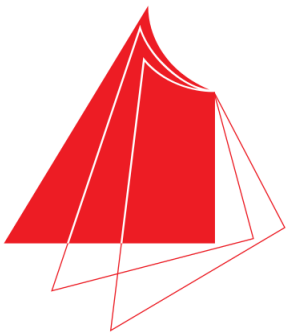

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

EIT Fakultät für Elektro- und Informationstechnik

Projektbericht
Algorithmen und Datenstruktur
Naval Warfare

Christoph Clouser | 4XXXX
cXXXXXXXXXX@hs-karlsruhe.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Aufgabenstellung

Dieses Projekt behandelt eine Strategieoptimierung für eine modifizierte Form des Spiels „Schiffe versenken“.

1.1 Problembeschreibung

Das Spiel „Schiffe versenken“ soll in seinen Regeln erweitert und dazu sollen mehrere Spielstrategien getestet werden. Eines der eigenen Schiffe kann hierbei nach einem Schuss des Gegners um ein Kästchen vorwärts oder rückwärts bewegt oder um 90 Grad gedreht werden. Der vom Gegner getroffene Rasterpunkt darf für drei weitere Schüsse von eigenen Schiffen nicht belegt werden. Sind jeweils nur noch zwei Schiffe vorhanden, dürfen diese nicht mehr bewegt werden. Es soll hierbei der Computer gegen sich selbst spielen. Zunächst sollen zufallsorientiert oder über eine Textdatei vorgegeben die Positionen der Schiffe für zwei Widersacher generiert werden. Für die gilt es dann eine Angriffssequenz zu ermitteln um möglichst schnell, viele Treffer zu erreichen. Jeder Gegner soll seine eigene Strategie in einer Folge von Spielen ermitteln können.
Ansatzpunkte für eine Strategie könnten sein:

1. Auswahl von Punkten über einen Zufallsgenerator
2. Auswahl der Punkte in Form eines parametrierbaren Gitters
3. Punktauswahl über eine Grundstruktur und deren Verzweigung
4. Folgetabelle von Punkten aus erfolgreichen Treffern aus den vorausgegangenen Spielen
5. Raumfüllende Kurven z.B. Hilbertkurve
6. Ausweichmanöver, wenn z.B. eine Rasterstrategie des Gegners entdeckt wird.

1.2 Schnittstellenbeschreibung

1.2.1 Eingabedaten

Ein Schiff muss mindestens 2 Kästchen groß sein.
Spieler1
Schifftyp (Rasterxstart, Rasterystart), (Rasterxend, Rasteryend)
...
Spieler2
Schifftyp (Rasterxstart, Rasterystart), (Rasterxend, Rasteryend)
...

1.2.2 Ausgabe

Grafische Anzeige der Spielfelder mit den Ausgangssituationen und des Verlaufs des Spiels. Gegenüberstellung der Wirksamkeit der Strategien

Naval Warfare

Naval Warfare ist der Name des Programms und stellt eine modifizierte Form des Spiels “Schiffe versenken“ dar.
Es werden zwei Spieler erstellt, die gegeneinander antreten und jeweils durch Algorithmen ihre Aktionen ausführen. Dabei wird die bekannteste Variante zugrunde gelegt. Die Spieler setzen zunächst all ihre Schiffe, wobei kein Schiff über die Kartenbegrenzung oder über Eck gesetzt werden darf. Es gibt ein Schlachtschiff aus fünf Feldern, zwei Kreuzer aus vier, drei Zerstörer aus drei und vier Uboote aus zwei Feldern bestehend. Zwischen den Schiffen muss ein freies Wasserfeld erhalten bleiben. Die Ausrichtung der Schiffe ist entweder horizontal oder vertikal, niemals jedoch diagonal!
Die Spieler nennen nacheinander eine Koordinate als Angriffspunkt und der gegnerische Spieler antwortet darauf hin je nach Ausgang mit einem der folgenden Ausrufen:

- “Wasser!“: es wurde kein Schiff getroffen
- “Treffer!“: es wurde ein Schiffsfeld getroffen
- “Versenkt!“: es wurde das letzte Schiffsfeld getroffen und damit das Schiff zerstört

Die Erweiterung sieht nun vor, dass darauf hin der gerade angegriffene Spieler ein Schiff entweder ein Kästchen nach vorne oder nach hinten bewegen oder um 90 Grad drehen darf. Bei dieser Aktion darf kein Feld betreten werden, dass in den letzten drei Runden beschossen wurde und es müssen mehr als zwei Schiffe vorhanden sein, die noch nicht als versenkt gelten. Dies wird nun solange wiederholt, bis alle Schiffe eines Spielers versenkt wurden. Ein Schiff gilt als versenkt, wenn all seine Segmente mindestens einmal getroffen wurden. Eine solche Abfolge, in der beide Spieler sowohl Angriffs- als auch Bewegungsaktion ausgeführt haben, wird als eine “Runde“ verstanden. Ein Treffer ist an ein Schiffsfeld gebunden und wird mit einem roten “X“ markiert. Auf die Markierung des “Wasser!“-Ausrufs wurde

verzichtet, trotzdessen sie im normalen Spiel mit einem “O“ markiert wird. Doch hier ist der Informationsgehalt sehr gering, da bereits in der vierten Runde nach dem Schuss diese Information ungültig sein könnte, da ein gegnerisches Schiff die Position eingenommen haben könnte.

2.1 Aufbau des Programms

Das Programm gliedert sich in die folgenden vier Klassen:

- ships - ships.cpp, ships.h
  mit allen Informationen für ein Schiff
- player - cplayer.cpp, cplayer.h
  mit allen Informationen für einen Spieler und dessen graphische Darstellung
- sad - sad.cpp, sad.h
  mit den Informationen für den Algorithmus
- pattern - pattern.cpp, pattern.h
  speichert und erstellt Mustersequenzen

Sie sind eingebettet in die Programmierumgebung GDE (ADS_GDE_3_2015src.zip, Version: 14.12.2016) und der Rundenablauf ist in der user.cpp beschrieben.

Achtung: die GDE wurde allerdings dahingehend verändert, sodass mehrere Bitmap-Dateien (.bmp) geladen und übereinander gelegt werden können (siehe Seite 5).

Die Graphikdateien befinden sich im Ordner “bmps“ und die Textdateien für Platzierungs-, Angriffs- und Bewegungsaktion in“input“.
Nachdem die GDE gestartet wurde, wird das Programm durch “run“ gestartet. Es wird nun der Hintergrund geladen und in der Konsole nach einer Startaufstellungsnummer gefragt. Das Spiel startet sobald die Eingabe mit Enter bestätigt wurde. Beide Spieler nutzen im Default-Modus den SaD-Algorithmus.

Von der *user.cpp* aus werden die zwei Spieler der Klasse *cplayer* erstellt, dabei werden die Schiffsvektoren der Spieler mit den zehn Einträgen der Klasse *ships* initialisiert. Ebenfalls wird für jeden Spieler eine Instanz des SaD-Algorithmus initialisiert. Die Klasse *pattern* muss explizit aufgerufen werden und ist im Moment auskommentiert im Quellcode, da sie nur bei Bedarf Koordinatendateien (.txt) mit und ohne Ausrichtungsinformation erstellt.

2.2 Graphische Darstellung

Die Methoden für die grapische Darstellung finden sich in der Klasse des Spielers “cplayer.cpp“ und werden detailiert im Abschnitt 3.3.2 auf Seite 23 behandelt.
Die graphische Darstellung wird über die zur Verfügung gestellte GDE verwirklicht. Diese bot in ihrer Originalfassung die Möglichkeit eine Bitmap-Datei in den Buffer ab dem Koordinatenursprung zu laden und dann im Fenster der GDE anzeigen zu lassen. Um eine zufriedenstellende Darstellung für Naval Warfare zu erreichen wurde die GDE dahingehend umgeschrieben, dass beliebig viele Bitmap-Dateien ab verschiedenen Koordinaten im Fenster der GDE dargestellt werden können. Diese Dateien befinden sich in dem Unterordner “bmps“ und sind 16-bit Bitmap-Dateien.

