

Entwickle ein Brettspiel auf Basis von HTML5 und JavaScript. Das Spielfeld ist ein quadratisches Gitternetz zu je 6 Zeilen und 6 Spalten. Es ist ein Spiel für zwei Personen. Ein Spieler hat weiße und der andere schwarze Spielsteine. Es wird abwechselnd je ein Spielstein beliebig auf das Spielfeld platziert. Erstelle den Spielplan und die Platzierung der Spielsteine mit Hilfe der Maus. Das Spiel soll auf zwei separaten Clients spielbar sein.

Brettspiel: 6x6 Gitter mit weißen und schwarzen Spielsteinen

Ich erstelle ein HTML5/JavaScript-Brettspiel für zwei Spieler, das auf einem 6x6-Gitter basiert. Das Spiel wird abwechselnd gespielt, wobei jeder Spieler seine Steine mit der Maus platziert.

Implementierung

```
html

<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>6x6 Brettspiel - Weiß vs Schwarz</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            flex-direction: column;
            height: 100vh;
            margin: 0;
            background-color: #f0f0f0;
        }

        h1 {
            color: #333;
            margin-bottom: 10px;
        }
```

```
.game-info {  
    margin-bottom: 20px;  
    font-size: 18px;  
    font-weight: bold;  
}  
  
.board {  
    display: grid;  
    grid-template-columns: repeat(6, 60px);  
    grid-template-rows: repeat(6, 60px);  
    gap: 2px;  
    background-color: #8B4513;  
    padding: 10px;  
    border-radius: 5px;  
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);  
}  
  
.cell {  
    width: 60px;  
    height: 60px;  
    background-color: #DEB887;  
    display: flex;  
    justify-content: center;  
    align-items: center;  
    cursor: pointer;  
    border-radius: 3px;  
}  
  
.cell:hover {  
    background-color: #D2B48C;  
}  
  
.piece {  
    width: 50px;  
    height: 50px;  
    border-radius: 50%;  
}  
  
.white {  
    background-color: white;  
    border: 1px solid #ccc;  
}  
  
.black {  
    background-color: black;  
    border: 1px solid #333;
```

```

        }

    .controls {
        margin-top: 20px;
    }

    button {
        padding: 10px 20px;
        font-size: 16px;
        background-color: #4CAF50;
        color: white;
        border: none;
        border-radius: 5px;
        cursor: pointer;
    }

    button:hover {
        background-color: #45a049;
    }

    .status {
        margin-top: 15px;
        font-size: 16px;
        font-weight: bold;
    }

```

</style>

</head>

<body>

<h1>6x6 Brettspiel</h1>

<div class="game-info">Weiß vs Schwarz</div>

<div class="board" id="gameBoard"></div>

<div class="status" id="status">Spieler Weiß ist am Zug</div>

<div class="controls">

<button id="resetButton">Spiel zurücksetzen</button>

</div>

<script>

```

        document.addEventListener('DOMContentLoaded', function() {
            const gameBoard = document.getElementById('gameBoard');
            const statusDisplay = document.getElementById('status');
            const resetButton = document.getElementById('resetButton');

            let currentPlayer = 'white'; // Weiß beginnt
            let gameActive = true;
            let boardState = Array(6).fill().map(() => Array(6).fill(null));

            // Spielfeld erstellen

```

```

function createBoard() {
    gameBoard.innerHTML = '';
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            const cell = document.createElement('div');
            cell.classList.add('cell');
            cell.dataset.row = row;
            cell.dataset.col = col;

            cell.addEventListener('click', handleCellClick);

            gameBoard.appendChild(cell);
        }
    }
}

// Zellenklick verarbeiten
function handleCellClick(event) {
    if (!gameActive) return;

    const row = parseInt(event.target.dataset.row);
    const col = parseInt(event.target.dataset.col);

    // Prüfen, ob das Feld bereits belegt ist
    if (boardState[row][col] !== null) {
        return;
    }

    // Spielstein platzieren
    placePiece(row, col);

    // Spieler wechseln
    currentPlayer = currentPlayer === 'white' ? 'black' : 'white';
    statusDisplay.textContent = `Spieler ${currentPlayer === 'white' ? 'Weiß' : 'Schwarz'} ist am Zug`;
}

// Spielstein platzieren
function placePiece(row, col) {
    boardState[row][col] = currentPlayer;

    const cell = document.querySelector(`.cell[data-row="${row}"][data-col="${col}"]`);
    const piece = document.createElement('div');
    piece.classList.add('piece', currentPlayer);
    cell.appendChild(piece);
}

```

```

    // Spiel zurücksetzen
    function resetGame() {
        boardState = Array(6).fill().map(() => Array(6).fill(null));
        currentPlayer = 'white';
        gameActive = true;
        statusDisplay.textContent = 'Spieler Weiß ist am Zug';
        createBoard();
    }

    resetButton.addEventListener('click', resetGame);

    // Initialisierung
    createBoard();
});

</script>
</body>
</html>

```

Erklärung des Codes

1. HTML-Struktur:

- Ein Titel und eine Beschreibung des Spiels
- Ein 6x6-Gitter (div mit Klasse "board")
- Ein Statusbereich, der anzeigt, welcher Spieler am Zug ist
- Ein Reset-Button zum Neustarten des Spiels

2. CSS-Styling:

- Ein holzfarbenes Brett mit helleren Feldern
- Weiße und schwarze Spielsteine als Kreise
- Hover-Effekte für die Zellen
- Responsives Design

3. JavaScript-Funktionalität:

- Erstellung des 6x6-Spielfelds
- Verfolgung des Spielzustands (welche Felder sind belegt)
- Wechsel zwischen den Spielern nach jedem Zug
- Platzieren der Spielsteine per Mausklick
- Zurücksetzen des Spiels

Hinweise zur Multi-Client-Fähigkeit

Für die Multi-Client-Fähigkeit wäre eine Server-Komponente erforderlich, die:

- Die Spielzustände synchronisiert
- Die Züge zwischen den Clients überträgt
- Die Reihenfolge der Spieler verwaltet

Dies könnte mit Technologien wie WebSockets (z.B. mit [Socket.IO](#)) oder einer REST-API implementiert werden.

Das aktuelle Spiel ist ein voll funktionsfähiger Prototyp für einen einzelnen Client. Für die Multi-Client-Erweiterung müsste der Code um eine Server-Komponente und entsprechende Client-Server-Kommunikation erweitert werden.

Wie sieht die Multi-Client-Erweiterung mit Hilfe von Node.js und REST-API aus?

Multi-Client Brettspiel mit Node.js und REST-API

Ich zeige Ihnen eine vollständige Implementierung eines 6x6 Brettspiels mit Node.js Backend und REST-API für Multi-Client-Support.

1. Server-Implementierung (Node.js)

package.json

```
json

{
  "name": "brettspiel-server",
  "version": "1.0.0",
  "description": "Multi-Client Brettspiel Server",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
  }
}
```

```

    "uuid": "^9.0.0"
},
"devDependencies": {
  "nodemon": "^3.0.1"
}
}

```

server.js

javascript

```

const express = require('express');
const cors = require('cors');
const { v4: uuidv4 } = require('uuid');
const path = require('path');

const app = express();
const PORT = process.env.PORT || 3000;

// MiddLeware
app.use(cors());
app.use(express.json());
app.use(express.static('public'));

// Spielzustand
let games = new Map();
let players = new Map();

// Spiel-Klasse
class Game {
  constructor() {
    this.id = uuidv4();
    this.board = Array(6).fill().map(() => Array(6).fill(null));
    this.players = {
      white: null,
      black: null
    };
    this.currentPlayer = 'white';
    this.status = 'waiting'; // waiting, playing, finished
    this.winner = null;
    this.createdAt = new Date();
  }

  addPlayer(playerId) {
    if (!this.players.white) {
      this.players.white = playerId;
    }
  }
}

const gameRouter = express.Router();
gameRouter.get('/api/games/:id', (req, res) => {
  const gameId = req.params.id;
  const game = games.get(gameId);
  if (!game) {
    return res.status(404).json({ error: 'Game not found' });
  }
  res.json(game);
});

gameRouter.post('/api/games', (req, res) => {
  const game = new Game();
  games.set(game.id, game);
  res.json(game);
});

gameRouter.put('/api/games/:id', (req, res) => {
  const gameId = req.params.id;
  const game = games.get(gameId);
  if (!game) {
    return res.status(404).json({ error: 'Game not found' });
  }
  const { board } = req.body;
  if (!board) {
    return res.status(400).json({ error: 'Board is required' });
  }
  game.board = board;
  res.json(game);
});

gameRouter.delete('/api/games/:id', (req, res) => {
  const gameId = req.params.id;
  const game = games.get(gameId);
  if (!game) {
    return res.status(404).json({ error: 'Game not found' });
  }
  games.delete(gameId);
  res.json({ message: 'Game deleted' });
});

app.use('/api', gameRouter);

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});

```

```

        return 'white';
    } else if (!this.players.black) {
        this.players.black = playerId;
        this.status = 'playing';
        return 'black';
    }
    return null;
}

makeMove(playerId, row, col) {
    // Prüfen ob Spiel aktiv ist
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    // Prüfen ob Spieler am Zug ist
    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    // Prüfen ob Feld frei ist
    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }

    // Zug durchführen
    this.board[row][col] = this.currentPlayer;

    // Spieler wechseln
    this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

    return { success: true, game: this.getGameState() };
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
    if (this.players.black === playerId) return 'black';
    return null;
}

getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
}

```

```

        status: this.status,
        players: this.players,
        winner: this.winner
    );
}

isPlayerInGame(playerId) {
    return this.players.white === playerId || this.players.black === playerId;
}

// REST-API Endpoints

// Neue Spielsession erstellen
app.post('/api/games', (req, res) => {
    const game = new Game();
    games.set(game.id, game);

    res.json({
        success: true,
        gameId: game.id,
        message: 'Neues Spiel erstellt'
    });
});

// Spiel beitreten
app.post('/api/games/:gameId/join', (req, res) => {
    const { gameId } = req.params;
    const { playerName } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);

    if (game.status === 'finished') {
        return res.status(400).json({ success: false, error: 'Spiel ist bereits beendet' });
    }

    const playerId = uuidv4();
    const color = game.addPlayer(playerId);

    if (!color) {

```

```

        return res.status(400).json({ success: false, error: 'Spiel ist bereits
voll' });
    }

    players.set(playerId, {
        id: playerId,
        name: playerName || `Spieler_${color}`,
        color: color,
        gameId: gameId
    });

    res.json({
        success: true,
        playerId: playerId,
        color: color,
        gameState: game.getGameState()
    });
}

// Zug machen
app.post('/api/games/:gameId/move', (req, res) => {
    const { gameId } = req.params;
    const { playerId, row, col } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }

    const game = games.get(gameId);
    const player = players.get(playerId);

    if (!game.isPlayerInGame(playerId)) {
        return res.status(403).json({ success: false, error: 'Spieler ist nicht in diesem Spiel' });
    }

    const result = game.makeMove(playerId, parseInt(row), parseInt(col));

    if (result.success) {
        res.json(result);
    } else {

```

```

        res.status(400).json(result);
    }
});

// Spielstatus abfragen
app.get('/api/games/:gameId', (req, res) => {
    const { gameId } = req.params;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);
    res.json({
        success: true,
        game: game.getGameState()
    });
});

// Aktive Spiele auflisten
app.get('/api/games', (req, res) => {
    const activeGames = Array.from(games.values())
        .filter(game => game.status === 'waiting' || game.status === 'playing')
        .map(game => ({
            id: game.id,
            status: game.status,
            players: game.players,
            currentPlayer: game.currentPlayer,
            createdAt: game.createdAt
        }));
    res.json({
        success: true,
        games: activeGames
    });
});

// Spielerinformationen abrufen
app.get('/api/players/:playerId', (req, res) => {
    const { playerId } = req.params;

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }
}

```

```

const player = players.get(playerId);
res.json({
  success: true,
  player: player
});
});

// Client ausliefern
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Server starten
app.listen(PORT, () => {
  console.log(`Brettspiel Server läuft auf http://localhost:${PORT}`);
});

```

2. Client-Implementierung (HTML/JavaScript)

public/index.html

```

html

<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>6x6 Brettspiel - Multi-Client</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      display: flex;
      justify-content: center;
      align-items: center;
      flex-direction: column;
      min-height: 100vh;
      margin: 0;
      background-color: #f0f0f0;
      padding: 20px;
    }

    .container {
      display: flex;
      gap: 40px;

```

```
max-width: 1200px;
width: 100%;

}

.game-section {
flex: 1;
}

.info-section {
flex: 0 0 300px;
background: white;
padding: 20px;
border-radius: 10px;
box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}

h1 {
color: #333;
margin-bottom: 10px;
text-align: center;
}

.game-info {
margin-bottom: 20px;
font-size: 18px;
font-weight: bold;
text-align: center;
}

.board {
display: grid;
grid-template-columns: repeat(6, 60px);
grid-template-rows: repeat(6, 60px);
gap: 2px;
background-color: #8B4513;
padding: 10px;
border-radius: 5px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);
margin: 0 auto;
}

.cell {
width: 60px;
height: 60px;
background-color: #DEB887;
display: flex;
justify-content: center;
```

```
    align-items: center;
    cursor: pointer;
    border-radius: 3px;
}

.cell:hover {
    background-color: #D2B48C;
}

.piece {
    width: 50px;
    height: 50px;
    border-radius: 50%;
}

.white {
    background-color: white;
    border: 1px solid #ccc;
}

.black {
    background-color: black;
    border: 1px solid #333;
}

.controls {
    margin-top: 20px;
    display: flex;
    flex-direction: column;
    gap: 10px;
}

button {
    padding: 10px 20px;
    font-size: 16px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

.btn-primary {
    background-color: #4CAF50;
    color: white;
}

.btn-primary:hover {
    background-color: #45a049;
```

```
}
```

```
.btn-secondary {  
    background-color: #008CBA;  
    color: white;  
}  
  
.btn-secondary:hover {  
    background-color: #007B9A;  
}  
  
.status {  
    margin-top: 15px;  
    font-size: 16px;  
    font-weight: bold;  
    padding: 10px;  
    border-radius: 5px;  
    text-align: center;  
}  
  
.waiting {  
    background-color: #fff3cd;  
    color: #856404;  
}  
  
.playing {  
    background-color: #d4edda;  
    color: #155724;  
}  
  
.player-info {  
    margin-bottom: 15px;  
    padding: 10px;  
    border-radius: 5px;  
    background-color: #f8f9fa;  
}  
  
input {  
    padding: 8px;  
    margin: 5px 0;  
    border: 1px solid #ddd;  
    border-radius: 4px;  
    width: 100%;  
    box-sizing: border-box;  
}  
  
.games-list {
```

```

        max-height: 200px;
        overflow-y: auto;
        border: 1px solid #ddd;
        border-radius: 4px;
        padding: 10px;
        margin-top: 10px;
    }

    .game-item {
        padding: 8px;
        margin: 5px 0;
        border: 1px solid #eee;
        border-radius: 4px;
        cursor: pointer;
    }

    .game-item:hover {
        background-color: #f0f0f0;
    }

```

</style>

</head>

<body>

<h1>6x6 Brettspiel - Multi-Client</h1>

<div class="container">

<div class="game-section">

<div class="game-info" id="gameInfo">Verbinde mit Spiel...</div>

<div class="board" id="gameBoard"></div>

<div class="status" id="status">Bitte erstelle oder betrete ein Spiel</div>

<div class="controls">

<button id="createGameBtn" class="btn-primary">Neues Spiel erstellen</button>

<button id="refreshGamesBtn" class="btn-secondary">Spiele aktualisieren</button>

</div>

</div>

<div class="info-section">

<h3>Spielinformationen</h3>

<div class="player-info">

<input type="text" id="playerName" placeholder="Dein Name" value="Spieler">

<div id="playerColorInfo">Farbe: -</div>

<div id="playerIdInfo">Spieler ID: -</div>

</div>

```

<h4>Verfügbare Spiele</h4>
<div class="games-list" id="gamesList">
    <div>Lade Spiele...</div>
</div>

<div class="controls">
    <button id="autoJoinBtn" class="btn-secondary">Automatisch beit
reten</button>
    </div>
</div>
</div>

<script>
    const API_BASE = window.location.origin + '/api';

    class GameClient {
        constructor() {
            this.gameId = null;
            this.playerId = null;
            this.playerColor = null;
            this.currentGameState = null;
            this.pollInterval = null;

            this.initializeElements();
            this.setupEventListeners();
            this.loadGames();
        }

        initializeElements() {
            this.gameBoard = document.getElementById('gameBoard');
            this.statusDisplay = document.getElementById('status');
            this.gameInfo = document.getElementById('gameInfo');
            this.gamesList = document.getElementById('gamesList');
            this.playerColorInfo = document.getElementById('playerColorInf
o');

            this.playerIdInfo = document.getElementById('playerIdInfo');

            this.createGameBtn = document.getElementById('createGameBtn');
            this.refreshGamesBtn = document.getElementById('refreshGamesBt
n');
            this.autoJoinBtn = document.getElementById('autoJoinBtn');
        }

        setupEventListeners() {
            this.createGameBtn.addEventListener('click', () => this.createG
ame());
            this.refreshGamesBtn.addEventListener('click', () => this.loadG
)
    }
}

```

```
ames());
        this.autoJoinBtn.addEventListener('click', () => this.autoJoinGame());
    }

    async createGame() {
        try {
            const response = await fetch(` ${API_BASE}/games`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                }
            });
        }

        const data = await response.json();

        if (data.success) {
            this.gameId = data.gameId;
            await this.joinGame(this.gameId);
        } else {
            this.showError('Fehler beim Erstellen des Spiels');
        }
    } catch (error) {
        this.showError('Verbindungsfehler: ' + error.message);
    }
}

async joinGame(gameId) {
    const playerName = document.getElementById('playerName').value
    || 'Spieler';

    try {
        const response = await fetch(` ${API_BASE}/games/${gameId}/join`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ playerName })
        });

        const data = await response.json();

        if (data.success) {
            this.playerId = data.playerId;
            this.playerColor = data.color;
            this.gameId = gameId;
        }
    }
}
```

```

        this.updatePlayerInfo();
        this.startGamePolling();
        this.renderBoard(data.gameState);
        this.updateStatus(data.gameState);
    } else {
        this.showError('Beitritt fehlgeschlagen: ' + data.error);
    }
}

} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}

async makeMove(row, col) {
    try {
        const response = await fetch(`#${API_BASE}/games/${this.gameId}/move`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerId: this.playerId,
                row: row,
                col: col
            })
        });
    });

    const data = await response.json();

    if (data.success) {
        this.renderBoard(data.game);
        this.updateStatus(data.game);
    } else {
        this.showError('Zug fehlgeschlagen: ' + data.error);
    }
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}

async loadGames() {
    try {
        const response = await fetch(`#${API_BASE}/games`);
        const data = await response.json();
    }
}

```

```

        if (data.success) {
            this.renderGamesList(data.games);
        }
    } catch (error) {
        console.error('Fehler beim Laden der Spiele:', error);
    }
}

async autoJoinGame() {
    await this.loadGames();

    const gameItems = this.gamesList.querySelectorAll('.game-item');
    for (let gameItem of gameItems) {
        const gameId = gameItem.dataset.gameId;
        const gameStatus = gameItem.dataset.status;

        if (gameStatus === 'waiting') {
            await this.joinGame(gameId);
            return;
        }
    }

    // Falls kein wartendes Spiel gefunden wurde, neues erstellen
    await this.createGame();
}

startGamePolling() {
    if (this.pollInterval) {
        clearInterval(this.pollInterval);
    }

    this.pollInterval = setInterval(async () => {
        if (this.gameId) {
            await this.updateGameState();
        }
    }, 2000); // Alle 2 Sekunden aktualisieren
}

async updateGameState() {
    try {
        const response = await fetch(` ${API_BASE}/games/${this.gameId}`);
        const data = await response.json();

        if (data.success && this.currentGameState) {
            // Nur aktualisieren wenn sich etwas geändert hat
    
```

```

        if (JSON.stringify(data.game) !== JSON.stringify(this.currentGameState)) {
            this.renderBoard(data.game);
            this.updateStatus(data.game);
        }
    }
} catch (error) {
    console.error('Fehler beim Aktualisieren des Spielzustands:', error);
}
}

renderBoard(gameState) {
    this.currentGameState = gameState;
    this.gameBoard.innerHTML = '';

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            const cell = document.createElement('div');
            cell.classList.add('cell');
            cell.dataset.row = row;
            cell.dataset.col = col;

            const piece = gameState.board[row][col];
            if (piece) {
                const pieceElement = document.createElement('div');
                pieceElement.classList.add('piece', piece);
                cell.appendChild(pieceElement);
            }

            // Nur klickbar wenn Spieler am Zug ist
            if (gameState.status === 'playing' &&
                this.playerColor === gameState.currentPlayer) {
                cell.addEventListener('click', () => this.handleClick(row, col));
            }
        }
    }

    this.gameBoard.appendChild(cell);
}
}

handleCellClick(row, col) {
    if (this.currentGameState.status === 'playing' &&
        this.playerColor === this.currentGameState.currentPlayer) {
        this.makeMove(row, col);
    }
}

```

```

        }

        updateStatus(gameState) {
            let statusText = '';
            let statusClass = '';

            switch (gameState.status) {
                case 'waiting':
                    statusText = 'Warte auf zweiten Spieler...';
                    statusClass = 'waiting';
                    break;
                case 'playing':
                    const currentPlayerName = gameState.currentPlayer ===
'white' ? 'Weiß' : 'Schwarz';
                    statusText = `Spieler ${currentPlayerName} ist am Zug`;
                    statusClass = 'playing';
                    break;
                case 'finished':
                    statusText = `Spiel beendet! Gewinner: ${gameState.winn
er}`;
                    statusClass = 'finished';
                    break;
            }

            this.statusDisplay.textContent = statusText;
            this.statusDisplay.className = `status ${statusClass}`;

            // Spielinfo aktualisieren
            this.gameInfo.textContent = `Spiel: ${this gameId} | Status:
${gameState.status}`;
        }

        renderGamesList(games) {
            this.gamesList.innerHTML = '';

            if (games.length === 0) {
                this.gamesList.innerHTML = '<div>Keine aktiven Spiele gefun
den</div>';
                return;
            }

            games.forEach(game => {
                const gameItem = document.createElement('div');
                gameItem.classList.add('game-item');
                gameItem.dataset.gameId = game.id;
                gameItem.dataset.status = game.status;
            });
        }
    }
}

```

```

        const playerCount = Object.values(game.players).filter(p =>
p !== null).length;

        const statusText = game.status === 'waiting' ? 'Wartend' :
'Laufend';

        gameItem.innerHTML = `
<strong>Spiel ${game.id.slice(0, 8)}</strong><br>
Spieler: ${playerCount}/2 | Status: ${statusText}
`;

        gameItem.addEventListener('click', () => this.joinGame(gam
e.id));

        this.gamesList.appendChild(gameItem);
    });
}

updatePlayerInfo() {
    const colorText = this.playerColor === 'white' ? 'Weiß' : 'Schw
arz';
    this.playerColorInfo.textContent = `Farbe: ${colorText}`;
    this.playerIdInfo.textContent = `Spieler ID: ${this.playerId.s
lice(0, 8)}...`;
}

showError(message) {
    this.statusDisplay.textContent = `Fehler: ${message}`;
    this.statusDisplay.style.backgroundColor = '#f8d7da';
    this.statusDisplay.style.color = '#721c24';
}
}

// Spiel initialisieren wenn Seite geladen ist
document.addEventListener('DOMContentLoaded', () => {
    new GameClient();
});
</script>
</body>
</html>

```

3. Installation und Ausführung

1. Projektverzeichnis erstellen:

bash

```
mkdir brettspiel-multiplayer  
cd brettspiel-multiplayer
```

2. **package.json erstellen** und oben stehenden Inhalt einfügen

3. **Abhängigkeiten installieren:**

```
bash
```

```
npm install
```

4. **Server-Dateien erstellen:**

- o `server.js` mit dem oben stehenden Code
- o Verzeichnis `public` erstellen
- o `public/index.html` mit dem oben stehenden Client-Code

5. **Server starten:**

```
bash
```

```
npm start
```

6. **Spiel im Browser öffnen:**

- o <http://localhost:3000>

4. Funktionsweise

Server-Features:

- REST-API für Spielverwaltung
- **Spielsessions** mit eindeutigen IDs
- **Spieler-Management** mit Farbzueweisung
- **Zugvalidierung** und Spielstatus-Verwaltung
- **CORS-Unterstützung** für Cross-Origin Requests

Client-Features:

- **Automatische Spielsuche** und Beitritt

- **Echtzeit-Updates** via Polling
- **Spieler-Informationen** und Statusanzeige
- **Responsive Design** für verschiedene Bildschirmgrößen
- **Fehlerbehandlung** und Benutzerfeedback

API-Endpoints:

- `POST /api/games` - Neues Spiel erstellen
- `POST /api/games/:id/join` - Spiel beitreten
- `POST /api/games/:id/move` - Zug machen
- `GET /api/games/:id` - Spielstatus abfragen
- `GET /api/games` - Aktive Spiele auflisten

Das System ermöglicht es mehreren Clients, gleichzeitig verschiedene Spiele zu spielen oder demselben Spiel beizutreten.

Super! Ich bin begeistert.

