

Código Limpo:

Habilidades Práticas do Agile Software



Capítulo 12: Emergência

**Este material foi desenvolvido pelo grupo PET/ADS do
IFSP São Carlos**

Introdução

Em sistemas complexos, quaisquer ações que pareçam inofensivas podem causar reações inesperadas e indesejadas.

Dessa forma, torna-se prioridade a realização de projetos simples, fáceis de manipular e manter, e previsíveis.

Seguindo as Quatro Regras do Projeto Simples de Kent Beck, é possível garantir o desenvolvimento de sistemas que, mesmo complexos, ainda são manejáveis.

As regras estão organizadas por ordem de importância.

Regra 1: Efetue todos os testes

Acima de tudo, um projeto deve gerar um sistema que funcione como o esperado. Isso implica na criação de um sistema em que todas suas funções são facilmente testáveis de maneira detalhada.

Devemos tentar realizar a maior cobertura possível de testes. Como benefício, isso nos direciona cada vez mais a seguir boas práticas de código e a respeitar princípios da programação.

Classes simples e com responsabilidade única são mais facilmente testáveis. Quantos mais testes criarmos, mais seremos direcionados às coisas simples.

Regra 1: Efetue todos os testes

O alto acoplamento dificulta a criação de testes.

Por isso, quanto mais testes criarmos, maior será a tendência de adotarmos um projeto baseado em interfaces e no Princípio da Inversão de Dependência.

Ao seguir essa regra simples, é possível alcançar o ideal da Orientação a Objetos: fraco acoplamento e coesão alta.

Criar testes leva a projetos melhores.

Regra 1: Efetue todos os testes

Com a existência de testes efetivos vêm código e classes limpas. Os testes fornecem liberdade para refatorar livremente o projeto.

Caso haja a necessidade de alterar algo, basta rodar novamente os testes para garantir a integridade do código.

Durante a refatoração, podem ser aplicados quaisquer conceitos que levem a bons projetos de software. Além disso, devemos aplicar as seguintes regras:

- Sem duplicação de código
- Expressar o propósito do programador
- Minimizar o número de classes e métodos

Regra 2: Sem duplicação de código

A repetição de código traz consigo trabalho, risco e complexidade desnecessária. A duplicação pode ocorrer de maneiras que diferem de simplesmente linhas idênticas de código:

```
//implementação de uma coleção qualquer

public int size() {
    //implementação
}

public boolean isEmpty() {
    //implementação com elementos repetidos
}

-----
// podemos melhorar a implementação de isEmpty reutilizando size:

public boolean isEmpty() {
    return size() == 0;
}
```

Regra 2: Sem duplicação de código

Devemos reduzir a duplicação mesmo em lugares em que ela está presente em pouquíssimas linhas de código.

Ao extrairmos uma pequena função, podemos perceber violações no Princípio de Responsabilidade Única (SRP), indicando a necessidade da criação de outras classes.

Isso ajuda reduzir a complexidade desnecessária no código.

Regra 2: Sem duplicação de código

Uma forma de reduzir a duplicação em métodos é o uso do padrão de projeto “*Template Method*”.

Em situações nas quais diversas partes de código estão sendo repetidas por classes, com alterações mínimas em algumas partes, esse padrão sugere que se quebrem tais algoritmos em vários passos (métodos), a serem colocados em um “método modelo”.

Os passos do Template Method podem ser abstratos ou possuírem implementações padrão. As classes nas quais se encontrava a repetição se tornam clientes da classe Template, alterando as funções fornecidas de acordo com suas necessidades.

Regra 2: Sem duplicação de código

Nesse exemplo, as implementações dos métodos export, de ambas classes, possuem diversos elementos repetidos.

Esses passos repetidos podem ser extraídos para um Template Method.

PdfExporter
+export(file)

+extractData(data)

+formatDataPdf(extracted)

+analyzeData(formated)

+generatePdf(analyzed)

JpegExporter
+export(file)

+extractData(data)

<- duplicated

+formatDataJpeg(extracted)

