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 "Bibliotheksbenutzer" 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 beschrenkt.

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) lest 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, welchefür 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 ”verfollständigt” 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

```cpp
#pragma once

#include <functional>
#include <map>
#include <memory>

/*
 * This file contains forward declarations for types that are important for all classes.
 * That way cyclic dependencies can be avoided.
 */
```

```cpp

namespace Chess {

class Game;

class Board;

class Chessman;

using ChessmanPtr = std::shared_ptr<Chessman>;

/**
 * Represents a tile on the chess board.
 */
using Position = std::pair<unsigned int, char>;

std::istream &operator>>(std::istream &is, Position &pos);

using Move = std::pair<Position, Position>;

/**
 * The characters surrounding the representation
 * of a chess tile / chessman.
 */
using Highlight = std::pair<char, char>;

/**
 * Because booleans felt too hacky.
 */
enum class Color
{
    BLACK, WHITE
};

std::ostream &operator<<(std::ostream &os, Color color);

} // namespace Chess
```

Listing 2: Game.hpp

```cpp
#pragma once

#include <iostream>
#include "Types.hpp"
#include "Board.hpp"

namespace Chess {

class Game final
{
public: // methods

    /**
     * Initializes a game by setting up a chessboard with chessmen.
     */
    explicit Game(unsigned side_length = 8)
        : _board{*this, side_length}
    {}

    /**
     * Reads a move from the supplied input stream. if the move is valid,
     * the game is advanced, otherwise the stream is read until it yields
     * a valid move.
     * @param is The input stream to read the new move from.
     * @param os The output stream to write the new state of the game to.
     */
    void next_move(std::istream &is, std::ostream &os);

    /**
     * Calls the supplied function to retrieve a move. if the move is valid,
     * the game is advanced, otherwise the function is recalled until it yields
     * a valid move.
     * @param get_pos A function which returns a move.
     * @param os The output stream to write the new state of the game to.
     */
    void next_move(std::function<Position()> get_pos, std::ostream &os);

    /**
     * @return True if one player has lost an essential chessman, False otherwise.
     */
    bool is_over() const;

    /**
     * @return The player who may make the next move in the game.
     */
    Color get_current_player() const;

private: // methods

    /**
     * Writes a prompt to the supplied output stream.
     * @param prompt The message to use for the prompt.
     * @param os The output stream to write the prompt to.
     */
    void _prompt(const std::string &prompt, std::ostream &os);

private: // members

    /**
     * The player "with the action".
     * (https://www.linguee.de/deutsch-englisch/uebersetzung/spieler+der+am+zug+ist.html)
     */
    Color _current_player{Color::BLACK};

    /**
     * The game board associated with this game.
     */
    Board _board;
};

}
```

Listing 3: Game.cpp

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

```cpp

#include <functional>

namespace Chess {

void Game::next_move(std::istream &is, std::ostream &os)
{
    auto read_from_is = [&is]() {
        Position x;
        is >> x;
        return x;
    };
    next_move(read_from_is, os);
}

void Game::next_move(std::function<Position()> get_pos, std::ostream &os)
{
    _board.render(os);

    // ========= picking ========= //
    _prompt("Chose a chessman to move (row col)", os);

    Position from = get_pos();

    while (!_board.is_pickable(from)) {
        os << "Invalid chessman, try another one!" << std::endl;
        _prompt("Chose a chessman to move (row col)", os);
        from = get_pos();
    }
    _board.pick(from);

    _board.render(os);

    // ========= dropping ========= //
    _prompt("Move selected chessman to", os);

    Position to = get_pos();

    while (!_board.is_current_chessman_droppable_at(to)) {
        os << "Can not move selected chessman to specified position!" << std::endl;
        _prompt("Move selected chessman to", os);
        to = get_pos();
    }
    _board.drop(to);

    if (!is_over()) { // dont flip after game is over so the winner can be queried!