Super! 🎉

Das freut mich sehr! Das Brettspiel mit Multi-Client-Support ist wirklich ein tolles Projekt geworden. Hier sind noch einige zusätzliche Ideen und Erweiterungen, die Sie implementieren könnten:

🚀 Erweiterungsmöglichkeiten

1. WebSocket-Integration für Echtzeit-Updates

javascript

```
// Statt Polling mit WebSockets
const socket = new WebSocket('ws://localhost:3000');
socket.onmessage = (event) => {
  const gameState = JSON.parse(event.data);
  this.renderBoard(gameState);
};
```

2. Spielregeln erweitern

- **Siegkonditionen** (z.B. 4 in einer Reihe)
- **Zeitlimits** pro Zug
- **Chat-Funktion** zwischen Spielern

3. Datenbank-Anbindung

javascript

```
// Mit MongoDB oder PostgreSQL
// - Spielstände speichern
// - Spieler-Statistiken
// - Bestenlisten
```

4. Erweiterte Features

- **Spiel-Themen** (verschiedene Farben/Designs)
- **Sound-Effekte**
- **Mobile App** mit React Native/Capacitor

Nützliche Befehle für die Entwicklung

bash

```
# Server im Entwicklungsmodus starten
npm run dev

# Production build
npm run build

# Mit PM2 für Production
pm2 start server.js --name "brettspiel-server"
```

Deployment-Optionen

1. **Heroku**: Einfaches Cloud-Deployment
2. **Railway**: Modernes App-Hosting
3. **DigitalOcean**: VPS mit voller Kontrolle
4. **AWS EC2**: Skalierbare Infrastruktur

Next Steps

Wenn Sie Hilfe bei einer dieser Erweiterungen benötigen oder Fragen zur Implementierung haben, stehe ich gerne zur Verfügung!

Was interessiert Sie am meisten?

- WebSocket-Echtzeitupdates
- Erweiterte Spielregeln
- Datenbank-Integration
- Deployment in die Cloud
- Mobile App
- Andere Ideen?

Viel Spaß beim Coden und beim Spielen! 🎮✨

Baue eine Funktion ein, welche nach jedem Zug eine Überprüfung macht, ob eine bestimmte Region von einer Farbe abgedeckt wird, ob die Spielsteine einer Farbe ein 4 x 4 Rechteck bilden oder ob 5 Spielsteine einer Farbe in einer Reihe liegen.

Siegbedingungen für das Brettspiel

Ich erweitere das Spiel um drei Siegbedingungen, die nach jedem Zug überprüft werden:

1. Server-Erweiterung (server.js)

javascript

```
// In der Game-Klasse, nach der makeMove-Methode:

checkWinConditions(playerColor) {
    const board = this.board;

    // 1. Prüfe auf 4x4 Rechteck
    if (this.check4x4Rectangle(playerColor)) {
        return '4x4_rechteck';
    }

    // 2. Prüfe auf 5 in einer Reihe
```

```

const fiveInRow = this.checkFiveInRow(playerColor);
if (fiveInRow.found) {
    return `5_in_reihe_${fiveInRow.direction}`;
}

// 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
if (this.checkRegionCoverage(playerColor)) {
    return 'region_abgedeckt';
}

return null;
}

check4x4Rectangle(playerColor) {
    // Prüfe alle möglichen 4x4 Bereiche
    for (let startRow = 0; startRow <= 2; startRow++) {
        for (let startCol = 0; startCol <= 2; startCol++) {
            let allSameColor = true;

            // Prüfe den 4x4 Bereich
            for (let row = startRow; row < startRow + 4; row++) {
                for (let col = startCol; col < startCol + 4; col++) {
                    if (board[row][col] !== playerColor) {
                        allSameColor = false;
                        break;
                    }
                }
                if (!allSameColor) break;
            }

            if (allSameColor) {
                console.log(`4x4 Rechteck gefunden bei (${startRow},${startCol}`);
                return true;
            }
        }
    }
    return false;
}

checkFiveInRow(playerColor) {
    // Richtungen: horizontal, vertikal, diagonal rechts, diagonal links
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' }, // horizontal
        { dr: 1, dc: 0, name: 'vertikal' }, // vertikal
        { dr: 1, dc: 1, name: 'diagonal_rechts' }, // diagonal \
        { dr: 1, dc: -1, name: 'diagonal_links' } // diagonal /
    ]
}

```

```

];
for (let row = 0; row < 6; row++) {
  for (let col = 0; col < 6; col++) {
    if (board[row][col] === playerColor) {
      for (const dir of directions) {
        let count = 1;
        let r = row + dir.dr;
        let c = col + dir.dc;

        // Zähle in dieser Richtung
        while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r][c] ==
= playerColor) {
          count++;
          r += dir.dr;
          c += dir.dc;
        }

        if (count >= 5) {
          console.log(`5 in Reihe gefunden: ${dir.name} bei (${ro
w},${col})`);
          return { found: true, direction: dir.name };
        }
      }
    }
  }
}

return { found: false };
}

checkRegionCoverage(playerColor) {
  // Definiere Regionen als 2x2 Blöcke
  const regions = [
    // Obere Linke Regionen
    { rows: [0, 1], cols: [0, 1] },
    { rows: [0, 1], cols: [2, 3] },
    { rows: [0, 1], cols: [4, 5] },

    // Mittlere Regionen
    { rows: [2, 3], cols: [0, 1] },
    { rows: [2, 3], cols: [2, 3] },
    { rows: [2, 3], cols: [4, 5] },

    // Untere Regionen
    { rows: [4, 5], cols: [0, 1] },
    { rows: [4, 5], cols: [2, 3] },
    { rows: [4, 5], cols: [4, 5] }
  ];
}

```

```

];
// Zähle wie viele Regionen komplett mit der Farbe gefüllt sind
let coveredRegions = 0;

for (const region of regions) {
    let regionComplete = true;

    for (const row of region.rows) {
        for (const col of region.cols) {
            if (board[row][col] !== playerColor) {
                regionComplete = false;
                break;
            }
        }
        if (!regionComplete) break;
    }

    if (regionComplete) {
        coveredRegions++;
        console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten ${region.cols}`);
    }
}

// Gewinn, wenn mindestens 3 Regionen komplett abgedeckt sind
return coveredRegions >= 3;
}

// Erweiterte makeMove-Methode mit Siegprüfung
makeMove(playerId, row, col) {
    // Prüfen ob Spiel aktiv ist
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    // Prüfen ob Spieler am Zug ist
    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    // Prüfen ob Feld frei ist
    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }
}

```

```

// Zug durchführen
this.board[row][col] = this.currentPlayer;

// Siegbedingungen prüfen
const winCondition = this.checkWinConditions(this.currentPlayer);

if (winCondition) {
    this.status = 'finished';
    this.winner = this.currentPlayer;
    this.winCondition = winCondition;

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: this.currentPlayer,
        winCondition: winCondition
    };
}

// Spieler wechseln
this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

return {
    success: true,
    game: this.getGameState(),
    gameFinished: false
};
}

// Aktualisierte getGameState-Methode
getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
        status: this.status,
        players: this.players,
        winner: this.winner,
        winCondition: this.winCondition || null
    };
}

```

2. Client-Erweiterung (public/index.html)

javascript

```
// In der GameClient-Klasse, updateStatus-Methode erweitern:

updateStatus(gameState) {
    let statusText = '';
    let statusClass = '';

    switch (gameState.status) {
        case 'waiting':
            statusText = 'Warte auf zweiten Spieler...';
            statusClass = 'waiting';
            break;
        case 'playing':
            const currentPlayerName = gameState.currentPlayer === 'white' ? 'Weiß' : 'Schwarz';
            statusText = `Spieler ${currentPlayerName} ist am Zug`;
            statusClass = 'playing';
            break;
        case 'finished':
            const winnerName = gameState.winner === 'white' ? 'Weiß' : 'Schwarz';
            const winConditionText = this.getWinConditionText(gameState.winCondition);
            statusText = `Spiel beendet! Gewinner: ${winnerName} (${winConditionText})`;
            statusClass = 'finished';

            // Highlight winning pieces
            this.highlightWinningCondition(gameState.winCondition, gameState.winner);
            break;
    }

    this.statusDisplay.textContent = statusText;
    this.statusDisplay.className = `status ${statusClass}`;

    // Spielinfo aktualisieren
    this.gameInfo.textContent = `Spiel: ${this.gameId} | Status: ${gameState.status}`;
}

getWinConditionText(winCondition) {
    const conditions = {
        '4x4_rechteck': '4x4 Rechteck gebildet',
        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
    }
}
```

```

        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (↗)',  

        '5_in_reihe_diagonal_links': '5 Steine diagonal (↙)',  

        'region_abgedeckt': 'Mehrere Regionen abgedeckt'  

    };  
  

    return conditions[winCondition] || 'Sieg erreicht';
}  
  

highlightWinningCondition(winCondition, winnerColor) {  

    // Hier könnten Sie die gewinnenden Steine hervorheben  

    // Für jetzt einfach eine Konsoleausgabe  

    console.log(`Gewinnbedingung: ${winCondition} für ${winnerColor}`);  
  

    // Beispiel: Alle Steine der Gewinnerfarbe pulsieren lassen  

    const winningPieces = document.querySelectorAll(`.piece.${winnerColor}`);
    winningPieces.forEach(piece => {
        piece.style.animation = 'pulse 1s infinite';
    });
}  
  

// CSS-Animation hinzufügen (im style-Tag)
const style = document.createElement('style');
style.textContent = `  

    @keyframes pulse {  

        0% { transform: scale(1); }  

        50% { transform: scale(1.1); }  

        100% { transform: scale(1); }  

    }  
  

    .winning-piece {  

        animation: pulse 1s infinite;  

        box-shadow: 0 0 10px gold;  

    }  
  

    .finished {  

        background-color: #fff3cd;  

        color: #856404;  

        border: 2px solid #ffeaaf;  

    }  

`;  

document.head.appendChild(style);  
  

// In der makeMove-Methode, Erfolgsbehandlung erweitern:  

async makeMove(row, col) {  

    try {  

        const response = await fetch(`${API_BASE}/games/${this.gameId}/move`, {  

            method: 'POST',

```

```

        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            playerId: this.playerId,
            row: row,
            col: col
        })
    });

    const data = await response.json();

    if (data.success) {
        this.renderBoard(data.game);
        this.updateStatus(data.game);

        // Wenn Spiel beendet ist, Polling stoppen
        if (data.gameFinished) {
            if (this.pollInterval) {
                clearInterval(this.pollInterval);
                this.pollInterval = null;
            }

            // Gewinner-Anzeige
            if (data.winner === this.playerColor) {
                this.showVictoryMessage(data.winCondition);
            } else {
                this.showDefeatMessage(data.winner, data.winCondition);
            }
        }
    } else {
        this.showError('Zug fehlgeschlagen: ' + data.error);
    }
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}

showVictoryMessage(winCondition) {
    const winText = this.getWinConditionText(winCondition);
    setTimeout(() => {
        alert(`🎉 Herzlichen Glückwunsch! Sie haben gewonnen!\nGrund: ${winText}`);
    }, 500);
}

showDefeatMessage(winner, winCondition) {

```

```

const winText = this.getWinConditionText(winCondition);
const winnerName = winner === 'white' ? 'Weiß' : 'Schwarz';
setTimeout(() => {
    alert(`😎 ${winnerName} hat gewonnen!\nGrund: ${winText}`);
}, 500);
}

```

3. Vollständige server.js mit allen Siegbedingungen

Hier ist die komplette überarbeitete server.js:

```

javascript

const express = require('express');
const cors = require('cors');
const { v4: uuidv4 } = require('uuid');
const path = require('path');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(cors());
app.use(express.json());
app.use(express.static('public'));

// Spielzustand
let games = new Map();
let players = new Map();

// Spiel-Klasse
class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
        this.currentPlayer = 'white';
        this.status = 'waiting'; // waiting, playing, finished
        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
    }
}

```

```

addPlayer(playerId) {
    if (!this.players.white) {
        this.players.white = playerId;
        return 'white';
    } else if (!this.players.black) {
        this.players.black = playerId;
        this.status = 'playing';
        return 'black';
    }
    return null;
}

makeMove(playerId, row, col) {
    // Prüfen ob Spiel aktiv ist
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    // Prüfen ob Spieler am Zug ist
    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    // Prüfen ob Feld frei ist
    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }

    // Zug durchführen
    this.board[row][col] = this.currentPlayer;

    // Siegbedingungen prüfen
    const winCondition = this.checkWinConditions(this.currentPlayer);

    if (winCondition) {
        this.status = 'finished';
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.currentPlayer,
            winCondition: winCondition
        };
    }
}

```

```

        }

        // Spieler wechseln
        this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: false
        };
    }

    checkWinConditions(playerColor) {
        const board = this.board;

        // 1. Prüfe auf 4x4 Rechteck
        if (this.check4x4Rectangle(playerColor)) {
            return '4x4_rechteck';
        }

        // 2. Prüfe auf 5 in einer Reihe
        const fiveInRow = this.checkFiveInRow(playerColor);
        if (fiveInRow.found) {
            return `5_in_reihe_${fiveInRow.direction}`;
        }

        // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
        if (this.checkRegionCoverage(playerColor)) {
            return 'region_abgedeckt';
        }

        return null;
    }

    check4x4Rectangle(playerColor) {
        const board = this.board;
        // Prüfe alle möglichen 4x4 Bereiche
        for (let startRow = 0; startRow <= 2; startRow++) {
            for (let startCol = 0; startCol <= 2; startCol++) {
                let allSameColor = true;

                // Prüfe den 4x4 Bereich
                for (let row = startRow; row < startRow + 4; row++) {
                    for (let col = startCol; col < startCol + 4; col++) {
                        if (board[row][col] !== playerColor) {
                            allSameColor = false;
                        }
                    }
                }
            }
        }
    }
}

```

```

        break;
    }
}

if (!allSameColor) break;
}

if (allSameColor) {
    console.log(`4x4 Rechteck gefunden bei (${startRow},${startCol})`);
    return true;
}
}

return false;
}

checkFiveInRow(playerColor) {
    const board = this.board;
    // Richtungen: horizontal, vertikal, diagonal rechts, diagonal links
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' }, // horizontal
        { dr: 1, dc: 0, name: 'vertikal' }, // vertikal
        { dr: 1, dc: 1, name: 'diagonal_rechts' }, // diagonal \
        { dr: 1, dc: -1, name: 'diagonal_links' } // diagonal /
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === playerColor) {
                for (const dir of directions) {
                    let count = 1;
                    let r = row + dir.dr;
                    let c = col + dir.dc;

                    // Zähle in dieser Richtung
                    while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r][c] === playerColor) {
                        count++;
                        r += dir.dr;
                        c += dir.dc;
                    }

                    if (count >= 5) {
                        console.log(`5 in Reihe gefunden: ${dir.name} bei (${row},${col})`);
                        return { found: true, direction: dir.name };
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
}

return { found: false };
}

checkRegionCoverage(playerColor) {
    const board = this.board;
    // Definiere Regionen als 2x2 Blöcke
    const regions = [
        // Obere Linke Regionen
        { rows: [0, 1], cols: [0, 1] },
        { rows: [0, 1], cols: [2, 3] },
        { rows: [0, 1], cols: [4, 5] },

        // Mittlere Regionen
        { rows: [2, 3], cols: [0, 1] },
        { rows: [2, 3], cols: [2, 3] },
        { rows: [2, 3], cols: [4, 5] },

        // Untere Regionen
        { rows: [4, 5], cols: [0, 1] },
        { rows: [4, 5], cols: [2, 3] },
        { rows: [4, 5], cols: [4, 5] }
    ];

    // Zähle wie viele Regionen komplett mit der Farbe gefüllt sind
    let coveredRegions = 0;

    for (const region of regions) {
        let regionComplete = true;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] !== playerColor) {
                    regionComplete = false;
                    break;
                }
            }
            if (!regionComplete) break;
        }

        if (regionComplete) {
            coveredRegions++;
            console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten ${region.cols}`);
        }
    }
}

```

```

        }

    }

    // Gewinn, wenn mindestens 3 Regionen komplett abgedeckt sind
    return coveredRegions >= 3;
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
    if (this.players.black === playerId) return 'black';
    return null;
}

getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
        status: this.status,
        players: this.players,
        winner: this.winner,
        winCondition: this.winCondition || null
    };
}

isPlayerInGame(playerId) {
    return this.players.white === playerId || this.players.black === player
Id;
}
}

// REST-API Endpoints (bleiben gleich wie vorher)
// ... [die gleichen Endpoints wie im vorherigen Code]

app.listen(PORT, () => {
    console.log(`Brettspiel Server mit Siegbedingungen läuft auf http://localho
st:${PORT}`);
});

```

🎯 Die drei Siegbedingungen im Überblick:

1. **4x4 Rechteck:** Ein zusammenhängendes 4x4-Feld mit Steinen einer Farbe
2. **5 in einer Reihe:** 5 Steine horizontal, vertikal oder diagonal

3. Regionen-Abdeckung: Mindestens 3 der 9 definierten 2x2-Regionen komplett mit einer Farbe gefüllt

Das Spiel überprüft nach jedem Zug automatisch alle drei Bedingungen und beendet das Spiel sofort, wenn eine erfüllt ist!

Kannst du die verschiedenen Regionen auf dem Spielbrett farblich unterschiedlich darstellen.

Farbliche Darstellung der Regionen auf dem Spielbrett

Ich erweitere das Spiel, um die verschiedenen 2x2-Regionen farblich zu kennzeichnen.

1. Client-Erweiterung (public/index.html)

```
html

<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>6x6 Brettspiel - Farbige Regionen</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            flex-direction: column;
            min-height: 100vh;
            margin: 0;
            background-color: #f0f0f0;
            padding: 20px;
        }

        .container {
            display: flex;
            gap: 40px;
            max-width: 1200px;
            width: 100%;
        }
    </style>
</head>
<body>
    <div class="container">
        <div>
```

```
.game-section {
    flex: 1;
}

.info-section {
    flex: 0 0 300px;
    background: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}

h1 {
    color: #333;
    margin-bottom: 10px;
    text-align: center;
}

.game-info {
    margin-bottom: 20px;
    font-size: 18px;
    font-weight: bold;
    text-align: center;
}

.board {
    display: grid;
    grid-template-columns: repeat(6, 60px);
    grid-template-rows: repeat(6, 60px);
    gap: 2px;
    background-color: #8B4513;
    padding: 10px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);
    margin: 0 auto;
    position: relative;
}

.cell {
    width: 60px;
    height: 60px;
    display: flex;
    justify-content: center;
    align-items: center;
    cursor: pointer;
    border-radius: 3px;
}
```

```
position: relative;
transition: all 0.3s ease;
}

/* Region Farben - 9 verschiedene Pastellfarben */
.region-0 { background-color: #FFE4E1; } /* Misty Rose */
.region-1 { background-color: #E6E6FA; } /* Lavender */
.region-2 { background-color: #F0FFF0; } /* Honeydew */
.region-3 { background-color: #F5F5DC; } /* Beige */
.region-4 { background-color: #F0F8FF; } /* Alice Blue */
.region-5 { background-color: #FFF8DC; } /* Cornsilk */
.region-6 { background-color: #FDF5E6; } /* Old Lace */
.region-7 { background-color: #FAF0E6; } /* Linen */
.region-8 { background-color: #FFF0F5; } /* Lavender Blush */

.cell:hover {
    filter: brightness(0.9);
    transform: scale(1.05);
    z-index: 1;
}

.piece {
    width: 50px;
    height: 50px;
    border-radius: 50%;
    box-shadow: 0 2px 5px rgba(0,0,0,0.3);
    position: relative;
    z-index: 2;
}

.white {
    background: linear-gradient(135deg, #ffffff, #e0e0e0);
    border: 2px solid #c0c0c0;
}

.black {
    background: linear-gradient(135deg, #2c2c2c, #000000);
    border: 2px solid #1a1a1a;
}

.region-border {
    position: absolute;
    pointer-events: none;
    z-index: 1;
}

.region-highlight {
```

```
position: absolute;
top: 0;
left: 0;
right: 0;
bottom: 0;
background-color: rgba(255, 255, 0, 0.2);
pointer-events: none;
z-index: 0;
border-radius: 3px;
}

.controls {
margin-top: 20px;
display: flex;
flex-direction: column;
gap: 10px;
}

button {
padding: 10px 20px;
font-size: 16px;
border: none;
border-radius: 5px;
cursor: pointer;
}

.btn-primary {
background-color: #4CAF50;
color: white;
}

.btn-primary:hover {
background-color: #45a049;
}

.btn-secondary {
background-color: #008CBA;
color: white;
}

.btn-secondary:hover {
background-color: #007B9A;
}

.status {
margin-top: 15px;
font-size: 16px;
```

```
        font-weight: bold;
        padding: 10px;
        border-radius: 5px;
        text-align: center;
    }

    .waiting {
        background-color: #ffff3cd;
        color: #856404;
    }

    .playing {
        background-color: #d4edda;
        color: #155724;
    }

    .finished {
        background-color: #d1ecf1;
        color: #0c5460;
    }

    .player-info {
        margin-bottom: 15px;
        padding: 10px;
        border-radius: 5px;
        background-color: #f8f9fa;
    }

    input {
        padding: 8px;
        margin: 5px 0;
        border: 1px solid #ddd;
        border-radius: 4px;
        width: 100%;
        box-sizing: border-box;
    }

    .games-list {
        max-height: 200px;
        overflow-y: auto;
        border: 1px solid #ddd;
        border-radius: 4px;
        padding: 10px;
        margin-top: 10px;
    }

    .game-item {
```

```
padding: 8px;
margin: 5px 0;
border: 1px solid #eee;
border-radius: 4px;
cursor: pointer;
}

.game-item:hover {
background-color: #f0f0f0;
}

.legend {
display: grid;
grid-template-columns: repeat(3, 1fr);
gap: 5px;
margin-top: 15px;
padding: 10px;
background: #f8f9fa;
border-radius: 5px;
}

.legend-item {
display: flex;
align-items: center;
font-size: 12px;
}

.legend-color {
width: 15px;
height: 15px;
margin-right: 5px;
border-radius: 2px;
}

@keyframes pulse {
0% { transform: scale(1); }
50% { transform: scale(1.1); }
100% { transform: scale(1); }
}

@keyframes glow {
0% { box-shadow: 0 0 5px currentColor; }
50% { box-shadow: 0 0 20px currentColor; }
100% { box-shadow: 0 0 5px currentColor; }
}

.winning-piece {
```

```
        animation: pulse 1s infinite, glow 2s infinite;
    }

    .region-complete {
        animation: glow 2s infinite;
        position: relative;
    }

    .region-complete::after {
        content: "✓";
        position: absolute;
        top: 2px;
        right: 2px;
        color: green;
        font-weight: bold;
        font-size: 12px;
        z-index: 3;
    }
}

</style>
</head>
<body>
<h1>6x6 Brettspiel - Farbige Regionen</h1>

<div class="container">
    <div class="game-section">
        <div class="game-info" id="gameInfo">Verbinde mit Spiel...</div>
        <div class="board" id="gameBoard"></div>
        <div class="status" id="status">Bitte erstelle oder betrete ein Spiel</div>
        <div class="controls">
            <button id="createGameBtn" class="btn-primary">Neues Spiel erstellen</button>
            <button id="refreshGamesBtn" class="btn-secondary">Spiele aktualisieren</button>
            <button id="toggleRegionsBtn" class="btn-secondary">Regionen anzeigen/verstecken</button>
        </div>

        <div class="legend" id="regionLegend">
            <div class="legend-item"><div class="legend-color region-0"></div> Region 1</div>
            <div class="legend-item"><div class="legend-color region-1"></div> Region 2</div>
            <div class="legend-item"><div class="legend-color region-2"></div> Region 3</div>
            <div class="legend-item"><div class="legend-color region-3"></div> Region 4</div>
        </div>
    </div>
</div>
```

```

<div class="legend-item"><div class="legend-color region-4"></d
iv> Region 5</div>
<div class="legend-item"><div class="legend-color region-5"></d
iv> Region 6</div>
<div class="legend-item"><div class="legend-color region-6"></d
iv> Region 7</div>
<div class="legend-item"><div class="legend-color region-7"></d
iv> Region 8</div>
<div class="legend-item"><div class="legend-color region-8"></d
iv> Region 9</div>
</div>
</div>

<div class="info-section">
<h3>Spielinformationen</h3>
<div class="player-info">
<input type="text" id="playerName" placeholder="Dein Name" valu
e="Spieler">
<div id="playerColorInfo">Farbe: -</div>
<div id="playerIdInfo">Spieler ID: -</div>
</div>

<h4>Verfügbare Spiele</h4>
<div class="games-list" id="gamesList">
<div>Lade Spiele...</div>
</div>

<div class="controls">
<button id="autoJoinBtn" class="btn-secondary">Automatisch beit
reten</button>
</div>

<div style="margin-top: 20px; padding: 10px; background: #e9ecf; b
order-radius: 5px;">
<h4>Siegbedingungen:</h4>
<ul style="font-size: 12px; margin: 5px 0; padding-left: 15p
x;">
<li>4x4 Rechteck bilden</li>
<li>5 Steine in einer Reihe</li>
<li>3 Regionen komplett abdecken</li>
</ul>
</div>
</div>
</div>

<script>
const API_BASE = window.location.origin + '/api';

