Praktikum 5 PAD1

Programmieren / Algorithmen und Datenstrukturen (Hochschule Darmstadt)

Programmieren / Algorithmen und Datenstrukturen 1 - Praktikum 5

Michael Roth

Lernziele

- Verwenden von Klassen
- Implementieren einfacher Programmlogik
- Umgang mit Kompositionen
- Umgang mit `std::vector` und `std::array`
- Absichern von Funktionen mittels Exceptions

1 Abgabe über openSubmit

Ihnen steht auf Moodle ein Vorgabeprojekt zur Verfügung, welches zu benutzen ist. Falls Sie eine andere IDE als QtCreator verwenden wollen, dann benutzen Sie dennoch die dort beigefügten `.h` und `.cpp` Dateien.

Erstellen Sie zum Abgeben Ihrer fertigen Lösung ein `.zip` File mit allen Header- und Quellcode Dateien (`.h` und `.cpp`) Ihres Projekts und laden Sie dieses über die openSubmit Website hoch.

1.1 Erweiterungen der Header

Sie dürfen die mitgelieferten Headerdateien des Vorgabeprojekts beliebig **erweitern**. Das bedeutet, Sie können zu den Klassen beliebig viele Attribute und Methoden **hinzufügen**, so lange diese **`private`** sind.

Sie dürfen aller keine der bestehenden Attribute oder Methoden **löschen** oder **verändern**, oder Methoden bzw. Attribute als `public` hinzufügen.

Kurz: Alles was `public` ist, bleibt unverändert!

2 Das Spiel „Schiffe versenken“ (engl. „Battleships“)

In diesem Praktikum implementieren Sie das (hoffentlich) bekannte Spiel „Schiffe versenken“[1] bzw. „Battleships“[2] (englisch). Die Wikipedia Seiten zu den Regeln sind in den Fußnoten verlinkt, sollten Sie damit nicht vertraut sein.

Das Spiel an sich ist für zwei Spieler gedacht, die eigentlich nicht das Spielbrett des anderen Spielers sehen sollten. In dieser Implementierung jedoch werden die Spielbretter beider Spieler auf der gleichen Konsole ausgegeben (Es geht ja nicht um die Erschaffung des perfekten Spiels, sondern um die eigentliche Programmierung dahinter!).

Grundsätzlich geht es darum, dass jeder Spieler auf einem Spielfeld von 10×10 Feldern insgesamt zehn Schiffe verschiedener Länge (zwei bis fünf Felder) verteilen. Der jeweilige Gegner „beschießt“ das Spielfeld des Spielers durch Angabe von Reihe und Spalte eines Feldes. Der Spieler sagt dann entweder „Wasser“ wenn kein Schiff getroffen wurde bzw. „Treffer“ wenn ein Schiff durch den Schuss getroffen wurde.

In der Regel besitzt jeder Spieler noch einen zusätzlichen „Schmierzettel“, auf dem dieser notiert welche Felder des Gegners bereits beschossen wurden.

[1] `https://de.wikipedia.org/wiki/Schiffe_versenken`
[2] `https://en.wikipedia.org/wiki/Battleship_(game)`

Für die Implementierung des Spiels werden insgesamt fünf Klassen entworfen:

`Player` stellt einen Spieler dar und speichert den Namen sowie jeweils die Anzahl der gewonnenen und verlorenen Spiele.

`Part` stellt ein Schiffsteil dar, welches genau ein Feld belegt. Ein Schiff besteht damit aus zwei bis fünf Objekten von `Part`. Ein Schiffsteil kann entweder unbeschädigt oder beschädigt sein (Wenn es getroffen wurde).

`Ship` stellt ein Schiff dar, welches eben aus verschieden vielen Teilen und damit aus Objekten von `Part` besteht. Das Schiff als solches ist entweder:

- unbeschädigt, wenn alle Schiffsteile unbeschädigt sind,
- beschädigt, wenn mindestens ein (aber nicht alle!) Schiffsteil beschädigt ist,
- oder aber versenkt, wenn alle Schiffsteile beschädigt sind

`GameBoard` verfügt über ein `std:array` mit den Schiffen eines Spielers sowie dem „Schmierzettel“ zur Markierung welches gegenerische Feld bereits beschossen wurde. Die Klasse ist ebenfalls zuständig für die Ausgabe des Spielfelds bzw. des „Schmierzettels“.

