Sumário

[1. Introdução 3](#_Toc335493858)

[2. Testes de software 4](#_Toc335493859)

[2.1. Definição: 4](#_Toc335493860)

[2.2. Os estágios do teste de software: 5](#_Toc335493861)

[2.3. Ciclo do teste de software 6](#_Toc335493862)

[2.4. Teste de unidade: 7](#_Toc335493863)

[2.4.1. Detalhamento do ciclo do teste unitário: 8](#_Toc335493864)

[2.4.2. Exemplo de teste unitário com JUnit 10](#_Toc335493865)

[2.5. Conclusão 13](#_Toc335493866)

[3. JSR-303 14](#_Toc335493867)

[3.1. Definição 14](#_Toc335493868)

[4. API Reflection Java 18](#_Toc335493869)

[4.1. Definição 18](#_Toc335493870)

[4.2. Utilização 18](#_Toc335493871)

[5. O Framework Make 21](#_Toc335493872)

[5.1. Antes do make 21](#_Toc335493873)

[5.2. Propósito 21](#_Toc335493874)

[5.3. Utilização 22](#_Toc335493875)

[5.3.1. Métodos de chamada 22](#_Toc335493876)

[5.3.2. Utilizando as fábricas especializadas 23](#_Toc335493877)

[5.3.3. Criando fábricas especializadas 25](#_Toc335493878)

[5.2.4. Reunindo tudo 27](#_Toc335493879)

[5.3. Arquitetura do framework 31](#_Toc335493880)

[5.3.1. Reflection no Make 32](#_Toc335493881)

[5.3.2. Camadas de distribuição: 33](#_Toc335493882)

[5.3.3. Primitivos aleatórios: 34](#_Toc335493883)

[5.3.4. Fábricas numéricas 37](#_Toc335493884)

[5.3.5. Fábricas de valores: 39](#_Toc335493885)

[5.3.6. Fábricas especializadas: 44](#_Toc335493886)

[5.4. Benefícios 47](#_Toc335493887)

[5.4.1. Preparação de testes unitários 47](#_Toc335493888)

[5.4.2. Popular banco de dados de desenvolvimento 48](#_Toc335493889)

[6. Conclusões 49](#_Toc335493890)

[7. Bibliografia 50](#_Toc335493891)

1. Introdução
2. Testes de software
   1. Definição:

Teste de software é atividade de verificar se o sistema está se comportando dentro da sua especificação e da maneira esperada em relação à expectativa do cliente.

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, 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, menor será a quantidade de falhas e defeitos, reduzindo a necessidade de trabalho para descobrir as causas.

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, requisitos e regras, com isso, os testes criados serão mais abrangentes na necessidade, avaliando a maior quantidade de risco possível com mais consistência, pontos irrelevantes serão desconsiderados.

Testes elaborados após o desenvolvimento de grande quantidade de funcionalidades podem causar dois impactos negativos aos testes:

1. Aumentar a consideração em riscos que deveriam ser ignorados ou serem elevados a uma escala de risco menor, gastando menos tempo em analises e execução de testes para estes.
2. Reduzir a consideração há riscos potenciais podendo até mesmo ignorar o mesmo na realização dos testes, podendo ignorar pontos cruciais que podem causar erros no funcionamento do sistema ou as regras de negócio.

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[...]”.

* 1. Os estágios do teste de software:
* 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.
  1. Ciclo do teste de software

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.

Figura - Ciclo do Teste

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.

* 1. 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 será 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, considerando 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 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.

* + 1. 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 deve ser realizado.
* 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. Após a preparação é definido um resultado esperado com base nos insumos gerados, quando o teste sobre o código de produção termina, tudo que é esperado é comparado com o resultado do código de produção.
* 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.

Figura - Ciclo do Teste, a preparação pode ser a tarefa mais complexa.

* + 1. 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:

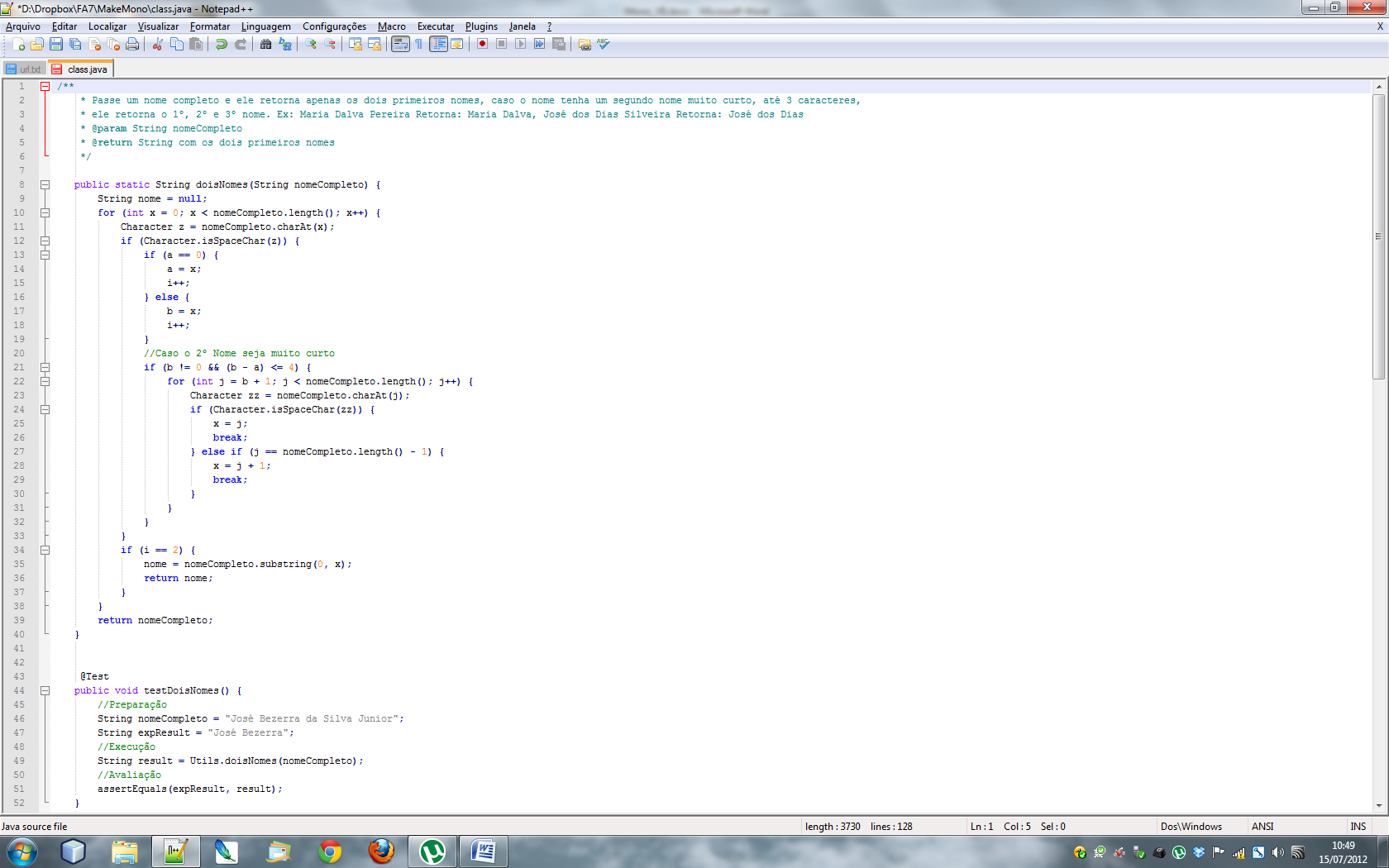


Figura 3 - Método encurtar nomes, o código foi encurtado, por motivo de espaço e foco.

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



Figura 4 - 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.

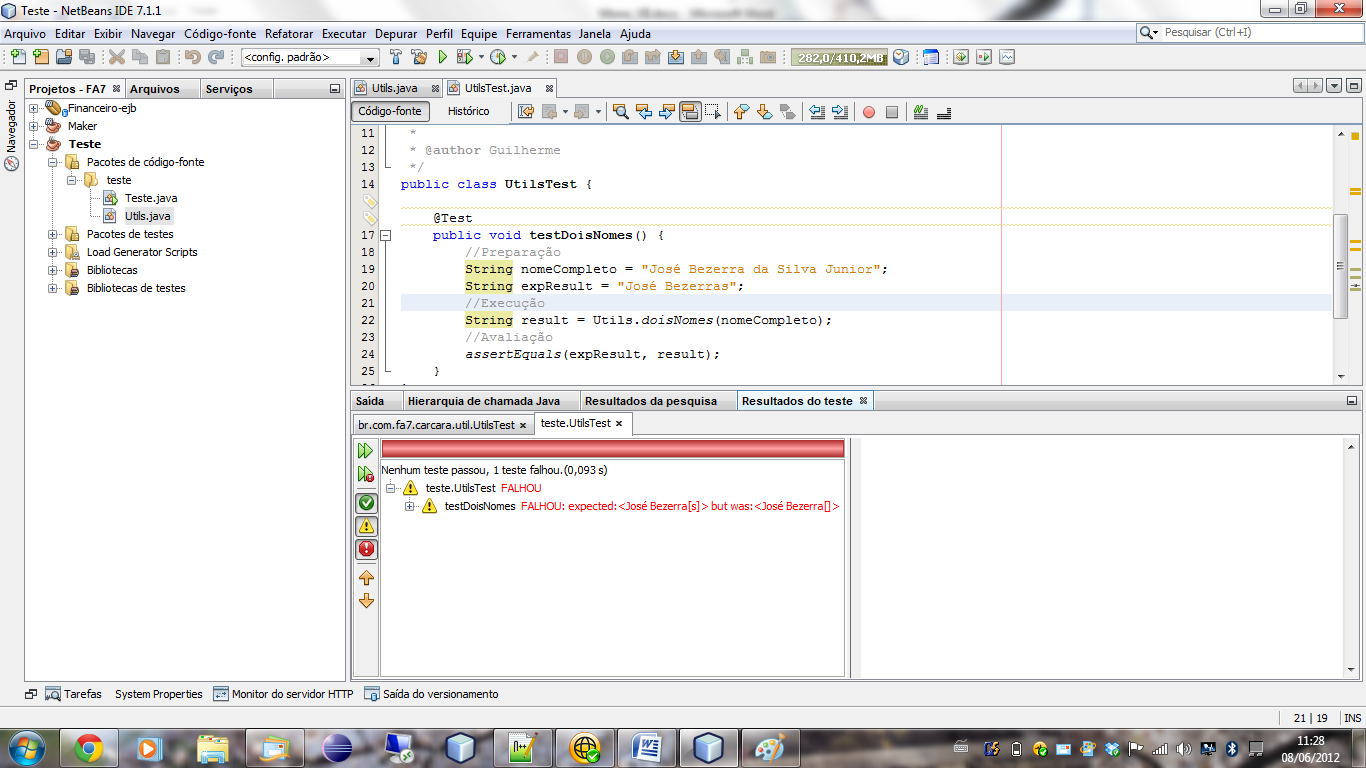
Conforme descrito na imagem do código do teste, três estágios são visíveis, a preparação: onde o nome a ser enviado ao código fonte e o resultado esperado foram definidos, a execução do código utilizado no sistema: realizando o trabalho propriamente dito e a avaliação do resultado: onde o JUnit realiza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

A cada teste executado o JUnit irá informar o status dos mesmos:

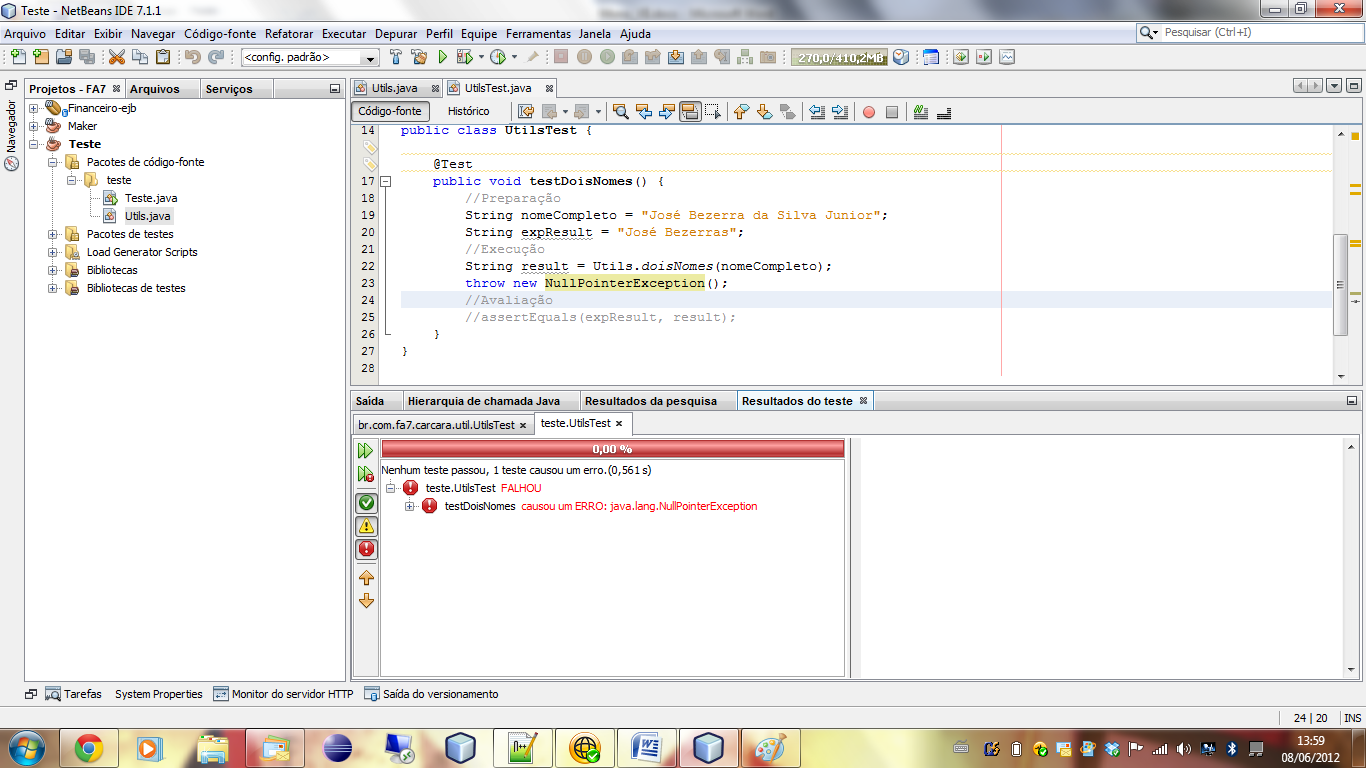
Quando os testes executarem sem nenhuma falha, quando o resultado esperado é igual ao resultado obtido no teste, o JUnit irá exibir um resultado semelhante a figura 5.

