FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

SCHAKEN

Ontwikkeling van een schaakbot met behulp van Negamax
met Alpha-Beta Pruning en multithreading met JavaScript Web Workers

TEAM 02

Brecht Stevens
Jelle De Moerloose
Stijn De Schrijver
Wiebe Vandendriessche

UNIVERSITEIT
GENT

2022-2023 Industrieel ingenieur informatica

Inhoud

Figuren 5
Inleiding 6
Structuur van het project 7
Het schaakmodel 8
  Coördinaten 8
  De schaakstukken 8
    Berekenen pseudolegale zetten 9
  Het schaakbord 10
    Board klasse 11
    LegalChecker 12
    FenConverter 15
De algoritmes 16
  Het minimaxalgoritme 16
    Basiswerking van minimax 16
    Alpha-beta pruning 17
    Implementatie 18
  Het negamaxalgoritme 19
    Basiswerking 19
    Alpha-beta pruning 19
  De evaluaties 19
    Schaakstukwaarden optellen 19
    Schaakmatevaluatie 19
    Schaakstuktafels 19
De botklasse 21
  Structuur 21
  Implementatie algoritme 21
De evaluatieklasse 23
  Structuur 23
  Werking 23
ViewController 25
  AGamestate 26
  GamestatePlay 26
  Movecacher 27
  Draw klasse 28
  Verbeteringen Viewcontroler 28
De View 29

Navigatiebalk 29
Instellingen 30
Kleur 30
Geluid 30
Belangerijke stukken code uitleggen: 31
Puzzels *mockAPI* 31
Multithreading 32
Sustainable webdesign 34
Donkere modus 34
Modulaire code 35
Efficiënte code 35
Angular of niet? 35
Conclusie 36
Referentielijst 37

Figuren

Figuur 1: Homescreen 6
Figuur 2: Structuur project 7
Figuur 3: Het schaakmodel 8
Figuur 4: Coordinate klasse 8
Figuur 5: Abstracte klasse Apiece 9
Figuur 6: Schaakbord klassendiagram 10
Figuur 7: VB van een speudolegale, maar illegale zet 11
Figuur 8: Board klasse 11
Figuur 9: LegalChecker klasse 12
Figuur 10: Voorwaarden rokeren 13
Figuur 11: rechts rokeren met wit 13
Figuur 12: pseudocode minimax (Sebastian Lague, 2021, 03:58) 16
Figuur 13: voorbeeld minimax zoekboom 16
Figuur 14: pseudocode minimax met alpha-beta pruning (Sebastian Lague, 2021, 08:52) 17
Figuur 15: voorbeeld alpha-beta pruning zoekboom 17
Figuur 16: hoofdgedeelte minimaxalgoritme 18
Figuur 17: whileloop 1 minimaxalgoritme 18
Figuur 18: whileloop 2 minimaxalgoritme 18
Figuur 19: pseudocode negamax (Negamax - Chessprogramming wiki, z.d.) 19
Figuur 20: Botklasse 21
Figuur 21: klassendiagram Evaluation 23
Figuur 22: viewcontroller klasse diagram 25
Figuur 23: voorbeeld getekende schaakpositie 28
Figuur 24: navigatiebalk 29
Figuur 25: colorpicker 30
Figuur 26: donkere modus 34

Inleiding

Dit verslag beschrijft de uitwerking van een schaakwebsite waar gebruikers de mogelijkheid hebben om te spelen tegen andere spelers of tegen bots. Men kan ook schaakpuzzels oplossen. Het project werd ontwikkeld door een groep van vier studenten als onderdeel van hun Softwareproject, waarbij zij verschillende vaardigheden en kennis uit eerdere opleidingsonderdelen hebben gebruikt. Het doel van de website is om mensen met elkaar te verbinden via hun gemeenschappelijke interesse in schaken. De site is ontwikkeld met aandacht voor duurzaamheid.

Er zijn verschillende lay-out features geïmplementeerd, zoals de mogelijkheid om instellingen aan te passen en een *darkmode* om de gebruikerservaring te verbeteren. Kleuren en geluid kunnen aangepast worden. Schaakpuzzels worden opgehaald uit een database en kunnen op verschillende niveaus worden opgelost. De bots zijn beschikbaar op verschillende moeilijkheidsgraden.

Het ontwikkelingsproces van de website was gericht op het bereiken van een duurzame en efficiënte ontwikkeling. Het projectteam heeft verschillende maatregelen genomen om de efficiëntie en duurzaamheid van het ontwikkelingsproces te verbeteren, zoals het gebruik van de Agile ontwikkelingsmethode en het verminderen van onnodige complexiteit in de code.

Dit rapport beschrijft gedetailleerd de verschillende aspecten van de schaakwebsite, inclusief de gebruikte technologieën, de architectuur van de website, de functionaliteiten en de implementatie van duurzaamheidspraktijken in het ontwikkelingsproces.

Figuur 1: Homescreen

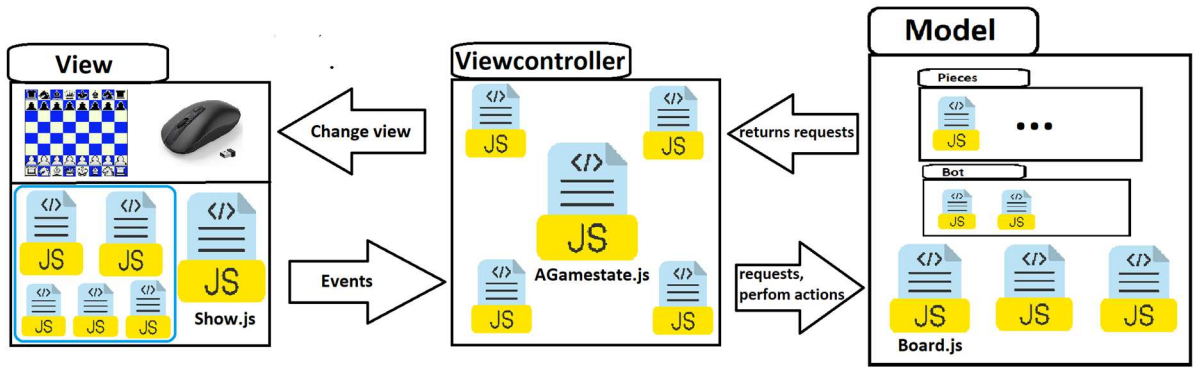
Structuur van het project

Het is belangrijk om alle bestanden te structureren om zo een duidelijk beeld te kunnen krijgen van hoe het project juist werkt. De structuur die hier wordt gebruikt kan vergeleken worden met een variant van het Model-View-Controller patroon (zie Figuur 2).

Hierbij is de view gedefinieerd als de elementen die de gebruiker van de website ziet en gebruikt, maar ook de javascript files die alle eventluisteraars definiëren. De files aangeduid met een blauw kader, zijn files die de directe functionaliteit van de website weergeven en geen model gebruiken (bv navigatiebar, animaties, thema's). De file "Show.js" gebruikt wel het model, deze koppelt events met functies uit de Viewcontroller en speelt dus een belangrijke rol in het delegeren van acties in het project.

De viewcontroller houdt alles in omtrent het spelgebeuren van het schaken, zo kan je tegen een vriend schaken, tegen een bot schaken of puzzels oplossen. Deze zal de view ook aanpassen door het canvas waarop gespeeld wordt te hertekenen of door pop-ups te laten verschijnen. De viewcontroller staat rechtstreeks in contact met het model, waaraan het dingen kan vragen of aanpassen.

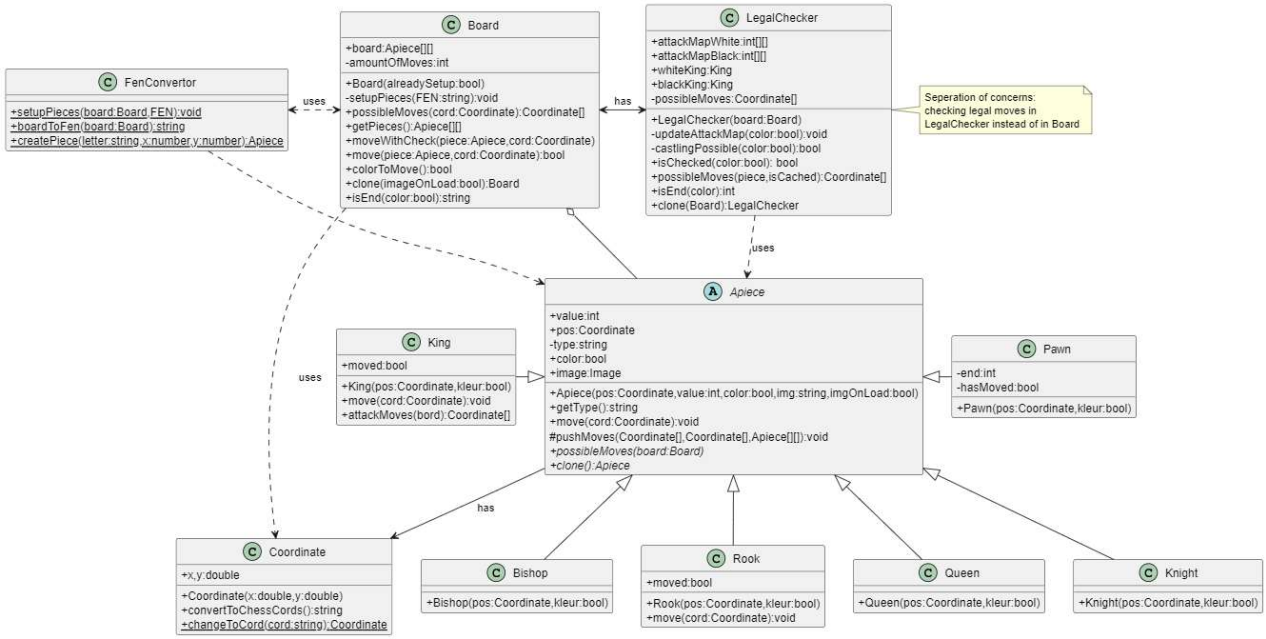
Het model houdt het schaakbord in met al zijn stukken en al zijn functionaliteit, ook heeft het een bot die kan gebruikt worden. Dit model werkt als een module die door andere viewcontrollers zou kunnen gebruikt worden. Zo kan bijvoorbeeld een andere ontwikkelaar dit model gebruiken om te schaken op een terminal of een mobile app in plaats van een website. Een modulair ontwerp zorgt zo voor een duurzaam ontwerp van de software.



Figuur 2: Structuur project

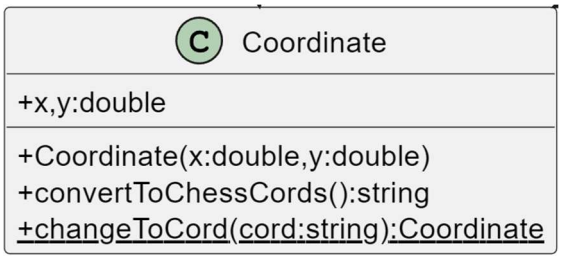
Het schaakmodel

Om een goed werkende schaakwebsite te maken, is het nodig om een efficiënt en duidelijk model te gebruiken waarin het schaakspel kan worden gespeeld. Dit model kan opgedeeld worden in 3 grote delen: de schaakstukken, het bord en de bot (besproken in Algoritme). Deze klassen maken beide ook vaak gebruik van de “Coordinate” klasse.



Figuur 3: Het schaakmodel

Coördinaten



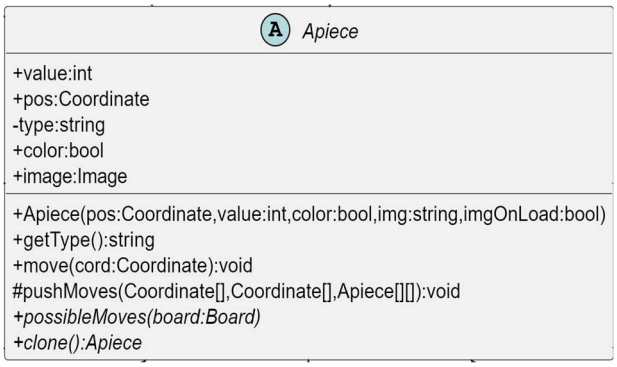
Figuur 4: Coordinate klasse

De “Coordinate” klasse wordt gebruikt om posities op het acht bij acht schaakbord te omschrijven. Daarnaast is dit ook de klasse waar de coördinaten omgezet kunnen worden naar effectieve schaak coördinaten met letters en cijfers. Dit wordt gedaan door de functie “convertToChessCords”. De klasse doet dit via een simpele array die hij ook bijhoudt in de klasse. Deze functie wordt gebruikt door de “Movecacher” (zie Movecacher).

