Checkliste

1. DRY - Don't repeat yourself

   Code sollte, wenn möglich, nicht redundant vorliegen

2. Vermeide negative Bedingungen

   nur Positive Bedingungen sind einheitlich und leichter zu verstehen

3. Vermeide Toten auskommentierten Code

   trägt nicht zur Ausführung des Code bei, sondern sorgt nur für Verwirrung

4. Entferne unnötige Kommentare

   Code sollte vom Methodennamen beschreiben werden

   Kommentare werden aus verschiedenen Gründen häufig nicht dem Code angepasst und werden obsolet, was verwirrt oder einen falschen Eindruck der eigentlichen Funktionalität des Code vermitteln kann

5. Investiere Zeit den besten Namen zu finden

   Namen für Felder, Methoden/Funktionen, Klassen, Interfaces usw sollten einen möglichst aussagekräftigen und beschreibenden Namen erhalten, um etwas über Ihre Funktionalität über den Namen vermitteln zu können

6. Keep it simple, stupid (KISS)

   unnötige Komplexität sollte vermieden werden und einfache Lösungen bevorzugt, wenn es sich anbietet

7. Kapseln komplexer Logikoperationen

   komplexe Algorithmen werden so - gerade auf den ersten Blick - verständlicher

8. Doing vs calling code

   versetzt 3. Person in die Lage Code einfacher nachvollziehen zu können, wenn klarer zwischen ausführendem Code und aufrufendem Code unterschieden wird

9. Magic Numbers vermeiden

   nutze Konstanten, deren Namen klar beschreiben, wofür der eigentliche Wert dient
   Werte und Konstantennamen lassen sich so schnell "refactoren"

10. Überkonfiguration vermeiden

    zu viele Einstellungsmöglichkeiten können ein Programm unnötig komplex machen

11. Richtlinien beibehalten

    unterschiedliche Formatierung des Codes oder Strukturierungsregeln fördern die unübersichtlichkeit des Codes

12. Single responsibility principle (SRP)

    Packages, Klassen, Funktionen, Felder sollten sich immer nur um genau eine Sache kümmern

Durchführung der Checklistenpunkte anhand meines eingereichten TicTacToe Spiels

Die oben aufgeführten Punkte der Checkliste habe ich, nach Analyse meines Codes auf unsauberen Code, versucht abzuarbeiten.

1. DRY - Don't repeat yourself

Die Klasse WinningStreak.java enthält 4 Funktionen zum ermitteln von Siegesreihen (alle Steine einer Art). Die Abfrage, ob genügend Steine dafür gelegt wurden sieht so aus:

if(count == Config.fieldCountWidth)

Diese kam jeweils 1mal in jeder der 4 Methoden vor. Nun wird die Abfrage, wie ein paar andere auch (isPositionOutOfGridBoundary(),isTokenFound) in eine eigene externe Methode ausgelagert und aufgerufen: isWinningSequenceReached( )

2. Vermeide negative Bedingungen

Klasse Start.java
Abfrage ob angegebene Koordinate bereits einen Spielstein enthält und somit belegt ist.

if(!grid.isEmpty(coordinate.position))

Ausgelagert in Methode isCoordinateOccupied( ), was es besser verständlich macht, da hier die positive Abfrage verwendet wird.

3. Vermeide Toten auskommentierten Code

Klasse Start.java enthielt auskommentierten Code zum Testen verschiedener Szenarien während der Entwicklung, den ich nicht mehr entfernt hatte. <-- Ist nun entfernt, da ich diesen wahrscheinlich nicht anpassen würde, wenn sich der Code ändern würde.

4. Entferne unnötige Kommentare

Klasse WinningStreak enthält condition zum Abfragen, ob sich ein Feld diagonal in der Reihe befindet. Um das klarer zu machen, habe ich nur einen Kommentar verwendet, der genau das aussagt. Kommentare lügen sagt Onkel Bob :) (aus verschiedenen Gründen)
Ich habe den Kommentar entfernt und die Abfrage in eine klar benannte Methode ausgelagert: isDiagonalField(position)

5. Investiere Zeit den besten Namen zu finden

Die meißten Namen sind eigentlich schon recht präzise und klar. Hier gab es nur kleine Veränderungen wie Umbenennung von Feld int pos zu position oder boolean won zu isGameWon.

6. Keep it simple, stupid (KISS)

Das ist im nachheinein vieleicht etwas gemogelt, aber ich benutze das, um in der Klasse Start.java (main) für die Bilrschirmausgaben (Spieler1 ist dran, Spieler1 gewonnen...) nicht noch einmal eine extra Printer Klasse zu schreiben. Für die Klasse GridPrinter.java hat das noch Sin gemacht, weil diese das relativ komplexe Grid in die Ausgabe schreibt. Für die main Methode in Start.java halte ich das für übertrieben und habe stattdessen die Ausgabe in jweils einzelne Funktionen ausgelagert.

7. Kapseln komplexer Logikoperationen

Das habe ich teilweise schon in Punkt 1Don't repeat yourself abgehandelt. Hier werden in der Klasse Winningstreak.java Teile des Algorithmus in eigene klar benannte Methode ausgelagert, die nur genau diese eine Sache erledigen. Das erhöht stark die Lesbarkeit, Verständlichkeit und Wartbarkeit des Codes.

8. Doing vs calling code

Klasse Start.java
Main Methode hatte vorher eine starke Mischung von beidem was sich nun zugunsten des Calling code Anteils geändert hat. Jetzt gibt es nur noch Kontrollstrukturen, Methodenaufrufe und Instanziierungen (die ich nicht extra auslagern möchte (KISS)). Die Main ist nun nicht nur kürzer geworden, sondern auch viel verständlicher.

9. Magic Numbers vermeiden

Darauf habe ich auch schon vorher geachtet. Bestimmte Konstante Werte sind in die Config.java ausgelagert worden, um zentral steuern zu können, welche Maße das Spielfeld haben soll.

10. Überkonfiguration vermeiden

Ich wollte zuerst auch andere Konstanten aufnehmen, die z.b. die minimale Länge einer Siegesreihe festlegen, habe mich dann aber dagegen entschieden, da ich gemerkt habe, dass die Algorithmen dann noch einmal an Komplexität zunehmen. Das Spiel ist - wie ich denke - für die Übungsaufgabe schon komplex genug.

11. Richtlinien beibehalten

Das Naming erfolgt nach CamelCase. Aussagekräftige Konstanten werden eingesetzt, um den Code zu verbessern.

12. Single responsibility principle (SRP)

Mit Aussnahme der Start.java Klasse (siehe Punkt 6 KISS) habe ich versucht alle Klassen und Methoden so zu gestalten, dass sie nur genau eine Aufgabe erfüllen. Das passt ganz gut mit einem konsequenten Naming überein, mit dessen Hilfe ich die Funktion einer Klasse oder Methode möglichst aussagekräftig beschreiben kann.
Die Start.java Klasse erfüllt dieses Prinzip nicht, da ich hier nicht nur die einzelnen Methoden Aufrufe sondern auch direkt durch diese Klasse in die Konsole schreibe.