Refactoring

Método disciplinado de limpeza de código que melhora o design de um sistema após este ter sido escrito, sem nunca alterar o seu comportamento externo. Trata-se de uma sucessão de passos mínimos que permitem transformar um design caótico em código de excelência.

O refactoring é uma resposta técnica a sinais específicos de degradação, conhecidos como "bad smells". Identificar estes gatilhos é o que separa um engenheiro de software de um mero codificador.

A distinção entre "**comportamento externo**" e "**estrutura interna**" é o pilar da confiança da equipa e da agilidade do negócio. Ao mantermos a funcionalidade intacta enquanto otimizamos a arquitetura, reduzimos o stress do processo de design.

- Funcionalidade exterior = INALTERADA
- Estrutura interna = MELHORADA
- Testes DEVEM passar
- Utilizador NÃO percebe mudança

```
❌ NÃO é adicionar features (é melhoria estrutural)
❌ NÃO é refactoring se muda comportamento externo
❌ NÃO é processo rápido e dirty (é disciplinado)
❌ NÃO é reescrever tudo de zero (é evolução)

✅ É processo disciplinado
✅ É melhoria de design post-hoc
✅ É sem alterar comportamento
✅ É série de pequenos passos
```

Problemas de Routine/Class (Rotinas e Classes)

- **Rotinas Longas e Ciclos Complexos:** Métodos extensos ou ciclos (loops) com aninhamento excessivo são focos de bugs.
- **Métodos Duplicados:** Código repetido em múltiplas localizações.
- **Baixa Coesão:** Classes que assumem demasiadas responsabilidades.
- **Interfaces Inconsistentes:** Falta de um nível de abstração coerente, dificultando o consumo da classe.
- **Listas de Parâmetros Excessivas:** Um indicador claro de que a rotina está a tentar fazer demasiado. Problemas de Relação
- **Acoplamento Íntimo:** Classes com dependência excessiva dos detalhes internos de outras.
- **Feature Envy (Inveja de Funcionalidade):** Quando uma rotina acede mais a funções de outra classe do que da sua própria.
- **Middleman (Intermediário):** Objetos que não executam trabalho real, servindo apenas para delegar chamadas (Slide 4). Problemas de Dados

- **Dados Membros Públicos:** Um "pecado arquitetural" que viola o encapsulamento (Slide 5).
- **Tramp Data (Dados Passageiros):** Cadeias de rotinas que passam dados apenas para que cheguem a um destino final, como uma encomenda que atravessa várias mãos sem nunca ser aberta até ao destino.
- **Dados Duplicados e Primitivos Overloaded:** Uso de tipos simples para conceitos que exigem classes próprias.

Problemas de Manutenção

- **Modificações Paralelas e Hierarquias de Herança:** Quando uma alteração exige mudar múltiplas classes ou hierarquias de herança em paralelo (Slide 4).
- **Speculative Generality:** Código implementado para necessidades que "poderão surgir um dia" (Slide 5). Se não é necessário hoje, é ruído.
- **Comentários como Muleta:** Se precisa de comentários para explicar código difícil, o código deve ser refatorado para ser autoexplicativo.

Princípios e Regras de Ouro do Processo de Refactoring

A metodologia deve sempre sobrepor-se à intuição. O refactoring é um processo incremental que exige rigor operacional para mitigar riscos. • **A Regra dos Dois Chapéus:** Nunca adicione funcionalidades e refatore simultaneamente. Se a estrutura atual dificulta uma nova função, primeiro refatore para abrir caminho; só depois implemente a funcionalidade. • **A Propriedade dos Pequenos Passos:** O refactoring deve ser executado em etapas mínimas. Erros e erros de lógica são inevitáveis; ao avançar em passos curtos, garantimos que qualquer bug seja "fácil de encontrar" e isolar (Slide 20). • **O Pré-requisito dos Testes:** É proibido iniciar qualquer refactoring sem uma suite de testes de verificação automática sólida. Os testes são o que garante que o comportamento externo permanece inalterado.

Técnicas de Refactoring

Abaixo, apresento as ferramentas fundamentais para a reestruturação de sistemas:

Extract Method (Extrair Método)

Transformar fragmentos de código com um propósito claro num novo método. Exemplo: Mover a lógica de cálculo de uma fatura para calcularTotal().

Move Method (Mover Método)

Relocalizar um método para a classe que ele mais utiliza. Se a lógica depende de dados da classe Rental, o método deve estar em Rental, não em Customer.

Rename Variables (Renomear Variáveis)

Fundamental para a comunicação entre humanos. Como afirma Martin Fowler (Slide 17): "Qualquer um consegue escrever código que um computador entenda; bons programadores escrevem código que humanos entendam."