`Battleship` ist die „Hauptklasse“ des Spiels und verwaltet zwei `Player` Objekte sowie die jeweils dazugehörigen `GameBoard` Objekte.

Hinweis: Lesen Sie unbedingt auch die mitgelieferte Dokumentation des Quellcodes, vor allen Dingen bei Unklarheiten!

3 Klasse `Player`

Die Klasse `Player` dient einfach nur der Verwaltung eines Spielers. Dort werden für jeden Spieler folgende Informationen gespeichert:

- Name des Spielers
- Anzahl der gewonnen Spiele
- Anzahl der verlorenen Spiele

Der (einzige) Konstruktor von `Player` bekommt einen `string` übergeben, welcher als Name verwendet werden soll. Weiterhin soll der Konstruktor die Anzahl der gewonnen bzw. verlorenen Spiele mit dem Wert 0 initialisieren.

Hinweis: Hier ist **explizit** eine Initialisierungsliste zu verwenden!

4 Klasse `Part`

Die Klasse `Part` stellt ein Schiffsteil dar, welches genau ein Feld des Spielfelds belegt. Daher „weiß“ ein `Part` Objekt, an welcher Position des Spielfelds es sich selbst befindet. Die Attribute von `Part` sind:

- Reihe und
- Spalte
- Status, also ob das Teil unbeschädigt oder beschädigt ist

Für die Position existieren jeweils die beiden Getter-Methoden `getRow` bzw. `getCol`. Für den Status existiert der Getter `isDamaged`, welcher `true` liefert falls dieses Teil beschädigt ist, sowie der Setter `setDamaged`, welcher den Status auf „beschädigt“ setzt. Eine „Reparatur“, also ein Zurücksetzen des Status auf „unbeschädigt“, ist nicht vorgesehen.

Der (einzige) Konstruktor von `Part` soll ein unbeschädigtes Schiffsteil an der in den Parametern übergebenen Position erstellen.

5 Klasse `Ship`

Die Klasse `Ship` stellt ein Schiff für das Spiel dar und besteht im Wesentlichen nur aus einer Sammlung an `Part` Objekten. Ein `Ship` Objekt selbst „weiß“ nicht an welcher Position es steht, daher sind die Positionen einzeln in den `Part` Objekten gespeichert. Als Datenstruktur für die `Ship` Objekte wird ein `std::vector` verwendet da Schiffe einer unterschiedliche Anzahl an Teilen besitzen (bzw. Schiffe unterschiedlich lang sind).

Der Konstruktor `Ship(int row, int col, int lengthOfShip, int direction)` wird verwendet, um ein neues Schiff zu erstellen. Dabei gilt:

- `row` und `col` geben die Position des ersten Schiffteils an
- Insgesamt sollen `lengthOfShip` Teile erstellt werden (Nach dem ersten also noch `lengthOfShip - 1` weitere Teile)
- Das Schiff soll in die Richtung `direction` ausgerichtet werden. Alle weiteren Teile nach dem ersten werden also wie folgt erstellt:
  - Nach oben, wenn `direction = 0`
  - Nach rechts, wenn `direction = 1`
  - Nach unten, wenn `direction = 2`
  - Nach links, wenn `direction = 3`

  Wobei die Reihe 0 die oberste Reihe, und Spalte 0 die linkeste Spalte ist.
- **Wichtig:** Der Konstruktor soll eine **Exception** vom Typ `std::invalid_argument` werfen, falls:
  - Eine ungültige `direction` übergeben wurde oder
  - (Mindestens) ein Schiffsteil außerhalb des Spielfeldes platziert werden müsste

5.1 Methode `hasPartIn(int row, int col)`

Diese Methode soll `true` zurück liefern, falls ein Teil dieses Schiffes an der angegebenen Position liegt. Dazu müssen Sie den `std::vector` mit den `Part` Objekten durchsuchen, ob eben ein solches Teil existiert. Falls nicht, soll die Methode `false` zurück geben.

5.2 Methode `getPartIn(int row, int col)`

Diese Methode liefert eine **Referenz** auf das Schiffsteil an der angegebenen Position zurück. Ist kein solches Teil vorhanden, soll die Methode eine **Exception** vom Typ `std::invalid_argument` liefern.

5.3 Methoden `isSunk()`, `allShipsSunk` und `isDamaged`

Diese Methoden sollen ermitteln, ob das Schiff versenkt wurde bzw. ob es beschädigt ist. Ein Schiff gilt als versenkt, sobald **alle** seine Teile beschädigt sind. Ein Schiff ist beschädigt, wenn mindestens eines der Teile beschädigt ist. Damit ist ein versenktes Schiff natürlich auch beschädigt.

