

☐ Gruppe 1 (J. Heinzelreiter)☐ Gruppe 2 (M. Hava)Name: Niklas VestAufwand [h]: 20☒ Gruppe 3 (P. Kulczycki)

Übungsleiter/Tutor: _____

Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (25 P + 60 P + 15 P)	70%	95%	50%

Beispiel 1: Schach (src/chess/)

Implementieren Sie in C++ unter Anwendung des objektorientierten Programmierparadigmas eine einfache Version des Brettspiels „Schach“. Beachten Sie dabei die folgenden Anforderungen und Hinweise.

Für alle Spielfiguren gibt es eine gemeinsame, abstrakte Basisklasse chessman, die eine Schnittstelle für mindestens die folgenden Funktionalitäten bietet:

1. sie liefert die Farbe einer Spielfigur,
2. sie liefert eine symbolische Darstellung einer Spielfigur in Form eines ASCII-Zeichens,
3. sie beantwortet die Frage, ob eine Spielfigur „essentiell“ ist (ein Verlust einer essentiellen Spielfigur bedeutet das Ende des Spiels) und
4. sie beantwortet die Frage, ob sich eine Spielfigur auf einem Schachbrett von Position $\{z_0, s_0\}$ auf Position $\{z_1, s_1\}$ bewegen kann.

Alle konkreten Spielfiguren (wie z.B. Bauer und Springer) sollen dann, insbesondere für die letzte Funktionalität, eigene Implementierungen besitzen. Erstellen Sie dazu von chessman abgeleitete Klassen für die typischen Schachfiguren.

1. Der König („king“ – K) kann sich in alle Richtungen um ein Feld bewegen. Das Zielfeld darf aber nicht durch eine Spielfigur der eigenen Farbe besetzt sein. Der König ist eine essentielle Spielfigur.
2. Die Dame („queen“ – Q) kann sich beliebig weit diagonal, horizontal oder vertikal auf dem Spielbett bewegen, solange keine andere Spielfigur im Weg ist.
3. Der Läufer („bishop“ – B) kann sich beliebig weit diagonal auf dem Spielbett bewegen, solange keine andere Spielfigur im Weg ist.
4. Der Turm („rook“ – R) kann sich beliebig weit horizontal oder vertikal auf dem Spielbett bewegen, solange keine andere Spielfigur im Weg ist.
5. Der Springer („knight“ – N) kann sich um jeweils zwei Felder in eine Richtung (horizontal oder vertikal) und gleichzeitig um ein Feld in die andere Richtung (vertikal oder horizontal) bewegen. Dabei kann er beliebig andere Spielfiguren überspringen.
6. Der Bauer („pawn“ – P) kann sich um ein Feld nach vor bewegen. Bei seinem ersten Zug können dies auch zwei Felder nach vor sein. Der Bauer kann nur seitlich schlagen.

Alle Spielfiguren können auf einer feindlichen Spielfigur landen, was diese aus dem Spiel befördert und zum eigenen Sieg beitragen kann. Die Spielfiguren befinden sich auf einem Spielbrett was z.B. mit einer Klasse chessboard implementiert werden kann. Dieses Spielfeld muss mindestens die folgenden Funktionalitäten aufweisen:

1. Es muss vorgesehen sein, ein beliebig großes Spielbrett anlegen zu können (jedoch mit einer sinnvollen Mindestgröße).
2. Es muss erkannt werden, wann ein Spiel beendet ist (nämlich dann, wenn einer der Spieler eine essentielle Figur verloren hat).

3. Für jede Position muss die darauf befindliche Figur ermittelt werden können.
4. Der aktuellen Spieler sowie die Größe des Spielfeldes müssen ebenfalls abgefragt werden können.
5. Die folgenden Fragen müssen für jedes Feld des Spielbretts beantwortet werden können:
 - a. Kann es „befahren“ werden bzw. ist es frei?
 - b. Kann eine Spielfigur mit einer bestimmten Farbe geschlagen werden?
 - c. Kann eine Spielfigur einer bestimmten Farbe darauf landen? Ist es also frei oder kann dort eine andere Figur geschlagen werden?
 - d. Kann man auf diesem Feld eine Figur momentan „in die Hand nehmen“? Also ist dort eine Figur und wenn ja, von der aktuell am Zug befindlichen Farbe?
 - e. Kann man die aktuell aktive („in der Hand befindliche“) Figur an diesem Feld abstellen? Darf die Figur also von ihrem aktuellen Startpunkt aus dort hinfahren und kann auch dortbleiben?
6. Kann man von einem Feld $\{z_0, s_0\}$ auf ein Feld $\{z_1, s_1\}$ fahren?
7. Folgende Aktionen müssen außerdem möglich sein:
 - a. Die Spielfigur an einer bestimmten Stelle aktivieren, also „in die Hand nehmen“.
 - b. Die aktive Spielfigur auf einem bestimmten Feld wieder abstellen (sofern das erlaubt ist).
8. Der aktuelle Spielstand (das Spielbrett mit den Spielfiguren) muss, genau so wie unten gezeigt, als ASCII-Grafik auf der Konsole ausgegeben werden.

Erledigen Sie nun die folgenden Aufgabenstellungen:

(a) Implementieren Sie alle klassischen Schachfiguren und testen Sie deren Bewegungsmuster ausführlich. Gehen Sie in der Lösungsidee auch darauf ein, wieviel Speicher pro Spielfigur benötigt wird.

(b) Schreiben Sie ein Hauptprogramm, mit dem zwei Spieler interaktiv Schach spielen können. Dabei müssen alle Zugmöglichkeiten der jeweils aktiven Spielfigur angezeigt werden.