        _current_player = (get_current_player() == Color::WHITE ? Color::BLACK : Color::WHITE);
    } else {
        os << "Player " << get_current_player() << " won!" << std::endl;
    }
}

bool Game::is_over() const
{
    return _board.essential_lost();
}

void Game::_prompt(const std::string &prompt, std::ostream &os)
{
    os << get_current_player() << ' ' << prompt << "> " << std::flush;
}

Color Game::get_current_player() const
{
    return _current_player;
}

}
```

Listing 4: Board.hpp

```
#pragma once

#include <vector>
#include "Types.hpp"

namespace Chess {

class Board final
{
public: // methods

    /**
     * Creates a new chess board for a specific game.
     * @param game The game with which this board is associated.
     * @param side_length The desired side length of this board.
     */
    Board(const Game &game, unsigned side_length);

    /**
     * @param pos The indices of the desired tile.
     * @return The chessman at the specified position. If
     *         that tile is empty, the result is equal
     *         to nullptr.
     */
    ChessmanPtr get_chessman_at(const Position &pos) const;

    /**
     * @return The side length of the chessboard.
     */
    unsigned get_side_length() const;

    /**
     * @param pos The position of the potentially beatable chessman.
     * @return True if the Chessman at the specified tile is owned
     *         by ”the opposing player” and the currently selected
     *         chessman could move to _pos_.
     */
    bool is_beatable(const Position &pos) const;

    /**
     * @param p_chessman The chessman to move.
     * @param pos The move destination.
     * @return True if the chessman (who is assumed to be the currently
     *         selected one) can move to the specified position, beating
     *         the chessman at the destination if necessary.
     * @throws std::invalid_argument If the supplied chessman is NOT the
     *                              currently selected one.
     */
    bool is_move_valid(const ChessmanPtr &p_chessman, const Position &pos) const;

    /**
     * @param pos The source tile from which to move.
     * @param pos The move destination.
     * @return True if the chessman (who is assumed to be the currently
     *         selected one) can move to the specified position, beating
     *         the chessman at the destination if necessary.
     */
    bool is_move_valid(const Position &from, const Position &to) const;

    /**
     * @param pos The position of the chessman to pick up (/ ”select”).
     * @return True if there is a (friendly) chessman at the specified
     *        position and there is at least one possible move for
     *        this chessman.
     */
    bool is_pickable(const Position &pos) const;

    /**
     * @param pos The tile to check for availability.
     * @return Returns the most recently picked up chessman can
     *         be dropped at the specified position. (i. e. No
     *         other chessman obstruct the path and the specified
     *         tile is not occupied by a friendly chessman.)
     */
    bool is_current_chessman_droppable_at(const Position &pos) const;

    /**
     * Selects the chessman at the specified position.
     * If that chessman is not selectable, no action is taken.
     * @param pos The position of the chessman to select.
     */
    void pick(const Position &pos);

    /**
     * Attempts to drop the currently selected chessman at the
     * specified position. If that is not possible, no action is taken.
     * @param pos The tile where the currently selected chessman
     *            should be dropped.
     */
    void drop(const Position &pos);

    /**
     * Writes the chess board to the supplied output stream.
     * @param os The output stream to write to.
     */
    void render(std::ostream &os) const;

    /**
```

```
     * @return True if the king has been beaten.
     */
    bool essential_lost() const;

    /**
     * @return True if the player currently at action has not
     *         moved before.
     */
    bool is_first_move() const;

private: // methods

    /**
     * Writes the column decoration (letters) to the supplied
     * output stream.
     * @param os The output stream to write to.
     */
    void _render_col_chromium(std::ostream &os) const;

    /**
     * Writes a horizontal chess border to the supplied
     * output stream.
     * @param os The output stream to write to.
     */
    void _render_hor_border(std::ostream &os) const;

    /**
     * @param row The row of the tile.
     * @param pos The column of the tile.
     * @return The highlighting for the specified tile.
     */
    Highlight _get_highlight(Position pos) const;

    /**
     * Writes a part of the chessboard to the supplied output.
     * @param os The output stream to write to.
     * @param tile The character representation of the tile.
