Testes automáticos:

Testes automáticos são testes executados sem necessidade da interferência humana. Tu cria eles, aperta um botão e pronto, eles foram executados.

Tecnologias de testes automatizadoas:

JUnit é um framework de desenvolvimento de testes automátizados para Java, ele também utiliza bibliotecas auxiliares como Hamcrest por exemplo.

Organização das classes de teste:

No pacote de testes geralmente você tem uma classe de teste para uma classe do programa do pacote main. A manutenção do teste tem que ser feita sempre que uma regra de negócio for alterada.

Explicação testes:

GerenciadorDeClientesTest_Ex1(explicação básica de como um teste funciona):

a anotação de **@Test** do Junit significa que o método anotado deve ser executado como um teste unitário. Para facilitar a compreensão do pre-set necessário para efetuação do teste é muito comunmente usado um cenário GIVEN/WHEN/THEN(Dado que, Quando, Então). Esse cenário ajuda decifrar passos de como o teste deve ocorrer, desde entender o que ele precisa(GIVEN/DADO QUE) até o que ele fará com isso(WHEN/QUANDO) e o resultado que esperamos dele(THEN/ENTÃO). O Hamcreste entrou no final com o AssertThat.

GerenciadorDeClientesTest_Ex2(como melhorar um teste com mais verificações):

O segundo teste efetuado foi o da remoção de cliente, onde o cenário para configuração foi o mesmo do segundo, usando o padrão GIVEN/WHEN/THEN executamos o que haavia de necessário para o teste. O destaque fica para o assertThat adicional, onde implementamos mais verificações que deveriam acontecer caso o método fosse bem sucedido, como a lista diminuir seu tamanho de 2 para 1 devido a exclusão do cliente, acertando o resultado de acordo com o retorno do método que nesse caso era um booleano, e verificando que se eu usasse o método de pesquisa de cliente usando o id do cliente excluido ele de fato me retornaria null.

GerenciadorDeClientesTest_Ex3(Otimizando o entendimento do teste):

Nesse exemplo a ideia é deixar bem claro os processos de execução de um teste, é basicamente o cenário GIVEN/WHEN/THEN sendo aplicado, mas com comentários bem separativos entre as etapas. O ponto é, QUANTO MAIS SIMPLES DE ENTENDER FOR SEU TESTE, MELHOR.

GerenciadorDeClientesTest_Ex4(Usando comentários como documentação):

Comentários de documentação acima do método são uma excelente prática, descrever o que aquele método faz com precisão e acertividade é a chave para manter um sistema bem manutenível a quem precisar lidar com ele. Então a partir daqui todos os métodos de testes unitários terão a documentação comentada acima e somente os GIVEN/WHEN/THEN como comentário do código em sí.

GerenciadorDeClientesTest_Ex5(Aplicando boas práticas de programação também em testes):

No exemplo 5 do teste unitário, apenas aplicamos uma boa prática da programação, como os atributos de id eram usadas em mais de um trecho do código as transformamos em váriaveis. Isso possibilita a manutenção do teste ser feita com mais facilidade, já que vc pode mudar a variável e consequentemente mudará todos os locais que a usam.

GerenciadorDeClientesTest_Ex7(Otimizando testes com cenários parecidos):

Olhando as classes de testes até agora e analisando os métodos testados podemos ver que eles tem códigos repetidos na montagem do cenário(GIVEN/DADO QUE), para excluir ou pesquisar clientes precisamos basicamente do mesmo cenário. Para otimizar esse código replicado podemos usar a anotação @Before que diz o que deve ser executado antes do método de @Test, nessa configuração montamos o cenário para os testes, ou seja, fazemos o setUp( ). Agora nossa condição de GIVEN/DADO QUE esta configurada no setUp e podemos dedicar nossos @Test apenas a condição de WHEN/THEN(Quando/Então) reduzindo o tamanho dele e otimizando o tempo de execução. Temos também a anotação de @After, lógicamente o que vai ser executado depois do @Test.

GerenciadorDeClientesTest_Ex8(Tornando os testes independentes):

O @After é executado ao fim de cada teste unitário, o poder dele é realmente o poder de tornar um teste independente do outro, você usa o @After para terminar tudo que o @Before criou, assim garantimos que o setUp( ) não acumule e interfira em testes que o usem. No código temos exemplos de como o @After pode ajudar nisso.

TodosOsTestesJava(Executando uma suíte de testes TestSuite):

Uma suíte de testes é uma classe que serve para rodarmos todos os testes de uma única vez. Ela utiliza duas anotações do JUnit, @RunWith(Suite.class) para informar que é uma classe suite, e a segunda anotação @SuiteClasses( ) passamos as classes que queremos executar nessa suite de testes.

GerenciadorDeClientesTest_Ex10(Multiplicando testes):

Talvez a primeira criação de um teste seja dificil ou leve mais tempo, mas a partir de um teste criado é facil criar outros. Tendo a base de um teste, você consegue com mais facilidade testar resultados diferentes, em suma é "depois de escrito um teste, o resto é speedrun".

GerenciadorDeClientesTest_Ex10(valores limites):

O método de validação de idade do cliente é entre 18 e 65 anos, temos testes que validam essas idades limites e também os valores limites que estão fora desse escopo. Então ficariamos com um teste para validar 18, um para validar 65, um para validar 17 e um para validar 66. Quando a idade está fora da idade permitida, o método lança uma exceção, então usamos o "fail( )" abaixo da linha do método que lançará uma exceção e então fazemos a validação da nossa exceção, o "fail()" serve somente para caso a exceção não seja lançada, então ele cai no fail( ) e falha nosso teste.

Teste de regressão: Quando você executa todos os testes unitários depois de ter feito qualquer alteração que seja no programa.