Zwei Beispiele: Im linken Beispiel ist der weiße Springer (N) in Zeile 1, Spalte b aktiv. Im rechten Beispiel ist die weiße Dame (Q) in Zeile 1, Spalte d, die einen gegnerischen schwarzen Bauern [p] in Zeile 4, Spalte d schlagen könnte, aktiv.

	a b c d e f g h			a b c d e f g h	
	---+-----+---			---+-----+---	
8	r n b q k b n r	8		8	r n b q k b n r
7	p p p p p p p p	7		7	p p p p * p p p
6	. * . * . * . *	6		6	. * . * . * . *
5	* . * . * . * .	5		5	* . * . * . * [.]
4	. * . * . * . *	4		4	[.] * . [p] . * [.] *
3	[*] . [*] . * . * .	3		3	* [.] P [.] P [.] * .
2	P P P P P P P P	2		2	P P [.] [*] [.] P P P
1	R (N) B Q K B N R	1		1	R N B (Q) K B N R
	---+-----+---			---+-----+---	
	a b c d e f g h			a b c d e f g h	

Abbildung: Die unterschiedlichen Farben der Spielfiguren sind als Groß- bzw. Kleinbuchstaben, die unterschiedlichen Farben der Felder durch Punkt bzw. Stern dargestellt. Die aktive Spielfigur ist in runde Klammern gesetzt. Die Felder, die die aktive Spielfigur betreten kann, sind in eckige Klammern gesetzt.

(c) Schreiben Sie ein alternatives Hauptprogramm (oder eine über die Kommandozeile auswählbare Variante), das bzw. die es erlaubt, einen zufälligen, und natürlich nur aus gültigen Zügen bestehenden, Spielablauf zu generieren.

Ausarbeitung 08

Niklas Vest

December 16, 2018

1 Schach

1.1 Lösungsidee

Da die Vorgaben in der Angabe ziemlich genau sind möchte ich hier nur auf Einzelheiten meiner Lösungsidee eingehen und abschließend Verbesserungsvorschläge abgeben, welche aufgrund meines Zeitmangels und Restriktionen der Angabe nicht realisiert werden konnten / durften.

1.1.1 Designentscheidungen

Ein Schachspiel besteht bei mir aus Interaktionen der Klassen *Game*, *Board* und *Chessman*. *Game* dient als Interaktionsschnittstelle zwischen "Bibliotheksbenunder" und dem tatsächlichen Schachspiel, indem es Benutzern den Input- und Output-Stream für das Einlesen von Zügen und Ausgeben von Spielzuständen spezifizieren lässt. Die Klasse *Board* kümmert sich um alle Spieldetails, die sich auf das Schachbrett beziehen [sic!]. *Chessman* dient als Abstrakter gemeinsamer Nenner, über den mittels dynamischer Bindung u.a. festgestellt werden kann, ob eine konkrete Schachfigur einen vom Benutzer geforderten Zug unterstützt. Ursprünglich sollte die Trennung zwischen Schachbrett und -figuren sehr strikt ausfallen, um eine möglichst weitgehende Kapselung der Klassen zu schaffen. Da Schachfiguren aber für die Entscheidung, ob ein Zug korrekt / möglich ist, einen Kontext brauchen, musste die Schnittstelle zwischen den Kommunikationspartnern erst recht wieder vergrößert werden. Bauern können sich z.B. diagonal bewegen, wenn sie einen Gegner schlagen, sind aber sonst auf vertikale Bewegungen beschränkt.

Um ein Spiel "voranzutreiben" stellt die Klasse *Game* zwei Methoden zur Verfügung:

`void next_move(std::istream&, std::ostream&)` (1) und
`void next_move(std::function<Position()>, std::ostream&)` (2).

(1) liest einen Zug von einem input stream und (2) verwendet eine Funktion, um sich Züge zu generieren. Dies wurde in Voraussicht auf Aufgabe (c) gemacht. Um einen Spielverlauf künstlich zu erzeugen, kann man nun eine Funktion an *next_move* übergeben, welche für jeden Aufruf einen neuen Zug zufällig generiert.

Für das Spielbrett verwende ich einen Vektor von Vektoren von `Chessman` (smart)pointern, was den Zugriff auf spezielle Felder angenehm und übersichtlich gestaltet und zusätzlich die Speicherverwaltung erleichtert. Somit reduziert sich z.B. die Frage "ist ein Feld leer" auf den Fall "Matrix enthält an dieser Stelle einen Null-pointer".

1.1.2 Spielablauf

- Ein Spieler (zu Beginn "BLACK") wird aufgefordert, eine Schachfigur zu wählen.
- Ist das gewählte Schachfeld eine ungültige Auswahl, wird erneut gefragt, sonst die Schachfigur gewählt.
- Der spieler wird aufgefordert, ein Zielfeld zu wählen.
- Ist das gewählte Zielfeld kein gültiges Ziel, wird erneut gefragt, sonst wird der Zug "vervollständigt" und ggf. eine gegnerische Figur geschlagen.
- Der "aktive Spieler" wird gewechselt und der Selbe ablauf wird wiederholt bis ein Spieler den König verliert.