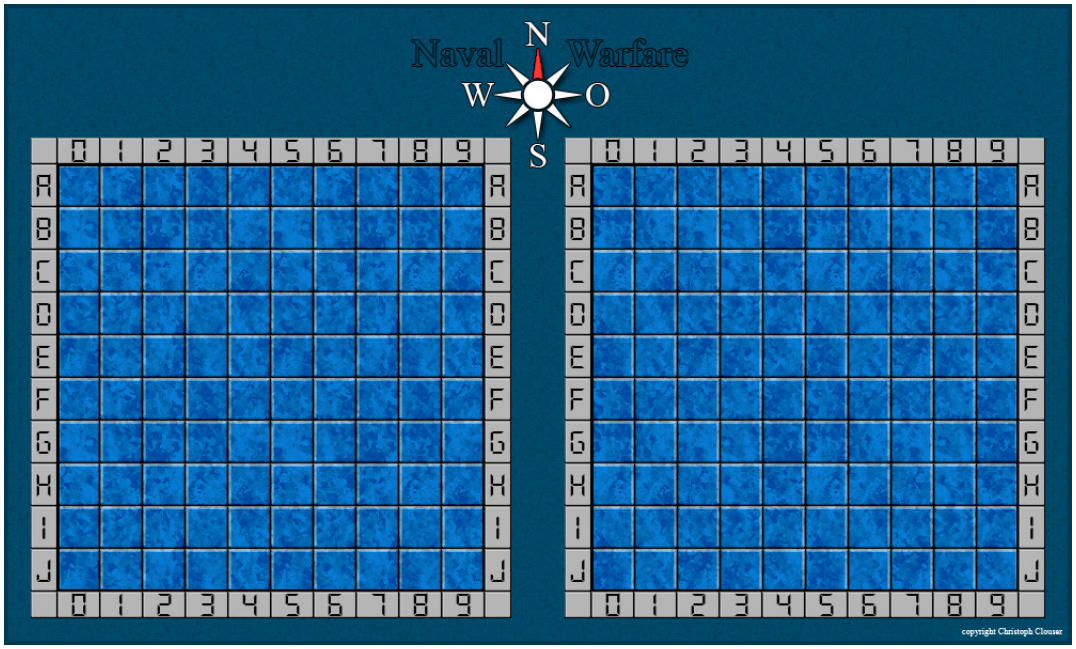

Abbildung 2.1: Spielfeld von Naval Warfare

Ein zusätzlich eingefügte globale Funktion ist die “drawships()“-Funktion, die auf die bereits in der GDE implementierte “OnFileOpen“-Funktion zurückgreift.

Listing 2.1: Globale drawship-Funktion in graphicfunctions.cpp

```
1 void drawship(CString ifilename, int ix, int iy)
2 {
3         theApp.vw->CGDE_3View::OnFileOpen(ifilename, ix, iy);
4         updatescr();
5 }
```

Durch diesen kleinen Hack in Listing 2.1 und 2.2 ist die Funktion Bitmap-Dateien über die GDE öffnen zu können verloren gegangen. Allerdings wird diese Funktion, wie sie bestand, nicht in diesem Programm benötigt und ist damit obsolet.

Listing 2.2: Angepasste OnFileOpen-Funktion in GDE_3View.cpp

```
1  void CGDE_3View::OnFileOpen(CString ifilename,int ix, int iy)
2  {
3          CString filename = ifilename;
4          CDib dib;
5          try {
6                  dib.Load(filename);
7          } catch(CImageException& e) {
8                  ::AfxMessageBox(LPCTSTR(e.what()));
9          }
10         CGDE_3Doc* pDoc = CGDE_3View::GetDocument();
11         dib.Load(filename);
12         int ww=dib.GetWidth();
13         int hh=dib.GetHeight();
14         dib.Draw(&(pDoc->buffer),ix,iy,ww,hh,SRCCOPY);
15 }
```

Das Laden und Darstellen mehrerer Bitmap-Dateien stellt somit kein Problem mehr dar, allerdings ist das Aktualisieren der Anzeige mit einer nicht für Echtzeitanimationen optimierte Umgebung problematisch. Es wird also nach einer Möglichkeit gesucht, Bitmap-Dateien darzustellen und auch in dem Zeichenbereich bzw. der Darstellungsebene zu verschieben. Elemente die nicht von Änderungen betroffen sind sollen auch nicht verändert werden.
Im ersten Ansatz wurde versucht die Anzeige über die zur Verfügung gestellte Funktion “clrscr()“ (siehe Dokumentation zur GDE) zu realisieren und dann alle Elemente neu zu zeichnen. Diese Funktion löscht die Darstellungsmatrix, schreibt den Buffer weiß und aktualisiert dann den Darstellungsbereich im Fenster der GDE. Doch dieser Ansatz scheitert in der Echtzeitfähigkeit, da er auf Grund des Schreibvorgangs über den Buffer und in die Darstellungsmatrix zu lange dauert und somit ein weißes Flimmern verursacht.

Abbildung 2.2: Bewegung

Abbildung 2.3: Rotation

Im zweiten Ansatz wurde daher von der “clrscr()“ abgewichen und stattdessen auf eine andere Möglichkeit zurückgegriffen. Statt alle graphischen Elemente in der Darstellungsebene zu löschen und neu zu zeichnen, wird nur das gerade betroffene

Element überschrieben und zwar genau so wie auch die Hintergrundgrafik (Abb 2.1) überschrieben wird mit der “drawships()“-Funktion. Hierzu mussten weitere draw-Funktionen (siehe Seite 23) implementiert werden, die bei Veränderung der Position oder Lage des Schiffes aufgerufen werden. Diese Elemente sind leere Wasserfelder, jedoch angepasst an die Aktion und den Typ des Schiffes, ob es sich nur ein Kästchen bewegt (Abb.2.2) oder eine Rotation (Abb. 2.3) ausführt. Dieser Ansatz liefert zufriedenstellende Ergebnisse, sodass eine Partie in Echtzeit betrachtet werden kann.
Die Darstellung der Schiffe ist nicht nur abhängig vom Typ des Schiffes, sondern auch von der Ausrichtung, die sich an den Himmelsrichtungen auf dem Spielfeld orientiert:

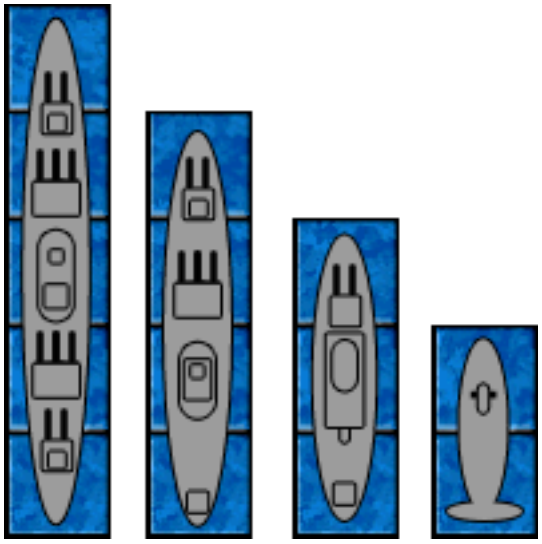

Abbildung 2.4: Schlachtschiff, Kreuzer, Zerstörer, Uboot

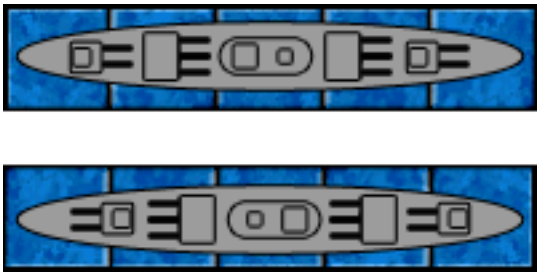

Abbildung 2.5: Ausrichtung eines Schlachschiffs (Ost und West)

Diese Darstellung ist intuitiv verständlich und übersichtlich. Dies ist auch der Grund weshalb vom Eingabeschema der Aufgabenstellung abgewichen wurde: statt ein Schiff vom Startpunkt $S_1 = (x_1, y_1)$ bis zum Endpunkt $S_2 = (x_2, y_2)$ zu definieren wird es über seinen Startpunkt $S = (x, y)$ und seiner Ausrichtung “Nord, Süd, West, Ost“ definiert. Hierbei wurde auch das Drehzentrum festgelegt, es ist der Startpunkt $S = (x, y)$, der sofort ersichtlich ist, da er im Heck des Schiffes liegt. Dies wurde auch in Hinsicht auf eine Erweiterung implementiert, auf die im Kapitel Aussicht eingegangen wird.
Detailierte Informationen zur Geometrie des Spielfeldes und den sich daraus ergebenden Berechnungen finden sich auf Seite 12.

2.3 Platzierung der Schiffe

Die Initialisierung der Spieler beinhaltet die Übergabe von je einem Initialisierungswert P1='1' und P2='2', sodass automatisch die draw-Methoden und weitere Berechnungen des 2. Spielers um den entsprechenden Offset verschoben sind. Die clrscr()-Funktion stellt den Hintergrund im Fenster der GDE dar, woraufhin die Abfrage eines Startwertes gefordert wird. Dieser Offset wird zum Auslesen der Startposition aus der *./input/start.txt* benutzt.