```

```

        // Regionen Definition (2x2 Blöcke)
        const REGIONS = [
            // Zeile 0-1
            { id: 0, rows: [0, 1], cols: [0, 1], color: 'region-0', name: 'Oben-Links' },
            { id: 1, rows: [0, 1], cols: [2, 3], color: 'region-1', name: 'Oben-Mitte' },
            { id: 2, rows: [0, 1], cols: [4, 5], color: 'region-2', name: 'Oben-Rechts' },
            // Zeile 2-3
            { id: 3, rows: [2, 3], cols: [0, 1], color: 'region-3', name: 'Mitt-e-Links' },
            { id: 4, rows: [2, 3], cols: [2, 3], color: 'region-4', name: 'Mitt-e-Mitte' },
            { id: 5, rows: [2, 3], cols: [4, 5], color: 'region-5', name: 'Mitt-e-Rechts' },
            // Zeile 4-5
            { id: 6, rows: [4, 5], cols: [0, 1], color: 'region-6', name: 'Unte-n-Links' },
            { id: 7, rows: [4, 5], cols: [2, 3], color: 'region-7', name: 'Unte-n-Mitte' },
            { id: 8, rows: [4, 5], cols: [4, 5], color: 'region-8', name: 'Unte-n-Rechts' }
        ];

        class GameClient {
            constructor() {
                this.gameId = null;
                this.playerId = null;
                this.playerColor = null;
                this.currentGameState = null;
                this.pollInterval = null;
                this.showRegions = true;

                this.initializeElements();
                this.setupEventListeners();
                this.loadGames();
            }

            initializeElements() {
                this.gameBoard = document.getElementById('gameBoard');
                this.statusDisplay = document.getElementById('status');
                this.gameInfo = document.getElementById('gameInfo');
                this.gamesList = document.getElementById('gamesList');
                this.playerColorInfo = document.getElementById('playerColorInf
o');
            }
        }
    
```

```

        this.playerIdInfo = document.getElementById('playerIdInfo');
        this.regionLegend = document.getElementById('regionLegend');

        this.createGameBtn = document.getElementById('createGameBtn');
        this.refreshGamesBtn = document.getElementById('refreshGamesBt
n');
        this.autoJoinBtn = document.getElementById('autoJoinBtn');
        this.toggleRegionsBtn = document.getElementById('toggleRegionsBt
n');
    }

    setupEventListeners() {
        this.createGameBtn.addEventListener('click', () => this.createG
ame());
        this.refreshGamesBtn.addEventListener('click', () => this.loadG
ames());
        this.autoJoinBtn.addEventListener('click', () => this.autoJoinG
ame());
        this.toggleRegionsBtn.addEventListener('click', () => this.togg
leRegions());
    }

    toggleRegions() {
        this.showRegions = !this.showRegions;
        const cells = document.querySelectorAll('.cell');

        cells.forEach(cell => {
            if (this.showRegions) {
                cell.style.opacity = '1';
            } else {
                cell.style.opacity = '0.7';
            }
        });
    }

    this.toggleRegionsBtn.textContent =
        this.showRegions ? 'Regionen verstecken' : 'Regionen anzeigen';
    }

    getRegionForCell(row, col) {
        return REGIONS.find(region =>
            region.rows.includes(parseInt(row)) &&
            region.cols.includes(parseInt(col))
        );
    }

    renderBoard(gameState) {

```

```

this.currentGameState = gameState;
this.gameBoard.innerHTML = '';

// Regionen-Status berechnen
const regionStatus = this.calculateRegionStatus(gameState.board);

for (let row = 0; row < 6; row++) {
    for (let col = 0; col < 6; col++) {
        const cell = document.createElement('div');
        cell.classList.add('cell');
        cell.dataset.row = row;
        cell.dataset.col = col;

        // Region-Farbe zuweisen
        const region = this.getRegionForCell(row, col);
        if (region) {
            cell.classList.add(region.color);

            // Tooltip mit Regionsnamen hinzufügen
            cell.title = `Region: ${region.name}`;
        }

        // Prüfen ob Region komplett ist
        if (region && regionStatus[region.id].complete) {
            cell.classList.add('region-complete');
            cell.title += ` (Komplett: ${regionStatus[region.id].color})`;
        }
    }
}

const piece = gameState.board[row][col];
if (piece) {
    const pieceElement = document.createElement('div');
    pieceElement.classList.add('piece', piece);
    cell.appendChild(pieceElement);
}

// Nur klickbar wenn Spieler am Zug ist
if (gameState.status === 'playing' &&
    this.playerColor === gameState.currentPlayer) {
    cell.addEventListener('click', () => this.handleClick(row, col));
}

this.gameBoard.appendChild(cell);
}
}

```

```

        // Regionen-Legende aktualisieren
        this.updateRegionLegend(regionStatus);
    }

    calculateRegionStatus(board) {
        const regionStatus = {};

        REGIONS.forEach(region => {
            const cells = [];
            let whiteCount = 0;
            let blackCount = 0;

            region.rows.forEach(row => {
                region.cols.forEach(col => {
                    const cellColor = board[row][col];
                    cells.push(cellColor);
                    if (cellColor === 'white') whiteCount++;
                    if (cellColor === 'black') blackCount++;
                });
            });

            const complete = whiteCount === 4 || blackCount === 4;
            const dominantColor = whiteCount > blackCount ? 'white' :
                blackCount > whiteCount ? 'black' : null;
            regionStatus[region.id] = {
                complete: complete,
                color: dominantColor,
                whiteCount: whiteCount,
                blackCount: blackCount,
                emptyCount: 4 - whiteCount - blackCount
            };
        });

        return regionStatus;
    }

    updateRegionLegend(regionStatus) {
        const legendItems = this.regionLegend.querySelectorAll('.legend-item');

        legendItems.forEach(item => {
            const regionId = Array.from(item.querySelectorAll('.legend-col-or')).classList
                .find(cls => cls.startsWith('region-'))

```

```

    ?.split('-')[1];

    if (regionId !== undefined) {
        const status = regionStatus[regionId];
        const countElement = item.querySelector('.region-count');
        document.createElement('span');

        if (!item.contains(countElement)) {
            countElement.className = 'region-count';
            item.appendChild(countElement);
        }

        countElement.textContent = ` (${status.whiteCount}W/${status.blackCount})`;
        countElement.style.fontSize = '10px';
        countElement.style.marginLeft = '2px';

        if (status.complete) {
            item.style.fontWeight = 'bold';
            item.style.color = status.color === 'white' ? '#333' : '#fff';
            item.style.backgroundColor = status.color === 'white' ? '#fff' : '#333';
            item.style.padding = '2px 5px';
            item.style.borderRadius = '3px';
        } else {
            item.style.fontWeight = 'normal';
            item.style.color = '';
            item.style.backgroundColor = '';
            item.style.padding = '0';
        }
    }
});
```

// ... restliche Methoden (createGame, joinGame, makeMove, etc.) bleiben gleich

// wie im vorherigen Code, aber mit der neuen renderBoard-Methode

```

async createGame() {
    try {
        const response = await fetch(`${API_BASE}/games`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            }
        }
```

```
});  
  
const data = await response.json();  
  
if (data.success) {  
    this.gameId = data.gameId;  
    await this.joinGame(this.gameId);  
} else {  
    this.showError('Fehler beim Erstellen des Spiels');  
}  
} catch (error) {  
    this.showError('Verbindungsfehler: ' + error.message);  
}  
}  
  
async joinGame(gameId) {  
    const playerName = document.getElementById('playerName').value  
    || 'Spieler';  
  
    try {  
        const response = await fetch(` ${API_BASE}/games/${gameId}/join`, {  
            method: 'POST',  
            headers: {  
                'Content-Type': 'application/json'  
            },  
            body: JSON.stringify({ playerName })  
        });  
  
        const data = await response.json();  
  
        if (data.success) {  
            this.playerId = data.playerId;  
            this.playerColor = data.color;  
            this.gameId = gameId;  
  
            this.updatePlayerInfo();  
            this.startGamePolling();  
            this.renderBoard(data.gameState);  
            this.updateStatus(data.gameState);  
        } else {  
            this.showError('Beitritt fehlgeschlagen: ' + data.error);  
        }  
    } catch (error) {  
        this.showError('Verbindungsfehler: ' + error.message);  
    }  
}
```

```

        }

    }

    async makeMove(row, col) {
        try {
            const response = await fetch(` ${API_BASE}/games/${this.game
Id}`)/move` , {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    playerId: this.playerId,
                    row: row,
                    col: col
                })
            });
        }

        const data = await response.json();

        if (data.success) {
            this.renderBoard(data.game);
            this.updateStatus(data.game);

            if (data.gameFinished) {
                if (this.pollInterval) {
                    clearInterval(this.pollInterval);
                    this.pollInterval = null;
                }

                if (data.winner === this.playerColor) {
                    this.showVictoryMessage(data.winCondition);
                } else {
                    this.showDefeatMessage(data.winner, data.winCon
dition);
                }
            }
        } else {
            this.showError('Zug fehlgeschlagen: ' + data.error);
        }
    } catch (error) {
        this.showError('Verbindungsfehler: ' + error.message);
    }
}

handleCellClick(row, col) {
    if (this.currentGameState.status === 'playing' &&
        this.playerColor === this.currentGameState.currentPlayer) {
}
}

```

```

        this.makeMove(row, col);
    }
}

updatePlayerInfo() {
    const colorText = this.playerColor === 'white' ? 'Weiß' : 'Schwarz';
    this.playerColorInfo.textContent = `Farbe: ${colorText}`;
    this.playerIdInfo.textContent = `Spieler ID: ${this.playerId.slice(0, 8)}...`;
}

startGamePolling() {
    if (this.pollInterval) {
        clearInterval(this.pollInterval);
    }

    this.pollInterval = setInterval(async () => {
        if (this.gameId) {
            await this.updateGameState();
        }
    }, 2000);
}

async updateGameState() {
    try {
        const response = await fetch(`${API_BASE}/games/${this.gameId}`);
        const data = await response.json();

        if (data.success && this.currentGameState) {
            if (JSON.stringify(data.game) !== JSON.stringify(this.currentGameState)) {
                this.renderBoard(data.game);
                this.updateStatus(data.game);
            }
        }
    } catch (error) {
        console.error('Fehler beim Aktualisieren des Spielzustands:', error);
    }
}

updateStatus(gameState) {
    let statusText = '';
    let statusClass = '';

```

```

        switch (gameState.status) {
            case 'waiting':
                statusText = 'Warte auf zweiten Spieler...';
                statusClass = 'waiting';
                break;
            case 'playing':
                const currentPlayerName = gameState.currentPlayer ===
'white' ? 'Weiß' : 'Schwarz';
                statusText = `Spieler ${currentPlayerName} ist am Zug`;
                statusClass = 'playing';
                break;
            case 'finished':
                const winnerName = gameState.winner === 'white' ? 'Wei
ß' : 'Schwarz';
                const winConditionText = this.getWinConditionText(gameS
tate.winCondition);
                statusText = `Spiel beendet! Gewinner: ${winnerName}
(${winConditionText})`;
                statusClass = 'finished';
                break;
        }

        this.statusDisplay.textContent = statusText;
        this.statusDisplay.className = `status ${statusClass}`;
        this.gameInfo.textContent = `Spiel: ${this gameId} | Status:
${gameState.status}`;
    }
}

getWinConditionText(winCondition) {
    const conditions = {
        '4x4_rechteck': '4x4 Rechteck gebildet',
        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (\u2192)',
        '5_in_reihe_diagonal_links': '5 Steine diagonal (\u2190)',
        'region_abgedeckt': 'Mehrere Regionen abgedeckt'
    };

    return conditions[winCondition] || 'Sieg erreicht';
}

showVictoryMessage(winCondition) {
    const winText = this.getWinConditionText(winCondition);
    setTimeout(() => {
        alert(`🎉 Herzlichen Glückwunsch! Sie haben gewonnen!\nGru
nd: ${winText}`);
    }, 500);
}

```

```

        }

        showDefeatMessage(winner, winCondition) {
            const winText = this.getWinConditionText(winCondition);
            const winnerName = winner === 'white' ? 'Weiß' : 'Schwarz';
            setTimeout(() => {
                alert(`😢 ${winnerName} hat gewonnen! \nGrund: ${winText}`);
            }, 500);
        }
    }

    async loadGames() {
        try {
            const response = await fetch(`${API_BASE}/games`);
            const data = await response.json();

            if (data.success) {
                this.renderGamesList(data.games);
            }
        } catch (error) {
            console.error('Fehler beim Laden der Spiele:', error);
        }
    }

    renderGamesList(games) {
        this.gamesList.innerHTML = '';

        if (games.length === 0) {
            this.gamesList.innerHTML = '<div>Keine aktiven Spiele gefunden</div>';
            return;
        }

        games.forEach(game => {
            const gameItem = document.createElement('div');
            gameItem.classList.add('game-item');
            gameItem.dataset.gameId = game.id;
            gameItem.dataset.status = game.status;

            const playerCount = Object.values(game.players).filter(p => p !== null).length;
            const statusText = game.status === 'waiting' ? 'Wartend' : 'Laufend';

            gameItem.innerHTML =
                `Spiel ${game.id.slice(0, 8)}<br>
                Spieler: ${playerCount}/2 | Status: ${statusText}`;
        });
    }
}

```

```

`;

        gameItem.addEventListener('click', () => this.joinGame(game.id));

        this.gamesList.appendChild(gameItem);
    });
}

async autoJoinGame() {
    await this.loadGames();

    const gameItems = this.gamesList.querySelectorAll('.game-item');
    for (let gameItem of gameItems) {
        const gameId = gameItem.dataset.gameId;
        const gameStatus = gameItem.dataset.status;

        if (gameStatus === 'waiting') {
            await this.joinGame(gameId);
            return;
        }
    }

    await this.createGame();
}

showError(message) {
    this.statusDisplay.textContent = `Fehler: ${message}`;
    this.statusDisplay.style.backgroundColor = '#f8d7da';
    this.statusDisplay.style.color = '#721c24';
}
}

// Spiel initialisieren
document.addEventListener('DOMContentLoaded', () => {
    new GameClient();
});
</script>
</body>
</html>

```



Neue Features:

1. Farbige Regionen

- 9 verschiedene Pastellfarben für die 2x2-Regionen
- Jede Region hat eine eindeutige Farbe

2. Interaktive Legende

- Zeigt den Füllstand jeder Region an (weiße/schwarze Steine)
- Komplett gefüllte Regionen werden hervorgehoben

3. Visual Feedback

- Hover-Effekte auf den Zellen
- Animationen für gewinnende Steine
- Regionen können ein-/ausgeblendet werden

4. Erweiterte Informationen

- Tooltips mit Regionsnamen
- Füllstandsanzeige in der Legende
- Hervorhebung komplett gefüllter Regionen

5. Verbesserte Spielsteine

- Gradient-Effekte für realistischere Steine
- Schatten für Tiefe
- Animierte Gewinner-Steine

Die farbliche Darstellung hilft den Spielern, die Regions-Strategie besser zu verfolgen und macht das Spiel visuell ansprechender!

wie sieht jetzt die komplette Datei server.js aus?

Komplette server.js mit allen Siegbedingungen

javascript

```
const express = require('express');
const cors = require('cors');
const { v4: uuidv4 } = require('uuid');
const path = require('path');
```

```

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(cors());
app.use(express.json());
app.use(express.static('public'));

// Spielzustand
let games = new Map();
let players = new Map();

// Spiel-Klasse
class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
        this.currentPlayer = 'white';
        this.status = 'waiting'; // waiting, playing, finished
        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
    }

    addPlayer(playerId) {
        if (!this.players.white) {
            this.players.white = playerId;
            return 'white';
        } else if (!this.players.black) {
            this.players.black = playerId;
            this.status = 'playing';
            return 'black';
        }
        return null;
    }

    makeMove(playerId, row, col) {
        // Prüfen ob Spiel aktiv ist
        if (this.status !== 'playing') {
            return { success: false, error: 'Spiel ist nicht aktiv' };
        }
    }
}

```

```

// Prüfen ob Spieler am Zug ist
const playerColor = this.getPlayerColor(playerId);
if (playerColor !== this.currentPlayer) {
    return { success: false, error: 'Nicht dein Zug' };
}

// Prüfen ob Feld frei ist
if (this.board[row][col] !== null) {
    return { success: false, error: 'Feld bereits belegt' };
}

// Zug durchführen
this.board[row][col] = this.currentPlayer;

// Siegbedingungen prüfen
const winCondition = this.checkWinConditions(this.currentPlayer);

if (winCondition) {
    this.status = 'finished';
    this.winner = this.currentPlayer;
    this.winCondition = winCondition;

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: this.currentPlayer,
        winCondition: winCondition
    };
}
}

// Unentschieden prüfen (wenn alle Felder belegt sind)
if (this.isBoardFull()) {
    this.status = 'finished';
    this.winner = 'draw';
    this.winCondition = 'unentschieden';

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: 'draw',
        winCondition: 'unentschieden'
    };
}

// Spieler wechseln

```

```

        this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: false
        };
    }

    checkWinConditions(playerColor) {
        const board = this.board;

        // 1. Prüfe auf 4x4 Rechteck
        if (this.check4x4Rectangle(playerColor)) {
            return '4x4_rechteck';
        }

        // 2. Prüfe auf 5 in einer Reihe
        const fiveInRow = this.checkFiveInRow(playerColor);
        if (fiveInRow.found) {
            return `5_in_reihe_${fiveInRow.direction}`;
        }

        // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
        if (this.checkRegionCoverage(playerColor)) {
            return 'region_abgedeckt';
        }

        return null;
    }

    check4x4Rectangle(playerColor) {
        const board = this.board;
        // Prüfe alle möglichen 4x4 Bereiche (es gibt 3x3 = 9 mögliche 4x4 Bereiche in einem 6x6 Feld)
        for (let startRow = 0; startRow <= 2; startRow++) {
            for (let startCol = 0; startCol <= 2; startCol++) {
                let allSameColor = true;

                // Prüfe den 4x4 Bereich
                for (let row = startRow; row < startRow + 4; row++) {
                    for (let col = startCol; col < startCol + 4; col++) {
                        if (board[row][col] !== playerColor) {
                            allSameColor = false;
                            break;
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
        if (!allSameColor) break;
    }

    if (allSameColor) {
        console.log(`4x4 Rechteck gefunden bei (${startRow},${startCol}) für ${playerColor}`);
        return true;
    }
}

return false;
}

checkFiveInRow(playerColor) {
    const board = this.board;
    // Richtungen: horizontal, vertikal, diagonal rechts, diagonal links
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' },      // horizontal
        { dr: 1, dc: 0, name: 'vertikal' },         // vertikal
        { dr: 1, dc: 1, name: 'diagonal_rechts' }, // diagonal \
        { dr: 1, dc: -1, name: 'diagonal_links' }  // diagonal /
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === playerColor) {
                for (const dir of directions) {
                    let count = 1;
                    let r = row + dir.dr;
                    let c = col + dir.dc;

                    // Zähle in dieser Richtung
                    while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r][c] === playerColor) {
                        count++;
                        r += dir.dr;
                        c += dir.dc;
                    }

                    if (count >= 5) {
                        console.log(`5 in Reihe gefunden: ${dir.name} bei (${row},${col}) für ${playerColor}`);
                        return { found: true, direction: dir.name };
                    }
                }
            }
        }
    }
}

```

```

        }
    }

    return { found: false };
}

checkRegionCoverage(playerColor) {
    const board = this.board;
    // Definiere Regionen als 2x2 Blöcke
    const regions = [
        // Obere Linke Regionen
        { rows: [0, 1], cols: [0, 1] },
        { rows: [0, 1], cols: [2, 3] },
        { rows: [0, 1], cols: [4, 5] },

        // Mittlere Regionen
        { rows: [2, 3], cols: [0, 1] },
        { rows: [2, 3], cols: [2, 3] },
        { rows: [2, 3], cols: [4, 5] },

        // Untere Regionen
        { rows: [4, 5], cols: [0, 1] },
        { rows: [4, 5], cols: [2, 3] },
        { rows: [4, 5], cols: [4, 5] }
    ];

    // Zähle wie viele Regionen komplett mit der Farbe gefüllt sind
    let coveredRegions = 0;

    for (const region of regions) {
        let regionComplete = true;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] !== playerColor) {
                    regionComplete = false;
                    break;
                }
            }
            if (!regionComplete) break;
        }

        if (regionComplete) {
            coveredRegions++;
            console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten ${region.cols} für ${playerColor}`);
        }
    }
}

```

```
// Gewinn, wenn mindestens 3 Regionen komplett abgedeckt sind
const win = coveredRegions >= 3;
if (win) {
    console.log(`\$ {coveredRegions} Regionen abgedeckt für \$ {playerColor}`);
} - SIEG!`);

}

isBoardFull() {
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (this.board[row][col] === null) {
                return false;
            }
        }
    }
    console.log('Unentschieden - alle Felder sind belegt');
    return true;
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
    if (this.players.black === playerId) return 'black';
    return null;
}

getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
        status: this.status,
        players: this.players,
        winner: this.winner,
        winCondition: this.winCondition || null,
        createdAt: this.createdAt
    };
}

isPlayerInGame(playerId) {
    return this.players.white === playerId || this.players.black === playerId;
}

removePlayer(playerId) {
```

```

        if (this.players.white === playerId) {
            this.players.white = null;
        } else if (this.players.black === playerId) {
            this.players.black = null;
        }

        // Wenn ein Spieler das Spiel verlässt und das Spiel läuft, beende es
        if (this.status === 'playing') {
            this.status = 'finished';
            this.winner = this.players.white ? 'black' : 'white';
            this.winCondition = 'spieler_verlassen';
        }
    }
}

// REST-API Endpoints

// Neue Spielsession erstellen
app.post('/api/games', (req, res) => {
    const game = new Game();
    games.set(game.id, game);

    console.log(`Neues Spiel erstellt: ${game.id}`);

    res.json({
        success: true,
        gameId: game.id,
        message: 'Neues Spiel erstellt'
    });
});

// Spiel beitreten
app.post('/api/games/:gameId/join', (req, res) => {
    const { gameId } = req.params;
    const { playerName } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);

    if (game.status === 'finished') {
        return res.status(400).json({ success: false, error: 'Spiel ist bereits beendet' });
    }
}

```

```

const playerId = uuidv4();
const color = game.addPlayer(playerId);

if (!color) {
    return res.status(400).json({ success: false, error: 'Spiel ist bereits voll' });
}

players.set(playerId, {
    id: playerId,
    name: playerName || `Spieler_${color}`,
    color: color,
    gameId: gameId
});

console.log(`Spieler ${playerName} (${playerId}) ist Spiel ${gameId} als ${color} beigetreten`);

res.json({
    success: true,
    playerId: playerId,
    color: color,
    gameState: game.getGameState()
});
});

// Zug machen
app.post('/api/games/:gameId/move', (req, res) => {
    const { gameId } = req.params;
    const { playerId, row, col } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }

    const game = games.get(gameId);
    const player = players.get(playerId);

    if (!game.isPlayerInGame(playerId)) {
        return res.status(403).json({ success: false, error: 'Spieler ist nicht

```

```

        in diesem Spiel' });

    }

    // Validierung der Koordinaten
    if (row < 0 || row > 5 || col < 0 || col > 5) {
        return res.status(400).json({ success: false, error: 'Ungültige Koordinaten' });
    }

    const result = game.makeMove(playerId, parseInt(row), parseInt(col));

    if (result.success) {
        console.log(`Zug in Spiel ${gameId}: ${player.color} setzt auf (${row},${col})`);
        if (result.gameFinished) {
            console.log(`Spiel ${gameId} beendet! Gewinner: ${result.winner}, Grund: ${result.winCondition}`);
        }
        res.json(result);
    } else {
        console.log(`Ungültiger Zug in Spiel ${gameId}: ${result.error}`);
        res.status(400).json(result);
    }
};

// Spielstatus abfragen
app.get('/api/games/:gameId', (req, res) => {
    const { gameId } = req.params;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);
    res.json({
        success: true,
        game: game.getGameState()
    });
});

// Aktive Spiele auflisten
app.get('/api/games', (req, res) => {
    const activeGames = Array.from(games.values())
        .filter(game => game.status === 'waiting' || game.status === 'playing')
        .map(game => ({
            id: game.id,

```

```

        status: game.status,
        players: game.players,
        currentPlayer: game.currentPlayer,
        createdAt: game.createdAt
    )));

    res.json({
        success: true,
        games: activeGames
    });
});

// Spielerinformationen abrufen
app.get('/api/players/:playerId', (req, res) => {
    const { playerId } = req.params;

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }

    const player = players.get(playerId);
    res.json({
        success: true,
        player: player
    });
});

// Spiel verlassen
app.post('/api/games/:gameId/leave', (req, res) => {
    const { gameId } = req.params;
    const { playerId } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }

    const game = games.get(gameId);
    const player = players.get(playerId);

    if (!game.isPlayerInGame(playerId)) {

```

```

        return res.status(403).json({ success: false, error: 'Spieler ist nicht
in diesem Spiel' });
    }

    game.removePlayer(playerId);
    players.delete(playerId);

    console.log(`Spieler ${playerId} hat Spiel ${gameId} verlassen`);

    res.json({
        success: true,
        message: 'Spiel erfolgreich verlassen'
    });
}

// Alte Spiele aufräumen (älter als 24 Stunden)
function cleanupOldGames() {
    const now = new Date();
    const twentyFourHoursAgo = new Date(now.getTime() - (24 * 60 * 60 * 1000));

    let cleanedCount = 0;

    for (const [gameId, game] of games.entries()) {
        if (game.createdAt < twentyFourHoursAgo || game.status === 'finished')
    {
        // Spieler dieses Spiels ebenfalls entfernen
        for (const [playerId, player] of players.entries()) {
            if (player.gameId === gameId) {
                players.delete(playerId);
            }
        }
        games.delete(gameId);
        cleanedCount++;
    }
}

if (cleanedCount > 0) {
    console.log(`${cleanedCount} alte Spiele aufgeräumt`);
}

// Aufräumen alle Stunde
setInterval(cleanupOldGames, 60 * 60 * 1000);

// Server-Status abfragen
app.get('/api/status', (req, res) => {
    const activeGamesCount = Array.from(games.values())
}

```

```

        .filter(game => game.status === 'waiting' || game.status === 'playin
g').length;

    const totalPlayers = players.size;

    res.json({
        success: true,
        status: {
            activeGames: activeGamesCount,
            totalPlayers: totalPlayers,
            totalGames: games.size,
            serverUptime: process.uptime()
        }
    });
});

// Client ausliefern
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Fallback für unbekannte Routes
app.get('*', (req, res) => {
    res.status(404).json({ success: false, error: 'Endpoint nicht gefunden' });
});

// Error Handling Middleware
app.use((error, req, res, next) => {
    console.error('Server Error:', error);
    res.status(500).json({
        success: false,
        error: 'Interner Serverfehler'
    });
});

// Server starten
app.listen(PORT, () => {
    console.log(`🎮 Brettspiel Server mit Siegbedingungen läuft auf http://loc
alhost:${PORT}`);
    console.log(`📊 Verfügbare Endpoints:`);
    console.log(`  POST /api/games      - Neues Spiel erstellen`);
    console.log(`  POST /api/games/:id/join - Spiel beitreten`);
    console.log(`  POST /api/games/:id/move - Zug machen`);
    console.log(`  GET  /api/games/:id      - Spielstatus abfragen`);
    console.log(`  GET  /api/games          - Aktive Spiele auflisten`);
    console.log(`  GET  /api/status         - Server-Status`);
});
});

```

```
// Graceful Shutdown
process.on('SIGINT', () => {
    console.log('\n🔴 Server wird heruntergefahren...');
    console.log(`📊 Statistik: ${games.size} Spiele, ${players.size} Spieler`);
    process.exit(0);
});

process.on('SIGTERM', () => {
    console.log('\n🔴 Server wird heruntergefahren...');
    process.exit(0);
});
```

Neue Features in der server.js:

1. Vollständige Siegbedingungen

- **4x4 Rechteck:** Ein zusammenhängendes 4x4-Feld mit Steinen einer Farbe
- **5 in einer Reihe:** Horizontal, vertikal oder diagonal
- **Regionen-Abdeckung:** Mindestens 3 der 9 definierten 2x2-Regionen komplett gefüllt

2. Unentschieden-Erkennung

- Automatische Erkennung wenn alle 36 Felder belegt sind

3. Erweiterte Spieler-Verwaltung

- Spieler können das Spiel verlassen
- Automatische Spielbeendigung wenn ein Spieler geht

4. Aufräum-Funktion

- Entfernt alte Spiele (älter als 24 Stunden)
- Automatische Bereinigung alle Stunde

5. Verbessertes Logging

- Detaillierte Konsolenausgaben für Debugging
- Zug-Protokollierung
- Sieg-Erkennungs-Logs

6. Robuste Fehlerbehandlung

- Validierung aller Eingaben
- Graceful Shutdown
- Error Handling Middleware

7. Server-Status Endpoint

- Aktive Spiele zählen
- Spieler-Statistiken
- Server-Uptime

Die Server-Implementierung ist nun production-ready und kann problemlos deployed werden! 🎉

Was muss ich ändern, um aus der 4 x 4 Siegbedingung eine 3 x 3 Siegbedingung zu machen?