1.1.3 Verbesserungsvorschläge

Ich bin mir darüber im klaren, dass die Aufgabe das Verständnis für Objektorientierung vertiefen soll, weswegen die Aufteilung in eine abstrakte Klasse *Chessman* und konkrete Unterklassen für alle Schachfiguren explizit gefordert ist. Ich hätte aber vorgezogen, eine einzelne, konkrete Klasse *Chessman* mit einem Attribut vom Typ *MovementPattern* auszustatten. Diese *MovementPatterns* haben dann eine horizontale, diagonale etc. Ausprägung und können mittels Komposition (nicht Vererbung!) ineinander geschachtelt werden, um so z.B. aus dem Bewegungsmuster von Turm und Läufer das Muster der Dame zu bauen. So kann man einem "linearen Wachstum der Klassenhierarchie" - wenn auch nur minimal - entgegenwirken.

1.2 Implementierung

Listing 1: Types.hpp

```
1 #pragma once
2
3 #include <functional>
4 #include <map>
5 #include <memory>
6
7 /*
8  * This file contains forward declarations for types that are important for all
9  * classes.
10  * That way cyclic dependencies can be avoided.
11  */
```

```

11
12 namespace Chess {
13
14 class Game;
15
16 class Board;
17
18 class Chessman;
19
20 using ChessmanPtr = std::shared_ptr<Chessman>;
21
22 /**
23  * Represents a tile on the chess board.
24  */
25 using Position = std::pair<unsigned int, char>;
26
27 std::istream &operator>>(std::istream &is, Position &pos);
28
29 using Move = std::pair<Position, Position>;
30
31 /**
32  * The characters surrounding the representation
33  * of a chess tile / chessman.
34  */
35 using Highlight = std::pair<char, char>;
36
37 /**
38  * Because booleans felt too hacky.
39  */
40 enum class Color
41 {
42     BLACK, WHITE
43 };
44
45 std::ostream &operator<<(std::ostream &os, Color color);
46
47 } // namespace Chess

```

Listing 2: Game.hpp

```

1 #pragma once
2
3 #include <iostream>
4 #include "Types.hpp"
5 #include "Board.hpp"
6
7 namespace Chess {
8
9 class Game final
10 {
11 public: // methods
12
13     /**
14      * Initializes a game by setting up a chessboard with chessmen.
15      */
16     explicit Game(unsigned side_length = 8)
17         : _board{*this, side_length}
18     {}

```

```

19
20     /**
21      * Reads a move from the supplied input stream. if the move is valid,
22      * the game is advanced, otherwise the stream is read until it yields
23      * a valid move.
24      * @param is The input stream to read the new move from.
25      * @param os The output stream to write the new state of the game to.
26      */
27     void next_move(std::istream &is, std::ostream &os);
28
29     /**
30      * Calls the supplied function to retrieve a move. if the move is valid,
31      * the game is advanced, otherwise the function is recalled until it yields
32      * a valid move.
33      * @param get_pos A function which returns a move.
34      * @param os The output stream to write the new state of the game to.
35      */
36     void next_move(std::function<Position()> get_pos, std::ostream &os);
37
38     /**
39      * @return True if one player has lost an essential chessman, False otherwise.
40      */
41     bool is_over() const;
42
43     /**
44      * @return The player who may make the next move in the game.
45      */
46     Color get_current_player() const;
47
48 private: // methods
49
50     /**
51      * Writes a prompt to the supplied output stream.
52      * @param prompt The message to use for the prompt.
53      * @param os The output stream to write the prompt to.
54      */
55     void _prompt(const std::string &prompt, std::ostream &os);
56
57 private: // members
58
59     /**
60      * The player "with the action".
61      * (https://www.linguee.de/deutsch-englisch/uebersetzung/spieler+der+am+zug+ist.html)
62      */
63     Color _current_player{Color::BLACK};
64
65     /**
66      * The game board associated with this game.
67      */
68     Board _board;
69 };
70
71 }

```

Listing 3: Game.cpp

```
1 #include "Game.hpp"
```

```

2
3 #include <functional>
4
5 namespace Chess {
6
7 void Game::next_move(std::istream &is, std::ostream &os)
8 {
9     auto read_from_is = [&is]() {
10         Position x;
11         is >> x;
12         return x;
13     };
14     next_move(read_from_is, os);
15 }
16
17 void Game::next_move(std::function<Position()> get_pos, std::ostream &os)
18 {
19     _board.render(os);
20
21     // ===== picking ===== //
22     _prompt("Chose a chessman to move (row col)", os);
23
24     Position from = get_pos();
25
26     while (!_board.is_pickable(from)) {
27         os << "Invalid chessman, try another one!" << std::endl;
28         _prompt("Chose a chessman to move (row col)", os);
29         from = get_pos();
30     }
31     _board.pick(from);
32
33     _board.render(os);
34
35     // ===== dropping ===== //
36     _prompt("Move selected chessman to", os);
37
38     Position to = get_pos();
39
40     while (!_board.is_current_chessman_droppable_at(to)) {
41         os << "Can not move selected chessman to specified position!" << std
42         ::endl;
43         _prompt("Move selected chessman to", os);
44         to = get_pos();
45     }
46     _board.drop(to);
47
48     if (!is_over()) { // dont flip after game is over so the winner can be queried!
49         _current_player = (get_current_player() == Color::WHITE ? Color::
50         BLACK : Color::WHITE);
51     } else {
52         os << "Player " << get_current_player() << " won!" << std::endl;
53     }
54 }
55
56 bool Game::is_over() const
57 {
58     return _board.essential_lost();
59 }

```



```

57 }
58
59 void Game::_prompt(const std::string &prompt, std::ostream &os)
60 {
61     os << get_current_player() << ' ' << prompt << "> " << std::flush;
62 }
63
64 Color Game::get_current_player() const
65 {
66     return _current_player;
67 }
68
69 }

