1. Introdução
2. Testes
   * Teste de software: definição:

Teste de software é avaliação do software, checando se o mesmo está se comportando dentro da sua especificação e da maneira esperada em relação à expectativa do cliente, comprador do sistema.

De acordo com Sommerville, Ian(2010, p.371), “O teste é uma fase dispendiosa e trabalhosa do processo de software.” Por este motivo a forma de avaliação do software está em constante evolução, essa evolução é focada em redução de tempo, redução de custos e em maior assertividade nos testes, ou seja, busca-se: testar em menor tempo, com menor custo e descobrir uma maior quantidade de erros.

No processo de desenvolvimento ágil, os testes se iniciam, em alguns momentos, antes mesmo do código a ser testado existir, o desenvolvedor cria o teste e depois cria o código que deve passar no teste criado. Utilizado pelo Test Driven Development (TDD) em português: Desenvolvimento dirigido por testes.

Com a evolução no processo de desenvolvimento de sistemas, o processo de testes também evoluiu, principalmente dentro do processo de desenvolvimento ágil.

Conforme descrito por Sommerville, Ian(2010, p.355), “[...] o processo de teste de software tem duas metas distintas: Demonstrar ao desenvolvedor e ao cliente que o software atende aos requisitos [...]” e Sommerville, Ian(2010, p.356), “Descobrir falhas ou defeitos no software que apresenta comportamento incorreto, não desejável ou em não conformidade com sua especificação[...]”. Sendo assim o hoje o teste de software é chamado de processo de teste de software, em praticamente todas as etapas do desenvolvimento, é aplicada pelo menos uma metodologia de testes, acredita-se que quanto mais aderente ao primeiro processo, ou seja, quanto mais testes aplicados para demonstração de aderência do software aos requisitos, falhas e defeitos serão reduzidos tendo menor trabalho de descobrir para a(s) causa(s).

O desenvolvimento e os testes de funcionalidades são intercalados, o programador cria a funcionalidade e logo depois o teste ou até mesmo antes, quando utilizado TDD. Essa abordagem é mais objetiva ao contexto desenvolvido, com o teste desenvolvido no momento do desenvolvimento, o desenvolvedor estará totalmente envolvido com a funcionalidade, conhecendo suas limitações e regras a serem obedecidas, com isso, os testes criados serão mais abrangentes na necessidade, irá avaliar a maior quantidade de risco possível, e será mais consistente, pontos irrelevantes não serão considerados.

Testes feitos após o desenvolvimento de grande quantidade de funcionalidades aumenta a possibilidade de dar se mais importância a riscos que deveriam ser ignorados e riscos potenciais serem esquecidos.

Sommerville, Ian(2010, p.53) confirma esse pensamento ao escrever “[...] Os programadores elaboram seus próprios dados de teste e testam os códigos de forma incremental, [...]. Essa é uma abordagem sensível economicamente, pois o programador conhece melhor o componente[...]”.

Varias formas de testes foram criadas, e dentro de cada uma, existe um ciclo de aplicação do teste, embora a forma de aplicação de teste varie conforme a maneira de sua execução, o ciclo do teste, na maioria das vezes, é o mesmo:

* O planejamento do teste: neste momento é definido o que se deseja testar e a forma como o teste será feito.
* A preparação do teste: aqui é o momento de preparar o ambiente para executar o teste.
* A execução do teste: execução de tudo que foi planejado e montado.
* A avaliação do teste realizado: determina se o teste cobriu uma margem aceitável de possíveis situações e se o sistema está se comportando da forma esperada ou se é necessário realizar correções.

O ciclo de teste é executado sobre uma determinada regra de negócio, para cada nova regra implementada, um novo ciclo é aplicado. A aplicação do ciclo não espera que todo o sistema esteja pronto para executar, à medida que cada nova regra é terminada, um novo ciclo é aplicado em conjunto. O ciclo pode se repetir n vezes sobre a mesma regra, até que o resultado final da aplicação do ciclo seja um resultado satisfatório.

* + Os estágios:
* Teste de unidade: consiste em testar uma pequena funcionalidade, componentes individuais são testados, pode ser uma função, uma classe ou até mesmo em um pequeno trecho de código, avaliando se o comportamento esperado é o que ocorre. Quem desenvolve e realiza o teste é o desenvolvedor que desenvolveu o código a ser testado.
* Teste de integração: neste momento é realizada uma avaliação do resultado da junção dos componentes internos do sistema. É a junção dos componentes que foram testados nos testes unitários, avaliando o comportamento esperado versus o comportamento apresentado. Quando uma funcionalidade passa a funcionar em conjunto com outra funcionalidade, que até o momento não estavam vinculadas. Um exemplo poderia ser a produção de um WebService, a equipe A desenvolve o lado de servir o WebService e a equipe B desenvolve o lado que irá consumir o serviço do WebService, o teste de integração seria o ato de consumir e servir o WebService para analisar se tudo está correndo bem.
* Teste de sistema: todos os componentes começam a trabalhar em conjunto, os erros normalmente são encontrados nas interações não previstas entre os componentes. O teste é realizado em um contexto o mais parecido com a realidade que o sistema irá trabalhar quando estiver em produção. Na parte de hardware: espaço em disco, memória física, velocidade de rede, etc., na parte de software: sistema operacional, containers web, etc., carga operacional, tenta-se popular os dados do banco de dados com uma quantidade e qualidade de informações o mais próximo da realidade prevista possível.
* Teste de aceitação: teste realizado com dados de produção, normalmente fornecido pelo cliente comprador do sistema. As falhas mais comuns são relacionadas a desempenho e ou adequação de algumas funcionalidades aos dados reais de produção.
* Teste de operação: aqui a avaliação é feita por quem administra o sistema, avalia-se se a troca do sistema, para os casos de uma nova versão, ou implantação do sistema para os novos, será suportada pelo hardware de produção.
  + Teste de unidade:

Teste de unidade são testes realizados sobre uma pequena parte de código, uma classe, um método ou mesmo parte de um trecho de código de um método, cobrindo uma funcionalidade ou parte dela. No caso de um teste unitário que cobre somente parte da lógica, um segundo teste é utilizado para cobrir a outra parte. Este cenário é muito utilizado quando deseja avaliar o comportamento de um método com base em seus parâmetros, o teste pode avaliar o comportamento utilizado parâmetros com valores básicos, que não tenderá a causar erros críticos, avalia-se apenas o resultado. O segundo teste é utilizado para buscar erros, considerando o pior cenário, onde se tenta passar vários parâmetros que poderiam fazer com que o método causasse um erro inesperado.

Os testes de unidades são, na maioria das vezes, automatizados, ou seja, são testes que, depois de escritos, executam sem necessidade de intervenção humana. O teste é feito de forma que ao termino da execução do teste é exibido o resultado da execução. É mostrado se o resultado foi o esperado, ou seja, o teste passou, se o resultado foi o inesperado, ou seja, o teste falhou ou se erros inesperados foram causados, representando falha.

A grande vantagem dos testes automatizados é que depois de escritos eles podem ser reexecutados varias vezes, em poucos segundos. Se o sistema foi criado sobre testes unitários desde o inicio, toda vez que os testes executam, todo o sistema estará sendo avaliado, normalmente a duração destes testes são de segundos.

Um dos pontos fortes de ter testes unitários automatizados é quando alguma nova funcionalidade é necessária, que para introduzi-la será necessário alterar uma parte de algum código existente, sem alterar a funcionalidade antiga, é possível testar todo o sistema novamente e detectar a se a modificação que foi feita no código alterou a funcionalidade antiga com maior rapidez, sabendo o local onde o problema está ocorrendo.

Um exemplo de uma situação assim: um sistema possui a funcionalidade de confirmar o pagamento de uma compra enviando um e-mail para o cliente. Existe um teste que verifica se quando uma compra é confirmada, checa se o e-mail foi enviado. Uma nova funcionalidade é solicitada: antes de enviar o e-mail devera ser verificado se o cliente comprou mais de trezentos reais no ultimo mês, se sim, será necessário anexar ao e-mail um cupom promocional. Após a alteração solicitada, se algum erro na nova lógica impedir que o e-mail seja enviado, o teste, que foi criado quando a funcionalidade de enviar e-mails foi desenvolvida, irá apresentar o erro. Importante resaltar que testes para a nova funcionalidade não estão descartados, devem ser escritos para que quando uma terceira modificação for necessária, o novo comportamento escrito seja avaliado.

Os testes unitários são independentes, ou seja, um teste deve ser executado sem necessidade de qualquer tipo de ordenação dos mesmos. Quando um teste inicia ele deve preparar suas necessidades e quando terminar ele deve destruir o que utilizou. Necessário para não poluir o próximo teste com dados inconsistentes.

Detalhamento do ciclo do teste unitário:

* Planejamento: neste momento o desenvolvedor irá analisar a complexidade do trecho de código que será testado, analisando o que é necessário para executar o teste. Muitas vezes para que um teste possa ser aplicado é necessário que uma massa pré-existente de dados esteja preparada e é nesta faze que esse levantamento é realizado.
* A preparação: aqui o código a ser executado no teste é escrito, toda necessidade antes do teste ser executado é criada, dados no banco de dados, criação de objetos com o perfil necessário, conexões a servidores devem ser abertas, tudo que é necessário para reproduzir um ambiente real é feito. A função do código de preparação é entregar ao código a ser testado insumos que simulam a realidade esperada do que irá ocorrer na realidade com o trecho de código testado. Quando teste sobre o código de produção termina é escrito o código que irá avaliar o resultado, tudo que é esperado é comparado com o resultado.
* A execução do teste: neste ponto tudo que foi escrito na preparação é executado, o resultado é quase que instantâneo.
* A avaliação do teste: no teste unitário automatizado o resultado é avaliado pelo próprio teste, com base nas avaliações escritas na preparação, o teste informa ao executor se algo deu errado.

Para que a independência de testes unitários automatizados ocorra, existe um custo, a preparação do teste, conforme apresentado, se torna custosa, ainda mais quando a necessidade de dados preexistentes é imprescindível para execução do mesmo. Quando isso ocorre, o desenvolvedor deverá criar insumos para execução, criando objetos em tempo de execução do teste, muitas vezes esses objetos, para existirem, necessitam de outros objetos, que também deverão ser criados pelo desenvolvedor. A tarefa de criação pode ser a mais complexa do teste unitário, e se as dependências não foram bem avaliadas na etapa de planejamento, o custo da preparação será bem maior que o esperado.

*Neste momento todos os dados necessários para a execução do código a ser testado são criados.*

* + Exemplo de teste unitário com JUnit

Uma ferramenta utilizada para criar testes unitários é o framework JUnit, um framework open-source criado por Erich Gamma e Kent Beck, criando testes automatizados na linguajem de programação Java. Esta ferramenta possui classes que o desenvolvedor utiliza, estendendo, para criar um ambiente de testes automatizado.

JUnit tem a finalidade de simplificar a execução e avaliação dos resultados dos testes, após a escrita do teste o JUnit pode ser executado várias vezes em pouco tempo.

