Design aspekte

UX/UI design:

- Intuitives Design, durch frühen Prototyp mit Adobe XD, um alle Nutzer Anforderungen zu testen und ein gutes Layout und Navigation zu finden.
- Möglichst viel nutzer Feedback durch Sounds oder Animation (z.B. invalid attack).
- Signalfarbe (grün) für alle interaktiven Elemente.
- Scaling behält Komponenten nahe zusammen (Größe ändert sich aber mit), da das zu einer besseren User-Experience führt, als Verzerrung des Layouts.

Code/Software design:

MVVM-Pattern für UI

- Teilt die Aufgaben auf und macht einzelne Teile unabhängiger voneinander. Das führt zu weniger Fehler und erlaubt die Wiederverwendung von Elementen.
    - View: stellt einfach nur den Zustand (den es vom ViewModel bekommt) dar.
    - ViewModel: wandelt die backend Informationen in von der View verwertbare Informationen um.
    - Model: Haupt Logik, z.B. Netzwerk, Verwalten der Spielfelder ...
- View > ViewModel > Model hat "direkten" Zugriff auf die nächste "Layer". Andersherum erfolgt die Kommunikation indirekt (mit Events und Properties).

- UI-Interaktion und UI-Updates in der View.
- UI layout in fxml files.
- UI styling in css.

- Dies erlaubt sehr schnelle Änderungen und Ergänzungen, ohne andere Teile zu beeinflussen.

Navigator

- Erlaubt einfache Navigations-Aufrufe zwischen verschiedenen Ansichten.
- Behandelte verschiedene Views unterschiedlich (z.B. injected automatisch eine verdunkelnde Pane in den Hintergrund von Dialogen), oder erlaubt nur die Darstellung eines Dialogs über normalen Views, der Rest wird im navStack gespeichert.
- Erlaubt das Übergeben von Daten zwischen Views.
- Erlaubt Backtracking zu alten Views mittels NavigationStack. Es behält nur aktive Views im Speicher/Stack.
- Erlaubt Echtzeit-Updaten aller geladener Views (z.B. für Sprache) durch manuellen SceneGraph-traversal aller Views im NavStack.
- Theoretisch wäre es möglich, Unternavigation innerhalb von Views zu machen.

Eigener EventManager

- Objekte können sich als Listener registrieren und alle Methoden mit @HandlesEvent werden mittels Java Reflection in einer Map gespeichert.
- Listener Methoden werden beim Feuern von relevanten Events aus Queue abgearbeitet.
- Erlaubt einfache Kommunikation zwischen Client und Bot, bzw Netzwerk-Teil (Client bleibt Gegnertyp unabhängig).
- Erlaubt die Einhaltung von MVVM-Pattern.

HeatMap für schweren Bot

- Repräsentiert eine simplifizierte Kombination aller möglichen übrigen Schiffe und Platzierungen.
- Im Vergleich mit mehreren anderen Algorithmen, die besten Resultate für eine Echtzeit Anwendung.
- Optimiert und 4-5x schneller als "naive" implementation:
    - Nutzt Bitmask, um blockierte Positionen darzustellen (int[] statt int[][], spart Daten und erlaubt Vergleiche ganzer Zeilen auf einmal).
    - Optimierte Iterations Reihenfolge erlaubt mit einfachem Bitshifts die Wiederverwendung der Masken.
    - Spielfeld-Bitmaske und Heatmap sind in alle Richtungen um 1 größer als tatsächliches Spielfeld, um isInBounds() zu vermeiden.
    - Update Methode berechnet meist nur tatsächlich veränderte Positionen (und deren Einfluss bereich) neu, statt alles immer neu zu berechnen

Netzwerk

- Netzwerk Transceiver listened auf lokale CommEvents, parsed sie in den Netzwerk-String und sendet sie. Anders herum empfängt es Netzwerk-Strings, parsed und feuert sie als lokale Events.
- Nutzt JavaFX Task-System für das Öffnen der Verbindung und die Kommunikation. So bleibt das UI reaktionsfähig und der SceneGraph und die Logik werden nur vom FX Thread aus verändert.

Gameplay Logik

- Datenstrukturen einfach gehalten. Sie verwalten ihre eigenen Daten und erlauben nach außen einfache grobere Veränderungen, z.B. getShipFromPos(), sinkShip(), …
- Haupt Logik in Client/Bot oder abstrakter Player Klasse für das aktive Spiel und GameSetupManager für Dinge wie: Neues Spiel, Spiel laden, Spiel speicher...

Dinge die man hätte besser machen können

- Während der Erstellung von Spielen ist ein Teil des Ablaufes im Navigations-State integriert (aus verschiedenen Gründen). Es wäre besser, wenn alles zentral geregelt wäre und anderweitig sichergestellt würde, dass das UI bereit ist.
- Auch wenn wir einige UI-Framework mäßige Aspekte selbst implementiert haben, muss man teils immer noch auf kompliziertes State-Management achten. (Für die Anwendung wäre das aber Overkill)
- css in mehrere Dateien aufteilen, da nicht überall alle Komponenten gebraucht werden.