Eliminar Variáveis Temporárias

Reduzir parâmetros desnecessários que poluem a lógica e dificultam o rastreio do fluxo de dados.

Inline Method/Class (Incorporar Método/Classe)

O inverso da extração. Se o corpo de uma função é tão óbvio quanto o seu nome, deve ser incorporado no chamador para reduzir a fragmentação desnecessária.

Benefícios Principais

1. **Improves Design (Melhora Design)**
    - Estrutura interna fica melhor
    - Design evolui organicamente
    - Código fica mais profissional
2. **Makes Software Easier to Understand (Mais Fácil Compreender)**
    - Métodos têm propósito claro
    - Nomes são descritivos
    - Estrutura é óbvia
    - Novo developer entende rápido
3. **Helps Find Bugs (Ajuda Encontrar Bugs)**
    - Refactoring força ler código cuidadosamente
    - Problemas lógicos aparecem
    - Edge cases identificados
    - Testes encontram bugs
4. **Helps Program Faster (Ajuda Programar Mais Rápido)**
    - Código organizado = menos tempo procurando
    - Reutilização = menos código novo
    - Menos bugs = menos debug
    - Resultado: Desenvolvimento mais rápido

INTEGRAÇÃO COMPLETA: DO REQUISITO À IMPLEMENTAÇÃO

Pipeline Processual Integrado

```
PROBLEM UNDERSTANDING
    ↓
RAS-2: REQUIREMENTS DEFINITION
├─ Functional vs Non-functional
├─ User vs System requirements
└─ Candidate requirements negotiation
    ↓
RAS-3: REQUIREMENTS ENGINEERING (7 Activities)
├─ 1. Inception (understand problem)
├─ 2. Elicitation (capture needs)
├─ 3. Elaboration (analyze)
├─ 4. Negotiation (resolve conflicts)
├─ 5. Documentation (formalize)
├─ 6. Validation (verify)
└─ 7. Management (manage changes)
    ↓
RAS-5: ELICITATION TECHNIQUES
├─ Individual: Interviews, Surveys, Introspection, Domain Analysis
├─ Groups: Brainstorming, Group Dynamics
└─ Artefacts: Prototyping, Personas, Scenarios
    ↓
RAS-6: MODELLING (UML)
├─ Domain Model (vocabulary, concepts)
├─ Use Cases (functionalities)
├─ Class Model (structure)
├─ Sequence (interactions)
├─ State (behavior)
└─ Activity (process flow)
    ↓
RAS-4: WRITING REQUIREMENTS (Natural Language)
├─ 27-section Template
├─ Technical writing guidelines
├─ Standardized formats
└─ Ambiguity avoidance
    ↓
ARCHITECTURE DESIGN PHASE
    ↓
RAS-7: ARCHITECTURE INTRODUCTION
├─ Design principles
├─ Functional vs quality attributes
├─ Static vs dynamic structures
└─ Big ball of mud avoidance
    ↓
RAS-8: RISK-DRIVEN APPROACH
├─ Risk identification
├─ Risk prioritization
├─ Design styles: NDUF, BDUF, LDUF
└─ Effort proportional to risk
    ↓
RAS-11: ARCHITECTURAL STYLES
├─ Layered (modifiability)
├─ Pipes-Filter (transformation)
├─ Client-Server (synchronous)
```

```
├─ MVC (decoupling)
├─ Publish-Subscribe (event-driven)
└─ 6 additional styles
    ↓
RAS-10: DESIGN PATTERNS
├─ Strategy (encapsulate algorithms)
├─ Observer (one-to-many notification)
└─ Decorator (dynamic responsibilities)
    ↓
RAS-9: DESIGN TACTICS
├─ Availability (fault detection/recovery/prevention)
└─ Performance (resource management)
    ↓
IMPLEMENTATION & CODING PHASE
    ↓
RAS-12: REFACTORING
├─ Code smell identification
├─ 9 refactoring techniques
├─ Small steps + always test
└─ Continuous improvement
    ↓
QUALITY ASSURANCE & DEPLOYMENT
```

13.2 Quality Attributes Traceability

```
RAS-2: Define NFRs (Appearance, Usability, Performance, etc.)
        ↓
RAS-8: Identify risks related to quality attributes
        ↓
RAS-11: Select architectural style addressing quality priorities
        ↓
RAS-9: Apply specific tactics per quality attribute
        ↓
RAS-10: Use patterns supporting quality
        ↓
RAS-12: Refactor code maintaining quality
```