     * @param p_chessman The chessman to render. If p_chessman is a
     *                   nullptr, the character representation is rendered.
     * @param highlight The highlight for the given tile.
     */
    void _render_highlighted(std::ostream &os,
                             char tile,
                             const ChessmanPtr &p_chessman,
                             const Highlight &highlight) const;

private : // members

    /**
     * If this boi is set to true, the game can not be advanced any further.
     */
    bool _essential_lost {false};

    /**
     * The side length of the board.
     */
    unsigned _side_length;
```

```cpp

    /**
     * The game this chessboard is associated with.
     */
    const Game &_parent_game;

    /**
     * The position of the currently selected chessman.
     * @Note If this is {0,'\0'}, no chessman is selected.
     */
    Position _current_selection {};

    /**
     * The matrix representing the chessboard.
     */
    std::vector<std::vector<ChessmanPtr>> _matrix{};

    /**
     * A map for bookkeeping which player has already moved a chessman.
     * This is only used for the pawns first move.
     */
    std::map<Color, bool> _first_move_map {
        {Color::WHITE, true},
        {Color::BLACK, true}
    };
};

} // namespace Chess
```

Listing 5: Board.cpp

```cpp
#include "Board.hpp"
#include "Chessman.hpp"

namespace Chess {

std::istream &operator>>(std::istream &is, Position &pos)
{
    is >> pos.first >> pos.second;
    pos.first--; // chess starts with row 1, program logic with row 0
    return is;
}

std::ostream &operator<<(std::ostream &os, Color color)
{
    return os << '[' << (color == Color::WHITE ? "White" : "Black") << ']';
}

/**
 * @param c The character to "convert".
 * @return The 0-indexed column represented by the supplied character.
 *         This is mainly used for transformations between indices and
 *         chess board columns.
 */
static int char_to_col(char c)
{
    return std::tolower(c) - 'a';
}
```

```cpp

Board::Board(const Game &game, unsigned side_length)
        : _side_length{side_length}, _parent_game{game}
{
    if (side_length < 8) {
        throw std::invalid_argument("Chess board side length must be greater than 8!");
    }

    // Initialize the chess board matrix to the correct dimensions
    _matrix.resize(get_side_length(), std::vector<ChessmanPtr>());
    for (auto &row : _matrix) {
        row.resize(get_side_length(), ChessmanPtr());
    }

    // write special black figures to first row
    _matrix[0][0] = std::make_shared<Rook>(Color::BLACK);
    _matrix[0][1] = std::make_shared<Knight>(Color::BLACK);
    _matrix[0][2] = std::make_shared<Bishop>(Color::BLACK);
    _matrix[0][3] = std::make_shared<King>(Color::BLACK);
    _matrix[0][4] = std::make_shared<Queen>(Color::BLACK);
    _matrix[0][5] = std::make_shared<Bishop>(Color::BLACK);
    _matrix[0][6] = std::make_shared<Knight>(Color::BLACK);
    _matrix[0][7] = std::make_shared<Rook>(Color::BLACK);

    // write black pawns to 2nd row
    for (auto &tile : _matrix[1]) {
        tile = std::make_shared<Pawn>(Color::BLACK);
    }

    // some complex mathematical computation involving calculus,
    // polar representation of the cartesian coordinate system
    // and one whole lot of cookies.