Para exemplificar a utilização do JUnit vamos a um exemplo de utilização:

Uma aplicação necessita exibir o nome de usuários logados em partes do sistema, mas os nomes devem ser limitados somente aos dois primeiros nomes, mas no caso do segundo nome ser muito curto, até três caracteres, o terceiro nome deverá ser exibido, por exemplo, se o cliente chamar José Bandeira Lima, somente José Bandeira deverá aparecer, mas se for José de Lima Bandeira, deverá aparecer José de Lima e não José de, o método abaixo executa essa tarefa:

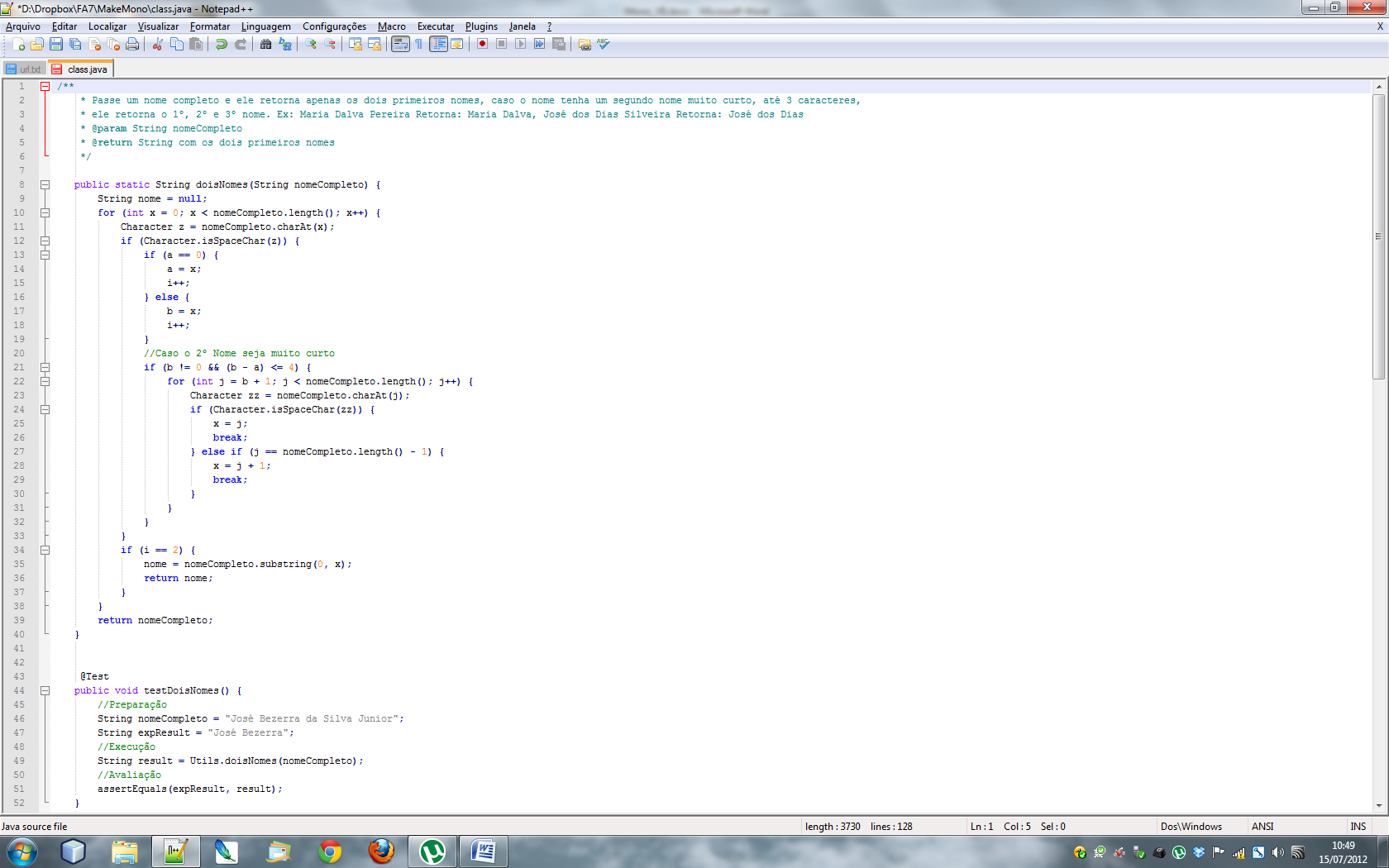


Figura 1 - Método encurtar nomes, o código foi encurtado, por motivo de espaço e foco.

Ao codificar essa funcionalidade o desenvolvedor criou o seguinte teste no JUnit:

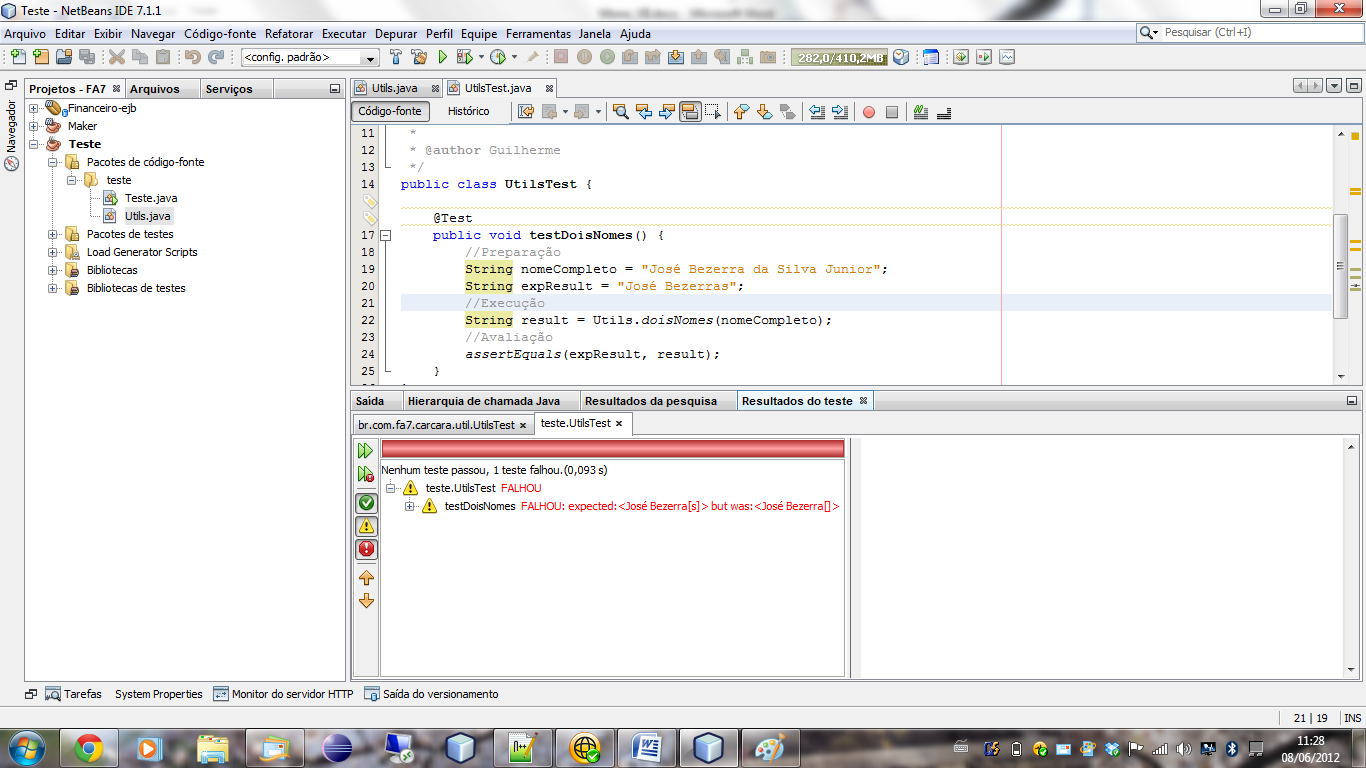


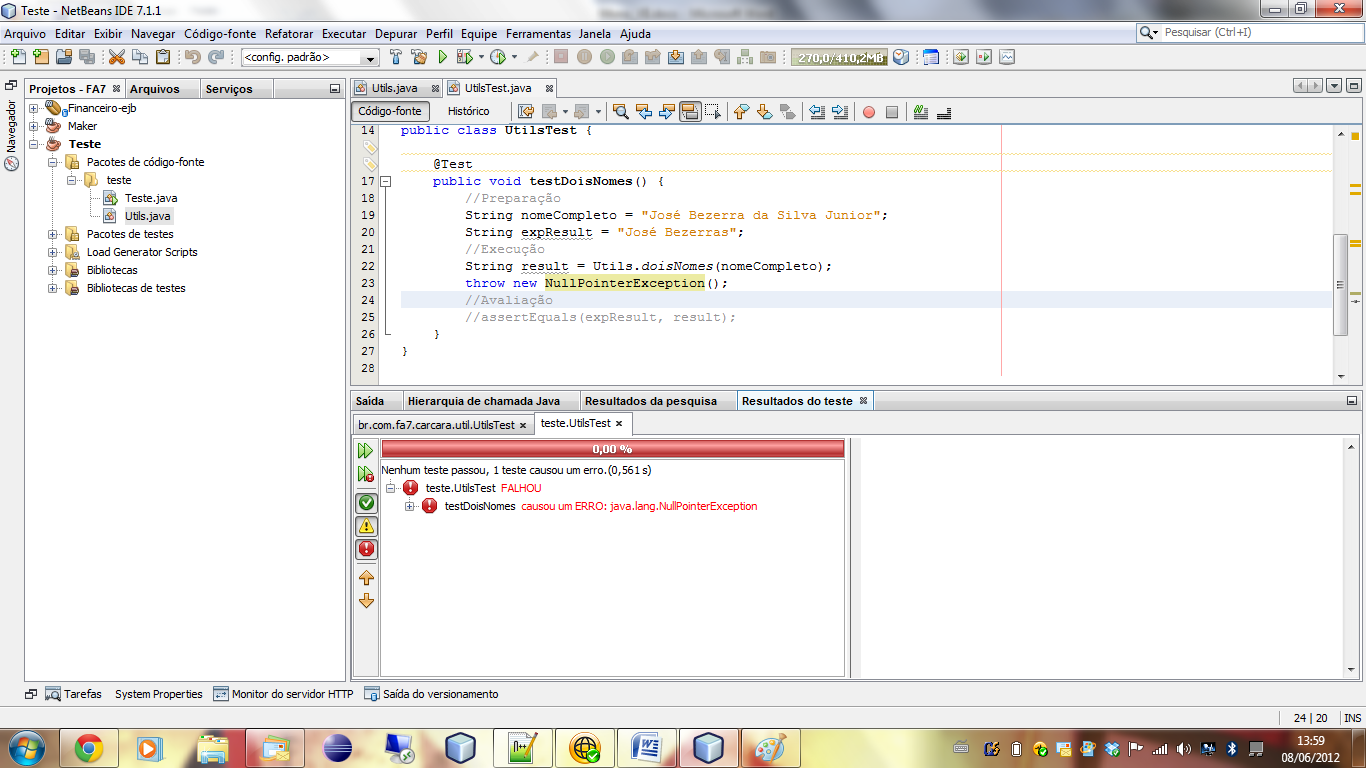
Figura 2 - Teste executado com JUnit

O teste consiste em passar um nome básico e é verificado se o método se comportou conforme esperado, retornando apenas os dois primeiros nomes do cliente.

Conforme marcado na imagem do código do teste, três estágios são visíveis, a preparação: onde o nome a ser enviado ao código fonte e o resultado esperado foram definidos, a execução do código utilizado no sistema: realizando o trabalho propriamente dito e a avaliação do resultado: onde o JUnit realiza a comparação do que se espera versus o que se obteve.

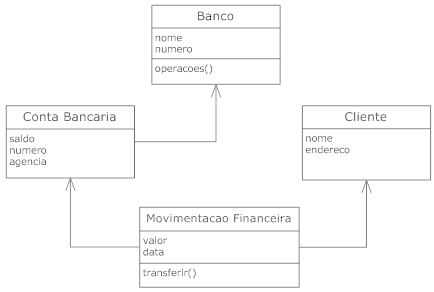
A cada teste executado o JUnit irá informar o status dos mesmos: *Figura 3 - JUnit teste passou.*