```

Listing 4: Board.hpp

```

1  #pragma once
2
3  #include <vector>
4  #include "Types.hpp"
5
6  namespace Chess {
7
8  class Board final
9  {
10 public: // methods
11
12     /**
13      * Creates a new chess board for a specific game.
14      * @param game The game with which this board is associated.
15      * @param side_length The desired side length of this board.
16      */
17     Board(const Game &game, unsigned side_length);
18
19     /**
20      * @param pos The indices of the desired tile.
21      * @return The chessman at the specified position. If
22      *         that tile is empty, the result is equal
23      *         to nullptr.
24      */
25     ChessmanPtr get_chessman_at(const Position &pos) const;
26
27     /**
28      * @return The side length of the chessboard.
29      */
30     unsigned get_side_length() const;
31
32     /**
33      * @param pos The position of the potentially beatable chessman.
34      * @return True if the Chessman at the specified tile is owned
35      *         by "the opposing player" and the currently selected
36      *         chessman could move to _pos_.
37      */
38     bool is_beatable(const Position &pos) const;
39
40     /**
41      * @param p_chessman The chessman to move.
42      * @param pos The move destination.

```

```

43  * @return True if the chessman (who is assumed to be the currently
44  *         selected one) can move to the specified position, beating
45  *         the chessman at the destination if necessary.
46  * @throws std::invalid_argument If the supplied chessman is NOT the
47  *         currently selected one.
48  */
49  bool is_move_valid(const ChessmanPtr &p_chessman, const Position &pos)
    const;
50
51  /**
52   * @param pos The source tile from which to move.
53   * @param pos The move destination.
54   * @return True if the chessman (who is assumed to be the currently
55   *         selected one) can move to the specified position, beating
56   *         the chessman at the destination if necessary.
57   */
58  bool is_move_valid(const Position &from, const Position &to) const;
59
60  /**
61   * @param pos The position of the chessman to pick up (/ "select").
62   * @return True if there is a (friendly) chessman at the specified
63   *         position and there is at least one possible move for
64   *         this chessman.
65   */
66  bool is_pickable(const Position &pos) const;
67
68  /**
69   * @param pos The tile to check for availability .
70   * @return Returns the most recently picked up chessman can
71   *         be dropped at the specified position. (i. e. No
72   *         other chessman obstruct the path and the specified
73   *         tile is not occupied by a friendly chessman.)
74   */
75  bool is_current_chessman_droppable_at(const Position &pos) const;
76
77  /**
78   * Selects the chessman at the specified position.
79   * If that chessman is not selectable, no action is taken.
80   * @param pos The position of the chessman to select.
81   */
82  void pick(const Position &pos);
83
84  /**
85   * Attempts to drop the currently selected chessman at the
86   * specified position. If that is not possible, no action is taken.
87   * @param pos The tile where the currently selected chessman
88   *         should be dropped.
89   */
90  void drop(const Position &pos);
91
92  /**
93   * Writes the chess board to the supplied output stream.
94   * @param os The output stream to write to.
95   */
96  void render(std::ostream &os) const;
97
98  /**

```

```

99      * @return True if the king has been beaten.
100     */
101     bool essential_lost() const;
102
103     /**
104      * @return True if the player currently at action has not
105      *         moved before.
106     */
107     bool is_first_move() const;
108
109     private: // methods
110
111     /**
112      * Writes the column decoration (letters) to the supplied
113      * output stream.
114      * @param os The output stream to write to.
115     */
116     void _render_col_chromium(std::ostream &os) const;
117
118     /**
119      * Writes a horizontal chess border to the supplied
120      * output stream.
121      * @param os The output stream to write to.
122     */
123     void _render_hor_border(std::ostream &os) const;
124
125     /**
126      * @param row The row of the tile.
127      * @param pos The column of the tile.
128      * @return The highlighting for the specified tile.
129     */
130     Highlight _get_highlight(Position pos) const;
131
132     /**
133      * Writes a part of the chessboard to the supplied output.
134      * @param os The output stream to write to.
135      * @param tile The character representation of the tile.
136      * @param p_chessman The chessman to render. If p_chessman is a
137      *                   nullptr, the character representation is rendered.
138      * @param highlight The highlight for the given tile.
139     */
140     void _render_highlighted(std::ostream &os,
141                             char tile,
142                             const ChessmanPtr &p_chessman,
143                             const Highlight &highlight) const;
144
145     private : // members
146
147     /**
148      * If this boi is set to true, the game can not be advanced any further.
149     */
150     bool _essential_lost {false};
151
152     /**
153      * The side length of the board.
154     */
155     unsigned _side_length;

```

```

156
157      /**
158       * The game this chessboard is associated with.
159       */
160      const Game &parent_game;
161
162      /**
163       * The position of the currently selected chessman.
164       * @Note If this is {0,^0}, no chessman is selected.
165       */
166      Position _current_selection {};
167
168      /**
169       * The matrix representing the chessboard.
170       */
171      std::vector<std::vector<ChessmanPtr>> _matrix{};
172
173      /**
174       * A map for bookkeeping which player has already moved a chessman.
175       * This is only used for the pawns first move.
176       */
177      std::map<Color, bool> _first_move_map {
178          {Color::WHITE, true},
179          {Color::BLACK, true}
180      };
181 };
182
183 } // namespace Chess

