Capítulo 7

O autor começa destacando por que testar um sistema vai muito além de apenas encontrar bugs – os testes garantem confiabilidade, facilitam a manutenção e permitem que o software evolua com segurança.

Uma das primeiras abordagens discutidas é a piramide de testes, que organiza os tipos de testes em camadas, mostrando a importância de cada nível. Na base da pirâmide estão os testes de unidade, que verificam pequenas partes do código, como funções ou métodos individuais. São rápidos e baratos de rodar, por isso devem ser a maioria dos testes em um sistema. No meio da pirâmide vêm os testes de integração, que verificam como diferentes módulos ou componentes interagem entre si. No topo estão os testes de aceitação (ou de sistema), que simulam o comportamento do usuário final e garantem que a aplicação funciona como esperado em um ambiente real.

Outro conceito fundamental abordado no capítulo é o TDD (Test-Driven Development). O autor explica como essa técnica inverte a ordem tradicional de desenvolvimento: primeiro se escreve um teste para um novo recurso, depois se implementa o código para fazer o teste passar e, por fim, o código é refatorado para melhorar sua qualidade. Essa abordagem força os desenvolvedores a pensarem nos requisitos antes mesmo de escreverem a implementação, resultando em código mais modular e confiável.

Além disso, o capítulo explora a cobertura de testes, uma métrica usada para avaliar o quanto do código foi realmente testado. Embora um alto percentual de cobertura seja desejável, o autor alerta que isso por si só não garante a qualidade dos testes – um código pode ter 100% de cobertura e ainda assim não capturar cenários importantes se os testes forem mal planejados.

O autor também discute ferramentas e boas práticas para automação de testes, mostrando como frameworks como JUnit e pytest facilitam a escrita e execução dos testes. Além disso, ele explica o conceito de testabilidade, ou seja, o quão fácil é testar um determinado código. Classes muito complexas ou fortemente acopladas tendem a ser difíceis de testar, o que reforça a importância de um bom design de software.

No fim das contas, esse capítulo ensina que testes não são apenas uma etapa final do desenvolvimento,

mas uma parte essencial de todo o processo. Um software bem testado não só evita problemas em produção, como também reduz o custo da manutenção a longo prazo. Mais do que uma obrigação, testar é um investimento na qualidade do código.

Capítulo 9

Neste capítulo, a situação de implantar mudanças no software sem planejamento e acabar causando problemas é evidenciada como um grande desafio no desenvolvimento moderno. Para evitar esse tipo de caos, o autor apresenta três práticas essenciais: integração contínua, entrega contínua e deployment contínuo.

A integração contínua (CI) resolve um problema clássico: juntar o trabalho de vários desenvolvedores sem gerar conflitos ou falhas inesperadas. Em vez de esperar semanas para integrar tudo, os commits são feitos com frequência, e cada alteração já passa por testes automáticos. Isso evita aqueles momentos de desespero quando, ao unir o código de todo mundo, nada funciona como deveria. Ferramentas como GitHub Actions, GitLab CI/CD e Jenkins ajudam a automatizar esse processo.

Mas apenas integrar o código não basta se a entrega ainda for um processo demorado. É aí que entra a entrega contínua (CD - Continuous Delivery). Com essa abordagem, o código não só é testado automaticamente, mas também fica sempre pronto para ser implantado em produção. Isso significa que novas versões podem ser lançadas rapidamente, sem a necessidade de grandes retrabalhos ou medo de quebrar algo.

Já o deployment contínuo (Continuous Deployment) leva essa ideia ao extremo. Aqui, qualquer mudança que passa pelos testes é automaticamente implantada em produção, sem intervenção manual. Isso permite atualizações constantes e rápidas, mas exige um alto nível de maturidade da equipe e uma infraestrutura robusta.

O capítulo também explora boas práticas para criar uma pipeline de CI/CD eficiente, destacando a importância do monitoramento, rollback automatizado e estratégias para garantir que tudo funcione bem. No fim das contas, a mensagem principal é clara: CI/CD não é um luxo para grandes empresas, mas uma necessidade para qualquer equipe que deseja desenvolver e entregar software com mais segurança e rapidez.