*Figura 4 - JUnit teste falhou*

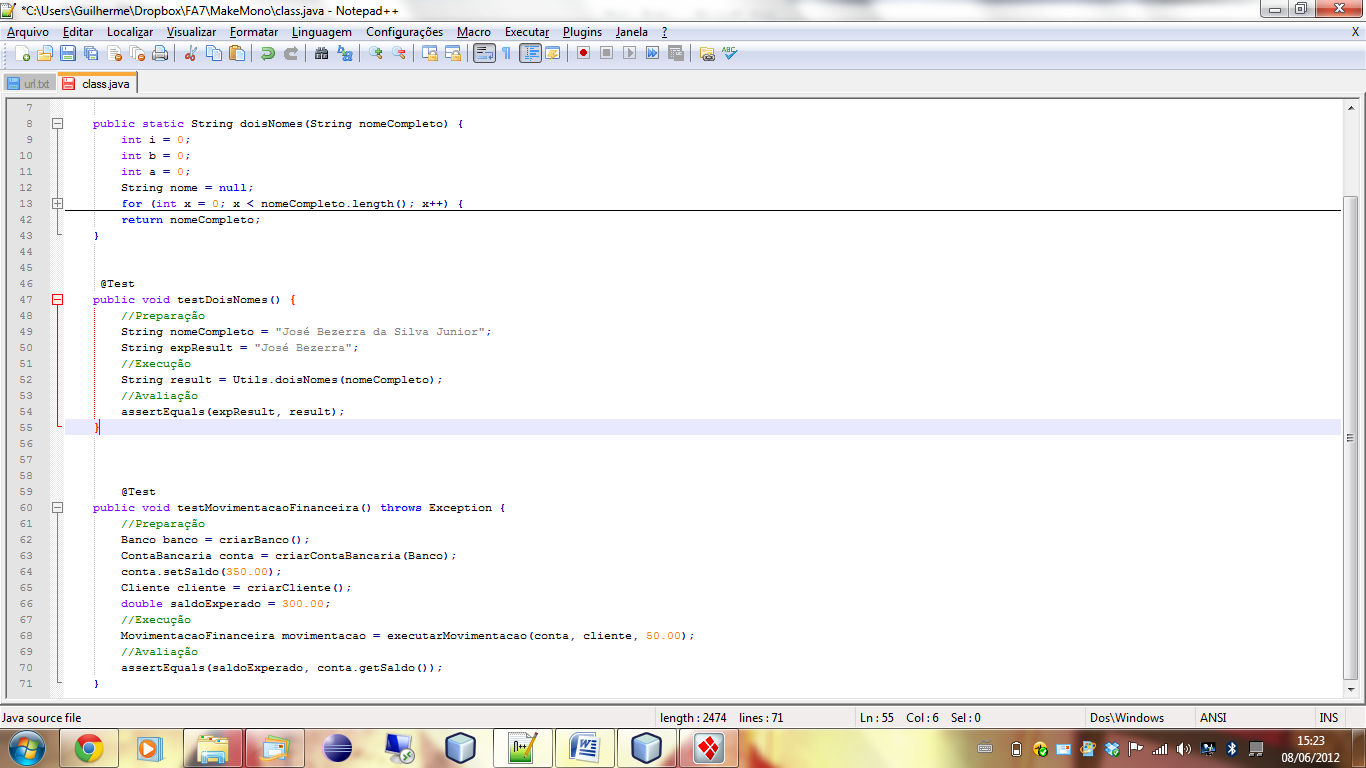
  
*Figura 5 - JUnit teste deu erro*

A demonstração apresentada mostra um teste que possui a preparação bem simples, bastava um nome para preparar o teste para ser executado. Mas, na grande maioria das vezes, a preparação de um teste é bem mais complexa, é necessário criar objetos com relacionamentos extensos, gravar o mesmo em uma base de dados e então executar o teste.

Um exemplo um pouco mais complexo, um teste realizado em uma classe que depende de outras classes, por exemplo: em um sistema financeiro temos a movimentação financeira que representa movimentações de uma conta, para realizar cada movimentação é necessária uma conta bancaria, que depende da existência de um banco. Quando for realizar a preparação do teste, será preciso criar um banco, uma conta e um cliente, para então poder executar o teste de registro da movimentação.

*******Figura 6 - Classes dependentes, um trabalho extra na preparação do teste.*

Um exemplo de um possível exemplo de teste feito no JUnit para exemplificar como teste seria:

 *Figura 7 - Teste unitário com maior preparação*

Embora esse teste pareça simples, algumas tarefas que devem ser consideradas não estão sendo exibidas, são os métodos: *criarBanco(); criarContaBancaria(Banco); criarCliente();* Estes métodos executam tarefas de criação de classes complexas para a execução do teste, muitas vezes a preparação do teste se torna um problema, desviando a atenção do teste em si para resolver problemas secundários e tornando a tarefa que deveria ser simples, muitas vezes, mais complexas que a criação do próprio código a ser testado.

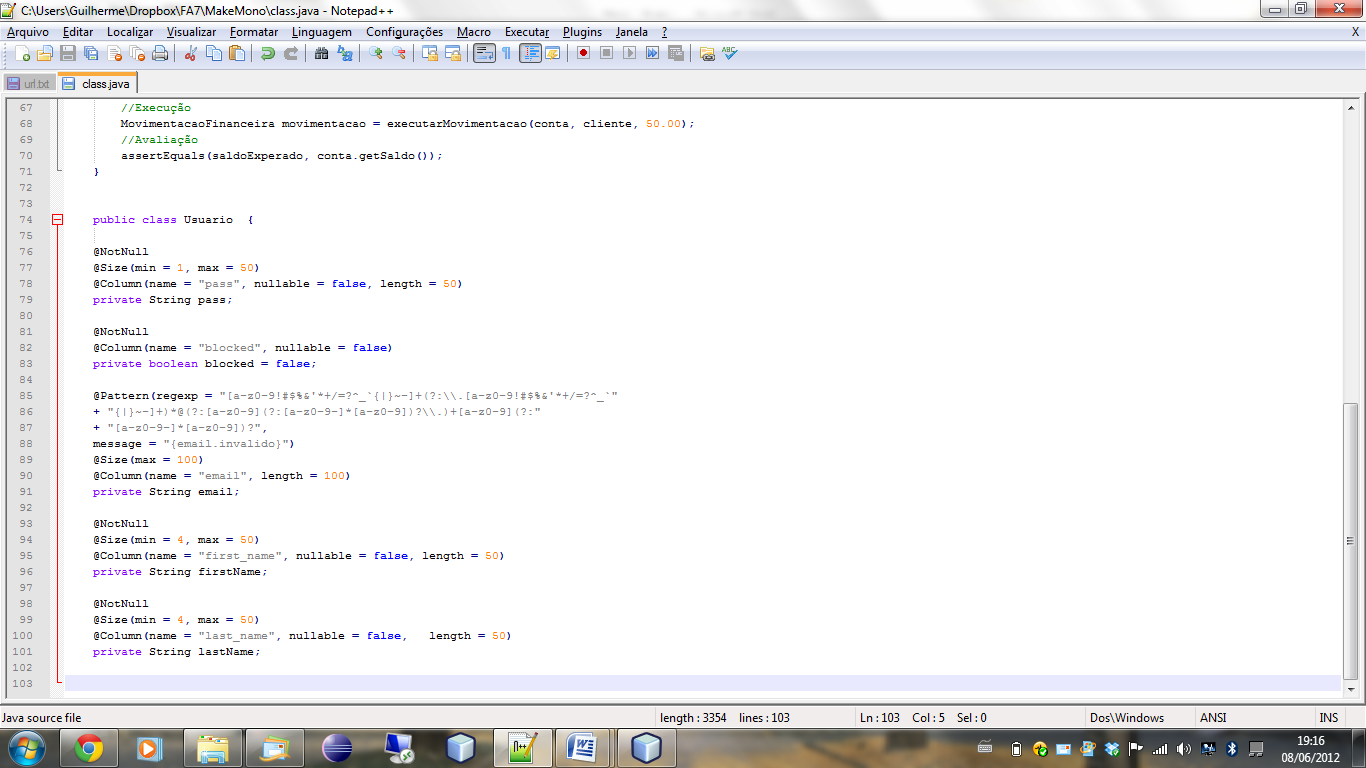
1. JSR-330
   * O que é, para que serve

Java Bean Validation (JSR 303) é uma estrutura que foi aprovada pelo JCP (Java Community Process) a partir de 16 de novembro de 2009 e é aceita como parte da especificação Java EE 6. Essa especificação define um modelo de anotações e API para validação de atributos em uma classe.

A equipe que desenvolve o Hibernate fornece com Hibernate Validator a implementação de referência do Bean Validation.

Com as anotações nos atributos da classe, a validação se torna uma tarefa simples, na Java Persistence2 (JSR 317: JavaTM Persistence 2.0), API de especificação Java para persistência de dados por meio de objeto-relacional , antes de salvar qualquer objeto na base de dados, a especificação JPA2 define que os objetos devem ser validados pelas regras da JSR303, JSR 317: JavaTM Persistence 2.0 “It is anticipated that the Java Persistence API will be aligned with the results of JSR-303 to provide support for validation.”

Outro que utiliza a especificação JSR303 é o JSF2(JSR-314 JavaServerTM Faces 2.0), framework MVC para aplicações WEB, ele realiza a validação de um objeto Java que possui as anotações da JSR303 em formulários HTML, executando a validação dos dados informados pelo usuário na submissão do mesmo. JSR-314 JavaServerTM Faces 2.0 “A JSF implentation must support JSR 303 Bean Validation if the environment in which the JSF runtime is included requires JSR 303 Bean Validation”.

Um exemplo de classe Java anotava com as anotações da especificação JSR303:  
*Figura 8 - Exemplo de classe anotada com validação JSR303*

As anotações @NotNull, @Size e @Pattern são oriundas da especificação JSR303

* + Anotações definidas na especificação:

|  |  |  |
| --- | --- | --- |
| Anotação | Descrição | Exemplo |
| @AssertFalse | O valor deve ser false, falso. | @AssertFalse  boolean isUnsupported; |
| @AssertTrue | O valor deve ser true, verdadeiro. | @AssertTrue  boolean isActive; |
| @DecimalMax | Define o valor máximo, incluindo as casas decimais. Pode ser utilizada em Strings também. | @DecimalMax("30.00")  BigDecimal discount; |
| @DecimalMin | Define o valor mínimo, incluindo as casas decimais. Pode ser utilizada em Strings também. | @DecimalMin("5.00")  BigDecimal discount; |
| @Digits | Quantidade máxima de números inteiros, antes da virgula, e a quantidade fracionária, após a virgula. Pode ser utilizada em Strings também. | @Digits(integer=6, fraction=2)  BigDecimal price; |
| @Future | Determina que um capo data seja no futuro. | @Future  Date eventDate; |
| @Max | Valor máximo que um campo numérico pode ser. Pode ser utilizada em Strings também. | @Max(10)  int quantity; |
| @Min | Valor mínimo que um campo numérico pode ser. Pode ser utilizada em Strings também. | @Min(5)  int quantity; |
| @NotNull | Determina que um valor não pode ser nulo. | @NotNull  String username; |
| @Null | Obriga que um determinado valor seja nulo. | @Null  String unusedString; |
| @Past | Determina que um capo data seja no passado. | @Past  Date birthday; |
| @Pattern | Valida um campo com base em uma expressão regular. | @Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")  String phoneNumber; |
| @Size | Determina a quantidade máxima de caracteres em uma String ou a quantidade máxima de elementos em uma coleção. | @Size(min=2, max=240)  String briefMessage; |

Tabela - Anotações padrão da especificação JSR303

Embora as anotações cubram grande parte da necessidade, ainda pode ser necessário criar anotações específicas para serem utilizadas em situações extremas de alguns projetos. Um exemplo pode ser o CPF, a validação do mesmo não pode ser feita somente com quantidade de caracteres e limitar a somente numéricos, deve também haver a validação do digito verificador. A especificação permite que desenvolvedores criem suas próprias validações, através da implementação de algumas regras definidas na especificação.

1. API Reflection Java

A Sun (ORACLE 2011), em seu site de documentações (http://docs.oracle.com/javase/tutorial/reflect/index.html), definiu a API reflexão da seguinte forma: Sun (ORACLE 2011), *“*Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.*”*

Ou seja, Reflexão é comumente utilizada por programas que necessitam examinar ou modificar o comportamento de aplicações em execução na maquina virtual Java.

Através da API Reflection Java podemos “enxergar” quase tudo que é necessário saber sobre uma classe Java.

* As variáveis declaradas: seu tipo (String, Integer, int, Boolean, Double, etc.), suas anotações (@Size, @NotNull, etc.), sua hierarquia, seu tipo de acesso (public, private),
* Os métodos: nomes, tipo de retorno, parâmetros, anotações.
* Os construtores: seus argumentos, utilização para instanciar um novo objeto.

Para o framework Make, a Reflection se torna fundamental para reduzir o trabalho do desenvolvedor para configurar o framework para gerar seus objetos. O Make, em tempo de execução irá acessar a classe solicitada, através da reflexão, e irá ler todas as variáveis declaradas na classe, sabendo assim o tipo que deve ser criado para definir o valor na variável, e avaliar as anotações da JSR303, gerando valores dentro limites de valores válidos que devem ser criados.

Sem a API Reflection o Make iria necessitar que o desenvolvedor preenchesse um arquivo de configuração para cada classe e descrevesse cada variável dessa classe, tornando o trabalho complexo e muitas vezes inviável pelo tempo gasto que iria ser necessário para criação de um teste.