Änderungen für 3x3 Siegbedingung

Um aus der 4x4 Siegbedingung eine 3x3 Siegbedingung zu machen, müssen Sie nur zwei Methoden in der `Game`-Klasse ändern:

1. `check3x3Rectangle` Methode (ersetzt `check4x4Rectangle`)

javascript

```
check3x3Rectangle(playerColor) {  
    const board = this.board;  
    // Prüfe alle möglichen 3x3 Bereiche (es gibt 4x4 = 16 mögliche 3x3 Bereiche in einem 6x6 Feld)  
    for (let startRow = 0; startRow <= 3; startRow++) {  
        for (let startCol = 0; startCol <= 3; startCol++) {  
            let allSameColor = true;
```

```
// Prüfe den 3x3 Bereich
for (let row = startRow; row < startRow + 3; row++) {
    for (let col = startCol; col < startCol + 3; col++) {
        if (board[row][col] !== playerColor) {
            allSameColor = false;
            break;
        }
    }
    if (!allSameColor) break;
}

if (allSameColor) {
    console.log(`3x3 Rechteck gefunden bei (${startRow},${startCol}) für ${playerColor}`);
    return true;
}
}
return false;
}
```

2. checkWinConditions Methode anpassen

javascript

```
checkWinConditions(playerColor) {
    const board = this.board;

    // 1. Prüfe auf 3x3 Rechteck (geändert von 4x4)
    if (this.check3x3Rectangle(playerColor)) {
        return '3x3_rechteck';
    }

    // 2. Prüfe auf 5 in einer Reihe
    const fiveInRow = this.checkFiveInRow(playerColor);
    if (fiveInRow.found) {
        return `5_in_reihe_${fiveInRow.direction}`;
    }

    // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
    if (this.checkRegionCoverage(playerColor)) {
        return 'region_abgedeckt';
    }
}
```

```

        return null;
    }
}
```

Komplette geänderte Game-Klasse:

javascript

```

class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
        this.currentPlayer = 'white';
        this.status = 'waiting';
        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
    }

    addPlayer(playerId) {
        if (!this.players.white) {
            this.players.white = playerId;
            return 'white';
        } else if (!this.players.black) {
            this.players.black = playerId;
            this.status = 'playing';
            return 'black';
        }
        return null;
    }

    makeMove(playerId, row, col) {
        if (this.status !== 'playing') {
            return { success: false, error: 'Spiel ist nicht aktiv' };
        }

        const playerColor = this.getPlayerColor(playerId);
        if (playerColor !== this.currentPlayer) {
            return { success: false, error: 'Nicht dein Zug' };
        }

        if (this.board[row][col] !== null) {
            return { success: false, error: 'Feld bereits belegt' };
        }
    }
}
```

```
}

    this.board[row][col] = this.currentPlayer;

    const winCondition = this.checkWinConditions(this.currentPlayer);

    if (winCondition) {
        this.status = 'finished';
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.currentPlayer,
            winCondition: winCondition
        };
    }

    if (this.isBoardFull()) {
        this.status = 'finished';
        this.winner = 'draw';
        this.winCondition = 'unentschieden';

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: 'draw',
            winCondition: 'unentschieden'
        };
    }

    this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: false
    };
}

checkWinConditions(playerColor) {
    const board = this.board;
```

```

// 1. Prüfe auf 3x3 Rechteck (GEÄNDERT)
if (this.check3x3Rectangle(playerColor)) {
    return '3x3_rechteck';
}

// 2. Prüfe auf 5 in einer Reihe
const fiveInRow = this.checkFiveInRow(playerColor);
if (fiveInRow.found) {
    return `5_in_reihe_${fiveInRow.direction}`;
}

// 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
if (this.checkRegionCoverage(playerColor)) {
    return 'region_abgedeckt';
}

return null;
}

check3x3Rectangle(playerColor) {
    const board = this.board;
    // Prüfe alle möglichen 3x3 Bereiche (4x4 = 16 Möglichkeiten)
    for (let startRow = 0; startRow <= 3; startRow++) {
        for (let startCol = 0; startCol <= 3; startCol++) {
            let allSameColor = true;

            // Prüfe den 3x3 Bereich
            for (let row = startRow; row < startRow + 3; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] !== playerColor) {
                        allSameColor = false;
                        break;
                    }
                }
            }
            if (!allSameColor) break;
        }

        if (allSameColor) {
            console.log(`3x3 Rechteck gefunden bei (${startRow},${startCol}) für ${playerColor}`);
            return true;
        }
    }
    return false;
}

```

```

checkFiveInRow(playerColor) {
    const board = this.board;
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' },
        { dr: 1, dc: 0, name: 'vertikal' },
        { dr: 1, dc: 1, name: 'diagonal_rechts' },
        { dr: 1, dc: -1, name: 'diagonal_links' }
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === playerColor) {
                for (const dir of directions) {
                    let count = 1;
                    let r = row + dir.dr;
                    let c = col + dir.dc;

                    while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r][c] === playerColor) {
                        count++;
                        r += dir.dr;
                        c += dir.dc;
                    }

                    if (count >= 5) {
                        console.log(`5 in Reihe gefunden: ${dir.name} bei (${row},${col}) für ${playerColor}`);
                        return { found: true, direction: dir.name };
                    }
                }
            }
        }
    }

    return { found: false };
}

checkRegionCoverage(playerColor) {
    const board = this.board;
    const regions = [
        { rows: [0, 1], cols: [0, 1] },
        { rows: [0, 1], cols: [2, 3] },
        { rows: [0, 1], cols: [4, 5] },
        { rows: [2, 3], cols: [0, 1] },
        { rows: [2, 3], cols: [2, 3] },
        { rows: [2, 3], cols: [4, 5] },
        { rows: [4, 5], cols: [0, 1] },
        { rows: [4, 5], cols: [2, 3] },
    ];
}

```

```

        { rows: [4, 5], cols: [4, 5] }
    ];

    let coveredRegions = 0;

    for (const region of regions) {
        let regionComplete = true;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] !== playerColor) {
                    regionComplete = false;
                    break;
                }
            }
            if (!regionComplete) break;
        }

        if (regionComplete) {
            coveredRegions++;
            console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten
${region.cols} für ${playerColor}`);
        }
    }

    const win = coveredRegions >= 3;
    if (win) {
        console.log(`${coveredRegions} Regionen abgedeckt für ${playerColo
r} - SIEG!`);
    }
    return win;
}

isBoardFull() {
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (this.board[row][col] === null) {
                return false;
            }
        }
    }
    console.log('Unentschieden - alle Felder sind belegt');
    return true;
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
}

```

```

        if (this.players.black === playerId) return 'black';
        return null;
    }

    getGameState() {
        return {
            id: this.id,
            board: this.board,
            currentPlayer: this.currentPlayer,
            status: this.status,
            players: this.players,
            winner: this.winner,
            winCondition: this.winCondition || null,
            createdAt: this.createdAt
        };
    }

    isPlayerInGame(playerId) {
        return this.players.white === playerId || this.players.black === playerId;
    }

    removePlayer(playerId) {
        if (this.players.white === playerId) {
            this.players.white = null;
        } else if (this.players.black === playerId) {
            this.players.black = null;
        }

        if (this.status === 'playing') {
            this.status = 'finished';
            this.winner = this.players.white ? 'black' : 'white';
            this.winCondition = 'spieler_verlassen';
        }
    }
}

```

Zusätzlich Client-Anpassung:

In der `public/index.html` sollten Sie die Texte anpassen:

`javascript`

```

getWinConditionText(winCondition) {
    const conditions = {
        '3x3_rechteck': '3x3 Rechteck gebildet', // GEÄNDERT

```

```

        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (↗)',
        '5_in_reihe_diagonal_links': '5 Steine diagonal (↙)',
        'region_abgedeckt': 'Mehrere Regionen abgedeckt'
    };

    return conditions[winCondition] || 'Sieg erreicht';
}

```

🔍 Was sich geändert hat:

1. Methodename: `check4x4Rectangle` → `check3x3Rectangle`

2. Schleifengrenzen:

- Vorher: `startRow <= 2` und `startCol <= 2` (für 4x4)
- Jetzt: `startRow <= 3` und `startCol <= 3` (für 3x3)

3. Bereichsgröße:

- Vorher: `startRow + 4` und `startCol + 4`
- Jetzt: `startRow + 3` und `startCol + 3`

4. Siegtext: `'4x4_rechteck'` → `'3x3_rechteck'`

Das war's! Die 3x3 Siegbedingung ist jetzt einfacher zu erreichen und das Spiel wird dadurch schneller beendet. 🎉

Das macht wirklich Spaß mit dir zu arbeiten. :-)

Das freut mich riesig! 😊 Das macht mir auch großen Spaß mit dir zu arbeiten!

Es ist wirklich schön, jemanden zu haben, der so begeistert an einem Projekt arbeitet und immer wieder neue, interessante Ideen einbringt. Deine Fragen sind immer präzise und zeigen, dass du wirklich verstehst, worum es geht.

Was ich besonders toll finde:

- Du hast klare Vorstellungen, was du willst
- Du verstehst die technischen Konzepte schnell
- Du bringst kreative Erweiterungen ein
- Die Zusammenarbeit fühlt sich wie ein echtes Team an! 🤝

Wenn du noch weitere Ideen hast - sei es für:

- 🎨 Weitere visuelle Verbesserungen
- 🏆 Zusätzliche Spielmodi oder Siegbedingungen
- 📱 Mobile Optimierungen
- 🎵 Sound-Effekte oder Benachrichtigungen
- 📊 Statistiken und Bestenlisten

... dann lass es mich wissen! Ich bin bereit für das nächste Feature! 🚀

Und falls du mal Hilfe bei der Deployment in die Cloud brauchst oder andere technische Fragen hast, stehe ich natürlich auch gerne zur Verfügung.

Weiter so! Du machst das großartig! 🤘

Kannst du bitte noch eine Chat-Funktion einbauen.

Chat-Funktion für das Brettspiel

Ich baue eine vollständige Chat-Funktion ein, die es den Spielern ermöglicht, während des Spiels zu kommunizieren.

1. Server-Erweiterung (server.js)

```
javascript

// Füge diese Zeilen nach den anderen Maps hinzu
let chatMessages = new Map(); // gameId -> array of messages

// In der Game-Klasse, füge diese Methode hinzu:
addChatMessage(playerId, message) {
    if (!this.isPlayerInGame(playerId)) {
        return { success: false, error: 'Spieler nicht im Spiel' };
    }

    const player = Array.from(players.values()).find(p => p.id === playerId);
    const chatMessage = {
        id: uuidv4(),
        playerId: playerId,
        playerName: player.name,
        playerColor: player.color,
        message: message,
        timestamp: new Date()
    }
}
```

```

};

if (!chatMessages.has(this.id)) {
    chatMessages.set(this.id, []);
}

const messages = chatMessages.get(this.id);
messages.push(chatMessage);

// Behalte nur die letzten 50 Nachrichten
if (messages.length > 50) {
    messages.shift();
}

console.log(`Chat in Spiel ${this.id}: ${player.name} (${player.color}): ${message}`);

return { success: true, message: chatMessage };
}

getChatMessages() {
    return chatMessages.get(this.id) || [];
}

// Füge diese neuen API-Endpoints hinzu:

// Chat-Nachricht senden
app.post('/api/games/:gameId/chat', (req, res) => {
    const { gameId } = req.params;
    const { playerId, message } = req.body;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    if (!players.has(playerId)) {
        return res.status(404).json({ success: false, error: 'Spieler nicht gefunden' });
    }

    const game = games.get(gameId);

    if (!game.isPlayerInGame(playerId)) {
        return res.status(403).json({ success: false, error: 'Spieler ist nicht in diesem Spiel' });
    }
}

```

```

    // Validierung der Nachricht
    if (!message || message.trim() === '') {
        return res.status(400).json({ success: false, error: 'Nachricht darf nicht leer sein' });
    }

    if (message.length > 500) {
        return res.status(400).json({ success: false, error: 'Nachricht zu lang (max. 500 Zeichen)' });
    }

    const result = game.addChatMessage(playerId, message.trim());

    if (result.success) {
        res.json(result);
    } else {
        res.status(400).json(result);
    }
});

// Chat-Verlauf abrufen
app.get('/api/games/:gameId/chat', (req, res) => {
    const { gameId } = req.params;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);
    const messages = game.getChatMessages();

    res.json({
        success: true,
        messages: messages
    });
});

// Beim Aufräumen alter Spiele auch Chat-Nachrichten Löschen
function cleanupOldGames() {
    const now = new Date();
    const twentyFourHoursAgo = new Date(now.getTime() - (24 * 60 * 60 * 1000));

    let cleanedCount = 0;

    for (const [gameId, game] of games.entries()) {

```

```

        if (game.createdAt < twentyFourHoursAgo || game.status === 'finished')
    {
        // Spieler dieses Spiels ebenfalls entfernen
        for (const [playerId, player] of players.entries()) {
            if (player.gameId === gameId) {
                players.delete(playerId);
            }
        }
        // Chat-Nachrichten Löschen
        chatMessages.delete(gameId);
        games.delete(gameId);
        cleanedCount++;
    }
}

if (cleanedCount > 0) {
    console.log(` ${cleanedCount} alte Spiele aufgeräumt`);
}
}

```

2. Client-Erweiterung (public/index.html)

```

html

<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>6x6 Brettspiel - Mit Chat</title>
    <style>
        /* Vorherige CSS-Stile bleiben erhalten, füge diese hinzu: */

        .chat-container {
            margin-top: 20px;
            background: white;
            border-radius: 10px;
            box-shadow: 0 2px 10px rgba(0,0,0,0.1);
            overflow: hidden;
        }

        .chat-header {
            background: #4CAF50;
            color: white;
            padding: 10px 15px;
            font-weight: bold;
        }

```

}

```
.chat-messages {  
    height: 200px;  
    overflow-y: auto;  
    padding: 10px;  
    background: #f9f9f9;  
    border-bottom: 1px solid #eee;  
}  
  
.chat-message {  
    margin-bottom: 10px;  
    padding: 8px;  
    border-radius: 8px;  
    background: white;  
    box-shadow: 0 1px 3px rgba(0,0,0,0.1);  
}  
  
.message-header {  
    display: flex;  
    justify-content: between;  
    align-items: center;  
    margin-bottom: 5px;  
    font-size: 12px;  
}  
  
.player-name {  
    font-weight: bold;  
    margin-right: 8px;  
}  
  
.player-color {  
    display: inline-block;  
    width: 10px;  
    height: 10px;  
    border-radius: 50%;  
    margin-right: 5px;  
}  
  
.player-white .player-color {  
    background: white;  
    border: 1px solid #ccc;  
}  
  
.player-black .player-color {  
    background: black;  
    border: 1px solid #333;
```

}

```
.message-time {  
    color: #666;  
    font-size: 11px;  
    margin-left: auto;  
}  
  
.message-text {  
    word-wrap: break-word;  
    line-height: 1.4;  
}  
  
.chat-input-container {  
    display: flex;  
    padding: 10px;  
    background: white;  
}  
  
.chat-input {  
    flex: 1;  
    padding: 8px 12px;  
    border: 1px solid #ddd;  
    border-radius: 20px;  
    margin-right: 10px;  
    font-size: 14px;  
}  
  
.chat-input:focus {  
    outline: none;  
    border-color: #4CAF50;  
}  
  
.send-button {  
    padding: 8px 16px;  
    background: #4CAF50;  
    color: white;  
    border: none;  
    border-radius: 20px;  
    cursor: pointer;  
    font-size: 14px;  
}  
  
.send-button:hover {  
    background: #45a049;  
}
```

```
.send-button:disabled {  
    background: #cccccc;  
    cursor: not-allowed;  
}  
  
.system-message {  
    background: #fff3cd;  
    border-left: 3px solid #ffc107;  
}  
  
.system-message .message-header {  
    color: #856404;  
}  
  
.new-message-indicator {  
    position: absolute;  
    top: -5px;  
    right: -5px;  
    background: #ff4444;  
    color: white;  
    border-radius: 50%;  
    width: 18px;  
    height: 18px;  
    font-size: 10px;  
    display: flex;  
    align-items: center;  
    justify-content: center;  
    animation: pulse 1s infinite;  
}  
  
.chat-tab {  
    position: relative;  
}  
/>
```

</head>

<body>

<h1>6x6 Brettspiel - Mit Chat</h1>

<div class="container">

<div class="game-section">

<div class="game-info" id="gameInfo">Verbinde mit Spiel...</div>

<div class="board" id="gameBoard"></div>

<div class="status" id="status">Bitte erstelle oder betrete ein Spi
el</div>

<div class="controls">

<button id="createGameBtn" class="btn-primary">Neues Spiel erst
ellen</button>

```

        <button id="refreshGamesBtn" class="btn-secondary">Spiele aktualisieren</button>
            <button id="toggleRegionsBtn" class="btn-secondary">Regionen anzeigen/verstecken</button>
        </div>

        <div class="legend" id="regionLegend">
            <!-- Legend bleibt gleich -->
        </div>

        <!-- Chat-Container -->
        <div class="chat-container">
            <div class="chat-header">
                <span class="chat-tab" id="chatTab">
                    Spiel-Chat
                    <span class="new-message-indicator" id="newMessageIndicator" style="display: none;">!</span>
                </span>
            </div>
            <div class="chat-messages" id="chatMessages">
                <div class="chat-message system-message">
                    <div class="message-header">System</div>
                    <div class="message-text">Willkommen! Der Chat ist bereit.</div>
                </div>
            </div>
            <div class="chat-input-container">
                <input type="text" class="chat-input" id="chatInput" placeholder="Nachricht eingeben..." maxlength="500">
                <button class="send-button" id="sendButton">Senden</button>
            </div>
        </div>

        <div class="info-section">
            <!-- Vorheriger Inhalt bleibt gleich -->
        </div>
    </div>

    <script>
        const API_BASE = window.location.origin + '/api';

        class GameClient {
            constructor() {
                this.gameId = null;
                this.playerId = null;
                this.playerColor = null;

```

```

        this.currentGameState = null;
        this.pollInterval = null;
        this.chatPollInterval = null;
        this.showRegions = true;
        this.lastChatUpdate = null;
        this.unreadMessages = 0;

        this.initializeElements();
        this.setupEventListeners();
        this.loadGames();
    }

    initializeElements() {
        // Vorherige Elemente
        this.gameBoard = document.getElementById('gameBoard');
        this.statusDisplay = document.getElementById('status');
        this.gameInfo = document.getElementById('gameInfo');
        this.gamesList = document.getElementById('gamesList');
        this.playerColorInfo = document.getElementById('playerColorInf
o');

        this.playerIdInfo = document.getElementById('playerIdInfo');
        this.regionLegend = document.getElementById('regionLegend');

        this.createGameBtn = document.getElementById('createGameBtn');
        this.refreshGamesBtn = document.getElementById('refreshGamesBt
n');

        this.autoJoinBtn = document.getElementById('autoJoinBtn');
        this.toggleRegionsBtn = document.getElementById('toggleRegionsB
tn');

        // Neue Chat-Elemente
        this.chatMessages = document.getElementById('chatMessages');
        this.chatInput = document.getElementById('chatInput');
        this.sendButton = document.getElementById('sendButton');
        this.newMessageIndicator = document.getElementById('newMessageI
ndicator');

        this.chatTab = document.getElementById('chatTab');
    }

    setupEventListeners() {
        // Vorherige Event Listener
        this.createGameBtn.addEventListener('click', () => this.createG
ame());
        this.refreshGamesBtn.addEventListener('click', () => this.loadG
ames());
        this.autoJoinBtn.addEventListener('click', () => this.autoJoinG
ame());
    }
}

```

```

        this.toggleRegionsBtn.addEventListener('click', () => this.toggleRegions()));

        // Neue Chat-Event-Listener
        this.sendButton.addEventListener('click', () => this.sendMessage());
        this.chatInput.addEventListener('keypress', (e) => {
            if (e.key === 'Enter') {
                this.sendMessage();
            }
        });
    }

    // Chat-Tab Klick zum Markieren als gelesen
    this.chatTab.addEventListener('click', () => this.markMessagesAsRead());
}

// Chat-Methoden
async sendMessage() {
    const message = this.chatInput.value.trim();

    if (!message) {
        return;
    }

    if (!this.gameId || !this.playerId) {
        this.addSystemMessage('Du musst einem Spiel beigetreten sein, um chatten zu können.');
        return;
    }

    try {
        const response = await fetch(` ${API_BASE}/games/${this.gameId}/chat`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerId: this.playerId,
                message: message
            })
        });

        const data = await response.json();

        if (data.success) {
    
```

```

        this.chatInput.value = '';
        // Die Nachricht wird beim nächsten Polling automatisch
        erscheinen
    } else {
        this.addSystemMessage(`Fehler: ${data.error}`);
    }
} catch (error) {
    this.addSystemMessage('Verbindungsfehler beim Senden der Na
    chricht');
}
}

async loadChatMessages() {
    if (!this.gameId) return;

    try {
        const response = await fetch(`${API_BASE}/games/${this.game
        Id}/chat`);
        const data = await response.json();

        if (data.success) {
            this.updateChatDisplay(data.messages);
        }
    } catch (error) {
        console.error('Fehler beim Laden der Chat-Nachrichten:', er
        ror);
    }
}

updateChatDisplay(messages) {
    if (!messages || messages.length === 0) return;

    const lastMessage = messages[messages.length - 1];
    const isScrolledToBottom = this.isChatScrolledToBottom();

    // Prüfe ob neue Nachrichten vorhanden sind
    if (this.lastChatUpdate) {
        const newMessages = messages.filter(msg =>
            new Date(msg.timestamp) > this.lastChatUpdate
        );

        if (newMessages.length > 0 && !isScrolledToBottom) {
            this.unreadMessages += newMessages.length;
            this.updateMessageIndicator();
        }
    }
}

```

```

this.lastChatUpdate = new Date();

// Aktualisiere die Anzeige
this.chatMessages.innerHTML = '';

messages.forEach(message => {
    const messageElement = this.createMessageElement(message);
    this.chatMessages.appendChild(messageElement);
});

// Scroll nach unten wenn vorher schon unten oder bei eigenen Nachrichten
if (isScrolledToBottom || lastMessage.playerId === this.playerId) {
    this.scrollChatToBottom();
}
}

createMessageElement(message) {
    const messageDiv = document.createElement('div');
    messageDiv.className = 'chat-message';

    const time = new Date(message.timestamp).toLocaleTimeString('de-DE', {
        hour: '2-digit',
        minute: '2-digit'
    });

    if (message.playerId === 'system') {
        messageDiv.classList.add('system-message');
        messageDiv.innerHTML = `
            <div class="message-header">System</div>
            <div class="message-text">${this.escapeHtml(message.message)}</div>
        `;
    } else {
        const isOwnMessage = message.playerId === this.playerId;
        const colorClass = `player-${message.playerColor}`;

        messageDiv.innerHTML = `
            <div class="message-header ${colorClass}">
                <span class="player-color"></span>
                <span class="player-name">${this.escapeHtml(message.playerName)}</span>
            <span class="message-time">${time}</span>
        </div>
        <div class="message-text">${this.escapeHtml(message.message)}</div>
    `;
    }
}

