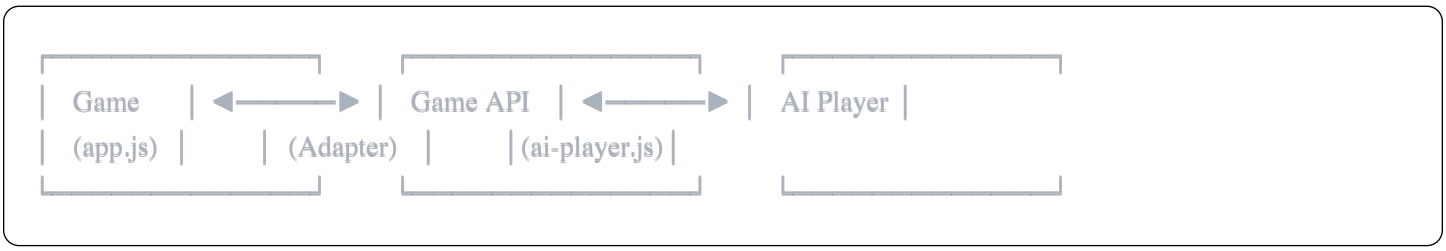
Architecture: Game ↔ AI Interaction

🎯Overview

Взаимодействие между игрой и AI построено на **паттерне Adapter** с использованием **Game API** как промежуточного слоя.



Ключевой принцип: AI **никогда не обращается** к игровому коду напрямую, только через API.

📐Three-Layer Architecture

Layer 1: Game Core (app.js)

Роль: Управление состоянием игры, правила, UI

```javascript
// Game state
const Game = {
  grid: Array,
  currentPlayer: Number,
  diceValues: [Number, Number],
  gameOver: Boolean,
  // ...
};

// Game functions
function rollDice() { ... }
function placeRectangle() { ... }
function canPlaceRectangle(x, y) { ... }
```

Layer 2: Game API (ai-player.js → createGameAPI)

Роль: Адаптер между игрой и AI

```javascript
const gameAPI = createGameAPI(Game, {
  rollDice,
  placeRectangle,
  canPlaceRectangle,
  skipTurn
});
```

Layer 3: AI Player (ai-player.js → AIPlayer)

Роль: Принятие решений, выбор ходов

```javascript
const ai = new AIPlayer('easy', 2);
await ai.takeTurn(gameAPI);
```

🔄 Complete Interaction Flow

Step 1: Инициализация (при загрузке игры)

```javascript
// 1. Game инициализируется
function initializeGame() {
  setupDOMReferences();
  setupCanvas();
  setupEventListeners();

  // 2. Создаем Game API для AI
  initializeGameAPI();  // ← Создает gameAPI

  // 3. Настраиваем UI для AI
  initAIUI();  // ← Привязывает кнопки
}

// 2. Создание Game API
function initializeGameAPI() {
  gameAPI = createGameAPI(Game, {
    rollDice: rollDice,          // ← Передаем функции игры
    placeRectangle: placeRectangle,
    rotateRectangle: rotateRectangle,
    skipTurn: skipTurn,
    canPlaceRectangle: canPlaceRectangle
  });
}
```

Что происходит:

```
app.js: initializeGame()
  ↓
app.js: initializeGameAPI()
  ↓
ai-player.js: createGameAPI(Game, functions)
  ↓
Returns: gameAPI object
  ↓
app.js: gameAPI is ready
```

Step 2: Включение AI (пользователь нажимает кнопку)

```javascript
// User clicks "Enable AI"
function enableAI(difficulty = 'easy', playerId = 2) {
  // 1. Обновляем состояние AI
  AIState.enabled = true;
  AIState.playerId = playerId;
  AIState.difficulty = difficulty;

  // 2. Создаем экземпляр AI игрока
  AIState.player = new AIPlayer(difficulty, playerId, {
    moveDelay: 800,
    thinkingDelay: 400
  });

  // 3. Проверяем, нужно ли AI сейчас ходить
  checkAITurn();
}
```

Поток данных:

```
User: Click "Enable AI"
  ↓
app.js: enableAI()
  ↓
ai-player.js: new AIPlayer()
  ↓
app.js: AIState.player = ai instance
  ↓
app.js: checkAITurn()
```