De omgekeerde functie “ChangeToCord” gebruikt de array dan in de omgekeerde richting. Die zoekt dan de plaats van een bepaalde letter in de array. De “ChangeToCord” wordt gebruikt om de zetten van een puzzel te vertalen naar zetten met coördinaten die het model kent.

De schaakstukken

Omdat alle schaakstukken gelijkaardige karakteristieken vertonen, wordt er gebruikt gemaakt van een abstracte klasse die deze attributen en methoden zal definiëren. Zo wordt er geprogrammeerd naar een abstractie (OO-ontwerpprincipe) en wordt *duplicated* code vermeden.



Figuur 5: Abstracte klasse Apiece

Elk schaakstuk heeft een waarde, een positie, een kleur, en een afbeelding. Ook kan er worden beslist of de afbeelding wordt aangemaakt of niet, want dit is niet altijd nodig. In het algoritme (zie De algoritmes) wordt er gebruik gemaakt van gekloonde schaakborden, daar is er dus veel performantieverlies als er telkens een afbeelding aangemaakt moet worden per nieuw schaakstuk. De abstracte klasse heeft ook een type attribuut met een corresponderende *Getter*. Deze wordt afgeleid van de "img" parameter in de *constructor* en is gemaakt om de trage "instanceof" functie te vermijden. Dit draagt dan weer bij aan de performantie van de website.

De klasse legt aan alle afgeleide klassen (de specifieke schaakstukken) op om de methoden "possibleMoves" en "clone" te overschrijven, zodat elk stuk zijn pseudolegale zetten kan teruggeven en gekloond kan worden. Pseudolegale zetten zijn de zetten die een schaakstuk kan maken in zijn situatie zonder rekening te houden met aanvallen op de koning. Zo is een pion die de koning beschermt een vakje vooruitzetten wel een pseudolegale zet, maar geen legale zet (zie Figuur 7). Ook heeft de klasse de methode "move" die de coördinaat van het schaakstuk aanpast (deze kan worden overschreven) en een hulpfunctie "pushMoves" die later zal worden uitgelegd.
De klassen "Pawn","King" en "Rook" vertonen ander gedrag nadat ze al een keer hebben bewogen, om die reden krijgen zij een attribuut "moved", deze zal worden gezet op true in de overschreven "move"-functie.

Berekenen pseudolegale zetten

Voor het berekenen van de pseudolegale zetten, is er kennis vereist van de manier van bewegen van elk schaakstuk. Deze kan teruggevonden worden op de website bij Info>Spelregels, bij een klik op een schaakstuk zal de informatie verschijnen.

Om deze functie zo efficiënt mogelijk te maken, werden weer de gelijkenissen gebruikt tussen bepaalde stukken. Als je de koningin, de toren en de loper bekijkt, kan er opgemerkt worden dat deze telkens in een paar richtingen zo ver kunnen gaan in het bord als de speler maar wenst zolang er geen ander schaakstuk in de weg staat. Enkel als dit blokkerend stuk een ander kleur heeft als het bewegende stuk, dan kan het bewegende stuk het blokkerende stuk slaan. Voor deze gemeenschappelijke functionaliteit gebruiken we de protected functie "pushMoves" in "Apiece", die de mogelijke zetten teruggeeft aan de hand van de richtingen die je de functie meegeeft (horizontaal, verticaal, diagonaal). Dit wordt getoond met de loper:

```
possibleMoves(bord) { //loper
    let veld=bord.getPieces(); // 2D-array met alle pieces/nullen
    let possiblemoves=[]; // container voor alle speelbare moves
    // alle beweegingsrichitngen ( bewegen als een kruis X)
    let moves=[new Coordinate(1,-1), new Coordinate(1,1),
        new Coordinate(-1,1), new Coordinate(-1,-1)];
    this.pushMoves(possiblemoves,moves,veld); // functie van Apiece
    return possiblemoves;
}
```

De "pushMoves" methode gaat als volgt:

```
pushMoves(possiblemoves,moves,veld){
    for (let i = 0; i < moves.length; i++) {
        let x = this.pos.x;
        let y = this.pos.y;
```

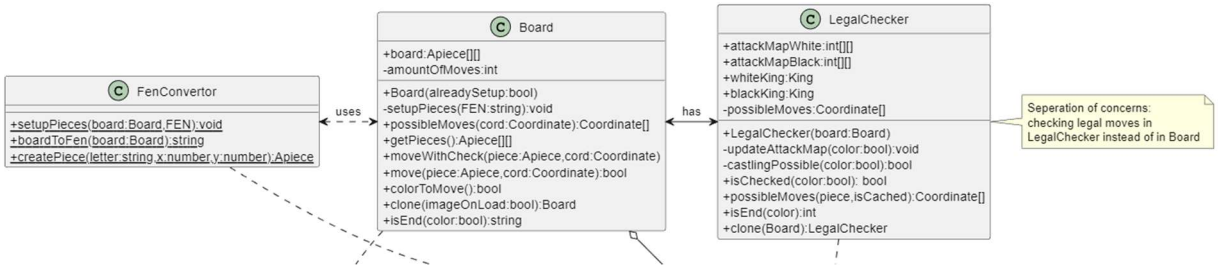
```
        let blocked = false;
        while (!blocked) { //zolang er een vrije weg is
            x += moves[i].x;
            y += moves[i].y;
            //blijft het binnen het bord?
            if (x === -1 || y === -1 || y === 8 || x == 8) {
                blocked = true;
            } else {
                let move = new Coordinate(x, y)
                if (veld[y][x] === 0) { //is het vak leeg
                    possiblemoves.push(move);
                } else {
                    let apiece = veld[y][x];
                    // staat er een vijand
                    if (apiece.kleur !== this.kleur) {
                        possiblemoves.push(move);
                    }
                    blocked = true;
                }
            }
        }
    }
}
```

Bij de pion, koning en paard werden specifiekere technieken gebruikt die zorgden voor juiste mogelijke zetten. Deze gebruiken andere methoden omdat ze iets anders bewegen (zie code), de koning kan ook nog rokeren maar dit wordt later uitgelegd.
Elk schaakstuk kan ook gekloond worden zodat dit later kan gebruikt worden om het hele spelbord te klonen, hier is het niet altijd gewenst om een afbeelding aan te maken dus dit wordt meegegeven a.d.h.v. een parameter.

```
clone(imageOnLoad) {//vb koning
    let king=new King(new Coordinate(this.pos.x,this.pos.y),
                                        this.kleur,imageOnLoad);
    king.moved=this.moved; //extra parameter
    return king;
}
```

Het schaakbord

Om te vermijden dat de klasse “Board” te veel verantwoordelijkheden kreeg, werd deze opgesplitst in drie delen: het schaakbord “Board”, controle-element “Legalchecker” en een converter “FENConverter” (OO-ontwerpprincipe: één klasse, één verantwoordelijkheid).



Figuur 6: Schaakbord klassendiagram

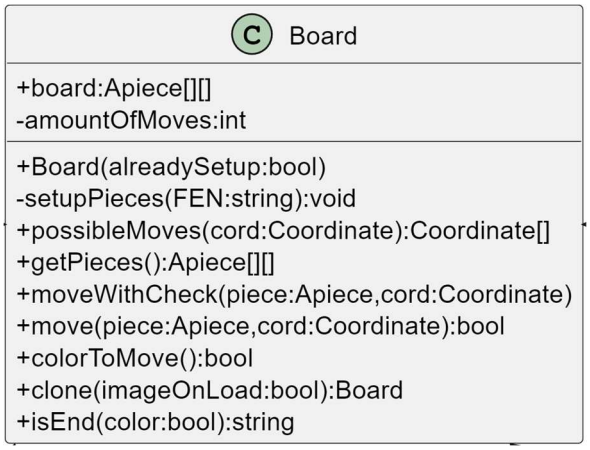
Het schaakbord behandelt alle verzoeken die gaan omtrent het bewegen van schaakstukken, zo kan de gebruiker van het model vragen welk kleur mag bewegen, welke zetten een stuk mag uitvoeren en een schaakstuk effectief verzetten (op een legale of niet legale manier). Ook kan de gebruiker de status van het spel opvragen met de “isEnd” methode.

Het controle element “Legalchecker” bekijkt of een zet volledig legaal kan gespeeld worden zonder de koning schaak te zetten. Het bord kan vragen aan dit element om de legale moves te berekenen voor een schaakstuk a.d.h.v. het schaakstuk zijn pseudolegale zetten en aanvalsmappen (zie LegalChecker). Het bord haalt de status van het spel ook van deze klasse, want enkel deze weet wanneer er een schaakmat of een patstelling (stalemate) voorkomt. Ook kan deze klasse controleren of een koning kan rokeren (zie verder).

Figuur 7: VB van een speudolegale, maar illegale zet

De “FENConverter” zet een stringrepresentatie van een schaakbord om in een instantie van de klasse “Board” en maakt het ook mogelijk om een “Board” object weer om te zetten naar een string. FEN of Forsyth-Edwards Notation is een standaard voor het beschrijven van spelposities bij het schaken met behulp van ASCII-karakters. Deze klasse is nodig omdat de puzzels die we ophalen uit de database in dit formaat worden teruggegeven. Ook wordt dit gebruikt om een compleet nieuw bord te initialiseren in de “Board” klasse in de “setupPieces” methode.

Board klasse



Figuur 8: Board klasse

Deze klasse houdt een 2D-array bij van schaakstukken van acht bij acht, dit is het veld waarop in het model wordt gespeeld. De array is op plekken waar geen schaakstuk staat gevuld met een nul. Ook wordt het aantal moves bijgehouden zodat er kan worden bepaald aan welke kleur de beurt is met de “colorToMove” methode. De klasse heeft een referentie naar een LegalChecker object (zie Figuur 6), waarvan hij gebruikt maakt bij 2 methodes: “possibleMoves” en “moveWithCheck”. Ook kan het bord gekloond worden, dit zal gebruikt worden in onder andere de “Legalchecker” en het algoritme.

Omdat de speler eerst moet klikken op een schaakstuk en dan de mogelijke legale zetten te zien krijgt en daarna één van die zetten speelt is het goed om deze te bij te houden om meervoudig rekenwerk te vermijden, dit wordt gedaan in de LegalChecker klasse. De volgende stukken code tonen de “possibleMoves” en “moveWithCheck” functie in de Board klasse:

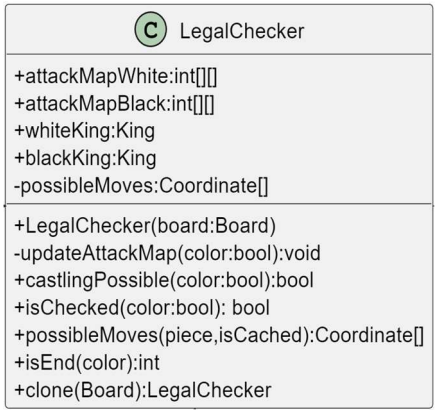
```
possibleMoves(cord) {
    let piece = this.board[cord.y][cord.x];
    //eerste klik op de pion bij spelen ->cachen
    return this.legalchecker.possibleMoves(piece,false);//bool is voor cachen
}
moveWithCheck(piece,cord){
```

```
    //tweede klik -> al gecached
    let realmoves=this.legalchecker.possibleMoves(piece,true);
    //als er een legale move wordt aangeklikt
    if(realmoves.some(move=>move.x===cord.x && move.y===cord.y)){
        //verzet het schaakstuk in 2D-array en wijzigt zijn coord (eventueel castlen)
        this.move(piece,cord);
        this.amountOfMoves++;
        return true;
    }else {
        return false;
    }
}
```

De gebruiker van het model krijgt “false” terug indien hij een zet wou doen die illegaal is, zo kan hij controleren of de zet doorgegaan is of niet.

LegalChecker

Zoals eerder vermeldt voert de klasse LegalChecker de controle uit van de correctheid van elke zet. Dit doet de klasse a.d.h.v. aanvalsmappen, dit zijn twee 2D-arrays van getallen die elk vakje zijn aanvallers telt. De mappen kunnen worden aangepast telkens de private methode “updateAttackMap” wordt aangeroepen. Deze functie vermeerderd de vakjes in de aanvalsmappen met één voor alle mogelijke aanvallende zetten van elk schaakstuk van die kleur. Voor de pion en de koning zijn niet alle mogelijke zetten aanvallend (voor de andere stukken wel), daarom bezitten ze de private functie “attackMoves”. Deze functie berekent alle mogelijke moves van de pion behalve het vooruit verplaatsen en van een koning buiten het rokeren, want hiermee kan je nooit een ander stuk slaan. Deze functie wordt niet opgeroepen in “possibleMoves” van de pion en zorgt voor duplicated code, dit komt doordat een extra functieoproep zorgt voor een groot performatieverlies in het algoritme (zie De algoritmes).