Die Methode `allShipsSunk()` liefert `true` falls **alle** Schiffe versenkt sind.

6 Klasse `GameBoard`

Die Klasse `GameBoard` stellt eine Aufzählung der Schiffe eines Spielers zur Verfügung (als `std::array<Ship>`). Weiterhin verwaltet die Klasse den „Schmierzettel“ (als `std::array<std::array<char, 10>, 10>`auf dem notiert werden soll, welche Positionen der Spieler beim Gegner bereits beschossen hat.

6.1 Kleiner Exkurs: „Mehrdimensionale“ Arrays

Für den „Schmierzettel“ wird hier ein so genanntes mehrdimensionales Array verwendet. Dabei handelt es sich einfach um ein Array, welches als einzelne Elemente weitere Arrays enthält. Sie können sich das für den Kontext des Spiels so vorstellen, das ein `array<char, 10>` eine **Reihe** des Schmierzettels darstellt. Zehn Reihen wiederum ergeben dann natürlich das 10×10 Felder große Spielfeld.

Wenn der „Datentyp“ für eine **Reihe** also `array<char, 10>` ist, dann ist ein Array mit zehn Elementen vom Typ „Reihe“ genau ein `array<array<char, 10>, 10>`.

Als kleines Beispielprogramm:

```
array<array<char, 10>, 10> fields;
array<char, 10> foo = fields.at(2); //Call at() to get a specific row (e.g. the whole row!)
cout << foo.at(7); //Call at() on a row to get a specific field (e.g. a char)

//Or, simplier:
cout << fields.at(2).at(7); //The first at() call returns an item from 'fields', which is an
  array on which we can call at()!
```

6.2 Methoden `printBoard()` und `printEnemyBoard()`

Diese Methoden sollen das Spielfeld des Spielers sowie seinen „Schmierzettel“ ausgeben.

Für die Ausgabe des Spielfelds muss ermittelt werden, an welchen Positionen sich die Schiffe bzw. deren Teile befinden. Geben Sie für jedes Feld des 10×10 Spielfeldes genau ein Zeichen aus, welches anzeigt was sich auf dem Feld befindet. Ein Feld kann dabei folgende „Zustände“ haben:

- Ein Wasserfeld, auf dem sich kein Schiff befindet. Benutzen Sie hierfür beispielsweise einen Punkt `'.'`
- Ein unbeschädigtes Schiffsteil, dargestellt durch die Nummer des zugehörigen Schiffes. Die Nummer ist identisch zum Index des Schiffes innerhalb des Arrays.
- Ein beschädigtes Schiffsteil kann durch ein `'X'` dargestellt werden
- Ein Teil eines versenkten Schiffes kann beispielsweise durch `'S'` oder auch `'#'` dargestellt werden.

Der Schmierzettel besteht aus lauter `char` Werten, wobei jedes Feld drei Zustände haben kann:

- Noch kein Schuss ausprobiert: `'.'`
- Treffer an dieser Stelle: `'X'`
- Wasser an dieser Stelle: `'O'`

6.3 Methoden `hit(int row, col)` und `mark(int row, col, bool wasHit)`

Die Methode `hit` wird vom Gegner aufgerufen und stellt den Beschuss des angegebenen Feldes dar. Falls sich dort ein Schiffsteil befindet, so wird dieses beschädigt und `true` zurück gegeben. Falls nicht, wird `false` zurück gegeben (Der gegnerische Spieler muss wissen, ob er etwas getroffen hat oder nicht).

Mit der Methode `mark` wird an der angegebenen Stelle des „Schmierzettels“ markiert, ob dort ein Schiff getroffen wurde oder nicht. Die Methode soll dann an die entsprechende Stelle ein `'X'` bzw. ein `'O'` schreiben.

6.4 Methode `randomPlaceShips()`

Diese Methode soll zufällig auf dem Spielfeld die folgenden Schiffe verteilen:

- Ein 'Schlachtschiff' bestehend aus fünf Teilen
- Zwei 'Kreuzer' bestehend aus vier Teilen
- Drei 'Zerstörer' bestehend aus drei Teilen
- Vier 'U-Boote' bestehend aus zwei Teilen

Die Schiffe sollen dabei so platziert werden, dass diese sich nicht gegenseitig überschneiden und alle Teile tatsächlich auf dem Spielfeld sind (Siehe auch den Konstruktur von `Ship` in Aufgabe 5 auf Seite 3).

