CCD am Beispiel eines Java Raytracers

Im Zuge der Einsendeaufgabe **CCD** habe ich versucht den Code meines Java Raytracers aufzuräumen. Nach meinen Ergebnissen der Einsendeaufgaben **Metriken**, habe ich primär an der Lesbarkeit und der Einhaltung von Konventionen gearbeitet.

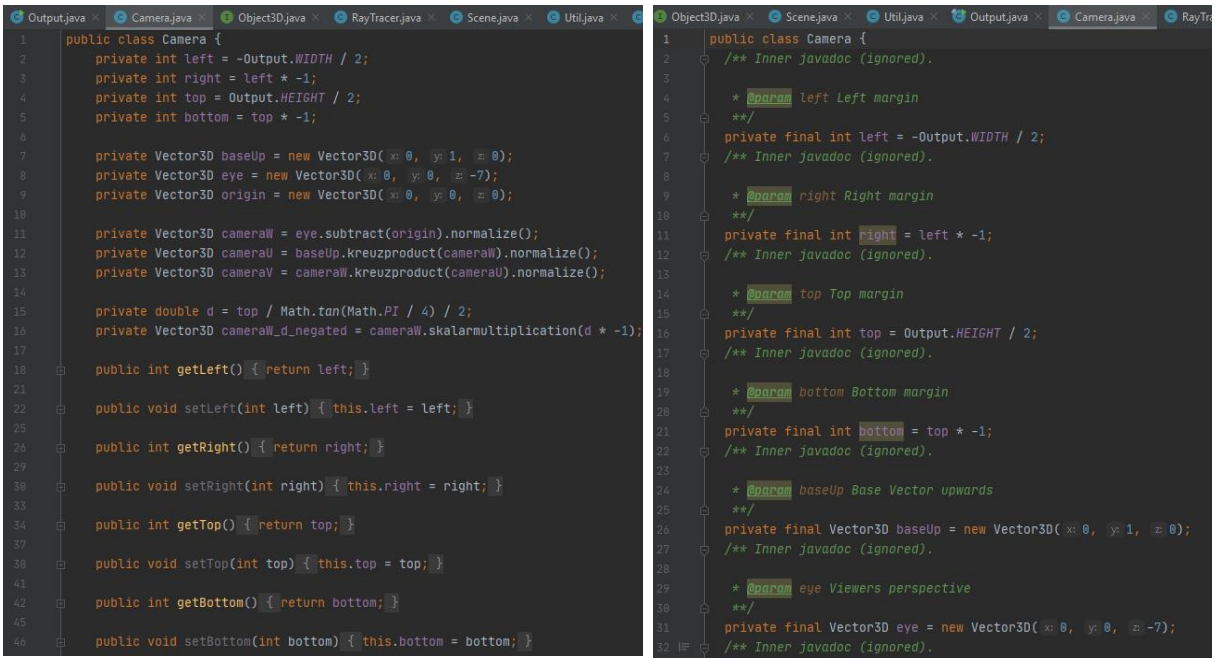
Als Gelegenheitsprogrammierer ist mir besonders aufgefallen, wie wichtig das Einhalten von Coding Standards ist. Vor allem wenn ich auf die Hilfe erfahrener Programmierer angewiesen bin ist es sinnvoll, den Code leicht verständlich und gut kommentiert zu schreiben. Auf diese Weise fällt es Anderen einfacher meine Gedanken nachzuvollziehen. Außerdem ist es so einfacher für mich, einen älteren Code aufzuarbeiten und fortzuführen.

Die Tabelle zeigt ein Cheat Sheet, welches von mir im Rahmen der Einsendeaufgabe erstellt wurde und als Referenz für den *Refactoring*-Prozess verwendet wurde.

