No desenvolvimento de software, assim como qualquer produto de engenharia, é fundamental que o sistema seja testado. Isso se aplica também às atividades de manutenção. O software não é algo estático; ele evolui ao longo do tempo e, por isso, deve ser constantemente mantido e aprimorado. Existem diferentes tipos de manutenção, como a manutenção corretiva, que ocorre quando um bug é detectado, a manutenção evolutiva, que se dá quando novas funcionalidades são solicitadas, e a manutenção adaptativa, necessária quando há mudanças nas regras de negócio ou nas tecnologias utilizadas. Manutenções regulares são essenciais para garantir que o software se mantenha eficiente, relevante e capaz de atender às novas demandas do mercado.

Meir Lehman, já na década de 1970, observou que, com o tempo, os sistemas de software tendem a envelhecer, assim como seres vivos. A partir disso, ele formulou as Leis da Evolução de Software, que explicam o fenômeno do envelhecimento, da qualidade interna e da evolução dos sistemas. A primeira dessas leis destaca a necessidade de manutenção contínua para que o sistema se adapte ao ambiente. No entanto, ela também sugere que, em determinado momento, pode ser mais vantajoso substituir o sistema por um novo. A segunda lei aponta que, à medida que um sistema passa por manutenções, sua complexidade interna tende a aumentar, e a qualidade de sua estrutura pode se deteriorar, a não ser que ações específicas sejam tomadas para estabilizar o sistema. Esse processo de estabilização e preservação da qualidade do software é o que chamamos de refactoring.

Refactorings são mudanças no código de um sistema que visam melhorar sua manutenibilidade, sem alterar o seu comportamento. Isso significa que, ao realizar refactorings, a funcionalidade do sistema deve ser preservada, mas seu código pode ser reorganizado, otimizado e simplificado, tornando-o mais legível e fácil de entender. Por exemplo, refactoring pode envolver a renomeação de variáveis, a divisão de funções longas em funções menores ou a extração de interfaces. O objetivo é tornar o código mais modular e fácil de modificar no futuro, o que ajuda a evitar a deterioração da qualidade do sistema mencionada por Lehman. A prática de refactoring se popularizou no início dos anos 90, quando William Opdyke introduziu o termo em sua tese de doutorado, e mais tarde, Martin Fowler lançou um livro dedicado ao tema, com um catálogo de refactorings. Desde então, a prática tem sido parte essencial do processo de desenvolvimento de software, sendo fundamental para a manutenção e evolução saudável de sistemas ao longo do tempo.

A extração de método é uma técnica muito útil quando um trecho de código se repete ou se torna complexo demais dentro de um método. Em vez de deixar esse código disperso, podemos movê-lo para um novo método com um nome descritivo, tornando o código mais modular e legível. Ao fazer isso, conseguimos reduzir a duplicação e organizar melhor a lógica, facilitando a compreensão e a manutenção do sistema. Por outro lado, às vezes o melhor é aplicar o inline de método. Nesse caso, em vez de manter um método separado, podemos simplesmente mover seu conteúdo diretamente para onde ele é chamado. Essa técnica é particularmente útil quando o método não está agregando muito valor, como quando ele apenas retorna um valor simples ou delega uma tarefa trivial, tornando o código mais direto e menos fragmentado.

Em certos casos, um método pode estar mal posicionado em uma classe, o que torna a movimentação de método uma boa estratégia. Se um método estiver lidando com dados ou responsabilidades que pertencem mais a outra classe, movê-lo para o lugar certo melhora a coesão e a organização do código. Isso também pode facilitar a manutenção, já que o código ficará mais alinhado com a estrutura do sistema. Além disso, a renomeação de método pode ser uma ação simples, mas poderosa. Quando um método não descreve claramente o que faz, mudar seu nome para algo mais explícito ajuda a tornar o código mais legível e compreensível. Isso beneficia qualquer desenvolvedor que trabalhe no sistema, pois o nome do método passará a refletir melhor a tarefa que ele executa, tornando o código mais intuitivo.

É importante também estar atento aos code smells, que são sinais de que o código pode estar mal estruturado ou precisar de ajustes. Embora não sejam erros diretos, eles indicam áreas que podem ser refatoradas para melhorar a qualidade do código. Identificar esses "cheiros" pode ajudar a direcionar o esforço de refatoração, seja para eliminar duplicação, melhorar a legibilidade ou reduzir a complexidade de um método ou classe. Quando um método grande é identificado, ele costuma ser um dos principais culpados por problemas de legibilidade e manutenibilidade. Métodos com muitas linhas de código frequentemente têm responsabilidades demais, o que dificulta o entendimento e a realização de testes. A solução aqui é dividir o método em partes menores, cada uma com um objetivo claro e específico. Isso não apenas torna o código mais fácil de entender, mas também melhora sua reutilização e facilita a criação de testes unitários mais eficazes.

Classes muito grandes são um típico code smell que indica que uma classe está assumindo muitas responsabilidades, o que torna difícil de entender, testar e manter. Quando uma classe cresce demais, ela pode começar a se tornar um “monstro” de código, sobrecarregada com funções e dados que não têm uma relação clara entre si. Isso viola o princípio da Responsabilidade Única, que afirma que uma classe deve ter uma única razão para mudar. A melhor abordagem para resolver esse problema é refatorar a classe, quebrando-a em várias classes menores, cada uma com uma responsabilidade bem definida. Essa refatoração torna o sistema mais modular, com código mais fácil de entender e modificar, além de melhorar a testabilidade.

Outro problema comum ocorre quando há obsessão com tipos primitivos. Isso acontece quando os desenvolvedores utilizam tipos básicos, como inteiros ou strings, para representar conceitos mais complexos ou entidades que poderiam ser modeladas de forma mais robusta. Por exemplo, em vez de usar um tipo String para representar uma data, poderíamos criar uma classe Data que encapsula as operações específicas e validações necessárias para manipular datas de maneira mais clara. O uso excessivo de tipos primitivos pode tornar o código frágil e propenso a erros, já que não oferece validações ou abstrações adequadas. A refatoração para tipos mais expressivos ou a criação de objetos de valor pode melhorar a legibilidade e a manutenção do sistema, tornando-o mais semântico e seguro.

Face ao exposto, é inegável que o principal tópico para facilitar a refatoração, sustentabilidade e manutenibilidade de um código ao longo do

tempo é sua modularidade, raramente sistemas coesos tem classes muito grandes, ou funções responsáveis por realizar mais de uma ação.