wieder nicht funktioniert hat. Ich bin sogar so weit gegangen, mir die Schritte mit Buntstiften auf Papier zu malen, damit ich meine eigene Lösung ansatzweise verstehen kann. Hat lange Zeit auch nichts genützt.

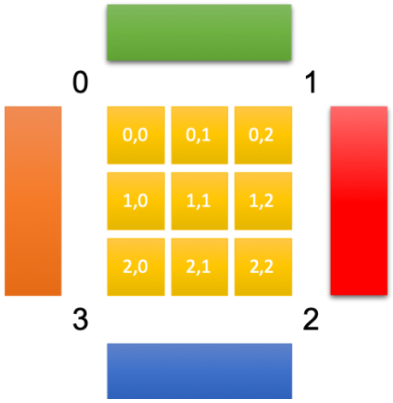
Wer meine Verzweiflung nachvollziehen will, muss sich einfach den Code der Klasse anschauen. Knackpunkt ist vor allem die Methode

```
private static boolean isCornerSolved(
        final Cube cube,
        final int coordinatesIndex,
        final int colorIndex) {
```

Sie muss die Drehung des ganzen Würfels am Anfang, die Schritte für jede Ecke (welche Ecke mit welchen Farben ist gerade dran?), und die Rotationen der gelben Fläche so berücksichtigen, dass sie die richtigen Farbwerte des richtigen Teils prüft. Der Hammer.

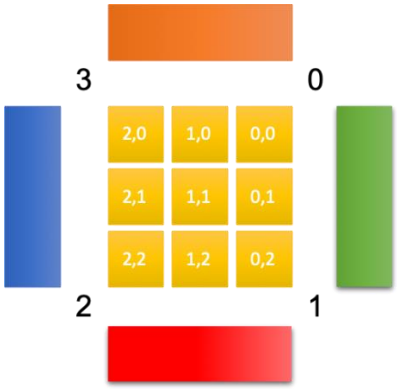
Die Idee ist, dass der **coordinatesIndex** die Ecke angibt, welche bei der aktuellen Sicht auf den Würfel die rechte untere Ecke ist. Diese bleibt nach dem ersten Drehen des ganzen Würfels

während des letzten Schritts konstant. Damit habe ich bei der Prüfung immer die Koordinaten der zu prüfenden Ecke.



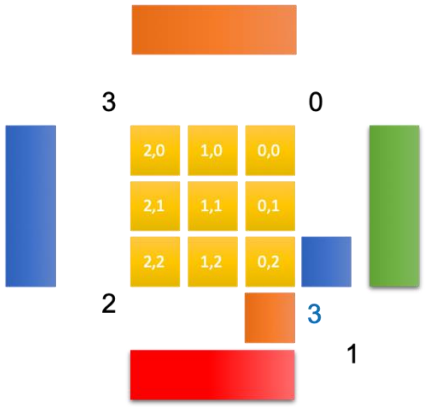
Das Bild oben ist die Ausgangslage, wenn der Würfel über die x-Achse so gedreht wird, dass die gelbe Seite oben ist. Die Indexwerte der Ecken sind in schwarzer Schrift angegeben. Die Farbbalken geben an, welche Farbflächen sich auf welcher Seite der gelben Fläche befinden. Kann ich noch verstehen.

Jetzt nehmen wir an, die Ecke mit den Koordinaten 0,2 ist verdreht. Der Algorithmus gibt an, man soll den ganzen Würfel so drehen, dass die erste verdrehte Ecke rechts unten liegt:



Schon dieses Bild vermittelt einen Eindruck davon wovon ich rede. **isCornerSolved** prüft immer die gerade rechts unten befindliche Ecke. **coordinatesIndex** ist in diesem Beispiel 1. Um die Ausrichtung dieser Ecke zu prüfen, muss ich die Farben der gelben Fläche an den Koordinaten 0,2, die der roten an 2,0, und die der grünen an 2,2 verwenden.