Weiterhin soll jeder Aufruf tatsächlich eine neue Verteilung der Schiffe generieren. Insbesondere sollen beide Spieler verschiedene Verteilungen beim Start bekommen.[3]

6.5 Konstruktor

Der Konstruktor von `GameBoard` soll den „Schmierzettel“ mit dem Zeichen für „Noch kein Schuss ausprobiert“ befüllen und kann wahlweise ebenfalls `randomPlaceShips()` aufrufen.

7 Klasse `Battleship`

In der Klasse `Battleship` findet das eigentliche Spiel statt.

Dem Konstruktor der Klasse werden dabei zwei `string` Objekte übergeben, mit denen der Konstruktor die beiden `Player` Objekte **initialisieren** soll.

Ein Aufruf der Methode `play()` startet nun eine **Spielrunde**. Das bedeutet, dass zwei „frische“ GameBoard Objekte erzeugt werden müssen, auf denen die beiden Spieler spielen können. Die `Player` Objekte werden einmalig im Konstruktor erzeugt und initialisiert.

Anschließend wird ein Startspieler ermittelt und jeder Spieler bekommt abwechselnd sein Spielfeld sowie seinen „Schmierzettel“ ausgegeben und kann anschließend ein Feld angeben, welches er beim Gegner beschießen möchte. Der aktive Spieler wird informiert, ob der Schuss getroffen hat oder nicht. Das Ergebnis wird automatisch auf dem „Schmierzettel“ des Spielers vermerkt.

Nachdem der aktive Spieler seinen Zug gemacht hat, ist der Gegner dran.

Die beiden Spieler wechseln sich so lange ab, bis ein Spieler gewonnen hat. Dies ist der Fall, wenn alle Schiffe des Gegners versenkt wurden. Beim Ende des Spiels sollen ebenfalls die Spielstatistiken in den Spielern aktualisiert werden.

[3] Es gibt exakt 26.509.655.816.984 verschiedene Möglichkeiten die Schiffe aufzustellen. Wenn ihr Verfahren hinreichend zufällig ist, können Sie also davon ausgehen, dass beide Spieler unterschiedliche Startbedingungen haben.

Lösungsvorschlag _ Praktikum 5 _ PAD1

Hochschule Darmstadt

Battle Ships Game

battleship.cpp Class

```
#include "battleship.h"
#include <stdexcept>


Battleship::Battleship(const string &player1Name, const string &player2Name) :
m_players{Player(player1Name) , Player(player2Name)} {}


/**
 * @brief Play one game of Battleships!
 *
 * This function shall do the following:
 *
 *    - Initialize both players game boards:
 *    - The ships shall be placed at randomized locations
 *    - The 'cheat sheets' shall be empty
 *    - This can easily be achieved by creating new \ref GameBoard objects
 * and storing them in \ref m_boards, overwriting the old boards which may be
 * there already
 *
 * Then, do the following in a loop:
 *   1. Determine the active Player either by random coin toss or first player
 * always gets to start
 *   2. Print both the game board and the 'cheat sheet' for the active player
 *   3. Ask the active player a location she wants to shoot at
 *   4. Call the \ref GameBoard::hit function for that location on the
 * **inactive** Player's board
 *   5. Inform the active player if she hit something and make the correct
 * mark on the cheat sheet
 *   6. Switch the players, e.g. active player becomes inactive and inactive
 * Player becomes active
 *   7. Repeat Steps 1 to 6 until one player has lost
 *   8. Exit the function
 */



void Battleship::play(){

   m_boards.at(0).randomPlaceShips();
   m_boards.at(1).randomPlaceShips();

   bool exit = false;
   bool startFlag = true;
```

```
  int actPlayer = 0;
  int inactPlayer = 0;



  while(!exit){


    if(m_boards.at(actPlayer).allShipsSunk()){
      std::cout<<"================================="<<std::endl;
      std::cout<<m_players[actPlayer].getName()<< "'s ships are all sunk"<<std::endl;
      std::cout<<"Game Over!"<<std::endl;
      std::cout<<"Player " <<m_players[actPlayer].getName()<< " hast lost!" <<std::endl;
      std::cout<<"================================="<<std::endl;
      m_players.at(actPlayer).addGameLost();
      m_players.at(inactPlayer).addGameWon();
      exit = true;
    }


    if(exit == false){


    if (startFlag == true)
    {
      actPlayer = rand() % (2);
      if (actPlayer == 0){
        inactPlayer = 1;
      }
      startFlag = false;
    }

    std::cout<<" ====================
"<<m_players[actPlayer].getName()<<"'s turn
====================="<<std::endl;
    std::cout<<m_players[actPlayer].getName()<<"'s Map: "<<std::endl;
    m_boards.at(actPlayer).printBoard();
    std::cout<<std::endl;
    std::cout<<m_players[actPlayer].getName()<<"'s Cheat Sheet: "<<std::endl;
    m_boards.at(actPlayer).printEnemyBoard();
    std::cout<<std::endl;


    std::cout<<std::endl;
    std::cout<<"Which Location do you wanna shoot at?"<<std::endl;
    int row,col;
    std::cout<<"Row: ";
    std::cin>>row;
    std::cout<<std::endl;
    std::cout<<"Column: ";
    std::cin>>col;
    std::cout<<std::endl;
    if (row < 0 || row > 9 || col < 0 || col > 9){
```

```
        throw std::invalid_argument("Invalid Input! - Row and Column should be both between
0-9");
      }

      if(m_boards.at(inactPlayer).hit(row, col)){
         std::cout<< "Good Job! Enemy was hit!" <<std::endl;
         std::cout<<std::endl;
         m_boards.at(actPlayer).mark(row, col, true);
      } else {
         std::cout<< "Unfortunately Enemy was not hit!" <<std::endl;
         std::cout<<std::endl;
         m_boards.at(actPlayer).mark(row, col, false);
      }


      //switching players
      int temp = actPlayer;
      actPlayer = inactPlayer;
      inactPlayer = temp;

      }
   }
```