  
Figura 5 - JUnit teste passou.

Ocorrendo alguma falha, o resultado esperado não é igual ao resultado obtido na execução do teste, o JUnit irá exibir um resultado semelhante a figura6.

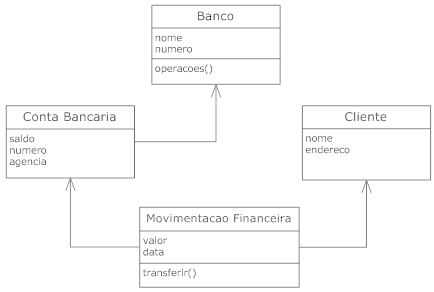
  
Figura 6 - JUnit teste falhou

Caso ocorra um erro inesperado, o código a ser testado teve sua execução interrompida devido um erro, neste caso não existe um resultado obtido, O JUnit irá exibir um resultado semelhante a figura 7.

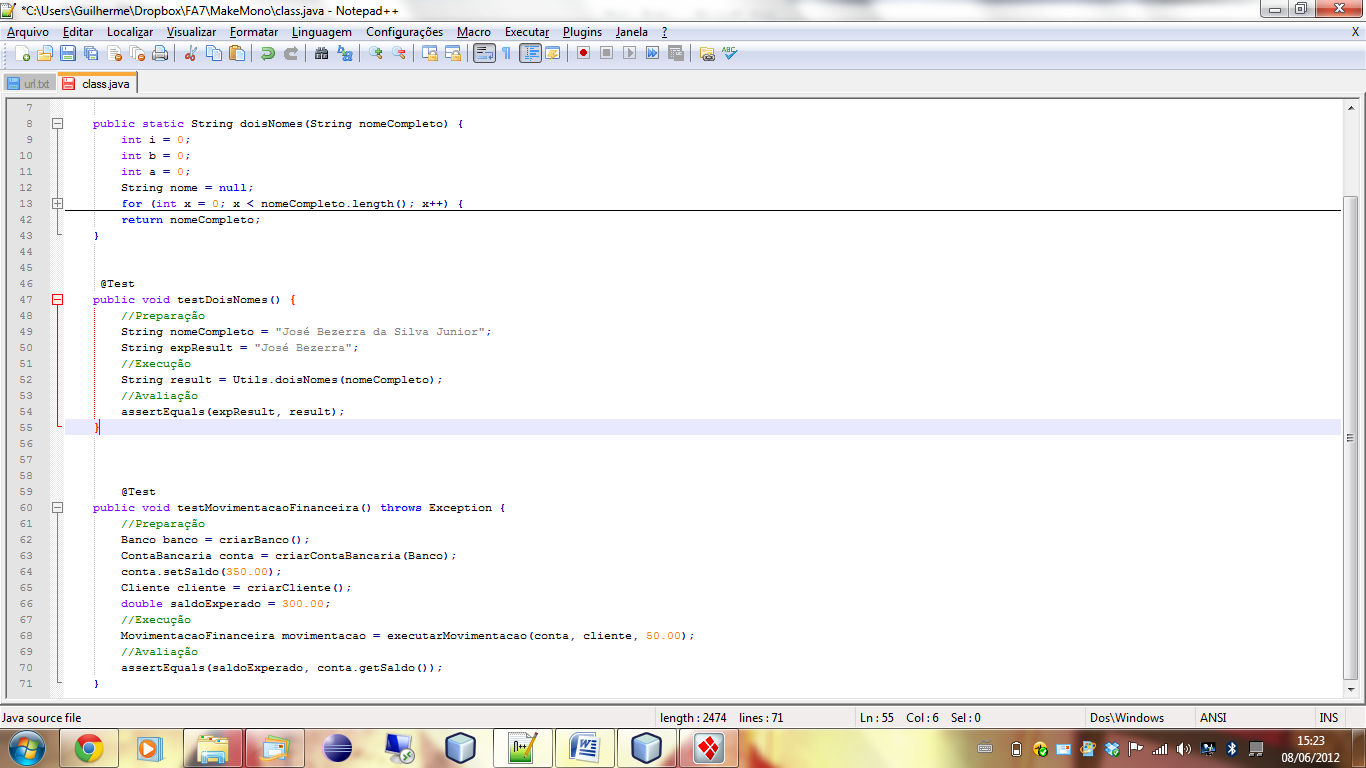
  
Figura 7 - JUnit teste deu erro

A demonstração apresentada mostra um teste que possui uma 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 bancária 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 8 - 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 9 - 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. Conclusão

Testes reduzem a incidência de erros no sistema de software, reduzindo retrabalho com correções de problemas e acertos de funcionalidades. Podemos dizer que: quanto mais testes validando funcionalidades e quanto menor for o tempo para criação dos testes; maior será a rentabilidade do sistema, menor quantidade retrabalho deverá ser realizada e maior será a satisfação do cliente.

Para criação de testes unitários o desenvolvedor deve dividir o tempo em que está produzindo com o tempo de criação do teste, a tarefa de criação do teste deve ser no menor tempo possível, tornando menor o custo do projeto.

Nos testes automatizados, o tempo de preparação do teste consome a maior parte do tempo gasto em todo o ciclo do teste unitário, insumos para a execução do teste devem ser criados respeitando e avaliando regras de validação, tornando a tarefa ainda mais complexa e extensa para o desenvolvedor.

Encontrar maneiras para reduzir o tempo de criação dos testes, é encontrar maneiras de rentabilizar o sistema como um todo, é fundamental que o desenvolvedor gaste maior parte do tempo com funcionalidades do que com desenvolvimento de testes paras as mesmas, para tanto, é necessário automatizar ao máximo a preparação dos insumos para a realização dos testes automatizados, aonde vimos que depende em sua totalidade de intervenção do desenvolvedor.

1. JSR-303
   1. Definição

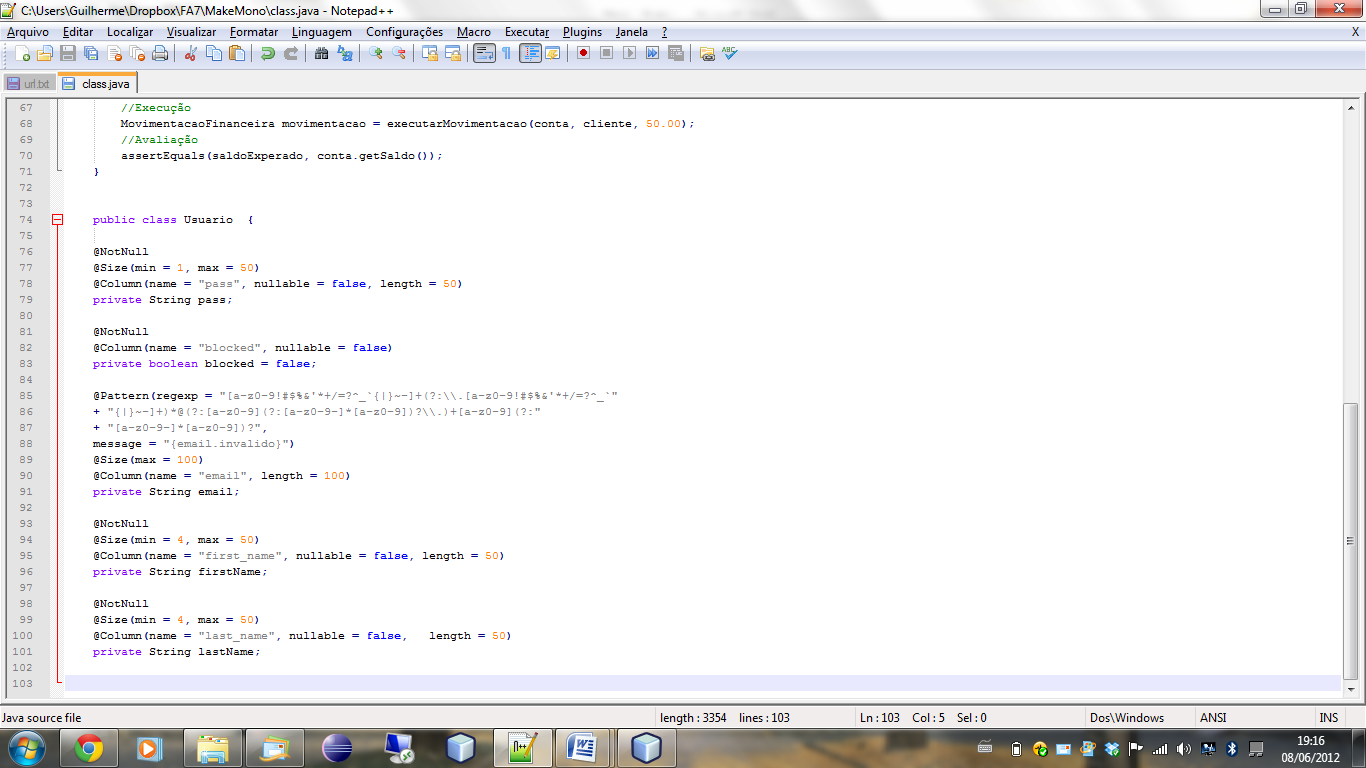
Java Bean Validation (JSR-303) é uma especificação que foi aprovada pelo JCP (Java Community Process) a partir de 16 de novembro de 2009 e é aceita como parte da especificação Java EE 6. Ela define um modelo de anotações e API para validação de atributos em uma classe.

Com as anotações nos atributos da classe, a validação se torna uma tarefa simples. Na Java Persistence 2 (JSR-317: JavaTM Persistence 2.0), API de especificação Java para persistência de dados por meio de mapeamento objeto-relacional , antes de salvar qualquer objeto na base de dados, a especificação JPA 2 define que os objetos devem ser validados pelas regras da JSR-303, 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.”

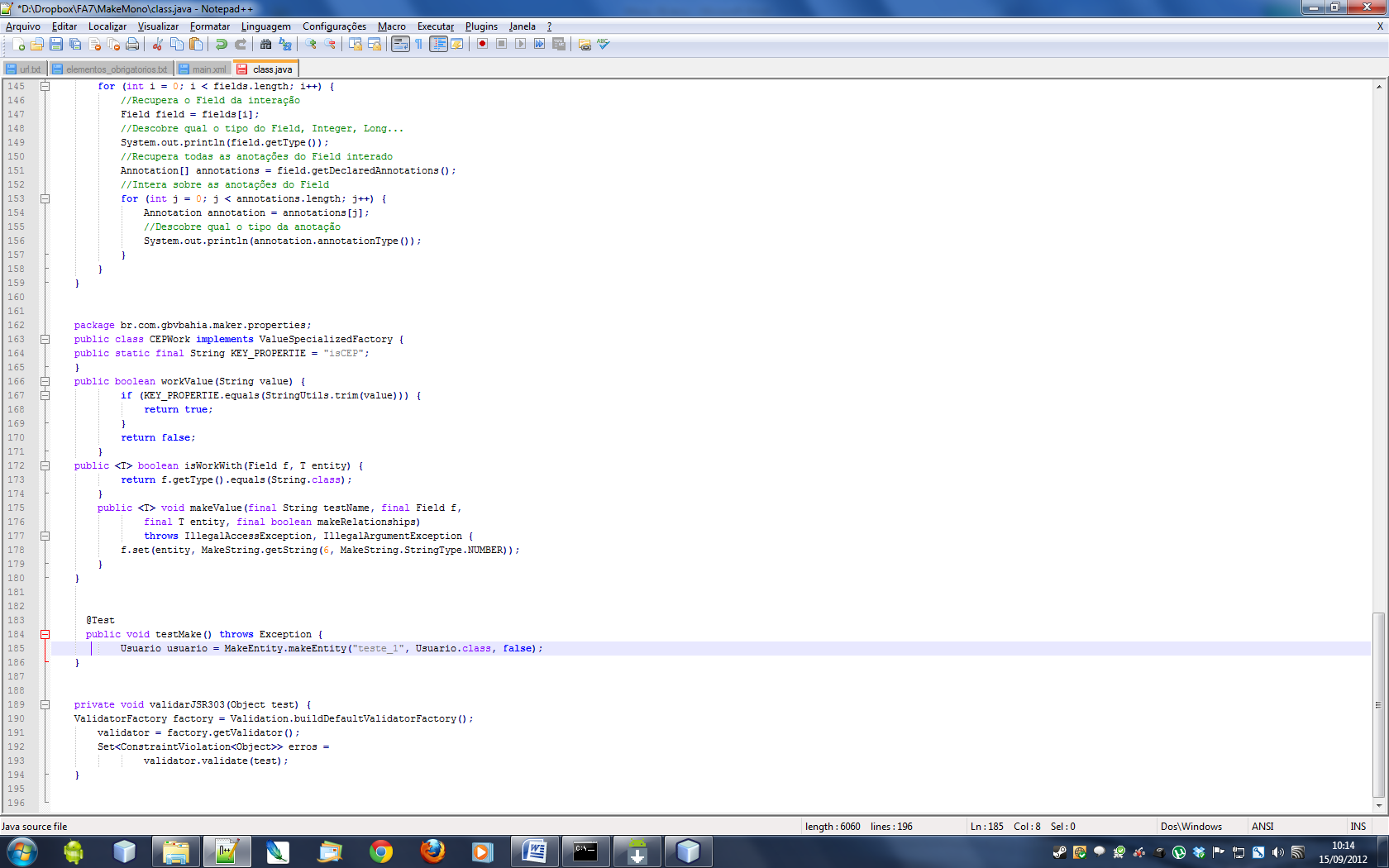
Outra especificação que utiliza a JSR-303 é o JSF2(JSR-314 JavaServerTM Faces 2.0), framework MVC para aplicações WEB. Ele realiza a validação de um objeto Java que possui as anotações da JSR-303 em formulários HTML, executando a validação dos dados informados pelo usuário na submissão do mesmo. JSR-314 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”.

