



Technische Universität Berlin

Software and Embedded Systems Engineering Group

Prof. Dr. Sabine Glesner

www.sese.tu-berlin.de    Sekr. TEL 12-4    Ernst-Reuter-Platz 7    10587 Berlin



# Softwaretechnik und Programmierparadigmen WiSe 2023/2024

Prof. Dr. Sabine Glesner  
Julian Klein  
Simon Schwan

## Hausaufgabe Haskell (30 Punkte)

Ausgabe: 04.12.2023

Abgabe: 11.01.2024 bis 23:59 Uhr via TU-Gitlab

Die Hausaufgabe befasst sich mit der Entwicklung eines Bots für das Spiel Death Stacks in Haskell. Zusätzlich zur funktionalen Implementierung in Haskell beinhaltet die Hausaufgabe Anteile der Qualitätssicherung durch Testen und die Verwendung der Versionsverwaltung Git.

Der 1. Abschnitt erläutert die (angepassten) Spielregeln von Death Stacks sowie relevante Notationen für die Hausaufgabe. Anschließend stellt der 2. Abschnitt die Vorgabe und die Schnittstellen, die ihr in der Hausaufgabe implementiert, vor. Zuletzt folgen im 3. Abschnitt Hinweise zum Ablauf der Hausaufgabe.

Viel Erfolg!

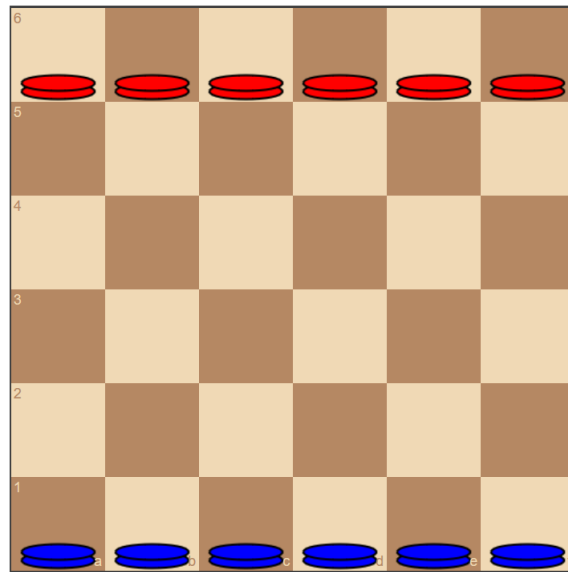
### 1. Death Stacks

**Death Stacks** ist ein abstraktes Strategiespiel, indem Spielfiguren bewegt und zu Stapeln aufgetürmt werden, bis alle Stapel einem der beiden Spieler gehören (d.h. ihm die obere Figur gehört).

#### Regeln

Gespielt wird auf einem auf 6 x 6 Felder verkleinerten Schachfeld mit je 12 Spielsteinen in Rot und Blau. In der Startaufstellung werden jeweils zwei der Figuren auf jedem Feld der letzten Reihe (Rot) und der ersten Reihe (Blau). Rot beginnt.

Die Spieler bewegen abwechselnd einen ihrer Stapel. Dabei können Bewegungen horizontal, vertikal und diagonal erfolgen (so wie die Dame im Schach sich bewegen darf). Die