Quando um dos métodos de criação do Make é chamado, o desenvolvedor passa somente a classe que ele deseja que o framework crie para ele: MakeEntity.makeEntity(final Class<T> entity), um exemplo de utilização seria: MakeEntity.makeEntity(Carro.class), aqui a utilização da API Reflection seguiria os seguintes passos através Make:

1. Uma nova instância da classe Carro seria gerada, utilizando o construtor default, ou seja, o construtor sem argumentos. É importante ser observado que todas as entidades, classes anotadas com @Entity da JPA, devem ter um construtor default, não é o Make que obriga que as classes tenham esse construtor, mas a API de persistência
2. O Make inicia a leitura da primeira variável da classe, a ordem de leitura não obedece a ordem que as variáveis são declaradas.
3. A cada variável percorrida o Make analisa as anotações da JSR303 e seu tipo.
4. Com base nas informações do terceiro passo, o Make irá gerar o valor adequado para a variável, variáveis sem a anotação @NotNull da JSR303 são ignoradas.
5. O Make insere o valor gerado na variável da classe instanciada e inicia a leitura da próxima variável da classe.

**Figura 8 - Tarefas realizadas através da Reflexão Java.**

Após a leitura de todas as variáveis e inserção dos valores nas mesmas, o Make entrega a classe com valores gerados ao desenvolvedor.

1. O framework Make
   * Propósito

O Make é um framework desenvolvido para gerar valores fictícios para realização de testes unitários ou população de banco de dados de desenvolvimento.

Os valores fictícios são limitados à validação definida nas anotações da especificação JSR303, todos os valores gerados pelo framework são valores válidos definidos pelo desenvolvedor utilizando as anotações da especificação JSR303.

O framework possibilita realizar uma restrição maior sobre os dados gerados, algumas vezes a limitação de valores dentro da abrangência da validação não é suficiente para a geração de dados que simule a realidade que a aplicação irá enfrentar. Por exemplo: um campo que recebe o valor de empréstimo de uma pessoa física. Esse campo teria uma validação mínima aceitável de R$100,00, mas não uma máxima, o Make poderá ser calibrado para gerar valores entre R$100,00 até R$1.000.000,00, ao invés de R$100,00 até   
R$ 9.223.372.036.854.775.807,00, que é o valor possível de acordo com a validação aplicada.

O framework pode ser utilizado em um sistema que não possui as anotações da JSR303, para gerar os valores o desenvolvedor deverá informar os limites ou uma faixa de valores para os campos a serem gerados.

* + Arquitetura do framework

O Make funciona sobre quatro conjuntos de tipos de classes:

* Classes que geram valores tipos primitivos, basicamente números e letras.
* Classes de valores de maior complexidade, como: datas, calendários, números de maior complexidade como BigDecimal e BigInteger.
* Classes fábricas de valores, classes utilizadas pelo Make para criar valores randômicos respeitando o tipo de valor e os limites da validação.
* Classes de fábricas especializadas, classes que geram dados dentro de um comportamento definido, por exemplo: gerando CPF aleatórios, porém válidos.

Quando falo em fabricar ou fábricas, estou realmente referenciando o padrão fábrica (Factoring Methods), uma vez que, segundo Freeman, Eric e Freeman, Elisabeth(2007, p. 105) devemos “identificar os aspectos que variam e os separar do que continua igual” (Tradução) Esse padrão, no framework Make, irá gerenciar as variações nos tipos de dados .

Quanto mais especializado o tipo de classe, maior dependência existente das classes de menor especialização, por exemplo: a classe que gera CPF possui dependência da classe que gera Strings numéricas e, esta ultima, possui dependência da classe que gera caracteres numéricos. O inverso não ocorre, as classes de menor especialização não dependem das classes de maior especialização, ou seja, a classes que geram números aleatórios não dependem de classes que geram datas.

**Figura 9 - Relação entre componentes Make**

Todas as informações geradas por essas classes são, basicamente, originadas de três classes: MakeLong, MakeDouble e MakeCharacter, essas classes são responsáveis por gerar tipos inteiros, ponto flutuante e caracteres, respectivamente e estão dentro de Primitivos Aleatórios.

Cada tipo de classe deve implementar um conjunto de regras definido por contrato, isso garante a comunicação entre as classes de maior especialização com as de menor especialização, unificando todas as dependências em um só conjunto de regras. Conforme orientado por Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides (1995 p.31)” Favor object composition over class inheritance.” Em livre tradução: “Programe para interface, não para implementação”. A interface, que também pode ser uma classe abstrata, é o contrato.

Cada tipo, primitivos, complexos, fábricas e fábricas especializadas, possui um comportamento definido, descrito a seguir:

* + 1. Primitivos Aleatórios:

As fabricas de primitivos auxiliam o Make em gerar valores aleatórios para estes tipos, essas classes possuem métodos definidos por contrato e métodos estáticos, que criam valores primitivos aleatórios delimitados pelo desenvolvedor.

As classes que geram números estendem a classe abstrata MakeNumber, essa classe possui alguns métodos implementados, métodos que possuem funcionalidades, e métodos abstratos, quem a estende deverá implementa-los.

Os métodos que MakeNumber implementa realizam o trabalho que pode ser compartilhado por todas as classes que a estendem: são métodos que toda classe primitiva que fabrica números podem utilizar, responsáveis por ver se existe anotações de validações da JSR303, tratar quantidade de dígitos (mínima ou máxima) e tratar as casas decimais dos números de ponto flutuante.

Os métodos abstratos são os contratos assinados pelas classes que estendem MakeNumber, através deste contrato as fábricas gerenciam os valores que serão gerados:

* boolean isMyType(Field): Este método informa se o primitivo atual trabalha com o tipo campo que receberá o valor. Por exemplo: se o tipo do campo for um Long ou long e isMyType for executado na classe primitiva MakeLong, a resposta será verdadeira, mas se for executado na classe MakeDouble, a resposta será falsa. Isso possibilita a fábrica numérica saber qual classe primitiva trabalha com aquele tipo numérico sem saber qual o tipo numérico real do campo.
* void insertValue(Field, Entity): Este método insere o valor na variável (Field) da classe ou entidade (Entity) limitado a validação da JSR303. Quando insertValue é chamado ele utiliza os métodos que herda de MakeNumber para saber os limites da validação da JSR303 e gerar valores dentro do limite esperado.
* void insertValue(Field, Entity, Value): A sobrecarga de insertValue é utilizado pelas fábricas especializadas, esse tipo de fabrica, como será visto mais adiante, insere o valor passado no atributo value, convertendo o mesmo para o tipo do campo. Este método não checa a validação JSR303, as fábricas especializadas devem conhecer os limites da validação e se responsabilizam em gerar os valores para a variável.

Existem dois fatores para geração de números pelo Make ser baseado em um contrato:

* A grande variedade de tipos numéricos, Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal e Strings (embora seja do tipo que recebe letras em um âmbito genérico, existem casos que devem receber apenas números).
* O segundo motivo é a variedade de anotações de validação da JSR303 para os mesmos, as anotações @Min, @Max, @Digits, @MaxDecimal e @MinDecimal.

Todas essas variações requerem especialização em leitura de anotações, checagem de tipos, checagem de valores delimitadores e geração de valores delimitados.

O mesmo trabalho não é necessário para a geração de valores textuais, o tipo texto pode variar apenas em caracteres, somente um caractere pode ser definido no valor, ou Strings, vários caracteres por valor. Somente uma anotação de validação é definida na JSR303 que pode ser utilizada com texto, @Size.

Para os métodos estáticos não existe contrato, contudo um comportamento foi definido em todas as classes que geram valores agregados. Todas as classes, com exceção de MakeBoolean e MakeCharacter, de geração de primitivos, possuem os seguintes métodos estáticos:

getIntervalo(*tipo* min, *tipo* max): cria um valor maior ou igual a min e menor ou igual a max.

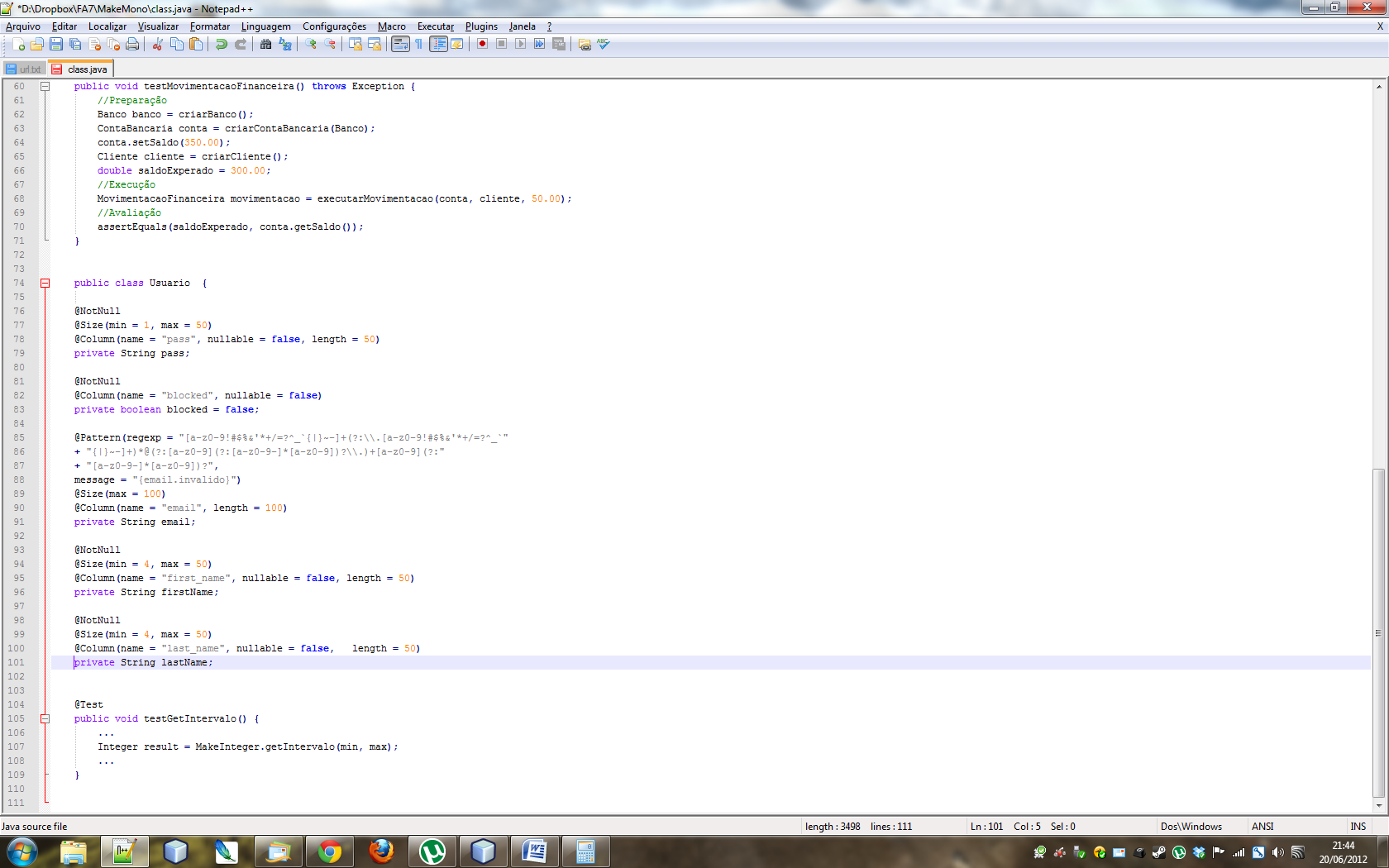
getMax(*tipo* max): cria um valor menor ou igual ao max. O mínimo aceitável como argumento é o valor 1 (um), o mínimo que será retornado é o valor zero.

Para cada classe, o tipo do argumento será o tipo gerado pela classe, por exemplo, MakeInteger o *tipo* será int, MakeLong o *tipo* será long e assim será cada um para seu tipo.

As classes que geram os valores primitivos são:

**Figura 10 - Classes Fábricas de primitivos**

Um exemplo de utilização de um dos métodos estáticos pode ser visto na figura 8, neste caso, se *min* for igual a 3 (três) e *max* for igual a 5 (cinco), o valor da variável *result* será limitado a 3 ou 4 ou 5.

**  
Figura 11 - Utilização de geração de primitivos aleatórios.**

A classe MakeBoolean cria valores booleanos aleatoriamente, possui apenas um método estático, chamado getBoolean(), que pode retornar verdadeiro ou falso.

A classe MakeCharacter pode gerar caracteres divididos em três categorias: letras, números e símbolos.

Possui um método estático para cada categoria e um genérico:

Character getLetter(): gera um caractere aleatório do tipo letra.

Character getSymbols(): gera um caractere aleatório entre as seguintes possibilidades: '!', '@', '#', '$', '&', '%', '?', '-', '+'.