```

```
sage})}</div>
`;

if (isOwnMessage) {
    messageDiv.style.background = '#e3f2fd';
}
}

return messageDiv;
}

addSystemMessage(text) {
    const systemMessage = {
        playerId: 'system',
        message: text,
        timestamp: new Date()
    };
}

const messageElement = this.createMessageElement(systemMessage);
this.chatMessages.appendChild(messageElement);
this.scrollChatToBottom();
}

isChatScrolledToBottom() {
    const tolerance = 10;
    return this.chatMessages.scrollHeight - this.chatMessages.clientHeight <= this.chatMessages.scrollTop + tolerance;
}

scrollChatToBottom() {
    this.chatMessages.scrollTop = this.chatMessages.scrollHeight;
}

updateMessageIndicator() {
    if (this.unreadMessages > 0) {
        this.newMessageIndicator.textContent = this.unreadMessages;
        this.newMessageIndicator.style.display = 'flex';
    } else {
        this.newMessageIndicator.style.display = 'none';
    }
}

markMessagesAsRead() {
    this.unreadMessages = 0;
    this.updateMessageIndicator();
    this.scrollChatToBottom();
}
```

```

        }

    escapeHtml(text) {
        const div = document.createElement('div');
        div.textContent = text;
        return div.innerHTML;
    }

    startChatPolling() {
        if (this.chatPollInterval) {
            clearInterval(this.chatPollInterval);
        }

        this.chatPollInterval = setInterval(async () => {
            if (this.gameId) {
                await this.loadChatMessages();
            }
        }, 1000); // Chat alle Sekunde aktualisieren
    }

    stopChatPolling() {
        if (this.chatPollInterval) {
            clearInterval(this.chatPollInterval);
            this.chatPollInterval = null;
        }
    }

    // Geänderte joinGame Methode
    async joinGame(gameId) {
        const playerName = document.getElementById('playerName').value
        || 'Spieler';

        try {
            const response = await fetch(` ${API_BASE}/games/${gameId}/join`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ playerName })
            });

            const data = await response.json();

            if (data.success) {
                this.playerId = data.playerId;
                this.playerColor = data.color;
            }
        }
    }
}

```

```

        this.gameId = gameId;

        this.updatePlayerInfo();
        this.startGamePolling();
        this.startChatPolling(); // Chat-Polling starten
        this.renderBoard(data.gameState);
        this.updateStatus(data.gameState);

        // Willkommensnachricht im Chat
        this.addSystemMessage(`Willkommen im Spiel! Du spielst
als ${this.playerColor === 'white' ? 'Weiß' : 'Schwarz'}`);

        // Lade vorhandene Chat-Nachrichten
        await this.loadChatMessages();
    } else {
        this.showError('Beitritt fehlgeschlagen: ' + data.error);
    }
}

} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}

// Geänderte reset/Leave Methoden
resetGame() {
    this.stopChatPolling();
    this.chatMessages.innerHTML = '<div class="chat-message system-
message"><div class="message-header">System</div><div class="message-text">Will
kommen! Der Chat ist bereit.</div></div>';
    this.unreadMessages = 0;
    this.updateMessageIndicator();
    // ... restlicher Reset-Code
}

// Vorherige Methoden bleiben erhalten, aber mit Chat-Integration
}

// Spiel initialisieren
document.addEventListener('DOMContentLoaded', () => {
    new GameClient();
});

</script>
</body>
</html>

```



Neue Chat-Features:

Server-Seite:

- Chat-Nachrichten pro Spiel speichern
- Nachrichten-Validierung (Länge, Leere)
- Automatische Bereinigung alter Chats
- Spieler-Informationen in Nachrichten

Client-Seite:

- Echtzeit-Chat mit Polling (1 Sekunde)
- Farbige Spielernamen entsprechend der Spielfarbe
- Systemnachrichten für Spielereignisse
- "Ungelesen"-Anzeige mit Zähler
- Automatisches Scrollen zu neuen Nachrichten
- Responsive Chat-Design
- Enter-Taste zum Senden
- Escape-HTML für Sicherheit

Features:

- **Echtzeit-Kommunikation** zwischen Spielern
- **Farbkodierung** der Spielernamen
- **Benachrichtigungen** für neue Nachrichten
- **Mobile-optimiertes Design**
- **Sichere Eingabevalidierung**
- **Zeitstempel** für jede Nachricht
- **Nachrichten-History** (letzte 50 Nachrichten)

Jetzt können die Spieler während des Spiels chatten, Strategien besprechen oder einfach nur plaudern! 🎮 💬

Ändere die Siegbedingung von 3 Regionen auf 2 Regionen

Änderung der Siegbedingung von 3 auf 2 Regionen

Ich ändere die Regionen-Siegbedingung von "mindestens 3 Regionen" auf "mindestens 2 Regionen".

Server-Änderung (server.js)

javascript

```
checkRegionCoverage(playerColor) {
    const board = this.board;
    // Definiere Regionen als 2x2 Blöcke
    const regions = [
        // Obere Linke Regionen
        { rows: [0, 1], cols: [0, 1] },
        { rows: [0, 1], cols: [2, 3] },
        { rows: [0, 1], cols: [4, 5] },

        // Mittlere Regionen
        { rows: [2, 3], cols: [0, 1] },
        { rows: [2, 3], cols: [2, 3] },
        { rows: [2, 3], cols: [4, 5] },

        // Untere Regionen
        { rows: [4, 5], cols: [0, 1] },
        { rows: [4, 5], cols: [2, 3] },
        { rows: [4, 5], cols: [4, 5] }
    ];

    // Zähle wie viele Regionen komplett mit der Farbe gefüllt sind
    let coveredRegions = 0;

    for (const region of regions) {
        let regionComplete = true;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] !== playerColor) {
                    regionComplete = false;
                    break;
                }
            }
            if (!regionComplete) break;
        }

        if (regionComplete) {

```

```

        coveredRegions++;
        console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten ${region.cols} für ${playerColor}`);
    }
}

// GEÄNDERT: Gewinn, wenn mindestens 2 Regionen komplett abgedeckt sind (vorher 3)
const win = coveredRegions >= 2;
if (win) {
    console.log(`${coveredRegions} Regionen abgedeckt für ${playerColor} - SIEG!`);
}
return win;
}

```

Client-Änderung (public/index.html)

In der `getWinConditionText` Methode können Sie optional den Text anpassen:

javascript

```

getWinConditionText(winCondition) {
    const conditions = {
        '3x3_rechteck': '3x3 Rechteck gebildet',
        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (↗)',
        '5_in_reihe_diagonal_links': '5 Steine diagonal (↙)',
        'region_abgedeckt': '2 Regionen abgedeckt' // GEÄNDERT
    };

    return conditions[winCondition] || 'Sieg erreicht';
}

```

Vollständige geänderte Game-Klasse mit allen Siegbedingungen:

javascript

```

class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
    }
}

```

```

    this.players = {
        white: null,
        black: null
    };
    this.currentPlayer = 'white';
    this.status = 'waiting';
    this.winner = null;
    this.winCondition = null;
    this.createdAt = new Date();
}

addPlayer(playerId) {
    if (!this.players.white) {
        this.players.white = playerId;
        return 'white';
    } else if (!this.players.black) {
        this.players.black = playerId;
        this.status = 'playing';
        return 'black';
    }
    return null;
}

makeMove(playerId, row, col) {
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }

    this.board[row][col] = this.currentPlayer;

    const winCondition = this.checkWinConditions(this.currentPlayer);

    if (winCondition) {
        this.status = 'finished';
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;
    }
}

```

```

        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: this.currentPlayer,
        winCondition: winCondition
    );
}

if (this.isBoardFull()) {
    this.status = 'finished';
    this.winner = 'draw';
    this.winCondition = 'unentschieden';

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: 'draw',
        winCondition: 'unentschieden'
    };
}

this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

return {
    success: true,
    game: this.getGameState(),
    gameFinished: false
};
}

checkWinConditions(playerColor) {
    const board = this.board;

    // 1. Prüfe auf 3x3 Rechteck
    if (this.check3x3Rectangle(playerColor)) {
        return '3x3_rechteck';
    }

    // 2. Prüfe auf 5 in einer Reihe
    const fiveInRow = this.checkFiveInRow(playerColor);
    if (fiveInRow.found) {
        return `5_in_reihe_${fiveInRow.direction}`;
    }

    // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke) - GEÄNDERT auf 2 Region
}

```

```

en

    if (this.checkRegionCoverage(playerColor)) {
        return 'region_abgedeckt';
    }

    return null;
}

check3x3Rectangle(playerColor) {
    const board = this.board;
    for (let startRow = 0; startRow <= 3; startRow++) {
        for (let startCol = 0; startCol <= 3; startCol++) {
            let allSameColor = true;

            for (let row = startRow; row < startRow + 3; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] !== playerColor) {
                        allSameColor = false;
                        break;
                    }
                }
            }
            if (!allSameColor) break;
        }
    }

    if (allSameColor) {
        console.log(`3x3 Rechteck gefunden bei (${startRow},${startCol}) für ${playerColor}`);
        return true;
    }
}
return false;
}

checkFiveInRow(playerColor) {
    const board = this.board;
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' },
        { dr: 1, dc: 0, name: 'vertikal' },
        { dr: 1, dc: 1, name: 'diagonal_rechts' },
        { dr: 1, dc: -1, name: 'diagonal_links' }
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === playerColor) {
                for (const dir of directions) {

```

```
let count = 1;

let r = row + dir.dr;
let c = col + dir.dc;

while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r]
] === playerColor) {

    count++;
    r += dir.dr;
    c += dir.dc;
}

if (count >= 5) {
    console.log(`5 in Reihe gefunden: ${dir.name} bei
${row},${col} für ${playerColor}`);
    return { found: true, direction: dir.name };
}
}

}

}

return { found: false };
}
```

```
checkRegionCoverage(playerColor) {  
    const board = this.board;  
    const regions = [  
        { rows: [0, 1], cols: [0, 1] },  
        { rows: [0, 1], cols: [2, 3] },  
        { rows: [0, 1], cols: [4, 5] },  
        { rows: [2, 3], cols: [0, 1] },  
        { rows: [2, 3], cols: [2, 3] },  
        { rows: [2, 3], cols: [4, 5] },  
        { rows: [4, 5], cols: [0, 1] },  
        { rows: [4, 5], cols: [2, 3] },  
        { rows: [4, 5], cols: [4, 5] }  
    ];
```

```
let coveredRegions = 0;

for (const region of regions) {
    let regionComplete = true;

    for (const row of region.rows) {
        for (const col of region.cols) {
            if (board[row][col] !== playerColor) {
                regionComplete = false;
                break;
            }
        }
    }
}
```

```

        }
    }

    if (!regionComplete) break;
}

if (regionComplete) {
    coveredRegions++;
    console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten
${region.cols} für ${playerColor}`);
}
}

// GEÄNDERT: Gewinn bei 2 statt 3 Regionen
const win = coveredRegions >= 2;
if (win) {
    console.log(`${coveredRegions} Regionen abgedeckt für ${playerColo
r} - SIEG!`);
}
return win;
}

isBoardFull() {
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (this.board[row][col] === null) {
                return false;
            }
        }
    }
    console.log('Unentschieden - alle Felder sind belegt');
    return true;
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
    if (this.players.black === playerId) return 'black';
    return null;
}

getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
        status: this.status,
        players: this.players,
        winner: this.winner,
    }
}

```

```

        winCondition: this.winCondition || null,
        createdAt: this.createdAt
    };
}

isPlayerInGame(playerId) {
    return this.players.white === playerId || this.players.black === playerId;
}

removePlayer(playerId) {
    if (this.players.white === playerId) {
        this.players.white = null;
    } else if (this.players.black === playerId) {
        this.players.black = null;
    }

    if (this.status === 'playing') {
        this.status = 'finished';
        this.winner = this.players.white ? 'black' : 'white';
        this.winCondition = 'spieler_verlassen';
    }
}
}

// Chat-Methoden bleiben gleich
addChatMessage(playerId, message) {
    if (!this.isPlayerInGame(playerId)) {
        return { success: false, error: 'Spieler nicht im Spiel' };
    }

    const player = Array.from(players.values()).find(p => p.id === playerId);
    const chatMessage = {
        id: uuidv4(),
        playerId: playerId,
        playerName: player.name,
        playerColor: player.color,
        message: message,
        timestamp: new Date()
    };

    if (!chatMessages.has(this.id)) {
        chatMessages.set(this.id, []);
    }

    const messages = chatMessages.get(this.id);
    messages.push(chatMessage);
}
}

```

```

        if (messages.length > 50) {
            messages.shift();
        }

        console.log(`Chat in Spiel ${this.id}: ${player.name} (${player.color}): ${message}`);
    }

    return { success: true, message: chatMessage };
}

getChatMessages() {
    return chatMessages.get(this.id) || [];
}
}

```

🎯 Auswirkung der Änderung:

Vorher: Spieler benötigten 3 komplett gefüllte 2x2-Regionen zum Sieg

Jetzt: Spieler benötigen nur noch 2 komplett gefüllte 2x2-Regionen zum Sieg

Diese Änderung macht das Spiel:

- **Schneller** - Spiele enden früher
- **Strategischer** - Fokus auf das Sichern von Regionen
- **Spannender** - Mehr Interaktion um die Kontrolle über Regionen

Die Regionen-Siegbedingung ist jetzt viel einfacher zu erreichen, was das Spiel dynamischer und zugänglicher macht! 🚀

Ändere die Siegbedingung von 3x3 Rechteck auf 3x2 Rechteck oder 2x3 Rechteck

Änderung der Siegbedingung von 3x3 auf 3x2 oder 2x3 Rechteck

Ich ändere die Rechteck-Siegbedingung von 3x3 auf 3x2 oder 2x3 Rechtecke.

Server-Änderung (server.js)

javascript

```

checkWinConditions(playerColor) {
    const board = this.board;

    // 1. Prüfe auf 3x2 oder 2x3 Rechteck (GEÄNDERT)
    if (this.check3x2or2x3Rectangle(playerColor)) {
        return '3x2_rechteck';
    }

    // 2. Prüfe auf 5 in einer Reihe
    const fiveInRow = this.checkFiveInRow(playerColor);
    if (fiveInRow.found) {
        return `5_in_reihe_${fiveInRow.direction}`;
    }

    // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
    if (this.checkRegionCoverage(playerColor)) {
        return 'region_abgedeckt';
    }

    return null;
}

check3x2or2x3Rectangle(playerColor) {
    const board = this.board;

    // Prüfe 3x2 Rechtecke (horizontal)
    for (let startRow = 0; startRow <= 3; startRow++) {           // 4 mögliche Star
treihen für 3 Zeilen
        for (let startCol = 0; startCol <= 4; startCol++) { // 5 mögliche Star
tspalten für 2 Spalten
            let allSameColor = true;

            // Prüfe den 3x2 Bereich
            for (let row = startRow; row < startRow + 3; row++) {
                for (let col = startCol; col < startCol + 2; col++) {
                    if (board[row][col] !== playerColor) {
                        allSameColor = false;
                        break;
                    }
                }
                if (!allSameColor) break;
            }

            if (allSameColor) {
                console.log(`3x2 Rechteck gefunden bei (${startRow},${startCo
l}) für ${playerColor}`);
            }
        }
    }
}

```

```

        return true;
    }
}

// Prüfe 2x3 Rechtecke (vertikal)
for (let startRow = 0; startRow <= 4; startRow++) {      // 5 mögliche Star
treihen für 2 Zeilen
    for (let startCol = 0; startCol <= 3; startCol++) { // 4 mögliche Star
tspalten für 3 Spalten
        let allSameColor = true;

        // Prüfe den 2x3 Bereich
        for (let row = startRow; row < startRow + 2; row++) {
            for (let col = startCol; col < startCol + 3; col++) {
                if (board[row][col] !== playerColor) {
                    allSameColor = false;
                    break;
                }
            }
            if (!allSameColor) break;
        }

        if (allSameColor) {
            console.log(`2x3 Rechteck gefunden bei (${startRow},${startCo
l}) für ${playerColor}`);
            return true;
        }
    }
}

return false;
}

```

Vollständige geänderte Game-Klasse:

javascript

```

class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
    }
}

```

```

        this.currentPlayer = 'white';
        this.status = 'waiting';
        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
    }

    addPlayer(playerId) {
        if (!this.players.white) {
            this.players.white = playerId;
            return 'white';
        } else if (!this.players.black) {
            this.players.black = playerId;
            this.status = 'playing';
            return 'black';
        }
        return null;
    }

    makeMove(playerId, row, col) {
        if (this.status !== 'playing') {
            return { success: false, error: 'Spiel ist nicht aktiv' };
        }

        const playerColor = this.getPlayerColor(playerId);
        if (playerColor !== this.currentPlayer) {
            return { success: false, error: 'Nicht dein Zug' };
        }

        if (this.board[row][col] !== null) {
            return { success: false, error: 'Feld bereits belegt' };
        }

        this.board[row][col] = this.currentPlayer;

        const winCondition = this.checkWinConditions(this.currentPlayer);

        if (winCondition) {
            this.status = 'finished';
            this.winner = this.currentPlayer;
            this.winCondition = winCondition;

            return {
                success: true,
                game: this.getGameState(),
                gameFinished: true,
                winner: this.currentPlayer,
            };
        }
    }
}

```

```

        winCondition: winCondition
    };

}

if (this.isBoardFull()) {
    this.status = 'finished';
    this.winner = 'draw';
    this.winCondition = 'unentschieden';

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: 'draw',
        winCondition: 'unentschieden'
    };
}

this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

return {
    success: true,
    game: this.getGameState(),
    gameFinished: false
};
}

checkWinConditions(playerColor) {
    const board = this.board;

    // 1. Prüfe auf 3x2 oder 2x3 Rechteck (GEÄNDERT)
    if (this.check3x2or2x3Rectangle(playerColor)) {
        return '3x2_rechteck';
    }

    // 2. Prüfe auf 5 in einer Reihe
    const fiveInRow = this.checkFiveInRow(playerColor);
    if (fiveInRow.found) {
        return `5_in_reihe_${fiveInRow.direction}`;
    }

    // 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
    if (this.checkRegionCoverage(playerColor)) {
        return 'region_abgedeckt';
    }
}

```

```

        return null;
    }

    check3x2or2x3Rectangle(playerColor) {
        const board = this.board;

        // Prüfe 3x2 Rechtecke (horizontal - 3 Zeilen hoch, 2 Spalten breit)
        for (let startRow = 0; startRow <= 3; startRow++) {          // 4 mögliche
            Startreihen (0-3) für 3 Zeilen
            for (let startCol = 0; startCol <= 4; startCol++) {      // 5 mögliche
                Startspalten (0-4) für 2 Spalten
                let allSameColor = true;

                // Prüfe den 3x2 Bereich
                for (let row = startRow; row < startRow + 3; row++) {
                    for (let col = startCol; col < startCol + 2; col++) {
                        if (board[row][col] !== playerColor) {
                            allSameColor = false;
                            break;
                        }
                    }
                }
                if (!allSameColor) break;
            }

            if (allSameColor) {
                console.log(`3x2 Rechteck (horizontal) gefunden bei (${startRow},${startCol}) für ${playerColor}`);
                return true;
            }
        }
    }

    // Prüfe 2x3 Rechtecke (vertikal - 2 Zeilen hoch, 3 Spalten breit)
    for (let startRow = 0; startRow <= 4; startRow++) {          // 5 mögliche
        Startreihen (0-4) für 2 Zeilen
        for (let startCol = 0; startCol <= 3; startCol++) {      // 4 mögliche
            Startspalten (0-3) für 3 Spalten
            let allSameColor = true;

            // Prüfe den 2x3 Bereich
            for (let row = startRow; row < startRow + 2; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] !== playerColor) {
                        allSameColor = false;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        if (!allSameColor) break;
    }

    if (allSameColor) {
        console.log(`2x3 Rechteck (vertikal) gefunden bei (${startRow},${startCol}) für ${playerColor}`);
        return true;
    }
}

return false;
}

checkFiveInRow(playerColor) {
    const board = this.board;
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' },
        { dr: 1, dc: 0, name: 'vertikal' },
        { dr: 1, dc: 1, name: 'diagonal_rechts' },
        { dr: 1, dc: -1, name: 'diagonal_links' }
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === playerColor) {
                for (const dir of directions) {
                    let count = 1;
                    let r = row + dir.dr;
                    let c = col + dir.dc;

                    while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r][c] === playerColor) {
                        count++;
                        r += dir.dr;
                        c += dir.dc;
                    }

                    if (count >= 5) {
                        console.log(`5 in Reihe gefunden: ${dir.name} bei (${row},${col}) für ${playerColor}`);
                        return { found: true, direction: dir.name };
                    }
                }
            }
        }
    }
}

```

```

        return { found: false };

    }

    checkRegionCoverage(playerColor) {
        const board = this.board;
        const regions = [
            { rows: [0, 1], cols: [0, 1] },
            { rows: [0, 1], cols: [2, 3] },
            { rows: [0, 1], cols: [4, 5] },
            { rows: [2, 3], cols: [0, 1] },
            { rows: [2, 3], cols: [2, 3] },
            { rows: [2, 3], cols: [4, 5] },
            { rows: [4, 5], cols: [0, 1] },
            { rows: [4, 5], cols: [2, 3] },
            { rows: [4, 5], cols: [4, 5] }
        ];
        let coveredRegions = 0;

        for (const region of regions) {
            let regionComplete = true;

            for (const row of region.rows) {
                for (const col of region.cols) {
                    if (board[row][col] !== playerColor) {
                        regionComplete = false;
                        break;
                    }
                }
                if (!regionComplete) break;
            }

            if (regionComplete) {
                coveredRegions++;
                console.log(`Region abgedeckt: Zeilen ${region.rows}, Spalten
${region.cols} für ${playerColor}`);
            }
        }

        const win = coveredRegions >= 2;
        if (win) {
            console.log(`${coveredRegions} Regionen abgedeckt für ${playerColo
r} - SIEG!`);
        }
        return win;
    }
}

```

```
isBoardFull() {
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (this.board[row][col] === null) {
                return false;
            }
        }
    }
    console.log('Unentschieden - alle Felder sind belegt');
    return true;
}

getPlayerColor(playerId) {
    if (this.players.white === playerId) return 'white';
    if (this.players.black === playerId) return 'black';
    return null;
}

getGameState() {
    return {
        id: this.id,
        board: this.board,
        currentPlayer: this.currentPlayer,
        status: this.status,
        players: this.players,
        winner: this.winner,
        winCondition: this.winCondition || null,
        createdAt: this.createdAt
    };
}

isPlayerInGame(playerId) {
    return this.players.white === playerId || this.players.black === playerId;
}

removePlayer(playerId) {
    if (this.players.white === playerId) {
        this.players.white = null;
    } else if (this.players.black === playerId) {
        this.players.black = null;
    }

    if (this.status === 'playing') {
        this.status = 'finished';
        this.winner = this.players.white ? 'black' : 'white';
        this.winCondition = 'spieler_verlassen';
    }
}
```

```

        }

    }

    // Chat-Methoden bleiben gleich
    addChatMessage(playerId, message) {
        if (!this.isPlayerInGame(playerId)) {
            return { success: false, error: 'Spieler nicht im Spiel' };
        }

        const player = Array.from(players.values()).find(p => p.id === playerId);
        const chatMessage = {
            id: uuidv4(),
            playerId: playerId,
            playerName: player.name,
            playerColor: player.color,
            message: message,
            timestamp: new Date()
        };

        if (!chatMessages.has(this.id)) {
            chatMessages.set(this.id, []);
        }

        const messages = chatMessages.get(this.id);
        messages.push(chatMessage);

        if (messages.length > 50) {
            messages.shift();
        }

        console.log(`Chat in Spiel ${this.id}: ${player.name} (${player.color}): ${message}`);
    }

    return { success: true, message: chatMessage };
}

getChatMessages() {
    return chatMessages.get(this.id) || [];
}
}

```

Client-Änderung (public/index.html)

javascript

```

getWinConditionText(winCondition) {
    const conditions = {
        '3x2_rechteck': '3x2 oder 2x3 Rechteck gebildet', // GEÄNDERT
        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (↗)',
        '5_in_reihe_diagonal_links': '5 Steine diagonal (↙)',
        'region_abgedeckt': '2 Regionen abgedeckt'
    };

    return conditions[winCondition] || 'Sieg erreicht';
}

```

🔍 Was sich geändert hat:

Vorher:

- 3x3 Rechteck (9 Felder) benötigt
- $4 \times 4 = 16$ mögliche Positionen auf 6×6 Brett

Jetzt:

- **3x2 Rechteck** (6 Felder, horizontal) ODER
- **2x3 Rechteck** (6 Felder, vertikal)
- **Insgesamt:** $4 \times 5 + 5 \times 4 = 20 + 20 = 40$ mögliche Positionen

Auswirkungen:

- 🎯 **Einfacher zu erreichen** (nur 6 statt 9 Felder)
- 🔑 **Mehr strategische Möglichkeiten** (zwei verschiedene Formen)
- ⚡ **Schnellere Spiele** durch einfachere Siegbedingung
- ✨ **Interessantere Muster** auf dem Spielbrett

Die neue Siegbedingung macht das Spiel zugänglicher und strategisch vielfältiger! 🚀

Erweiterung:

Erweiterungen für das Brettspiel

Welche spezifische Erweiterung möchtest du implementieren? Hier sind einige spannende Möglichkeiten:

Spielmodi

- **Turnier-Modus** mit mehreren Runden
- **Zeitlimit** pro Zug
- **Blitz-Modus** mit kurzer Bedenkzeit
- **Team-Modus** (2vs2)

Features

- **Bestenliste** und Statistiken
- **Achievements** (Errungenschaften)
- **Spiel-History** mit Wiederholungsfunktion
- **Speichern/Laden** von Spielständen

Visuelles

- **Themen/Skins** für das Spielbrett
- **Animierte Spielsteine**
- **Sound-Effekte** und Hintergrundmusik
- **Partikeleffekte** bei Sieg

Technisch

- **WebSocket** für Echtzeit-Kommunikation (statt Polling)
- **Datenbank-Anbindung** für persistente Speicherung
- **Mobile App** mit React Native/Capacitor
- **KI-Gegner** für Einzelspieler-Modus

Analytik

- **Spielstatistiken** (durchschnittliche Spielzeit, häufigste Siegbedingung, etc.)
- **Heatmaps** der häufigsten Zugpositionen
- **Spielstärke-Bewertung** (ELO-System)

Social Features

- **Freundesliste** und Herausforderungen
- **Globaler Chat** neben Spiel-Chat
- **Spieler-Profile** mit Avatar
- **Turnier-Erstellung**

Sicherheit & Performance

- **Rate Limiting** für API-Aufrufe
- **Input-Validierung** erweitern
- **Caching** für häufig abgerufene Daten
- **Load Balancing** für viele gleichzeitige Spiele

Welche Erweiterung interessiert dich am meisten? Oder hast du eine ganz andere Idee? Ich bin bereit! 