    const std::size_t last = get_side_length() - 1;
    const std::size_t snd_to_last = last - 1;

    // write white pawns to 2nd to last row
    for (auto &tile : _matrix[snd_to_last]) {
        tile = std::make_shared<Pawn>(Color::WHITE);
    }

    // write special white figures to last row
    _matrix[last][0] = std::make_shared<Rook>(Color::WHITE);
    _matrix[last][1] = std::make_shared<Knight>(Color::WHITE);
    _matrix[last][2] = std::make_shared<Bishop>(Color::WHITE);
    _matrix[last][3] = std::make_shared<King>(Color::WHITE);
    _matrix[last][4] = std::make_shared<Queen>(Color::WHITE);
    _matrix[last][5] = std::make_shared<Bishop>(Color::WHITE);
    _matrix[last][6] = std::make_shared<Knight>(Color::WHITE);
    _matrix[last][7] = std::make_shared<Rook>(Color::WHITE);
}

ChessmanPtr Board::get_chessman_at(const Position &pos) const
{
    auto row = pos.first;
    auto col = char_to_col(pos.second);
    ChessmanPtr p_chessman = nullptr;

    if (row < get_side_length() && col < get_side_length()) {
        p_chessman = _matrix[row][col];
    }
    return p_chessman;
}

unsigned Board::get_side_length() const
{
    return _side_length;
}

bool Board::is_beatable(const Position &pos) const
{
    bool beatable = false;
    auto p_chessman = get_chessman_at(pos);
    if (p_chessman != nullptr) {
        // the chessman is owned by the opponent
        beatable = p_chessman->color != _parent_game.get_current_player();
        // the chessman can be reached from _current_selection
        beatable = beatable && is_move_valid(_current_selection, pos);
    }
    return beatable;
}

bool Board::is_move_valid(const ChessmanPtr &p_chessman, const Position &pos) const
{
    if (get_chessman_at(_current_selection) != p_chessman) {
        // design flaw, but meh :/
        throw std::invalid_argument("The supplied chessman is not selected.");
    }
    // verifies that there is no friendly chessman @ pos
    auto target_cm = get_chessman_at(pos);
    bool noFriendly = target_cm == nullptr ||
                      (target_cm != nullptr && target_cm->color != _parent_game.get_current_player());

    // and the chessman supports the move
    return noFriendly && p_chessman != nullptr && p_chessman->can_move(*this, {_current_selection, pos});
}

bool Board::is_move_valid(const Position &from, const Position &to) const
{
    // this is the second MAJOR design flaw. IMO you should never need a const cast
    // except when using delegation for const overloads. The delegation here was necessary
    // due to improper planning.
    const auto &buff = _current_selection;
    const_cast<Board *>(this)->_current_selection = from;
    bool v = is_move_valid(get_chessman_at(from), to);
    const_cast<Board *>(this)->_current_selection = buff;
    return v;
}
```

```cpp

static Position ints_to_pos(unsigned row, unsigned col)
{
    return { row, static_cast<char>('a' + col) };
}

bool Board::is_pickable(const Position &pos) const
{
    auto p_chessman = get_chessman_at(pos);
    bool pickable = false;
    if (p_chessman != nullptr) {
        // verifies that the chessman at _pos_ is owned by the current player
        bool correct_color = p_chessman->color == _parent_game.get_current_player();
        // and that the chessman can move at all.
        // if the color is not correct in the first place, there is no need
        // to check for possible moves. Hence the initialization with !correct_color.
        bool one_move_possible = !correct_color;
        int i = 0;
        unsigned tiles = get_side_length() * get_side_length();
        while (i < tiles && !one_move_possible) {
            Position tmp = ints_to_pos(i/get_side_length(), i%get_side_length());
            if (pos != tmp) {
                one_move_possible = is_move_valid(pos, tmp);
            }
            ++i;
        }
        pickable = correct_color && one_move_possible;
    }
    return pickable;
}

bool Board::is_current_chessman_droppable_at(const Position &pos) const
{
    return is_move_valid(_current_selection, pos);
}

void Board::pick(const Position &pos)
{
    // insane code, do not attempt
    // to comprehend its complexity
    // for the sake of your mental
    // health.
    if (is_pickable(pos)) {
        _current_selection = pos;
    }
}

void Board::drop(const Position &pos)
{
    if (get_chessman_at(_current_selection) != nullptr &&
        is_move_valid(_current_selection, pos)) {

        auto p_target = get_chessman_at(pos);
        if (p_target != nullptr && p_target->is_essential()) {
                _essential_lost = true;
        }
        // drop chessman (this also kicks previous chessman at _pos_ from the board)
        // also note that p_target is a smart pointer, kicked chessmen are deallocated autom.