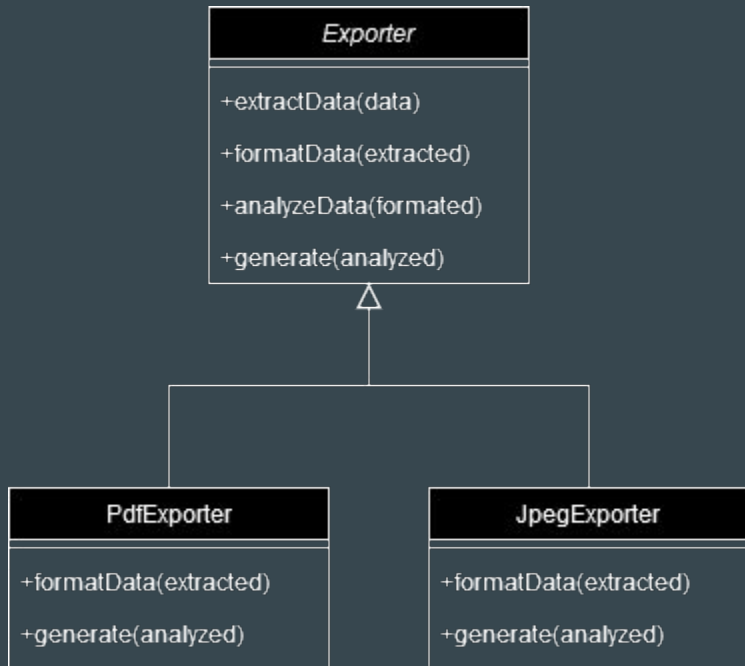
+analyzeData(formated) <- duplicated

+generateJpeg(analyzed)

Regra 2: Sem duplicação de código

Assim, criamos a classe `Exporter`, que possui implementações padrão para os métodos em comum: `extractData` e `analyzeData`.

Já os métodos `formatData` e `generate` devem ser sobrescritos pelas subclasses (isso é, são necessariamente abstratos.)



Regra 3: Expressividade

A maior parte do custo de um projeto de longo prazo está na etapa da manutenção. Para minimizar o risco das alterações no código, o desenvolvedor deve compreender o código que vai mudar.

Conforme os sistemas crescem e se tornam mais complexos, essa compreensão se torna mais demorada; e a possibilidade de mal entendidos se torna maior.

Por isso, o código deve expressar claramente o propósito de seu autor. Quanto mais compreensível o código é, por si só, menos tempo outros desenvolvedores terão que gastar para entendê-lo.

Regra 3: Expressividade

Essa expressividade pode ser alcançada por meio da escolha de bons nomes, que explicam sucintamente o motivo de existência daquela classe, método ou atributo.

Além disso, quanto menores e mais aderentes ao SRP as classes e funções forem, mais fácil é seu entendimento.

Também é recomendado o uso de nomenclatura padrão. Um artefato de código ou entidade deve possuir o mesmo nome em todo o projeto, para evitar confusões.

O uso de nomes de padrões de projeto também deve ser seguidos. Classes que implementam padrões como Publisher/Observer devem ser nomeadas de acordo com o padrão.

Regra 3: Expressividade

Testes de unidade também devem ser escritos de maneira expressiva. Ao ler os testes, qualquer um deve conseguir entender rapidamente o propósito de uma classe.

O fator mais importante na criação da expressividade é tentar. Devemos tratar nosso código com orgulho, gastando um pouco de tempo na refatoração de classes e métodos para aperfeiçoá-los.

Ao criar uma solução, certifique-se de que ela é legível e compreensível para o próximo desenvolvedor que irá lê-la. É bem provável que o outro desenvolvedor seja você mesmo.

Regra 4: Poucos métodos e classes

Ao tentar seguir dogmaticamente os conceitos mais fundamentais, como a eliminação de duplicação, expressividade e o SRP, podemos acabar criando inúmeras estruturas minúsculas.

Esse regra recomenda minimizar o número de classes e métodos, com o propósito de não inchar o projeto com um número incontável de elementos.

O objetivo da regra é criar um sistema geral pequeno e, ao mesmo, criar classes e métodos também pequenos.

Entretanto, por ser a última regra, ela é de menor prioridade. Eliminar a duplicação e alcançar um alto nível de expressividade são mais importantes.

Conclusão

As práticas descritas neste capítulo permitem a criação de sistemas que resolvem os problemas para os quais estes foram projetados, mantendo um nível de simplicidade e compreensão para ajudar na manutenção do código.

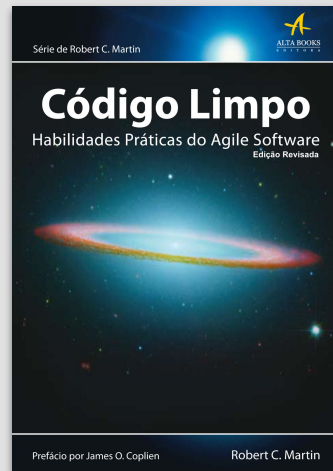
Ao segui-las, os desenvolvedores são incentivados a aderir a bons princípios e padrões que, de outra forma, levariam anos para aprender.

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.



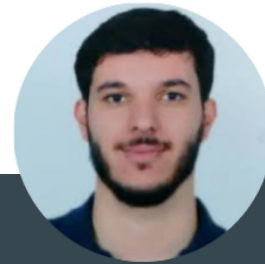
A equipe



Gabriel Z Manhani, Autor

Aluno e bolsista do PET/ADS desde
maio de 2023

[LinkedIn](#)



Lucas Oliveira, Revisor

Professor de Computação, é tutor do
PET/ADS desde janeiro de 2023.

[LinkedIn](#)