Figuur 9: LegalChecker klasse

De klasse bevat ook een referentie naar de koningen van het schaakbord, dit zorgt ervoor dat er niet telkens moet gezocht worden op welk vakje de koning staat door het speelveld van de klasse Board (2D-array) te overlopen. De belangrijkste functie van de “LegalChecker”-klasse is “possibleMoves”, deze maakt voor alle mogelijke zetten van een schaakstuk en gekloond board waarin hij het overeenkomstige gekloonde stuk verplaatst over al zijn mogelijke pseudolegale zetten. Na deze zet op een vals bord, kijkt het a.d.h.v. aanvalsmappen of de koning een aanvaller heeft, als dat niet zo is, is de zet legaal. Omdat deze 2 maal worden opgevraagd door de speler (zie Board klasse), is er een mogelijk om deze op te slaan in een attribuut van het “LegalChecker”-object.

```
possibleMoves(piece,isCached){
    if(isCached){ //indien hier net voor al eens gevraagd
        return this.possiblemoves;
    }else{
        let pseudolegalmoves=piece.possibleMoves(this.Board); //pseudolegale moves
        let realmoves=[];
        for (let coord of pseudolegalmoves){
            let cloneBoard=this.Board.clone(false);// klonen bord
            let virtpiece= cloneBoard.getPieces()[piece.pos.y][piece.pos.x];
            cloneBoard.move(virtpiece,coord,pseudolegalmoves); //verzet gekloond stuk
            if (!cloneBoard.legalchecker.isChecked(piece.kleur)){ // koning veilig
                realmoves.push(coord); //volledig legale zet
            }
```

```
        }
        this.possiblemoves=realmoves; //opslaan indien opnieuw vragen
        return realmoves;
    }
}
```

De “isChecked” methode vernieuwt de juiste *attackmap* en kijkt of de koning veilig staat:

```
isChecked(color) {

    let attackmap=this.updateAttackMap(!color); //attackers van andere kleur bekijken
    let king= color? this.whiteking: this.blackking;
    if (attackmap[king.pos.y][king.pos.x]!==0){
        return true;
    }else{

        return false;
    }
}
```

Ook kan de “LegalChecker” zien in welke fase van het spel de spelers zich begeven, dit kan hij doen door op alle stukken van het bord van een kleur de functie “possibleMoves(piece, false)” op te vragen. Zolang er zeker één stuk is die kan bewegen, is het spel nog niet gedaan. Indien er geen enkel stuk kan bewegen, wordt met de “IsChecked” functie gekeken of de koning schaak staat. Als de koning dan schaak staat dan is het schaakmat, anders is het gelijkspel (patstelling of stalemate).

Als laatste speelt de “LegalChecker” nog een cruciale rol in het rokeren (castling). Dit is een speciale zet waarbij de koning zich veiligstelt achter de toren in een hoek. Het kan enkel gebeuren als alle voorwaarde voldaan zijn. Ten eerste mogen de koning en de toren langs de kant van het rokeren, nog niet verplaats zijn geweest gedurende het spel. Ten tweede mogen er geen stukken tussen de koning en de desbetreffende toren staan. Als laatste (én moeilijkste) mag de koning en de vakjes tussen de koning en de toren niet aangevallen worden door de vijand (zie rode vakjes op Figuur 10). Voor de laatste maken we gebruik van de aanvalsmappen.

Figuur 10: Voorwaarden rokeren

Figuur 11: rechts rokeren met wit

De functie “castlingPossible” geeft voor een bepaald kleur de mogelijkheden van rokeren door a.d.h.v. een string, als de string links/rechts bevat kan de kleur links/rechts rokeren. Als het een lege string is, kan er niet worden gerokeerd en als het “links” en “rechts” bevat dan kan er langs beide kanten worden gerokeerd. De functie gaat als volgt:

```
castlingPossible(color){ // king checked zelf al of hij zelf niet gemoved heeft
    let speelveld=this.Board.getPieces();
    let attackmap=this.updateAttackMap(!color);
    let y_axis=color?7:0;// om te weten op welke y as de lege plekken moeten worden
```

```
gechecked
    if (attackmap[y_axis][4]>0) {//als koning schaak staat sws niks returnen
        return "";
    }
    let rookleft =speelveld[y_axis][0];
    let rookright=speelveld[y_axis][7];
    let string="";
    // rechtse kant checken
    if (rookleft!==0 && rookleft.getType().endsWith("rook") && rookleft.moved===false){
        let all_empty=true;//al plaatsen leeg
        for (let i = 1; i < 4; i++) {
            //kijken of geen stukken tussen toren en koning en niks aangevallen
            if(speelveld[y_axis][i]!==0 || attackmap[y_axis][i]>0 ){
                all_empty=false;
            }
        }
        if (all_empty){
            string+="left";
        }
    }
// linkse kant checken
    if (rookright!==0 && rookright.getType().endsWith("rook") &&
rookright.moved===false){
        let all_empty=true;//al plaatsen leeg
        for (let i = 5; i < 7; i++) {
            if(speelveld[y_axis][i]!==0 || attackmap[y_axis][i]>0 ){
                all_empty=false;
            }
        }
        if (all_empty){
            string+="right";
        }
    }
    return string;
}
```

De koning gebruikt deze functie dan bij het berekenen van zijn eigen “possibleMoves(Board)”. Dit enkel is niet genoeg, ook de toren in kwestie moet verplaatst worden. Hiervoor werd de “move” functie in de klasse “Board” lichtjes uitgebreid:

```
move(piece,cord){
    this.board[piece.pos.y][piece.pos.x] = 0; //vakje wordt leeg
    this.board[cord.y][cord.x] = piece; //verplaats naar cord
    //kijken voor rokeren (altijd een zet die verder gaat dan 1 vakje)
    if (piece.getType().endsWith("king") && Math.abs(piece.pos.x-cord.x)>1){
        // rechts
        if (cord.x===6){
            let rook=this.board[piece.pos.y][7];
            this.move(rook,new Coordinate(5,piece.pos.y));//rechtse toren verpl
        }
        //links
        if (cord.x===2){
            let rook=this.board[piece.pos.y][0];
            this.move(rook,new Coordinate(3,piece.pos.y));//linkse toren verpl
        }
    }
    piece.move(cord);
    // als het de pion aan einde komt-> promoveren naar queen
    if(piece.getType().endsWith("pawn") && piece.pos.y===piece.endY){
        this.board[piece.pos.y][piece.pos.x]=new
Queen(piece.pos,piece.kleur,true);
    }
}
```

FenConverter

De converter is geschreven om een positie op een bord om te zetten naar een string met de regels van FEN-notatie. Hiermee kan er een bord worden aangemaakt in een bepaalde positie. Dit was nodig om puzzels van de API te gebruiken (https://645b63c3a8f9e4d6e767035c.mockapi.io/Puzzels) en om bepaalde posities te testen. De functie die het bord codeert in een string bestaat ook. Deze wordt voorzien om later een terug-knop (undoMove) te maken die kan gebruikt worden tegen de bot. De bot speelt namelijk asynchroon op een andere *thread* a.d.h.v. javascript webworkers en kan dus enkel communiceren met serializeerbare objecten (bv JSON). Het gedeelte op de andere *thread* kan het schaakbordobject dus niet delen met de *main-thread*, dus zou een FEN-string de andere *thread* informeren hoe het bord er ten allen tijden uitziet (zie meer bij Multithreading). Het effectieve implementeren van de codeer en decodeer functies van FEN-notatie naar een instantie van de klasse “Board” worden hier niet uitgebreid besproken omdat deze eerder triviaal zijn.

De algoritmes

Om de schaakbot te laten werken zijn er algoritmes nodig voor het evalueren en selecteren van bepaalde zetten. In dit gedeelte wordt het minimax en negamax algoritme besproken alsook de algoritmes voor het evalueren van een bepaalde zet op het schaakbord. Er is gekozen voor het minimaxalgortime te vervangen door het negamaxalgoritme maar omdat beide gebruikt werden worden ze allebei besproken en wordt de keuze later duidelijk gemaakt.

Het minimaxalgoritme

Het minimaxalgoritme werd gekozen omdat het de optimale zet teruggeeft door over alle mogelijke zetten te itereren en die te evalueren. Het algoritme minimaliseert het maximale verlies door de beste zetten van de tegenstander proberen te voorspellen op basis van de evaluaties.

Basiswerking van minimax

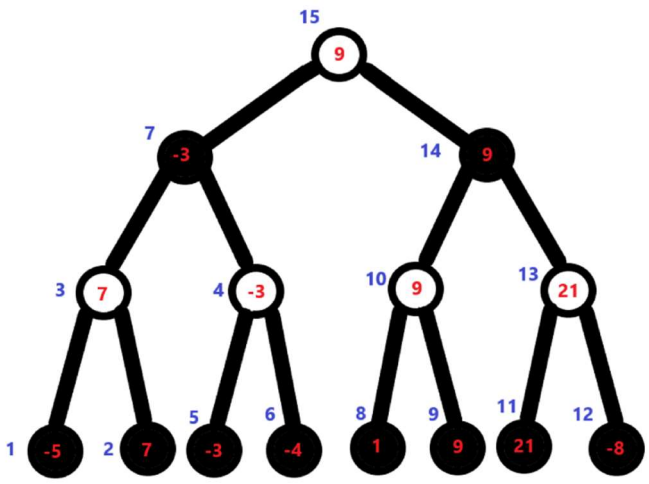
```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, false)
         maxEval = max(maxEval, eval)
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, true)
         minEval = min(minEval, eval)
      return minEval
```

Figuur 12: pseudocode minimax (Sebastian Lague, 2021, 03:58)

Bij de oproep van het minimax algoritme wordt er telkens een schaakbord, diepte en kleur (“true” of “false”) meegegeven zoals in Figuur 12, in Figuur 12 noemt het schaakbord “position”.
Dan wordt er eerst gecheckt of de diepte 0 is en één van de kanten schaakmat staat. Als dit zo zou zijn wordt de evaluatiefunctie opgeroepen die de meegegeven toestand van het bord evalueert en een score teruggeeft.
Daarna wordt er gecontroleerd of er over de witte mogelijke zetten moet worden geïtereerd of de zwarte door de booleaanse grootheid in “maximazingPlayer” te checken.
Als de minimaxfunctie is opgeroepen voor de witte kant zal er een variabele worden aangemaakt voor het bijhouden van de hoogste score, op Figuur 12 is dat “maxEval”. Hierna zal er over elke mogelijk zet worden geïtereerd en voor elke zet wordt minimax terug opgeroepen maar met het bord waarop een mogelijk zet is gedaan, de diepte-1 en voor tegenovergestelde kant. Dit moet voor de tegenovergestelde kant worden opgeroepen zodat er een voorspelling kan gedaan worden wat de tegenstander zal doen zodat hieruit de best zet zal worden gekozen. De score van elke evaluatie zal worden vergeleken met “maxEval” en “maxEval” zal dan de hoogste waarde krijgen van de twee.
Hetzelfde gebeurt voor de zwarte kant maar hier wordt de laagste score genomen en teruggegeven.
Omdat het algoritme zichzelf altijd oproept tot een bepaalde diepte voor elke mogelijk zet voor een kant kan dit voorgesteld worden in een boomstructuur zoals in Figuur 13Figuur 13: voorbeeld minimax zoekboom.
Voor het principe simpel te kunnen uitleggen wordt er hier een versimpelde versie van de boomstructuur gebruikt, in de praktijk zouden er uit elke knoop (veel) meer dan 2 takken komen.
De bovenste knoop is de eerste aanroep van het minimaxalgoritme. Zo werkt het algoritme zich een baan naar beneden en begint bij de linkse tak. De blauwe cijfers is de volgorde waarin alle waarden worden berekend in de knopen. Hier is visueel te zien dat bij de eerste aanroep de diepte 3 is en dat de witte knopen een zo hoog mogelijke score willen en dat de zwarte



Figuur 13: voorbeeld minimax zoekboom

knopen de laagste verkiezen. In de knopen helemaal onderaan in Figuur 13 is de verkregen diepte 0 en wordt de evaluatiefunctie opgeroepen die dan de waarden die in de knopen te zien zijn teruggeven.

Alpha-beta pruning

Het minimaxalgoritme kan uitgebreid worden zodat het veel efficiënter werkt. Dit kan met behulp van alpha-beta pruning. Hierbij worden bepaalde knopen niet meer berekent omdat deze overbodig zijn waardoor er veel minder rekenwerk voor nodig is.