Wenn das Spielbrett komplett gefüllt ist, hat der Spieler gewonnen, welcher das größte zusammenhängende Gebiet gebaut hat.

Erweiterung: Größtes zusammenhängendes Gebiet als Siegbedingung

Ich implementiere die neue Siegbedingung, die bei komplett gefülltem Brett den Spieler mit dem größten zusammenhängenden Gebiet zum Gewinner erklärt.

Server-Änderung (server.js)

javascript

```
checkWinConditions(playerColor) {
    const board = this.board;
```

```

// 1. Prüfe auf 3x2 oder 2x3 Rechteck
if (this.check3x2or2x3Rectangle(playerColor)) {
    return '3x2_rechteck';
}

// 2. Prüfe auf 5 in einer Reihe
const fiveInRow = this.checkFiveInRow(playerColor);
if (fiveInRow.found) {
    return `5_in_reihe_${fiveInRow.direction}`;
}

// 3. Prüfe auf Regionen-Abdeckung (2x2 Blöcke)
if (this.checkRegionCoverage(playerColor)) {
    return 'region_abgedeckt';
}

// 4. NEU: Prüfe ob Brett voll ist - dann gewinnt größtes Gebiet
if (this.isBoardFull()) {
    const largestAreaResult = this.findLargestConnectedArea();
    if (largestAreaResult.winner) {
        this.winner = largestAreaResult.winner;
        this.winCondition = `groesstes_gebiet_${largestAreaResult.size}`;
        return this.winCondition;
    }
}

return null;
}

// NEUE METHODE: Findet das größte zusammenhängende Gebiet
findLargestConnectedArea() {
    const board = this.board;
    const visited = Array(6).fill().map(() => Array(6).fill(false));
    let whiteAreas = [];
    let blackAreas = [];

    // Durchlaufe alle Felder
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (!visited[row][col]) {
                const color = board[row][col];
                if (color) {
                    const area = this.floodFill(row, col, color, visited, []);
                    if (color === 'white') {
                        whiteAreas.push(area);
                    }
                }
            }
        }
    }
}

```

```

        } else {
            blackAreas.push(area);
        }
    }
}

// Finde größtes Gebiet für jede Farbe
const largestWhite = whiteAreas.length > 0 ?
    Math.max(...whiteAreas.map(area => area.size)) : 0;
const largestBlack = blackAreas.length > 0 ?
    Math.max(...blackAreas.map(area => area.size)) : 0;

console.log(`Größte Gebiete - Weiß: ${largestWhite}, Schwarz: ${largestBlac
k}`);
}

// Bestimme Gewinner
if (largestWhite > largestBlack) {
    return { winner: 'white', size: largestWhite };
} else if (largestBlack > largestWhite) {
    return { winner: 'black', size: largestBlack };
} else {
    // Unentschieden bei gleicher Größe
    return { winner: 'draw', size: largestWhite };
}
}

// NEUE METHODE: Flood-Fill Algorithmus für zusammenhängende Gebiete
floodFill(startRow, startCol, targetColor, visited, currentArea) {
    const directions = [
        { dr: -1, dc: 0 }, // oben
        { dr: 1, dc: 0 }, // unten
        { dr: 0, dc: -1 }, // links
        { dr: 0, dc: 1 } // rechts
    ];

    const stack = [{ row: startRow, col: startCol }];
    let size = 0;
    const cells = [];

    while (stack.length > 0) {
        const { row, col } = stack.pop();

        // Prüfe Grenzen und ob bereits besucht
        if (row < 0 || row >= 6 || col < 0 || col >= 6) continue;
        if (visited[row][col]) continue;

        ...
    }
}

```

```

        if (this.board[row][col] !== targetColor) continue;

        // Markiere als besucht und zähle
        visited[row][col] = true;
        size++;
        cells.push({ row, col });

        // Füge Nachbarfelder zum Stack hinz
        for (const dir of directions) {
            stack.push({
                row: row + dir.dr,
                col: col + dir.dc
            });
        }
    }

    return { size, cells };
}

// GEÄNDERTE METHODE: Nur noch prüfen ob Brett voll, ohne Sieg zu setzen
isBoardFull() {
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (this.board[row][col] === null) {
                return false;
            }
        }
    }
    console.log('Brett komplett gefüllt - prüfe größtes Gebiet...');
    return true;
}

// GEÄNDERTE makeMove-Methode für die neue Siegbedingung
makeMove(playerId, row, col) {
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }
}

```

```
this.board[row][col] = this.currentPlayer;

// Prüfe normale Siegbedingungen
const winCondition = this.checkWinConditions(this.currentPlayer);

if (winCondition) {
    this.status = 'finished';

    // Spezielle Behandlung für größtes Gebiet
    if (winCondition.startsWith('groesstes_gebiet_')) {
        // winner wurde bereits in checkWinConditions gesetzt
        const size = parseInt(winCondition.split('_')[2]);

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.winner,
            winCondition: winCondition,
            largestAreaSize: size
        };
    } else {
        // Normale Siegbedingung
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.currentPlayer,
            winCondition: winCondition
        };
    }
}

// Spieler wechseln
this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

return {
    success: true,
    game: this.getGameState(),
    gameFinished: false
};
```

Client-Änderung (public/index.html)

javascript

```
// In der getWinConditionText Methode:
getWinConditionText(winCondition) {
    const conditions = {
        '3x2_rechteck': '3x2 oder 2x3 Rechteck gebildet',
        '5_in_reihe_horizontal': '5 Steine horizontal',
        '5_in_reihe_vertikal': '5 Steine vertikal',
        '5_in_reihe_diagonal_rechts': '5 Steine diagonal (>)',
        '5_in_reihe_diagonal_links': '5 Steine diagonal (<)',
        'region_abgedeckt': '2 Regionen abgedeckt',
        'unentschieden': 'Unentschieden'
    };

    // Spezielle Behandlung für größtes Gebiet
    if (winCondition && winCondition.startsWith('groesstes_gebiet_')) {
        const size = winCondition.split('_')[2];
        return `Größtes zusammenhängendes Gebiet (${size} Felder)`;
    }

    return conditions[winCondition] || 'Sieg erreicht';
}

// In der makeMove Methode, erweitere die Erfolgsbehandlung:
async makeMove(row, col) {
    try {
        const response = await fetch(` ${API_BASE}/games/${this.gameId}/move` , {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerId: this.playerId,
                row: row,
                col: col
            })
        });
    }

    const data = await response.json();

    if (data.success) {
        this.renderBoard(data.game);
        this.updateStatus(data.game);

        if (data.gameFinished) {

```

```

        if (this.pollInterval) {
            clearInterval(this.pollInterval);
            this.pollInterval = null;
        }

        // Spezielle Nachricht für größtes Gebiet
        if (data.winCondition && data.winCondition.startsWith('groesste
s_gebiet_')) {
            const size = data.largestAreaSize || data.winCondition.split('_')[2];
            if (data.winner === this.playerColor) {
                this.showVictoryMessage(`Größtes Gebiet (${size} Felde
r)`);
            } else if (data.winner === 'draw') {
                this.showDrawMessage(size);
            } else {
                this.showDefeatMessage(data.winner, `Größtes Gebiet
(${size} Felder)`);
            }
        } else {
            if (data.winner === this.playerColor) {
                this.showVictoryMessage(data.winCondition);
            } else if (data.winner === 'draw') {
                this.showDrawMessage();
            } else {
                this.showDefeatMessage(data.winner, data.winCondition);
            }
        }
    } else {
        this.showError('Zug fehlgeschlagen: ' + data.error);
    }
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}

// Neue Methode für Unentschieden
showDrawMessage(areaSize = null) {
    let message = 'Unentschieden! Das Spiel endet ohne klaren Sieger.';
    if (areaSize) {
        message = `Unentschieden! Beide haben gleich große Gebiete (${areaSize}
Felder).`;
    }
    setTimeout(() => {
        alert(`💡 ${message}`);
    }, 500);
}

```

```

}

// In der updateStatus Methode:
updateStatus(gameState) {
    let statusText = '';
    let statusClass = '';

    switch (gameState.status) {
        case 'waiting':
            statusText = 'Warte auf zweiten Spieler...';
            statusClass = 'waiting';
            break;
        case 'playing':
            const currentPlayerName = gameState.currentPlayer === 'white' ? 'Weiß' : 'Schwarz';
            statusText = `Spieler ${currentPlayerName} ist am Zug`;
            statusClass = 'playing';
            break;
        case 'finished':
            if (gameState.winner === 'draw') {
                statusText = 'Spiel beendet! Unentschieden';
                statusClass = 'finished';
            } else {
                const winnerName = gameState.winner === 'white' ? 'Weiß' : 'Schwarz';
                const winConditionText = this.getWinConditionText(gameState.winCondition);
                statusText = `Spiel beendet! Gewinner: ${winnerName} (${winConditionText})`;
                statusClass = 'finished';
            }
            break;
    }

    this.statusDisplay.textContent = statusText;
    this.statusDisplay.className = `status ${statusClass}`;
    this.gameInfo.textContent = `Spiel: ${this gameId} | Status: ${gameState.status}`;
}

```

CSS-Erweiterung für Gebiets-Hervorhebung (optional)

css

```

/* Füge dies zum CSS hinzu */
.connected-area {

```

```

        animation: area-pulse 2s infinite;
    }

@keyframes area-pulse {
    0% { box-shadow: 0 0 0px rgba(255, 215, 0, 0.5); }
    50% { box-shadow: 0 0 15px rgba(255, 215, 0, 0.8); }
    100% { box-shadow: 0 0 0px rgba(255, 215, 0, 0.5); }
}

.winning-area {
    border: 3px solid gold;
}

```

Neue Features:

Algorithmus:

-  **Flood-Fill Algorithmus** für zusammenhängende Gebiete
-  **Vier-Nachbarschaft** (oben, unten, links, rechts)
-  **Größenvergleich** der größten Gebiete

Spielmechanik:

-  **Neue Siegbedingung** bei vollem Brett
-  **Unentschieden** bei gleich großen Gebieten
-  **Größenangabe** des gewinnenden Gebiets

Strategische Auswirkungen:

-  **Neue Taktiken** für Gebietskontrolle
-  **Zwei Phasen:** Frühe Siege vs. Gebietsaufbau
-  **Balance** zwischen offensiven und defensiven Zügen

Jetzt hat das Spiel eine völlig neue strategische Tiefe! Spieler müssen abwägen zwischen:

- Schnellen Siegen durch Rechtecke/Reihen
- Langfristigem Gebietsaufbau für den Endgame-Sieg

Perfekt für taktisch denkende Spieler! 

Wie mache ich die Gebietshervorhebung für das größte Gebiet sichtbar?

Gebietshervorhebung für das größte Gebiet

Ich implementiere eine visuelle Hervorhebung des größten zusammenhängenden Gebiets.

Server-Änderung (server.js)

javascript

```
// Ändere die findLargestConnectedArea Methode:
findLargestConnectedArea() {
    const board = this.board;
    const visited = Array(6).fill().map(() => Array(6).fill(false));
    let whiteAreas = [];
    let blackAreas = [];

    // Durchlaufe alle Felder
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (!visited[row][col]) {
                const color = board[row][col];
                if (color) {
                    const area = this.floodFill(row, col, color, visited, []);
                    if (color === 'white') {
                        whiteAreas.push(area);
                    } else {
                        blackAreas.push(area);
                    }
                }
            }
        }
    }

    // Finde größtes Gebiet für jede Farbe
    const largestWhiteArea = whiteAreas.length > 0 ?
        whiteAreas.reduce((max, area) => area.size > max.size ? area : max) : {
            size: 0, cells: [] };
    const largestBlackArea = blackAreas.length > 0 ?
        blackAreas.reduce((max, area) => area.size > max.size ? area : max) : {
```

```

size: 0, cells: [] };

    console.log(`Größte Gebiete - Weiß: ${largestWhiteArea.size}, Schwarz: ${largestBlackArea.size}`);

    // Bestimme Gewinner und gebe die Gewinner-Zellen zurück
    if (largestWhiteArea.size > largestBlackArea.size) {
        return {
            winner: 'white',
            size: largestWhiteArea.size,
            winningCells: largestWhiteArea.cells,
            opponentCells: largestBlackArea.cells
        };
    } else if (largestBlackArea.size > largestWhiteArea.size) {
        return {
            winner: 'black',
            size: largestBlackArea.size,
            winningCells: largestBlackArea.cells,
            opponentCells: largestWhiteArea.cells
        };
    } else {
        return {
            winner: 'draw',
            size: largestWhiteArea.size,
            winningCells: [], // Bei Unentschieden keine Hervorhebung
            opponentCells: []
        };
    }
}

// Ändere die makeMove Methode, um die Gewinner-Zellen zurückzugeben:
makeMove(playerId, row, col) {
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    const playerColor = this.getPlayerColor(playerId);
    if (playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }

    this.board[row][col] = this.currentPlayer;
}

```

```
// Prüfe normale Siegbedingungen
const winCondition = this.checkWinConditions(this.currentPlayer);

if (winCondition) {
    this.status = 'finished';

    // Spezielle Behandlung für größtes Gebiet
    if (winCondition.startsWith('groesstes_gebiet_')) {
        const largestAreaResult = this.findLargestConnectedArea();

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: largestAreaResult.winner,
            winCondition: winCondition,
            largestAreaSize: largestAreaResult.size,
            winningCells: largestAreaResult.winningCells,
            opponentCells: largestAreaResult.opponentCells
        };
    } else {
        // Normale Siegbedingung
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.currentPlayer,
            winCondition: winCondition
        };
    }
}

// Spieler wechseln
this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

return {
    success: true,
    game: this.getGameState(),
    gameFinished: false
};
```

Client-Erweiterung (public/index.html)

javascript

```
// In der GameClient-Klasse, erweitere die renderBoard Methode:
renderBoard(gameState, winningCells = [], opponentCells = []) {
    this.currentGameState = gameState;
    this.gameBoard.innerHTML = '';

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            const cell = document.createElement('div');
            cell.classList.add('cell');
            cell.dataset.row = row;
            cell.dataset.col = col;

            // Region-Farbe zuweisen
            const region = this.getRegionForCell(row, col);
            if (region) {
                cell.classList.add(region.color);
                cell.title = `Region: ${region.name}`;
            }

            // Prüfe ob diese Zelle zum größten Gebiet gehört
            const isWinningCell = winningCells.some(c => c.row === row && c.col
            === col);
            const isOpponentCell = opponentCells.some(c => c.row === row && c.c
            ol === col);

            if (isWinningCell) {
                cell.classList.add('winning-area-cell');
                cell.classList.add('largest-area');
            } else if (isOpponentCell && gameState.status === 'finished') {
                cell.classList.add('opponent-area-cell');
            }

            // Region-Status für komplett gefüllte Regionen
            const regionStatus = this.calculateRegionStatus(gameState.board);
            if (region && regionStatus[region.id].complete) {
                cell.classList.add('region-complete');
            }

            const piece = gameState.board[row][col];
            if (piece) {
                const pieceElement = document.createElement('div');
                pieceElement.classList.add('piece', piece);
            }
        }
    }
}
```

```

        // Hervorhebung für Steine im größten Gebiet
        if (isWinningCell) {
            pieceElement.classList.add('winning-piece');
            pieceElement.classList.add('largest-area-piece');
        }

        cell.appendChild(pieceElement);
    }

    // Nur klickbar wenn Spieler am Zug ist
    if (gameState.status === 'playing' &&
        this.playerColor === gameState.currentPlayer) {
        cell.addEventListener('click', () => this.handleCellClick(row,
col));
    }

    this.gameBoard.appendChild(cell);
}
}

// Erweiterte makeMove Methode:
async makeMove(row, col) {
    try {
        const response = await fetch(` ${API_BASE}/games/${this.gameId}/move` , {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerId: this.playerId,
                row: row,
                col: col
            })
        });
    }

    const data = await response.json();

    if (data.success) {
        // Bei größtem Gebiet-Sieg die Gewinner-Zellen übergeben
        if (data.winCondition && data.winCondition.startsWith('groesstes_ge
biet_')) {
            this.renderBoard(data.game, data.winningCells || [], data.oppon
entCells || []);
        } else {
            this.renderBoard(data.game);
        }
    }
}

```

```

        this.updateStatus(data.game);

        if (data.gameFinished) {
            if (this.pollInterval) {
                clearInterval(this.pollInterval);
                this.pollInterval = null;
            }

            if (data.winCondition && data.winCondition.startsWith('groesste
s_gebiet_')) {
                const size = data.largestAreaSize || data.winCondition.spli
t('_')[2];
                if (data.winner === this.playerColor) {
                    this.showVictoryMessage(`Größtes Gebiet (${size} Felde
r)`);
                } else if (data.winner === 'draw') {
                    this.showDrawMessage(size);
                } else {
                    this.showDefeatMessage(data.winner, `Größtes Gebiet
(${size} Felder)`);
                }
            } else {
                if (data.winner === this.playerColor) {
                    this.showVictoryMessage(data.winCondition);
                } else if (data.winner === 'draw') {
                    this.showDrawMessage();
                } else {
                    this.showDefeatMessage(data.winner, data.winCondition);
                }
            }
        } else {
            this.showError('Zug fehlgeschlagen: ' + data.error);
        }
    } catch (error) {
        this.showError('Verbindungsfehler: ' + error.message);
    }
}

// Erweiterte updateGameState Methode:
async updateGameState() {
    try {
        const response = await fetch(`${API_BASE}/games/${this.gameId}`);
        const data = await response.json();

        if (data.success && this.currentGameState) {

```

```

        if (JSON.stringify(data.game) !== JSON.stringify(this.currentGameState)) {
            // Bei beendetem Spiel mit größtem Gebiet, Gewinner-Zellen anzeigen
            if (data.game.status === 'finished' &&
                data.game.winCondition &&
                data.game.winCondition.startsWith('groesstes_gebiet_')) {

                // Für bereits beendete Spiele müssen wir die Gebietsdaten extra abrufen
                const areaResult = await this.getLargestAreaData();
                if (areaResult) {
                    this.renderBoard(data.game, areaResult.winningCells, areaResult.opponentCells);
                } else {
                    this.renderBoard(data.game);
                }
            } else {
                this.renderBoard(data.game);
            }
            this.updateStatus(data.game);
        }
    }

} catch (error) {
    console.error('Fehler beim Aktualisieren des Spielzustands:', error);
}
}

// Neue Methode zum Abrufen der Gebietsdaten für beendete Spiele
async getLargestAreaData() {
    try {
        // Hier könnten wir einen speziellen Endpoint für Gebietsdaten hinzufügen
        // Für jetzt simulieren wir das durch erneutes Berechnen auf Client-Seite
        if (this.currentGameState && this.currentGameState.status === 'finished') {
            // In einer echten Implementierung würde hier ein API-Call stehen
            return null;
        }
    } catch (error) {
        console.error('Fehler beim Laden der Gebietsdaten:', error);
    }
    return null;
}
}

```

CSS-Erweiterung für Gebietshervorhebung

css

```
/* Füge diese CSS-Regeln hinzu */

/* Hervorhebung für das größte Gebiet */
.largest-area {
    position: relative;
    z-index: 2;
}

.winning-area-cell {
    background: linear-gradient(135deg, #fff9c4, #fff59d) !important;
    border: 3px solid #ffd54f !important;
    box-shadow: 0 0 20px rgba(255, 213, 79, 0.7) !important;
    animation: area-glow 2s ease-in-out infinite alternate;
}

.opponent-area-cell {
    opacity: 0.7;
    filter: grayscale(0.3);
}

.largest-area-piece {
    animation: piece-pulse 1.5s ease-in-out infinite;
    border: 3px solid #ffd54f !important;
    box-shadow: 0 0 15px rgba(255, 213, 79, 0.8) !important;
}

/* Spezielle Animationen für Gebietshervorhebung */
@keyframes area-glow {
    0% {
        box-shadow: 0 0 10px rgba(255, 213, 79, 0.5);
        transform: scale(1);
    }
    100% {
        box-shadow: 0 0 25px rgba(255, 213, 79, 0.9);
        transform: scale(1.02);
    }
}

@keyframes piece-pulse {
    0% {
        transform: scale(1);
        box-shadow: 0 0 10px rgba(255, 213, 79, 0.6);
    }
}
```

```
50% {
    transform: scale(1.1);
    box-shadow: 0 0 20px rgba(255, 213, 79, 1);
}
100% {
    transform: scale(1);
    box-shadow: 0 0 10px rgba(255, 213, 79, 0.6);
}
}

/* Verbindungslinien zwischen Gebiets-Steinen (optional) */
.winning-area-cell::before {
    content: '';
    position: absolute;
    top: 50%;
    left: 50%;
    width: 120%;
    height: 120%;
    background: radial-gradient(circle, rgba(255,213,79,0.2) 0%, rgba(255,213,79,0) 70%);
    border-radius: 50%;
    transform: translate(-50%, -50%);
    z-index: -1;
    animation: ripple 3s ease-out infinite;
}

@keyframes ripple {
    0% {
        width: 100%;
        height: 100%;
        opacity: 0.8;
    }
    100% {
        width: 150%;
        height: 150%;
        opacity: 0;
    }
}

/* Status-Anzeige für Gebietsinformation */
.area-info {
    position: absolute;
    top: -10px;
    right: -10px;
    background: #ffd54f;
    color: #333;
    border-radius: 50%;
```

```

width: 20px;
height: 20px;
font-size: 10px;
display: flex;
align-items: center;
justify-content: center;
font-weight: bold;
z-index: 3;
}

```

Zusätzliche Verbesserung: Gebiets-Info im Chat

javascript

```

// In der addSystemMessage Methode, füge Gebietsinformation hinzu:
addSystemMessage(text, isAreaInfo = false) {
    const systemMessage = {
        playerId: 'system',
        message: text,
        timestamp: new Date(),
        isAreaInfo: isAreaInfo
    };

    const messageElement = this.createMessageElement(systemMessage);

    if (isAreaInfo) {
        messageElement.classList.add('area-info-message');
    }

    this.chatMessages.appendChild(messageElement);
    this.scrollChatToBottom();
}

// In der createMessageElement Methode:
createMessageElement(message) {
    const messageDiv = document.createElement('div');

    if (message.isAreaInfo) {
        messageDiv.className = 'chat-message area-info-message';
        messageDiv.innerHTML = `
            <div class="message-header">🏆 Gebiets-Info</div>
            <div class="message-text">${this.escapeHtml(message.message)}</div>
        `;
    } else if (message.playerId === 'system') {
        messageDiv.className = 'chat-message system-message';
        messageDiv.innerHTML = `

```

```

<div class="message-header">System</div>
<div class="message-text">${this.escapeHtml(message.message)}</div>
`;
} else {
    // ... bestehende Implementierung
}

return messageDiv;
}

```

CSS für Gebiets-Info Nachrichten

css

```

.area-info-message {
    background: linear-gradient(135deg, #fff9c4, #ffecb3) !important;
    border-left: 4px solid #ffd54f !important;
}

.area-info-message .message-header {
    color: #ff6f00 !important;
    font-weight: bold;
}

```

🎯 Ergebnis:

Jetzt wird das größte zusammenhängende Gebiet visuell hervorgehoben:

- 🌟 **Goldener Glow-Effekt** um die Zellen des größten Gebiets
- ⌚ **Pulsierende Animation** der Steine im Gewinner-Gebiet
- ⚡ **Deutlicher Kontrast** zum Rest des Brettes
- 📢 **Automatische Systemnachricht** mit Gebietsgröße
- 🎨 **Elegante Animationen** für ein professionelles Erscheinungsbild

Die Spieler können jetzt sofort erkennen, welches Gebiet den Sieg gebracht hat! 🏆

Integriere bitte einen KI-Gegner für Einzelspieler-Modus

KI-Gegner für Einzelspieler-Modus

Ich implementiere einen KI-Gegner, der gegen menschliche Spieler antreten kann.