Listing 2.3: Hintergrund und Spieler Initalisierung

```
1  clrscr();                                    // Zeichne Hintergrund
2  wait(5);                                     // Warten auf Aktualisierung
3  player *ptr_player;                          // Zeiger auf Spieler
4  ptr_player = new player[2]{P1, P2};          // Zwei neue Spieler
5  // Eingabeaufforderung:
6  std::cout << "Willkommen bei Naval Warfare (beta)!" << std::endl
7          << "Bitte geben Sie einen Startpositionswert zwischen 0-99 ein:";
8  std::cin >> startposition_offset;
9  startposition_offset = startposition_offset % FIELDLENGTH;
10 (ptr_player+0)->player::input(startposition_offset);     // Spieler 1
11 (ptr_player+1)->player::input(startposition_offset);     // Spieler 2
```

Dabei wird *./input/start.txt* solange durchlaufen, bis das Schiff eine gültige Position gefunden hat. Es werden für die Koordinate alle Ausrichtungen getestet, bevor die nächste Koordinate abgerufen wird. Die Platzierung der Schiffe beginnt bei dem größten Schiffstyp, dem Schlachtschiff und endet mit dem kleinsten, dem Uboot. Hierbei werden zwei verschiedene Testfunktionen durchlaufen, die jeweils sicherstellen, dass die Schiffe weder außerhalb der Karte, noch überlappend oder angrenzend aneinander platziert werden. Die erste Testfunktion findet sich in der Klasse *ships* (siehe Seite 16) und testet, ob das Schiff die Kartengrenze überschreitet. Die zweite Testfunktion ist in der Klasse *cplayer* enthalten (siehe Seite 22) und testet ob der zu platzierende Koordinatenvektor des Schiffes eine regelkonforme Lage zu den anderen Koordinatenvektoren der Schiffe hat. Hierzu wird ein Koordinatenvektor des zu platziernden Schiffes auf der Grundlage der Startkoordinate $S = (x, y)$, dessen Typ und Ausrichtung erstellt, der den Umkreis des Schiffes enthält. Die Einträge dieses Koordinatenvektors werden nun gegen alle Einträge der bereits platzierten Schiffe bzw. deren Koordinatenvektoren getestet. Gibt es eine Übereinstimmung, so ist die Platzierung regelwidrig und es wird entsprechend die nächste Ausrichtung getestet oder die nächste Koordinate.

2.4 Spielrunde

Der Ablauf der Spielrunde ist in der *user.cpp* der GDE beschrieben, alle benötigten Verweise auf die benutzten Klassen finden sich in der user.h Header-Datei. Auf Seite 14 findet sich eine graphische Darstellung eines komplette Spielzugs.
Die Initialisierung beinhaltet die Erstellung der Spieler und den Zugriff auf eine Programmierschnittstelle (API) von Microsoft, den QuerryPerformanceCounter (QPC), zur präzisen Zeitmessung ($1\mu s$). Die QPC ist auf nativen Code, der auf *einer* Maschine ausgeführt wird, ausgelegt und von der Systemzeit unabhängig:

Listing 2.4: QuerryPerformanceCounter (QPC)

```
1 LARGE_INTEGER frequency;                    // Ticks pro Sekunde
2 LARGE_INTEGER t1, t2;                       // Ticks
3 double elapsedTime;                         // Vergangene Zeit, wird berechnet
4 QueryPerformanceFrequency(&frequency);      // Abfrage: ticks/s dieses Systems
5 QueryPerformanceCounter(&t1);               // Abfrage: momentaner Stand
6 \\ ZU MESSENDER CODE HIER
7 QueryPerformanceCounter(&t2);               // Abfrage: momentaner Stand
8 \\ Berechnung der vergangenen Zeit in ms:
9 elapsedTime=(t2.QuadPart-t1.QuadPart)*1000.0/frequency.QuadPart;
```

Der Spielzug des **zweiten Spielers** ist durch den Zugriff auf die Methoden der Spielerklasse des ersten Spielers gekennzeichnet. Dies mag im ersten Moment ungewöhnlich erscheinen, aber es erlaubt die konsequente Nutzung des Spielerzeigers $(ptr\_player + 0)$ für alle Aktionen in der Spielrunde des zweiten Spielers:

Listing 2.5: Spielzug des 2. Spielers

```
1 round++;                                              // zaehle Runde hoch
2 std::cout << "Runde: " << round << std::endl;        // print: Runde
3 do {
4  coordinates = (ptr_player + 0)->player::get_input(FILENAME_MUSTER, ii);
5  vergleich = (ptr_player + 0)->player::check_coor(coordinates);
6  if (vergleich)
7  {
8   ii++;
9   ii = ii % (2 * FIELDLENGTH);
10 }
11 } while (vergleich);
12 if (!((ptr_player + 0)->player::get_stopmuster())) // Algorithmus?
13 {
14  ii++;
15  ii = ii % (2 * FIELDLENGTH);
16 }
17 (ptr_player + 0)->player::check_hit(coordinates); // Treffer?
18 if ((ptr_player + 0)->player::get_canmove())
19                  {(ptr_player + 0)->player::input_action(round);}
```

```
20 turn = 0;                                    // Spieler 1
```

Nachdem der Rundenzähler aktualisiert wurde, beginnt die Koordinatenabfrage aus der entsprechenden Datei (Zeile 4). Diese Koordinate wird mit den Einträgen des Treffervektors verglichen (Zeile 5), so lange bis noch kein Treffer vorlag (Zeile 11). Während der Algorithmus aktiv ist (Zeile 12) wird ebenfalls der Zugriffsoffset in die Datei weitergezählt. Dies ist nicht zwingend notwendig, wegen der Abfrage des Treffervektors, aber es streut die Mustersequenz noch etwas.
Insgesamt lässt sich die Abfolge des Zuges genau so auf den ersten Spieler übertragen, nur das der Spielerzeiger ($ptr\_player + 1$) ist.
Die Züge sind in eine Schleife eingebettet, die am Ende jedes Zsuges das Flag _isalive beider Spieler abfragt:

Listing 2.6: Schleife der Runde

```
1 do {
2  if (turn >= 1)
3                  { // CODE SPIELER 2 }
4  else
5                  { // CODE SPIELER 1 }
6 } while (ptr_player->player::get_isalive() &&
7                          (ptr_player + 1)->player::get_isalive());
```

Je nach Ausgang wird nun die Zeitmessung gestoppt und eine kurze Mitteilung in der Konsole ausgegeben, wer gewonnen hat, wie lange das Spiel gedauert hat und wieviele Runden insgesamt gespielt wurden. Das Spielfeld mit dem Ergebnis des Spiels ist weiterhin in der GDE sichtbar. Ein neues Spiel kann durch “run“ direkt gestartet werden.

2.4.1 Angriff

Ein Angriff wird durch die bereits erwähnte Abfrage der Koordinatendatei des Spielers eingeleitet und wird ggf. erweitert durch den SaD-Algorithmus. Zu Beginn wird bis ein Treffer erfolgt immer die Koordinatendatei durchlaufen. Kommt es zu einem Treffer wird das flag_stopmuster gesetzt, die Mustersequenz aus der Koordinatendatei angehalten und der SaD-Algorithmus übernimmt die Erstellung der nächsten Angriffskoordinaten. Hierzu wird eine Angriffsmatrix auf Basis der letzten Trefferkoordinate erstellt, eine (4x4)-Matrix. In den folgenden Zügen wird mit diesen Koordinaten und resultierenden Treffern herausgefunden wie das Schiff liegt, entwededer horizontal oder vertikal. Wurde ein zweiter Treffer verzeichnet, so ist die Lage des Schiffes eindeutig und es wird nur noch auf Koordinaten der

Angriffsmatrix zugegriffen, die auf der Geraden liegen, die durch die zwei Treffer beschrieben wird. Überläufe über den Kartenrand werden dabei vom SaD-Algorithmus erkannt und übersprungen, ebenfalls bereits getroffene Bereiche. Wird eine der Abbruchbedinungen erreicht, so wird der SaD-Algorithmus zurückgesetzt und wieder auf die Mustersequenz aus der Koordinatendatei zurückgegriffen, bis der nächste Treffer erfolgt. Die Treffer werden jeweils direkt nach ihrer Bestätigung eingezeichnet, in dem Treffervektor des Schiffes und im Treffervektor des angreifenden Spielers vermerkt. So kann getestet werden, wann die _isalive-Flag eines Schiffes ungültig wird, ob ein angreifender Spieler dieses Feld bereits angegriffen hat und entsprechend eine andere Koordinate abrufen muss.