gameboard.cpp Class

```
#include "gameboard.h"


/**
 * @brief GameBoard constructor
 *
 * This shall initialize the enemy board (attribute \ref m_enemyBoard) with
 * dot characters '.'. The player's ships can either be placed here also, or
 * later by calling \ref randomPlaceShips.
 *
 *
 */


GameBoard::GameBoard(){
   for(int i{0}; i<10; i++){
      for(int j{0}; j<10; j++){
         m_enemyBoard[i][j] = '.';
      }
   }
}
```

```
/**
 * @brief Prints the player's board.
 *
 * This function prints the player's board to the console screen. You are
 * relatively free to be creative here, but make sure that all ships are
 * displayed properly.
 *
 * Some suggestions are:
 *
 *   - A location containing water is represented by printing '.'
 *   - A location containing an intact (e.g. undamaged) ship part is
 * represented by its ship number (e.g. the ships index in the array)
 *   - A location containing a damaged ship part of an **unsunken ship** is
 * represented by an 'X'
 *   - A location containing a part of a sunken ship is represented by an 'S'
 *
 * You can, as mentioned before, use other characters, but those four types of
 * locations should be distinguishable from another.
 */
void GameBoard::printBoard(){

   array<array<char, 10>, 10> myBoard;

   for(int i{0}; i<10; i++){
      for(int j{0}; j<10; j++){
         myBoard[i][j] = '.';
      }
   }

   for(int i{0}; i<m_ships.size(); i++){
      for(Part &part : m_ships[i].get_m_parts()){
         if(part.isDamaged() && m_ships[i].isSunk()){
            myBoard[part.getRow()][part.getCol()] = 's';    //ship is sunk
         }
         else if (part.isDamaged()){
            myBoard[part.getRow()][part.getCol()] = 'x';    //a part of the ship is damaged
         } else {
            myBoard[part.getRow()][part.getCol()] = static_cast<char>(i+48); //ship is safe &
sound
         }
      }
   }

   //newly Edited
   for(int i{0}; i<10; i++){
      for(int j{0}; j<10; j++){
         std::cout<<myBoard[i][j]<< " ";
      }
      std::cout<<std::endl;
   }
}
```

```
/**
 * @brief Prints the 'cheat sheet', containing markings where this player has
 * hit or missed.
 *
 * When trying to hit the other player's ships, it does not make sense to
 * shoot the same location more than once. In order to 'memorize' those
 * locations, the two-dimensional array is used. In there, the characters have
 * the following meanings:
 *
 *   - '.' represents a location which has not yet beens shot
 *   - 'X' represents a hit ship at that locations
 *   - 'O' represents a shot into open water
 *
 * As with \ref printBoard you are free to change the characters if you fancy
 * something else.
 */

void GameBoard::printEnemyBoard(){
   for(int i{0}; i<10; i++){
      for(int j{0}; j<10; j++){
         std::cout<<m_enemyBoard[i][j]<< " ";
      }
      std::cout<<std::endl;
   }
}

/**
 * @brief The enemy player's shot on our board
 * @param[in] row The row where we are being shot
 * @param[in] col The column where we are being shot
 * @return True if the shot hit any of our ships, false otherwise.
 *
 * This function is called when the enemy is trying to hit any of our ships.
 * Generally a shot is directed at a specific location denoted by `row` and
 * `col`.
 *
 * If there is a ship part at that location, that part shall be damaged by
 * that shot.
 *
 * \see Part::setDamaged
 * \see printBoard
 */
bool GameBoard::hit(int row, int col){
   for(Ship &ship : m_ships){
      if(ship.hasPartIn(row, col)){
         ship.getPartIn(row, col).setDamaged();
         return true;
      }
   }
   return false;
}

/**
 * @brief Mark locations on the enemy board where we already shot at
```

```
 * @param[in] row The row where we shot
 * @param[in] col The column where we shot
 * @param[in] wasHit True if the shot was a hit (e.g. we hit a ship)
 *
 * This function shall make a mark on the 'cheat sheet'. The mark shall be
 * different depending on if we hit something there.
 *
 * \see hit
 * \see printEnemyBoard
 */

void GameBoard::mark(int row, int col, bool wasHit){
   if(wasHit){
      m_enemyBoard[row][col] = 'X';
   } else {
      m_enemyBoard[row][col] = '0';
   }
}









template <size_t S>
bool combinationIsValid( std::array<Ship,S> arr, int row, int col, int len, int dir){

   if (dir == 0){
      int rangeEnd = row - len +1;
      if (rangeEnd < 0){
         return false;
      }

      Ship ship = Ship(row, col, len, dir);
      for (int i{row}; i>=rangeEnd; i--){
         for(Ship &ship : arr){
            if (ship.hasPartIn(i, col))
               return false;
         }
      }
   }

   else if (dir == 1){
      int rangeEnd = col + len -1;
      if (rangeEnd > 9){
         return false;
      }
      Ship ship = Ship(row, col, len, dir);
      for (int i{col}; i<=rangeEnd; i++){
         for(Ship &ship : arr){
            if (ship.hasPartIn(row, i))
               return false;
         }
```

```
        }
    }

    else if (dir == 2){
        int rangeEnd = row + len -1;
        if (rangeEnd > 9){
            return false;
        }
        Ship ship = Ship(row, col, len, dir);
        for (int i{row}; i<=rangeEnd; i++){
            for(Ship &ship : arr){
                if (ship.hasPartIn(i, col))
                    return false;
            }
        }
    }

    else if (dir == 3){
        int rangeEnd = col - len +1;
        if (rangeEnd < 0){
            return false;
        }
        Ship ship = Ship(row, col, len, dir);
        for (int i{col}; i>=rangeEnd; i--){
            for(Ship &ship : arr){
                if (ship.hasPartIn(row, i))
                    return false;
            }
        }
    }

    return true;
}




/**
 * @brief Randomly place ships.
 *
 * This function randomly places ships on the board, e.g. it populates the
 * \ref m_ships vector.
 *
 * The following ships shall be placed:
 *   - 1 'Dreadnought' with 5 parts
 *   - 2 'Cruisers' with 4 parts
 *   - 3 'Destroyers' with 3 parts
 *   - 4 'Submarines' with 2 parts
 *
 * The ships shall be placed so that:
 *   - No ships intersect each other
 *   - No ship has parts outside the playing area
```

```
*   - When this is called for both players, the resulting placements shall be
* different
*
*/



void GameBoard::randomPlaceShips(){

   int partsNum = 5;
   int m_ships_index = 0;
   int randRow = 0;
   int randCol = 0;
   int randDir = 0;

   for (int shipType{1}   ; shipType<=1  ; shipType++){    //shiptype <= 4
      for (int shipsNum {1}   ; shipsNum <=shipType   ; shipsNum++){

            //srand(time(nullptr));
            bool validCombination = false;

            while (!validCombination){

               randRow = rand() % (10);    // 0...9
               randCol = rand() % (10);    // 0...9
               randDir = rand() % (4);     // 0...3
               if( combinationIsValid(m_ships,randRow,randCol,partsNum,randDir) ){
                  validCombination = true;
               }
            }

            m_ships[m_ships_index] = Ship(randRow, randCol, partsNum, randDir);
            m_ships_index++;
         }

      partsNum--;
   }
}




/**
 * @brief Test if all ships are sunk
 * @return True if all ships on this board are sunk
 *
 * This function is used to determine if the player hast lost the game. As a
 * reminder: The player has lost the game when she has no floating ship left.
 */
bool GameBoard::allShipsSunk(){
   for(Ship &ship : m_ships){
      if(!ship.isSunk())
```

```
            return false;
        }
        return true;
}
```