        _matrix[pos.first][char_to_col(pos.second)] = get_chessman_at(_current_selection);
        _matrix[_current_selection.first][char_to_col(_current_selection.second)] = nullptr;
        _current_selection = {};

        _first_move_map[_parent_game.get_current_player()] = false;
    }
}

void Board::render(std::ostream &os) const
{
    static const char black = '=';
    static const char white = '+';
    char tile = black;

    _render_col_chromium(os);
    _render_hor_border(os);
    for (unsigned row = 0; row < _matrix.size(); ++row) {
        os << row + 1 << " |"; // left border
        for (unsigned col = 0; col < _matrix[row].size(); ++col) {
            const auto &p_chessman = _matrix[row][col];
            _render_highlighted(os, tile, p_chessman, _get_highlight(ints_to_pos(row, col)));
            tile = tile == black ? white : black; // checkerboard pattern!!!! :D
        }
        // switching tile sequence for each outer loop iteration
        // to achieve the alternation effect !
        tile = tile == black ? white : black;
        os << "| " << row + 1 << std::endl; // right border
    }
    _render_hor_border(os);
    _render_col_chromium(os);
}

void Board::_render_highlighted(std::ostream &os, char tile, const ChessmanPtr &p_chessman,
                                const Highlight &highlight) const
{
    os << highlight.first;
    if (p_chessman == nullptr) {
        os << tile;
    } else {
        p_chessman->render(os);
    }
    os << highlight.second;
}

Highlight Board::_get_highlight(Position pos) const
{
    Highlight highlight;
    auto p_chessman = get_chessman_at(pos);
    if (is_move_valid(get_chessman_at(_current_selection), pos)) {
```

```cpp
        highlight = {'[', ']'};
    } else if (p_chessman != nullptr && p_chessman == get_chessman_at(_current_selection)) {
        highlight = {'(', ')'};
    } else {
        highlight = {' ', ' '};
    }
    return highlight;
}

bool Board::essential_lost() const
{
    return _essential_lost;
}

void Board::_render_col_chromium(std::ostream &os) const
{
    os << "  |";
    for (int i = 0; i < get_side_length(); ++i) {
        os << ' ' << static_cast<char>('a' + i) << ' ';
    }
    os << "|" << std::endl;
}

void Board::_render_hor_border(std::ostream &os) const
{
    os << "--+";
    for (int i = 0; i < get_side_length(); ++i) {
        os << "---";
    }
    os << "+--" << std::endl;
}

bool Board::is_first_move() const
{
    return _first_move_map.at(_parent_game.get_current_player());
}

}
```

Listing 6: Chessman.hpp

```cpp
#pragma once

#include <map>
#include <memory>
#include "Types.hpp"
#include "Game.hpp"

namespace Chess {

class Chessman
{
public: // methods

    /**
     * A figure can only be instanciated with a color
     * @param color_ The color of the chessman.
     */
    explicit Chessman(Color color_)
            : color{color_}
    {}

    /**
     *  Writes a chessmans representation to the
     *  supplied output stream.
     */
    void render(std::ostream &os) const;

    /**
     * @return True if the removal of a concrete chessman ends a game of chess.
     */
    virtual bool is_essential() const;

    /**
     * @param move The move to verify.
     * @return True if the supplied move is valid vor a specific chessman.
     */
    virtual bool can_move(const Chess::Board &board, const Move &move) const = 0;

public: // members

    /**
     * Describes the circumference of the chessmans body
     * multiplied by the maximum number of tiles the chessman
     * can move within one turn.
     */
    const Color color;

protected: // methods
    /**
     * @return The "visual" representation of the chessman.