2.4.2 Bewegung

Der Positionsvektor eines zufällig gewählten Schiffes aus dem Schiffsvektor des Spielers wird getestet, ob eine der folgenden Bewegungsaktionen zu einer regelkonformen Position auf der Karte führt:

- ein Feld vorwärts fahren
- ein Feld rückwärts fahren
- eine Drehung um 90Grad im Uhrzeigersinn
- eine Drehung um 90Grad gegen den Uhrzeigersinn

Die erste gültige Bewegungsaktion wird durchgeführt. Hierzu wird auf die Test-Methode in der Klasse *cplayer* zurückgegriffen, die bereits für die Platzierung genutzt wurde. Es wird ein temporärer Positionsvektor erstellt und getestet ob dieser platziert werden darf. Ist die Position regelkonform, so wird der temporäre Vektor zum aktuellen Positionsvektor des Schiffs und es werden die draw-Methoden in der gleichen Klasse aufgerufen. Je nach Bewegungsaktion wird entweder ein leeres Wasserfeld gezeichnet oder ein leeres Wasserfeld der Länge des Schiffes. Dies löscht die Darstellung der alten Position des Schiffs in der GDE, die neue Position kann nun gezeichnet werden und danach die eventuell schon bestehenden Treffer im Treffervektor des Schiffes.

Die Klassen im Detail

Die Klasse *ships* beinhaltet alle relevanten Informationen **eines** Schiffs: den Typ, die Koordinaten, die Ausrichtung (Himmelsrichtung), den Besitzer, die Trefferpunkte und eine Abfrage, ob das Schiff noch regelkonform auf der Karte liegt.
Die Klasse *cplayer* verwaltet alle Informationen **eines** Spielers: die Nummer, den Namen, die Schiffe, ihre Platzierung und Bewegung, wie und ob diese noch bewegt werden dürfen, die letzten drei Treffer des Gegners, die eigenen Treffer während des Spiels, ob des Spielers Schiffe zerstört sind, den SaD-Algorithmus und die graphische Darstellung der eigenen Schiffe und Treffer.
Die Klasse *sad* erstellt nach einem Treffer ein Angriffsmuster und testet, welche Lage das gerade getroffene Schiff hat und versucht es in den folgenden Runden zu zerstören.
Die Klasse *pattern* erstellt und speichert Koordinatenmuster für Angriffs- und Bewegungsaktionen in Text-Dateien (.txt).

Der geometrische Aufbau des Spielfeldes findet sich in der Header-Datei *cplayer.h* und ist essentiell für die Berechnungen in den Klassen. Das Spielfeld ist zunächst durch die Koordinaten 0 bis 9 und A bis J und die Himmelsrichtungen (Nord, Süd, West, Ost) beschrieben. Überträgt man dies nun auf die x- und y-Koordinaten der GDE so startet die Rasterkarte bei XSTART = 50px und YSTART = 150px. Diese Werte ergeben sich aus der Graphikdatei “Spielfeld.bmp“, die aus diesem Grund so konzipiert wurde. Die Feldgröße beträgt 40x40px, der Offset für den zweiten Spieler beträgt 500px in x-Richtung. In *cplayer.h* können darüber hinaus Offsets für die Schiffs-Bitmaps und das Treffer “X“ eingestellt werden.

3.1 Allgemeine Speicherorganisation

Die Speicherorganisation ist mit Blick auf das kartesische Koordinatensystem verwirklicht, sodass wichtige Informationen über Vektoren und eindimensionale Matrizen organisiert werden. Auch wenn Vektoren eindimensionale Matrizen sind

wird hier ganz bewusst dieser Unterschied betont, denn die Speicherorganisation in C++ erlaubt es auch mehrdimensionale Matrizen zu nutzen, doch sind diese in der Laufzeit einer eindimensionalen unterlegen. So wird beispielsweise das Angriffsmuster des SaD-Algorithmus in einer eindimensionalen Matrix verwirklicht, obwohl diese als (4x4)-Matrix begriffen und entsprechend auf sie zugegriffen wird. Die Laufzeitoptimierung ist nicht der einzige Vorteil, auch der gebräuchliche Umgang mit Vektorstrukturen erlaubt es einen leserfreundlicheren Code zu generieren. Dies fällt insbesondere in Hinsicht auf das Debuggen und der Erweiterbarkeit durch Dritte ins Gewicht, sodass auch ein Fokus auf Teamarbeit gelegt wird, weil auf eine bekannte, gemeinsame Basis zurückgegriffen wird.

Ebene	**Spiel**	**Spieler**	**SaD-Algorithmus**	**Schiffe**	**Muster**
Datei	user.cpp/.h	cplayer.cpp/.h	sad.cpp/.h	ships.cpp/.h	pattern.cpp/.h
Informationen	Runde, Reihenfolge	Name, Nr, Status	Start/Stop-Flag	Typ, Ausrichtung, Status	Dateipfad
Vektoren	Spieler	Schiffe, Treffer	Angriffsmuster	Treffer, Koordinaten	-
Methoden	-	**Get/Set-Methoden**			**Koordinaten:** Muster, Zufall, Aktion
		Test: Treffer, Bewegung, Platzierung, Status **Eingabe:** Koordinate, Aktion **Erstellung:** Schiffe **Zeichnen:** Schiffe, Treffer	Angriffsmustererstellung Koordinatenauswahl	**Test:** Platzierung auf Karte, Status **Eingabe:** Startkoordinate, Ausrichtung **Erstellung:** Koordinatenvektor nach Aktion, Umkreisvektor	
Anmerkung	Zeitmessung	Definitionen zur Geometrie, Dateien und Verzeichnissen in Header			bei Bedarf

Abbildung 3.6: Speicherorganisation und Klassenhierarchie

Eine Übersicht über die verwalteten Informationen bietet Abbildung 3.6 . Die Klassen lassen sich verschiedenen Ebenen einer Klassenhierarchie zuordnen. Die Schnittstellen sind dabei vielfältig, einzelne Zugriffe werden in den folgenden Abschnitten beispielhaft erläutert, sodass das Zusammenspiel der Methoden deutlich wird. Ab der Spieler-Ebene sind alle Informationen geschützt und nur über die entsprechenden Set/Get-Methoden erreichbar, dies gilt insbesondere auch für die Vektoren. Die Muster-Klasse steht neben der Hierarchie, da sie nur bei Bedarf aufgerufen wird und die Dateien mit Mustersequenzen erstellt. Diese Dateien finden dann wiederum Anwendung in den anderen Klassen.
In der folgenden Abbildung 3.7 auf Seite 14 wird der Zug eines Spielers mit dem Programmverlauf dargestellt. Um die Lesbarkeit nicht völlig aufzugeben wurde auf die Darstellung einiger Zugriffe, z.B. die Umkreiserstellung, in der Klasse “ships“ bewusst verzichtet.