Basiswerking alpha-beta pruning

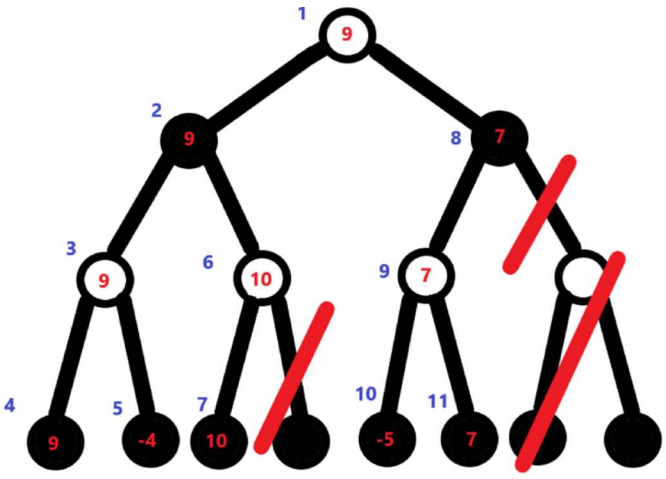
Er valt te zien dat in Figuur 14 is er weinig verschil met de originele werking van het minimaxalgoritme, er worden enkel 2 extra parameters meegegeven en er worden enkele extra controles uitgevoerd.
Bij de initiële oproep van het algoritme zullen de waarden van “alpha” en “beta” respectievelijk “-oneindig” en “+oneindig” zijn. De “alpha” zal hier de hoogste waarde in de zoekboom bijhouden en de “beta” de laagste. Als “beta” lager of gelijk is aan de waarde van alpha zal “maxEval” of “minEval” (hangt af van de waarde van “maximazingPlayer”) direct teruggeven worden.

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
        return minEval
```

Figuur 14: pseudocode minimax met alpha-beta pruning (Sebastian Lague, 2021, 08:52)

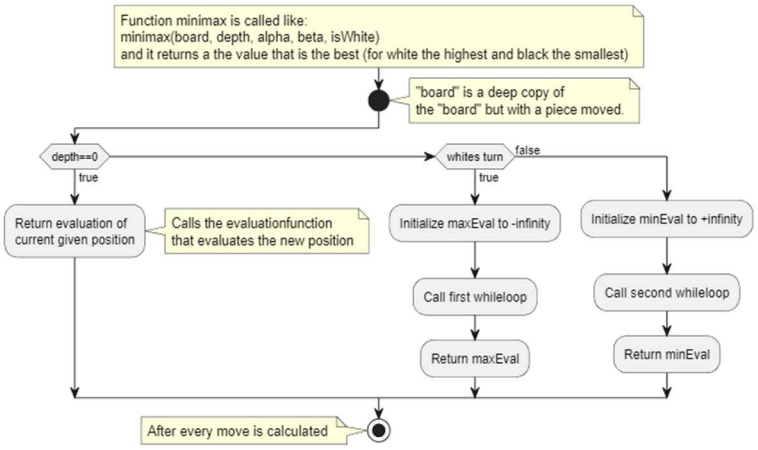


Figuur 15: voorbeeld alpha-beta pruning zoekboom

De manier waarop de boomstructuur tot stand komt is hetzelfde als die bij het standaard minimaxalgoritme maar hier zoals in Figuur 15 worden er enkele takken geschrapt omdat die overbodig zijn.
De blauwe nummering stelt de volgorde voor waarin het algoritme wordt opgeroepen.
Knopen “4” en “5” zullen als eerste worden berekend en knoop “3” zal dan de hoogste van de twee kiezen. Daarna zal knoop “7” berekent worden die de waarde 10 heeft. Rekening houdend met knoop “6” die de hoogste waarde van de onderliggende knopen gaat nemen zal de waarde van knoop “6” groter dan of gelijk zijn aan 10. Knoop “2” zal altijd de laagste waarde nemen de zijn 2 onderliggende knopen, als knoop “6” niet kleiner kan zijn als 10 gaat knoop “2” gegarandeerd voor de waarde van knoop 3 kiezen en is het berekenen van de tweede vertakking van “6” overbodig.
Hetzelfde geldt bij de onderliggende knopen van knoop “8”.

Implementatie

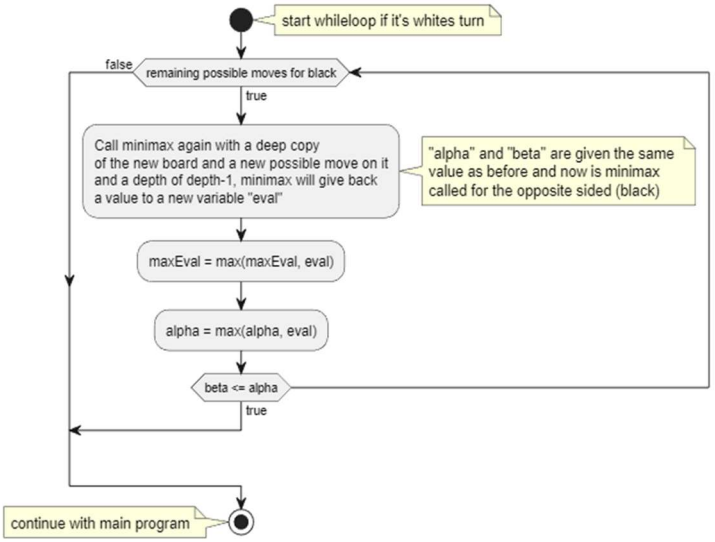
Het algoritme was geïmplementeerd in de botklasse zodat het zo efficiënt mogelijk een bepaalde zet kan berekenen als het nodig is.
Hiernaast zijn Figuur 16, Figuur 17 en Figuur 18 te zien die in het geheel één groot activiteitendiagram moeten voorstellen. Maar voor praktische redenen zijn deze opgedeeld in 3 verschillende stukken. In het hoofdgedeelte worden de twee while-lussen opgeroepen. De implementatie zelf kan teruggevonden worden in het bestand van de botklasse (in commentaar).



Figuur 16: hoofdgedeelte minimaxalgoritme

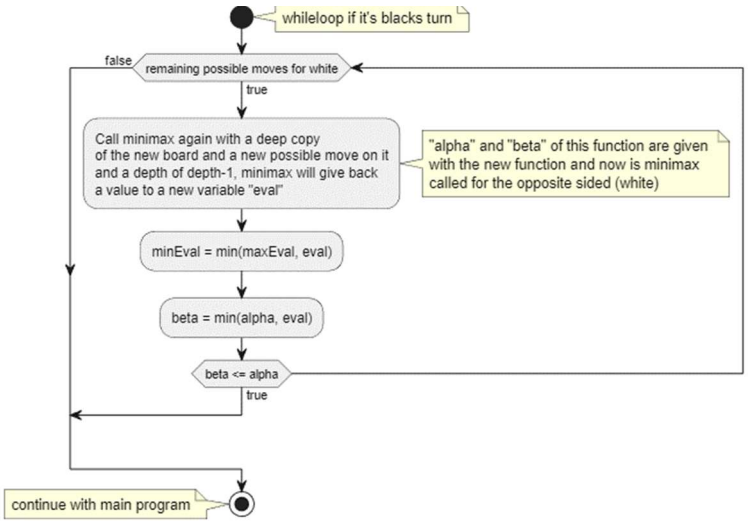
Het algoritme wordt opgeroepen door het schaakbord in zijn huidige toestand mee te geven, de gewenste zoekdiepte, waarden voor “alpha”, “beta” en een boolean als indicatie voor welke kleur het algoritme een zet moet kiezen.

Figuur 16 is het hoofdgedeelte van het algoritme en hier wordt eerst gekeken of de diepte niet gelijk is aan nul, als dit wel het geval is, wordt de evaluatiefunctie opgeroepen uit de evaluatieklasse. Anders wordt er gekeken voor welke kleur een waarde moet teruggegeven worden.



Figuur 17: whileloop 1 minimaxalgoritme

Als de variabele “isWhite” de waarde “true” heeft dan wordt er geïtereerd over elke mogelijke zet van de schaakstukken van wit zoals in Figuur 17 te zien is. Voor elke mogelijk zet wordt er een diepe kopie gemaakt van het schaakbord met een nieuwe zet op. Die wordt dan meegegeven bij het opnieuw oproepen van het algoritme. Bij het opnieuw oproepen van het algoritme wordt de diepte met 1 verminderd en de meegegeven waarde voor “isWhite” is hier nu “false”. De teruggegeven waarde van het algoritme wordt in een nieuwe variabele “eval” opgeslagen. Hierna wordt er in “maxEval” en in “alpha” het maximum tussen “maxEval” en “val” gestoken. Dan wordt er gecheckt of beta kleiner of gelijk is aan alpha voor de alpha-beta pruning en als dit het geval is wordt de lus onderbroken en gaat het verder met het hoofdgedeelte in Figuur 16 waar “maxEval” wordt teruggegeven.



Figuur 18: whileloop 2 minimaxalgoritme

In Figuur 18 worden dezelfde stappen gevolgd op een paar details na. Hier wordt er geïtereerd over elke mogelijk zet van de zwarte stukken en wordt het algoritme terug opgeroepen met “isWhite” op true. Er wordt ook gewerkt met “minEval i.p.v. “maxEval” en dat is omdat hier telkens het minimum van “eval” en “minEval” wordt genomen om in “minEval” en “beta” te stoppen.

Het negamaxalgoritme

Het negamaxalgoritme heeft in principe dezelfde werking als het minimaxalgoritme maar heeft veel minder gedupliceerde code en is iets efficiënter qua werking.

Basiswerking

Het valt in Figuur 19 op dat er hier veel minder code is dan in het minimaxalgoritme. Dit komt omdat er hier niet moet gecontroleerd worden voor welke kleur er moet geëvalueerd worden. De evaluatie die hier wordt opgeroepen zal hier positief zijn voor zowel wit als zwart en als het algoritme zichzelf oproept wordt de teruggegeven waarde van teken gewisseld en de diepte met 1 afgetrokken. Hierdoor moet er enkel gekeken worden naar de grootste waarde voor de beste zet te kunnen berekenen.

```
int negaMax( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

Figuur 19: pseudocode negamax (Negamax - Chessprogramming wiki, z.d.)

Alpha-beta pruning

Hier is het principe nog altijd hetzelfde als die bij het minimaxalgoritme. Elke keer als het negamaxalgoritme zichzelf oproept zullen alpha en beta wisselen van positie en veranderen van teken zodat hetzelfde resultaat wordt bekomen als de berekening bij het minimaxalgoritme.

De evaluaties

De evaluatiealgoritmes zijn nodig voor de toestand van een schaakbord (posities van schaakstukken) te evalueren. Dit lukt omdat voor er voor elke positie een evaluatie zal gebeuren door negamaxberekeningen of deze evaluatiealgoritmes. Het schaakbord zal dus altijd meegegeven worden bij elke oproep van de evaluaties. Hiermee kunnen bepaalde zetten worden geëvalueerd door de bot waardoor hij de beste zet kan kiezen op basis van de evaluaties. Er zijn voorlopig drie soorten evaluaties in gebruik door de bot waaronder: waarde van de schaakstukken opgeteld, schaakmatevaluatie en schaakstuktafels. De teruggeefwaarde van elke evaluatie zal worden opgeteld in een centrale functie die dan de uiteindelijke score teruggeeft aan de oproeper.

Schaakstukwaarden optellen

Schaakstukken hebben elk een waarde die afhangt van het stuk. De waarde die toegekend is aan een pion is 100, aan een paard is 300, aan een loper is 300, aan een toren is 500, aan een dame is 900 en de koning heeft een waarde van 5000.
Hiermee kan deze evaluatie worden uitgevoerd door over elk schaakstuk te itereren en die waarde op te tellen bij de uiteindelijke evaluatiescore.

Schaakmatevaluatie

Bij de schaakmatevaluatie zal er worden gekeken of in het meegegeven bord een koning schaakmat staat. Als dit het geval is zal die de gepaste waarde teruggeven. Als wit schaakmat staat is dit voordelig voor zwart en zal er een score van -10 000 worden teruggegeven en als zwart schaakmat staat zal er een waarde van 10 000 worden teruggegeven. Deze waarden waren gekozen omdat in het minimaxalgoritme zwart voordeel geeft aan de laagste waarde en omgekeerd voor wit.