Character getNumber(): gera um caractere aleatório numérico.

Character getCharacter(): gera um caractere aleatório utilizando um dos três métodos anteriores.

Os métodos estáticos possibilitam um controle adicional em um comportamento de um teste. Supondo que o Make gerou uma classe com valores válidos, o desenvolvedor pode utilizar métodos estáticos para gerar valores inválidos, ou gerar valores em intervalos menores que a validação da JSR303, ou para valores de superclasse, ou qualquer outra tarefa que o desenvolvedor necessite.

* + 1. Dados Complexos:

Os dados complexos utilizam dados primitivos de vários tipos e origem para formar uma informação. Sua criação não é simples e dependem de valores primitivos para serem gerados.

Os tipos complexos são:

* MakeString: Gera vários tipos de Strings, podendo ser Strings numéricas, String textos, String símbolos, String variadas e textos organizados. Seus métodos estáticos permitem gerar valores com comprimento variado, String getString(min, max, StringType), min e max determina o comprimento mínimo e máximo, respectivamente, da String e String type indica o tipo de String, LETTER para somente letras, NUMBER para somente números, SYMBOLL para somente símbolos e ALL para todos os tipos.  
  Outra possibilidade é utilizar o método String getString(caracteres, StringType), onde caracteres é a quantidade exata de caracteres que conterá na String.  
  Para gerar textos mais voltados para a realidade, quando o método String getLoren(caracteres) é utilizado ou quando o texto solicitado é limitado a somente letras (LETTER), o texto gerado é extraído do arquivo loren\_make.properties. Este arquivo contém duzentas e seis linhas de texto, cada uma com cem caracteres extraídos do texto Loren Ipsum. Loren Ipsum é um texto simulado da industria tipográfica, tendo a vantagem de ter uma distribuição normal de letras mas seu conteúdo real não tem sentido.   
  Ao gerar texto Loren Ipsum o Make busca aleatoriamente linhas do arquivo loren\_make.properties até que a quantidade de caracteres solicitada seja gerada.
* MakeCalendar e MakeDate: Geram objetos que representam datas, três métodos podem ser utilizados: Calendar getInFuture() ou Date getInFuture(), gera uma data no futuro, o futuro será uma data randômica gerada a partir da data do dia seguinte no momento da execução. Por exemplo: se executado dia 01/01/2010 a primeira data possível será 02/01/2010 ou maior. Calendar getInPast() ou Date getInPast() gera uma data no passado, sendo uma data randômica gerada a partir da data do dia anterior do momento da execução. Por exemplo: se executado dia 01/01/2010 a primeira data possível será 31/12/2009. Calendar getCalendar() ou Date getDate() gera uma data que pode ser futura, presente ou passado.
* MakeBigInteger e MakeBigDecimal: BingInteger e BigDecimal são classes que trabalham com números inteiros ou ponto flutuantes, respectivamente, com precisão elevada. As classes do Make que geram esses valores utilizam MakeLong para criar os valores BigInteger, e MakeDouble para gerar os valores BigDecimal. Essas duas classes possuem um método para gerar valor em um intervalo, getIntervalo(min, max) e um método para gerar valor com limite máximo de resultado e o mínimo sendo zero, getMax(max).

Todas as classes apresentadas dependem de classes que geram valores primitivos, por isso sua criação foi classificada como complexa, devido à dependência de valores e tipos primitivos.

* + 1. Fábricas de valores:

O Make possui uma classe principal para a geração de valor para uma classe, essa classe é chamada de MakeEntity, para criar uma entidade o desenvolvedor deverá chamar um dos métodos sobrecarregados MakeEntity.makeEntity:

Figura 12 - Métodos de criação de objetos populados de MakeEntity (O retorno T, indica o retorno variável de acordo com o tipo da classe passada como argumento Class<T>)

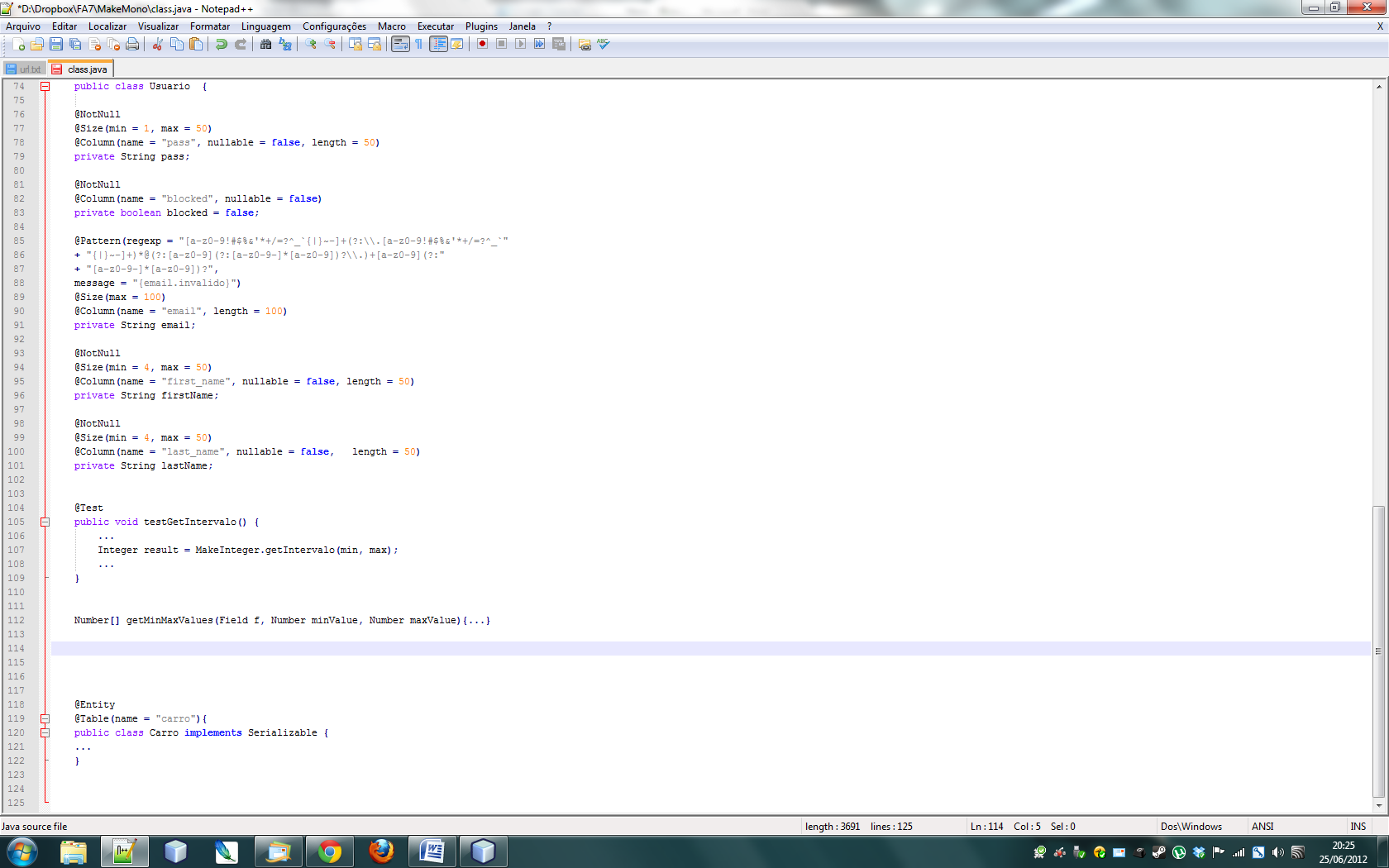
Quando um destes métodos é utilizado, internamente, o framework irá definir um conjunto de fábricas, cada uma responsável por gerar um valor para ser inserido em cada Field da entidade a ser criada.

As fábricas são responsáveis por fabricar os valores randomicamente, limitados a validação definido pelo desenvolvedor através da especificação JSR303.

**Figura** **13 - Interface ValueFactory (Contrato obrigatório para as fábricas)**

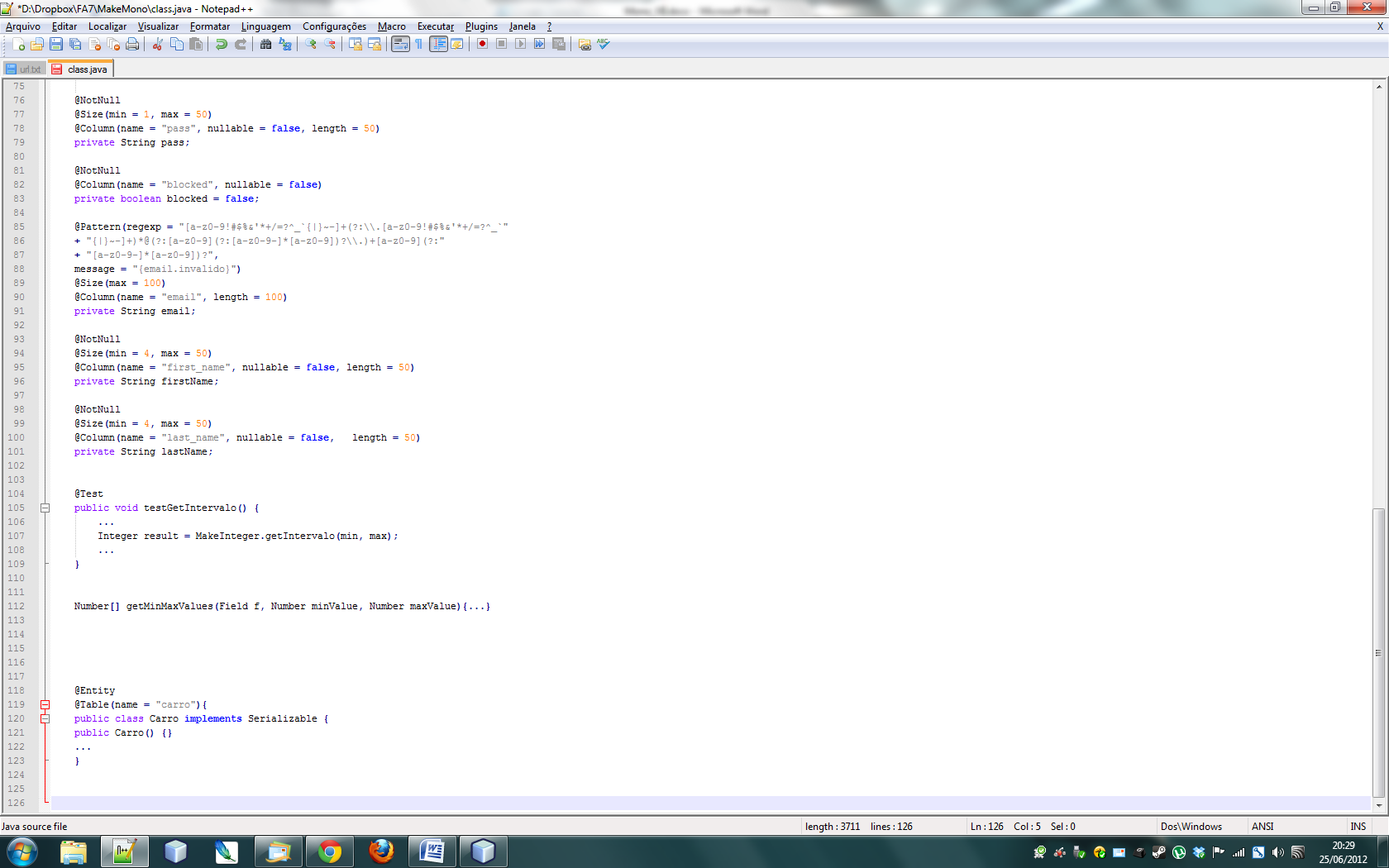
Para ser uma fábrica, é necessário que a classe implemente a interface ValueFactory, essa interface determina um contrato que todas as fábricas devem seguir, esse contrato é a implementação de métodos que definem como os valores são inseridos no Field, através do método MakeEntity.makeValue(testName, Field, entity, relationships). O atributo testName é o nome do teste, utilizado nas fábricas especializadas, Field é o que receberá o valor gerado e relationships define se entidades que se relacionam com a entidade criada deverão ser também criadas. Por exemplo: se uma classe carro for gerada pelo Make e essa possuir um Field que é uma entidade Motorista, se relationships for verdadeiro, o Motorista será também gerado pelo Make, através de recursão, o método de criação de entidades de MakeEntity é chamado novamente para a classe Motorista.