Part.cpp Class

```
#include "part.h"

/**
 * @brief Part constructor
 * @param[in] row The grid row on which this part is to be placed
 * @param[in] col The grid column on which this part is to be placed
 *
 * This is supposed to be the only Part constructor. The values for \ref m_row
 * and \ref m_col shall be assigned from `row` and `col` respectively.
 * The status \ref m_status shall be initialized with 0. indicating 'no
 * damage'.
 */
Part::Part(int row, int col) : m_row(row) , m_col(col) , m_status(0) {}

/**
 * @brief Returns wether or not this part is damaged
 * @return True if damaged, False if undamaged
 *
 * Note that there is no extra status for a sunken \ref Ship.
 * By definition, a ship is sunk when all parts were hit, e.g. are damaged.
 */

bool Part::isDamaged(){
    return m_status;
}

/**
 * @brief Sets the status of this part to a valued representing 'damaged'.
 *
 * The attribute \ref m_status is supposed to be 0 if there is no damage,
 * and 1 if this part was damaged.
 * This method shall thereby set \ref m_status to 1.
 */

void Part::setDamaged(){
    m_status = 1;
}
```

```
/**
 * @brief Returns the row with which this part was constructed
 * @return The row with which this part was constructed
 */

int Part::getRow(){
   return m_row;
}

/**
 * @brief Returns the column with which this part was constructed
 * @return The column with which this part was constructed
 */

int Part::getCol(){
   return m_col;
}
```

