Arrumando o teste de email:

Pensando sempre na **legibilidade do código**, vamos **refatorar algumas coisas** no **BuilderLocacao e no UsuarioBuilder**, para tornar a leitura mais amigável. Primeiro vamos fazer um método para alterar o nome do usuário(já que é isso que torna o objeto usuário diferente um do outro de acordo com a sobrescrita do equals):

```
public UsuarioBuilder comNome(String nome) {
    usuario.setNome(nome);
    return this;
}
```

Pronto, agora no teste podemos criar um usuário diferente através do builder, esse usuário não estara com a data de retorno da locação atrasada:

```
@Test
public void deveEnviarEmailAUsuarioComLocacaoAtrasada() {

    //GIVEN/Dado que
    Usuario usuarioComLocacaoAtrasada = UsuarioBuilder.umUsuario().agora();
    Usuario usuarioComLocacaoEmDia = UsuarioBuilder.umUsuario().comNome("Jorge").agora();
```

Agora vamos simplificar a criação de uma locação atrasada no **LocacaoBuilder** através de um método também:

```
public LocacaoBuilder atrasada() {
    locacao.setDataLocacao(DataUtils.obterDataComDiferencaDias(-4));
    locacao.setDataRetorno(DataUtils.obterDataComDiferencaDias(-2));
    return this;
}
```

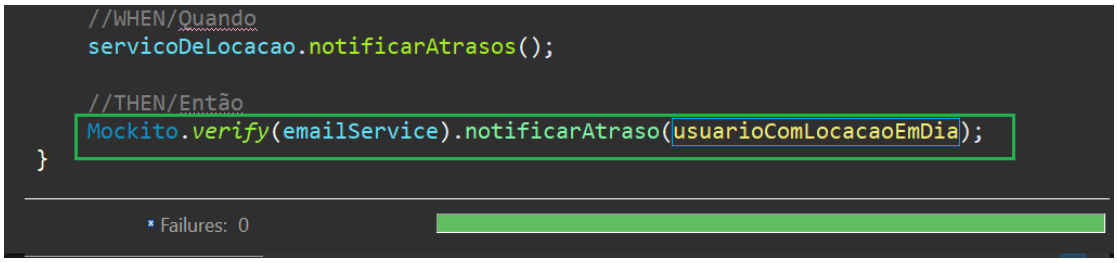
Note que mudamos **dois atributos em um mesmo método**, dessa maneira **conseguimos consistência** na hora de **instânciar uma locação atra**sada, pois definimos sua data de locação para 4 dias atras, e sua data de retorno para 2 dias atras, ou seja, **ela esta atrasada em 2 dias.**

Agora podemos voltar ao teste e simplificar a criação do cenário, agora temos uma locação para o o **usuarioComLocacaoAtrasada** e uma locação para o **usuarioComLocacaoEmDia**:

```
List<Locacao> locacoesPendentes = Arrays.asList(
        LocacaoBuilder.umaLocacao().atrasada().comUsuario(usuarioComLocacaoAtrasada).agora(),
        LocacaoBuilder.umaLocacao().comUsuario(usuarioComLocacaoEmDia).agora());
```

Basta testarmos nosso método de enviar email agora nas duas situações:

Com o **usuarioComLocacaoAtrasa obtemos sucesso**, perfeito, era esse o **comportamento esperado**. Vamos ver agora com o usuário que não tem locação atrasada:



E temos um erro, o **usuarioComLocacaoEmDia** não deveria esta recebendo esse email.

Entendendo o que deu errado:

Nosso método não faz distinção em relação as datas:

```
public void notificarAtrasos() {
    List<Locacao> locacoes = dao.obterLocacoesPendentes();

    for(Locacao locacao: locacoes) {
        emailService.notificarAtraso(locacao.getUsuario());
    }
}
```

Vamos então adicionar um **if** que irá **verificar se a data de retorno esta antes da data atual**, o que significa que ela está atrasada. Esse **before** é do **próprio Date**:

```
public void notificarAtrasos() {
    List<Locacao> locacoes = dao.obterLocacoesPendentes();

    for(Locacao locacao: locacoes) {
        if(locacao.getDataRetorno().before(new Date())) {
            emailService.notificarAtraso(locacao.getUsuario());
        }
    }
}
```

Agora temos o resultado esperado, não enviaremos email para o **usuarioComLocacaoEmDia**:

```
    //WHEN/Quando
    servicoDeLocacao.notificarAtrasos();

    //THEN/Então
    Mockito.verify(emailService).notificarAtraso(usuarioComLocacaoEmDia);
}
```

Fortalecendo as verificações:

Podemos adicionar **mais um usuário atrasado** e garantir que ele também irá receber o email:

```
//GIVEN/Dado que
Usuario usuarioComLocacaoAtrasada = UsuarioBuilder.umUsuario().agora();
Usuario outroUsuarioatrasada = UsuarioBuilder.umUsuario().comNome("Valdir").agora();
Usuario usuarioComLocacaoEmDia = UsuarioBuilder.umUsuario().comNome("Jorge").agora();
```

```
LocacaoBuilder.umaLocacao().atrasada().comUsuario(outroUsuarioatrasada).agora(),
LocacaoBuilder.umaLocacao().comUsuario(usuarioComLocacaoEmDia).agora());
```

```
Mockito.verify(emailService).notificarAtraso(usuarioComLocacaoAtrasada);
Mockito.verify(emailService).notificarAtraso(outroUsuarioatrasada);
```

Para verificar que o **UsurioComLocacaoEmDia nunca receba esse email**, podemos usar o **Mockito.never()** como parâmetro de verify:

```
Mockito.verify(emailService).notificarAtraso(usuarioComLocacaoAtrasada);
Mockito.verify(emailService).notificarAtraso(outroUsuarioatrasada);
Mockito.verify(emailService, Mockito.never()).notificarAtraso(usuarioComLocacaoEmDia);
```

```
rors: 0        * Failures: 0
```

Quer deixar mais seguro ainda? então podemos **garantir que depois de ter feito suas funções, o emailService não foi mais chamado**. Usando o **Mockito.verifyNoMoreInteraction( )** passando **como parâmetro** aquilo que queremos **verificar que não houve mais interação**, no caso o **emailService:**

```
    //THEN/Então
    Mockito.verify(emailService).notificarAtraso(usuarioComLocacaoAtrasada);
    Mockito.verify(emailService).notificarAtraso(outroUsuarioatrasada);
    Mockito.verify(emailService, Mockito.never()).notificarAtraso(usuarioComLocacaoEmDia);
    Mockito.verifyNoMoreInteractions(emailService);
```

```
Errors: 0        * Failures: 0
```

Com isso temos **uma cobertura excelente do cenário**, garantimos que o os usuários **atrasados receberam o email**, que o **usuário que não esta atrasado nunca recebeu o email** e que depois de concluir suas funções **o emailService não foi mais usado.**

Se um usuário tivesse mais de uma locação atrasada:

E se algum usuário tivesse mais de uma locação atrasada, vamos fazer isso com o "Valdir" vulgo **outroUsuarioAtrasada:**

```
LocacaoBuilder.umaLocacao().atrasada().comUsuario(outroUsuarioatrasada).agora(),
LocacaoBuilder.umaLocacao().atrasada().comUsuario(outroUsuarioatrasada).agora(),
LocacaoBuilder.umaLocacao().comUsuario(usuarioComLocacaoEmDia).agora());
```

Podemos usar os **utilitários do Mockito no verify** de algumas maneiras, podemos usar o **atLeastOnce() para informar que o Valdir irá receber PELO MENOS um email**, poderiamos usar o **atLeast( ) passando por parâmetro a quantidade de vezes que ele irá receber um email no MINIMO**, e o **atMos() passando por parâmetro a quantidade MÁXIMA que ele pode receber de emais**:

```
Mockito.verify(emailService, Mockito.atLeastOnce()).notificarAtraso(outroUsuarioatrasada);

Mockito.verify(emailService, Mockito.atLeast(2)).notificarAtraso(outroUsuarioatrasada);

Mockito.verify(emailService, Mockito.atMost(5)).notificarAtraso(outroUsuarioatrasada);
```

E se por acaso quisessimos uma **verficação mais genérica**, sem se importar com qual **usuário exatamente eu estou enviando o email ou não**, podiamos usar então outro método do **Mockito**:

```
//THEN/Então
Mockito.verify(emailService, Mockito.times(3)).notificarAtraso(Mockito.any(Usuario.class));
```

Aqui dizemos para **verificar se o emailService foi executado 3 vezes** o **método notificarAtraso** usando um **matcher coringa do Mockito**, o **Mockito.any( )** passando a classe **de Usuario**, isso significa que não importa o **Usuario em sí mas se o método foi utilizado essas vezes estipuladas para qualquer instância da classe de usuário**.

Isso torna a verificação de fato mais genérica, mas adiciona praticidade e deixa o método de teste menor. No fim das contas cabe a você decidir o que usar.

Consideração sobre os matchers: Se um método testado usar mais de um parâmetro e eu passar um matcher como um desses parâmetros, então terei que usar matchers nos outros também.