Para sistemas que não utiliza as especificação JSR-317 e/ou a especificação JSR-314 ainda é possível utilizar a especificação JSR-303, mas a validação deverá ser realizada de forma programática quando necessário.

Um exemplo de classe Java que utiliza as anotações da especificação JSR-303:

  
Figura - Exemplo de classe que utiliza validação JSR-303. As anotações @NotNull, @Size e @Pattern são parte da especificação.

A validação programática é bem simples: após chamar uma referência a um objeto validador através de uma fábrica, basta executar a validação no objeto desejado. Se houver problemas, os mesmos serão retornados em uma coleção de erros. Caso contrário, a coleção estará vazia. Um exemplo de validação programática da especificação JSR-303 pode ser visto na figura 11.

  
Figura - Forma de validação programática utilizando a JSR-303

A especificação da JSR-303 define algumas anotações que podem ser utilizadas pelo desenvolvedor. Essas anotações podem ser vistas na tabela 1.

|  |  |  |
| --- | --- | --- |
| Anotação | Descrição | Exemplo |
| @AssertFalse | O valor deve ser falso. | @AssertFalse  boolean isUnsupported; |
| @AssertTrue | O valor deve ser 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 campo 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 possa ser nulo. | @NotNull  String username; |
| @Null | Obriga que um determinado valor seja nulo. | @Null  String unusedString; |
| @Past | Determina que um campo 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 1 - Anotações padrão da especificação JSR-303

Embora as anotações cubram grande parte da necessidade de validação de um sistema, ainda pode ser necessário criar novas anotações para serem utilizadas em situações específicas 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, devendo haver a validação do digito verificador. A especificação da JSR-303 também permite que o desenvolvedor crie novas anotações de validação.

* 1. Conclusão

A utilização da especificação JSR-303 inicialmente aparenta ser trabalhoso.O desenvolvedor após, definir toda a classe, atributos, métodos, construtores, forma de relacionamento com o banco de dados, terá mais um trabalho a ser realizado, definir quais faixas de valores são válidos para cada atributo utilizado e mapeado no banco de dados.

Essa aparente verbosidade tende a ser eliminada conforme o sistema tem várias regras de negócio implementadas, onde o desenvolvedor não terá que se preocupar com valores inseridos em qualquer contexto, uma vez que os dados sempre serão validados antes de serem inseridos no banco de dados.

A utilização da especificação JSR-303 traz algumas vantagens ao sistema como todo:

* Centralização da validação de dados em um único local: caso alguma validação seja feita de modo incorreto, basta alterar em um único local para que todo o sistema seja corrigido.
* Desenvolvimento de regras de negócio mais focadas: ao desenvolver uma regra de negócio o desenvolvedor não precisará se preocupar em validar informação dentro da regra a implementar, essa informação pode chegar validada ou pode ser validada com a utilização de um simples método (ver figura 11).
* Integração com frameworks relacionados: os frameworks utilizados para desenvolvimento, criados após a JSR-303, podem ser integrados com a mesma, tornando qualquer validação programática desnecessária e a tarefa de desenvolver funcionalidades mais focada em regras que validação.
* Redução de camadas a serem desenvolvidas: antes da especificação era necessário criar uma camada exclusiva para validar dados, sendo necessário alterar a mesma a cada alteração nos objetos que necessitam serem validados. Com a JSR-303 é necessário apenas alterar a anotação de validação.

1. API Reflection Java
   1. Definição

A Sun (ORACLE 2011), em seu site de documentações (http://docs.oracle.com/javase/tutorial/reflect/index.html), definiu a API Reflection 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.*”* A definição de Linden(2011, p.9) para este texto: “[...] capacidade de um programa de examinar ou modificar o comportamento de aplicações rodando na Java Virtual Machine em tempo de execução.” Essa API permite ao desenvolvedor criar softwares que olham para si mesmos, e executem tarefas dependendo da necessidade e/ou dos valores encontrados.

Esta API permitiu a criação ou melhoria de vários frameworks utilizados para simplificar atividades no desenvolvimento de sistemas, como a Bean Validation (JSR-303), JSF 2 (JSR-314), EJB 3.0 (JSR-220), Spring 2.0, entre outros. Esses framewokrs com base em reflexão analisam os valores e tomam decisões em tempo de execução das aplicações. Por exemplo, ao receber um objeto para salvar em uma tabela de banco de dados, a API JPA verifica o nome da tabela ao qual o objeto a ser salvo pertence olhando para o código que está dentro do objeto. Após ver o nome da tabela, o framework continua a procurar pelos nomes das colunas e demais informações necessárias.

* 1. Utilização

Através da API Reflection Java podemos “enxergar” quase tudo que é necessário saber sobre uma classe Java. Para trabalhar com uma classe de qualquer objeto basta utilizar o método getClass(), existente em todos os objetos Java. Este método irá retornar um objeto que torna possível trabalhar com dados de uma classe como um objeto qualquer, esse objeto tem o nome de Class.

Após a recuperação do objeto Class é possível trabalhar com todas as variáveis declaradas na classe, isso é possível com o auxílio do objeto Field, que representa uma variável declarada em uma classe. Para ter acesso aos Fields de uma classe, utilizamos o método getDeclaredFields() no objeto Class recuperado. Este método irá retornar todos os Fields declarados na classe dentro de um Array.

Ao realizar a iteração em cada Field declarado na classe, é possível recuperar as anotações de cada um utilizando o método getDeclaredAnnotations() da classe Field. Como cada atributo declarado em uma classe pode receber infinitas anotações, este método irá retornar um Array de objetos Annotation. Caso seja necessária a identificação de alguma anotação, deve-se realizar a iteração sobre o Array de Annotation. Toda codificação desse exemplo pode ser visto na figura 12.

Tendo como referência a classe Usuário demonstrada na figura 10, para utilizar o método reflectionExemple da figura 12, bastava realizar a seguinte chamada:   
reflectionExemple(Usuario.class);

  
Figura - Exemplo de utilização da Reflexão Java (o tratamento de exceções foi omitido para melhor visualização).

Ao iniciar a execução do método reflectionExemple teríamos:

* Um novo objeto usuário seria criado através de newInstance().
* Todos os Fields de Usuário, pass, blocked, email, etc., seriam recuperados em  
   Field[] fields = clazz.getDeclaredFields();.
* Seria descoberto o tipo do Field, Integer, Long, Calendar, etc., através da chamada getType() de Field. Mesmo se o tipo for uma classe criada pelo desenvolvedor, o seu tipo seria retornado.
* Ao executar field.getDeclaredAnnotations(); todas as anotações do Field da iteração seriam retornadas, por exemplo: se a iteração do Field estiver sobre a variável pass as anotações retornadas na execução de field.getDeclaredAnnotations(); seriam: @NotNull, @Size e @Column.
* É possível saber o tipo da anotação através do método de Annotation annotation.annotationType(); onde é possível saber o tipo da anotação e realizar a leitura dos atributos da mesma.
  1. Conclusão

A API reflection permitiu a criação de vários frameworks que trabalharam em várias camadas distintas da aplicação, acelerando o desenvolvimento de todas as camadas:

* Visão: JSF2, Spring MVC, V-Raptor, etc.
* Negócio: Spring Beans e EJB3
* Persistência: Hibernate, EclipseLink, JPA, etc.
* Segurança: Spring Security.
* Validação: JSR-303
* Testes: JUnit, Make

A API de reflexão possibilita trabalhar com diferentes tipos de objetos da mesmo forma, mantendo um baixo acoplamento entre diferentes tipos de objetos podendo trabalhar com os mesmos em diferentes situações.

Ao utilizar a API Reflection o desenvolvedor programa para a Classe e não para o tipo do objeto, ou seja, alterações no objeto, possivelmente, não irá alterar o comportamento desenvolvido para trabalhar com reflexão.

1. O Framework Make
   1. Antes do make

Estávamos desenvolvendo um projeto totalmente novo, não havia massa de dados para geração de testes nem de desenvolvimento, o desenvolvedor criava uma tabela e não existia informação para ser inserida na mesma. Nosso projeto levou a importância das telas de cadastro para última implementação, o cliente foi enfático ao afirmar que era aceitável que os dados fossem manipulados diretamente no banco de dados, mas certas funcionalidades da regra de negócio deveriam ser entregues, estávamos desenvolvendo funcionalidades críticas ao projeto, sem ter telas para cadastrar os dados no banco de dados para testar o que desenvolvíamos. Nosso prazo era extremamente curto e desenvolver algo que não ia agregar valor ao projeto estava fora de questão.

Nesta situação após definir a arquitetura do projeto iniciamos o desenvolvimento. Sempre que um objeto era definido, utilizávamos o framework de persistência para gerar ou atualizar o banco de dados e tínhamos que inserir os dados de testes de forma manual. Apesar dos scripts SQL de geração destes dados serem armazenados, quando os objetos eram alterados devido a uma nova funcionalidade ou mesmo erro na definição do objeto, era necessário um esforço extra para o trabalho de atualização dos scripts.

A inserção de informação no banco de dados estava cada vez mais complexa e atualização dos scripts cada vez mais dispendiosa. Em alguns momentos, perdíamos a metade do dia de trabalho somente para inserir valores no banco de dados.

Para piorar mais a situação, descobrimos um outro problema: como os dados inseridos no banco de dados sempre eram o mesmos, alguns testes unitários realizados por alguns desenvolvedores inexperientes estavam “viciados”, ou seja, se o valor fosse modificado o teste iria falhar ou apresentar um erro.

Procuramos alguns frameworks que gerassem valores aleatórios para serem inseridos no banco de dados, mas os que encontramos apresentavam o mesmo problema de dados estáticos e também uma complexidade de utilização considerável.

Dentro desta realidade resolvemos criar um framework que resolvesse todos esses problemas, ou que pelo menos, auxiliasse esse trabalho tão importante para o desenvolvimento de sistemas.

* 1. Propósito

O Make é um framework desenvolvido para gerar valores fictícios para realização de testes unitários ou população de banco de dados de desenvolvimento. Estes valores são limitados à validação definida nas anotações da especificação JSR-303, utilizada pelo desenvolvedor.

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 pode ter uma validação mínima aceitável de R$100,00, mas não uma máxima. O Make pode ser configurado 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.

É possível utilizar o framework em um sistema que não possui as anotações da JSR-303, para gerar os valores o desenvolvedor deverá informar os limites ou uma faixa de valores para os campos a serem gerados. Para isso será necessário utilizar uma fábrica especializada em cada atributo que se deseja popular.