player.cpp Class

```
#include "player.h"


/**
 * @brief Player constructor.
 * @param[in] playerName The player's name.
 *
 * The attribute \ref m_playerName shall be initialized with the
 * `playerName` parameter.
 *
 * The attributes \ref m_gamesWon and \ref m_gamesLost
 * shall be initialized with 0.
 */
Player::Player(const string &playerName) : m_playerName{playerName} , m_gamesWon{0} ,
m_gamesLost{0} {}

/**
 * @brief Get games won.
 * @return The number of games which were won by this player.
 */
int Player::getGamesWon(){
   return m_gamesWon;
```

```
}

/**
 * @brief Get games lost.
 * @return The number of games which were lost by this player.
 */
int Player::getGamesLost(){
   return m_gamesLost;
}

/**
 * @brief Get games played.
 * @return The total number of games this player has played.
 *
 * \see getGamesWon
 * \see getGamesLost
 */
int Player::getGamesPlayed(){
   return (getGamesWon() + getGamesLost());
}

/**
 * @brief Add another won game.
 *
 * This shall increase the number of won games by 1.
 */
void Player::addGameWon(){
   m_gamesWon++;
}

/**
 * @brief Add another lost game.
 *
 * This shall increase  the number of lost games by 1
 */
void Player::addGameLost(){
   m_gamesLost++;
}

/**
 * @brief Get the player's name
 * @return The player's name
 */
string Player::getName(){
   return m_playerName;
}
```