Schaakstuktafels

De schaakstuktafels waarvan het idee is gehaald van de site (*Simplified Evaluation Function - Chessprogramming wiki*, z.d.) zorgen ervoor dat de schaakstukken die op niet gunstige plaatsen staan strafpunten krijgen en omgekeerd. Omdat de toestand van het schaakbord wordt opgeslagen in een twee dimensionale array waarbij elk stuk kan opgeroepen worden door een coördinaat worden deze tafels ook zo opgeslagen in een array. Maar in plaats van een schaakstuk of niks terug te krijgen van die coördinaat

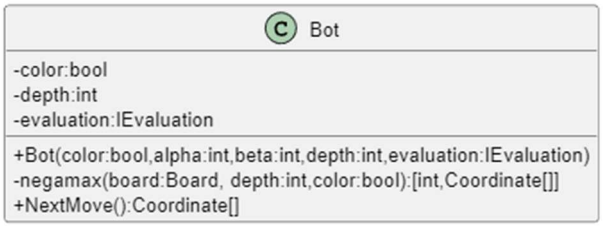
zul je een bepaald score terugkrijgen van die positie. De score op een positie hangt ook af van welk stuk er moet worden geëvalueerd. Zo is het bijvoorbeeld beter om een paard in het midden van het bord te zetten zodat die meer vakjes beschermt/aanvalt, dan de koning in het midden zetten (waar deze in gevaar staat).

De botklasse

Om de schaakwebsite volledig te maken is er een bot toegevoegd waartegen gespeeld kan worden. Het is belangrijk dat deze klasse zo efficiënt mogelijk werkt.

Structuur

De bot heeft slechts drie attributen nodig waaronder de kleur waarmee hij speelt, de diepte waarop het negamaxalgoritme moet zoeken en een evaluatieattribuut waardoor er *Dependency Injection* kan gebruikt worden. In de constructor worden deze attributen dan meegeven. Hier wordt dan ook het negamaxalgoritme geimplementeerd zodat de bot de best zet kan genereren op basis van de evaluaties.



Figuur 20: Botklasse

De functie “NextMove” wordt gebruikt voor de volgende zet terug te geven die het negamaxalgoritme heeft berekend. Deze functie geeft dan een array terug met de coördinaat van het te verplaatsen schaakstuk en heeft het coördinaat terug van de nieuwe positie van het schaakstuk.

Implementatie algoritme

In het begin wordt er een variabele “speelveld” gedeclareerd die de array van het schaakbord bijhoudt met elk schaakstuk op, er wordt ook een variabele “bestScore” gedeclareerd die de beste score bijhoudt en “move” die een array zal bijhouden met de beste score, coördinaat van het te verplaatsen stuk en de nieuwe coördinaat voor het te verplaatsen stuk.
Hierna wordt er gecontroleerd of de diepte 0 is en voor welke kleur er moet geëvalueerd worden. Voor zwart is de evaluatie steeds negatief, dus deze wordt van teken omgedraaid. Er zal een array worden teruggegeven met de score die de evaluatiefunctie teruggeeft en een “null”. Die “null” staat daarvoor consistent te blijven met het formaat van de teruggeefwaarde van de functie.

```
let speelveld = board.board;
let bestScore = undefined;
let move = [];
if(depth === 0) {
    if(color)
        return [this.evaluation.evaluate(board), null];
    else
        return [-this.evaluation.evaluate(board), null];
}
```

Bij de onderstaande code wordt er geïtereerd over alle schaakstukken van de kleur waarmee de “negamax” functie is opgeroepen. Daarna worden alle mogelijke zetten opgehaald met de functie “possibleMoves” die de pseudolegale zetten teruggeeft. Dan wordt er geïtereerd over de mogelijke zetten. Voor elke mogelijke zet wordt er een kopie gemaakt van het bord en de “piece”. In “fakePiece” zit hetzelfde stuk als in “piece” maar “fakePiece” bevindt zich op de kopie. Dan wordt het “fakePiece” verzet naar de nieuwe coördinaat en daarna wordt er gecontroleerd dit een legale zet was (kijken dat een koning al dan niet schaak staat). Dit gebeurt door de “isChecked” functie te gebruiken van het “legalchecker”-object, die een attribuut is van het gekloonde bord. Dit proces is heel gelijkaardig als de “possibleMoves” functie in de legalchecker klasse (zie Legalchecker), maar toch wordt deze niet gebruikt. Dit komt omdat voor het algoritme al een kloon van het bord nodig was en de functie “possibleMoves” het bord ook meerdere keren moet klonen, dit zou veel overhead creëren bij het uitvoeren van het algoritme.

Bij de legale zetten zal het Negamaxalgoritme zichzelf terug oproepen met de gekloonde versie van het bord, een diepteniveau lager, alpha en beta (omgedraaid en veranderd van teken) en met de tegenovergestelde kleur (“true” of “false”). Omdat negamax altijd een array zal teruggeven met twee

elementen (de evaluatie en een array van 2 coördinaten), kunnen er hier gemakkelijk variabelen aan toegekend worden. De variabele “array” wordt hier nergens gebruikt omdat deze enkel nodig is voor de functie die “negamax” voor de eerste keer oproept, in dit geval is dit “nextMoves” functie die de array gebruikt om de volgende beste zet van de schaakbot terug te geven. De score wordt van teken omgedraaid omdat de beste score gewenst is voor de kleur waarmee de functie is mee opgeroepen
Vervolgens wordt er gecontroleerd of de score groter is dan beta en als dit zo is zal een array van de score en de gepaste coördinaten teruggegeven worden. Hier wordt er in de array naast de score de x-en y-waarde teruggegeven van de huidige positie en dan het nieuwe coordinaat. Dit gebeurt zodat er niet altijd een coordinaat-object moet aangemaakt worden.
Als de score niet groter is dan beta dan wordt er gecontroleerd of de “bestScore” nog niet is gedefinieerd of als de score groter is dan bestScore. In het geval dat één van de twee waar zijn, zal “bestScore” de waarde krijgen van “score” en zal “move” de bijhorende array met score en coördinaten bijhouden. Hierdoor wordt telkens alleen de beste zet en zijn score bijgehouden. Als laatste wordt er ook gekeken of “score” groter is dan “alpha” zodat in het geval dat dit waar is, “alpha” de waarde van “score” overneemt.

```
for(let y = 0; y < 8; y++){
    for(let x = 0; x < 8; x++){
        let piece = speelveld[y][x];
        if(piece !== 0 && piece.kleur === color){ // is het een wit/zwart stuk
            let posMoves = piece.possibleMoves(board); //speudolegale zetten
            for(let cord of posMoves){ //itereren over zetten
                let cloneBoard = board.clone(false);
                let fakePiece = cloneBoard.board[piece.pos.y][piece.pos.x];
                cloneBoard.move(fakePiece, cord); // zet doen op vals bord
                if (!cloneBoard.legalchecker.isChecked(piece.kleur)) { //legal?
                    let [score, array] = this.negamax(cloneBoard, depth-1, -
beta, -alpha, !color);
                    score *= -1;
                // alpha beta pruning
                    if(score >= beta){
                        return [score, [x, y, cord]];
                    }
                // is het de beste score?
                    if(bestScore === undefined || score > bestScore){
                        bestScore = score; // nu de beste score
                        move = [x, y, cord]; // beste move
                        if(score > alpha){
                            alpha = score;
                        }
                    }
                }
            }
        }
    }
}
```

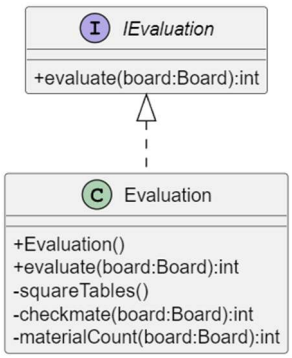
Hierna gebeurt er nog een controle of “bestScore” ongedefinieerd is. Dit kan gebeuren als er geen mogelijke zetten meer waren voor elk schaakstuk (schaakmat of patstelling). Als dit niet zo is zal de best berekende zet met zijn score worden teruggegeven. Indien dit wel zo is, is er nog geen zet beste zet berekend en wordt enkel de evaluatie van die positie teruggegeven.

```
if(bestScore === undefined){
    if(color)
        return [this.evaluation.evaluate(board), null];
    else
        return [-this.evaluation.evaluate(board), null];
}
return [bestScore, move];
```

De evaluatieklasse

Structuur

De evaluatieklasse “Evaluation” wordt afgeleid van de interface “IEvaluation” zodat er later gemakkelijk uitgebreid kan worden en andere evaluatiemethoden kunnen gebruikt worden. De interface heeft de functie “evaluate” waarbij telkens een bord moet worden meegegeven zoals er te zien is in Figuur 21.
Omdat “Evaluation” dit implementeert zal die de “evaluate” functie moeten overschrijven. De “evaluate” functie maakt gebruik van 3 private functies om de positie van een schaakspel te evalueren.



Figuur 21: klassendiagram Evaluation

Werking

De private functie “squareTables” dient enkel voor het initialiseren van de schaakstuktafels. Dit werd gedaan omdat dit de structuur van de code een stuk overzichtelijker maakt.

In de functie “materialCount” die hieronder weergegeven staat, zal er zoals in het negamaxalgoritme over elk coordinaat geïtereerd worden in de array van het bord. Als er een stuk op dat coordinaat staat zal er gekeken worden van welk type het is door de “getType” functie op te roepen van “piece” en zal er bij de score de waarde worden opgeteld van het stuk. Ook worden de posities van de koningen opgeslaan. Dit werd gedaan voor een evaluatiefunctie die nog zou geïmplementeerd worden, maar nog niet in gebruik is omdat die nog meer moet getest worden. Als het geen koning is zal er in de “tableDictionary” worden gekeken, die elke schaakstuktafel bijhoudt. Deze zal de gepaste tafel voor het gepaste stuk oproepen en die zal volgens het schaakstuk zijn coördinaat een waarde teruggeven die bij de score zal worden opgeteld. Uiteindelijk zal de totaalscore teruggegeven worden.
De evaluatie van de schaaktafels gebeurt dus ook in de functie “materialCount” en dit omdat er maar één keer over alle coördinaten zou moet worden geïtereerd.

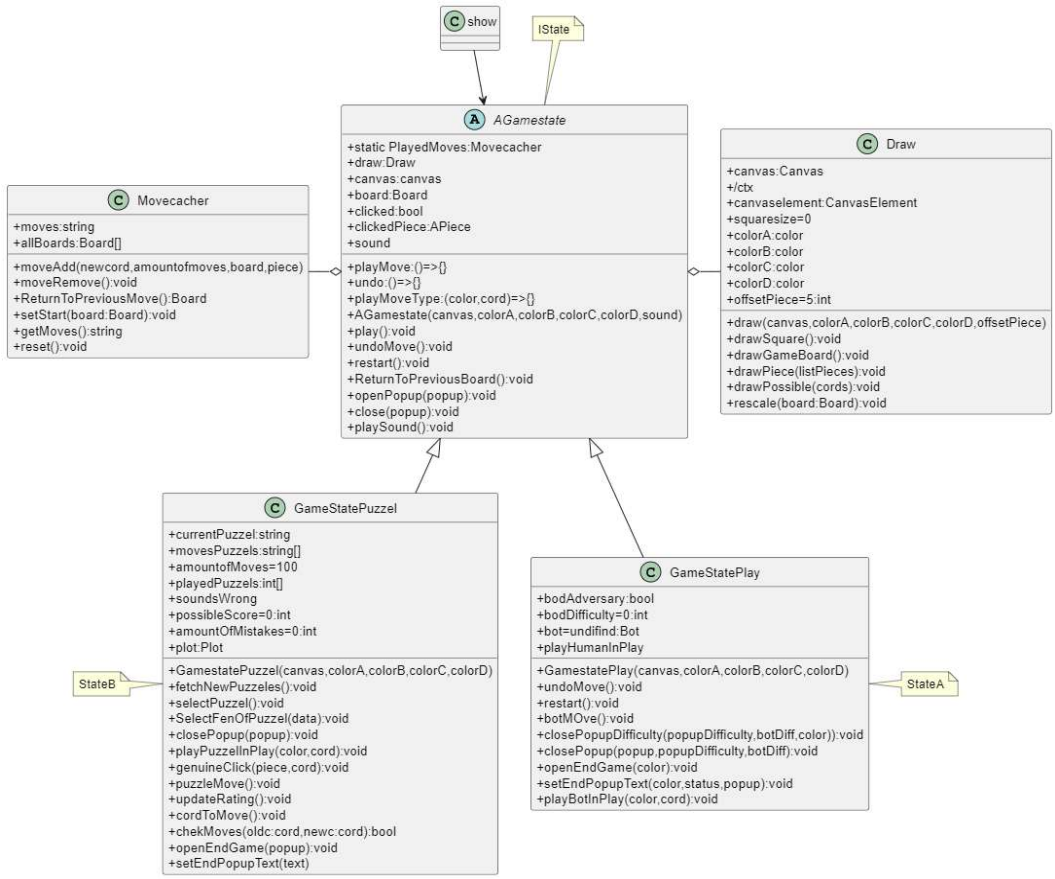
```
materialCount(board) {
    let speelveld = board.board;
    let score = 0;
    for (let y = 0; y < 8; y++) {
        for (let x = 0; x < 8; x++) {
            let piece = speelveld[y][x];
            if (piece !== 0) {
                let type = piece.getType();
                score += piece.value;
                if (type.includes("king")) {
                    if (piece.kleur) {
                        this.whiteking = piece;
                    } else {
                        this.blackking = piece;
                    }
                } else {
                    score += this.tableDictionary.get(type)[y][x];
                }
            }
        }
    }
    return score;
}
```