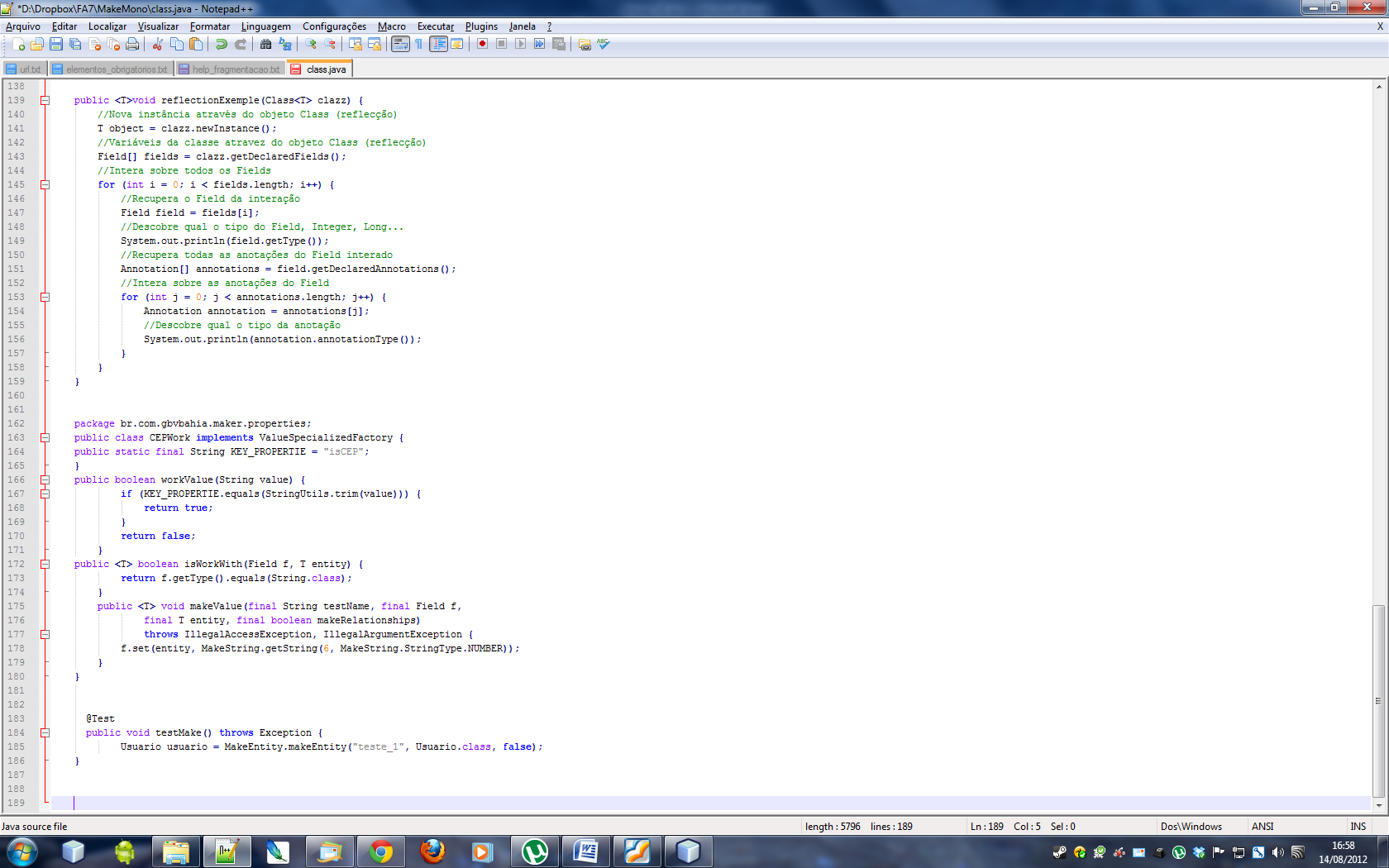
* 1. Utilização

O Make pode ser utilizado juntamente com o código de testes da aplicação, criando objetos para serem utilizados nos testes unitários atendendo as validações impostas pela JSR-303 ou respeitando os limites impostos pelo desenvolvedor. Também pode ser utilizado para gerar dados válidos, porém irreais, para popular uma base de dados de desenvolvimento. Nesta segunda utilização, recomendamos que seja feito um projeto em paralelo ao projeto aplicado, não utilizando o código de testes para isso, para não poluir o código de testes com outra tarefa totalmente distinta.

* + 1. Métodos de chamada

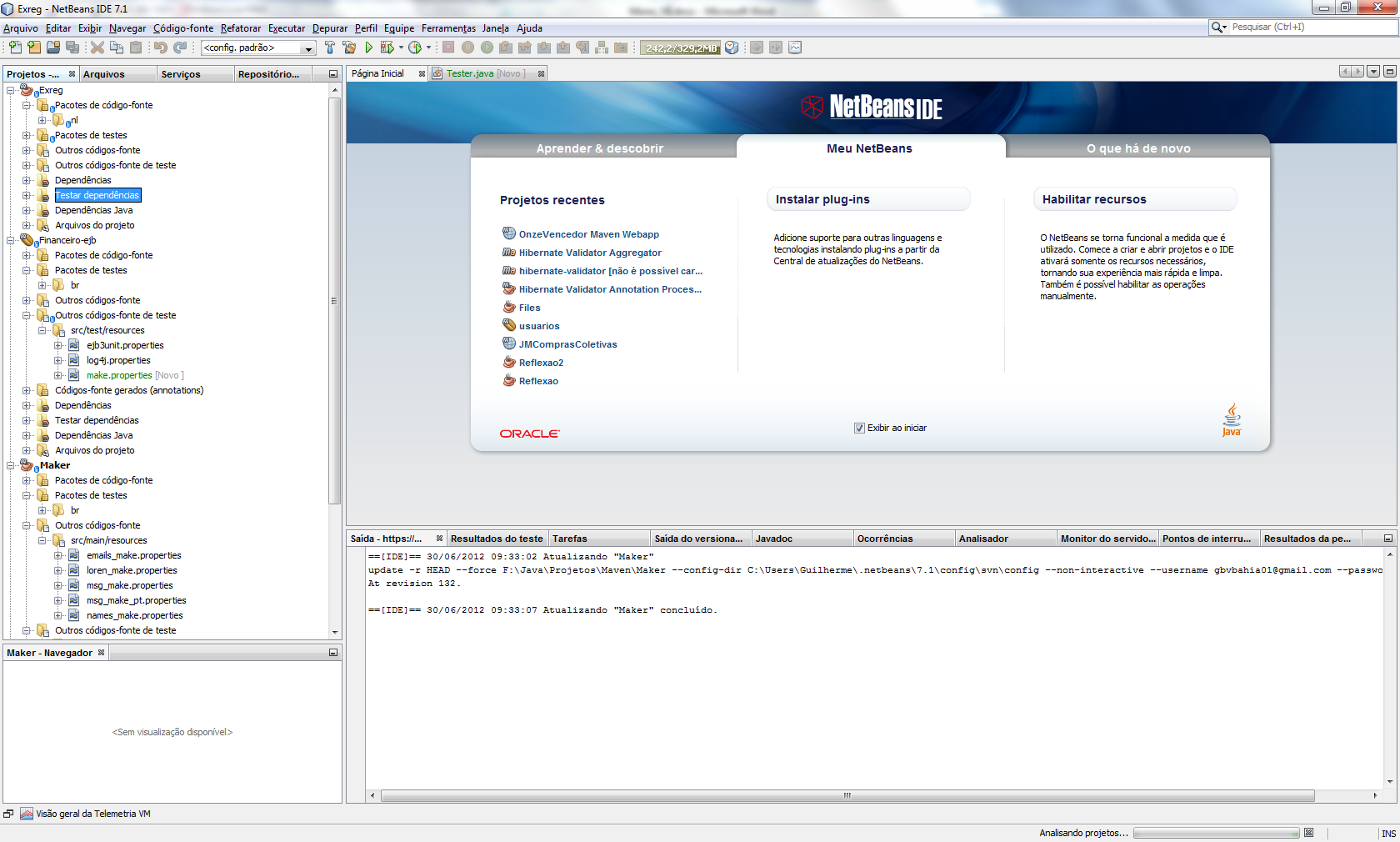
O Make possui uma classe principal para a utilização, essa classe é chamada de MakeEntity, para criar uma entidade o desenvolvedor deverá chamar um dos métodos sobrecarregados MakeEntity.makeEntity:

* T makeEntity(Class<T> entity): Recebe a classe a ser criada, irá ignorar relacionamentos com objetos criados pelo desenvolvedor e não irá procurar o arquivo auxiliar make.properties.
* T makeEntity(Class<T> entity,boolean makeRelationships): O segundo parâmetro, makeRelationships, possibilita ao desenvolvedor solicitar ao Make que crie objetos relacionados anotados com @Entity da JPA. Não irá procurar o arquivo auxiliar make.properties.
* T makeEntity(String testName,Class<T> entity): Possibilita informar o nome do teste para carregar informações do arquivo make.properties. Irá ignorar relacionamentos com objetos criados pelo desenvolvedor.
* T makeEntity(String testName, Class<T> entity,boolean makeRelationships): Possibilita informar o nome do teste e solicitar ao Make que crie objetos relacionados anotados com @Entity da JPA.
* List<T> makeEntities(String testName, Class<T> entity, int amount, boolean makeRelationships): O parâmetro amount possibilita criar uma lista com vários objetos gerados pelo Make, com ou sem relacionamentos e com ou sem definições no arquivo make.properties.

  
Figura – Exemplo de utilização de criação de um objeto com dados aleatórios pelo Make.

* + 1. Utilizando as fábricas especializadas

Alguns dos métodos sobrecarregados da classe MakeEntity recebem o nome do teste, “testName”. Esse parâmetro carrega informações necessárias ao teste referente às classes especializadas, essas informações ficam no arquivo make.properties, na forma chave = valor. Ele deve ficar na raiz do pacote de testes, conforme figura 14.

  
Figura - make.properties em um projeto Maven

Dentro deste arquivo especificam-se as fábricas especializadas que devem ser utilizadas para os Fields que necessitem delas. 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 do teste é uma forma de identificação para o mesmo, permitindo que possam ser definidas classes especializadas diferentes para o mesmo atributo em testes diferentes. Qualquer nome pode ser dado, desde que não inicie com a palavra “work”, pois este nome é reservado para as fábricas especializadas criadas pelo desenvolvedor.

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) e em um teste o desenvolvedor necessita que esse valor seja criado com um valor de CPF válido. O desenvolvedor pode criar o valor deste atributo utilizando a fábrica especializada do Make, chamada MakeCPF, configurando o 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 fábrica especializada na criação do objeto 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.