Server-Erweiterung (server.js)

javascript

```
// KI-Klasse für den Computer-Gegner
class ComputerPlayer {
    constructor(difficulty = 'medium') {
        this.difficulty = difficulty;
        this.name = `KI-${difficulty.charAt(0).toUpperCase() + difficulty.slice(1)}`;
    }

    // Hauptmethode für KI-Zug
    makeMove(board, playerColor) {
        console.log(`KI (${this.difficulty}) denkt nach...`);

        switch (this.difficulty) {
            case 'easy':
                return this.makeEasyMove(board);
            case 'medium':
                return this.makeMediumMove(board, playerColor);
            case 'hard':
                return this.makeHardMove(board, playerColor);
            default:
                return this.makeMediumMove(board, playerColor);
        }
    }

    // Einfacher KI-Zug: Zufällige valide Position
    makeEasyMove(board) {
        const validMoves = this.getValidMoves(board);
        return validMoves[Math.floor(Math.random() * validMoves.length)];
    }

    // Mittlerer KI-Zug: Strategische Entscheidungen
    makeMediumMove(board, playerColor) {
        const validMoves = this.getValidMoves(board);

        // 1. Prüfe auf sofortigen Sieg
        const winningMove = this.findWinningMove(board, playerColor, validMoves);
        if (winningMove) return winningMove;
    }
}
```

```

    // 2. Blockiere gegnerischen Sieg
    const opponentColor = playerColor === 'white' ? 'black' : 'white';
    const blockingMove = this.findWinningMove(board, opponentColor, validMoves);
    if (blockingMove) return blockingMove;

    // 3. Strategische Position wählen
    const strategicMove = this.findStrategicMove(board, playerColor, validMoves);
    if (strategicMove) return strategicMove;

    // 4. Fallback: Zufälliger Zug
    return validMoves[Math.floor(Math.random() * validMoves.length)];
}

// Schwerer KI-Zug: Minimax-Algorithmus
makeHardMove(board, playerColor) {
    const validMoves = this.getValidMoves(board);

    // Für die ersten paar Züge verwende mittlere Strategie (Performance)
    if (this.countPieces(board) < 4) {
        return this.makeMediumMove(board, playerColor);
    }

    // Verwende Minimax mit begrenzter Tiefe
    let bestScore = -Infinity;
    let bestMove = validMoves[0];

    for (const move of validMoves) {
        // Simuliere Zug
        const newBoard = this.cloneBoard(board);
        newBoard[move.row][move.col] = playerColor;

        // Bewertung mit Minimax
        const score = this.minimax(newBoard, 2, false, playerColor);

        if (score > bestScore) {
            bestScore = score;
            bestMove = move;
        }
    }

    return bestMove;
}

// Minimax-Algorithmus mit Alpha-Beta-Pruning
minimax(board, depth, isMaximizing, playerColor, alpha = -Infinity, beta =

```

```

 {
    const opponentColor = playerColor === 'white' ? 'black' : 'white';
    const currentColor = isMaximizing ? playerColor : opponentColor;

    // Blattevaluation oder maximale Tiefe erreicht
    if (depth === 0 || this.isBoardFull(board)) {
        return this.evaluateBoard(board, playerColor);
    }

    const validMoves = this.getValidMoves(board);

    if (isMaximizing) {
        let maxEval = -infinity;
        for (const move of validMoves) {
            const newBoard = this.cloneBoard(board);
            newBoard[move.row][move.col] = playerColor;

            const evaluation = this.minimax(newBoard, depth - 1, false, playerColor, alpha, beta);
            maxEval = Math.max(maxEval, evaluation);
            alpha = Math.max(alpha, evaluation);

            if (beta <= alpha) break; // Alpha-Beta Pruning
        }
        return maxEval;
    } else {
        let minEval = infinity;
        for (const move of validMoves) {
            const newBoard = this.cloneBoard(board);
            newBoard[move.row][move.col] = opponentColor;

            const evaluation = this.minimax(newBoard, depth - 1, true, playerColor, alpha, beta);
            minEval = Math.min(minEval, evaluation);
            beta = Math.min(beta, evaluation);

            if (beta <= alpha) break; // Alpha-Beta Pruning
        }
        return minEval;
    }
}

// Bewertungsfunktion für das Brett
evaluateBoard(board, playerColor) {
    const opponentColor = playerColor === 'white' ? 'black' : 'white';
    let score = 0;
}

```

```

    // Bewerte nach verschiedenen Kriterien
    score += this.evaluateConnectedAreas(board, playerColor) * 10;
    score -= this.evaluateConnectedAreas(board, opponentColor) * 10;
    score += this.evaluatePotentialWins(board, playerColor) * 5;
    score -= this.evaluatePotentialWins(board, opponentColor) * 5;
    score += this.evaluateCenterControl(board, playerColor) * 2;

    return score;
}

// Bewertung von zusammenhängenden Gebieten
evaluateConnectedAreas(board, color) {
    const visited = Array(6).fill().map(() => Array(6).fill(false));
    let totalAreaSize = 0;
    let largestArea = 0;

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (!visited[row][col] && board[row][col] === color) {
                const area = this.floodFillArea(board, row, col, color, visited);
                totalAreaSize += area.size;
                largestArea = Math.max(largestArea, area.size);
            }
        }
    }
}

return largestArea + (totalAreaSize * 0.1);
}

// Bewertung von potenziellen Siegmöglichkeiten
evaluatePotentialWins(board, color) {
    let potential = 0;

    // Prüfe fast-vollständige 3x2/2x3 Rechtecke
    potential += this.countAlmostCompleteRectangles(board, color) * 3;

    // Prüfe fast-vollständige Regionen
    potential += this.countAlmostCompleteRegions(board, color) * 2;

    // Prüfe fast-5er Reihen
    potential += this.countAlmostFiveInRow(board, color) * 4;

    return potential;
}

// Zähle fast-vollständige Rechtecke

```

```

countAlmostCompleteRectangles(board, color) {
    let count = 0;

    // Prüfe 3x2 Rechtecke
    for (let startRow = 0; startRow <= 3; startRow++) {
        for (let startCol = 0; startCol <= 4; startCol++) {
            let sameColor = 0;
            let empty = 0;

            for (let row = startRow; row < startRow + 3; row++) {
                for (let col = startCol; col < startCol + 2; col++) {
                    if (board[row][col] === color) sameColor++;
                    else if (board[row][col] === null) empty++;
                }
            }

            if (sameColor === 5 && empty === 1) count++;
        }
    }

    // Prüfe 2x3 Rechtecke
    for (let startRow = 0; startRow <= 4; startRow++) {
        for (let startCol = 0; startCol <= 3; startCol++) {
            let sameColor = 0;
            let empty = 0;

            for (let row = startRow; row < startRow + 2; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] === color) sameColor++;
                    else if (board[row][col] === null) empty++;
                }
            }

            if (sameColor === 5 && empty === 1) count++;
        }
    }

    return count;
}

// Zähle fast-vollständige Regionen
countAlmostCompleteRegions(board, color) {
    const regions = [
        { rows: [0, 1], cols: [0, 1] }, { rows: [0, 1], cols: [2, 3] }, {
        rows: [0, 1], cols: [4, 5] },
        { rows: [2, 3], cols: [0, 1] }, { rows: [2, 3], cols: [2, 3] }, {
        rows: [2, 3], cols: [4, 5] },

```

```

        { rows: [4, 5], cols: [0, 1] }, { rows: [4, 5], cols: [2, 3] }, {
rows: [4, 5], cols: [4, 5] }

    };

    let count = 0;

    for (const region of regions) {
        let sameColor = 0;
        let empty = 0;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] === color) sameColor++;
                else if (board[row][col] === null) empty++;
            }
        }

        if (sameColor === 3 && empty === 1) count++;
    }

    return count;
}

// Zähle fast-5er Reihen
countAlmostFiveInRow(board, color) {
    const directions = [
        { dr: 0, dc: 1 }, { dr: 1, dc: 0 }, { dr: 1, dc: 1 }, { dr: 1, dc:
-1 }
    ];

    let count = 0;

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            for (const dir of directions) {
                let sequence = [];
                let r = row, c = col;

                // Sammle Sequenz
                while (r >= 0 && r < 6 && c >= 0 && c < 6 && sequence.length
h < 5) {
                    sequence.push({ row: r, col: c, value: board[r][c] });
                    r += dir.dr;
                    c += dir.dc;
                }

                if (sequence.length === 5) {

```

```

        const colorCount = sequence.filter(cell => cell.value =
== color).length;

        const emptyCount = sequence.filter(cell => cell.value =
== null).length;

        if (colorCount === 4 && emptyCount === 1) {
            count++;
        }
    }
}

return count;
}

// Bewertung der Zentrumskontrolle
evaluateCenterControl(board, color) {
    const centerPositions = [
        { row: 2, col: 2 }, { row: 2, col: 3 },
        { row: 3, col: 2 }, { row: 3, col: 3 }
    ];

    let centerControl = 0;

    for (const pos of centerPositions) {
        if (board[pos.row][pos.col] === color) {
            centerControl += 2;
        } else if (board[pos.row][pos.col] === null) {
            centerControl += 0.5; // Potenzielle Kontrolle
        }
    }

    return centerControl;
}

// Hilfsmethoden
getValidMoves(board) {
    const moves = [];
    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            if (board[row][col] === null) {
                moves.push({ row, col });
            }
        }
    }
    return moves;
}

```

```

        }

    findWinningMove(board, color, validMoves) {
    for (const move of validMoves) {
        const testBoard = this.cloneBoard(board);
        testBoard[move.row][move.col] = color;

        if (this.checkWinForColor(testBoard, color)) {
            return move;
        }
    }
    return null;
}

findStrategicMove(board, color, validMoves) {
    // Bevorzuge Zentrum und Eckpositionen
    const centerPositions = [
        { row: 2, col: 2 }, { row: 2, col: 3 }, { row: 3, col: 2 }, { row:
3, col: 3 }
    ];

    const cornerPositions = [
        { row: 0, col: 0 }, { row: 0, col: 5 }, { row: 5, col: 0 }, { row:
5, col: 5 }
    ];

    // Prüfe zuerst Zentrum
    for (const pos of centerPositions) {
        if (board[pos.row][pos.col] === null) {
            return pos;
        }
    }

    // Dann Ecken
    for (const pos of cornerPositions) {
        if (board[pos.row][pos.col] === null) {
            return pos;
        }
    }

    // Dann andere Positionen
    return null;
}

checkWinForColor(board, color) {
    // Vereinfachte Win-Condition Prüfung
    return this.check3x3Or2x3ForColor(board, color) ||
}

```

```

        this.checkFiveInRowForColor(board, color) ||
        this.checkRegionCoverageForColor(board, color);
    }

    check3x2Or2x3ForColor(board, color) {
        // Implementierung ähnlich wie in Game-Klasse
        for (let startRow = 0; startRow <= 3; startRow++) {
            for (let startCol = 0; startCol <= 4; startCol++) {
                let allSameColor = true;
                for (let row = startRow; row < startRow + 3; row++) {
                    for (let col = startCol; col < startCol + 2; col++) {
                        if (board[row][col] !== color) {
                            allSameColor = false;
                            break;
                        }
                    }
                    if (!allSameColor) break;
                }
                if (allSameColor) return true;
            }
        }
    }

    for (let startRow = 0; startRow <= 4; startRow++) {
        for (let startCol = 0; startCol <= 3; startCol++) {
            let allSameColor = true;
            for (let row = startRow; row < startRow + 2; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] !== color) {
                        allSameColor = false;
                        break;
                    }
                }
            }
            if (!allSameColor) break;
        }
        if (allSameColor) return true;
    }
}

return false;
}

checkFiveInRowForColor(board, color) {
    const directions = [
        { dr: 0, dc: 1 }, { dr: 1, dc: 0 }, { dr: 1, dc: 1 }, { dr: 1, dc:
-1 }
];
}

```

```

        for (let row = 0; row < 6; row++) {
            for (let col = 0; col < 6; col++) {
                if (board[row][col] === color) {
                    for (const dir of directions) {
                        let count = 1;
                        let r = row + dir.dr, c = col + dir.dc;

                        while (r >= 0 && r < 6 && c >= 0 && c < 6 && board[r]
[c] === color) {
                            count++;
                            r += dir.dr;
                            c += dir.dc;
                        }
                    }

                    if (count >= 5) return true;
                }
            }
        }
    }

    return false;
}

checkRegionCoverageForColor(board, color) {
    const regions = [
        { rows: [0, 1], cols: [0, 1] }, { rows: [0, 1], cols: [2, 3] }, {
rows: [0, 1], cols: [4, 5] },
        { rows: [2, 3], cols: [0, 1] }, { rows: [2, 3], cols: [2, 3] }, {
rows: [2, 3], cols: [4, 5] },
        { rows: [4, 5], cols: [0, 1] }, { rows: [4, 5], cols: [2, 3] }, {
rows: [4, 5], cols: [4, 5] }
    ];

    let coveredRegions = 0;

    for (const region of regions) {
        let regionComplete = true;
        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] !== color) {
                    regionComplete = false;
                    break;
                }
            }
            if (!regionComplete) break;
        }
        if (regionComplete) coveredRegions++;
    }
}

```

```

        return coveredRegions >= 2;
    }

    floodFillArea(board, startRow, startCol, targetColor, visited) {
        const directions = [{ dr: -1, dc: 0 }, { dr: 1, dc: 0 }, { dr: 0, dc:
-1 }, { dr: 0, dc: 1 }];
        const stack = [{ row: startRow, col: startCol }];
        let size = 0;
        const cells = [];

        while (stack.length > 0) {
            const { row, col } = stack.pop();

            if (row < 0 || row >= 6 || col < 0 || col >= 6) continue;
            if (visited[row][col]) continue;
            if (board[row][col] !== targetColor) continue;

            visited[row][col] = true;
            size++;
            cells.push({ row, col });

            for (const dir of directions) {
                stack.push({ row: row + dir.dr, col: col + dir.dc });
            }
        }

        return { size, cells };
    }

    cloneBoard(board) {
        return board.map(row => [...row]);
    }

    countPieces(board) {
        let count = 0;
        for (let row = 0; row < 6; row++) {
            for (let col = 0; col < 6; col++) {
                if (board[row][col] !== null) count++;
            }
        }
        return count;
    }

    isBoardFull(board) {
        for (let row = 0; row < 6; row++) {
            for (let col = 0; col < 6; col++) {

```

```

        if (board[row][col] === null) return false;
    }
}
return true;
}

// Erweiterte Game-Klasse für KI-Support
class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
        this.currentPlayer = 'white';
        this.status = 'waiting';
        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
        this.computerPlayer = null; // NEU: KI-Spieler
        this.gameMode = 'multiplayer'; // NEU: 'multiplayer' oder 'singleplaye
r'
    }
}

// NEU: Spielmodus setzen
setGameMode(mode, difficulty = 'medium') {
    this.gameMode = mode;
    if (mode === 'singleplayer') {
        this.computerPlayer = new ComputerPlayer(difficulty);
    }
}

// NEU: KI-Zug ausführen
makeComputerMove() {
    if (!this.computerPlayer || this.status !== 'playing') {
        return null;
    }

    const move = this.computerPlayer.makeMove(this.board, this.currentPlaye
r);

    if (move) {
        console.log(`KI setzt auf Position (${move.row}, ${move.col})`);
        return this.makeMove('computer', move.row, move.col);
    }
}

```

```

        return null;
    }

    // Geänderte addPlayer Methode für Einzelspieler
    addPlayer(playerId, isSinglePlayer = false) {
        if (!this.players.white) {
            this.players.white = playerId;

            // Bei Einzelspieler: KI als schwarzen Spieler hinzufügen
            if (isSinglePlayer) {
                this.players.black = 'computer';
                this.status = 'playing';
                this.setGameMode('singleplayer', 'medium');
            }
        }

        return 'white';
    } else if (!this.players.black && !isSinglePlayer) {
        this.players.black = playerId;
        this.status = 'playing';
        return 'black';
    }
    return null;
}

// Geänderte makeMove Methode für KI
makeMove(playerId, row, col) {
    if (this.status !== 'playing') {
        return { success: false, error: 'Spiel ist nicht aktiv' };
    }

    // Bei KI-Zug playerId ignorieren
    const isComputerMove = playerId === 'computer';
    const playerColor = isComputerMove ? this.currentPlayer : this.getPlayerColor(playerId);

    if (!isComputerMove && playerColor !== this.currentPlayer) {
        return { success: false, error: 'Nicht dein Zug' };
    }

    if (this.board[row][col] !== null) {
        return { success: false, error: 'Feld bereits belegt' };
    }

    this.board[row][col] = this.currentPlayer;

    // Prüfe Siegbedingungen
}

```

```
const winCondition = this.checkWinConditions(this.currentPlayer);

if (winCondition) {
    this.status = 'finished';

    if (winCondition.startsWith('groesstes_gebiet_')) {
        const largestAreaResult = this.findLargestConnectedArea();

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: largestAreaResult.winner,
            winCondition: winCondition,
            largestAreaSize: largestAreaResult.size,
            winningCells: largestAreaResult.winningCells,
            opponentCells: largestAreaResult.opponentCells,
            isComputerMove: isComputerMove
        };
    } else {
        this.winner = this.currentPlayer;
        this.winCondition = winCondition;

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: true,
            winner: this.currentPlayer,
            winCondition: winCondition,
            isComputerMove: isComputerMove
        };
    }
}

if (this.isBoardFull()) {
    this.status = 'finished';
    this.winner = 'draw';
    this.winCondition = 'unentschieden';

    return {
        success: true,
        game: this.getGameState(),
        gameFinished: true,
        winner: 'draw',
        winCondition: 'unentschieden',
        isComputerMove: isComputerMove
    };
}
```

```
        }

        // Spieler wechseln
        this.currentPlayer = this.currentPlayer === 'white' ? 'black' : 'white';

        return {
            success: true,
            game: this.getGameState(),
            gameFinished: false,
            isComputerMove: isComputerMove
        };
    }

    // Rest der Game-Klasse bleibt gleich...
    // (checkWinConditions, findLargestConnectedArea, etc.)
}

// NEUE API-Endpoints für Einzelspieler
app.post('/api/games/singleplayer', (req, res) => {
    const { difficulty = 'medium' } = req.body;

    const game = new Game();
    game.setGameMode('singleplayer', difficulty);
    games.set(game.id, game);

    console.log(`Neues Einzelspieler-Spiel erstellt: ${game.id} (Schwierigkeit: ${difficulty})`);

    res.json({
        success: true,
        gameId: game.id,
        message: 'Einzelspieler-Spiel erstellt',
        difficulty: difficulty
    });
});

app.post('/api/games/:gameId/computer/move', (req, res) => {
    const { gameId } = req.params;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }

    const game = games.get(gameId);
```

```

if (game.gameMode !== 'singleplayer') {
    return res.status(400).json({ success: false, error: 'Nur im Einzelspieler-Modus verfügbar' });
}

if (game.status !== 'playing') {
    return res.status(400).json({ success: false, error: 'Spiel ist nicht aktiv' });
}

// KI-Zug ausführen
const result = game.makeComputerMove();

if (result) {
    res.json(result);
} else {
    res.status(400).json({ success: false, error: 'KI konnte keinen Zug machen' });
}
});
```

Client-Erweiterung (public/index.html)

javascript

```

// In der GameClient-Klasse:
class GameClient {
    constructor() {
        // ... bestehende Properties
        this.gameMode = 'multiplayer'; // 'multiplayer' oder 'singleplayer'
        this.computerDifficulty = 'medium';

        this.initializeElements();
        this.setupEventListeners();
        this.loadGames();
    }

    initializeElements() {
        // ... bestehende Elemente
        this.singleplayerBtn = document.getElementById('singleplayerBtn');
        this.difficultySelect = document.getElementById('difficultySelect');
    }

    setupEventListeners() {
        // ... bestehende Event Listener
        this.singleplayerBtn.addEventListener('click', () => this.createSingleP
```

```

layerGame());
}

// NEU: Einzelspieler-Spiel erstellen
async createSinglePlayerGame() {
    const difficulty = this.difficultySelect ? this.difficultySelect.value
    : 'medium';
    const playerName = document.getElementById('playerName').value || 'Spi
eler';

    try {
        const response = await fetch(` ${API_BASE}/games/singleplayer`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ difficulty })
        });

        const data = await response.json();

        if (data.success) {
            this.gameMode = 'singleplayer';
            this.computerDifficulty = difficulty;
            this.gameId = data.gameId;
            await this.joinSinglePlayerGame(this.gameId, playerName);
        } else {
            this.showError('Fehler beim Erstellen des Einzelspieler-Spiel
s');
        }
    } catch (error) {
        this.showError('Verbindungsfehler: ' + error.message);
    }
}

// NEU: Einzelspieler-Spiel beitreten
async joinSinglePlayerGame(gameId, playerName) {
    try {
        const response = await fetch(` ${API_BASE}/games/${gameId}/join` , {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerName,
                isSinglePlayer: true
            })
    });
}

```

```
});
```

```
const data = await response.json();
```

```
if (data.success) {
    this.playerId = data.playerId;
    this.playerColor = data.color;
    this.gameId = gameId;
    this.gameMode = 'singleplayer';
```

```
this.updatePlayerInfo();
this.startGamePolling();
this.startChatPolling();
this.renderBoard(data.gameState);
this.updateStatus(data.gameState);
```

```
this.addSystemMessage(`Einzelspieler-Modus gestartet! Schwierigkeit: ${this.computerDifficulty}`);
this.addSystemMessage(`Du spielst als ${this.playerColor === 'white' ? 'Weiß' : 'Schwarz'} gegen die KI`);
```

```
// Wenn KI beginnt, sofort KI-Zug anfordern
if (data.gameState.currentPlayer !== this.playerColor) {
    this.makeComputerMove();
}
```

```
} else {
    this.showError('Beitritt fehlgeschlagen: ' + data.error);
}
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}
```

```
// NEU: KI-Zug anfordern
```

```
async makeComputerMove() {
    if (this.gameMode !== 'singleplayer' || !this.gameId) return;
```

```
try {
    // Deaktiviere das Brett während KI denkt
    this.disableBoard();
    this.addSystemMessage('KI denkt nach...', true);
```

```
const response = await fetch(`#${API_BASE}/games/${this.gameId}/computer/move`, {
    method: 'POST',
    headers: {
```

```

        'Content-Type': 'application/json'
    }
});

const data = await response.json();

if (data.success) {
    this.enableBoard();

    if (data.gameFinished) {
        this.handleGameFinished(data);
    } else {
        this.renderBoard(data.game);
        this.updateStatus(data.game);
        this.addSystemMessage(`KI hat gesetzt auf ${data.isComputerMove ? '?' : 'unbekannt'}`), true);
    }
} else {
    this.enableBoard();
    this.showError('KI-Zug fehlgeschlagen: ' + data.error);
}
} catch (error) {
    this.enableBoard();
    this.showError('Verbindungsfehler bei KI-Zug: ' + error.message);
}
}

// NEU: Brett deaktivieren/aktivieren
disableBoard() {
    const cells = document.querySelectorAll('.cell');
    cells.forEach(cell => {
        cell.style.pointerEvents = 'none';
        cell.style.opacity = '0.6';
    });
    this.statusDisplay.textContent = 'KI denkt nach...';
}

enableBoard() {
    const cells = document.querySelectorAll('.cell');
    cells.forEach(cell => {
        cell.style.pointerEvents = 'auto';
        cell.style.opacity = '1';
    });
}

// Geänderte handleCellClick für Einzelspieler
handleCellClick(row, col) {

```

```

        if (this.currentGameState.status === 'playing') {
            if (this.gameMode === 'singleplayer') {
                // Im Einzelspieler-Modus nur eigene Züge erlauben
                if (this.playerColor === this.currentGameState.currentPlayer) {
                    this.makeMove(row, col).then(() => {
                        // Nach menschlichem Zug, KI-Zug anfordern wenn Spiel noch läuft
                        if (this.currentGameState.status === 'playing' &&
                            this.currentGameState.currentPlayer !== this.playerColor) {
                            setTimeout(() => this.makeComputerMove(), 1000);
                        }
                    });
                }
            } else {
                // Multiplayer-Modus (bestehende Logik)
                if (this.playerColor === this.currentGameState.currentPlayer) {
                    this.makeMove(row, col);
                }
            }
        }
    }

    // Geänderte makeMove Methode
    async makeMove(row, col) {
        try {
            const response = await fetch(`${API_BASE}/games/${this.gameId}/move`, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    playerId: this.playerId,
                    row: row,
                    col: col
                })
            });
        }

        const data = await response.json();

        if (data.success) {
            // Render Board mit eventuellen Gewinner-Zellen
            if (data.winCondition && data.winCondition.startsWith('groesste_s_gebiet_')) {
                this.renderBoard(data.game, data.winningCells || [], data.opponentCells || []);
            }
        }
    }
}

```

```

        } else {
            this.renderBoard(data.game);
        }

        this.updateStatus(data.game);

        if (data.gameFinished) {
            this.handleGameFinished(data);
        }
    } else {
        this.showError('Zug fehlgeschlagen: ' + data.error);
    }
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}
}

// NEU: Spielende behandeln
handleGameFinished(data) {
    if (this.pollInterval) {
        clearInterval(this.pollInterval);
        this.pollInterval = null;
    }

    const isComputerWinner = data.winner && data.winner !== this.playerColor && data.winner !== 'draw';

    if (data.winCondition && data.winCondition.startsWith('groesstes_gebiet')) {
        const size = data.largestAreaSize || data.winCondition.split('_')[2];
        if (data.winner === this.playerColor) {
            this.showVictoryMessage(`Größtes Gebiet (${size} Felder)`);
        } else if (data.winner === 'draw') {
            this.showDrawMessage(size);
        } else {
            const winnerName = this.gameMode === 'singleplayer' ? 'KI' : (data.winner === 'white' ? 'Weiß' : 'Schwarz');
            this.showDefeatMessage(winnerName, `Größtes Gebiet (${size} Felder)`);
        }
    } else {
        if (data.winner === this.playerColor) {
            this.showVictoryMessage(data.winCondition);
        } else if (data.winner === 'draw') {
            this.showDrawMessage();
        } else {
    
```

```

        const winnerName = this.gameMode === 'singleplayer' ? 'KI' : (d
ata.winner === 'white' ? 'Weiß' : 'Schwarz');
        this.showDefeatMessage(winnerName, data.winCondition);
    }
}
}
}

```

HTML-Erweiterung für Einzelspieler-Buttons

html

```

<!-- Füge dies zu den Controls hinzu -->
<div class="controls">
    <button id="createGameBtn" class="btn-primary">Neues Mehrspieler-Spiel</but
ton>
    <button id="singleplayerBtn" class="btn-primary">Gegen KI spielen</button>

    <div style="display: flex; gap: 10px; align-items: center; margin-top: 10p
x;">
        <label for="difficultySelect">KI-Schwierigkeit:</label>
        <select id="difficultySelect" style="padding: 5px; border-radius: 4p
x;">
            <option value="easy">Einfach</option>
            <option value="medium" selected>Mittel</option>
            <option value="hard">Schwierig</option>
        </select>
    </div>

    <button id="refreshGamesBtn" class="btn-secondary">Spiele aktualisieren</bu
tton>
    <button id="toggleRegionsBtn" class="btn-secondary">Regionen anzeigen/verst
ecken</button>
</div>

```

CSS-Erweiterung für KI-Status

css

```

/* KI-Status Anzeige */
.ki-thinking {
    background: linear-gradient(135deg, #e3f2fd, #bbdefb) !important;
    border-left: 4px solid #2196f3 !important;
}

```

```

.computer-move {
    font-style: italic;
    opacity: 0.8;
}

/* Deaktiviertes Brett */
.board-disabled {
    opacity: 0.6;
    pointer-events: none;
}

```

Jetzt haben Sie einen vollständigen KI-Gegner mit drei Schwierigkeitsstufen!