Ship.cpp Class

```cpp
#include "ship.h"


/**
 * @brief Ship Standard constructor
 *
 * Should do nothing. This is just needed in order for the std::array in \ref
 * GameBoard to work.
 */
Ship::Ship(){}

/**
 * @brief Ship constructor
 * @param[in] row The row where the aft part of the ship shall be placed
 * @param[in] col The column where the aft part of the ship shall be placed
 * @param[in] shipLength Which length the ship shall have (usually 2 to 5)
 * @param[in] dir Which dir the ship should point to with:
 *   - 0 meaning up
 *   - 1 meaning right
 *   - 2 meaning down
 *   - 3 meaning left
 * @throws std::invalid_argument if either the dir if the ship is an
 * invalid value **or** if parts of the ship would be placed outside of the
 * 10x10 grid.
 *
 * This constructs a ship with the given parameters. The aft section of the
 * ship is always placed at the given row and column. From there on,
 * (shipLength - 1) number of parts are placed in the given dir.
 *
 * If, for example, a ship shall be constructed at `row = 3` and `col = 2`,
 * a length of 3 and `dir = 1` this means the following part objects are
 * created:
 *   - Part at 3 / 2
 *   - Part at 3 / 3
 *   - Part at 3 / 4
 */



Ship::Ship(int row, int col, int shipLength, int dir){

  if(dir == 0){
    if (row - shipLength +1 < 0) {
      throw std::invalid_argument("Out Of Play Grid Exception! - Direction is 0");
    }
    for(int i{0}; i<shipLength ; i++){
      m_parts.push_back(Part(row, col));
      row--;
    }
  }

  else if (dir == 1){
```

```
      if (col + shipLength -1 > 9) {
         throw std::invalid_argument("Out Of Play Grid Exception! - Direction is 1");
      }
      for(int i{0}; i<shipLength ; i++){
         m_parts.push_back(Part(row, col));
         col++;
      }
   }

   else if (dir == 2){
      if (row + shipLength -1 > 9) {
         throw std::invalid_argument("Out Of Play Grid Exception! - Direction is 2");
      }
      for(int i{0}; i<shipLength ; i++){
         m_parts.push_back(Part(row, col));
         row++;
      }
   }

   else if (dir == 3){
      if (col - shipLength +1 < 0) {
         throw std::invalid_argument("Out Of Play Grid Exception! - Direction is 3");
      }
      for(int i{0}; i<shipLength ; i++){
         m_parts.push_back(Part(row, col));
         col--;
      }
   }

   else {
      throw std::invalid_argument("dir is Invalid!");
   }
}



/**
 * @brief Evaluates if this ship extends to the given row and col
 * @param[in] row Row which is to be checked
 * @param[in] col Column which is to be checked
 * @return True if ship extends to the field row/col, false otherwise.
 *
 * This function is used to determine wether or not a ship is present in the
 * given row and col. This is easily answered by iterating over the ship's
 * parts and checking if any of those are located on this position.
 *
 * \see getPartIn
 */

bool Ship::hasPartIn(int row, int col){
   for (Part &part : m_parts){
      if (part.getRow() == row && part.getCol() == col)
         return true;
   }
```

```
    return false;
}


/**
 * @brief Returns the index at which the desired object part exists
 * @return index if object exists otherwise -1
 */

int Ship::whichPartIn(int row, int col){
    for(int i{0}; i<m_parts.size(); i++){
        if (m_parts[i].getRow() == row && m_parts[i].getCol() == col){
            return i;
        }
    }
    return -1;
}



/**
 * @brief Returns the ship's part which is in the given row and col
 * @param[in] row Row of the Part
 * @param[in] col Column of the Part
 * @return Reference to the ship part at the given location.
 * @throws std::invalid_argument if there is no such part at the given
 * position.
 *
 * It is advised to only use this function if you are sure that the given
 * position really contains a part of this ship, e.g. calling \ref hasPartIn
 * first.
 *
 * \see hasPartIn
 */

Part &Ship::getPartIn(int row, int col){

    int ind = whichPartIn(row, col);
    if (ind == -1)
    {
    throw std::invalid_argument("There is no such part at the given position!");
    }
    return m_parts[ind];
}




/**
 * @brief Returns wether or not the ship is damaged.
 * @return True if the ship is damaged.
 *
 * A ship is considered to be damaged of there is at least on of it's parts
 * damaged. If **all** of the parts are damaged, the ship is considered to be
```

```
 * sunk, but this also counts as damaged.
 *
 * \see isSunk
 */

bool Ship::isDamaged(){
   for (Part &part : m_parts){
      if (part.isDamaged())
         return true;
   }
   return false;
}




/**
 * @brief Returns wether or not the ship is sunk.
 * @return True if the ship is sunk.
 *
 * A ship is considered to be sunk if **all** of it's parts are damaged.
 *
 * \see isDamaged
 */
bool Ship::isSunk(){
   for (Part &part : m_parts){
      if (!part.isDamaged())
         return false;
   }
   return true;
}




vector<Part> &Ship::get_m_parts(){
   return m_parts;
}
```

main.cpp CLass

```cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "battleship.h"

int main() {
   srand(time(nullptr));
   Battleship battleship = Battleship("Alex", "Tom");
   battleship.play();

  return 0;
}
```