```

Listing 5: Board.cpp

```

1  #include "Board.hpp"
2  #include "Chessman.hpp"
3
4  namespace Chess {
5
6  std::istream &operator>>(std::istream &is, Position &pos)
7  {
8      is >> pos.first >> pos.second;
9      pos.first--; // chess starts with row 1, program logic with row 0
10     return is;
11 }
12
13 std::ostream &operator<<(std::ostream &os, Color color)
14 {
15     return os << '[' << (color == Color::WHITE ? "White" : "Black") << ']';
16 }
17
18 /**
19  * @param c The character to "convert".
20  * @return The 0-indexed column represented by the supplied character.
21  * This is mainly used for transformations between indices and
22  * chess board columns.
23  */
24 static int char_to_col(char c)
25 {
26     return std::tolower(c) - 'a';
27 }

```

```

28
29 Board::Board(const Game &game, unsigned side_length)
30     : _side_length{side_length}, _parent_game{game}
31 {
32     if (side_length < 8) {
33         throw std::invalid_argument("Chess board side length must be greater
34             than 8!");
35     }
36     // Initialize the chess board matrix to the correct dimensions
37     _matrix.resize(get_side_length(), std::vector<ChessmanPtr>());
38     for (auto &row : _matrix) {
39         row.resize(get_side_length(), ChessmanPtr());
40     }
41
42     // write special black figures to first row
43     _matrix[0][0] = std::make_shared<Rook>(Color::BLACK);
44     _matrix[0][1] = std::make_shared<Knight>(Color::BLACK);
45     _matrix[0][2] = std::make_shared<Bishop>(Color::BLACK);
46     _matrix[0][3] = std::make_shared<King>(Color::BLACK);
47     _matrix[0][4] = std::make_shared<Queen>(Color::BLACK);
48     _matrix[0][5] = std::make_shared<Bishop>(Color::BLACK);
49     _matrix[0][6] = std::make_shared<Knight>(Color::BLACK);
50     _matrix[0][7] = std::make_shared<Rook>(Color::BLACK);
51
52     // write black pawns to 2nd row
53     for (auto &tile : _matrix[1]) {
54         tile = std::make_shared<Pawn>(Color::BLACK);
55     }
56
57     // some complex mathematical computation involving calculus,
58     // polar representation of the cartesian coordinate system
59     // and one whole lot of cookies.
60     const std::size_t last = get_side_length() - 1;
61     const std::size_t snd_to_last = last - 1;
62
63     // write white pawns to 2nd to last row
64     for (auto &tile : _matrix[snd_to_last]) {
65         tile = std::make_shared<Pawn>(Color::WHITE);
66     }
67
68     // write special white figures to last row
69     _matrix[last][0] = std::make_shared<Rook>(Color::WHITE);
70     _matrix[last][1] = std::make_shared<Knight>(Color::WHITE);
71     _matrix[last][2] = std::make_shared<Bishop>(Color::WHITE);
72     _matrix[last][3] = std::make_shared<King>(Color::WHITE);
73     _matrix[last][4] = std::make_shared<Queen>(Color::WHITE);
74     _matrix[last][5] = std::make_shared<Bishop>(Color::WHITE);
75     _matrix[last][6] = std::make_shared<Knight>(Color::WHITE);
76     _matrix[last][7] = std::make_shared<Rook>(Color::WHITE);
77 }
78
79 ChessmanPtr Board::get_chessman_at(const Position &pos) const
80 {
81     auto row = pos.first;
82     auto col = char_to_col(pos.second);
83     ChessmanPtr p_chessman = nullptr;

```

```

84
85     if (row < get_side_length() && col < get_side_length()) {
86         p_chessman = _matrix[row][col];
87     }
88     return p_chessman;
89 }
90
91 unsigned Board::get_side_length() const
92 {
93     return _side_length;
94 }
95
96 bool Board::is_beatable(const Position &pos) const
97 {
98     bool beatable = false;
99     auto p_chessman = get_chessman_at(pos);
100     if (p_chessman != nullptr) {
101         // the chessman is owned by the opponent
102         beatable = p_chessman->color != _parent_game.get_current_player();
103         // the chessman can be reached from _current_selection
104         beatable = beatable && is_move_valid(_current_selection, pos);
105     }
106     return beatable;
107 }
108
109 bool Board::is_move_valid(const ChessmanPtr &p_chessman, const Position &pos)
110     const
111 {
112     if (get_chessman_at(_current_selection) != p_chessman) {
113         // design flaw, but meh :/
114         throw std::invalid_argument("The supplied chessman is not selected.");
115     }
116     // verifies that there is no friendly chessman @ pos
117     auto target_cm = get_chessman_at(pos);
118     bool noFriendly = target_cm == nullptr ||
119         (target_cm != nullptr && target_cm->color !=
120         _parent_game.get_current_player());
121     // and the chessman supports the move
122     return noFriendly && p_chessman != nullptr && p_chessman->can_move(*this,
123         {_current_selection, pos});
124 }
125
126 bool Board::is_move_valid(const Position &from, const Position &to) const
127 {
128     // this is the second M A J O R design flaw. IMO you should never need a const
129     cast
130     // except when using delegation for const overloads. The delegation here was
131     necessary
132     // due to improper planning.
133     const auto &buff = _current_selection;
134     const_cast<Board *>(this)->_current_selection = from;
135     bool v = is_move_valid(get_chessman_at(from), to);
136     const_cast<Board *>(this)->_current_selection = buff;
137     return v;
138 }