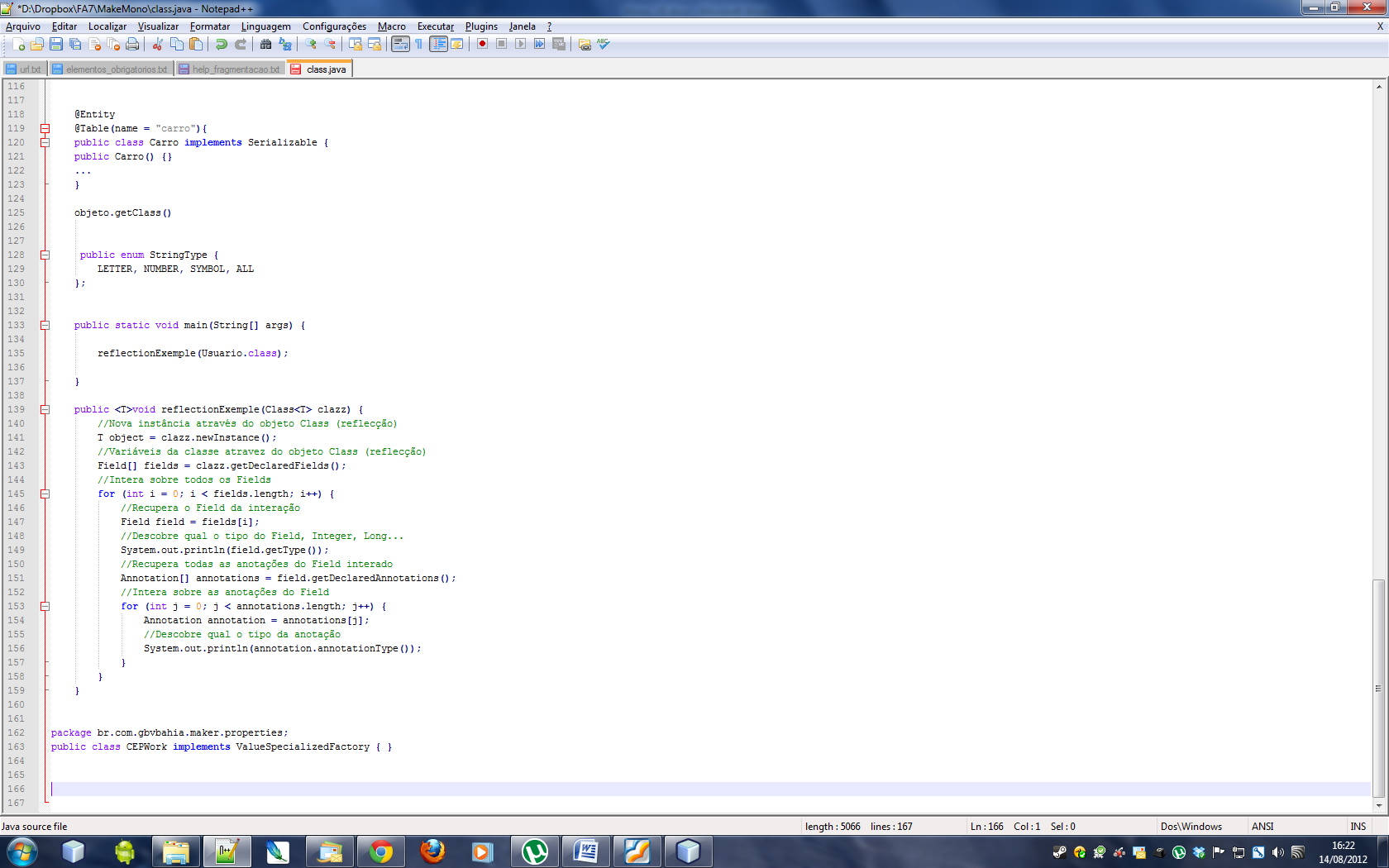
* + 1. Criando fábricas especializadas

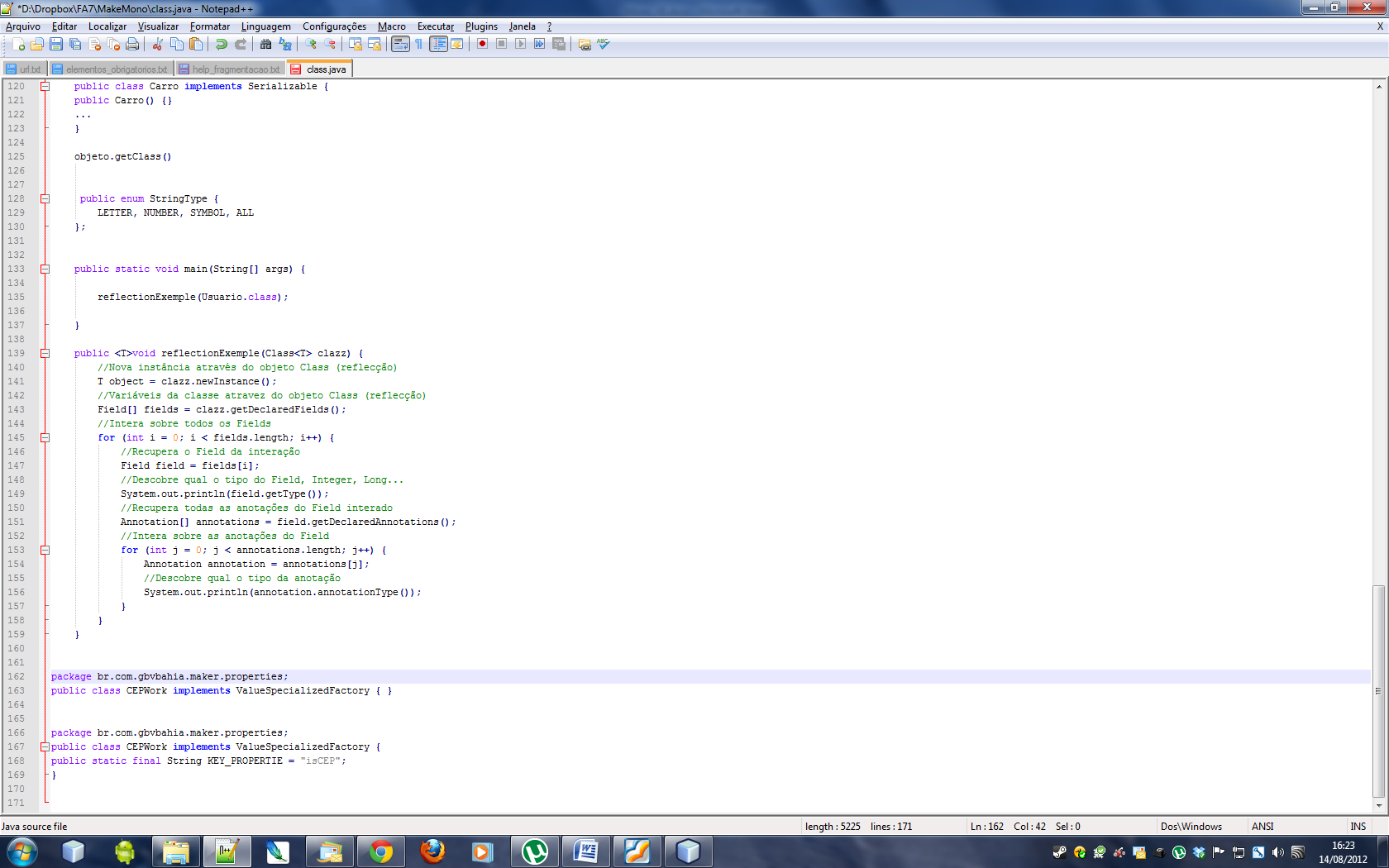
O Make possibilita que o desenvolvedor crie suas Fábricas Especializadas, isso é importante uma vez que a necessidade de valores personalizados não pode ser prevista para todas as situações.

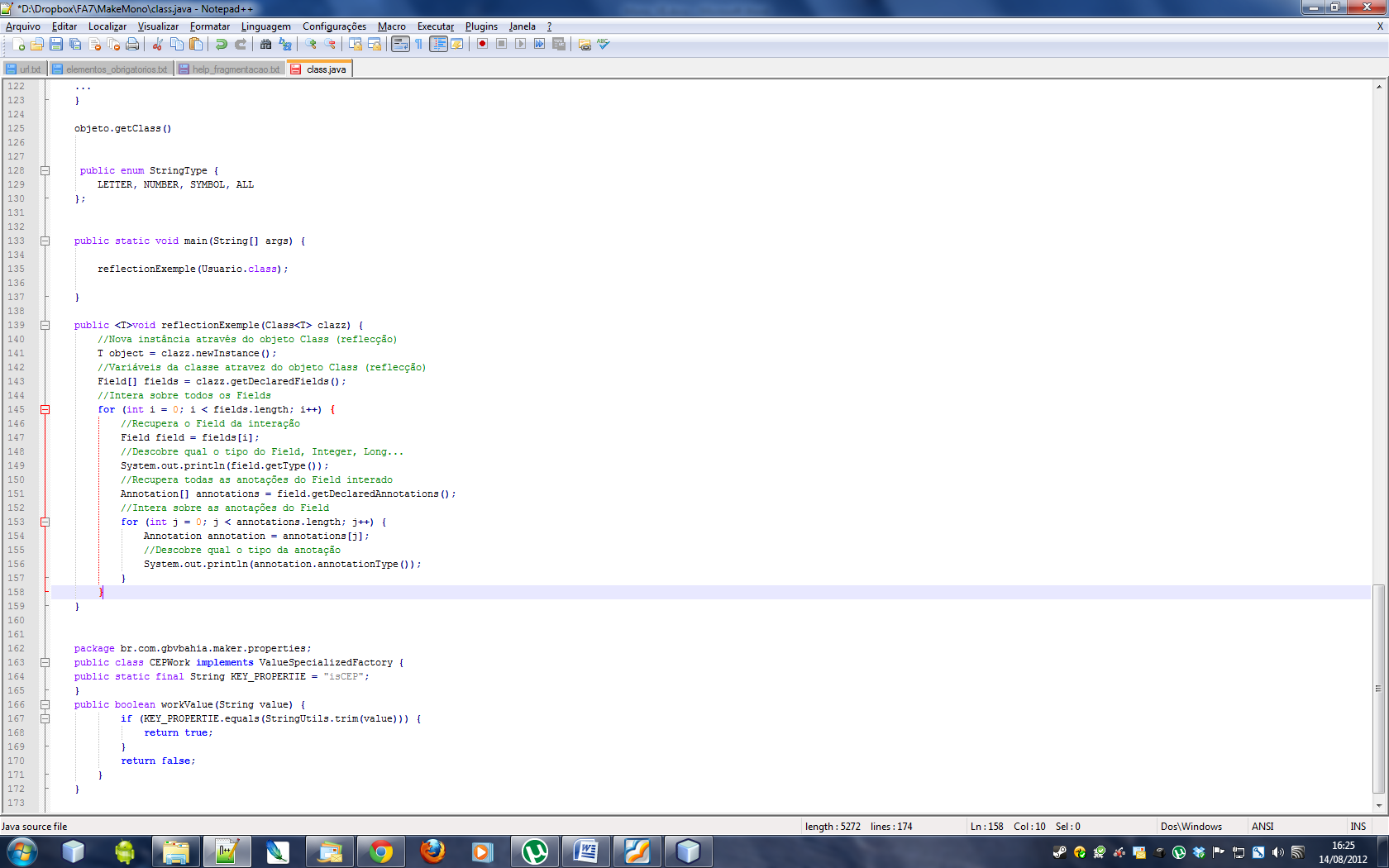
Para executar essa tarefa dois passos devem ser seguidos:

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

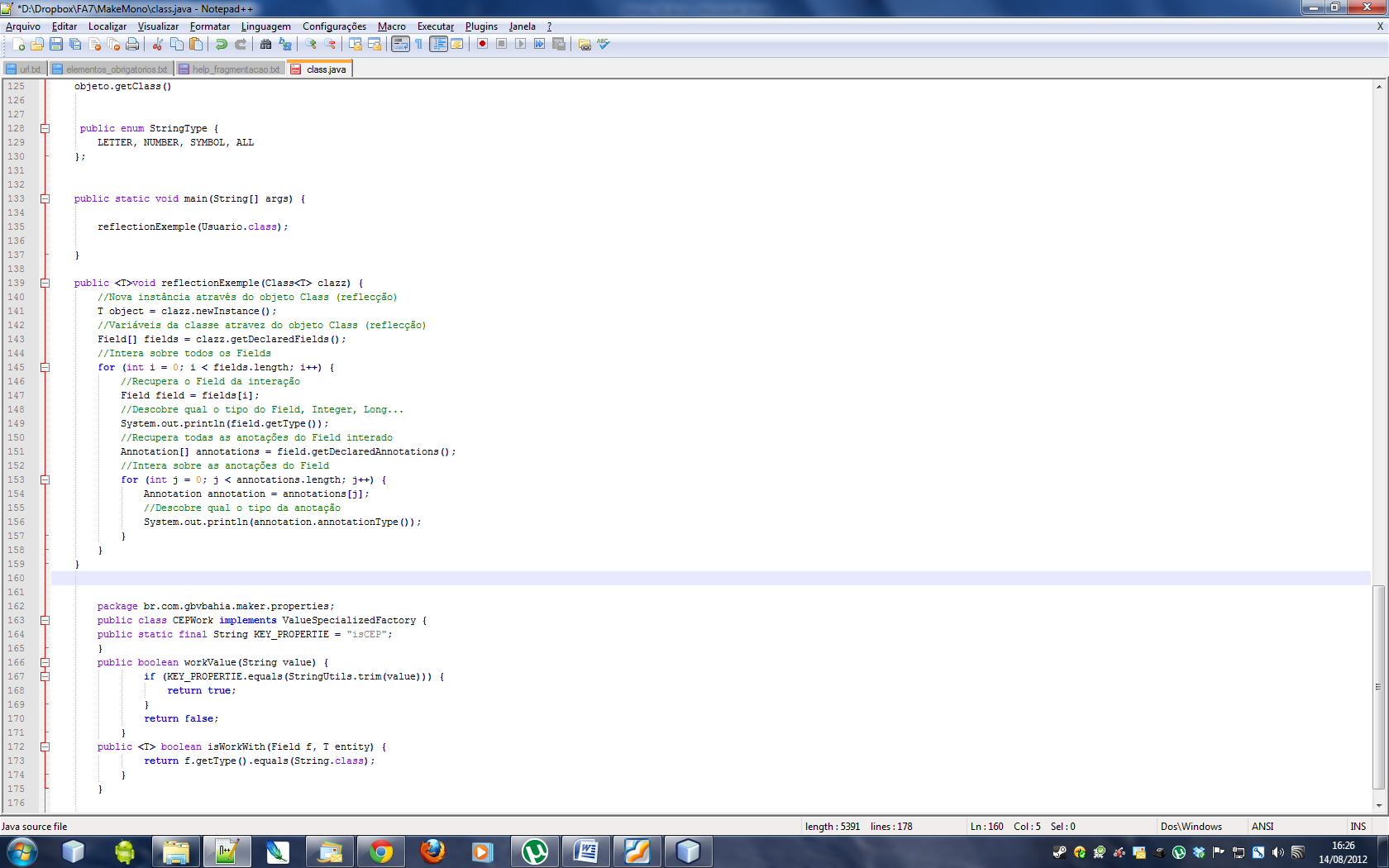
Para exemplificar vamos demonstrar como criar uma fábrica especializada que gere CEP´s, de ruas de forma aleatória.