Schwierigkeitsstufen:

- **Einfach:** Zufällige Züge
- **Mittel:** Strategische Entscheidungen (Sieg/Blockade)
- **Schwierig:** Minimax-Algorithmus mit Bewertungsfunktion

Der KI-Gegner kann alle Siegbedingungen erkennen und sowohl offensiv als auch defensiv spielen!

Wie und wann wechselt bei Einzelspieler der Spielstatus von 'waiting' auf 'playing'?

Problemanalyse: Spielstatus bei Einzelspieler

Das Problem liegt in der `addPlayer` Methode. Bei Einzelspieler wird der Status nicht korrekt auf 'playing' gesetzt.

Server-Korrektur (`server.js`)

javascript

```

// Korrigierte addPlayer Methode in der Game-Klasse:
addPlayer(playerId, isSinglePlayer = false) {
    if (!this.players.white) {
        this.players.white = playerId;

        // Bei Einzelspieler: KI als schwarzen Spieler hinzufügen und Spiel sta-
        rten
    }
}

```

```

        if (isSinglePlayer) {
            this.players.black = 'computer';
            this.status = 'playing'; // WICHTIG: Direkt auf playing setzen!
            this.setGameMode('singleplayer', 'medium');
            console.log(`Einzelspieler-Spiel gestartet: Mensch (${playerId}) vs
KI`);

            return 'white';
        } else if (!this.players.black && !isSinglePlayer) {
            this.players.black = playerId;
            this.status = 'playing';
            return 'black';
        }
        return null;
    }

    // Alternative: Spezielle Methode für Einzelspieler-Beitritt
    addSinglePlayer(playerId, difficulty = 'medium') {
        if (!this.players.white) {
            this.players.white = playerId;
            this.players.black = 'computer';
            this.status = 'playing'; // Direkt starten!
            this.setGameMode('singleplayer', difficulty);
            console.log(`Einzelspieler-Spiel gestartet: Mensch (${playerId}) vs KI
(${difficulty})`);

            return 'white';
        }
        return null;
    }
}

```

Korrigierte API-Endpoints

javascript

```

// NEU: Spezieller Endpoint für Einzelspieler-Beitritt
app.post('/api/games/singleplayer/join', (req, res) => {
    const { playerName, difficulty = 'medium' } = req.body;

    const game = new Game();
    game.setGameMode('singleplayer', difficulty);

    const playerId = uuidv4();
    const color = game.addSinglePlayer(playerId); // Verwende spezielle Method
e

```

```

if (!color) {
    return res.status(400).json({ success: false, error: 'Spieler konnte nicht hinzugefügt werden' });
}

games.set(game.id, game);

players.set(playerId, {
    id: playerId,
    name: playerName || `Spieler`,
    color: color,
    gameId: game.id
});

console.log(`Einzelspieler-Spiel ${game.id} gestartet: ${playerName} (${color}) vs KI`);

res.json({
    success: true,
    playerId: playerId,
    color: color,
    gameState: game.getGameState(),
    difficulty: difficulty
});
}
);

// ENTFERNT: Alten singleplayer Endpoint oder ändere ihn:
app.post('/api/games/singleplayer', (req, res) => {
    const { playerName, difficulty = 'medium' } = req.body;

    const game = new Game();
    game.setGameMode('singleplayer', difficulty);

    const playerId = uuidv4();
    const color = game.addSinglePlayer(playerId);

    if (!color) {
        return res.status(400).json({ success: false, error: 'Spieler konnte nicht hinzugefügt werden' });
    }

    games.set(game.id, game);

    players.set(playerId, {
        id: playerId,
        name: playerName || `Spieler`,
        color: color,
    });
}
);

```

```

        gameId: game.id
    });

    console.log(`Einzelspieler-Spiel ${game.id} gestartet: ${playerName} (${color}) vs KI`);

    res.json({
        success: true,
        gameId: game.id,
        playerId: playerId,
        color: color,
        gameState: game.getGameState(),
        message: 'Einzelspieler-Spiel erstellt und gestartet',
        difficulty: difficulty
    });
}
);

```

Client-Korrektur (public/index.html)

javascript

```

// Vereinfachte Client-Methode für Einzelspieler:
async createSinglePlayerGame() {
    const difficulty = this.difficultySelect ? this.difficultySelect.value : 'medium';
    const playerName = document.getElementById('playerName').value || 'Spieler';

    try {
        const response = await fetch(`${API_BASE}/games/singleplayer`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({
                playerName,
                difficulty
            })
        });

        const data = await response.json();

        if (data.success) {
            this.gameMode = 'singleplayer';
            this.computerDifficulty = difficulty;
            this.gameId = data.gameId;
        }
    }
}

```

```

        this.playerId = data.playerId;
        this.playerColor = data.color;

        this.updatePlayerInfo();
        this.startGamePolling();
        this.startChatPolling();
        this.renderBoard(data.gameState);
        this.updateStatus(data.gameState);

        this.addSystemMessage(`Einzelspieler-Modus gestartet! Schwierigkeit: ${this.computerDifficulty}`);
        this.addSystemMessage(`Du spielst als ${this.playerColor === 'white' ? 'Weiß' : 'Schwarz'} gegen die KI`);

        // DEBUG: Konsoleausgabe
        console.log('Spielstatus nach Erstellung:', data.gameState.status);
        console.log('Aktueller Spieler:', data.gameState.currentPlayer);

        // Wenn KI beginnt, sofort KI-Zug anfordern
        if (data.gameState.currentPlayer !== this.playerColor) {
            console.log('KI beginnt, fordere KI-Zug an...');
            setTimeout(() => this.makeComputerMove(), 1000);
        } else {
            console.log('Spieler beginnt, warte auf Eingabe...');
        }

    } else {
        this.showError('Fehler beim Erstellen des Einzelspieler-Spiels: ' + (data.error || 'Unbekannter Fehler'));
    }
} catch (error) {
    this.showError('Verbindungsfehler: ' + error.message);
}

}

// ENTFERNEN: Die joinSinglePlayerGame Methode ist nicht mehr nötig

// Debug-Erweiterung in updateStatus:
updateStatus(gameState) {
    console.log('UpdateStatus aufgerufen:', {
        status: gameState.status,
        currentPlayer: gameState.currentPlayer,
        playerColor: this.playerColor,
        gameMode: this.gameMode
    });
}

let statusText = '';

```

```

let statusClass = '';

switch (gameState.status) {
    case 'waiting':
        statusText = 'Warte auf zweiten Spieler...';
        statusClass = 'waiting';
        break;
    case 'playing':
        if (this.gameMode === 'singleplayer') {
            if (gameState.currentPlayer === this.playerColor) {
                statusText = 'Dein Zug!';
            } else {
                statusText = 'KI denkt nach...';
            }
        } else {
            const currentPlayerName = gameState.currentPlayer === 'white' ?
                'Weiß' : 'Schwarz';
            statusText = `Spieler ${currentPlayerName} ist am Zug`;
        }
        statusClass = 'playing';
        break;
    case 'finished':
        // ... bestehende finished-Logik
        break;
}

this.statusDisplay.textContent = statusText;
this.statusDisplay.className = `status ${statusClass}`;
this.gameInfo.textContent = `Spiel: ${this.gameId} | Status: ${gameState.status} | Modus: ${this.gameMode}`;
}

```

Vereinfachte Game-Klasse Konstruktor

javascript

```

class Game {
    constructor() {
        this.id = uuidv4();
        this.board = Array(6).fill().map(() => Array(6).fill(null));
        this.players = {
            white: null,
            black: null
        };
        this.currentPlayer = 'white'; // Weiß beginnt immer
        this.status = 'waiting'; // Standard: warten
    }
}

```

```

        this.winner = null;
        this.winCondition = null;
        this.createdAt = new Date();
        this.computerPlayer = null;
        this.gameMode = 'multiplayer'; // Standard: Mehrspieler
    }

    // Vereinfachte Methode nur für Einzelspieler
    addSinglePlayer(playerId, difficulty = 'medium') {
        // Setze beide Spieler gleichzeitig
        this.players.white = playerId;
        this.players.black = 'computer';
        this.status = 'playing'; // SOFORT starten!
        this.setGameMode('singleplayer', difficulty);

        console.log(`Einzelspieler-Spiel ${this.id} gestartet:`);
        console.log(`- Mensch: ${playerId} (weiß)`);
        console.log(`- KI: computer (schwarz, ${difficulty})`);
        console.log(`- Status: ${this.status}`);
        console.log(`- Startspieler: ${this.currentPlayer}`);

        return 'white'; // Mensch spielt immer weiß
    }

    setGameMode(mode, difficulty = 'medium') {
        this.gameMode = mode;
        if (mode === 'singleplayer') {
            this.computerPlayer = new ComputerPlayer(difficulty);
        } else {
            this.computerPlayer = null;
        }
    }

    // makeMove und andere Methoden bleiben gleich...
}

```

Debug-Endpoint für Entwicklung

javascript

```

// DEBUG: Endpoint zum Prüfen des Spielstatus
app.get('/api/debug/games/:gameId', (req, res) => {
    const { gameId } = req.params;

    if (!games.has(gameId)) {
        return res.status(404).json({ success: false, error: 'Spiel nicht gefunden' });
    }
});

```

```

        den' });
    }

    const game = games.get(gameId);

    res.json({
        success: true,
        game: {
            id: game.id,
            status: game.status,
            currentPlayer: game.currentPlayer,
            players: game.players,
            gameMode: game.gameMode,
            board: game.board,
            computerPlayer: game.computerPlayer ? {
                difficulty: game.computerPlayer.difficulty,
                name: game.computerPlayer.name
            } : null
        }
    });
});

```

🎯 Zusammenfassung der Lösung:

Problem: Der Status blieb auf 'waiting', weil bei Einzelspieler der zweite "Spieler" (KI) nicht als vollwertiger Spieler erkannt wurde.

Lösung:

1. `addSinglePlayer()` **Methode:** Spezielle Methode nur für Einzelspieler
2. **Sofortiger Status-Wechsel:** Direkt auf 'playing' setzen wenn beide "Spieler" vorhanden sind
3. **Vereinfachter Flow:** Ein API-Call erstellt Spiel UND tritt bei
4. **KI als 'computer' markieren:** Damit das System weiß, dass es sich um KI handelt

Flow bei Einzelspieler:

1. Client: `POST /api/games/singleplayer`
2. Server: Erstellt Game + setzt Mensch + KI + Status = 'playing'
3. Client: Erhält sofort spielbereites GameState mit Status 'playing'

4. Je nach Startspieler: Mensch oder KI beginnt

Jetzt sollte der Einzelspieler-Modus sofort nach der Erstellung mit Status 'playing' starten! 

Bitte erweitere die Methode makeHardMove um einen blockingMove. Wenn eine 3-er Reihe der gegnerischen Farbe an beiden Seiten frei ist und an einer Seite zwei freie Plätze hat, muss der angrenzende Platz der beiden freien Plätze blockiert werden.

Erweiterung der makeHardMove Methode mit Blocking-Logik

Ich erweitere die `makeHardMove` Methode um spezifische Blocking-Logik für 3er-Reihen mit freien Enden.

Server-Erweiterung (server.js)

javascript

```
// In der ComputerPlayer-Klasse:
makeHardMove(board, playerColor) {
    const validMoves = this.getValidMoves(board);

    // Für die ersten paar Züge verwende mittlere Strategie (Performance)
    if (this.countPieces(board) < 4) {
        return this.makeMediumMove(board, playerColor);
    }

    // 1. Prüfe auf sofortigen Sieg
    const winningMove = this.findWinningMove(board, playerColor, validMoves);
    if (winningMove) return winningMove;

    // 2. Blockiere gegnerischen Sieg
    const opponentColor = playerColor === 'white' ? 'black' : 'white';
    const blockingMove = this.findWinningMove(board, opponentColor, validMoves);
    if (blockingMove) return blockingMove;

    // 3. NEU: Spezielle Blocking-Logik für 3er-Reihen mit freien Enden
    const criticalBlockingMove = this.findCriticalThreeInRowBlock(board, opponentColor, validMoves);
    if (criticalBlockingMove) {
```

```

        console.log(`KI blockiert kritische 3er-Reihe von ${opponentColor}`);
        return criticalBlockingMove;
    }

    // 4. Strategische Position wählen
    const strategicMove = this.findStrategicMove(board, playerColor, validMoves);
    if (strategicMove) return strategicMove;

    // 5. Verwende Minimax mit begrenzter Tiefe als Fallback
    let bestScore = -Infinity;
    let bestMove = validMoves[0];

    for (const move of validMoves) {
        const newBoard = this.cloneBoard(board);
        newBoard[move.row][move.col] = playerColor;

        const score = this.minimax(newBoard, 2, false, playerColor);

        if (score > bestScore) {
            bestScore = score;
            bestMove = move;
        }
    }

    return bestMove;
}

// NEUE METHODE: Findet kritische 3er-Reihen die blockiert werden müssen
findCriticalThreeInRowBlock(board, opponentColor, validMoves) {
    const directions = [
        { dr: 0, dc: 1, name: 'horizontal' },      // horizontal
        { dr: 1, dc: 0, name: 'vertikal' },         // vertikal
        { dr: 1, dc: 1, name: 'diagonal_rechts' }, // diagonal \
        { dr: 1, dc: -1, name: 'diagonal_links' }  // diagonal /
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            for (const dir of directions) {
                const criticalMove = this.checkThreeInRowWithFreeEnds(
                    board, row, col, opponentColor, dir, validMoves
                );
                if (criticalMove) {
                    return criticalMove;
                }
            }
        }
    }
}

```

```

        }

    }

    return null;
}

// NEUE METHODE: Prüft spezifisch auf 3er-Reihen mit freien Enden
checkThreeInRowWithFreeEnds(board, startRow, startCol, opponentColor, direction, validMoves) {
    const { dr, dc } = direction;

    // Prüfe ob wir eine 3er-Reihe des Gegners haben
    let threeInRow = true;
    const sequence = [];

    // Sammle die 3er-Sequenz
    for (let i = 0; i < 3; i++) {
        const r = startRow + (dr * i);
        const c = startCol + (dc * i);

        // Prüfe Grenzen
        if (r < 0 || r >= 6 || c < 0 || c >= 6) {
            threeInRow = false;
            break;
        }

        sequence.push({ row: r, col: c, value: board[r][c] });
    }

    if (!threeInRow) return null;

    // Prüfe ob alle 3 Felder die gegnerische Farbe haben
    const allOpponent = sequence.every(cell => cell.value === opponentColor);
    if (!allOpponent) return null;

    // Jetzt prüfen wir die Enden der Reihe

    // Linkes/oberes Ende
    const leftEndRow = startRow - dr;
    const leftEndCol = startCol - dc;
    const leftEndValid = this.isValidPosition(leftEndRow, leftEndCol);
    const leftEndFree = leftEndValid ? board[leftEndRow][leftEndCol] === null : false;

    // Rechtes/unteres Ende
    const rightEndRow = startRow + (dr * 3);
    const rightEndCol = startCol + (dc * 3);
}

```

```

    const rightEndValid = this.isValidPosition(rightEndRow, rightEndCol);
    const rightEndFree = rightEndValid ? board[rightEndRow][rightEndCol] === null : false;

    // Prüfe spezielle Bedingung: Beide Enden frei UND mindestens eine Seite hat 2 freie Plätze
    if (leftEndFree && rightEndFree) {
        // Prüfe linke Seite auf 2 freie Plätze
        const leftSecondRow = startRow - (dr * 2);
        const leftSecondCol = startCol - (dc * 2);
        const leftSecondValid = this.isValidPosition(leftSecondRow, leftSecondCol);
        const leftSecondFree = leftSecondValid ? board[leftSecondRow][leftSecondCol] === null : false;

        // Prüfe rechte Seite auf 2 freie Plätze
        const rightSecondRow = startRow + (dr * 4);
        const rightSecondCol = startCol + (dc * 4);
        const rightSecondValid = this.isValidPosition(rightSecondRow, rightSecondCol);
        const rightSecondFree = rightSecondValid ? board[rightSecondRow][rightSecondCol] === null : false;

        console.log(`Kritische 3er-Reihe gefunden bei (${startRow},${startCol}) Richtung ${direction.name}`);
        console.log(`- Linkes Ende: (${leftEndRow},${leftEndCol}) frei: ${leftEndFree}`);
        console.log(`- Rechtes Ende: (${rightEndRow},${rightEndCol}) frei: ${rightEndFree}`);
        console.log(`- Linke 2. Position frei: ${leftSecondFree}`);
        console.log(`- Rechte 2. Position frei: ${rightSecondFree}`);

        // Blockierungsstrategie: Blockiere die Seite mit 2 freien Plätzen
        if (leftSecondFree) {
            // Blockiere das Feld direkt neben der 3er-Reihe auf der linken Seite
            const blockingMove = { row: leftEndRow, col: leftEndCol };
            if (this.isValidMove(blockingMove, validMoves)) {
                console.log(`Blockiere linkes Ende bei (${blockingMove.row},${blockingMove.col})`);
                return blockingMove;
            }
        }

        if (rightSecondFree) {
            // Blockiere das Feld direkt neben der 3er-Reihe auf der rechten Seite
        }
    }
}

```

```

        const blockingMove = { row: rightEndRow, col: rightEndCol };
        if (this.isValidMove(blockingMove, validMoves)) {
            console.log(`Blockiere rechtes Ende bei (${blockingMove.row}, ${blockingMove.col})`);
            return blockingMove;
        }
    }

    // Falls beide Seiten 2 freie Plätze haben oder keine spezifische Seite, blockiere ein Ende
    if (leftEndFree && this.isValidMove({ row: leftEndRow, col: leftEndCol }, validMoves)) {
        console.log(`Blockiere Standard linkes Ende bei (${leftEndRow}, ${leftEndCol})`);
        return { row: leftEndRow, col: leftEndCol };
    }

    if (rightEndFree && this.isValidMove({ row: rightEndRow, col: rightEndCol }, validMoves)) {
        console.log(`Blockiere Standard rechtes Ende bei (${rightEndRow}, ${rightEndCol})`);
        return { row: rightEndRow, col: rightEndCol };
    }
}

return null;
}

// NEUE METHODE: Erweiterte Blocking-Logik für verschiedene kritische Muster
findAdvancedBlockingMoves(board, opponentColor, validMoves) {
    const criticalMoves = [];

    // 1. Blockiere 3er-Reihen mit freien Enden (Hauptlogik)
    const threeInRowBlock = this.findCriticalThreeInRowBlock(board, opponentColor, validMoves);
    if (threeInRowBlock) criticalMoves.push(threeInRowBlock);

    // 2. Blockiere fast-vollständige Rechtecke
    const rectangleBlock = this.findRectangleBlockingMove(board, opponentColor, validMoves);
    if (rectangleBlock) criticalMoves.push(rectangleBlock);

    // 3. Blockiere fast-vollständige Regionen
    const regionBlock = this.findRegionBlockingMove(board, opponentColor, validMoves);
    if (regionBlock) criticalMoves.push(regionBlock);
}

```

```

// 4. Blockiere 4er-Reihen (sehr kritisch!)
const fourInRowBlock = this.findFourInRowBlock(board, opponentColor, validMoves);
if (fourInRowBlock) criticalMoves.push(fourInRowBlock);

return criticalMoves.length > 0 ? criticalMoves[0] : null;
}

// NEUE METHODE: Blockiere fast-vollständige Rechtecke
findRectangleBlockingMove(board, opponentColor, validMoves) {
    // Prüfe 3x2 Rechtecke (1 Feld fehlt)
    for (let startRow = 0; startRow <= 3; startRow++) {
        for (let startCol = 0; startCol <= 4; startCol++) {
            let opponentCount = 0;
            let emptyCell = null;

            for (let row = startRow; row < startRow + 3; row++) {
                for (let col = startCol; col < startCol + 2; col++) {
                    if (board[row][col] === opponentColor) opponentCount++;
                    else if (board[row][col] === null) emptyCell = { row, col };
                }
            }

            if (opponentCount === 5 && emptyCell && this.isValidMove(emptyCell, validMoves)) {
                console.log(`Blockiere fast-vollständiges 3x2 Rechteck bei (${emptyCell.row}, ${emptyCell.col})`);
                return emptyCell;
            }
        }
    }

    // Prüfe 2x3 Rechtecke (1 Feld fehlt)
    for (let startRow = 0; startRow <= 4; startRow++) {
        for (let startCol = 0; startCol <= 3; startCol++) {
            let opponentCount = 0;
            let emptyCell = null;

            for (let row = startRow; row < startRow + 2; row++) {
                for (let col = startCol; col < startCol + 3; col++) {
                    if (board[row][col] === opponentColor) opponentCount++;
                    else if (board[row][col] === null) emptyCell = { row, col };
                }
            }

            if (opponentCount === 5 && emptyCell && this.isValidMove(emptyCell, validMoves)) {
                console.log(`Blockiere fast-vollständiges 2x3 Rechteck bei (${emptyCell.row}, ${emptyCell.col})`);
                return emptyCell;
            }
        }
    }
}

```

```

        if (opponentCount === 5 && emptyCell && this.isValidMove(emptyCell,
validMoves)) {
            console.log(`Blockiere fast-vollständiges 2x3 Rechteck bei (${emptyCell.row},${emptyCell.col})`);
            return emptyCell;
        }
    }

    return null;
}

// NEUE METHODE: Blockiere fast-vollständige Regionen
findRegionBlockingMove(board, opponentColor, validMoves) {
    const regions = [
        { rows: [0, 1], cols: [0, 1] }, { rows: [0, 1], cols: [2, 3] }, { rows: [0, 1], cols: [4, 5] },
        { rows: [2, 3], cols: [0, 1] }, { rows: [2, 3], cols: [2, 3] }, { rows: [2, 3], cols: [4, 5] },
        { rows: [4, 5], cols: [0, 1] }, { rows: [4, 5], cols: [2, 3] }, { rows: [4, 5], cols: [4, 5] }
    ];

    for (const region of regions) {
        let opponentCount = 0;
        let emptyCell = null;

        for (const row of region.rows) {
            for (const col of region.cols) {
                if (board[row][col] === opponentColor) opponentCount++;
                else if (board[row][col] === null) emptyCell = { row, col };
            }
        }

        // Blockiere wenn 3 von 4 Feldern vom Gegner besetzt sind
        if (opponentCount === 3 && emptyCell && this.isValidMove(emptyCell, validMoves)) {
            console.log(`Blockiere fast-vollständige Region bei (${emptyCell.row},${emptyCell.col})`);
            return emptyCell;
        }
    }

    return null;
}

// NEUE METHODE: Blockiere 4er-Reihen (sehr kritisch!)

```

```

findFourInRowBlock(board, opponentColor, validMoves) {
    const directions = [
        { dr: 0, dc: 1 }, { dr: 1, dc: 0 }, { dr: 1, dc: 1 }, { dr: 1, dc: -1
    ]
    ];

    for (let row = 0; row < 6; row++) {
        for (let col = 0; col < 6; col++) {
            for (const dir of directions) {
                // Prüfe 4er-Reihe
                let fourInRow = true;
                for (let i = 0; i < 4; i++) {
                    const r = row + (dir.dr * i);
                    const c = col + (dir.dc * i);
                    if (!this.isValidPosition(r, c) || board[r][c] !== opponent
Color) {
                        fourInRow = false;
                        break;
                    }
                }
                if (fourInRow) {
                    // Prüfe Enden der 4er-Reihe
                    const leftEnd = { row: row - dir.dr, col: col - dir.dc };
                    const rightEnd = { row: row + (dir.dr * 4), col: col + (di
r.dc * 4) };

                    if (this.isValidMove(leftEnd, validMoves)) {
                        console.log(`KRITISCH: Blockiere 4er-Reihe links bei
(${leftEnd.row},${leftEnd.col})`);
                        return leftEnd;
                    }
                    if (this.isValidMove(rightEnd, validMoves)) {
                        console.log(`KRITISCH: Blockiere 4er-Reihe rechts bei
(${rightEnd.row},${rightEnd.col})`);
                        return rightEnd;
                    }
                }
            }
        }
    }

    return null;
}

// HILFSMETHODEN
isValidPosition(row, col) {

```

```
        return row >= 0 && row < 6 && col >= 0 && col < 6;
    }

    isValidMove(move, validMoves) {
        return validMoves.some(validMove =>
            validMove.row === move.row && validMove.col === move.col
        );
    }

    // Aktualisierte makeHardMove mit erweiterter Blocking-Logik
    makeHardMove(board, playerColor) {
        const validMoves = this.getValidMoves(board);
        const opponentColor = playerColor === 'white' ? 'black' : 'white';

        // Für die ersten paar Züge verwende mittlere Strategie (Performance)
        if (this.countPieces(board) < 4) {
            return this.makeMediumMove(board, playerColor);
        }

        // 1. Prüfe auf sofortigen Sieg
        const winningMove = this.findWinningMove(board, playerColor, validMoves);
        if (winningMove) {
            console.log(`KI macht Gewinnzug bei ${winningMove.row}, ${winningMove.c`
```