Tabelle 1: Cheat Sheet für die Einsendeaufgabe CCD

CLEAN CODE		
Allgemein	**Verständlichkeit**	**Benennung**
Konventionen beachten Tools wie bspw. Checkstyle für gängige Konventionen verwenden	Konsistenz bestimmte Patterns bei ähnlichen Ausführungen beibehalten	Eindeutige Namen Variablen müssen eindeutig und im Kontext des Einsatzes benannt werden
Warnungen und Errors direktes Beheben verhindert spätere Probleme	Verständlich Kommentieren Vorgehensweise und Funktionen einzelner Code-Abschnitte müssen erklärt werden	Methoden Benennen was eine Methode tut, nicht wie sie es tut

Die nachfolgenden Ausschnitte zeigen Vorher-Nachher-Vergleiche der Anpassungen und Verbesserungen am Beispiel meines Java Raytracers. Als Referenz wurden die Anmerkungen und Errors des *Sun Checks* und *Google Checks* des Werkzeugs **Checkstyle** verwendet. Der Code wurde vom mir solange *refactored*, bis weder *Sun Checks* noch *Google Checks* Errors ausgegeben haben. Dadurch sollte mein Code einem Clean-Code ein großes Stück nähergekommen sein.

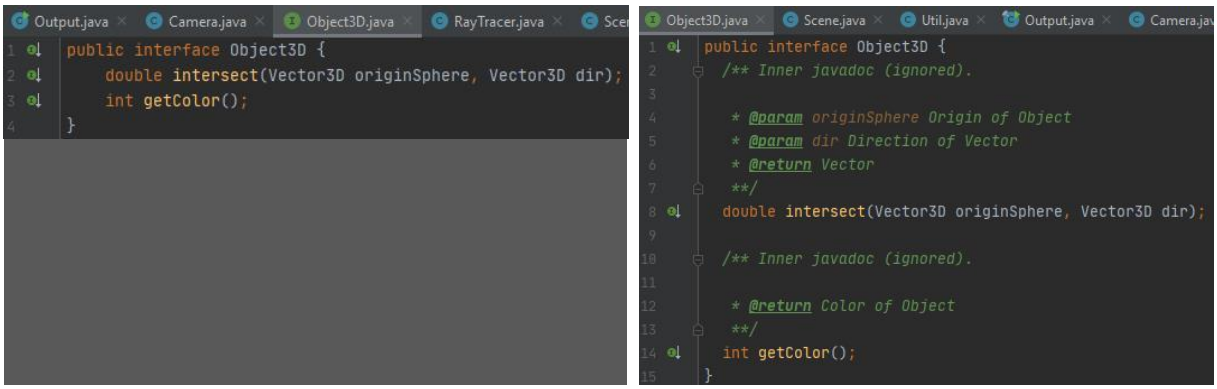


```java
public class Camera {
    private int left = -Output.WIDTH / 2;
    private int right = left * -1;
    private int top = Output.HEIGHT / 2;
    private int bottom = top * -1;

    private Vector3D baseUp = new Vector3D( x: 0,  y: 1,  z: 0);
    private Vector3D eye = new Vector3D( x: 0,  y: 0,  z: -7);
    private Vector3D origin = new Vector3D( x: 0,  y: 0,  z: 0);

    private Vector3D cameraW = eye.subtract(origin).normalize();
    private Vector3D cameraU = baseUp.kreuzproduct(cameraW).normalize();
    private Vector3D cameraV = cameraW.kreuzproduct(cameraU).normalize();

    private double d = top / Math.tan(Math.PI / 4) / 2;
    private Vector3D cameraW_d_negated = cameraW.skalarmultiplication(d * -1);

    public int getLeft() { return left; }

    public void setLeft(int left) { this.left = left; }

    public int getRight() { return right; }

    public void setRight(int right) { this.right = right; }

    public int getTop() { return top; }

    public void setTop(int top) { this.top = top; }

    public int getBottom() { return bottom; }

    public void setBottom(int bottom) { this.bottom = bottom; }
```

```java
public class Camera {
    /** Inner javadoc (ignored).

     * @param left Left margin
     **/
    private final int left = -Output.WIDTH / 2;
    /** Inner javadoc (ignored).

     * @param right Right margin
     **/
    private final int right = left * -1;
    /** Inner javadoc (ignored).

     * @param top Top margin
     **/
    private final int top = Output.HEIGHT / 2;
    /** Inner javadoc (ignored).

     * @param bottom Bottom margin
     **/
    private final int bottom = top * -1;
    /** Inner javadoc (ignored).

     * @param baseUp Base Vector upwards
     **/
    private final Vector3D baseUp = new Vector3D( x: 0,  y: 1,  z: 0);
    /** Inner javadoc (ignored).

     * @param eye Viewers perspective
     **/
    private final Vector3D eye = new Vector3D( x: 0,  y: 0,  z: -7);
    /** Inner javadoc (ignored).
```