Das ist schon grenzwertig, aber nach wenigen Jahren intensiver Meditation in einem buddhistischen Kloster auf einem beliebigen Gipfel im Himalaya noch gerade so beherrschbar. Endgültig zur Raketentechnik wird das Ganze, wenn die gelbe Fläche (und nur diese!) nach der Lösung dieser Ecke zur nächsten verdrehten gedreht wird. Dann kommt ein abweichender **colorIndex** ins Spiel:



Das Bild oben zeigt die Situation, wenn nach der ersten gelösten Ecke die gelbe Fläche zweimal im Uhrzeigersinn auf die Ecke mit Index 3 (der **colorIndex**) gedreht wurde. Der Algorithmus gibt dies vor, wenn diese Ecke als nächste verdrehte gefunden wurde. Dann befinden sich die zu prüfenden Seitenfarben Orange und Blau dieser Ecke auf der roten und grünen Seite des Würfels. War ich zu schnell?

isCornerSolved muss also während der Lösung diesen ganzen Mist richtig berücksichtigen. Die aufrufende Methode muss dafür die richtigen Werte für **coordinationIndex** und **colorIndex** liefern. In der Klasse **YellowCornersOrientationStep** sind zwei Arrays mit Konstanten, aus welchen die Methode die richtigen Farben und Koordinaten für die jeweiligen Indexwerte herauslesen kann. Inklusive JavaDoc-Kommentierung ist die Klasse dann auch die mit den meisten Codezeilen geworden (über 300). Jede dieser Zeilen hat mich gefühlt eine Woche meines Lebens gekostet.

Das Frustierendste ist, dass ich das sichere Gefühl habe, dass jeder Leser dieses Papiers sich wahrscheinlich zu Recht sagt: „Wieso hat der Vollhonk das nicht einfach *so* gemacht?", wobei „*so*" eine beliebige andere, maximal simple und korrekte Lösung ist. Ich fühle mich manchmal einfach ein bisschen minderwertig.

Dokumentieren und Testen

Dokumentieren und Testen sind Tätigkeiten, die Programmierern richtig Spaß machen. Etwa so wie einen Basketball durch einen Gartenschlauch zu pressen. Mir gefällt so etwas auch sehr. Deswegen habe ich meine Klassen immer gleich mit Unittests versehen und alles mit **JavaDoc** auf Englisch beschrieben. Zumindest in einer Sprache, die ich für Englisch halte.

Der Unittest für die Klasse **LayerByLayerSolver** ist der entscheidende Anhaltspunkt dafür, dass mein Zeug irgendwas Sinnvolles tun könnte:

```
@Test
void solve_RandomCubes() {
   for (int i = 0; i < 500; i++) {
```

```
        Cube cube = new Cube();
        CubeFaceRotationRecords records =
            CubeScrambler.scrambleCube(15 + i % 10, cube.getDimension());
        CubeFaceRotator rotator = new CubeFaceRotator(cube);
        CubeFaceRotationPlayer player = new CubeFaceRotationPlayer(rotator);
        player.play(records);

        CubeFaceRotationRecords solution = LayerByLayerSolver.solve(cube);
        Cube solvedCube = CubeFactory.create(cube, solution);

        assertCube(solvedCube);
    }
}
```

Die Methode **assertCube** habe ich so gestaltet, dass sie einen gelösten Würfel erwartet. Falls diese Erwartung enttäuscht wird, reagiert sie entsprechend zickig:

```
private static void assertCube(
        final Cube solvedCube) {
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.White, "WWW WWW WWW");
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.Orange, "OOO OOO OOO");
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.Green, "GGG GGG GGG");
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.Red, "RRR RRR RRR");
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.Blue, "BBB BBB BBB");
    CubeAssertion.assertCubeFace(solvedCube, CubeColor.Yellow, "YYY YYY YYY");
}
```

Dafür habe ich die Klasse **CubeAssertion** und die Methode **assertCubeFace** geschrieben, welche die Beschreibung einer Würfelseite in leidlich lesbarer Form erwartet. Der Test verwürfelt den Würfel 500 Mal mit allerlei zufälligen Werten, versucht ihn zu lösen und prüft, ob er anschließend wieder schön ordentlich ist.

Nach etlichen Läufen bin ich mittlerweile geneigt, dem Test zu glauben. Es gibt immer wieder Fälle, dass Tests nichts testen, weil sie selber Fehler enthalten. Sie tun so, als würde alles richtig funktionieren. Mir ist das dann natürlich egal. Hauptsache, man sieht einen grünen Haken. Ich vermute aber, dass meine Testfunktion tatsächlich etwas testet. Deswegen schließe ich das Thema und dieses Dokument hiermit vorläufig ab.