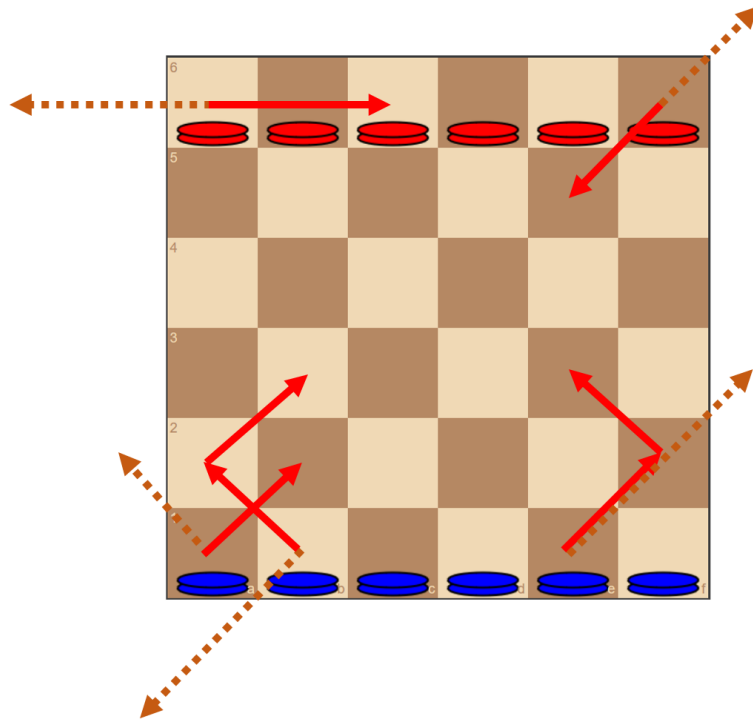


Startposition

Anzahl der Figuren, die man in einem Zug bewegt, bestimmt auch die Anzahl der Felder, die sich bewegt wird. Eine einzelne Figur bewegt sich auf ein benachbartes Feld, ein Stapel aus zwei Figuren wird genau zwei Felder weit bewegt, usw.. Bewegungen werden nicht durch Figuren entlang der Bewegung behindert, d.h. man überspringt Figuren auf dem Weg.

Bewegungen gehen immer entlang einer geraden Linie, es sei denn man würde das Spielfeld verlassen. Dann wird stattdessen die Bewegungsrichtung umgekehrt, d.h. es wird an der Mitte der Randfelder gespiegelt (wie in Physik - Einfallswinkel = Ausfallswinkel). Somit ist jede Bewegung entsprechend der Anzahl der Figuren möglich, mit einer Ausnahme: Der Zustand des Spielfelds muss sich verändern. An dieser Stelle weichen wir von den Original-Regeln ab, die das nicht explizit verbieten. Das heißt zum Beispiel, dass man sich nicht zehn Felder nach links bewegen darf, da man sonst auf dem Ursprungsfeld landet.

Um die Spiegelung an der Feldmitte noch einmal deutlich zu machen, enthält die folgende Abbildung einige Beispiele einer Bewegung gegen die Wand (braun gestrichelt) und die tatsächlich resultierende Bewegung (rot). Bewegungen, die von Randfeldern aus in Richtung des Randes gehen, fangen also direkt mit der Spiegelung an. Da alle diese Bewegungen aber durch andere Bewegungen mit gleichem Ausgang ersetzt werden können, empfiehlt sich die Vorstellung, dass in Randfeldern startende Bewegungen gar nicht in Richtung Rand gehen dürfen.



Ein Stapel gehört dem Spieler, dem die obere Spielfigur gehört. Eine Bewegung kann auch in einem Feld enden, in dem bereits ein Stapel steht. In dem Fall werden einfach die bewegten Figuren in der gleichen Reihenfolge oben auf den Stapel gestellt. Falls der Stapel der schon auf dem Ziel-Feld stand, ein gegnerischer ist, wurde also dieser Turm "gekapert". Die Stapel enthalten eine beliebige Reihenfolge roter oder blauer Figuren, die nicht geändert werden darf.

Bei Bewegungen darf auch nur ein Teil des Stapels bewegt werden. In dem Fall werden Figuren von oben vom Stapel genommen und entsprechend der oben erklärten Regeln um genau die Anzahl Felder bewegt wie Figuren bewegt werden (wie viele stehen bleiben spielt für die Bewegung keine Rolle. Beispiel: Ein Stapel hat fünf Figuren (von oben rot-blau-rot-blau-blau), und man will zwei bewegen, dann muss das Zielfeld zwei Felder entfernt sein. Nach der Bewegung gehören beide Stapel dem roten Spieler, da die roten Figuren oben sind.

Die Too-Tall-Regel besagt, dass, sofern vor dem Zug ein eigener Stapel mit mehr als vier Figuren existiert, dieser (teilweise) bewegt werden muss, so dass auf dem Ausgangsfeld höchstens vier Figuren verbleiben. Auf dem Zielfeld darf dabei ein beliebig hoher Stapel entstehen. Beispiel: Rot ist dran, und auf einem Feld steht ein Stapel mit sieben Figuren, die oberste ist rot. Dann geht der Zug von diesem Stapel aus und dabei dürfen drei, vier fünf sechs oder auch alle sieben Figuren bewegt werden. Weniger als drei geht nicht. An dieser Stelle weichen wir wieder von den Original-Regeln ab indem wir die Too-Tall-Regel zur Pflicht machen (In den Original-Regeln kann ein zu hoher Stapel verbleiben und dies kann vom Gegner angezeigt und bestraft werden).


