Relatório de Engenharia Reversa - Sistema Morelli
Aluno: Jessé R. Rogério

Metodologia Utilizada

A abordagem metodológica foi em etapas.

2.1 Setup do Ambiente

- Instalação do JDK 11 e 21 para compatibilidade entre código-fonte e servidor;
- Utilização do IntelliJ Ultimate como IDE principal por conta de *plugins*/resolução de problemas com *debugger* & mais;
- Configuração de *plugins* de análise estática (PMD [Qualidade de código], Flow [Flow chart plugin], SonarQube/Lint [Qualidade de código]);
- Ativação de *plugins* desativados previamente por conta de configuração própria, para análise completa do código.

2.2 Coleta de Informações

- Busca abrangente por documentação existente (UML, requisitos, manuais);
- Varredura por artefatos de persistência (scripts SQL, configurações de banco, .h2, .mwb, etc…);
- Análise de comentários internos e mecanismos de logging (Log4j);
- Mapeamento da estrutura de pacotes e arquitetura geral (através de visualização dos pacotes → “Project “→ “Packages” em Intellij IDEA).

2.3 Análise Estática

- Identificação de código não utilizado e redundante (com “/TODO”, “/*”, “//” e variáveis/métodos “cinzas” em Intellij IDEA & SonarQube);
- Detecção de código morto e duplicado (variáveis/métodos “cinzas” em Intellij IDEA & SonarQube);
- Avaliação de conformidade com padrões de estilo e convenções (SonarQube & PMD);
- Análise de complexidade algorítmica e dependências (SonarQube).

2.4 Modelagem e Abstração

- Criação de diagramas UML (classes, sequência, componentes) – (usando arquivos ‘.md’ com mermaid, gerados pela própria IDEA e o plugin Flow);
- Identificação das camadas arquiteturais (mermaid & .md);
- Documentação dos fluxos principais do sistema (mermaid & .md).

3. *Resultados Obtidos*

3.1 Documentação e Artefatos

- Documentação técnica: Extremamente limitada, apenas um README.txt básico
- Diagramas UML: Inexistentes no projeto original
- Requisitos formais: Não documentados
- Scripts de banco: Sistema não utiliza persistência de dados

3.2 Estrutura do Sistema

- Arquitetura: Monolítica simples sem separação clara de responsabilidades
- Pacotes principais:
- entidades: Classes de domínio (Jogador, Tabuleiro)
- interfaceGrafica: Componentes de UI (AtorJogador, TelaJogador)
- mensagens: Internacionalização (inglês/português)
- morelli: Classe principal (Main)

3.3 Tecnologias Utilizadas

- Linguagem: Java 11+
- Interface gráfica: Swing (JFrame)
- Gerenciamento de dependências: Manual (sem Maven/Gradle)
- Persistência: Estado em memória (sem banco de dados)
- Logging: Log4j (configurado)
- Servidor: Integração com NetGames para funcionalidade multiplayer

3.4 Principais Problemas Identificados

3.4.1 Código Não Utilizado e Redundante

- Variáveis não utilizadas em várias classes (JogadaMorelli, Tabuleiro, AtorJogador, Posicao)
- Métodos nunca chamados (ImagemDeTabuleiro, Tabuleiro, Posicao, AtorJogador)
- Classe ImagemDeTabuleiro inteiramente não utilizada

3.4.2 Problemas de Qualidade de Código

- Código duplicado: Métodos calcularMovimentoLinha() e calcularMovimentoColuna() com estruturas similares
- Convenções de código: Nomenclatura inconsistente, enums em minúsculo, mistura de idiomas
- Documentação: Quase inexistente, excesso de comentários TODO

3.4.3 Complexidade Algorítmica

- Alta complexidade cognitiva: Métodos verificarAdjacentes(), calcularTomadaTrono(), calcularCaptura()
- Problemas de performance: Loops aninhados em verificarAdjacentes() (potencial $O(n^3)$)

3.4.4 Problemas Arquiteturais

- Violação de separação de concerns: Tabuleiro depende diretamente de AtorJogador
- Acoplamento excessivo: Conhecimento detalhado entre componentes
- Falta de interfaces bem definidas entre camadas
- Ausência completa de camada de persistência
- Integração frágil com NetGames: Dificulta migração e manutenção

3.4.5 Complexidade Computacional (Classe Tabuleiro)

- O(1): Métodos de acesso simples e operações básicas
- O(M): limparTabuleiro() (M = número de faixas ≈ 7)
- O(N): distribuiPecas(), verificarAdjacentes(), calcularTomadaTrono() (N = posições/faixa)
- O(M*N): calcularMovimentoLinha(), calcularMovimentoColuna()

4. Soluções Propostas

4.1 Refatoração Imediata

1. Remoção de código não utilizado: Eliminar variáveis, métodos e classes não referenciados
2. Eliminação de duplicação: Criar abstração comum para calcularMovimentoLinha() e calcularMovimentoColuna()
3. Correção de convenções: Padronizar nomenclatura, corrigir erros ortográficos, usar enums em maiúsculo
4. Documentação básica: Adicionar comentários Javadoc nas classes e métodos principais

4.2 Melhorias de Arquitetura

1. Introduzir padrão MVC: Separar claramente Model, View e Controller
2. Desacoplar Tabuleiro de AtorJogador: Introduzir interface de notificação
3. Definir contratos de interface entre camadas
4. Migração do servidor NetGames: Desenvolver versão própria do servidor integrada diretamente ao código

4.3 Migração do Servidor NetGames

4.3.1 Análise da Situação Atual

O sistema depende atualmente do servidor NetGames externo para funcionalidades multiplayer, o que representa:

- Ponto único de falha externo
- Dificuldade de manutenção e evolução
- Limitações de funcionalidades específicas do jogo
- Problemas de compatibilidade futura

4.3.2 Estratégia de Migração

1. Implementar servidor embutido: Desenvolver versão simplificada do protocolo NetGames diretamente integrada
2. Manter compatibilidade: Implementar inicialmente compatibilidade com protocolo existente
3. Abstração de conexão: Criar camada de abstração para comunicação em rede
4. Migração gradual: Manter suporte ao NetGames original durante transição

4.3.3 Benefícios Esperados

- Maior controle sobre funcionalidades multiplayer
- Eliminação de dependência externa
- Melhor integração com lógica específica do jogo Morelli
- Possibilidade de adicionar features exclusivas
- Melhor performance e estabilidade

4.4 Otimizações de Performance

1. Revisar algoritmos complexos: Simplificar verificarAdjacentes() e calcularTomadaTrono()
2. Introduzir caching: Memorizar resultados de cálculos repetitivos
3. Otimizar estruturas de dados: Avaliar uso de collections mais eficientes

4.5 Funcionalidades Faltantes

1. Persistência de jogo: Salvar/recuperar estado do jogo
2. Histórico de partidas: Registrar resultados e estatísticas
3. Modo de jogo adicional: Implementar variantes ou dificuldades
4. Melhorias de UI: Modernizar interface gráfica