Step 3: Ход игрока (человека)

```javascript
// Human player makes move
function placeRectangle() {
  // 1. Размещаем фигуру
  // ... game logic ...

  // 2. Логируем ход
  logMove('placement', { ... });

  // 3. Проверяем окончание игры
  checkGameOver();

  // 4. Переключаем игрока (если игра не окончена)
  if (!Game.gameOver) {
    switchPlayer();  // ← Здесь происходит переключение на AI
  }
}

function switchPlayer() {
  // 1. Меняем текущего игрока
  Game.currentPlayer = Game.currentPlayer === 1 ? 2 : 1;

  // 2. Обновляем UI
  updateUI();

  // 3. КРИТИЧЕСКИЙ МОМЕНТ: Проверяем, нужно ли AI ходить
  checkAITurn();  // ← Если currentPlayer == AIState.playerId
}
```

Поток:

```
Human: Places piece
  ↓
app.js: placeRectangle()
  ↓
app.js: checkGameOver()
  ↓
app.js: switchPlayer()
  ↓
app.js: checkAITurn()  ← Передаем управление AI
```

Step 4: Ход AI (автоматически)

javascript

```javascript
// app.js
async function checkAITurn() {
  // 1. Проверки безопасности
  if (!AIState.enabled) return;           // AI не включен
  if (Game.gameOver) return;              // Игра окончена
  if (AIState.isThinking) return;         // AI уже думает
  if (Game.isReplayMode) return;          // Режим воспроизведения
  if (Game.currentPlayer !== AIState.playerId) return;  // Не ход AI

  console.log('AI turn detected');

  // 2. Блокируем UI
  AIState.isThinking = true;
  disableUserInput();

  // 3. Обновляем статус
  DOM.gameStatus.textContent = `AI (Player ${AIState.playerId}) is thinking...`;

  try {
    // 4. КЛЮЧЕВОЙ МОМЕНТ: Передаем управление AI через API
    const result = await AIState.player.takeTurn(gameAPI);
    //                    ↑                  ↑
    //                AI экземпляр        Game API

    console.log('AI move completed:', result);
  } catch (error) {
    console.error('AI error:', error);
  } finally {
    // 5. Восстанавливаем UI
    AIState.isThinking = false;
    enableUserInput();
  }
}
```

Что передается AI:

```javascript
gameAPI = {
  isDiceRolled: Function,     // Проверка состояния кубиков
  getGameState: Function,     // Получение снимка игры
  rollDice: Function,         // Бросок кубиков
  placePiece: Function,       // Размещение фигуры
  skipTurn: Function          // Пропуск хода
}
```

Step 5: AI принимает решение

javascript

```javascript
// ai-player.js
class AIPlayer {
  async takeTurn(gameAPI) {
    console.log(`AI Player ${this.playerId} taking turn...`);

    // 1. Задержка (имитация "раздумий")
    await this._delay(this.options.thinkingDelay);  // 400ms

    // 2. Бросаем кубики (если не брошены)
    if (!gameAPI.isDiceRolled()) {
      console.log('AI: Rolling dice...');
      await gameAPI.rollDice();  // ← Вызов через API
      await this._delay(this.options.moveDelay);  // 800ms
    }

    // 3. Получаем состояние игры
    const gameState = gameAPI.getGameState();  // ← Запрос через API
    //    ↓
    //    { grid, width, height, diceValues, isKush, currentPlayer, validationFn }

    // 4. Выбираем куда поставить
    const placement = this.choosePlacement(gameState);

    if (!placement) {
      // Нет валидных ходов
      console.log('AI: No valid placement found, skipping turn');
      await gameAPI.skipTurn();  // ← Пропуск через API
      return { action: 'skip' };
    }

    // 5. Размещаем фигуру
    console.log(`AI: Placing at (${placement.x}, ${placement.y})`);
    await gameAPI.placePiece(placement.x, placement.y, placement.orientation);
    //          ↑
    //    Вызов через API

    return {
      action: 'place',
      x: placement.x,
      y: placement.y,
      orientation: placement.orientation
    };
  }
}
```