Die Repeating-State-Rule von Death Stacks ist für die Hausaufgabe nicht relevant.

Ein Spieler hat gewonnen, wenn der Gegner an der Reihe ist und keinen Zug spielen kann.

## Notation

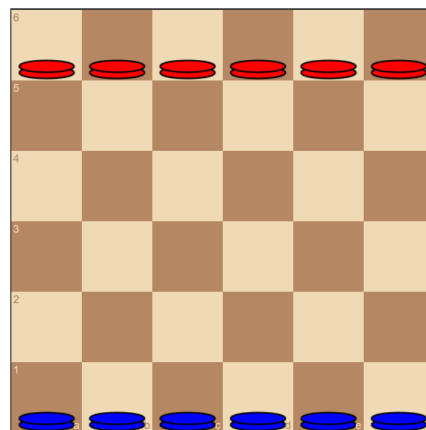
Für die Beschreibung der Spielsituation und Spielzüge verwenden wir eine abgewandelte FEN-Notation (bekannt von Schach). Das heißt, die Spalten werden mit kleinen Buchstaben von a bis f beschriftet, die Reihen mit Zahlen von 1 bis 6. Um ein Feld eindeutig zu bestimmen, wird immer zuerst die Spalte und dann die Zeile angegeben. Dabei bezeichnet **a1** die untere linke Ecke, **f6** die obere rechte. Zu Beginn stehen die blauen Figuren auf den "unteren" Feldern in Reihe 1, die roten auf den "oberen" in Reihe 6.

## Spielbrett

Das Spielbrett wird durch einen String beschrieben, der wie folgt aufgebaut ist: Reihen werden mit "/" getrennt, und innerhalb der Reihen werden die Felder mit "," getrennt. Am Anfang und am Ende des Strings gibt es kein "/", und am Anfang und Ende jeder Reihe auch kein ",". Zuerst wird das Feld **a6** angegeben, zuletzt das Feld **f1**. Innerhalb eines Feldes kann ein Stapel stehen, der aus beliebig vielen roten Figuren "r" und blauen Figuren "b" besteht. Die Figuren werden von oben nach unten dargestellt, zum Beispiel der Stapel  so: "rrb".

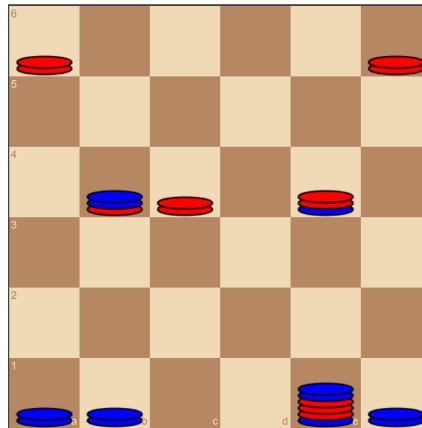
Das Spielbrett zu Beginn wird also wie folgt beschrieben:

`rr,rr,rr,rr,rr,rr/,/,/,/,/,/,/,/,/,/,/,/,/,/,/,/,bb,bb,bb,bb,bb`



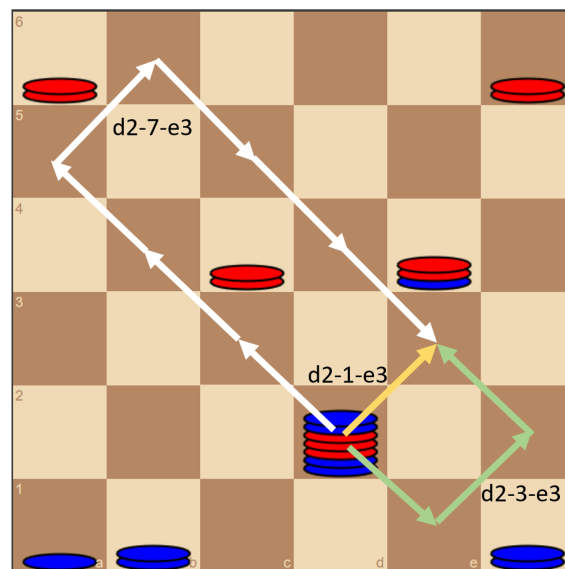
Ein weiteres Beispiel:

`rr,/,/,/,/,/,bbr,rr,,rrb/,/,/,/,/,/,/,bb,bb,,,bbrrrb,bb`



## Züge

Züge werden durch einen String der Form `<start>-<schritte>-<ziel>` beschrieben, wobei `<schritte>` die Entfernung der Bewegung in Anzahl Schritten enthält und Start und Ziel Felder im oben genannten Format sind. Die Angabe der Schritte ist nötig, um die Bewegung eindeutig zu machen, d.h. um anzugeben wie viele Figuren bewegt werden sollen, wenn es mehrere Möglichkeiten gibt, sich von Start nach Ziel zu bewegen. Die folgende Abbildung veranschaulicht das anhand der Bewegung von d2 nach e3:



## 2. Implementierung der Schnittstellen

Das Ziel der Hausaufgabe ist die Entwicklung eines Bots für Death Stacks. Zur Implementierung ist eine Vorlage in Form eines Haskell Stack-Projekts verfügbar. Dieses muss als Grundstruktur zur Implementierung verwendet werden. Zur Implementierung des Death

Stacks-Bots werden 7 Schnittstellen in Form von Funktionssignaturen vorgegeben. Jede Funktion wird anhand ihrer funktionalen Anforderungen bewertet (*FP: funktionale Punkte*), die unten genauer beschrieben werden. Zusätzlich gibt es einen nicht-funktionalen Anteil, indem die Implementierung des Bots getestet werden soll. Der von euch mit Unit-Tests geprüfte Anteil der Implementierung wird anhand der *Haskell Program Coverage (HPC)* ermittelt und bewertet.

Implementiert und testet die folgenden Schnittstellen im *src* und *test* Ordner:

1. Modul Board: `validateFEN :: String -> Bool` 2 FP + 1 CP  
Die Funktion erhält einen String und überprüft diesen anhand des beschriebenen FEN-Formats für Death Stacks (s. Abschnitte Notation und Spielbrett). Dabei gibt die Funktion genau dann `True` zurück, wenn es sich um einen gültigen FEN-String handelt. Die Erreichbarkeit der Spielsituation muss nicht überprüft werden.
2. Modul Board: `buildBoard :: String -> Board` 2 FP + 1 CP  
Die Funktion erhält einen geprüften FEN-String, aus dem der Spielzustand des vorgegebenen Typs `type Board = [[Cell]]` erzeugt werden soll. Dabei entspricht das Element des Zustands `(board!!0)!!0` der Ecke oben links (Feld `a6`). Das Element `(board!!0)!!5` entspricht der Ecke oben rechts (Feld `f6`).  
Hinweis: der `!!`-Operator greift auf den Index einer Liste zu.
3. Modul Board: `path :: Pos -> Dir -> Int -> [Pos]` 4 FP + 1 CP  
Die Funktion erhält drei Argumente (1: Startposition, 2: Richtung, 3. Anzahl der Schritte). Zurückgegeben wird ein Pfad als Liste von Positionen. Der Pfad beginnt in der Startposition und folgt der gegebenen Richtung für die Anzahl der Schritte. Positionen des vorgegebenen Typs `Pos` entsprechen den Bezeichnern des Spielbretts (`a1` bis `f6`). Der Rückgabewert ist sortiert und beinhaltet als erstes Element die Startposition und als letztes Element die Zielposition. Es kann davon ausgegangen werden, dass übergebene Positionen innerhalb des Feldes liegen.
4. Modul Deathstacks: `playerWon :: Board -> Maybe Player` 4 FP + 1 CP  
Die Funktion überprüft, ob und welcher Spieler das Spiel gewonnen hat. Der Zustand (`Board`) entspricht dabei einer erreichbaren Spielsituation. Die Funktion gibt `Nothing` zurück, wenn das Spiel nicht beendet ist. Ansonsten wird der Gewinner zurückgegeben.
5. Modul Deathstacks: `possibleMoves :: Pos -> Cell -> [Move]` 4 FP + 1 CP  
Die Funktion ermittelt für das übergebene Feld (`Cell`) auf der übergebenen Position (`Pos`) alle möglichen Spielzüge und gibt diese als Liste zurück. Dabei wird kein expliziter Spielzustand berücksichtigt. Der übergebene Parameter `Pos` ist eine Position des Spielfeldes und der Parameter `Cell` beinhalten einen Stack. Züge müssen eine Änderung des Zustands bewirken und dürfen nicht mehrfach vorkommen.