Criar a classe CEPWork que implementa ValueSpecializedFactory:

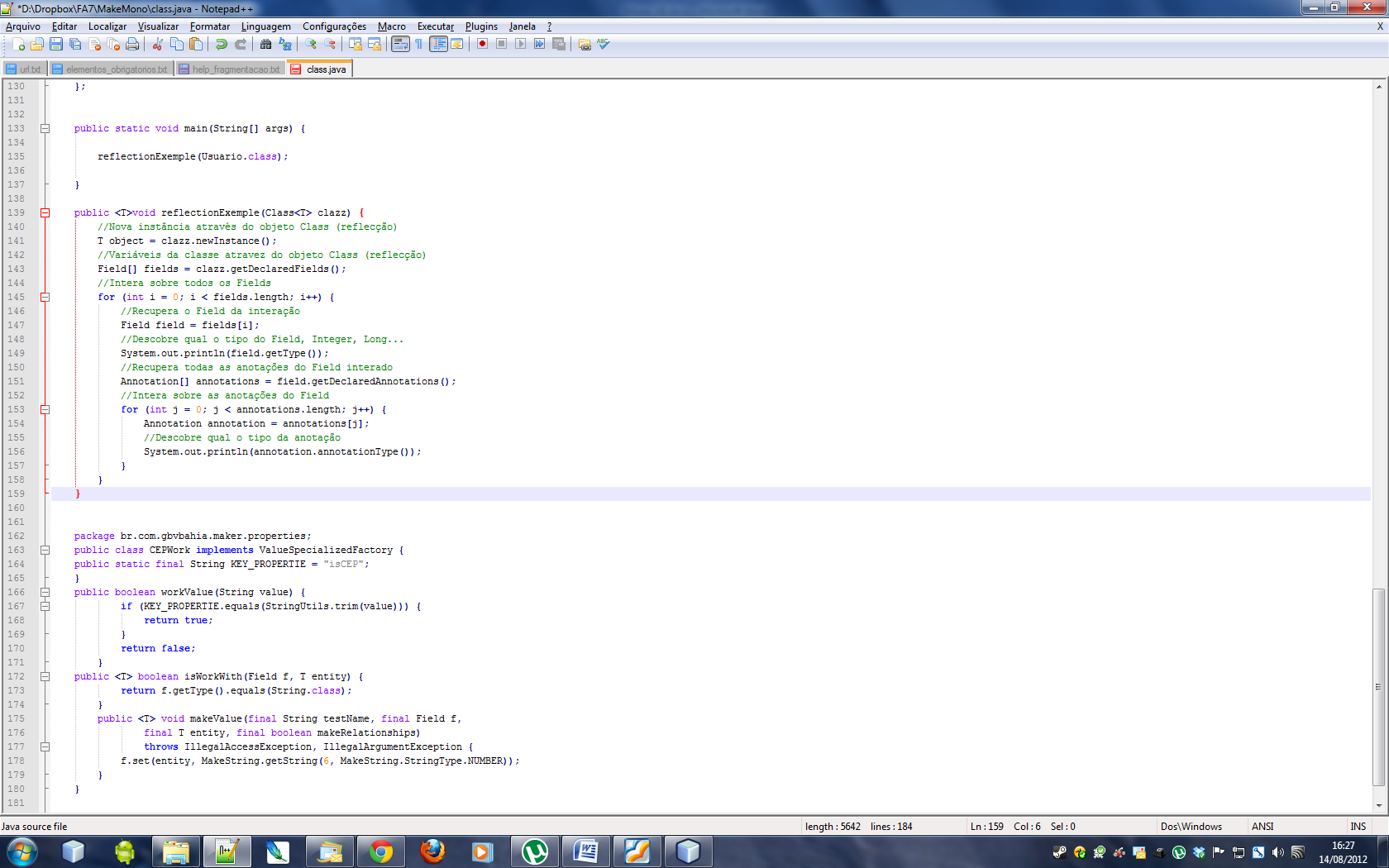
Definir que a assinatura de identificação para utilização de CEPWork será isCEP:  
Deste modo, no arquivo make.properties, o Field que for definido com isCEP:  
 teste1.br.com.Endereco.cep = isCEP,  
 terá a classe CEPWork como fábrica especializada.

Implementar o método de ValueSpecializedFactory, boolean workValue(String value):  
O método workValue irá receber o valor declarado no arquivo make.properties, sendo isCEP o método retorna verdadeiro, caso contrário irá retornar falso.

Implementar 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:



Por último, o segundo método de ValueFactory é implementado, makeValue:

  
Figura - Implementação de uma Fábrica Especializada do Make

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 ter o valor gerado, 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 métodos de MakeEntity para gerar novos objetos, se for necessário.

Para que o Make possa reconhecer a fábrica especializada criada é necessário 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 à 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 (de execução).

* + 1. Reunindo tudo

Em um projeto que estava participando, a evolução constante das classes que representavam as tabelas do banco de dados estava tornando a população de dados um empecilho para evolução continua do sistema. As classes eram alteradas e toda a informação criada para popular o banco de dados deveria ser reavaliada e refeita, gastando um tempo significativo para a geração de informação e muitas vezes parte da equipe ficava parada esperando ou auxiliando a geração da informação.

Para minimizar o problema utilizamos o Make para gerar valores para popular o banco, quando alterações estruturais eram realizadas, apagávamos todo o banco e gerávamos tudo novamente com auxilio do Make, com base nessa situação iremos apresentar uma pequena parte da solução adotada de população do banco de dados.

O exemplo será o da classe de Ponto de Atendimento, essa classe representa um ponto de atendimento a clientes de um determinado setor ou empresa, seus dados foram encurtados por uma questão de didática, mas acreditamos ser suficiente para total entendimento o conteúdo demonstrado.

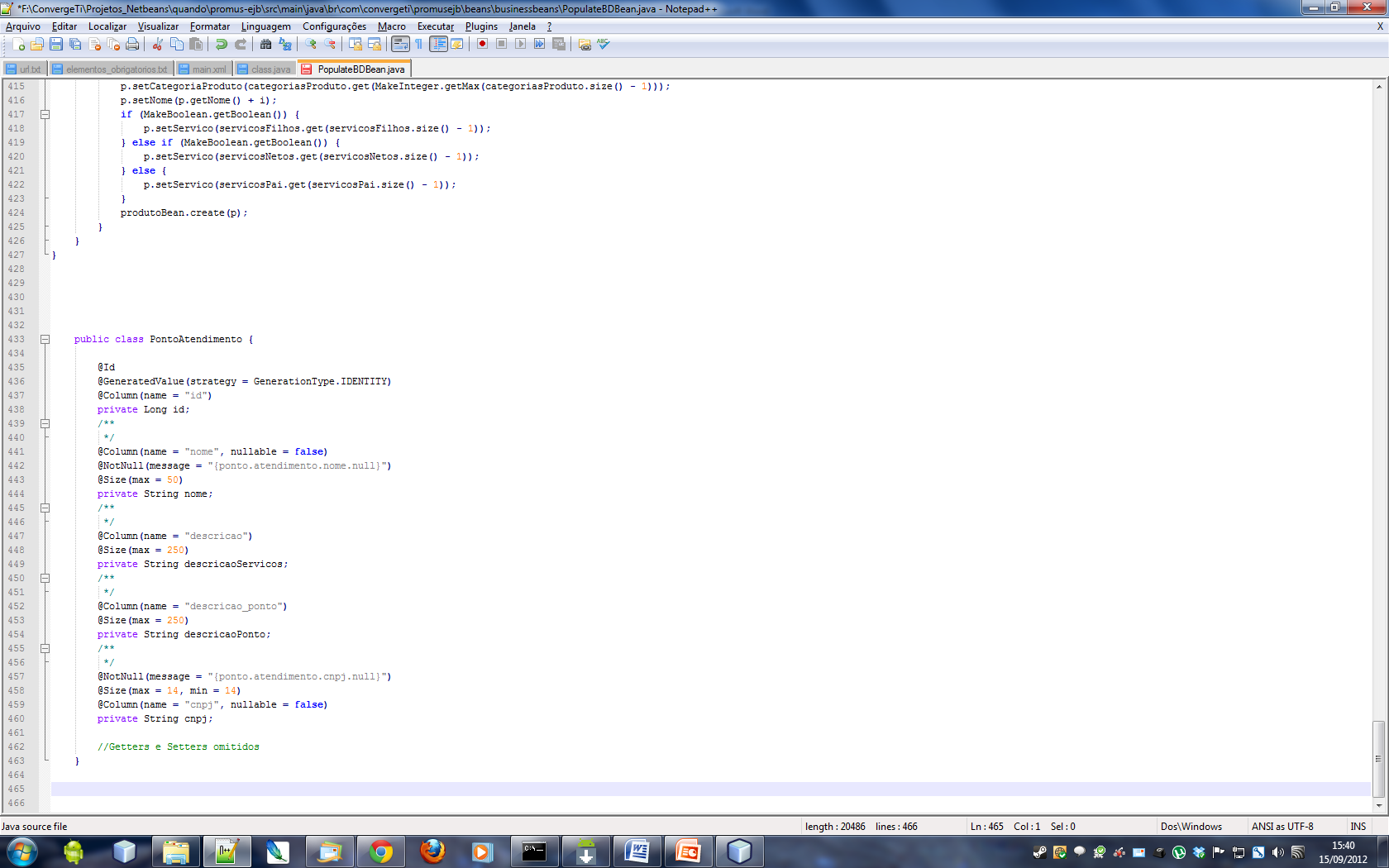


Figura - Classe Ponto de Atendimento: exemplo de criação de valores com o Make

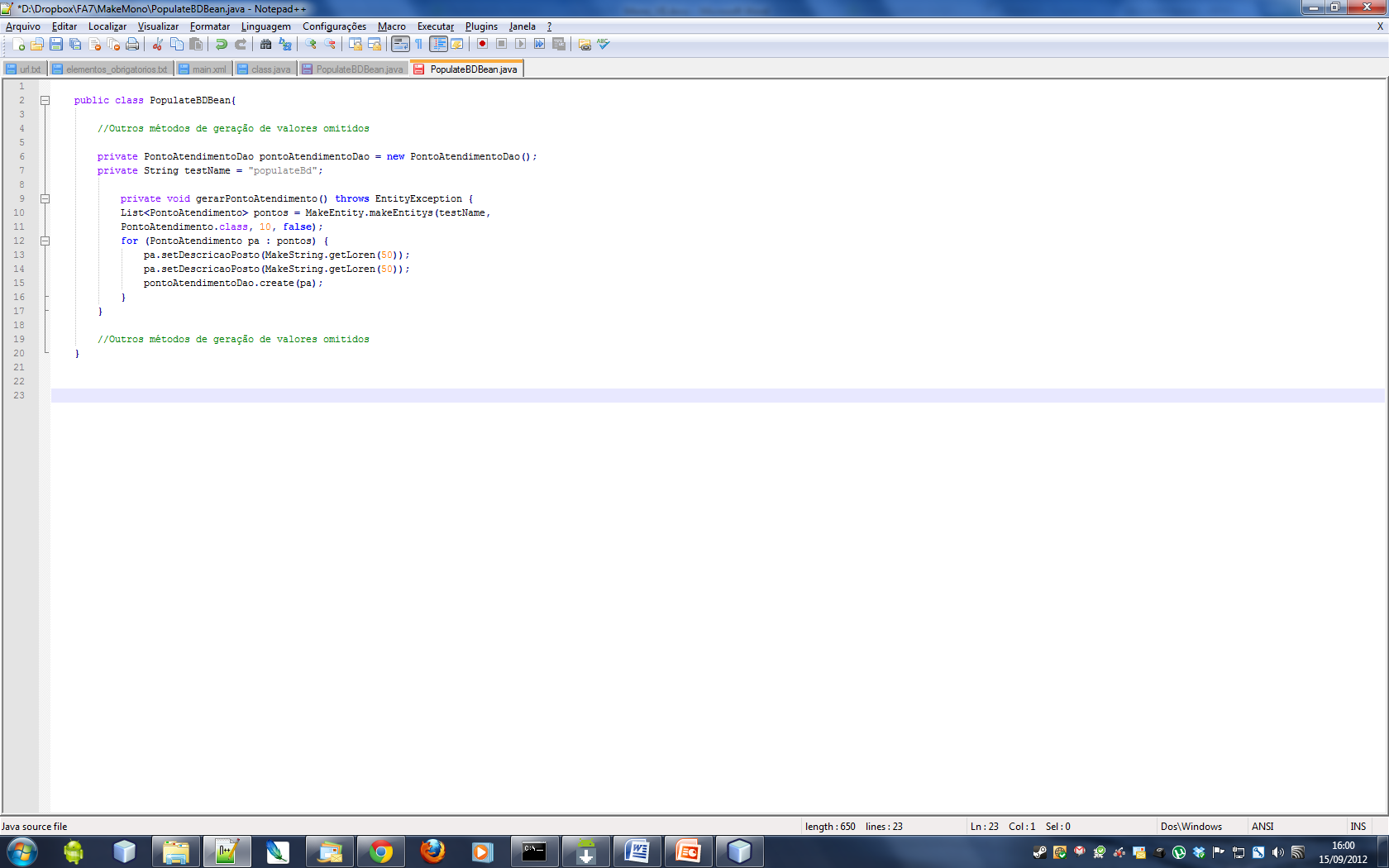
Essa classe apresenta quatro atributos, nome, cnpj, descrição do ponto e descrição dos serviços prestados. É importante observar que os dois campos de descrição não são definidos com @NotNull e por isso por padrão o Make não gera valores para os mesmos.

O campo nome, o Make irá gerar valores retirados do texto Lorem Ipsum, no projeto real não foi mantido o real, porque em alguns pontos do sistema esse nome deveria ser de fácil reconhecimento e por isso tivemos que alterar criando uma fábrica especializada para o mesmo.

O campo CNPJ utilizamos a fábrica especializada MakeCNPJ que já vem no Make.

Criamos uma projeto exclusivo para criação de valores para popular o banco de dados, esse projeto utilizava a camada DAO do projeto original para gravar os valores e utilizada o Make para gerar os valores.

Para a geração dos valores no novo projeto somente uma classes foi criada, essa classe possui métodos para geração de objetos a serem inseridos no banco de dados, não exibimos o código completo dessa classe por uma questão de foco, somente o que for referente a geração de valores para a classe Ponto Atendimento foi exibido.