     */
    virtual char get_representation() const = 0;
};

class King : public Chessman
{
public: // methods
    explicit King(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

    bool is_essential() const override;

protected: // methods
    char get_representation() const override;
};

class Queen : public Chessman
{
```

```
public: // methods
    explicit Queen(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

protected: // methods
    char get_representation() const override;
};

class Bishop : public Chessman
{
public: // methods
    explicit Bishop(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

protected: // methods
    char get_representation() const override;
};

class Rook : public Chessman
{
public: // methods
    explicit Rook(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

protected: // methods
    char get_representation() const override;
};

class Knight : public Chessman
{
public: // methods
    explicit Knight(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

protected: // methods
    char get_representation() const override;
};

class Pawn : public Chessman
{
public: // methods
```

```
    explicit Pawn(Color color_)
            : Chessman{color_}
    {}

    bool can_move(const Chess::Board &board, const Move &move) const override;

protected: // methods
    char get_representation() const override;
};

} // namespace Chess
```

Listing 7: Chessman.cpp

```
#include "Chessman.hpp"
#include <functional>

namespace Chess {

/**
 * Helper class
 */
class Distance {
public: // methods
    explicit Distance(const Move &move)
    {
        const auto &from = move.first;
        const auto &to = move.second;

        x = std::tolower(to.second) - std::tolower(from.second);
        y = to.first - from.first;
    }
public: // members
    int x;
    int y;
};

using TileInterpolator = std::function<Position (int)>;

/**
 * This is absolute magic.
 * I tried explaining this to myself as if I had not written
 * this piece of code and I failed flawlessly. Think of it as
 * ”an iterable interpolation for a given difference between
 * to chessmen (positions)”.
 * @param pos The position from which to start.
 * @param delta The difference to interpolate.
 * @return ”An iterable interpolation”.
 */
static TileInterpolator interpolate(Position pos, const Distance &delta)
{
    // .
    auto map = [](Position start, Distance delta, int i) -> Position {
        // the factors providing a ”mapping direction”
        int hor_fact = delta.x == 0 ? 0 : delta.x / std::abs(delta.x);
        int vert_fact = delta.y == 0 ? 0 :  delta.y / std::abs(delta.y);

        int row = start.first + (delta.y == 0 ? 0 : i) * vert_fact;
        int col = start.second + (delta.x == 0 ? 0 : i) * hor_fact;
        return { static_cast<unsigned>(row), static_cast<char>(col) };
    };
    return std::bind(map, pos, delta, std::placeholders::_1);
}

/**
 * @return The value of the parameter that is not 0.
 */
static int not0(int i1, int i2) {
    if (i1 != 0 && i2 != 0) {
        throw std::invalid_argument("Both arguments where != 0");
    }
    return i1 == 0 ? i2 : i1;
}

// ------------------ ~chessman~ ------------------ //

bool Chessman::is_essential() const
{
    return false;
}

void Chessman::render(std::ostream &os) const
{
    auto transform = (color == Color::BLACK) ? toupper : tolower;
    os << static_cast<char>(transform(get_representation()));
}

// ------------------ king ------------------ //

bool King::is_essential() const
{
    return true;
}

char King::get_representation() const
{
    return 'k';
}

bool King::can_move(const Board &board, const Move &move) const
{
    Distance d(move);
    // the king must not move more than 1 tile at a time!
    return std::abs(d.x) <= 1 && std::abs(d.y) <= 1;
}

// ------------------ queen ------------------ //

char Queen::get_representation() const
{
    return 'q';
```

```cpp
}

bool Queen::can_move(const Board &board, const Move &move) const
{
    Distance d(move);
    // the queen must only move horizontally, vertically or diagonally
    bool pattern_correct =  std::abs(d.x) == std::abs(d.y) || d.y == 0 || d.x == 0;

    // initialized with pattern_correct to prevent useless
    // board traversals because the move will not become
    // valid if the pattern is incorrect in the first place.
    bool no_cm_inbetween = pattern_correct;

    // for all steps in the move vector,
    // verify that there is no chessman sitting
    // at the currently checked tile
    TileInterpolator tile_at = interpolate(move.first, d);
    int i = std::abs(d.x) - 1;
    while (i > 0 && no_cm_inbetween) {
        Position potential_boi = tile_at(i);
        no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr;
        --i;
    }
    return pattern_correct && no_cm_inbetween;
}

// ----------------------- bishop ----------------------- //

char Bishop::get_representation() const
{
    return 'b';
}

bool Bishop::can_move(const Chess::Board &board, const Move &move) const
{
    Distance d(move);
    // a bishop may only move diagonally!