De “checkmate” zal controleren of er een koning schaakmat staat en zal de juiste waarde teruggeven naargelang de kleur dat schaakmat staat. Er zal eerst gecontroleerd worden of er een koning schaak staat en daarna pas of er één schaakmat staat omdat de “isEnd” functie van het “board” een zware functie is.

```
checkmate(board){
    let val = 0;
    if (board.legalchecker.isChecked(true) && board.isEnd(true) === "checkmate") {
        val += -10000;
    }
    if (board.legalchecker.isChecked(false) && board.isEnd(false) === "checkmate") {
        val += 10000;
    }
    return val;
}
```

ViewController

Er zijn verschillende manieren om te schaken op de website. De viewcontroller monitort deze manieren van schaken. De eerste manier is tegen een andere persoon spelen op dezelfde computer. Als je alleen bent kan de site nog altijd gebruikt worden. Dit kan namelijk door middel van puzzels te maken of door tegen een bot te spelen. Dit wordt verwezenlijkt door een template method pattern. Hier zou ook gebruik kunnen gemaakt worden van een strategy pattern. In dit geval is gekozen voor de template method, om op die manier duplicate code te vermijden. Dit zou moeilijker geweest zijn met het strategy pattern. De gebruikte template method omvat de functie play en de functie “playMoveType”. Deze laatste functie wordt aangepast naargelang de staat. De template method is gebruikt omdat grote delen van de code in de functie play anders gelijk zouden zijn. Door deze in de abstracte klasse te plaatsen, worden deze duplicaten van code weggewerkt. Er wordt later teruggekomen op de code van play na het schetsen van de taken van de klassen. De klasse “AGamestate” en zijn afgeleide klassen staan in voor alle functionaliteiten op de site. Hiermee wordt bedoeld: de knoppen in het schaakscherm en dus niet de navigatiebalk. Ook het effectief spelen van het spel wordt in deze klasse gedaan. De mogelijkheid om zetten te doen wordt in de klasse uitgewerkt. Hiervoor dient dan ook de “Draw”-klasse, deze zorgt dat een verandering op het bord door een speler ook getoond wordt. Daarnaast heeft “AGamestate” nog een andere klasse namelijk “Movecacher”. Deze klasse staat in voor het tonen van de zetten. Dit wordt gedaan aan de hand van een string notatie, gebaseerd op de regels van de FEN-notatie. De zetten worden dan getoond aan de zijkant van het scherm. Deze drie klassen zitten een beetje in elkaar verweven. Ze zijn zeer sterk afhankelijk van elkaar. “AGamestate” zou niet kunnen werken zonder de “Draw”-klasse. “Movecacher” zit ook sterk vervlochten in de klassen. Het verband tussen de drie klassen gaat zeker teruggevonden worden in de code. Op die code kan nu specifieker ingegaan worden met de voorafgaande kennis over de klassen.



Figuur 22: viewcontroller klasse diagram

AGamestate

De eerste klasse waar de code van dichterbij wordt bekeken is de “AGamestate”-klasse. In deze klasse wordt de functie play besproken.

```
play(event){
    let rect=this.canvas.getBoundingClientRect();
    let x=Math.floor((event.clientX-rect.x)/this.draw.squareSize);
    let y=Math.floor((event.clientY-rect.y)/this.draw.squareSize);
    let piece_clicked_now=this.board.getPieces()[y][x];
    let cord=new Coordinate(x,y);
    let color=this.board.colorToMove();
    if(this.clicked){

        this.playMoveType(color,cord);

        this.clicked=false;
        this.clickedPiece=0;

    }else{
        if(piece_clicked_now!==0 && piece_clicked_now.kleur===color){
            this.draw.drawPossible(this.board.possibleMoves(cord));
            this.clickedPiece=piece_clicked_now
            this.clicked=true;
        }
    }
}
```

Deze functie reageert anders naar gelang de staat waarin het spel zich bevindt. Daarnaast voert deze functie iets anders uit bij de eerste klik dan bij de tweede klik. Vandaar dat de variabele “clicked” wordt bijgehouden. Deze staat wordt op “true” gezet als er geklikt wordt op een schaakstuk, indien niet blijft deze op “false” staan. Deze klik wordt altijd op dezelfde manier behandeld. Vanuit het verzonden klik event worden de coördinaten op het scherm gevraagd. Daarna worden de coördinaten van de linkerbovenhoek van het canvas afgetrokken van de klik coördinaten. Dit zorgt ervoor dat we de nulpunten van het assenstelsel niet meer hebben staan in de linkerbovenhoek van het hele scherm, maar wel in de linkerbovenhoek van het canvas. De overgebleven waarden worden dan gedeeld door de grote van een vierkantje. De grootte van een zijde wordt bijgehouden in de Draw klasse. Hierdoor krijgen we een getal tussen nul en acht, acht niet inbegrepen. Door de *floor* krijgen we dan een getal uit het discreet interval nul tot en met zeven. Dit interval werd bevestigd door te testen. Daarna wordt er gekeken wat er op die plaats op het bord staat. Als laatste stap wordt de klik omgezet naar een “Coordinaat”-object. Al deze stappen zijn voor elke klik hetzelfde. Doordat dit altijd gelijk is, paste dit deel perfect in de abstracte klasse.
Ook de code die uitgevoerd moet worden als er nog niet geklikt is op een schaakstuk past in deze template Method. Als dit wel gebeurt, gaan alle mogelijke zetten getoond worden door de ”Draw”-klasse. Daarna worden de parameters van de abstracte klasse “clickedPiece” en “clicked” aangepast. Bij de tweede klik wordt “PlayMoveType” uitgevoerd en dan worden de twee parameters weer naar hun standaardwaarde gezet. Voor “clicked” is dit “false” voor “clickedPiece” is dit nul. De functie “PlayMoveType” kan verschillen van invulling naar gelang de situatie. Voorbeelden van deze invullingen zijn: jouw zet doen en dan de schaakbot laten zetten of jouw zet doen en dan de volgende zet van de puzzel doen. Als je tegen een vriend speelt is de invulling nog anders. Hoe het verschil wordt gemaakt tussen het spelen tegen een bot en een andere persoon wordt in de GamestatePlay uitgewerkt.

GamestatePlay

In de “GamestatePlay”-klasse staan er nog een paar belangrijke functies voor het geheel. De eerste twee functies die hier besproken worden zijn “PlayBotInPlay” en “PlayPlayerInPlay” deze twee functies worden toegekend aan de parameter “PlayMoveType”. Dit wordt gedaan naar gelang de gekozen tegenstander.

Op de site gebeurt dit kiezen aan de hand van een pop-up. De uitwerking van deze pop-ups gebeurt door een combinatie van javascript, CSS en HTML. De pop-up wordt geopend door de functie open of close met als parameter een pop-up. Dit gebeurt aan de hand van een klasse die wordt toegevoegd aan de klassen van een HTML-element. Het HTML-element namelijk een “div” staat altijd op “hidden” bovenaan de pagina. De pop-up die bovenaan de pagina staat is ook herschaalt naar een hele klein pop-up. Wanneer de klasse hieraan wordt toegevoegd komt deze op zichtbaar en wordt de pop-up naar het midden van het scherm gezet. Via een CSS-animatie lijkt die dan te vergroten en naar het midden van het scherm te bewegen. Deze verplaatsing gebeurt van boven naar beneden. Onderstaande functie is een voorbeeld van hoe een pop-up behandeld wordt.

```
closePopup(popup,popupDifficulty,enemy){
    let type=parseInt(enemy.value)
    this.botAdversairy= type!==0;
    if(!this.botAdversairy){
        this.playMove=(event)=>{this.play(event)};
        this.playMoveType=(color, cord)=>{this.playHumanInPlay(color,cord)};
    }else{
        this.openPopup(popupDifficulty)
    }
    this.close(popup);
}
```

In deze functie wordt de informatie over de tegenstander gehaald. Naargelang de verkregen informatie wordt er iets specifieks gedaan. Als de tegenstander een speler blijkt te zijn, dan gaan de functies “playMove” en “playMoveType” worden ingevuld. Is de tegenstander een schaakbot, dan gaat eerst nog een andere pop-up getoond worden. Een keer dat de pop-ups allemaal gepasseerd zijn kan je beginnen met schaken.

Movecacher

Wanneer er geschaakt wordt, worden de zetten getoond aan de zijkant van het scherm. Hier wordt aan de hand van een string de hele geschiedenis van de zetten bijgehouden. Daarnaast wordt elke positie op het bord bijgehouden aan de hand van een array van gekloonde borden. Dit om aan de hand van deze borden de “undoMove” te realiseren. Dit kan ook op snellere manieren gedaan worden. Hierover is er meer uitleg gegeven bij het deel over het algoritme. Er is hier echter gekozen voor de iets tragere variant. Dit omwille van verschillende redenen. De snelheid van het bijhouden en ongedaan maken van de zetten was niet belangrijk omdat dit niet zo vaak uitgevoerd wordt. Daarnaast is het in dit geval ook mogelijk om meerdere zetten terug te gaan of zelfs onmiddellijk naar het begin te gaan. Dit zou voor latere uitbreidingen gemakkelijk kunnen zijn. Bijvoorbeeld voor het herbekijken van een spel kan je van zet zeven naar twaalf, en dan weer naar zeven enzovoort. Een tweede voordeel is dat het iets gemakkelijker is om uit te werken.

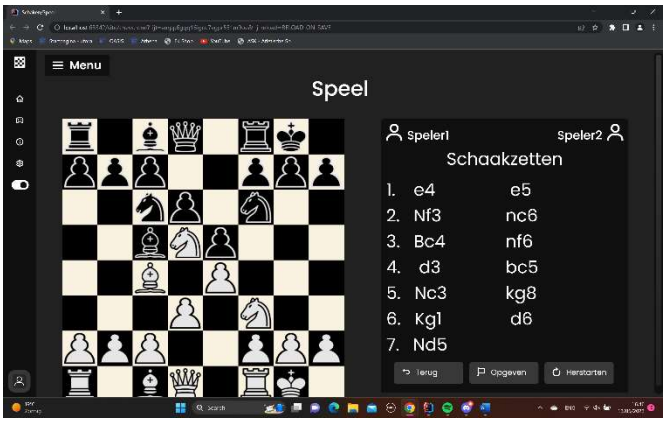
Dit wil echter niet zeggen dat dit zonder fouten gelukt is. Bij de functie “ReturnToPreviousMove” zijn er veel problemen geweest. Vandaar dat op deze functie nog eens dieper wordt ingegaan.

```
ReturnToPreviousMoves(){
    let len=this.allBoards.length;
    if(len>=1){
        this.MoveRemove();
        this.allBoards.pop();
        return this.allBoards[len - 2].clone(true);/
    }else{
        return this.allBoards[len-1];
    }
}
```

“ReturnToPreviousMove” gaat de laatste zet uit de string met zetten halen indien er meer dan één “Board”-object in de array met borden zit. Dit omdat het eerste bord de beginpositie is. Op dat moment zijn er dus ook nog geen zetten gedaan. De hulpfunctie “MoveRemove” verwijdert dan effectief de zet uit de string. Daarna wordt het laatste bord uit de array verwijdert. Als laatste wordt het bord teruggegeven als clone. Het klonen van het bord was nodig voor als er na de *undo move* weer verder gespeeld wordt. Als het bord

niet gekloond wordt dan ligt er een referentie tussen het huidige bord en het bord in de array. Als er dan een zet gedaan wordt gaat dit ook gebeuren op het bord dat in de array zit. Hierdoor gaat bij volgende *undo moves* er een fout optreden. Er gaat namelijk terug gegaan worden naar het bord met een referentie naar het huidige bord. Hierdoor gaat de “maak ongedaan” knop niet meer juist werken. Dit probleem is dus opgelost door het klonen van het bord zodat de referentie er niet meer ligt. Als er nog geen zet gemaakt is gaat men het startbord teruggeven. Naast het terugkeren naar vorige zetten moesten er ook zetten toegevoegd kunnen worden.