Поток принятия решения:

```
ai-player.js: takeTurn(gameAPI)
  ↓
ai-player.js: gameAPI.isDiceRolled()
  ↓ (если нет)
ai-player.js: gameAPI.rollDice()
  ↓ → app.js: rollDice()
  ↓
ai-player.js: gameAPI.getGameState()
  ↓ → app.js: returns Game state copy
  ↓
ai-player.js: choosePlacement(gameState)
  ↓ → _generatePossibleMoves()
  ↓ → _isValidPlacement() or validationFn()
  ↓ → _chooseRandomMove()
  ↓
ai-player.js: gameAPI.placePiece(x, y, orientation)
  ↓ → app.js: rotateRectangle() (if needed)
  ↓ → app.js: placeRectangle()
  ↓
Back to: app.js: checkAITurn()
```

🔑Key Components Deep Dive

1. Game API Factory (createGameAPI)

Назначение: Создает объект-адаптер между игрой и AI

javascript

```
function createGameAPI(Game, gameFunctions) {
  return {
    // ═══════════════════════════════════════════════════════════
    // READ OPERATIONS (AI получает информацию)
    // ═══════════════════════════════════════════════════════════

    isDiceRolled() {
      return Game.diceRolled;  // ← Прямой доступ к состоянию
    },

    getGameState() {
      return {
        grid: Game.grid.map(row => [...row]),  // ← Deep copy!
        width: Game.width,
        height: Game.height,
        diceValues: [...Game.diceValues],      // ← Copy!
        isKush: Game.isKush,
        currentPlayer: Game.currentPlayer,
        rectangleOrientation: Game.rectangleOrientation,

        // ═══════════════════════════════════════════════════════
        // VALIDATION FUNCTION (критическая часть!)
        // ═══════════════════════════════════════════════════════
        validationFn: (x, y, width, height) => {
          // AI передает: позицию (x, y) и размеры (width, height)
          // Нужно: определить правильную ориентацию и проверить валидность

          const savedOrientation = Game.rectangleOrientation;

          // Определяем ориентацию по размерам
          const dim0 = { width: Game.diceValues[0], height: Game.diceValues[1] };
          const dim1 = { width: Game.diceValues[1], height: Game.diceValues[0] };

          let neededOrientation;
          if (width === dim0.width && height === dim0.height) {
            neededOrientation = 0;
          } else if (width === dim1.width && height === dim1.height) {
            neededOrientation = 1;
          } else {
            return false;  // Невалидные размеры
          }

          // Временно устанавливаем ориентацию
          Game.rectangleOrientation = neededOrientation;
```

```
      // Используем ИГРОВУЮ функцию валидации
      const result = gameFunctions.canPlaceRectangle(x, y);
      //                    ↑
      //              Это функция из app.js!

      // Восстанавливаем ориентацию
      Game.rectangleOrientation = savedOrientation;

      return result;
    }
  };
},

// ═══════════════════════════════════════════════════════════
// WRITE OPERATIONS (AI выполняет действия)
// ═══════════════════════════════════════════════════════════

async rollDice() {
  return new Promise((resolve) => {
    gameFunctions.rollDice();  // ← Вызов игровой функции
    setTimeout(resolve, 1000);  // Ждем анимацию
  });
},

async placePiece(x, y, orientation) {
  return new Promise((resolve) => {
    // 1. Устанавливаем ориентацию (если нужно)
    if (Game.rectangleOrientation !== orientation) {
      gameFunctions.rotateRectangle();  // ← Вызов игровой функции
    }

    // 2. Устанавливаем позицию
    Game.rectanglePosition = { x, y };
    Game.isValidPlacement = true;

    // 3. Размещаем фигуру
    gameFunctions.placeRectangle();  // ← Вызов игровой функции

    setTimeout(resolve, 100);
  });
},

async skipTurn() {
  return new Promise((resolve) => {
    gameFunctions.skipTurn();  // ← Вызов игровой функции
    setTimeout(resolve, 100);
```

```
    });
  }
 };
}
```

Важные моменты:

1. **Deep Copy в getGameState():**

```
javascript

  grid: Game.grid.map(row => [...row])  // Копия, не ссылка!
```

Почему? AI не должен модифицировать игровой grid напрямую.