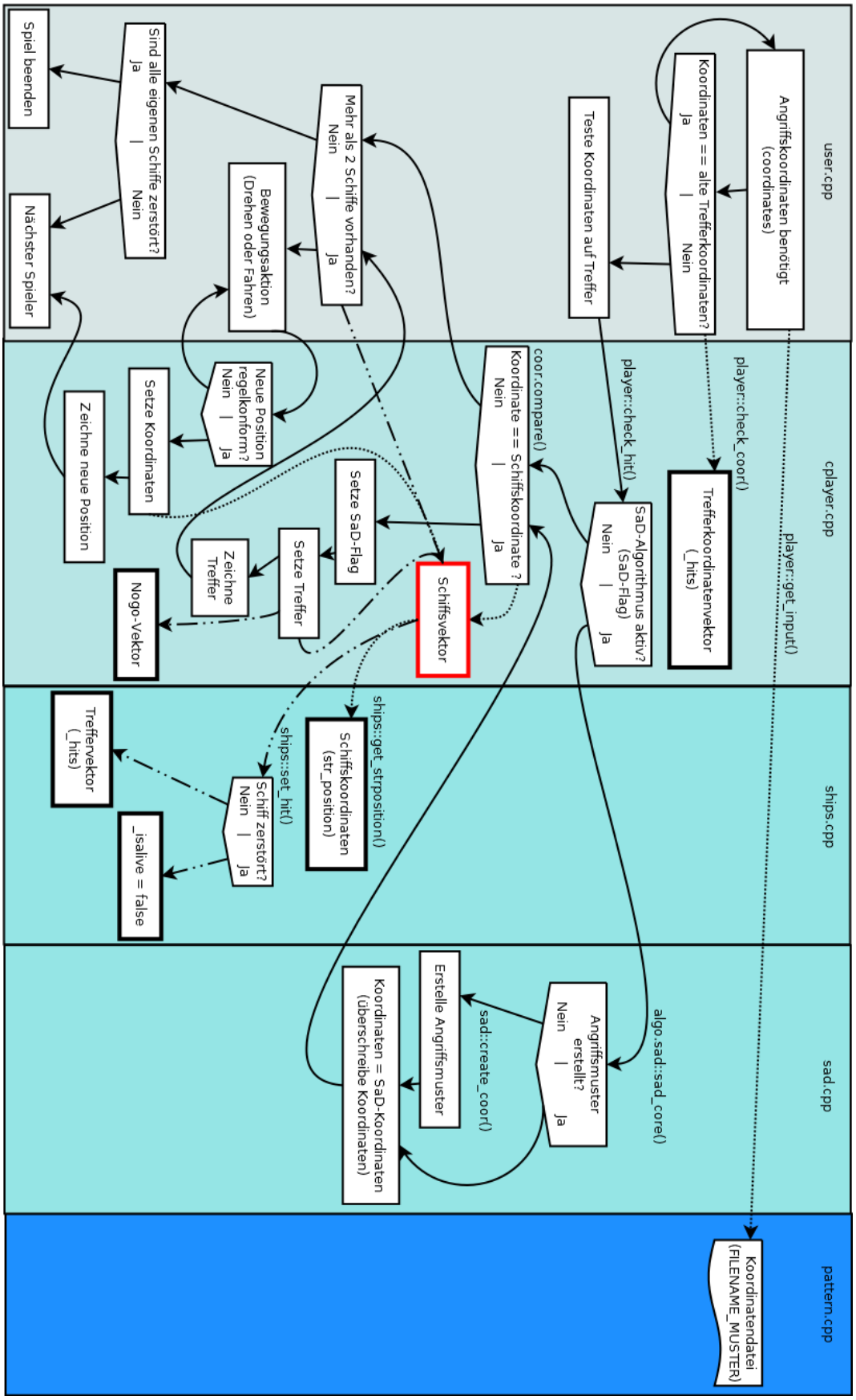

Abbildung 3.7: Spielerzug: Abfolge und Zugriffe (Auswahl)

3.2 Schiff: ships

Alle wichtigen Informationen zu einem Schiff finden sich in dieser Klasse. Definitionen zu den benutzten Begriffen finden sich in der ships.h-Datei.

- **Typ (_type)**: SCHLACHTSCHIFF, KREUZER, ZERSTOERER, UBOOT
- **Ausrichtung (_orientation)**: NORD, SUED, WEST, OST
- **Besitzer (_owner)**: P1, P2
- **Status (_isalive)**: true, false
- **Treffervektor (_hits)**: HIT, NOHIT
- **Positionsvektor (_position, str_position)**: 0-9, A-J

Der Zugriff auf diese Objekte ist nur per Set/Get-Methoden möglich. Je nach Anforderung gibt es reine Read und Read/Write-Methoden.
Der **Konstruktor** ships(char itype, char iowner) ist dabei dem default-Konstruktor vorzuziehen, ansonsten müssen Typ und Besitzer extra abgefragt werden. Durch die Angabe des Typs des Schiffes werden der Treffervektor und der Positionsvektor aufgebaut, sodass eine Eintragung bei der Platzierung möglich ist. Die Vektoren sind von großer Bedeutung und existieren deshalb bis zum Spielende, da ihre Informationen für die Anzeige für die gesamte Programmlaufzeit unabdingbar sind. Weitere wichtige Methoden sind hier:

- **Set/Get-Methoden** (insbesondere: set_position)
- **boundcheck()**: testet, ob Schiff sich auf Karte befindet
- **boundcheck_action()**: testet, ob Schiff sich nach Aktion noch auf Karte befindet
- **check_isalive()**: testet, ob Schiff zerstört ist
- **create_newposition()**: erzeugt temporären Positionsvektor auf Grundlage einer Aktion
- **create_strposradius()**: erzeugt temporären Positionsvektor mit Umkreiskoordinaten

Die **boundcheck/boundcheck_action**-Methoden finden immer dann Anwendung, wenn der Positionsvektor eines Schiffes verändert werden soll. Dies ist bei der Platzierung (boundcheck) und bei jeder Aktion (boundcheck_action) der Fall. Es wird nur getestet, ob ein Überlauf über den Kartenrand auftritt und ein true/false (Boolean) zurürckgegeben. Bevor die Platzierung eines Schiffes in dessen Positionsvektor eingetragen wird, wird also erst einmal auf diesen Überlauf getestet.
Die **check_isalive**-Methode durchläuft den Treffervektor, zählt und vergleicht die HIT Einträge mit dem Wert des Typs des Schiffes und gibt ein true/false (Boolean) zurück. Der Typ-Wert korrespondiert mit der Länge des Treffer- und Koordinatenvektors.
Die **set_position**-Methode füllt den Positionsvektor mit Koordinaten auf Grundlage der x, y Koordinaten und der Ausrichtung. Der Punkt S(x,y) stellt dabei den Ursprung des Schiffes dar und mit der Ausrichtung ergibt sich dann eine eindeutige Lage. Bevor die Funktion allerdings mit den Berechnungen der Koordinaten beginnt wird die **boundcheck**-Methode aufgerufen, um einen Überlauf auszuschließen:

Listing 3.7: Ausschnitt aus set_position

```
1  bool ships::set_position(char ix, char iy, char iaction)
2  {
3   unsigned short i, ii;
4   int size = this->ships::get_type();     // Typwandlung, Wert = Vektorlaenge
5   // Test ob Schiff noch auf Karte:
6   bool vergleich = this->ships::boundcheck_action(ix, iy, iaction);
7   // Test ob Eingabe gueltig und Schiff noch auf Karte:
8   if ((ix >= '0') && (ix <= '9') && (iy >= 'A') && (iy <= 'J') && (vergleich))
9   {
10          for (i = 0, ii = 0; i < size; i++)
11          {
12                  switch (this->ships::get_orientation())
13                  {
14                  case WEST: // nur x-Wert aendert sich ruecklaufend
15                          if (i % 2 == 0)
16                          {
17                                  *(this->ships::_position.data() + i) = ix - ii;
18                                  ii++;
19                          }
20                          else
21                          {
22                                  *(this->ships::_position.data() + i) = iy;
23                          }
24                          break;
25          ...
```

In dem Beispiel wird der Schiffsvektor also als die Menge von Koordinaten begriffen über die er sich erstreckt, mit einem Urpsrung in S(x,y) und der entsprechenden

Ausrichtung. Dies ist notwendig, da ein Schiff aus mehreren Segmenten besteht. Die **create_newposition**-Methode erstellt einen Positionsstring str_coor auf Grundlage von einer übergebenen x und y Koordinate und einer Aktion (NORD, SUED, WEST, OST, UZ, GZ). Hierbei entspricht die Bewegung um ein Kästchen der Ausrichtung des Schiffes. Hat also ein Schiff die Ausrichtung SUED und die Aktion ist NORD, so wird ein Positionsstring zurückgegeben, der das Ursprungsfeld des Schiffs ein Kästchen nach "hinten" legt. Wäre allerdings die Aktion WEST, so würde dies als unzulässige Aktion gewertet werden und der String "000" (initialisierter Wert) zurückgegeben:

Listing 3.8: Ausschnitt aus create_newposition

```
1   ...
2   switch (action)
3   {
4   case NORD:
5   {
6           switch (orientation)
7           {
8           case NORD:              // Faehrt eins vorwaerts
9                   x = ix;
10                  y = iy - 1;
11                  break;
12          case SUED:              // Faehrt eins rueckwaerts
13                  x = ix;
14                  y = iy + 1;
15                  break;
16          default:                // seitlich Fahren nicht erlaubt
17                  return  str_coor;
18          }
19          break;
20  }
21  ...
```

Bei der Aktion GZ (gegen Uhrzeigersinn) oder UZ (im Uhrzeigersinn) bleiben die x und y Koordinaten unverändert, es ändert sich nur die Ausrichtung und der zurückgegebene String unterscheidet sich nur in der Ausrichtungsangabe.

3.2.1 Umkreisvektor eines Schiffs

Die **create_strposradius**-Methode ist eine sehr wichtige und häufig genutzte Methode, da bei jeder Aktion und Platzierung eines Schiffes der Abstand zu einem weiteren Schiff mindestens ein Kästchen betragen muss. Hierfür bedarf es also einer Fläche die über die Koordinaten der Schiffsegmente hinaus geht. Deshalb soll sie nun genauer betrachtet werden.