```

```

135
136 static Position ints_to_pos(unsigned row, unsigned col)
137 {
138     return { row, static_cast<char>('a' + col) };
139 }
140
141 bool Board::is_pickable(const Position &pos) const
142 {
143     auto p_chessman = get_chessman_at(pos);
144     bool pickable = false;
145     if (p_chessman != nullptr) {
146         // verifies that the chessman at _pos_ is owned by the current player
147         bool correct_color = p_chessman->color == _parent_game.
            get_current_player();
148         // and that the chessman can move at all.
149         // if the color is not correct in the first place, there is no need
150         // to check for possible moves. Hence the initialization with !correct_color.
151         bool one_move_possible = !correct_color;
152         int i = 0;
153         unsigned tiles = get_side_length() * get_side_length();
154         while (i < tiles && !one_move_possible) {
155             Position tmp = ints_to_pos(i/get_side_length(), i%get_side_length
            ());
156             if (pos != tmp) {
157                 one_move_possible = is_move_valid(pos, tmp);
158             }
159             ++i;
160         }
161         pickable = correct_color && one_move_possible;
162     }
163     return pickable;
164 }
165
166 bool Board::is_current_chessman_droppable_at(const Position &pos) const
167 {
168     return is_move_valid(_current_selection, pos);
169 }
170
171 void Board::pick(const Position &pos)
172 {
173     // insane code, do not attempt
174     // to comprehend its complexity
175     // for the sake of your mental
176     // health.
177     if (is_pickable(pos)) {
178         _current_selection = pos;
179     }
180 }
181
182 void Board::drop(const Position &pos)
183 {
184     if (get_chessman_at(_current_selection) != nullptr &&
185         is_move_valid(_current_selection, pos)) {
186
187         auto p_target = get_chessman_at(pos);
188         if (p_target != nullptr && p_target->is_essential()) {
189             _essential_lost = true;

```

```

190     }
191     // drop chessman (this also kicks previous chessman at _pos_ from the board)
192     // also note that p_target is a smart pointer, kicked chessmen are
deallocated autom.
193     _matrix[pos.first][char_to_col(pos.second)] = get_chessman_at(
        _current_selection);
194     _matrix[_current_selection.first][char_to_col(_current_selection.
        second)] = nullptr;
195     _current_selection = {};
196
197     _first_move_map[_parent_game.get_current_player()] = false;
198 }
199 }
200
201 void Board::render(std::ostream &os) const
202 {
203     static const char black = '=';
204     static const char white = '+';
205     char tile = black;
206
207     _render_col_chromium(os);
208     _render_hor_border(os);
209     for (unsigned row = 0; row < _matrix.size(); ++row) {
210         os << row + 1 << " |"; // left border
211         for (unsigned col = 0; col < _matrix[row].size(); ++col) {
212             const auto &p_chessman = _matrix[row][col];
213             _render_highlighted(os, tile, p_chessman, _get_highlight(
                ints_to_pos(row, col)));
214             tile = tile == black ? white : black; // checkerboard pattern!!!! :D
215         }
216         // switching tile sequence for each outer loop iteration
217         // to achieve the alternation effect!
218         tile = tile == black ? white : black;
219         os << " | " << row + 1 << std::endl; // right border
220     }
221     _render_hor_border(os);
222     _render_col_chromium(os);
223 }
224
225 void Board::_render_highlighted(std::ostream &os, char tile, const
    ChessmanPtr &p_chessman,
226                                const Highlight &highlight) const
227 {
228     os << highlight.first;
229     if (p_chessman == nullptr) {
230         os << tile;
231     } else {
232         p_chessman->render(os);
233     }
234     os << highlight.second;
235 }
236
237 Highlight Board::_get_highlight(Position pos) const
238 {
239     Highlight highlight;
240     auto p_chessman = get_chessman_at(pos);
241     if (is_move_valid(get_chessman_at(_current_selection), pos)) {

```



```

242     highlight = {'[', ']'};
243 } else if (p_chessman != nullptr && p_chessman == get_chessman_at(
    _current_selection)) {
244     highlight = {'(', ')'};
245 } else {
246     highlight = {' ', ' '};
247 }
248 return highlight;
249 }
250
251 bool Board::essential_lost() const
252 {
253     return _essential_lost;
254 }
255
256 void Board::_render_col_chromium(std::ostream &os) const
257 {
258     os << " |";
259     for (int i = 0; i < get_side_length(); ++i) {
260         os << ' ' << static_cast<char>('a' + i) << ' ';
261     }
262     os << "|" << std::endl;
263 }
264
265 void Board::_render_hor_border(std::ostream &os) const
266 {
267     os << "--+";
268     for (int i = 0; i < get_side_length(); ++i) {
269         os << "---";
270     }
271     os << "+--" << std::endl;
272 }
273
274 bool Board::is_first_move() const
275 {
276     return _first_move_map.at(_parent_game.get_current_player());
277 }
278
279 }

```

Listing 6: Chessman.hpp

```

1  #pragma once
2
3  #include <map>
4  #include <memory>
5  #include "Types.hpp"
6  #include "Game.hpp"
7
8  namespace Chess {
9
10     class Chessman
11     {
12     public: // methods
13
14         /**
15          * A figure can only be instantiated with a color
16          * @param color_ The color of the chessman.