2. **validationFn - критическая функция:**
    - AI передает позицию и размеры
    - API определяет правильную ориентацию
    - API вызывает игровую функцию `canPlaceRectangle()`
    - AI получает результат: true/false
3. **Async/await для всех действий:**

```
javascript

  async rollDice() { ... }
  await gameAPI.rollDice();
```

Почему? Чтобы дождаться анимаций и обновления UI.

---

2. AI Decision Making Process

javascript

```
choosePlacement(gameState) {
  // 1. Генерируем все возможные ходы
  const possibleMoves = this._generatePossibleMoves(gameState);
  //   ↓
  //   [{x: 0, y: 5, orientation: 0}, {x: 1, y: 5, orientation: 0}, ...]

  if (possibleMoves.length === 0) {
    return null;  // Нет валидных ходов
  }

  // 2. Выбираем ход (для Easy - случайный)
  return this._chooseRandomMove(possibleMoves);
}

_generatePossibleMoves(gameState) {
  const { grid, width, height, diceValues, isKush, currentPlayer, validationFn } = gameState;
  const possibleMoves = [];

  // Размеры прямоугольника для обеих ориентаций
  const dim1 = { width: diceValues[0], height: diceValues[1] };
  const dim2 = { width: diceValues[1], height: diceValues[0] };

  // Пробуем обе ориентации (если не квадрат)
  const orientations = diceValues[0] === diceValues[1] ? [0] : [0, 1];

  for (let orientation of orientations) {
    const dim = orientation === 0 ? dim1 : dim2;

    // Пробуем все позиции на поле
    for (let y = 0; y <= height - dim.height; y++) {
      for (let x = 0; x <= width - dim.width; x++) {

        // КРИТИЧЕСКИЙ МОМЕНТ: Проверяем валидность
        let isValid;

        if (validationFn) {
          // Используем игровую функцию валидации через API
          isValid = validationFn(x, y, dim.width, dim.height);
          //        ↑
          //    Это вызовет canPlaceRectangle() в игре!
        } else {
          // Fallback на собственную валидацию (не используется)
          isValid = this._isValidPlacement(grid, x, y, dim.width, dim.height,
                                           currentPlayer, isKush, width, height);
        }
```

```
      }

      if (isValid) {
        possibleMoves.push({
          x: x,
          y: y,
          orientation: orientation,
          score: 0  // Для будущих улучшений
        });
      }
    }
  }
}

console.log(`AI: Found ${possibleMoves.length} possible moves`);
return possibleMoves;
}
```

Алгоритм:

```
1. Получить размеры (3×4 или 4×3)
2. Для каждой ориентации (0 и 1):
  3. Для каждой позиции y (0 до height-dim.height):
    4. Для каждой позиции x (0 до width-dim.width):
      5. Вызвать validationFn(x, y, dim.width, dim.height)
        ↓
        Game API → canPlaceRectangle(x, y)
        ↓
        Проверка правил игры
        ↓
        Возврат true/false
      6. Если true → добавить в possibleMoves
7. Вернуть список возможных ходов
```

📊 Data Flow Diagram

Complete Round Trip

COMPLETE AI TURN FLOW

1. USER ACTION OR GAME EVENT
   └→ Human places piece
      └→ app.js: placeRectangle()

2. GAME STATE UPDATE
   └→ app.js: Update Game.grid
   └→ app.js: logMove()
   └→ app.js: checkGameOver()

3. PLAYER SWITCH
   └→ app.js: switchPlayer()
      └→ Game.currentPlayer = 2 (AI)
      └→ app.js: checkAITurn()

4. AI TURN DETECTION
   └→ app.js: checkAITurn()
      ├─ Check: AIState.enabled? ✓
      ├─ Check: Game.gameOver? ✗
      ├─ Check: Game.currentPlayer == AIState.playerId? ✓
      └─ PROCEED →

5. AI INITIALIZATION
   └→ app.js: AIState.isThinking = true
   └→ app.js: disableUserInput()
   └→ app.js: Update UI: "AI thinking..."