    bool pattern_correct = std::abs(d.x) == std::abs(d.y);

    bool no_cm_inbetween = pattern_correct;
    TileInterpolator tile_at = interpolate(move.first, d);
    int i = std::abs(d.x) - 1;
    while (i > 0 && no_cm_inbetween) {
        Position potential_boi = tile_at(i);
        no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr;
        --i;
    }

    return pattern_correct && no_cm_inbetween;
}

// ----------------------- rook ----------------------- //

char Rook::get_representation() const
```

```
{
    return 'r';
}

bool Rook::can_move(const Chess::Board &board, const Move &move) const
{
    Distance d(move);
    // a rook can only move along the axes!
    bool pattern_correct = d.x == 0 || d.y == 0;
    bool no_cm_inbetween = pattern_correct;
    if (pattern_correct) {
        TileInterpolator map = interpolate(move.first, d);
        int i = std::abs(not0(d.x, d.y)) - 1;
        while (i != 0 && no_cm_inbetween) {
            Position potential_boi = map(i);
            no_cm_inbetween = board.get_chessman_at(potential_boi) == nullptr
     ;
            --i;
        }
    }

    return pattern_correct && no_cm_inbetween;
}

// ------------------------- knight
     ------------------------- //

char Knight::get_representation() const
{
    return 'n';
}

bool Knight::can_move(const Chess::Board &board, const Move &move) const
{
    Distance d(move);
    // A knight can only ove 2 tiles in one and 1 tile in the
    // other dimension.
    return (std::abs(d.x) == 2 && std::abs(d.y) == 1) ||
            (std::abs(d.x) == 1 && std::abs(d.y) == 2);
}

// ------------------------- pawn
     ------------------------- //

char Pawn::get_representation() const
{
    return 'p';
}

bool Pawn::can_move(const Chess::Board &board, const Move &move) const
{
    Distance d(move);

    int steps = board.is_first_move() ? 2 : 1;

    // black may only move down
    bool black_down = color == Color::BLACK && (0 < d.y && d.y <= steps);
```

```cpp

    // white may only move up
    bool white_up = color == Color::WHITE && (-steps <= d.y && d.y < 0);

    //
    bool straight = d.x == 0;

    // may only move diagonally if beating another boi
    bool diagonal_beat = std::abs(d.x) == 1 && std::abs(d.y) == 1 &&
            board.get_chessman_at(move.second) != nullptr;

    return (black_down || white_up) && (straight || diagonal_beat);
}

}
```

Listing 8: main.cpp

```cpp
#include <iostream>
#include <fstream>
#include "Types.hpp"
#include "Game.hpp"
#include "Chessman.hpp"

using namespace Chess;
using std::cout;
using std::endl;

static void run_multiplayer()
{
    Game game;
    while (!game.is_over()) {
        game.next_move(std::cin, std::cout);
    }
}

static Position generate_pos (std::ostream &os) {
    time_t t;
//    srand(time(&t)); // uncomment for madness
    Position pos = {
            static_cast<unsigned>(std::rand() % 8),
            static_cast<char>('a' + (std::rand() % 8))
    };
    // write new position to the supplied output
    // for visualization. (Otherwise you would not
    // be able to see directly what the generator
    // produces!)
    os << pos.first << ' ' << pos.second << std::endl;
    return pos;
};

static void run_generated()
{
    Game game;
    std::ofstream out("./output.txt");
    auto generate_pos_bound = [&out](){
        return generate_pos(out);
    };
    while (!game.is_over()) {
        game.next_move(generate_pos_bound, out);
    }
}

int main(int argc, char **)
{
    // lazy switch: if any program arguments were supplied
    // use chess game generator, otherwise start MULTIP
    // LAYER
    if (argc > 1) {
        run_generated();
    } else {
        run_multiplayer();
    }

    return 0;
}
```

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.