Draw klasse



Figuur 23: voorbeeld getekende schaakpositie

Voor het tonen van het schaakbord wordt een canvas gekozen. Dit kan op verschillende manieren gebruikt worden.
De eerste manier is gewoon foto’s van de stukken boven op het bord tekenen. Wanneer een foto van een schaakstuk verplaatst wordt, wordt er dan een foto op de volgende plaats getekend. Boven op de vorige foto wordt vervolgens het juiste vakje opnieuw getekend.
De tweede manier is met animaties werken. In javascript bestaat de functie “requestAnimationFrame” deze gaat het canvas mee updaten op hetzelfde tempo als het tempo waarmee Google Chrome zichzelf hertekend. In dit project is er gekozen voor de eerste optie. Dit omdat deze eenvoudiger uit te werken was en meer dan snel genoeg update om alles te tekenen. Echter voor heel snelle spelletjes is de tweede manier meer aan te raden.
De eerste manier is zodanig uitgewerkt dat de foto’s van alle schaakstukken in het begin ingeladen worden. Doordat dit inladen maar een keer hoeft te gebeuren kan de foto heel snel opnieuw opgehaald worden. Door dit te gebruiken kan het herteken van het bord snel gebeuren. Snel genoeg zodat er geen flikkeringen in het scherm lijken te zijn bij het hertekenen. Het tekenen van het bord wordt volledig gedaan door de “Draw”-klasse. In deze klassen worden zowel functies geschreven voor het tekenen van de schaakstukken, als voor het tekenen van het bord zelf. De “Draw”-klasse krijgt zijn kleuren mee in de constructor. Ook het canvas krijgt hij mee in de constructor.

Verbeteringen Viewcontroler

Aan het viewmodel kunnen nog verbeteringen toegebracht worden. De “GamestatePlay” kan nog uit elkaar getrokken worden in twee aparte klassen. Dit zou ervoor zorgen dat elke klasse maar een taak heeft. Deze taak is dan de manier waarop de website moet reageren op bepaalde events. Door het uit elkaar trekken van de GamestatePlay klasse zou ook de code rond de pop-ups naar een aparte klasse kunnen gezet worden. Deze zou dan de juiste gamestate aanmaken naar gelang het de situatie. Dit zou dan ook bijdragen aan het scheiden van belangen een van de OO-principes. Het uitwerken van deze optimalisatie van de code is nog niet gebeurd. Het opkuisen van de code was minder prioritair dan het afwerken van de bot en het schaakspel. Dit aangezien het afwerken van het spel een veel grotere invloed heeft op de gebruikers van de site. De achterliggende code is voor de gebruiker minder belangrijk. Vandaar zijn eerst andere delen van de code afgewerkt voor de code opgekuist is.

De View

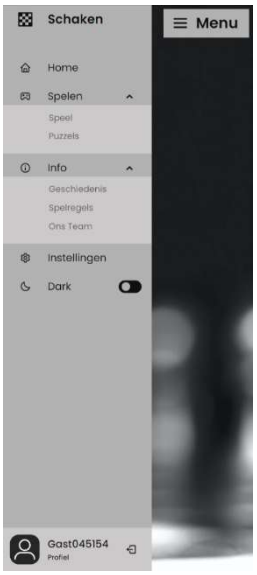
Het belangrijkste bestand in de *view* module is “Show.js”, deze wordt ingeladen door de HTML-pagina’s waar het schaken wordt gespeeld. Het bestand maakt alle nodige objecten aan en koppelt alle events aan de juiste functies van de *ViewController*-module. Ook zorgt het ervoor dat het schaakbord een responsieve lay-out heeft bij herschaling van het venster. In de “view” module zitten ook enkele andere javascript bestanden die functionaliteiten op andere HTML-pagina’s mogelijk maken. Enkele belangrijke functies van die andere bestanden worden nog kort besproken.

Navigatiebalk

Ook de navigatiebalk was een belangrijk onderdeel van het project. Een manier om makkelijk tussen de HTML-pagina’s te navigeren is een must. Er is dan ook gekozen om veel tijd te besteden aan dit aspect. De navigatiebalk werd volledig *custom* opgebouwd. Enkele interactieve functies werden ook toegevoegd, zoals het openen en sluiten van de navigatiebalk en het openen en sluiten van de subnavigatievensters bij onderdeel “Spelen” en “Info”. Ook werd een donkeremodus toegevoegd. Daarover meer in het onderdeel *Sustainable webdesign*. De niet-interactieve functies en opmaak werden volledig opgebouwd met HTML en CSS.

Om het inklappen en uitklappen van het navigatievenster te doen werken is gebruik gemaakt van een javascriptbestand “Sidebar.js”. Hier werden de functies uitgewerkt voor het interactieve deel van de sidebar. Als op de knop wordt geduwd wordt de classname in de HTML aangepast. De CSS zal dan reageren op de huidige classname en dus zal er verandering optreden, die in CSS is opgemaakt. De breedte van het venster zal onder andere dus versmallen. De inklapbare subvensters werken op een gelijkaardige manier.

Bij iedere oproep van het “onclick-event” wordt er ook een *boolean* opgeslagen in de localstorage. Zo zal bij herladen of bij openen van een andere pagina het uitgeklapte of ingeklapte venster bewaard blijven. Dit is geprogrammeerd in het javascriptbestand “Onload.js”. Indien Angular werd gebruikt ging dit makkelijker en beter uitwerkbaar zijn, want op deze manier wordt bij het veranderen van HTML-pagina telkens nog de CSS-animatie uitgevoerd, wat eigenlijk niet de bedoeling is. Hetzelfde geldt voor de donkeremodus. Er zal telkens een flits zichtbaar zijn bij het wisselen van pagina.



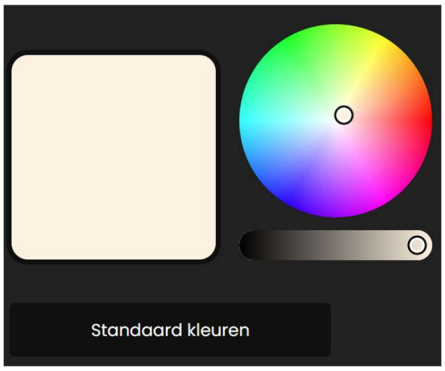
Figuur 24: navigatiebalk

```
function SidebarToggle(){
    const wasClosed = localStorage.getItem("closed") === "true";
    localStorage.setItem("closed", !wasClosed);
    const sidebar = document.querySelector(".sidebar");
    sidebar.classList.toggle("close", !wasClosed);
}
function OnloadMode(){
    document.querySelector(".sidebar").classList.toggle("close",
localStorage.getItem("closed") === "true");
```

Instellingen

Kleur

De instellingenpagina is een van de laatste toevoegingen aan het project. Het werd toegevoegd zodat de gebruiker meer opties heeft om de website aan zijn persoonlijke voorkeuren aan te passen. Zo kan men de kleuren van het schaakbord kiezen. Deze werden opnieuw opgeslagen in localstorage. Het “view.js” bestand zal die data ophalen en op basis van u gekozen kleuren een Gamestate (schaakspel) aanmaken. Voor het kiezen van de kleuren werd gebruik gemaakt van de iro ColorPicker. Dat is een bestaande API de online werd gevonden. De ColorPicker API werd toegepast zoals in onderstaande code.



Figuur 25: colorpicker

```
let colorIndicator = document.getElementById('color-indicator');
let colorPicker = new iro.ColorPicker('#color-picker', {width: 180, color:
"#fff"});
colorPicker.on('color:change', function (color) {
    colorIndicator.style.backgroundColor = color.hexString;
    OpslaanKleur1Localstorage(color.hexString);
    DrawPreview();
});
function OpslaanKleur1Localstorage(color){
    localStorage.setItem('color1', color);
}
```

Geluid

Geluid is ook toegevoegd aan de website. Dit verbetert de gebruikservaring, want spelers weten zo wanneer er een zet is gebeurd. Er wordt toegelaten om het geluid te veranderen of om aan en af te zetten op deze pagina.

```
function toggleSound(){
    sound = new Audio(localStorage.getItem("sound"));
    sound.muted = JSON.parse(localStorage.getItem("muted"));
    sound.muted = !sound.muted;
    sound.play();
    localStorage.setItem("sound", src);
    localStorage.setItem("muted", sound.muted);
}

function changeSound(event){
    let key = event.target.value;
    src = sounds[key];
    sound = new Audio(src);
    localStorage.setItem("sound",src);
    sound.muted = !(togglebutton.value);
    sound.play();
}
```

Belangerijke stukken code uitleggen:

Puzzels *mockAPI*

Er moest ook gebruik gemaakt worden van data die aangeleverd wordt door een externe bron. Hiervoor werd mockAPI.io gebruikt. Dat is een simulatie van een echte API, waar zelf de puzzeldata werd aan toegevoegd.

In de “PuzzleGameState” klasse wordt de functie “fetchNewPuzzels()” gebruikt om schaakpuzzels op te halen van de *mockAPI*. Deze functie maakt gebruik van een HTTP-verzoek (fetch()) om een specifieke URL aan te roepen waar de puzzelgegevens beschikbaar zijn.

Het URL-adres dat wordt gebruikt, verwijst naar de *mockAPI* (https://645b63c3a8f9e4d6e767035c.mockapi.io/Puzzels/), gevolgd door een specifiek puzzelnummer dat wordt geselecteerd met behulp van de functie “selectPuzzel()”.

Zodra het HTTP-verzoek is voltooid, wordt er een keten van *promises* gebruikt om de respons te verwerken. Eerst wordt de respons omgezet naar JSON-formaat met behulp van “response.json()”. Vervolgens worden de verkregen puzzelgegevens toegewezen aan de variabele puzzel.

De code gaat verder door de functie “selectFenOfPuzzel()” aan te roepen, waarbij de juiste FEN-notatie (Forsyth-Edwards-notatie) wordt geselecteerd op basis van de verkregen puzzelgegevens. Deze FEN-notatie wordt vervolgens gebruikt om de schaakstukken op te zetten met behulp van “FenConvertor.setupPieces()”. Het spelbord wordt vervolgens getekend met “draw.drawGameboard()” om de gebruiker de schaakpositie te tonen.

Ook de puzzel*rating* wordt bijgehouden en getoond. Op basis daarvan wordt ook een *userrating* berekend en bijgehouden in de *localstorage*. Een puzzel oplossen verhoogt je score. Hoe meer fouten hoe lager je score. Dit wordt weergegeven in een *CanvasJS Chart* in de klasse “PlotElo.js”. Hoe dit precies wordt verwezenlijkt wordt niet verder op ingegaan, omdat dit eerder een minder relevante extra functionaliteit is.

Al deze stappen worden uitgevoerd wanneer “fetchNewPuzzels()” wordt aangeroepen, wat resulteert in het ophalen van een nieuwe schaakpuzzel van de *mockAPI* en het bijwerken van de website-interface met de relevante gegevens van de puzzel.

```
fetchNewPuzzels(){

fetch(`https://645b63c3a8f9e4d6e767035c.mockapi.io/Puzzels/${this.selectPuzzel(
)}`)
        .then((response) => response.json())
        .then((puzzel)=>{
            this.selectFenOfPuzzel(puzzel);
            FenConvertor.setupPieces(this.board,this.currentPuzzel.FEN);
            this.draw.drawGameboard(this.board);
            this.puzzlemove();
            let ratingelement=document.getElementById("puz_rating");
            ratingelement.textContent=`Moeilijkheid puzzel :
${this.currentPuzzel.Rating}`;
            this.possibleScore=parseInt(this.currentPuzzel.Rating)/100;
            this.amountOfMistakes=0;
        })
}
```

Multithreading

Bij het implementeren van de schaakbot werd al snel een probleem geconstateerd, de site bevroor tot wanneer de schaakbot zijn volgende zet had berekend. De oplossing werd eerst gezocht bij het gebruik van asynchrone javascript. A.d.h.v. *promises* en asynchrone functies (*async)* werd dit gedaan, maar al snel viel op dat dit nog steeds de site bevroor en het algoritme vertraagde. Hoewel *promises en callbacks* asynchrone gedrag mogelijk maken, blijven ze nog steeds binnen de beperkingen van het *single-threaded* model werken. Dit betekent dat bij het uitvoeren van extreem intensieve berekeningen in de *hoofdthread*, de gebruikersinterface nog steeds kan bevriezen en ongevoelig worden.

Web workers daarentegen bieden echte parallelle uitvoering door extra *threads* te creëren die naast de *hoofdthread* werken. Hierdoor kunnen zware berekeningen en langlopende taken worden uitgevoerd zonder de hoofdthread te blokkeren. Dit houdt de gebruikersinterface responsief en voorkomt dat de browser als geheel traag wordt.

Het grootte nadeel aan *web workers* is dat ze geen complexe data kunnen delen en niet aan de DOM-elementen van de site kunnen. De Bot klasse heeft geen directe interactie met de DOM-structuur, dus dit vormt geen probleem. Echter, wanneer het gaat om het delen van gegevens, zoals het schaakbord met al zijn stukken, moet er wel een oplossing worden gevonden.

