Relatório de Refatoração de Código com Base nos Princípios do Clean Code

Aluno: Raul Melo Farias

Prontuário: GU3046923

Disciplina: Engenharia de Software (GRUENGS) - 2025.2

Projeto: Reconhecedor de Gramáticas em C

---

1. Introdução

Este relatório documenta o processo de refatoração de um projeto acadêmico em C, originalmente desenvolvido para a disciplina de Linguagens Formais e Autômatos. O objetivo do projeto é analisar uma string de definição de uma gramática formal e validar sua estrutura.

O código original, embora funcional, apresentava oportunidades de melhoria em termos de legibilidade e manutenibilidade. O objetivo desta tarefa foi aplicar os princípios e práticas do **Clean Code (Código Limpo)**, propostos por Robert C. Martin, para aprimorar a qualidade interna do software, sem alterar seu comportamento externo. As melhorias focaram em tornar o código mais claro, modular e fácil de manter e estender no futuro.

---

2. Análise do Código Não-Refatorado (Estado Inicial)

O código original era composto por três arquivos (main.c, reconhecedor_gramatica.c, reconhecedor_gramatica.h). A principal fragilidade do design residia no arquivo reconhecedor_gramatica.c, especificamente na função analisar_gramatica.

Esta função era **monolítica**, com mais de 100 linhas de código, e violava diretamente o **Princípio da Responsabilidade Única (SRP)**. Suas responsabilidades incluíam:

- Inicializar a estrutura de dados da gramática.
- Validar a sintaxe geral da string de entrada (ex: terminação com $).
- Analisar (fazer o *parsing*) de cada regra de produção individualmente.
- Gerenciar a alocação dinâmica de memória para as listas de regras e símbolos.
- Processar e validar os símbolos do lado esquerdo (LHS) e direito (RHS) de cada regra.

- Reportar erros diretamente no console com fprintf.

Essa centralização de responsabilidades tornava a função extremamente complexa, difícil de ler, depurar e modificar. O tratamento de erros, espalhado por toda a função, ofuscava a lógica principal de análise, dificultando o entendimento do fluxo de execução em caso de sucesso.

---

3. Processo de Refatoração e Eixos do Clean Code Adotados

A refatoração foi guiada pelos eixos do Clean Code apresentados em aula. As principais mudanças foram:

3.1. Eixo: Funções e Métodos

Este foi o eixo de maior impacto na refatoração. O objetivo foi decompor a função monolítica analisar_gramatica em funções menores e coesas, cada uma com uma responsabilidade única.

- A função analisar_gramatica foi transformada em uma **função orquestradora**, de alto nível, cujo trabalho é apenas validar a entrada e coordenar a análise de cada regra em um loop.
- A lógica de inicialização foi extraída para sua própria função: void inicializar_gramatica(Grammar *gramatica);.
- As validações iniciais da string foram movidas para: static CodigoErro validar_entrada_e_definir_simbolo_inicial(...).
- A complexa lógica de análise de uma única regra foi extraída para: static CodigoErro analisar_regra_unica(...).
- Esta, por sua vez, foi suportada por funções auxiliares ainda mais específicas:
    - garantir_espaco_para_regra(): Cuida apenas da realocação de memória para as regras.
    - processar_lhs(): Valida e processa apenas o lado esquerdo de uma regra.
    - processar_rhs(): Valida e processa apenas o lado direito de uma regra.

Essa nova estrutura respeita a separação de **Níveis de Abstração**, permitindo que o código seja lido de cima para baixo, como um artigo de jornal, partindo do conceito mais geral para os detalhes mais específicos.

3.2. Eixo: Nomes Significativos

Para melhorar a legibilidade e tornar o código mais auto-documentado, foram realizadas as seguintes renomeações em variáveis de escopo amplo:

Nome Original	Novo Nome	Justificativa
Grammar *g	Grammar *gramatica	g é um nome muito curto e genérico para um objeto central no programa. gramatica revela a intenção imediatamente.
const char* input_str	const char* definicao_gramatica	A string de entrada não é uma entrada qualquer, ela contém a definição da gramática, tornando o novo nome mais preciso.

Essas simples mudanças, especialmente nas assinaturas das funções, tornam o propósito de cada função muito mais claro sem a necessidade de comentários.

3.3. Eixo: Gerenciamento de Erros

O sistema de tratamento de erros foi completamente redesenhado para separar a **detecção** do **relatório** de erros.

- No lugar de fprintf espalhados pelo código de análise, foi criado um enum chamado CodigoErro no arquivo .h. Este enum define um código único para cada tipo de falha possível.
- As funções de análise foram modificadas para retornar um CodigoErro. Em caso de sucesso, retornam SUCESSO_ANALISE.
- No arquivo main.c, foi criada uma função centralizada, imprimir_mensagem_erro(CodigoErro codigo), que é a única responsável por traduzir um CodigoErro em uma mensagem amigável para o usuário.

Essa abordagem desacopla o módulo de análise da interface do usuário, tornando-o mais coeso, reutilizável e fácil de testar.

---

4. Dificuldades Enfrentadas e Lições Aprendidas

- **Dificuldade: Refatoração sem Testes Automatizados** A maior dificuldade do processo foi a ausência de uma suíte de testes unitários automatizados. Conforme discutido na aula 11I, testes garantem a confiabilidade do código durante a refatoração. Cada mudança estrutural exigia uma nova compilação e um teste manual com o arquivo gramatica_exemplo.txt para verificar se o comportamento externo não havia sido quebrado. Esse processo é lento e suscetível a falhas, evidenciando a

importância do Desenvolvimento Guiado por Testes (TDD) para a segurança e agilidade na manutenção de software.

- **Lição Aprendida: O Impacto da Decomposição de Funções** A principal lição foi constatar o quão mais simples um problema complexo se torna quando é dividido em partes menores e bem definidas. A lógica de análise não mudou, mas sua organização em funções pequenas e com responsabilidades únicas transformou um código confuso em um fluxo de trabalho claro e gerenciável.
- **Lição Aprendida: Robustez no Tratamento de Erros** A transição para um sistema de códigos de erro (enum) tornou o programa mais robusto. Em vez de retornar um 0 genérico para qualquer falha, o sistema agora identifica precisamente a causa do erro, permitindo que a camada externa (a função main) tome decisões mais inteligentes e informe o usuário de maneira mais específica.

---

5. Conclusão

A aplicação dos princípios do Clean Code resultou em uma versão do software que, embora funcionalmente idêntica à original, possui uma qualidade interna imensamente superior. O código refatorado é mais legível, modular, e sua estrutura baseada em responsabilidades bem definidas facilita futuras manutenções ou a adição de novas funcionalidades.

Este exercício prático demonstrou que a preocupação com a clareza e a organização do código não é um luxo, mas um investimento fundamental para a longevidade e o sucesso de um projeto de software, reduzindo o chamado **Débito Técnico** e aumentando a produtividade a longo prazo.