Para que objetos relacionados sejam criados duas regras devem ser observadas: a primeira é que o objeto relacionado tenha a anotação da API Java Persistence @Entity.

  
Figura - Exemplo de classe de entidade Java

Objetos Java marcados com @Entity representam um objeto Java que referencia uma tabela no banco de dados, onde cada atributo da classe tende a ser uma coluna da tabela.

A segunda regra, que também é obrigatória para classes que são anotadas com @Entity, é que a classe tenha um construtor default, ou seja, um construtor sem argumentos:

  
Figura - Exemplo de Construtor sem argumento.

Ao preparar o teste o desenvolvedor não irá escolher qual fábrica irá utilizar, essa tarefa é transparente para o mesmo. Ao executar qualquer um dos métodos de criação de entidades da classe MakeEntity, todos os Fields da entidade a ser criada serão percorridos, para cada Field é solicitado a classe Factory, que é a classe que gerencia todas as Fábricas, especializadas ou não, que determine qual fábrica deverá gerar o valor para ser inserido no Field em questão.

O método de Factory utilizado para criar uma fábrica é o método estático   
ValueFactory makeFactory(Field, entity), que recebe o Field que terá o valor criado e a entidade a ser criada. Esse método retorna uma fábrica que deve implementar a interface ValueFactory.

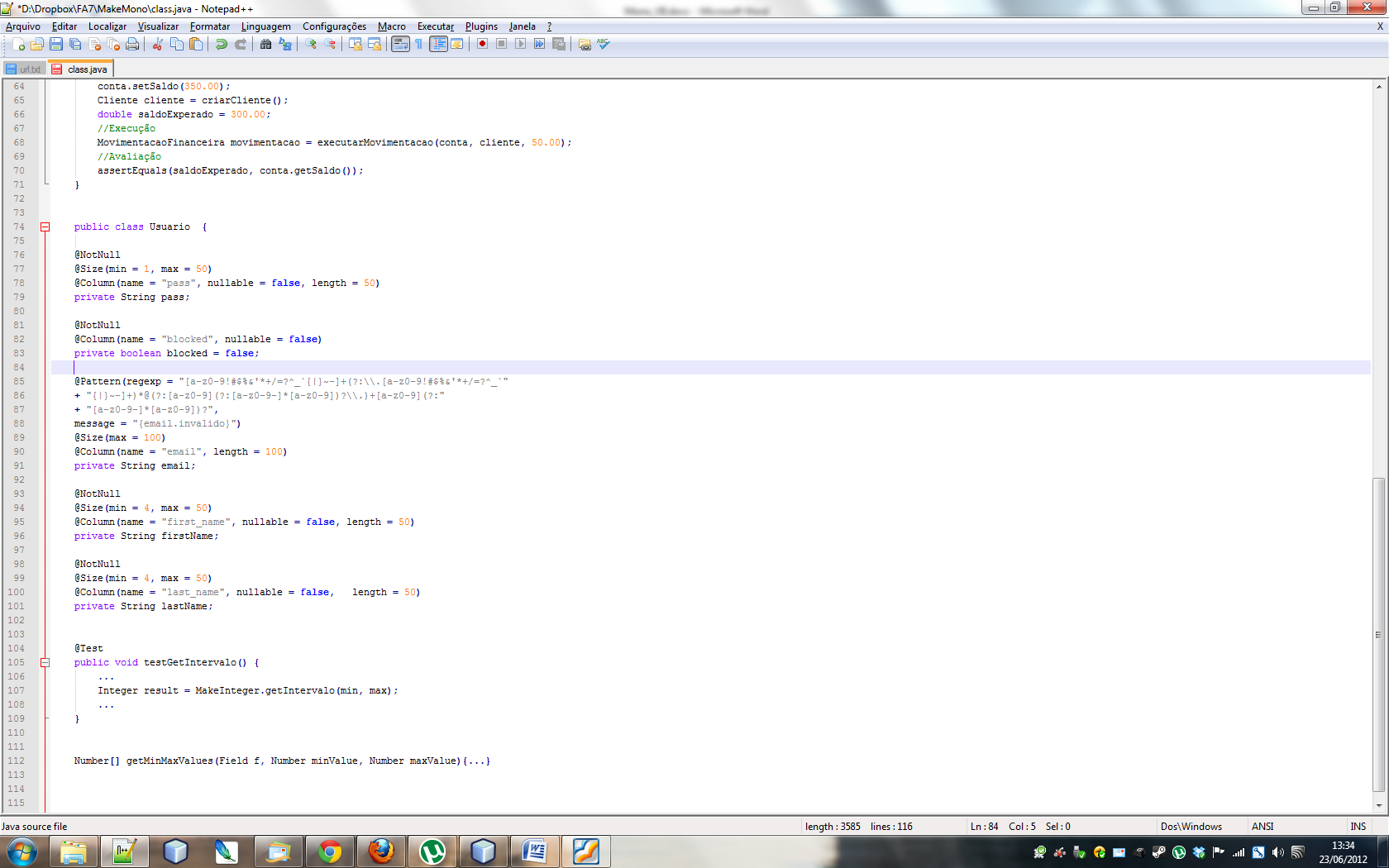
As fábricas foram divididas em tipos que trabalham para uma ou um grupo de anotações da especificação JSR303:

|  |  |
| --- | --- |
| Anotação JSR303 | Fábrica Make |
| @AssertFalse, @AssertTrue | TrueFalseFactory |
| @DecimalMax, @DecimalMin, @Digits, @Max, @Min | NumberFactory |
| @Future, @Past | DateFactory |
| @Size | SizeFactory\* |
| @NotNull | EnumFactory ou DefaultFactory |

Tabela - Fabricas Make para anotações JSR303   
\*SizeFactory trabalha com campos do tipo String, Coleções devem ser tratadas com fábricas especializadas.

As fábricas são utilizadas pelo Make ao gerar entidades, em tempo de execução o Make determina qual fábrica deverá ser executada para um determinado tipo de Field.

A especificação JSR determina que anotações como @NotNll podem ser inseridas juntamente com outras anotações, por exemplo:

  
 **Figura 16 - Duas anotações da JSR303 em um mesmo Field**

Veja que o Field do tipo String com nome de pass possui duas anotações da JSR303, @NotNull e @Size, nesta situação @NotFull não possui preferência e a geração do valor a ser inserido neste Field será da fábrica SizeFactory. Na realidade, DefaultFactory, somente será utilizado se o Field for anotado com apenas a anotação @NotNull e não houver nenhuma fábrica especializada para ele.

Duas anotações da JSR303 não foram mencionadas na tabela 2, @Null e @Pattern.

Quando um Field possui a anotação @Null o Make irá ignorar esse campo, a menos que uma fabrica especializada seja definida para o mesmo.

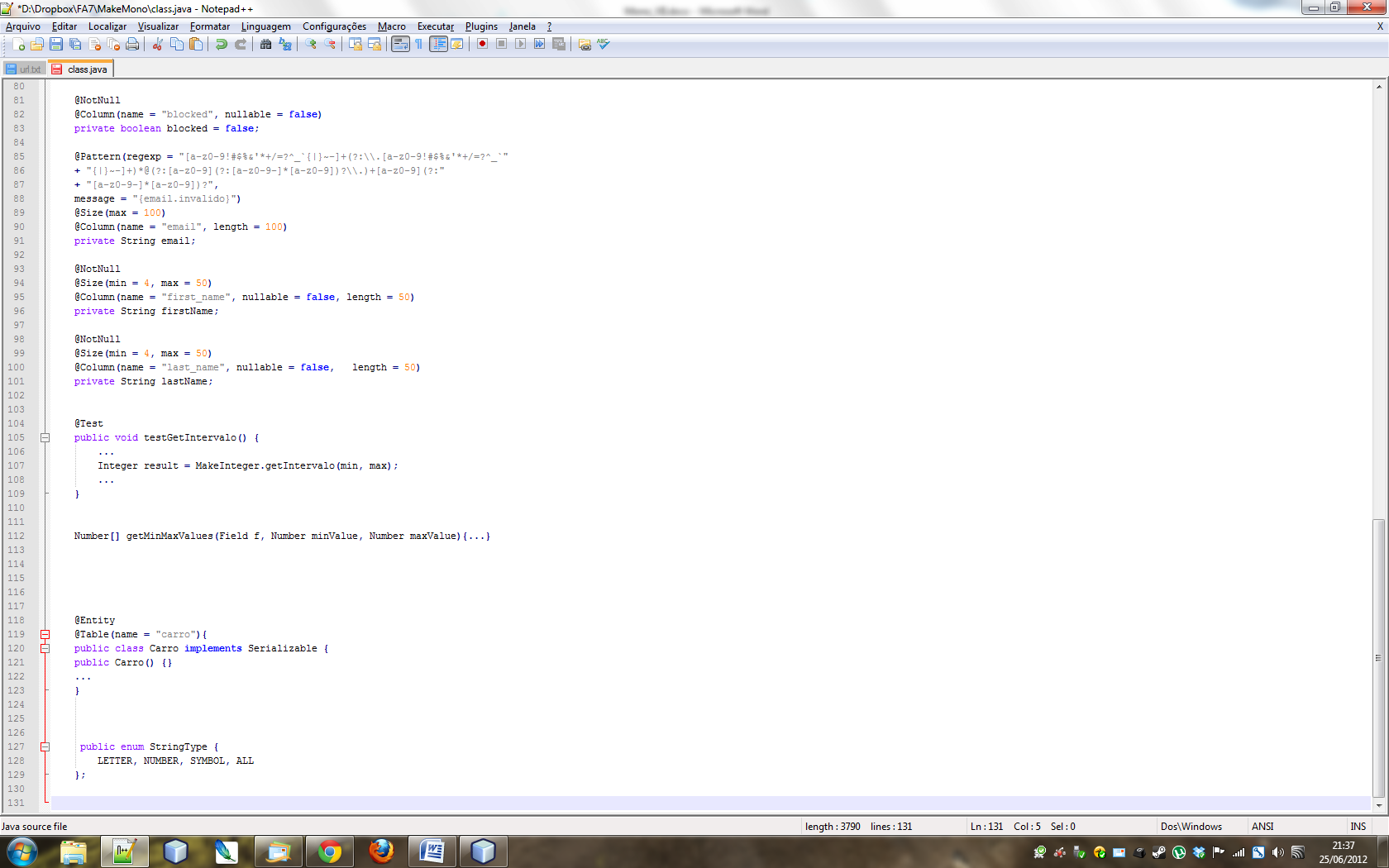
A anotação @Pattern determina que um objeto seja validado por uma expressão regular, gerar um valor com base em uma expressão regular é uma tarefa complexa e as fabricas especializadas tratam esse tipo de situação, como veremos adiante.

A anotação @Size determina o comprimento máximo de uma String ou quantidade de itens em uma coleção. A fábrica SizeFactory irá gerar um valor para ser inserido se o tipo do Field anotado for String, caso seja uma coleção essa fábrica não será utilizada e uma fábrica especializada deverá ser chamada.

Para que o valor de um Field seja criado, o Make utiliza o contrato definido para as fábricas, executando os seguintes passos:

Figura - Processo de geração de valor para ser inserido em um atributo de classe.

Para valores do tipo enum, que possuem a anotação @NotNull, é utilizado a fábrica EnumFactory, uma enum Java é uma forma elegante de definir valores limitados em uma variável declarada:

  
 Figura Exemplo de Enum Java

Por exemplo: a classe MakeString possue uma enum que é utilizada para definir o tipo de caracteres, letras (LETTER), números (NUMBER), símbolos (SYMBOL) ou todos (ALL), somente esses valores podem ser utilizados.

As posições das variáveis enum são definidas numericamente, ou seja, LETTER está na posição 0, NUMBER na posição 1 e assim sucessivamente. Para definir um valor aleatório basta definir uma enum na posição 0 até 3, neste exemplo.

Essa fábrica verifica quais as possibilidades existem para a enum e escolhe aleatoriamente, através do intervalo de posições das variáveis, qual das possibilidades inserir no Field.

