O próprio site do Selenium disponibiliza um guia de boas práticas, e nessa aula a ideia é analisarmos esse guia, que serve como um bom ponto de partida para o seu desenvolvimento.

A primeira delas, que já seguimos, se refere aos Page Objects, e **indica que não devemos acessar a API do Selenium dentro das classes de teste com o JUnit, favorecendo a separação de responsabilidades e a facilidade de manutenção.**

Outra recomendação é **utilizar uma linguagem específica de domínio** (ou DSL, de "Domain Specific Language"), algo que também já fizemos. No momento em que escrevemos os testes, é recomendado que os **nomes de métodos sejam legíveis e estejam orientados ao domínio**.

O ideal é que o Selenium não seja usado para preparar um estado da aplicação, mas que cada cenário de teste prepare tudo que ele precisa. Por exemplo, para cadastrarmos um leilão, primeiro tivemos que abrir o formulário de login, efetuar o login, navegar para a página de leilões e então navegar para a prática de formulário. Não é uma boa prática deixar isso pré-configurado, mas sim criar o passo-a-passo no próprio teste. Dessa forma, o teste não ficará baseado em um estado estático da aplicação, que poderá ser alterado futuramente.

Temos uma página sobre serviços externos. Se o seu teste precisa acessar uma API externa ou algo do gênero, o **ideal é fazermos um Mock, como o Mockito**, simulando os comportamentos desses serviços. Como os testes E2E já são demorados e precisam que a aplicação esteja rodando, de modo a abrir o navegador e clicar nos elementos, acessar recursos externos fará com que eles demorem mais ainda. A utilização de mocks favorece uma melhor performance nos testes.

A próxima seção é sobre o "**report**" de testes. O Selenium por si só não reporta os estados dos testes, e portanto recomenda que você utilize uma ferramenta de testes como o JUnit, que permite a configuração desses reports para que saibemos quais testes passaram, quanto tempo levou, quais demoraram mais e assim por diante.

A documentação também nos recomenda evitar o compartilhamento de estados. Cada teste deve rodar de maneira isolada - ou seja, um teste não deve guardar um estado para outros

testes o reaproveitem. Se um teste precisa acessar uma página de login, não devemos depender que ela já esteja carregada a partir de outro teste.

Foi exatamente isso que fizemos em nosso projeto: **cada método de teste começa a navegação do zero, desde a abertura do navegador até as verificações**. É um pouco mais trabalhoso, já que precisamos escrever mais códigos, mas é uma garantia a mais de que nossos testes serão fidedignos e de fácil manutenção no futuro. Além disso, podemos utilizar alguns padrões e a herança para reduzirmos a verbosidade dos códigos de teste.

O próximo conteúdo fala sobre a **independência dos testes**, conceito bastante relacionado ao anterior: cada teste deve rodar de maneira independente do outro.

Recomenda-se utilizar uma API fluente, contendo exemplos de como utilizar uma linguagem fluente para fazer encadeamentos de métodos.

Com isso, fica mais fácil ler e entender o código. Não fizemos esse tipo de construção no nosso projeto, mas fica como desafio se você quiser implementar.

A última recomendação é que **cada teste deve ter um navegador "limpo", ou seja, devemos abrir uma nova janela** ao invés de reaproveitarmos o que já estiver aberto do teste anterior. Isso porque um teste pode guardar alguma informação, como um cookie ou um estado, que pode influenciar nos testes seguintes.