```

```

17  */
18  explicit Chessman(Color color_)
19      : color{color_}
20  {}
21
22  /**
23   * Writes a chessmans representation to the
24   * supplied output stream.
25   */
26  void render(std::ostream &os) const;
27
28  /**
29   * @return True if the removal of a concrete chessman ends a game of chess.
30   */
31  virtual bool is_essential() const;
32
33  /**
34   * @param move The move to verify.
35   * @return True if the supplied move is valid vor a specific chessman.
36   */
37  virtual bool can_move(const Chess::Board &board, const Move &move) const
38      = 0;
39  public: // members
40
41  /**
42   * Describes the circumference of the chessmans body
43   * multiplied by the maximum number of tiles the chessman
44   * can move within one turn.
45   */
46  const Color color;
47
48  protected: // methods
49  /**
50   * @return The "visual" representation of the chessman.
51   */
52  virtual char get_representation() const = 0;
53  };
54
55  class King : public Chessman
56  {
57  public: // methods
58      explicit King(Color color_)
59          : Chessman{color_}
60      {}
61
62      bool can_move(const Chess::Board &board, const Move &move) const override
63          ;
64
65      bool is_essential() const override;
66  protected: // methods
67      char get_representation() const override;
68  };
69
70  class Queen : public Chessman
71  {

```

```
72 public: // methods
73     explicit Queen(Color color_)
74         : Chessman{color_}
75     {}
76
77     bool can_move(const Chess::Board &board, const Move &move) const override
78     ;
79 protected: // methods
80     char get_representation() const override;
81 };
82
83 class Bishop : public Chessman
84 {
85 public: // methods
86     explicit Bishop(Color color_)
87         : Chessman{color_}
88     {}
89
90     bool can_move(const Chess::Board &board, const Move &move) const override
91     ;
92 protected: // methods
93     char get_representation() const override;
94 };
95
96 class Rook : public Chessman
97 {
98 public: // methods
99     explicit Rook(Color color_)
100         : Chessman{color_}
101     {}
102
103     bool can_move(const Chess::Board &board, const Move &move) const override
104     ;
105 protected: // methods
106     char get_representation() const override;
107 };
108
109 class Knight : public Chessman
110 {
111 public: // methods
112     explicit Knight(Color color_)
113         : Chessman{color_}
114     {}
115
116     bool can_move(const Chess::Board &board, const Move &move) const override
117     ;
118 protected: // methods
119     char get_representation() const override;
120 };
121
122 class Pawn : public Chessman
123 {
124 public: // methods
```

```

125     explicit Pawn(Color color_)
126         : Chessman{color_}
127     {}
128
129     bool can_move(const Chess::Board &board, const Move &move) const override
130         ;
131 protected: // methods
132     char get_representation() const override;
133 };
134
135 } // namespace Chess

```

Listing 7: Chessman.cpp

```

1  #include "Chessman.hpp"
2  #include <functional>
3
4  namespace Chess {
5
6  /**
7   * Helper class
8   */
9  class Distance {
10 public: // methods
11     explicit Distance(const Move &move)
12     {
13         const auto &from = move.first;
14         const auto &to = move.second;
15
16         x = std::tolower(to.second) - std::tolower(from.second);
17         y = to.first - from.first;
18     }
19 public: // members
20     int x;
21     int y;
22 };
23
24 using TileInterpolator = std::function<Position (int)>;
25
26 /**
27  * This is absolute magic.
28  * I tried explaining this to myself as if I had not written
29  * this piece of code and I failed flawlessly. Think of it as
30  * "an iterable interpolation for a given difference between
31  * to chessmen (positions)".
32  * @param pos The position from which to start.
33  * @param delta The difference to interpolate.
34  * @return "An iterable interpolation ".
35  */
36 static TileInterpolator interpolate(Position pos, const Distance &delta)
37 {
38     // .
39     auto map = [](Position start, Distance delta, int i) -> Position {
40         // the factors providing a "mapping direction"
41         int hor_fact = delta.x == 0 ? 0 : delta.x / std::abs(delta.x);
42         int vert_fact = delta.y == 0 ? 0 : delta.y / std::abs(delta.y);
43

```

```

44     int row = start.first + (delta.y == 0 ? 0 : i) * vert_fact;
45     int col = start.second + (delta.x == 0 ? 0 : i) * hor_fact;
46     return { static_cast<unsigned>(row), static_cast<char>(col) };
47 };
48     return std::bind(map, pos, delta, std::placeholders::_1);
49 }
50
51 /**
52  * @return The value of the parameter that is not 0.
53  */
54 static int not0(int i1, int i2) {
55     if (i1 != 0 && i2 != 0) {
56         throw std::invalid_argument("Both arguments where != 0");
57     }
58     return i1 == 0 ? i2 : i1;
59 }
60
61 // ----- ~chessman~
62 // -----
63 bool Chessman::is_essential() const
64 {
65     return false;
66 }
67
68 void Chessman::render(std::ostream &os) const
69 {
70     auto transform = (color == Color::BLACK) ? toupper : tolower;
71     os << static_cast<char>(transform(get_representation()));
72 }
73
74 // ----- king
75 // -----
76 bool King::is_essential() const
77 {
78     return true;
79 }
80
81 char King::get_representation() const
82 {
83     return 'k';
84 }
85
86 bool King::can_move(const Board &board, const Move &move) const
87 {
88     Distance d(move);
89     // the king must not move more than 1 tile at a time!
90     return std::abs(d.x) <= 1 && std::abs(d.y) <= 1;
91 }
92
93 // ----- queen
94 // -----
95 char Queen::get_representation() const
96 {
97     return 'q';