DefaultFactory é a última opção de fábrica a ser chamada, essa fábrica somente irá gerar valor para o Field em questão se o mesmo for anotado com @NotNull e não for uma enum Java, caso contrário o valor do Field não será definido. Quando DefaultFactory encontra a anotação @NotNull é sinal que não existe outra anotação para o Field e/ou nenhuma fábrica especializada foi definida para o mesmo. Quando isso ocorre DefaultFactory gera o valor para o Field respeitando algumas regras definidas:

* Para Field´s numéricos primitivos: os valores de intervalo serão o máximo e o mínimo suportado pelo tipo do Field. Por exemplo, para Bytes o mínimo é -27 e o máximo é 27 -1, Short seria mínimo -215 e o máximo 215 -1.
* BigInteger ou BigDecimal: os limites são os mesmos dos primitivos long e Double, respectivamente.
* Para datas: será uma retirada dos métodos getDate() ou getCalendar() de MakeDate ou MakeCalendar, respectivamente, podendo ser passado, presente ou futuro.
* String: DefaultFactory irá respeitar as constantes MakeString.MIN\_LENGTH\_DEFAULT como quantidade mínima de caracteres e MakeString.MAX\_LENGTH\_DEFAULT para a quantidade máxima, entre 1 até 50 caracteres, respectivamente. O conteúdo será aleatório e retirado do texto Loren Ipsun, sendo somente letras.
* Character ou char: o valor será utilizado do método estático de MakeCharacter.getCharacter(), que retorna um caractere aleatório, podendo ser letra, número ou símbolos.
* Boolean: DefaultFactory retira o valor do método estático de MakeBoolean.getBoolean(), que pode ser verdadeiro ou falso.
* Entidades: Esses representam os objetos, criados pelo desenvolvedor, não representando tipos conhecidos da API Java. Caso o parâmetro relationships seja verdadeiro, DefaultFactory irá “começar tudo outra vez”. DefaultFactory irá chamar o método estático MakeEntity.makeEntity novamente para criar o objeto a ser inserido no Field da entidade solicitada. Para evitar uma referência cíclica, o Make cria somente três objetos entidades do mesmo tipo, a partir da quarta solicitação ele irá inserir objetos criados anteriormente.

* + 1. Fábricas especializadas:

As fábricas pré-existentes no Make criam valores válidos para as validações padrão da JSR303, mas em alguns momentos isso não será suficiente. Por exemplo, um desenvolvedor necessita que em um Field de uma classe seja um valor referente a um CNPJ, que é do tipo String, o valor utilizado no teste obriga que esse Field seja gerado dentro da validação esperada para um CNPJ, as fábricas comuns não irão conseguir executar essa tarefa, mas uma fábrica especializada pode.

Uma fábrica especializada, além assinar o contrato de uma fábrica comum, através da interface ValueFactoty, deve também assinar o contrato de especialização através da interface ValueSpecializedFactory.

Figura - Interface ValueSpecializedFactory (Herda o contrato obrigatório para as fábricas \* acrescentando o método workValue)

Quando a fábrica assina o contrato com ValueSpecializedFactory, automaticamente está assinando o contrato com ValueFactory. Portanto uma fábrica especializada deverá conter os métodos de ValueFactory, mais os métodos que existirem no contrato de ValueSpecializedFactory.

ValueSpecializedFactory declara um único método: boolean workValue(String expressao); Esse método recebe uma expressão, que é uma String, a fábrica especializada neste método interpreta se essa String é a sua chamada para criar o valor para o Field, respondendo se trabalha com o mesmo.

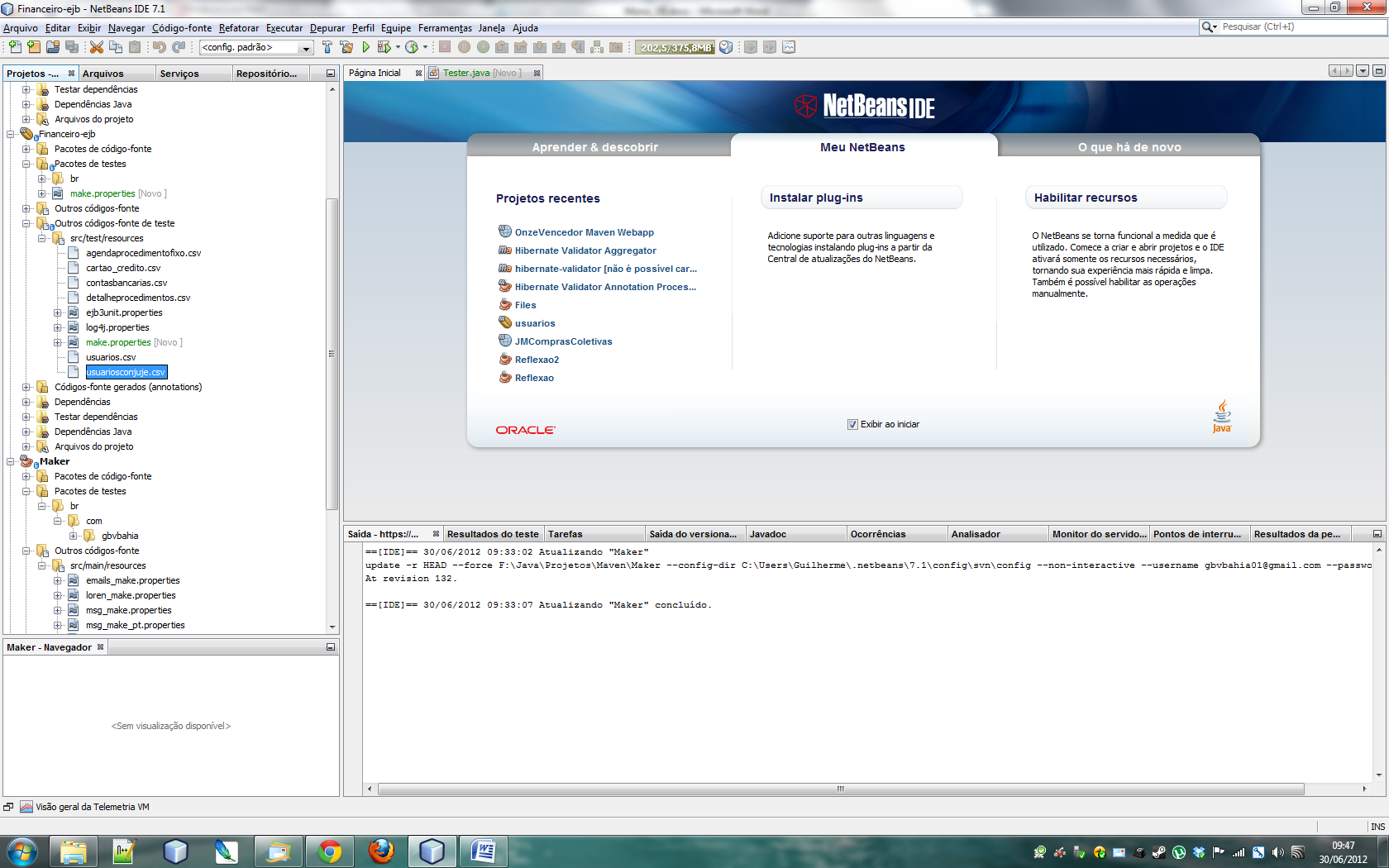
O Make possui algumas fábricas especializadas, mas existe a abertura para que o desenvolvedor crie sua própria fábrica especializada, quantas forem necessárias para a execução de um teste, ou seja, se um Field requer uma tarefa complexa, tudo pode ser encapsulado em uma fábrica especializada e reutilizado ao longo dos testes do projeto.

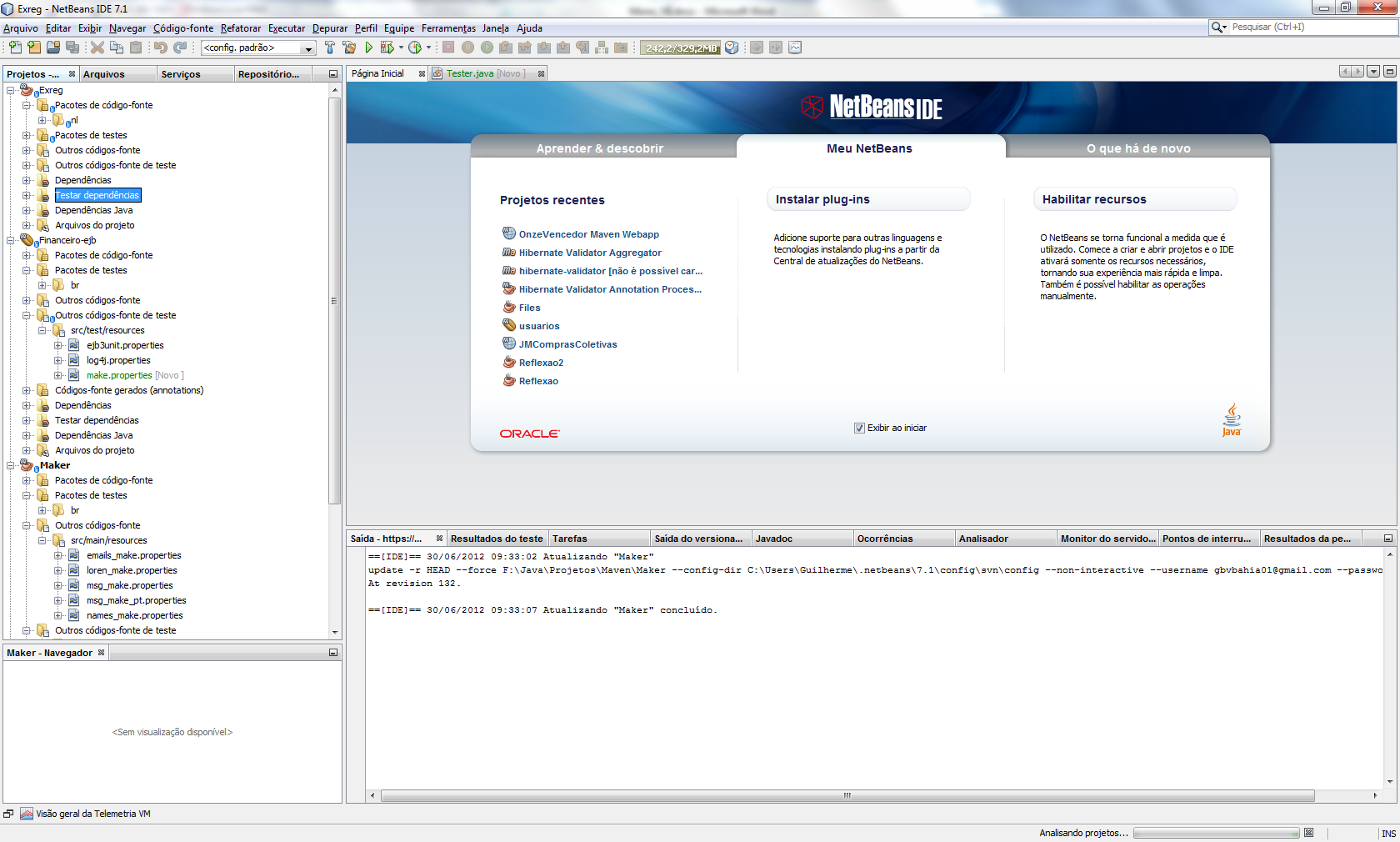
Irei descrever sobre o que cada fábrica especializada do Make pode fazer, em seguida, irei explicar como configurar uma fábrica especializada e posteriormente em como criar uma fábrica personalizada.

|  |  |  |  |
| --- | --- | --- | --- |
| Fábrica Especializada | Tarefa executada | Expressão | Exemplo |
| MakeBetween | Cria valores numéricos limitados a um intervalo. Pode ser utilizado em qualquer Field numérico. | between{i,f} | between{30, 50} (Somente valores entre 30 e 50 serão gerados |
| MakeIn | Insere valores limitados a lista informada. O que estiver entre colchetes será o separador da lista. O caractere separador default é a virgula “,”. Pode ser utilizado em qualquer Field numérico ou String | In{\*...}[?] | In{a|v|1|3|4}[|]  (Insere um dos valores a,v,1,3 ou 4 em um Field.) |
| MakeList | Cria uma java.util.List com objetos do tipo da classe da entidade informada. | isLit{classe}[i,f] | isList{br.com.Carro}[1,4] (Cria uma lista com objetos Carro, a lista terá de 1 a 4 objetos) |
| MakeSet | Cria uma java.util.Set com objetos do tipo da classe da entidade informada. Este requer atenção ao método equals, por não receber objetos duplicados a quantidade de elementos pode ser menor que esperada. | isSet{classe}[i,f] | isSet{br.com.Carro}[1,4] (Cria um set com objetos Carro, a lista terá de 1 a 4 objetos) |
| MakeName | Cria String no padrão de nomes. | isName | isName  (Cria uma String baseada em nomes do arquivo interno names\_make.properties) |
| MakeEmail | Cria String no formato de e-mails | isEmail | isEmail |
| MakeCPF | Cria String numérica referente a um CPF simulado poderem válido. | isCPF | isCPF  (Gera um CPF com dígito verificador correto) |
| MakeCNPJ | Cria String numérica referente a um CNPJ simulado poderem válido. | isCNPJ | isCNPJ  (Gera um CNPJ com dígito verificador correto) |