6. AI TAKES CONTROL
   └→ app.js: await AIState.player.takeTurn(gameAPI)
      └→ ai-player.js: takeTurn(gameAPI)
         │
         ├─ Step 1: Check dice
         │ └→ gameAPI.isDiceRolled()
         │   └→ Returns: Game.diceRolled
         │
         ├─ Step 2: Roll dice (if needed)
         │ └→ gameAPI.rollDice()
         │   └→ app.js: rollDice()
         │   └→ Updates: Game.diceValues
         │   └→ Updates: Game.diceRolled = true
         │   └→ Updates: UI (dice display)
         │
         ├─ Step 3: Get game state

```
│  └→ gameAPI.getGameState()
│    └→ Returns: {
│      grid: [...],          (deep copy)
│      width: 50,
│      height: 50,
│      diceValues: [3, 4],
│      isKush: false,
│      currentPlayer: 2,
│      validationFn: Function
│    }
│
├─ Step 4: Generate possible moves
│  └→ ai-player.js: _generatePossibleMoves(gameState)
│    └→ For each position (x, y):
│      └→ For each orientation (0, 1):
│        └→ validationFn(x, y, w, h)
│          └→ Game API intercepts
│            └→ Sets Game.rectangleOrientation
│            └→ app.js: canPlaceRectangle(x, y)
│              ├─ Check bounds
│              ├─ Check adjacency
│              ├─ Check KUSH rules
│              └─ Returns: true/false
│          └→ Returns to AI
│        └→ If valid: add to possibleMoves[]
│    └→ Returns: [{x, y, orientation}, ...]
│
├─ Step 5: Choose move
│  └→ ai-player.js: choosePlacement()
│    └→ ai-player.js: _chooseRandomMove(possibleMoves)
│      └→ Returns: {x: 23, y: 15, orientation: 0}
│
└─ Step 6: Execute move
  └→ gameAPI.placePiece(23, 15, 0)
    └→ Game API intercepts
      ├─ Set Game.rectangleOrientation = 0
      ├─ Set Game.rectanglePosition = {x: 23, y: 15}
      ├─ Set Game.isValidPlacement = true
      └→ app.js: placeRectangle()
        ├─ Update Game.grid
        ├─ Check contour captures
        ├─ logMove()
        ├─ checkGameOver()
        └─ if (!gameOver): switchPlayer()
          └→ Back to human player
```

```
7. AI COMPLETES
   └→ ai-player.js: returns {action: 'place', x: 23, y: 15, orientation: 0}
   └→ app.js: console.log('AI move completed')

8. CLEANUP
   └→ app.js: AIState.isThinking = false
   └→ app.js: enableUserInput()
   └→ app.js: updateUI()

9. READY FOR NEXT TURN
   └→ If currentPlayer == 1: Human's turn
   └→ If currentPlayer == 2 && AI enabled: AI's turn (goto step 4)
```

🔒 Safety & Isolation

Принципы изоляции:

1. **AI не может модифицировать игровое состояние напрямую**

```javascript
// ❌ AI НЕ МОЖЕТ:
Game.grid[y][x] = 2;
Game.currentPlayer = 1;
Game.gameOver = true;

// ✅ AI МОЖЕТ ТОЛЬКО:
gameAPI.rollDice();
gameAPI.placePiece(x, y, orientation);
gameAPI.skipTurn();
```

2. **AI получает копии данных, не ссылки**

```javascript
getGameState() {
  return {
    grid: Game.grid.map(row => [...row]),  // Deep copy
    diceValues: [...Game.diceValues],      // Copy
    // ...
  };
}
```

3. **Все действия проходят через контролируемый API**

```javascript
// AI не вызывает функции напрямую:
// rollDice() ❌

// AI вызывает через API:
gameAPI.rollDice() ✅
// ↓
// Game API проверяет
// ↓
// Вызывает игровую функцию
// ↓
// Возвращает результат
```

4. **Валидация всегда на стороне игры**

```javascript
// AI не решает, валиден ли ход
// AI спрашивает у игры через validationFn
const isValid = validationFn(x, y, width, height);
//              ↓
//         Вызывает canPlaceRectangle() в игре
//              ↓
//         Игра применяет ВСЕ правила
//              ↓
//         Возвращает результат AI
```

🎨Design Patterns Used

1. Adapter Pattern

Game API адаптирует интерфейс игры для AI

```javascript
// Игра имеет сложный интерфейс:
Game.grid, Game.currentPlayer, rollDice(), placeRectangle(), ...

// AI работает с простым интерфейсом:
gameAPI.getGameState(), gameAPI.rollDice(), gameAPI.placePiece()

// Adapter преобразует один в другой
```