```

```

98 }
99
100 bool Queen::can_move(const Board &board, const Move &move) const
101 {
102     Distance d(move);
103     // the queen must only move horizontally, vertically or diagonally
104     bool pattern_correct = std::abs(d.x) == std::abs(d.y) || d.y == 0 || d.x
        == 0;
105
106     // initialized with pattern_correct to prevent useless
107     // board traversals because the move will not become
108     // valid if the pattern is incorrect in the first place.
109     bool no_cm_inbetween = pattern_correct;
110
111     // for all steps in the move vector,
112     // verify that there is no chessman sitting
113     // at the currently checked tile
114     TileInterpolator tile_at = interpolate(move.first, d);
115     int i = std::abs(d.x) - 1;
116     while (i > 0 && no_cm_inbetween) {
117         Position potential_boi = tile_at(i);
118         no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr;
119         --i;
120     }
121     return pattern_correct && no_cm_inbetween;
122 }
123
124 // ----- bishop
125 ----- //
126 char Bishop::get_representation() const
127 {
128     return 'b';
129 }
130
131 bool Bishop::can_move(const Chess::Board &board, const Move &move) const
132 {
133     Distance d(move);
134     // a bishop may only move diagonally!
135     bool pattern_correct = std::abs(d.x) == std::abs(d.y);
136
137     bool no_cm_inbetween = pattern_correct;
138     TileInterpolator tile_at = interpolate(move.first, d);
139     int i = std::abs(d.x) - 1;
140     while (i > 0 && no_cm_inbetween) {
141         Position potential_boi = tile_at(i);
142         no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr;
143         --i;
144     }
145
146     return pattern_correct && no_cm_inbetween;
147 }
148
149 // ----- rook
150 ----- //
151 char Rook::get_representation() const

```

```

152 {
153     return 'r';
154 }
155
156 bool Rook::can_move(const Chess::Board &board, const Move &move) const
157 {
158     Distance d(move);
159     // a rook can only move along the axes!
160     bool pattern_correct = d.x == 0 || d.y == 0;
161     bool no_cm_inbetween = pattern_correct;
162     if (pattern_correct) {
163         TileInterpolator map = interpolate(move.first, d);
164         int i = std::abs(not0(d.x, d.y)) - 1;
165         while (i != 0 && no_cm_inbetween) {
166             Position potential_boi = map(i);
167             no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr
168             ;
169             --i;
170         }
171     }
172     return pattern_correct && no_cm_inbetween;
173 }
174
175 // ----- knight
176 // -----
177 char Knight::get_representation() const
178 {
179     return 'n';
180 }
181
182 bool Knight::can_move(const Chess::Board &board, const Move &move) const
183 {
184     Distance d(move);
185     // A knight can only move 2 tiles in one and 1 tile in the
186     // other dimension.
187     return (std::abs(d.x) == 2 && std::abs(d.y) == 1) ||
188            (std::abs(d.x) == 1 && std::abs(d.y) == 2);
189 }
190
191 // ----- pawn
192 // -----
193 char Pawn::get_representation() const
194 {
195     return 'p';
196 }
197
198 bool Pawn::can_move(const Chess::Board &board, const Move &move) const
199 {
200     Distance d(move);
201
202     int steps = board.is_first_move() ? 2 : 1;
203
204     // black may only move down
205     bool black_down = color == Color::BLACK && (0 < d.y && d.y <= steps);

```

```

206
207     // white may only move up
208     bool white_up = color == Color::WHITE && (-steps <= d.y && d.y < 0);
209
210     //
211     bool straight = d.x == 0;
212
213     // may only move diagonally if beating another boi
214     bool diagonal_beat = std::abs(d.x) == 1 && std::abs(d.y) == 1 &&
215         board.get_chessman_at(move.second) != nullptr;
216
217     return (black_down || white_up) && (straight || diagonal_beat);
218 }
219
220 }

```

Listing 8: main.cpp

```

1  #include <iostream>
2  #include <fstream>
3  #include "Types.hpp"
4  #include "Game.hpp"
5  #include "Chessman.hpp"
6
7  using namespace Chess;
8  using std::cout;
9  using std::endl;
10
11 static void run_multiplayer()
12 {
13     Game game;
14     while (!game.is_over()) {
15         game.next_move(std::cin, std::cout);
16     }
17 }
18
19 static Position generate_pos (std::ostream &os) {
20     time_t t;
21     // srand(time(&t)); // uncomment for madness
22     Position pos = {
23         static_cast<unsigned>(std::rand() % 8),
24         static_cast<char>('a' + (std::rand() % 8))
25     };
26     // write new position to the supplied output
27     // for visualization. (Otherwise you would not
28     // be able to see directly what the generator
29     // produces!)
30     os << pos.first << ' ' << pos.second << std::endl;
31     return pos;
32 };
33
34 static void run_generated()
35 {
36     Game game;
37     std::ofstream out("./output.txt");
38     auto generate_pos_bound = [&out]() {
39         return generate_pos(out);
40     };

```



```
41     while (!game.is_over()) {
42         game.next_move(generate_pos_bound, out);
43     }
44 }
45
46 int main(int argc, char **)
47 {
48     // lazy switch: if any program arguments were supplied
49     // use chess game generator, otherwise start M U L T I P
50     // L A Y E R
51     if (argc > 1) {
52         run_generated();
53     } else {
54         run_multiplayer();
55     }
56
57     return 0;
58 }
```

1.3 Tests

Die Tests fielen leider (ebenfalls aus Zeitgründen) sehr mager aus. Ich habe frech das per Zufall generierte Spiel als Test herangezogen und stichprobenartig überprüft, ob z.B. potentielle Züge richtig erkannt und keine falschen Züge durchgeführt werden. Da die Ausgabe des generierten Spiels ziemlich lang ist, bitte ich für Testfälle einfach in die beiliegende Datei "output.txt" zu schauen.