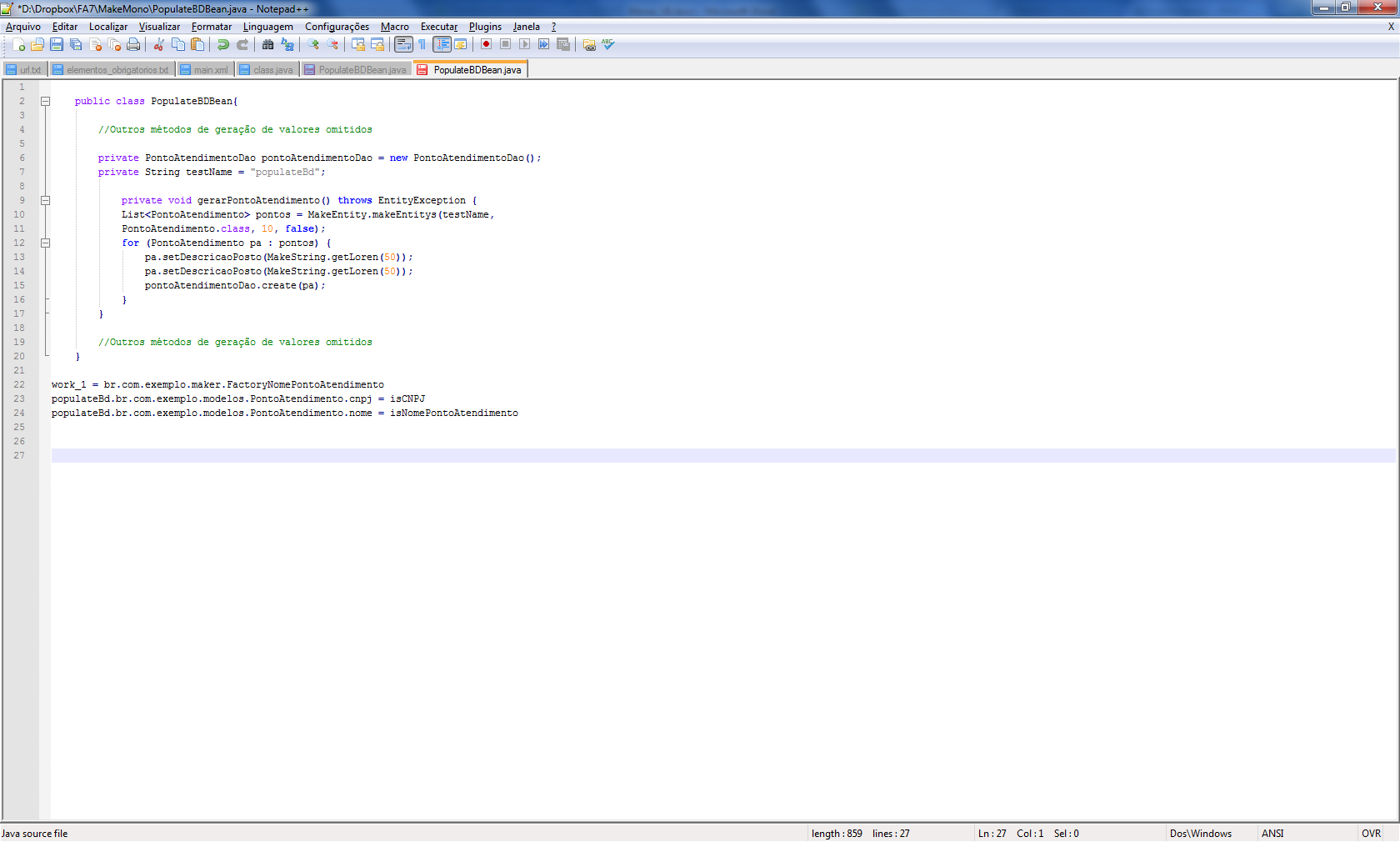
  
Figura - Criação dos Pontos de Atendimento

Os atributos do método MakeEntity.makeEntitys:

* testName: o nome do teste é “populateBd”, esse nome será utilizado para varrer o arquivo make.properties.
* PontoAtendimento.class: o tipo de classe a ser gerado é PontoAtendimento.
* 10: serão gerados dez classes do tipo PontoAtendimento.
* false: não se faz necessário criar entidades que se relacionam com PontoAtendimento.

Após os valores serem gerados pelo Make, as descrições são geradas pelos métodos estáticos MakeString.getLoren(50), onde 50 é a quantidade de caracteres que deverá ter cada descrição, conforme falado anteriormente, por não terem a anotação @NotNull o Make não gera os valores, uma segunda forma de gerar os mesmos seria por uma fábrica especializada, que neste caso não se faz necessário.

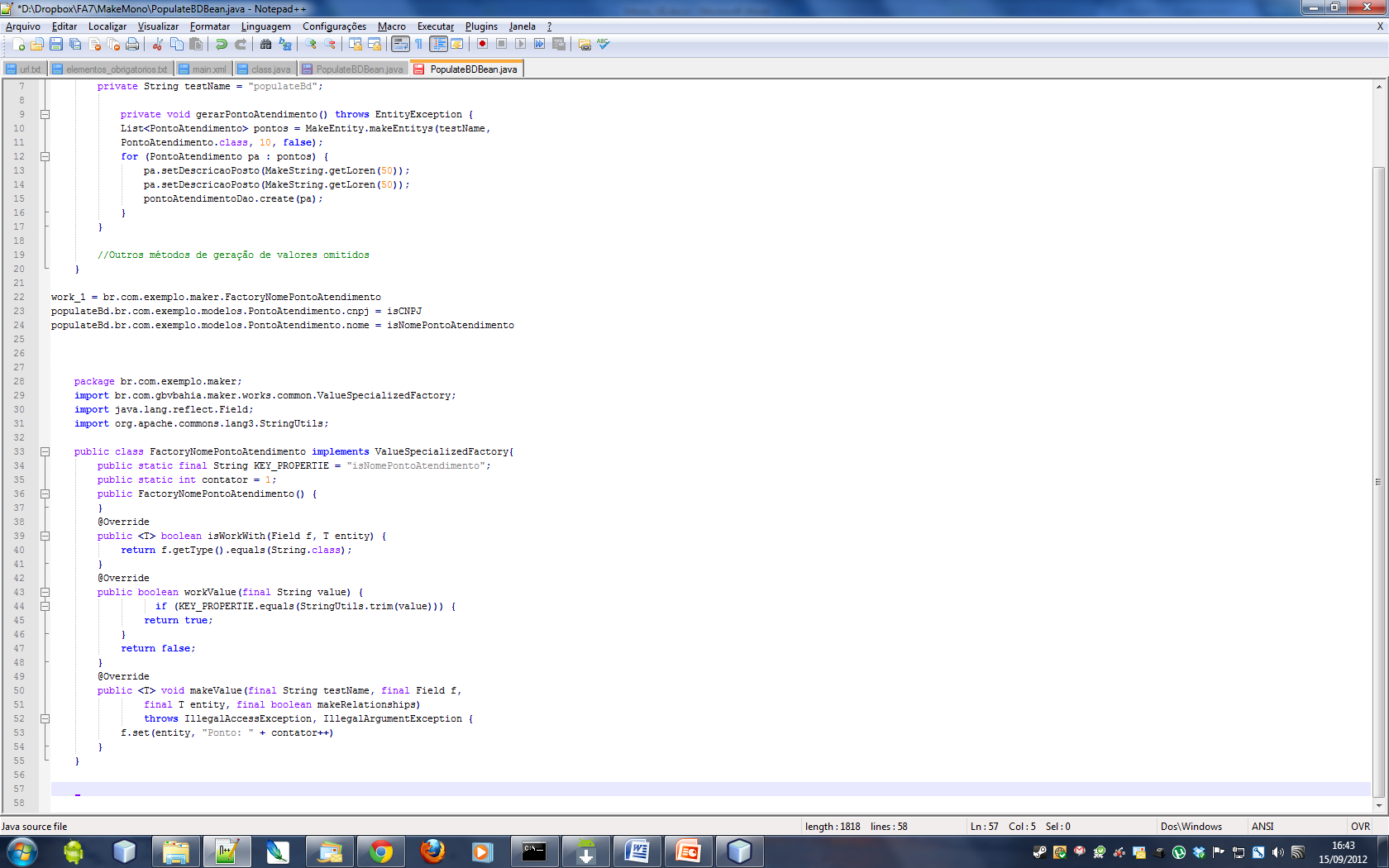
O arquivo make.properties para este teste recebeu as seguintes declarações:

  
Figura - Declarações no arquivo make.properties

A primeira declaração é referente à fábrica especializada responsável por criar os nomes dos pontos de atendimentos, o código desta classe será exibido e explicado a seguir.

A segunda linha informa que o parâmetro cnpj de PontoAtendimento deverá ser fabricado para fábrica especializada CNPJ, através do valor “isCNPJ”, essa fábrica é nativa do Make.

A terceira linha informa que uma fábrica especializada que recebe o valor “isNomePontoAtendimento” deverá gerar o valor esperado, essa é fábrica declarada na primeira linha de make.properties, seu código é bem simples e pode ser visto a seguir:

  
Figura - Fábrica Especializada para geração de nomes de PontoAtendimento.

Primeiramente é definido a chave de propriedade (KEY\_PROPERTIE) com o valor passado no arquivo make.propertie, um contador foi definido, o valor do nome do ponto de atendimento deverá ser Ponto: n, onde “n” é o número do ponto de atendimento definido no contador, tornando fácil a identificação do ponto de atendimento nos testes ao longo do desenvolvimento.

A implementação do método isWorkWith avalia o tipo, essa fábrica trabalha apenas com tipo String e a implementação de workValue avalia se a KEY\_PROPERTIE passada é igual a definida na fábrica.

Caso os dois métodos, isWorkWith e workValue retornem verdadeiro, o Make irá chamar makeValue para que seja gerado o valor necessário.

Essa foi a primeira implementação da população de dados de PontoAtendimento, mais adiante no projeto, verificamos que a quantidade de atributos desta classe estava demasiado e que seria necessário criar uma classes em separado para facilitar a manutenção da mesma, quando isso ocorreu a alteração no projeto do banco de dados e a população do mesmo foi fácil, uma vez que somente alteramos as declarações no arquivo make.properties nenhuma outra alteração foi necessária no código do projeto de geração de valores.

* 1. Arquitetura do framework

O framework Make necessita gerar valores que o seu tipo somente será conhecido somente em tempo de execução, se é um texto, um caractere, um número inteiro ou fracionado, dentre outras possibilidades, ele também deverá descobrir quais as limitações destes valores para passar na validação da JSR-303.

Para descobrir os tipos e as limitações o framework teve que utilizar a API de reflexão Java, lendo os atributos das classes, suas anotações e inserindo os valores gerados. Essa API, conforme descrito, permite acesso a todos os atributos, métodos e construtores de uma classe Java, tornando o mapeamento das limitações e do tipo de valor a ser gerado uma tarefa transparente ao desenvolvedor..

A segunda tarefa, a de gerar os valores para os campos, foi feita de modo separado, para que um alto acoplamento não venha impossibilitar futuras funcionalidades ou tornar a descoberta erros de desenvolvimento no framework uma tarefa complexa.

Para garantir o baixo acoplamento foi utilizado o padrão fábrica (Factoring Methods), uma vez que, segundo Freeman, Eric e Freeman, Elisabeth(2007, p. 105) “identifique os aspectos que variam e os separare do que continua igual”. Esse padrão, no framework Make, irá gerenciar as variações nos tipos de dados, simplificando a geração do tipo valor esperado.

Para garantir uma facilidade de implementação do framework, tanto o desenvolvimento do mesmo quanto o desenvolvimento das fábricas especializadas pelos desenvolvedores, o acoplamento da fábrica com as classes que geram valores foi realizado por contrato, ou seja, que assinar ao contrato irá poder gerar valores.

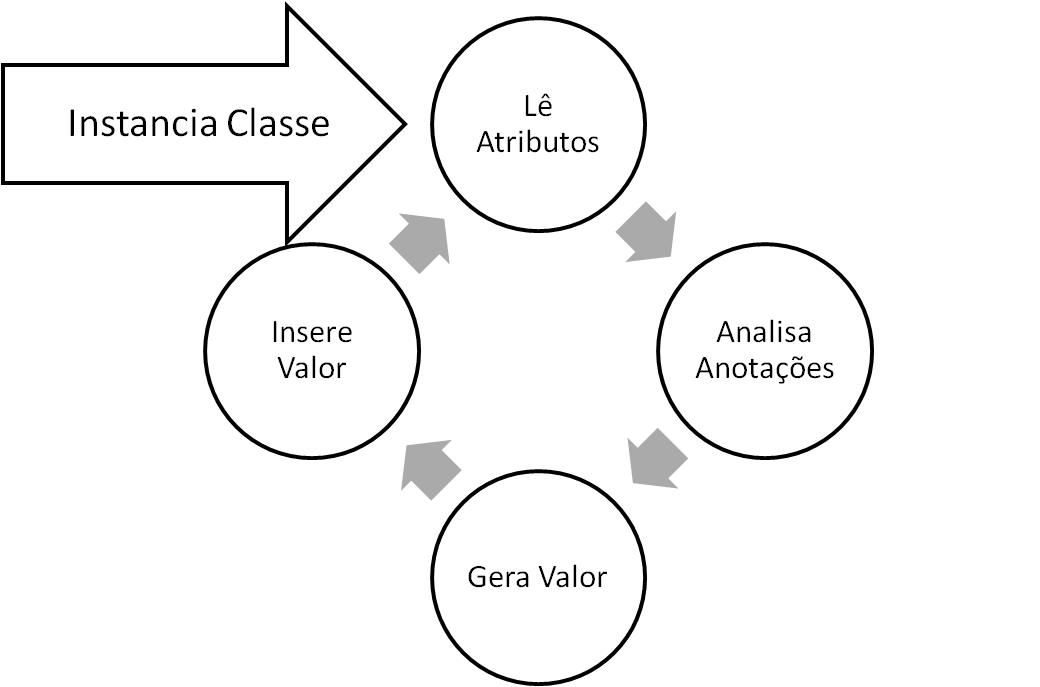
* + 1. Reflection no Make

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, lendo todas as variáveis declaradas na classe, sabendo desta maneira: o tipo que deve ser criado para definir o valor na variável, avaliar as anotações da JSR-303, detectar os 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 seria 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: MakeEntity.makeEntity(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 JavaTM Persistence 2.0.
2. O Make inicia a leitura da primeira variável da classe, a ordem de leitura não obedece a mesma ordem que as variáveis são declaradas.
3. O Make analisa as anotações da JSR-303 e seu tipo a cada variável percorrida.
4. Com base nas informações do terceiro passo, o Make irá gerar o valor adequado para a variável, variáveis sem a anotação @NotNull da JSR-303 são ignoradas, desde que não tenham um fábrica especializada apontando para a variável em questão.
5. O Make insere o valor gerado na variável da classe instanciada e inicia a leitura da próxima variável da classe.
6. O Make entrega um objeto do tipo da classe solicitada com valores gerados ao desenvolvedor.