Tabela Fábricas especializadas Make

Para utilizar uma fábrica especializada é necessário criar o arquivo make.properties na raiz do código fonte do teste, o arquivo contém informações do tipo chave = valor:

  
**Figura 20 - make.properties raíz do código fonte.**

  
**Figura 21 - make.properties em um projeto Maven.**

Dentro deste arquivo especificam-se as fábricas especializadas que devem ser utilizadas para os Fields que necessitem da mesma. A forma de vincular um Field a uma fábrica especializada no arquivo make.properties é seguinte:

*<nome do teste>.<nome da classe >.<nome do Field> = <expressão da fábrica especializada>*

O nome da classe deve ser totalmente qualificado, incluindo os pontos de separação entre os pacotes. Exemplo: a classe br.com.exemplo.Pessoa possui vários atributos, dentre eles, um é o CPF (declarado como cpf na classe), em um teste o desenvolvedor necessita que esse valor seja criado com um valor de CPF válido, o Make possui a fábrica especializada MakeCPF que pode ser utilizada para criar o valor para o desenvolvedor no arquivo make.properties da seguinte maneira:

testepessoa1.br.com.exemplo.Pessoa.cpf = isCPF

Nome do teste: testepessoa1

Nome da classe: br.com.exemplo.Pessoa

Nome do Field: cpf

Expressão da fábrica especializada: isCPF

Para utilizar a fabrica especializada na criação da entidade Pessoa o método que recebe o nome do teste de MakeEntity deve ser utilizado, neste caso:

Pessoa pessoa = MakeEntity. makeEntity(“testepessoa1”, Pessoa.class);

Neste caso a classe pessoa seria criada com seus atributos não nulos criados pelas fábricas comuns, mas o atributo CPF seria criado pela fábrica especializada MakeCPF.

Para garantir que uma fábrica especializada tenha sempre a preferência de criação de valores no lugar das fábricas comuns, o Make utiliza uma fábrica responsável por gerenciar todas as fábricas especializadas, MakeWorksFactory. Ao iniciar a procura por uma fábrica a primeira fábrica a tratar o Field será MakeWorksFactory, ela irá garantir que se houver alguma fábrica especializada que trate a expressão declarada para o Field esta será utilizada para criar o valor para o mesmo.

Para o caso de fábricas especializadas desenvolvidas pelo desenvolvedor do teste, estas terão preferência as classes especializadas pré-existentes no Make.

No exemplo dado da classe pessoa, se houvesse outro atributo na mesma classe, sendo este também necessário utilizar uma segunda fábrica especializada, bastava declarar como realizado no CPF. Supondo que fosse necessário especificar a idade entre 0 até 17 anos, que não podem ser limitados pela JSR303, mas somente em testes:

testepessoa1.br.com.exemplo.Pessoa.cpf = isCPF

testepessoa1.br.com.exemplo.Pessoa.idade = between{0,17}

Neste caso o atributo idade de Pessoa será criado entre os valores 0 e 17 anos e o CPF será válido, dentro da validação do CPF. Mas contendo valores totalmente aleatórios.

Outra necessidade poderia ser valores de maior especificação para classes relacionadas, supondo que Pessoa possui um relacionamento com Empresa e que empresa possui um Field CNPJ (declarado como cnpj), seria possível utilizar uma fábrica especializada no mesmo teste para uma diferente classe, mas neste caso, por ser um relacionamento que deve ser gerado pelo Make, o método de criação deveria ser o seguinte:

Pessoa pessoa = MakeEntity. makeEntity(“testepessoa1”, Pessoa.class, true);

No make.properties deveria ser declarado as três fábricas:

testepessoa1.br.com.exemplo.Pessoa.cpf = isCPF

testepessoa1.br.com.exemplo.Pessoa.idade = between{0,17}

testepessoa1.br.com.exemplo.Empresa.cnpj = isCNPJ

Existem várias necessidades de geração de valores especializados em cada Entidade de cada sistema, é impossível prever todas as necessidades de informação especializada e introduzir no Make uma fábrica especializada para cada necessidade. Por este motivo, o Make possibilita ao desenvolvedor criar sua própria fábrica especializada.

Para executar essa tarefa dois passos devem ser seguidos:

1. Criar uma classe e implementar a interface ValueSpecializedFactory.
2. Registrar a fábrica especializada no arquivo Make.properties.

Para exemplificar vou demonstrar como criar uma fábrica especializada que gere CEP´s, de ruas de forma aleatória.

1. Crio a classe CEPWork que implementa ValueSpecializedFactory:

package br.com.gbvbahia.maker.properties;

public class CEPWork implements ValueSpecializedFactory { }

1. Defino que a assinatura de identificação para utilização de CEPWork será isCEP:

package br.com.gbvbahia.maker.properties;

public class CEPWork implements ValueSpecializedFactory {

public static final String KEY\_PROPERTIE = "isCEP";

}

Deste modo, no arquivo Make.properties, o Field que for anotado com isCEP: teste1.br.com.Endereco.cep = isCEP terá a classe CEPWork como fábrica especializada.

1. Implemento o método de ValueSpecializedFactory, boolean workValue(String value):

package br.com.gbvbahia.maker.properties;

public class CEPWork implements ValueSpecializedFactory {

public static final String KEY\_PROPERTIE = "isCEP";

}

public boolean workValue(String value) {

if (KEY\_PROPERTIE.equals(StringUtils.trim(value))) {

return true;

}

return false;

}

}

O método workValue irá receber o valor declarado no arquivo make.properties, sendo isCEP o método retorna verdadeiro, caso contrário irá retornar falso.

1. Implementos os métodos de ValueFactory, o primeiro é boolean isWorkWith(Field f, T entity), que deverá retornar verdadeiro se trabalha para o tipo de classe do Field passado, neste exemplo, CEPWork aceita somente tipos Strings:

package br.com.gbvbahia.maker.properties;

public class CEPWork implements ValueSpecializedFactory {

public static final String KEY\_PROPERTIE = "isCEP";

}

public boolean workValue(String value) {

if (KEY\_PROPERTIE.equals(StringUtils.trim(value))) {

return true;

}

return false;

}

public <T> boolean isWorkWith(Field f, T entity) {

return f.getType().equals(String.class);

}

}

1. Por ultimo, o segundo método de ValueFactory é implementado, makeValue:

package br.com.gbvbahia.maker.properties;

public class CEPWork implements ValueSpecializedFactory {

public static final String KEY\_PROPERTIE = "isCEP";

}

public boolean workValue(String value) {

if (KEY\_PROPERTIE.equals(StringUtils.trim(value))) {

return true;

}

return false;

}

public <T> boolean isWorkWith(Field f, T entity) {

return f.getType().equals(String.class);

}

public <T> void makeValue(final String testName, final Field f,

final T entity, final boolean makeRelationships)

throws IllegalAccessException, IllegalArgumentException {

f.set(entity, MakeString.getString(6, MakeString.StringType.NUMBER));

}

}

O método makeValue irá executar a tarefa de criar um valor e inserir no Field passado como parâmetro, veja que makeValue recebe como parâmetro o nome do teste, o Field a ser populado, a entidade que contém o Field que está sendo gerado e a variável referente a gerar relacionamentos com outras entidades. Isso possibilita que a produção da fábrica especializada possa chamar novamente MakeEntity para gerar novos objetos.

Para que o Make possa reconhecer a fábrica especializada criada basta declarar a mesma no arquivo make.properties da seguinte forma:

*<work>\_? = <nome da classe >*

A declaração deve começar com a palavra “work” seguido de underscore mais um caractere coringa, este último se faz necessário porque a chave no arquivo properties não pode ser repetida, se isso ocorrer a segunda chave com o mesmo valor será ignorada e somente uma fabrica especializada seria considerada.

A classe CEPWork seria declarada da seguinte forma:

work\_1 = br.com.gbvbahia.maker.properties.CEPWorkTest

Por utilizar a palavra work para declarar as classes de fábricas especializadas os testes não podem ser nomeados começando com work, se isso ocorrer o Make não irá gerar a entidade e lançara uma exceção de runtime.

* + Benefícios
    1. Preparação de testes unitários

Quando o desenvolvedor inicia a fase de preparação do teste unitário, na maioria das vezes, necessita criar um ou mais objetos para realizar um teste, muitas vezes, um único campo de uma classe será testado, mas o desenvolvedor necessita popular todos os campos da classe para que possa realizar o teste. Esse trabalho fica ainda mais pesado, quando é necessário o envolvimento de várias classes.

O framework Make foi desenvolvido para gerar valores aleatórios, respeitando a validação esperada para cada atributo de classe, ou seja, se um atributo de uma classe pode receber apenas valores que estejam entre 0 e 10, o Make somente irá gerar números entre 0 e 10 para este atributo.

Para gerar valores válidos o Make lê as anotações da especificação JSR303, respeitando as regras especificadas nas anotações para gerar os valores, ou seja, os valores criados pelo Make são valores válidos, que passam pela validação feita pela especificação JSR303.

Quando utilizar o framework Make, o desenvolvedor irá focar na preparação dos campos com valores a serem testados, reduzindo o trabalho com os campos não participantes.

* + 1. Popular banco de dados de desenvolvimento

Algumas aplicações trabalham com informações sigilosas, que não podem ser expostas, nem mesmo aos desenvolvedores que desenvolvem e prestam manutenção ao sistema. Um exemplo seria uma base de dados de clientes de uma empresa de cartão de crédito.

Algumas empresas misturam as informações contidas na base de dados de produção para popular a base de desenvolvimento, o procedimento para realizar essa tarefa é complexo, demorado e requer grande esforço de validação, após a mistura ser realizada deverá ser feita uma checagem dos dados, todas as informações misturadas devem ser avaliadas, os dados misturados não podem coincidir com a realidade.

O Make pode ser utilizado para gerar informações válidas, criar relacionamentos de entidades, sendo calibrado para gerar uma informação totalmente falsa, ou seja, irreal, mas ao mesmo tempo, uma informação que pode ser inserida no banco de dados de desenvolvimento.

1. Conclusões
2. Bibliografia

SOMMERVILLE, Ian. Engenharia de softwre, 8ª edição; tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa; revisão técnica: Kechi Kirama. 8ª edição, São Paulo, Person Addison-Wesley, 2007.

FREEMAN, Eric. FREEMAN, Elisabeth. Padrões de Projeto Use a cabeça, 2ª edição, Rio de Janeiro: Alta Books, 2005.

SIERRA, Kathy. BATES, Bert. Certificação Sun para Programador Java 6: Guia de Estudo, 1ª edição, Rio de Janeiro: Alta Books,2008.

GAMMA , Erich. HELM, Richard. JOHNSON, Ralph. VLISSIDES , John M. Design Patterns: Elements of Reusable Object-Oriented Software, 1ª edição, Estados Unidos, Addison-Wesley, 2000.

ELIZA, Renata. LAGARES, Vivian. Processo de teste de software: Garantindo mais qualidade com a implantação de um novo processo. Java Magazine, Rio de Janeiro, Nº101, 70-74, Março 2012.

LINDEN, Ricardo. Começando a pensar reflexivamente. MundoJ, Rio de Janeiro, 46, 9-19, Mar/Abr 2011.

JAVA COMMUNITY PROCESS, JSR 303: Bean Validation. Disponível em: <http://jcp.org/en/jsr/detail?id=303>. Acesso em 13 ago. 2012

JAVA COMMUNITY PROCESS, JSR 314: JavaServerTM Faces 2.0. Disponível em: < http://jcp.org/aboutJava/communityprocess/final/jsr314/index.html>. Acesso em 13 ago. 2012.

JAVA COMMUNITY PROCESS, JSR 317: JavaTM Persistence 2.0. Disponível em: < http://jcp.org/en/jsr/detail?id=317>. Acesso em 13 ago. 2012

THE JAVA TUTORIALS, The Reflection API. Disponível em <http://docs.oracle.com/javase/tutorial/reflect/index.html>. Acesso em 13 ago. 2012.