Auf der Grundlage der übergebenen x,y Koordinaten (Ursprung oder neuer Ursprung) und der Ausrichtung wird ein String-Vektor erstellt. Dessen Einträge beinhalten alle Koordinaten der Schiffsegmente und der Kästchen, die sich in einem 1-Kästchen-Radius zu den Schiffsegmenten befinden. Somit ist ein Schiff und sein 1-Kästchen Umkreis erfasst.
Nachdem die Ursprungskoordinate und die Ausrichtung des Schiffs an die Funktion übergeben wurde, wird zunächst die Anzahl an Kästchen für den Schiffstyp berechnet, danach die Variable a, b und k abhängig von der Ausrichtung ausgewählt und dann folgt die Berechnung der Koordinaten.
Die Anzahl der Kästchen ergibt sich aus dem Schiffstyp, da dieser nach Typwandlung (char in int) mit Offsetabgleich der Länge des Schiffs entspricht, hinzu addiert werden noch das Kästchen vor und hinter dem Schiff und die Multiplikation mit drei ergibt dann die gewünschte Anzahl:

Listing 3.9: Ausschnitt 1: Anzahl der Kästchen

```
1  std::vector <std::string> stringvec;
2  stringvec.resize(((this->ships::get_type() - '0')+2)*3);
```

Um den Code möglichst kurz zu halten, wird die Symmetrie des Spielfeldes ausgenutzt und alle Fälle über die Zuweisung von x_0, y_0 an a,b und den Wert 1 bzw -1 an k abgehandelt. Beispielhaft werden die Fälle für die Ausrichtung NORD und SÜD betrachet:

Listing 3.10: Ausschnitt 2: Zuweisung

```
1 switch (iorientation)
2 {
3  case NORD:
4          a = y0;
5          b = x0;
6          k = (-1);
7          break;
8  case SUED:
9          a = y0;
10         b = x0;
11         k = 1;
12         break;
13 ...
```

Der Grund für die Zuweisung wird ersichtlich, wenn die Berechnung betrachtet wird. Es muessen $3 * (type + 2)$ Durchlaeufe gemacht werden, um alle Kaestchen berechnen zu können. Je nach Ausrichtung muss entweder die x oder die y Koordinate gehalten, die y oder x Koordinate durchlaufen und dann erste die x bzw. y Koordinate für den nächsten Durchgang hochgezählt werden:

Listing 3.11: Ausschnitt 3: Berechnung

```
1 for (i = -1, iii = 0; i < 2; i++)
2         for (ii = -1; ii < (fields -1); ii++, iii++)
3         {       // Symmetrie:
4                 if ((iorientation == NORD) || (iorientation== SUED))
5                 {
6                         str_insert[0] = b + i*k;  // x erst halten
7                         str_insert[1] = a + ii*k; // y laeuft
8                 }
9                 else    // fuer WEST und OST:
10                {
11                        str_insert[0] = a + ii*k; // x laeuft
12                        str_insert[1] = b + i*k;  // y erst halten
13                }
14                stringvec[iii] = str_insert;  // in String-Vektor
15        }
16 }
```

Um wieder zu dem Beispiel zurück zu kommen, wird nun angenommen die Ausrichtung des Schiffs, eines Uboots, sei NORD. Somit sind zwölf Koordinaten zu berechnen. Der String str_insert beinhaltet an der ersten Stelle die x, an der zweiten die y Koordinate. Den Variablen sind die folgenden Werte zugeordnet worden: $a = y_0$, $b = x_0$, $k = (-1)$. Es wird nun von dem Ursprung ausgehend die Koordinate des (x_0+1, y_0+1) Kästchen berechnet (siehe Abb 3.8). Wäre $x_0 = 2, y_0 = G$, so wäre das Kästchen und der erste Eintrag in stringvec 3H. Da der x-Wert über b nur mit dem Index i wächst, werden zuerste alle weiteren y-Werte über a berechnet und in stringvec geschrieben: 3H,3G,3F,3E. Nun wurde die Abbruchbedingung für Index ii wahr, da die Variable "fields" die Summe der Schiffsegmente mit einem Kästchen davor und dahinter ist. Somit ergibt sich für ein Uboot eine Vier. Nun wird i hochgezählt und die Berechnung beginnt erneut, nun ab $b = x_0 = 2$ und $a = y_0 + 1 = H$, in stringvec wird also geschrieben: 2H,2G,2F,2E. Die letzten vier Werte werden analog berechnet. Der Umkreisvektor mit den String-Koordinaten ist nun vollständig aufgebaut und kann zum Vergleich mit anderen Schiffen genutzt werden.

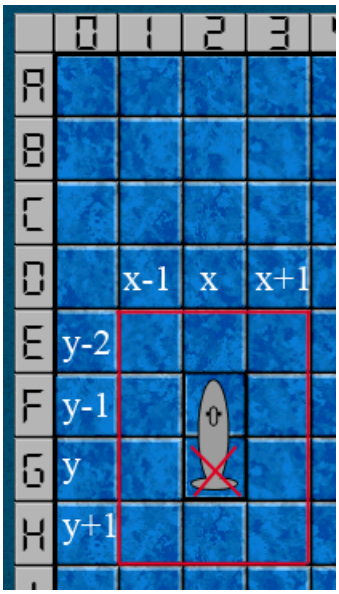

Abbildung 3.8: Umkreisberechnung

3.3 Spieler: player

Die Klasse “player“ ist die umfangreichste Klasse mit den umfangreichsten Methoden im Programm. In der Header-Datei “cplayer.h“ finden sich wichtige Definitionen zur Geometrie des Spielfeldes und die Dateipfade zu den graphischen Elementen und den Textdateien mit Koordinatenmustern. Weiter beinhaltet diese Klasse Methoden zur Eingabe von Angriffskoordinaten und Aktionen, Tests auf regelkonforme Platzierung der Schiffe untereinander und zur Anzeige der Schiffe und Treffer.
Die Speicherelemente dieser Klasse sind:

- **Spielernummer, Name**
- **sad algo**: SaD-Algorithmus
- **_isalive, _canmove**: Spielerflags
- **_ships**: Schiffsvektor mit allen Schiffstypen
- **_hits**: String-Vektor mit Koordinaten, die zu einem Treffer führten
- **_nogo**: String-Vektor mit den Koordinaten der letzten drei Treffer

Hinzu kommen die verschiedenen Methoden:

- **Set/Get-Methoden**
- **check_hit()**: testet ob Koordinaten zu Treffer führen, ruft SaD auf
- **check_coor()**: testet ob Koordinaten bereits zu einem Treffer führten
- **check_isalive()**: testet ob Spieler Aktion ausführen darf oder besiegt ist
- **positioncheck()**: testet auf regelkonforme Lage von Schiffen untereinander, auch nach Aktion
- **input()**: platziert Schiffe zu Spielbeginn, liest Koordinaten aus Datei
- **input_action()**: wählt ein Schiff und führt eine Aktion aus
- **get_input()**: liest Koordinatenpaar aus Datei, mit und ohne Aktion
- **verschiedene draw()**-Methoden zur graphischen Darstellung

Der **Konstruktor player**(char inr) benönigt eine Spielerangabe (P1 oder P2) und initialisiert damit alle Vektoren und baut die verschiedenen Schiffstypen im Schiffsvektor auf. Darüber hinaus wird durch die Spielerangabe der Offset für Spieler 2 gesetzt, sodass die graphische Darstellung entsprechend angepasst ist.
Die **input(int ioffset)**-Methode wird für beide Spieler vor dem Eintreten in die Spielrunden (while-Schleife in user.cpp) aufgerufen. Hiermit werden alle Schiffe eines Spielers gesetzt, indem aus einer Koordinatendatei (FILENAME_START) mit einem übergebenen Offset, der abgefragt werden muss, die Werte in x,y-Paaren ausgelesen werden. Dabei wird auf regelkonforme Platzierung auf der Karte und zwischen den Schiffen getestet. Die Koordinaten in der Koordinatendatei müssen also nicht bereits eine gültige Platzierung für alle Schiffe beinhalten, sie muss nur mindestens alle Koordinaten einmal enthalten. Ist ein Koordinatenpaar ungültig, so wird so lange das nächste eingelesen und getestet, bis eine gültige Position gefunden ist. Wird das Ende der Datei erreicht, so wird die Datei erneut durchlaufen. Das so gesetzte Schiff wird auch direkt eingezeichnet über den Aufruf der draw()-Methode:

Listing 3.12: Ausschnitt: input(int ioffset)

```
1 // Ausrichtung und Koordinaten des Schiffs werden gesetzt:
2 (ptr_ships + i)->ships::set_orientation(orientation);    // Ausrichtung
3 (ptr_ships + i)->ships::set_position(x, y);      // Koordinaten berechnen
4 type=(ptr_ships + i)->ships::get_type();
5 std::cout << *(ptr_names + ii) << "(" << i << ")" << std::endl
6 << "Koordinaten: " << x << y << orientation << std::endl << std::endl;
7 this->player::draw(x,y,type,orientation);        // Schiff zeichnen
8 updatescr();                                     // Anzeige aktualisieren
```

Die **input_action(int iround)**-Methode erlaubt es eine Aktion mit einem Schiff durchzuführen. Die Übergabe der momentanen Runde wird dabei als "Seed" für die zufällige Auswahl eines Schiffes benutzt für das dann alle möglichen Aktionen (Vorwärts-/Rückwärtsfahren oder Drehung im UZ oder GZ) getestet werden. Kann keine Aktion für dieses Schiff ausgeführt werden, so wird auf die Aktion verzichtet. Auch der Verzicht auf eine Aktion stellt eine regelkonforme Aktion dar und kann auch als taktisch sinnvoll angesehen werden, weil der Gegner auf bereits getroffene Positionen erneut schießen kann. Es führt allerdings auch je nach Startposition zu einer geringeren Bewegung auf dem Spielfeld. Dies kann in der player.h-Datei geändert werden, indem die Definition MUST_TAKE_ACTION auf "1" (default: 0) gesetzt wird, diese ist aber nur experimentell implementiert.
Die überladene **get_input(std::string ifilename, int ipos)-** bzw. **get_input(std::string ifilename, int ipos, int ival)**-Methode wird nur in ihrer ersten Form aufgerufen. Sie erlaubt es aus einer Koordinatendateien (FILENAME_MUSTER) Angriffskoordinatenpaare auszulesen. Die zweite Variante erlaubt es Koordinatenpaare mit Aktion einzulesen, sodass eine komplette Strategie in einer Datei gespei-

chert und abgerufen werden kann.

3.3.1 Testmethoden: Check

Die Check-Methoden sind essentielle Bausteine, die den menschlichen Spieler ersetzen und die komplette Automation des Spiels erlauben. Als Beispiel wird die **check_hit(std::string icoordinates)**-Methode behandelt. Sie startet und stoppt bei Bedarf den SaD-Algorithmus, schreibt Treffer in den Treffervektor des Schiffs, des Spielers und in den Nogo-Vektor und ruft die draw()-Methoden zur Anzeigenaktualisierung für Treffer auf.
Wird auf einen Treffer getestet, so müssen alle Einträge der Koordinatenvektoren der Schiffe eines Spielers durchlaufen und dabei die Koordinatenpaare jeweils mit dem Angriffskoordinatenpaar (coor) verglichen werden. Die Koordinatenpaare liegen als Strings vor und sind somit über ".compare()" prüfbar. Diese besitzt eine etwa lineare Zeitkomplexität. Hinzu kommen nun zwei For-Schleifen mit jeweils linearer Zeitkomplexität. Es wird bei der Berechnung auf die O-Notation zurückgegriffen:

$$\begin{aligned} T(n_1, n_2, n_3) &= O(n_1 * O_{type}(1) + n_1 * n_2 * O_{if}(1) * O_{compare}(n_3))) \\ &\overset{n=n_1=n_2=n_3}{=} O(n + n * n * n) \\ &= O(n + n^3) \\ &=> O(n^3) \end{aligned}$$

Grob abgeschätzt erhält man dadurch eine $O(n^3)$ kubische Zeitkomplexität. Doch diese wird durch die Anzahl der Elemente stark relativiert, denn im Spiel gilt maximal: $n_1 = 10, n_2 = 5, n_3 = 2$ und somit fällt die Laufzeit annehmbar aus. Wollte man allerdings das Spielfeld vergrößern, die Schiffanzahl (n_1) und Segmente der Schiffe (n_2) erhöhen und sogar im Mehrdimensionalen (n_3) spielen, so sollte man den Einsatz dieses Algorithmus überdenken.

Listing 3.13: Ausschnitt 1: check_hit()

```
1 for (i = 0; i < MAX; i++)          // MAX = 10 Schiffe im Schiffsvektor
2  {
3   type = (ptr_ships+i)->ships::get_type()-'0';  // Laenge der Schiffe
4   for (ii=0; ii<type; ii++)        // Koordinatenstring eines Schiffes
5   {
6    vergleich=coor.compare(*((ptr_ships+i)->ships::get_strposition()+ii));
7    if (!vergleich)
8    {...
9   }
10 }
```

Zeigt der Vergleich, das ein Treffer vorliegt, so wird als erstes das Trefferflag des SaD-Algorithmus gesetzt. Dadurch wird die nächste Koordinate des generierten Angriffsmusters beim nächsten Angriff genutzt. Erweist sich das Schiff als noch nicht versenkt, so werden die Trefferkoordinaten in den Treffervektor des Schiffes und in den Nogo-Vektor des Spielers geschrieben. Abschließend wird der Treffer eingezeichnet und der SaD-Algorithmus aktiv gehalten, indem die SaD-Flag gehalten wird.

Listing 3.14: Ausschnitt 2: check_hit()

```
1  if (!vergleich)          // str.compare liefert 0 bei Uebereinstimmung
2  {
3   std::cout << "Treffer!" << std::endl;
4   if (algo.sad::get_stopmuster())          // SaD aktiv?
5   {
6    algo.sad::set_hitflag(true);            // SaD-Trefferflag an
7   }
8   if ((ptr_ships + i)->ships::set_hit(x, y)) // Setze Treffer, Schiff zerstoert?
9   {
10   this->player::set_nogo(coor);           // Trefferkoordinaten setzen
11   this->player::draw_hits();              // Treffer zeichnen
12   wait(5);                                // Warte
13   this->player::check_isalive();          // Spieler besiegt?
14   algo.sad::set_stopmuster(true);         // SaD aktiv lassen
15   algo.sad::set_lasthit(coor);            // Trefferkoordinaten an SaD
16  }
17  else                                     // Schiff bereits zerstoert
18  {
19   algo.sad::set_hitflag(false);           // SaD-Trefferflag aus
20   return false;                           // Kein Treffer!
21  }
22  return true;                             // Treffer!
23  }
24 <...>
25 std::cout << "Wasser!" << std::endl;      // Kein Schiff getroffen
26 algo.sad::set_hitflag(false);             // SaD-Trefferflag aus
27 return false;                             // Kein Treffer!
```

Kam es allerdings nicht zu einem Treffer, so wird die SaD-Flag gelöscht und die nächsten Koordinaten für den Angriff werden wieder aus der Musterdatei entnommen.

3.3.2 Darstellungsmethoden: Draw

Die Draw-Methoden erlauben es sowohl den Hintergrund, die Spielernamen, Schiffe und Treffer zu zeichnen. Sie verwalten die graphische Darstellung des Spiels.

Alle Schiffe besitzen für jede Ausrichtung eine eigene .bmp-Datei, die das Schiff darstellt. Hinzu kommen noch "leere" bmp.-Dateien, die lediglich Wasserfelder zeigen. Die Definitionen zu den Dateipfaden finden sich in der cplayer.h-Datei. Die Methoden übernehmen automatisch den Offsetabgleich für den 2. Spieler.
Für eine spezifische Platzierung von einzelnen Schiffen wird **player::draw(char ix, char iy, char itype, char iori)** genutzt. Hierzu bedarf es des Ursprungs S(x,y), Typs und der Ausrichtung des Schiffs. Danach übernimmt die Funktion die Auswahl der .bmp-Datei, die Berechnung der Koordinaten inklusive Offset für den 2. Spieler und übergibt sie der globalen Funktion drawships(CSting filename, int x, int y). Die Berechnung beachtet dabei die GDE-eigene Herangehensweise, ab welchen Koordinaten eine .bmp-Dateien gezeichnet wird. Die drawships()-Funktion wurde der GDE hinzugefügt, damit eine graphische Darstellung über .bmp-Dateien möglich ist (siehe Seite 5).
Die **player::draw_hit(char ix, char iy)**-Methode lässt ebenfalls ganz gezielt Treffer zeichnen und greift dabei auf die text(x, y, 50, RED, "X")-Funktion der GDE zurück. Es wird einfach ein großes rotes "X" gezeichnet.
Die Methoden **player::draw_ships(), player::draw_hits()** und **player::draw_name()** durchlaufen den Schiffs-, Treffervektor und _name-String des Spielers, um die Informationen dann graphisch darzustellen.
Die **player::draw_blank(char ix, char iy, char itype, char iori)**-Methode zeichnet Wasserfelder. Dazu berechnet sie den Punkt ab dem ein Wasserfeld gesetzt werden muss, damit ein Schiff von der Karte gelöscht wird. Dies ist orientierungsabhängig, erfolgt aber wie bei der draw()-Methode.