2. Facade Pattern

Game API скрывает сложность игры за простым интерфейсом

```javascript
// Вместо:
if (Game.rectangleOrientation !== orientation) rotateRectangle();
Game.rectanglePosition = {x, y};
Game.isValidPlacement = true;
placeRectangle();

// AI делает:
gameAPI.placePiece(x, y, orientation);
```

3. Strategy Pattern

AIPlayer может использовать разные стратегии (Easy/Medium/Hard)

```javascript
class AIPlayer {
  choosePlacement(gameState) {
    switch (this.difficulty) {
      case 'easy': return this._chooseRandomMove(moves);
      case 'medium': return this._chooseStrategicMove(moves);
      case 'hard': return this._chooseOptimalMove(moves);
    }
  }
}
```

4. Observer Pattern (неявный)

Game уведомляет AI через `checkAITurn()`

```javascript
function switchPlayer() {
  Game.currentPlayer = ...;
  checkAITurn();  // ← Observer notification
}
```

💡 Key Insights

Почему такая архитектура?

1. Модульность

AI можно заменить без изменения игры
Игру можно изменить без изменения AI

2. Тестируемость

AI можно тестировать отдельно с mock API
Игру можно тестировать отдельно от AI

3. Безопасность

AI не может сломать игру
AI не может обойти правила
AI не может модифицировать состояние напрямую

4. Расширяемость

Легко добавить новые уровни сложности
Легко добавить новые типы AI
Легко добавить AI vs AI режим

5. Переиспользуемость

AI модуль можно использовать в других играх
Достаточно реализовать Game API

📈Performance Considerations

Optimization Points

1. **Deep Copy в getGameState()**

```
javascript

// Копия grid 50×50 = 2500 элементов
// ~0.1ms на современном CPU
// Вызывается 1 раз за ход AI
// Приемлемо ✓
```

2. **Validation Loop**

```
javascript

// Проверка ~5000 позиций на 50×50
// ~50ms total (0.01ms на позицию)
// Блокирует UI, но с async/await не критично
// Можно оптимизировать через Web Workers
```

3. **State Restoration**

```
javascript

// В validationFn() временно меняем ориентацию
// Сохранение → Изменение → Проверка → Восстановление
// ~0.001ms на операцию
// Не влияет на производительность ✓
```

🚀Future Improvements

Planned Enhancements

1. **Caching**

```javascript
// Кэшировать возможные ходы для одинаковых состояний
const cacheKey = `${grid.hash()}-${diceValues}`;
if (moveCache[cacheKey]) return moveCache[cacheKey];
```

2. **Web Workers**

```javascript
// Вычисление ходов в фоновом потоке
const worker = new Worker('ai-worker.js');
worker.postMessage({ gameState });
worker.onmessage = (e) => placePiece(e.data.move);
```

3. **Incremental Updates**

```javascript
// Вместо полного пересканирования проверять только новые области
const changedRegion = getChangedRegion(prevGrid, currentGrid);
const moves = scanRegion(changedRegion);
```

4. **Priority Queue**

```javascript
// Для Medium/Hard AI: приоритезировать проверку лучших позиций
const positions = priorityQueue([
  { x, y, score: territoryScore(x, y) }
]);
```

✅Summary

Взаимодействие Game ↔ AI:

1. **Инициализация:** Game создает API → передает AI
2. **Активация:** User включает AI → создается AIPlayer
3. **Ход игрока:** placeRectangle() → switchPlayer() → checkAITurn()
4. **Ход AI:** takeTurn(gameAPI) → выбор хода → placePiece()
5. **Валидация:** AI спрашивает через validationFn → игра проверяет
6. **Выполнение:** API вызывает игровые функции → обновление состояния
7. **Завершение:** AI отдает управление → следующий ход

Ключевые принципы:

- ✅Изоляция (AI не трогает Game напрямую)
- ✅Контроль (все через API)
- ✅Копии данных (не ссылки)
- ✅Игровая валидация (не AI валидация)
- ✅Async/await (плавный UI)

Результат: Безопасная, модульная, расширяемая архитектура! 🎉