6. Modul `Deathstacks`: `isValidMove :: Board -> Move -> Bool`      5 FP + 1 CP

Die Funktion erhält einen Spielzustand (`Board`) und einen Zug des vorgegebenen Typs `Move`. Der Rückgabewert ist genau dann `True`, wenn der Zug im aktuellen Zustand durchgeführt werden kann. Es kann davon ausgegangen werden, dass eine erreichbare Spielsituation übergeben wird. Der Zug enthält eine mögliche Bewegung passend zum Startfeld des Zugs. Die Werte sind konsistent zum übergebenen Spielzustand.

7. Modul `Deathstacks`: `listMoves :: Board -> Player -> [Move]`      2 FP + 1 CP

Die Funktion ermittelt für einen Spieler in einem zulässigen Spielzustand alle durchführbaren Züge. Wie bereits bei `possibleMoves` gilt: Züge müssen eine Änderung des Zustands bewirken und dürfen nicht mehrfach vorkommen.

## Unit-Tests und Testformat

Eure Unit-Tests implementiert ihr im Ordner `test` mithilfe von `Test.HSpec` (wie in den Übungen und den Validierungstests). Damit ihr *Coverage Punkte* durch eure Tests erhalten könnt, müssen eure Tests mithilfe von `stack test deathstacks:units` ausführbar sein. Coverage Punkte werden anteilig zur gesamten erreichten Haskell Program Coverage ( $= \sum \text{covered} / \sum \text{total}$ ) der Module `Board` und `Deathstacks` vergeben. Die anteiligen *Coverage Punkte (CP)* der Schnittstelle erhält man dabei nur, wenn mindestens die Hälfte der zugehörigen funktionalen Punkte erreicht wurden. Die Abdeckung eurer Implementierung durch eure Tests können durch den Befehl `stack test --coverage deathstacks:units` berechnet werden. Eure Tests werden nur gewertet, wenn diese erfolgreich durchlaufen.

## Dos and Don'ts

Folgendes ist zulässig:

- + Implementierung der Schnittstellen in `src/Board.hs` und `src/Deathstacks.hs`
- + Anpassen von `deriving Show` durch das Implementieren der Typklasse `Show`
- + Hinzufügen von Imports (ohne Anpassung der `stack.yaml`)
- + Hinzufügen von Unit-Tests im Testformat unter `test/Spec.hs`
- + Modifiziert **ausschließlich** die Dateien `src/board.hs` und `test/Spec.hs`.

Folgendes ist nicht zulässig:

- Veränderung der Signaturen der Schnittstellen
- Veränderung der vorgegebenen Implementierung (außer Do's)
- Anlegen von neuen Dateien

- Tests, die nicht dem gegebenen Format entsprechen
- Veränderung der Validierungstests
- Veränderung der `stack.yaml` und `package.yaml`
- Netzwerkzugriffe und die Einbettung anderer Programmiersprachen
- Verändern der Datei `gitlab-ci.yml`. Das Ändern führt zu einer Bewertung der gesamten Hausaufgabe von 0 Punkten.
- Fehlschlagende Unit-Tests. Wenn ein Testfall fehlschlägt, gibt es 0 Coverage-Punkte.

### 3. Ablauf der Hausaufgabe

Die Hausaufgabe ist in zwei Phasen aufgeteilt. In Phase 1 sollen die **Board**-Funktionen implementiert werden und in Phase 2 sollen die **Deathstacks**-Funktionen implementiert werden. Zwischen den Phasen führen wir einmalig einen Zwischentest mit eurem aktuellen Stand durch, der nicht in die Benotung der finalen Abgabe eingeht. Der Zwischentest dient lediglich als Feedback und liefert euch eine Einschätzung über die Qualität eurer Implementierung.

Zur Bearbeitung der Aufgabe müsst ihr das Gitlab der TU benutzen. Dafür haben wir für euch persönliche Repositories angelegt, auf die nur ihr Zugriff habt. Dort gibt es einen `main`-Branch und einen `development`-Branch. Ihr arbeitet ausschließlich auf dem `development`-Branch.

Wir nutzen Runner um einen **CI-Prozess** zu simulieren. Der Runner testet euren aktuellen Stand automatisch nach dem Hochladen ins Git-Repository und gibt euch Feedback. Zum Testen werden die Validierungstests ausgeführt, die ihr auch in der Vorgabe findet. Ihr bekommt für eine Schnittstelle am Ende nur Punkte, wenn die Validierungstests erfolgreich durchlaufen. Außerdem führt der Runner auch eure eigenen Unit-Tests aus und teilt euch mit ob, die Coverage berechnet werden kann. Wir verteilen für die Coverage nur Punkte wenn alle Unit-Tests erfolgreich ausgeführt werden können. Mit Hilfe unserer Runner könnt diese Konditionen also vorab prüfen und es ist transparent, ob bzw. welche Teile eurer Abgabe bewertet werden.