Abbildung 1: Vorher-Nachher-Vergleich der Klasse *Camera*

Abbildung 1 und **Abbildung 2** zeigen den Vorher-Nachher-Vergleich der Klassen *Camera* und *Object3D.* Es wurden fehlende Javadoc-Kommentare hinzugefügt, um Variablen und Methoden genauer zu beschreiben. Es wurde mit den von *Checkstyle* empfohlenen @param und @return Tags gearbeitet, um ein konsistentes Schema zu verwenden.



```java
public interface Object3D {
    double intersect(Vector3D originSphere, Vector3D dir);
    int getColor();
}
```

```java
public interface Object3D {
    /** Inner javadoc (ignored).

     * @param originSphere Origin of Object
     * @param dir Direction of Vector
     * @return Vector
     **/
    double intersect(Vector3D originSphere, Vector3D dir);

    /** Inner javadoc (ignored).

     * @return Color of Object
     **/
    int getColor();
}
```

Abbildung 2: Vorher-Nachher-Vergleich der Klasse *Object3D*

Auch das Benennungs-Schema von Variablen ist mir im Zuge der Analyse wichtiger geworden. Variablen müssen eindeutig, leicht verständlich und nach einem klaren Muster benannt werden, um für alle beteiligten Teamkollegen, externe Mitarbeiter, aber auch mein zukünftiges Ich verständlich zu sein. Ein solch überlegtes Benennungs-Schema kann zu einem Clean-Code beitragen und ist essenziell wichtig für die Nachhaltigkeit des Codes.

Die folgende **Abbildung 3** zeigt kleine, aber wichtige Änderungen im Benennungsschema der Klasse *Vector3D.* Eindeutige Variablennamen helfen beim Verständnis und der Einordnung.