3.4 Search and Destroy-Algorithmus: sad

Die folgenden Informationen werden in dieser Klasse verwaltet:

- **_s,_z,_nrhits**: Spalte und Zeile in Matrix, Anzahl an Treffern durch SaD
- **flag_stopmuster**: stoppt das Auslesen aus der Musterdatei
- **flag_aav**: zeigt an, ob Angriffsmatrix schon erstellt wurde
- **flag_hit**: zeigt an, ob letzter Angriff durch SaD-Koordinaten ein Treffer war
- **_av**: Angriffsmatrix, 1dim String-Matrix, mit Angriffskoordinaten
- **lasthit**: Koordinaten des letzten Treffers

- **orientation**: Ausrichtung des Schiffs (UNKNOWN, HORIZONTAL, VERTICAL)

Der SaD-Algorithmus versucht die Ausrichtung eines Schiffes herauszufinden, um dann gezielt die einzelnen Schiffsegmente anzugreifen. Hierzu wird eine Angriffsmatrix erstellt, die auf Grundlage des ersten Treffers nach einem Fehlschuss erstellt wird. Danach wird über die Treffer-Flag (hitflag) dem SaD-Algorithmus rückgemeldet, ob die gerade genutzte SaD-Angriffskoordinate zu einem Treffer führte. Je nach Status des SaD-Algorithmus wird dann verschieden verfahren. Diese Abfolge ist chronologisch:

1. Ausrichtung unbekannt: erste Zeile der Angriffsmatrix weiter durchlaufen, bis Fehlschuss oder “00“ auftritt, dann in nächste Zeile wechseln
2. Ausrichtung wird erkannt: auf entsprechende Zeile der Angriffsmatrix wechseln und durchlaufen bis Fehlschuss oder “00“ auftritt
3. Ausrichtung bekannt, aber Fehlschuss oder “00“: Angriffsmatrix durchlaufen, nun aber in die andere Richtung ab dem ersten Treffer
4. Ausrichtung bekannt, erneuter Fehlschuss oder “00“: Schiff wurde versenkt

Mit zwei Treffern lässt sich die Ausrichtung des Schiffs sicher feststellen, allerdings nicht welcher Schiffstyp getroffen wurde. Dies kann erst sicher festgestellt werden, wenn zwei Fehlschüsse gefallen sind, die dann aber auch anzeigen, dass das Schiff versenkt wurde. Eine Bewegung des Schiffs ist ausgeschlossen, da ein Treffer zu einem Nogo-Eintrag im Nogo-Vektor führt.

Der Index zum Zugriff der $(4x4)$Angriffsmatrix ergibt sich durch die Formel $i = 4 * z + s$, da sie als eindimensionaler Vektor verwirklicht ist. Wird bei der Erstellung der Kartenrand überlaufen, so wird diese Koordinate “00“ gesetzt, dies wird auch als Abbruchbedingung für eine Richtung genutzt. Für das Beispiel in Abb.3.9 ergibt sich die Angriffsmatrix:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 2C & 2B & 2A & 00 \\ 3D & 4D & 5D & 6D \\ 2E & 2F & 2G & 2H \\ 1D & 0D & 00 & 00 \end{pmatrix} \end{array}$$

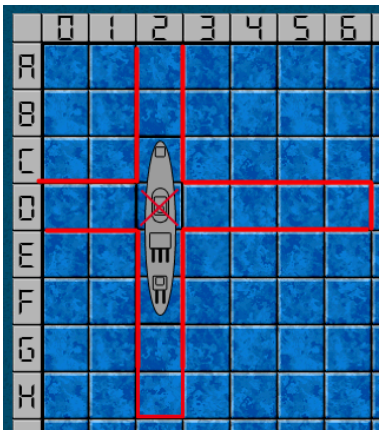

Abbildung 3.9: Angriffskreuz

3.5 Mustererstellung: pattern

Es stehen folgende Methoden zur Erstellung von Dateien zur Verfügung:

- **pattern::create_rndcf()**: erstellt pseudo-zufällige Koordinatensequenz
- **pattern::create_cf()**: erstellt geordnete Koordinatensequenz
- **pattern::create_muster()**: erstellt Mustersequenz mit Abständen
- **pattern::create_rnda()**: erstellt pseudo-zufällige Aktionsmuster

Die Mustererstellung erstellt jeweils .txt-Dateien, die dann zur Platzierung der Schiffe und zum Auslesen von Angriffskoordinaten genutzt werden. Zu beachten ist, dass die Eingabe zwischen Groß- und Kleinschreibung unterscheidet (case sensitive). Das Koordinatenintervalle läuft für x von 0-9 und für y von A-J und es wird wie gewohnt eine Koordinate geordnet mit "XY" angegeben. Siehe hierfür auch Seite 5.
Die Startsequenz ist in der Datei "start.txt" definiert. Der default-Wert sind alle Koordinatenpaare:

0A1A2A3A4A5A6A7A8A9A0B1B2B3B4B5B6B7B8B9B
0C1C2C3C4C5C6C7C8C9C0D1D2D3D4D5D6D7D8D9D
0E1E2E3E4E5E6E7E8E9E0F1F2F3F4F5F6F7F8F9F
0G1G2G3G4G5G6G7G8G9G0H1H2H3H4H5H6H7H8H9H
0I1I2I3I4I5I6I7I8I9I0J1J2J3J4J5J6J7J8J9J

Sie enthält eine Startsequenz für die Schiffe. Die Schiffe werden beginnend mit dem größten Schiff (Schlachtschiff, 5) hin zum kleinsten (Uboot, 2) gesetzt.
Die Treffersequenz aus der Musterdatei der Spieler sind unterschiedlich. Für Spieler 1 ergibt sich die Datei "hits1.txt", für Spieler 2 entsprechend "hits2.txt". Bei Bedarf kann die Definition in der cplayer.h-Datei und user.cpp, wo die Dateien an die Funktionen übergeben werden, geändert werden. Als default wird die "muster.txt" bei Spieler 1 benutzt. Diese enthält ein Muster, dass Lücken aufweist. Diese Abstände können beliebig in der **pattern::create_muster()**-Methode gewaehlt werden. Wichtig ist nur, dass darauf geachtet wird, dass innerhalb von 100 Einträgen jede Koordinate einmal vertreten ist. Die Sequenz wird immer wieder von Anfang an durchlaufen, da sich Schiffe bewegen können. Sie ist dazu gedacht mit dem SaD-Algorithmus möglichst schnell einen zufälligen Treffer zu erzielen, damit der SaD-Algorithmus die Ausgabe der Angriffskoordinaten übernimmt.
Die action.txt-Datei ist experimentell implementiert und muss erst im Quellcode in cplayer.cpp in der Methode **player::input_action()** auskommentiert werde.

Aussicht

Ein Programm lässt sich fortwährend optimieren. Funktionen können vereinfacht und sogar hinzugefügt, die Speicherorganisation übersichtlicher gestaltet werden. Programme bieten immer Potential zum Wachsen. Aus diesem Grund muss eine Abwägung getroffen werden, ab wann ein Programm im zufriedenstellenden Maße die Anforderungen erfüllt. Ansonsten droht dem Programmierer sich in den unendlichen Möglichkeiten der Gedanken und Ideen zu verlieren.
Eine Optimierung der **check_hit()**-Methode der player-Klasse wäre beispielsweise, wenn statt des gesamten Schiffsvektors des Spielers und damit alle Koordinatenvektoren der Schiffe nur die Schnittmenge zweier x-Korridor und y-Korridor, je mit einer Breite von fünf Segmenten, abgefragt werden würde. Mit einer solchen Variante könnte auch nur ein Korridor bei der Bewegung von Schiffen in der **positioncheck()**-Methode geprüft werden, dann müssten nicht alle Koordinatenvektoren der Schiffe durchlaufen werden. Ob sich die Zeitkomplexität durch die zusätzliche Initialisierung verringert, müsste dann getestet werden.
Die Eingabe der Koordinaten mit Ausrichtung des Schiffs wurde mit Blick auf die Umsetzung auf einem Tablet oder Smartphone verwirklicht, sodass ein Nutzer mit einer Geste das Schiff setzen kann und dann die Ausrichtung des Schiffs über eine Berührung in der Nähe der Ränder des Bildschirms bestimmt. Dies ist intuitiv und einfach handhabbar. Die Umsetzung des Spiels auf einem Tablet oder Smartphone wäre ein möglicher nächster Schritt.

Anhang

Das Visual Studio Projekt mit dem kompletten Quelltext befindet sich auf der beiliegenden DVD.