Damit ihr eure Implementierung testen könnt, stellen wir euch zusätzlich einen Webserver zum Download auf ISIS zur Verfügung. Hinweise zur Verwendung findet ihr in der Readme. Mithilfe von `stack exec deathstacks` erzeugt ihr eine ausführbare Datei eures Bots, bei dem in der `main`-Methode im Ordner `app` ein zufälliger Zug aus der Liste aller Züge ausgewählt wird. Die Züge werden durch eure `listMoves`-Funktion berechnet, die erst in der zweiten Phase implementiert wird.



**1. Phase bis zum 18.12.23** In der ersten Phase sollen die **Board-Funktionen** implementiert werden. Diese Funktionen können auch in der zweiten Phase bearbeitet bzw. geändert werden. Am Ende der ersten Phase werden wir im Verlauf des Tages (18.12.23) die aktuelle Version auf eurem **development**-Branch mit dem Zwischentest bewerten. Dieser einmalige Test geht nicht in die finale Bewertung der Hausaufgabe ein. Dennoch erhaltet ihr eine Bewertung der Funktionen, was euch Feedback zum aktuellen Stand eurer Arbeit liefert. Der Test wird durchgeführt auf dem **main**-Branch eures Repositories durchgeführt und die vollständige Bewertung eurer Implementierung findet ihr in der Ausgabe des Gitlab-Runners. **WICHTIG:** Euer aktueller Stand wird von uns nur bewertet, wenn er kompilierbar ist. Details dazu findet ihr vorab in den Ausgaben der Gitlab-Runner. Es liegt also in eurer Verantwortung eine bewertbare Lösung am 18.12.23 in eurem **development**-Branch zu haben. Es gibt nur einen Versuch der Zwischenbewertung.

**2. Phase** Nach dem Zwischentest sollen die **Deathstacks-Funktionen** implementiert werden. Auch hier bieten wir die Möglichkeit euren aktuellen Stand mit unserem CI-Prozess zu testen. Die Vorlage für die neuen Funktionen wird durch ein öffentliches Repository verfügbar sein. Eure Aufgabe ist es dann einen Merge in euer Projekt durchzuführen. Details zu diesem Schritt werden wir dann durch eine Ankündigung auf Isis bekannt geben.

**Finale Abgabe** Eure Bearbeitungszeit endet am 11.01.24 um 23:59. Die **developer**-Rechte werden euch dann automatisch entzogen und ihr seid nicht mehr in der Lage euer Projekt zu verändern. Wir bewerten anschließend den letzten Stand auf dem **development**-Branch. Die finale Bewertung wird analog zur Bewertung der Zwischentests durchgeführt und ihr könnt das Ergebnis im Output des Runners auf dem **main**-Branch einsehen.

Abbildung 0.2 zeigt den Ablauf der Bearbeitungszeit als Übersicht. Die gesamte Bearbeitungszeit beinhaltet 25 volle Werktage. Diese Zeit beinhaltet mögliche Krankheitstage. Im Krankheitsfall gibt es daher keine Verlängerung. Eine Abmeldung von der Hausaufgabe ist ab einer Krankheit von 5 Werktagen (Mo. - Fr.) möglich.

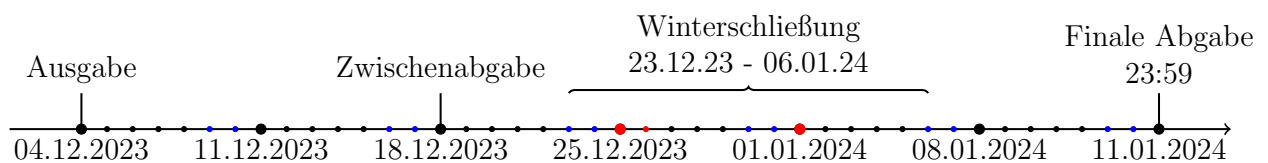


Abbildung 0.2: Zeitlicher Ablauf der Hausaufgabe. 26 Tage Bearbeitungszeit (**Feiertage** und **Wochenenden**).