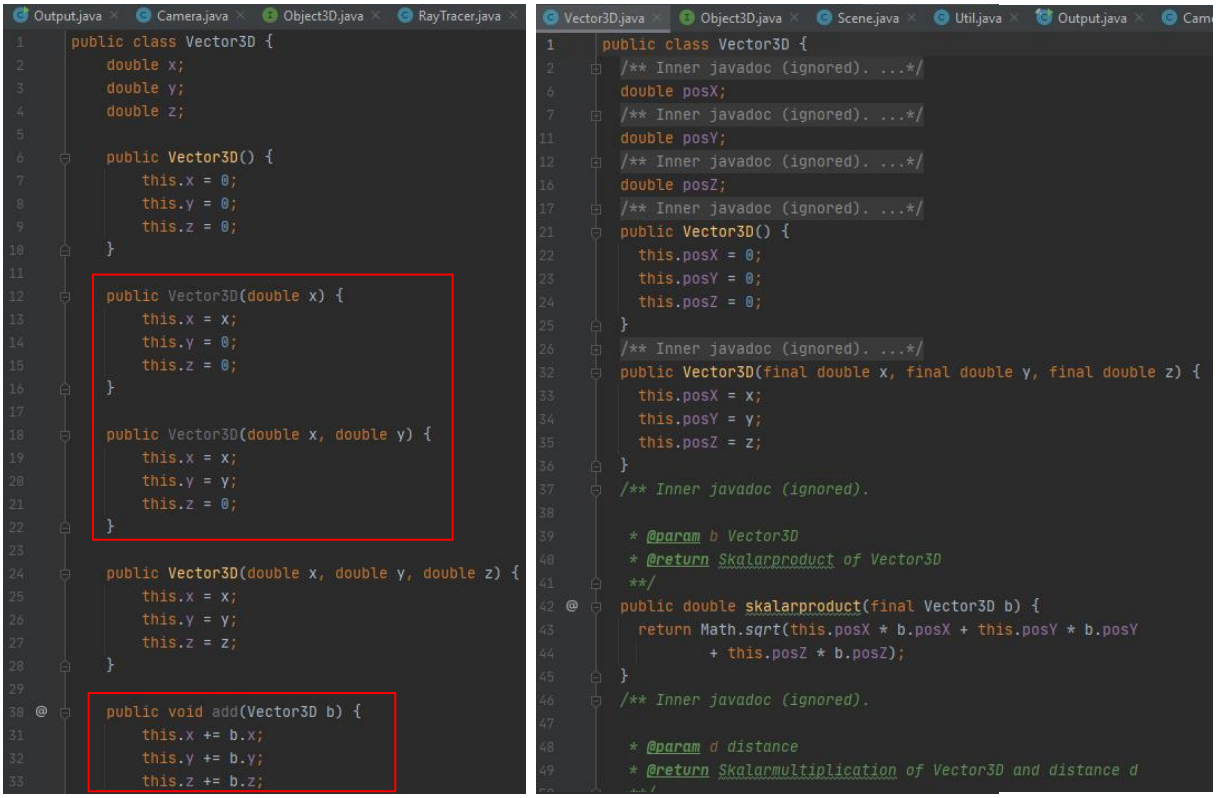


Abbildung 3: Änderungen im Benennungsschema der Klasse *Vector3D.*

Am Beispiel der ausgewählten Klasse wurden die ungenauen Variablen x, y und z umbenannt. Die neuen Variablennamen posX, posY und posZ geben direkt an, dass es sich um Positionen handelt. Außerdem wurden nicht verwendete Konstruktoren (siehe roter Kasten) im *Refactoring*-Prozess entfernt, um den Code auf das Wesentliche zu reduzieren.

Zuletzt wurden noch die Imports angepasst. *Checkstyle* hält dazu an, nach Möglichkeit auf sogenannte Stern-Importe zu verzichten und die Bibliotheken gezielt zu importieren. Beim Import wird nun auf unnötige, nicht verwendete Inhalte der Bibliotheken verzichtet. In **Abbildung 4** wird diese Anpassung am Beispiel der Klasse *Output* veranschaulicht.

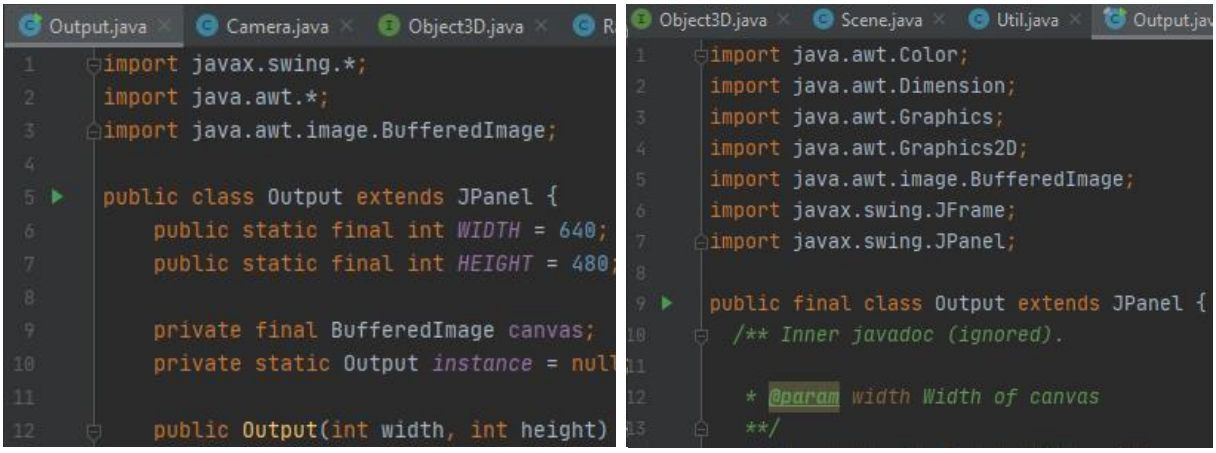


Abbildung 4: Änderungen der Imports in der Klasse *Output*