Dit wordt gerealiseerd a.d.h.v. communicatie met JSON-objecten. De *hoofdthread* en de *workerthread* zullen elk apart een eigen schaakbord hebben, en elke zet zal worden gecommuniceerd met het sturen van berichten in JSON-formaat. Hierdoor komt het bord van de ene *thread* ten alle tijden overeen met het bord van de andere *thread*.

Het aanmaken van een *workerthread* wordt gedaan in de klasse “GameStatePlay”:

```
this.bot=new Worker(`${baseURL}/scripts/Model/Bot/Bot.js`, { type: "module" });
```

Er moet worden aangegeven dat de *webworkers* modules importeert anders zal het importeren niet werken. De variabele “baseURL” wordt berekend omdat het gebruik van relatieve adressen niet werkt bij het aanmaken van de *worker*. Vervolgens wordt er direct een eerste instructie gestuurd naar de *thread* in JSON-formaat:

```
let data={//opdracht sturen naar de webworker --> zodat volledig async werkt
    "type":"maakbot", // maak een bot aan
    "color":col,
    "depth":this.botDifficulty
}
this.bot.addEventListener("message",(event)=>{ this.botMove(event)})// zodat ->
bot zijn move telkens kan terugsturen als wij hem data verzenden
this.bot.postMessage(data);
```

De communicatie wordt gedaan door middel van *eventlisteners* en de “postMessage” functie. De webworker ziet er als volgt uit:

```
const board= new Board(true);  //maak een parallel bord aan
let bot;
console.log("Web worker script loaded successfully");

self.addEventListener("message",(event)=>{ //voor als main thread bericht
stuurt
    let data=JSON.parse(event.data);
    let backdata;
    if (data.type==="maakbot"){ //moet er een bot worden gemaakt

        bot= new Bot(data.color,+data.depth, new Evaluation());
        if(data.color){
```

```
            backdata=move(); //bot zet move en geeft de move terug

            self.postMessage(JSON.stringify(backdata));
        }


    }else if(data.type==="move"){
        //als main thread een move maakt is message van type "move"
        board.move(board.board[data.cord1.y][data.cord1.x],data.cord2);
        //move uitvoeren van main thread
        backdata=move(); //bot zet move een geeft move terug

        self.postMessage(JSON.stringify(backdata)); //stuur move ->main thread

    }
});
```

Als het type van het bericht “maakbot” is, wordt er een instantie van de klasse “Bot” aangemaakt. Als deze wit is, wordt er direct een zet berekend en gezet op het bord met de functie “move()”. Deze functie geeft ook een object terug waarin de zet staat beschreven, hiermee kan deze informatie teruggestuurd worden naar de *hoofdthread.* Als de data van de *hoofdthread* van het type “move” is dan wordt deze zet op het schaakbord gedaan. Hierna berekent de schaakbot zijn eigen zet, voert deze zet uit en stuurt een bericht naar de main thread.

Deze code demonstreert hoe de communicatie tussen de meerdere *threads* mogelijk gemaakt wordt, en hoe de site nog steeds vloeiend verloopt wanneer er tegen de schaakbot wordt gespeeld.

Sustainable webdesign

Donkere modus

Er werd een donkere modus toegevoegd. Deze is op dezelfde manier geïmplementeerd als het inklappen en uitklappen van het navigatievenster. Namelijk met een veranderende classname in de HTML-code, die op zijn beurt CSS-code activeert en deactiveert. In CSS zijn namelijk kleurvariabelen gedeclareerd. CSS zal andere kleuren toepassen als het body-element, *classname* “dark”, bevat.

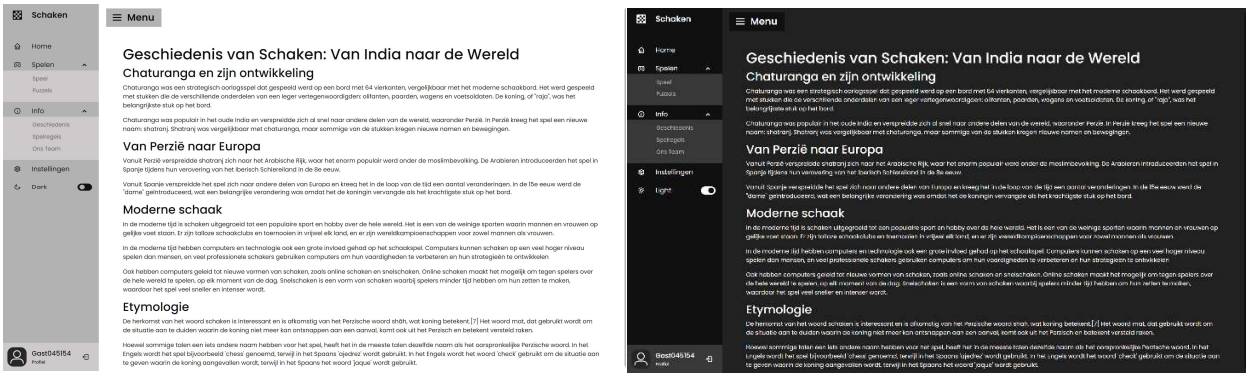
Donkere modus heeft verschillende voordelen, niet alleen op het gebied van duurzaam webdesign, maar ook op andere terreinen.

Door het gebruik van kleuren met een laag contrast en het verminderen van de helderheid van het scherm, zorgt de donkere modus ervoor dat apparaten met beeldschermen minder energie verbruiken. Dit draagt bij aan een lager totaal energieverbruik en bevordert daarmee de energie-efficiëntie, wat een belangrijk aspect is van duurzaamheid. Op mobiele apparaten betekent dit dat de noodzaak van batterijvervanging wordt verminderd.

De verminderde energieconsumptie van de donkere modus resulteert ook in een lagere uitstoot van kooldioxide, aangezien de elektriciteit die nodig is voor het functioneren van apparaten vaak wordt opgewekt uit fossiele brandstoffen.

Een ander voordeel van de donkere modus is dat het de slijtage van schermen kan verminderen. Doordat donkere kleuren minder intens zijn, wordt de activiteit van de pixels verminderd, vooral bij schermtechnologieën zoals OLED, waarbij individuele pixels worden verlicht. Minder pixelactiviteit kan degradatie en veroudering van de pixels vertragen, wat uiteindelijk resulteert in een langere levensduur van het scherm en minder elektronisch afval.

Ten slotte verbetert de donkere modus de leesbaarheid van tekst, vooral in omstandigheden met weinig omgevingslicht. Dit leidt tot minder oogvermoeidheid en een betere gebruikerservaring, waardoor mensen op een duurzame manier langer kunnen genieten van digitale inhoud.



Figuur 26: donkere modus

Modulaire code

In het kader van duurzaam webdesign is ook gebruikgemaakt van modulaire JavaScript-code (*modules*). De structuur van deze modules werd eerder in het verslag al toegelicht bij ‘structuur van het project’. Door het gebruik van deze modules kon *separation of concerns* ook makkelijk worden toegepast.

Dankzij deze manier van werken konden verschillende functionaliteiten van de schaakwebsite geïsoleerd worden in die afzonderlijke modules. Elk van deze modules heeft zijn eigen verantwoordelijkheid voor een specifiek aspect van de website, zoals bijvoorbeeld de schaakstukmodules in de map ‘model’ of de klasse ‘bot’.

Dit bevordert duurzaam webdesign. De modulaire structuur van het project heeft bijgedragen aan de onderhoudbaarheid, herbruikbaarheid en efficiëntie van de code, waardoor een solide basis is gecreëerd voor de verdere ontwikkeling en groei van de schaakwebsite.

Efficiënte code

In de context van duurzaam webdesign is het ook belangrijk om efficiënte code te schrijven die zo min mogelijk resources verbruikt. Hier zijn enkele voorbeelden van hoe onze code efficiënt is in relatie tot duurzaamheid:

Door het gebruikmaken van een abstracte klasse voor de schaakstukken van het model werd code makkelijk hergebruikt. In plaats van functionaliteit te dupliceren voor elk schaakstuk, werden gemeenschappelijke kenmerken in de abstracte klasse geplaatst of geërfd van een ander schaakstuk. Geen *duplicated code* heeft tal van voordelen zoals: een lager energieverbruik tijdens het uitvoeren van de applicatie, het maakt code overzichtelijker, minder grote opslag.

Ook werd het gebruik van trage *reflection functies* vermeden. Een voorbeeld hiervan is “instanceof()”. Deze functie bracht een aanzienlijke *overhead* met zich mee, aangezien het *runtime-analyse* van klassen en objecten vereist. Door het minimaliseren van het gebruik hiervan werden de prestaties van onze code aanzienlijk geoptimaliseerd en de verwerkingstijd verminderd. Dit heeft geleid tot een drastische verbetering van de snelheid van de schaakbot. Minder verwerkingstijd betekent minder energieverbruik, waardoor dit opnieuw leidt tot een duurzamer gebruik van code.

Angular of niet?

Voor dit schaakspel is geen gebruik gemaakt van Angular. In de beginfase van het project is dit wel overwogen, maar uiteindelijk niet gebruikt vanwege meer nadelen dan voordelen. Angular bood de mogelijkheid om alles in componenten te schrijven en vervolgens te communiceren op basis van die componenten. Dit zou het mogelijk hebben gemaakt om delen van de website beter op te splitsen en verbeterd te implementeren, wat een groot voordeel is. Echter, dit bleek het enige voordeel te zijn dat Angular te bieden had. Er waren meerdere nadelen. Een van de nadelen was dat het arbeidsintensiever is, om twee redenen. Ten eerste was er aan het begin van het project nog weinig kennis over Angular, dus het opzoeken van benodigde informatie zou de ontwikkeling van de website vertragen. Ten tweede vereist het programmeren in Angular meer uitgebreide en complexere code.

De communicatie tussen de componenten is opgelost door gebruik te maken van *localstorage*, die beschikbaar was voor alle HTML-pagina's. Dit bood de mogelijkheid om variabelen op te slaan in het hele project. Daarnaast was er al een begin van de code geschreven in HTML en JavaScript. Het converteren van alles naar TypeScript zou veel tijd in beslag nemen.

Conclusie

In dit verslag werd de website uitvoerig behandeld. Eerst werd de structuur van het project besproken, inclusief het schaakmodel van het schaakbord. Schaakstukken werden behandeld, evenals het berekenen van pseudolegale zetten. De implementatie van het schaakbord, de Board klasse, de LegalChecker en de FenConverter werden gedetailleerd beschreven.

Verder werden de al dan niet toegepaste algoritmes besproken. Minimaxalgoritme, Negamaxalgoritme en Alpha-beta pruning werd uitgelegd. De structuur en werking van de botklasse en evaluatieklasse kwam ook aan bod.

Vervolgens werd de implementatie van de Viewcontroller en zijn rol als brug tussen de View en het model besproken. Hoe de verschillende spelmodi werden mogelijk gemaakt, hoe er vloeiend op het canvas werd getekend en hoe het schaakmodel efficiënt gebruikt werd.

Bovendien werden nog enkele complexere stukken code toegelicht namelijk: Puzzels mockAPI, Multithreading, …

Tot slot werd er gereflecteerd over het gebruik van Sustainable webdesign: de implementatie van donkere modus en het belang van modulaire en efficiënte code.

Dit verslag heeft een gedetailleerd inzicht gegeven in de ontwikkeling en implementatie van de schaakwebsite. De besproken onderwerpen, van de schaaklogica en algoritmes tot complexe code-implementaties en verbeteringen, bieden een solide basis voor verdere ontwikkeling en optimalisatie van de schaakwebsite. De opgedane kennis en inzichten kunnen worden toegepast om de gebruikerservaring te verbeteren en het schaakspel op de website verder te verfijnen.

Referentielijst

1. Chess.com. (z.d.). *FEN Chess Notation*. Geraadpleegd op 8 maart 2023, van https://www.chess.com/terms/FEN-chess

2. Chess Programming Wiki. (z.d.). *Simplified Evaluation Function*. Geraadpleegd op 8 maart 2023, van https://www.chessprogramming.org/Simplified_Evaluation_Function

3. Sebastian Lague. (2021, 12 februari). Coding Adventure: Chess [Video]. YouTube. Geraadpleegd op 8 maart 2023, van https://www.youtube.com/watch?v=U4ogK0MIzqk

4. Sustainable Web Design. (z.d.). Geraadpleegd op 8 maart 2023, van https://sustainablewebdesign.org/

5. Wikipedia. (z.d.). *Forsyth-Edwards Notation*. In Wikipedia, De vrije encyclopedie. Geraadpleegd op 8 maart 2023, van https://nl.wikipedia.org/wiki/Forsyth-Edwards_Notation

6. Negamax - Chessprogramming wiki. (z.d.). Geraadpleegd op 15 mei 2023, van https://www.chessprogramming.org/Negamax