  
Figura 20 - Tarefas realizadas através da Reflexão Java.

* + 1. Camadas de distribuição:

O Make funciona sobre três conjuntos de tipos de classes:

* Classes que geram valores tipos primitivos, basicamente números, letras, booleanos e datas.
* 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.

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 21 - Dependência 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. A interface, que também pode ser uma classe abstrata, é o contrato.

* + 1. Primitivos aleatórios:

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

É importante entender que primitivos descritos a seguir não que dizer o mesmo que primitivos Java, que são double, float, long, int, short, byte e char. Quando referenciar primitivos estou generalizando entre números, textos, booleanos e datas.

As classes que geram valores primitivos possuem métodos estáticos para geração de valores aleatórios, para os métodos estáticos não existe contrato, contudo um comportamento foi definido em todas as classes numéricas que geram esses tipos de valores. Todas as classes numéricas 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 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 possuem métodos estáticos para gerar valores primitivos são:

Figura 22 - Classes Fábricas de primitivos

Um exemplo de utilização de um dos métodos estáticos pode ser visto na figura 23, 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 23 - Utilização de geração de primitivos aleatórios.

A classe MakeBoolean cria valores booleanos aleatoriamente, possui apenas um método estático:

* Boolean 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.

A classe 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, NUMBER para somente números, SYMBOLL para somente símbolos, LETTER para somente letras (neste caso será solicitado que o método String getLoren(caracteres) gere a informação) e ALL para todos os tipos.
* String getString(caracteres, StringType): onde caracteres é a quantidade exata de caracteres que conterá na String.
* String getLoren(caracteres): 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.

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 JSR-303, ou para valores de superclasse, ou qualquer outra tarefa que o desenvolvedor necessite.

* + 1. Fábricas numéricas

As classes que geram números estendem a classe abstrata MakeNumber, essa classe possui alguns métodos implementados, métodos que possuem funcionalidades, e métodos abstratos, esses últimos quem a estende deve implementar.

A criação de um contrato exclusivo para trabalhar com números foi necessária devido a grande possibilidades de tipos numéricos do Java, Double, Float, Long, Integer, Short, Byte, BigDecimal e BigInteger, sendo que os seis primeiros tipos possuem objetos e tipos primitivos. Essas variações impõem restrições de comprimento e precisão, causando necessidade de implementação individual para cada tipo.

Outro motivo da criação de um contrato individual para números é a variedade de anotações de validação da JSR-303 para os tipos numéricos, as anotações @Min, @Max, @Digits, @MaxDecimal e @MinDecimal, cada fábrica numérica deverá saber quais anotações deverá analisar para seu tipo numérico em questão.

Os métodos implementados dentro MakeNumber realizam o trabalho que pode ser compartilhado por todas as classes que a estendem, por independerem de tipos, são métodos que toda classe que é uma fábrica numérica pode utilizar, esses métodos são responsáveis por: verificar a existência de anotações de validação da JSR-303, tratar quantidade de dígitos (mínima ou máxima) e tratar as casas decimais dos números de ponto flutuante.

Os métodos abstratos determinam o contrato assinado pelas classes que estendem MakeNumber, através deste contrato as fábricas gerenciam os valores que serão gerados.

Esse contrato irá permitir transparência na comunicação de NumberFactory com as fábricas numéricas, NumberFactory irá perguntar a todas fábricas numéricas quem pode gerar o número solicitado pela Factory, a fábrica que responder “eu posso” irá realizar a tarefa.

Figura - Relacionamento entre Make, NumberFactory e as fábricas numéricas

Os métodos que definem o contrato de MakeNumber são:

* boolean isMyType(Field): Este método informa se a fábrica numérica 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 MakeLong, a resposta será verdadeira, mas se for executado na classe MakeDouble, a resposta será falsa. Isso possibilita a NumberFactory saber qual fábrica numérica 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 JSR-303. Quando insertValue é chamado ele utiliza os métodos que herda de MakeNumber para saber os limites da validação da JSR-303 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 JSR-303, as fábricas especializadas devem conhecer os limites da validação e se responsabilizam em gerar os valores para a variável.
  + 1. Fábricas de valores:

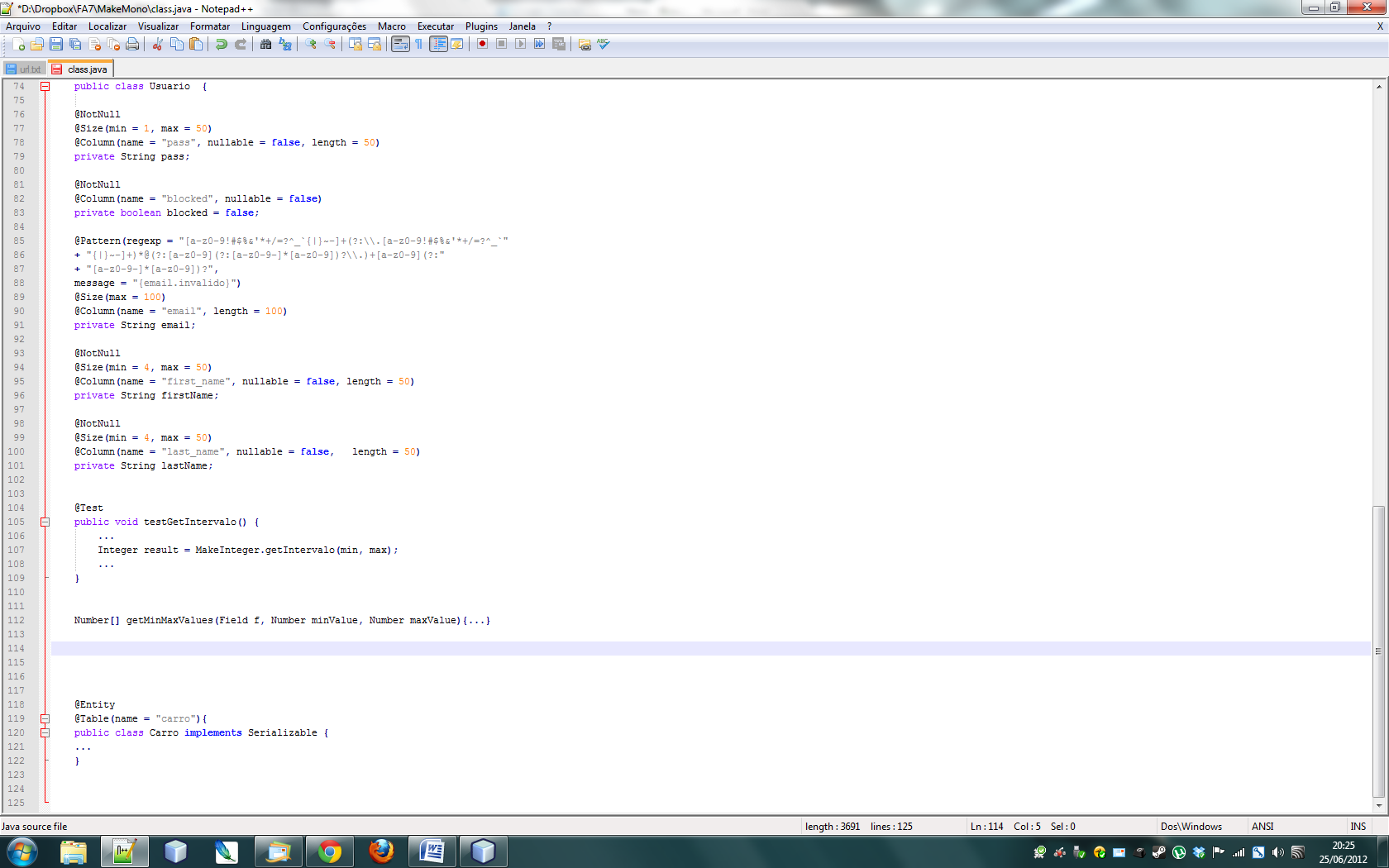
Quando um destes métodos é utilizado, internamente, o framework irá definir um conjunto de fábricas, cada uma responsável por gerar um valor para ser inserido em cada Field da entidade a ser criada.

As fábricas são responsáveis por fabricar os valores randomicamente, limitados a validação definido pelo desenvolvedor através da especificação JSR-303.

Figura 25 - Interface ValueFactory (Contrato obrigatório para as fábricas)

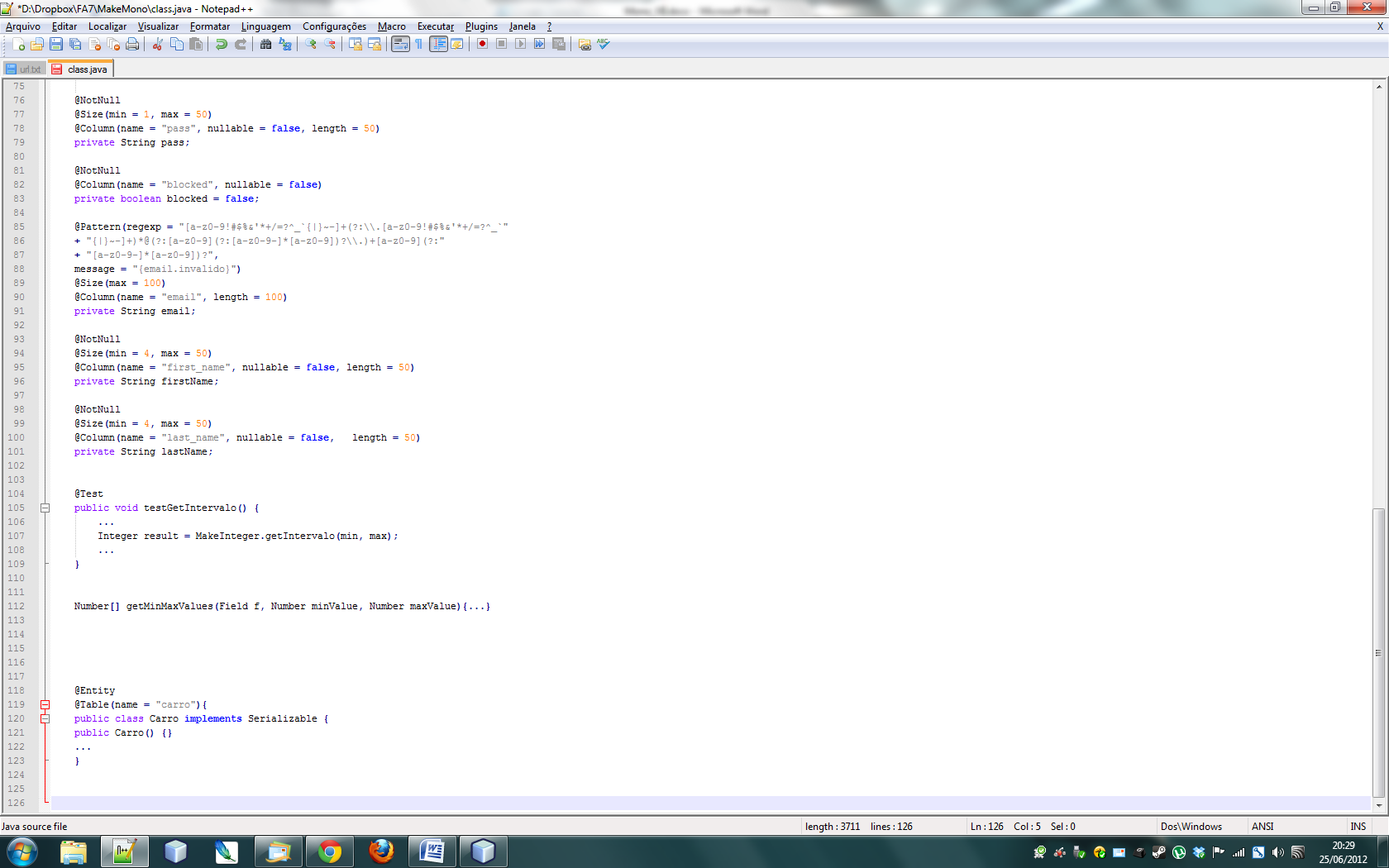
Para ser uma fábrica, é necessário que a classe implemente a interface ValueFactory, essa interface determina um contrato que todas as fábricas devem seguir, esse contrato é a implementação de métodos que definem como os valores são inseridos no Field, através do método ValueFactory.makeValue(testName, field, entity, relationships). O atributo testName é o nome do teste, utilizado nas fábricas especializadas, field é o que receberá o valor gerado, entity é o objeto instanciado proprietário do field que receberá o valor 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.

Figura - Relacionamento através de ValueFactory

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.

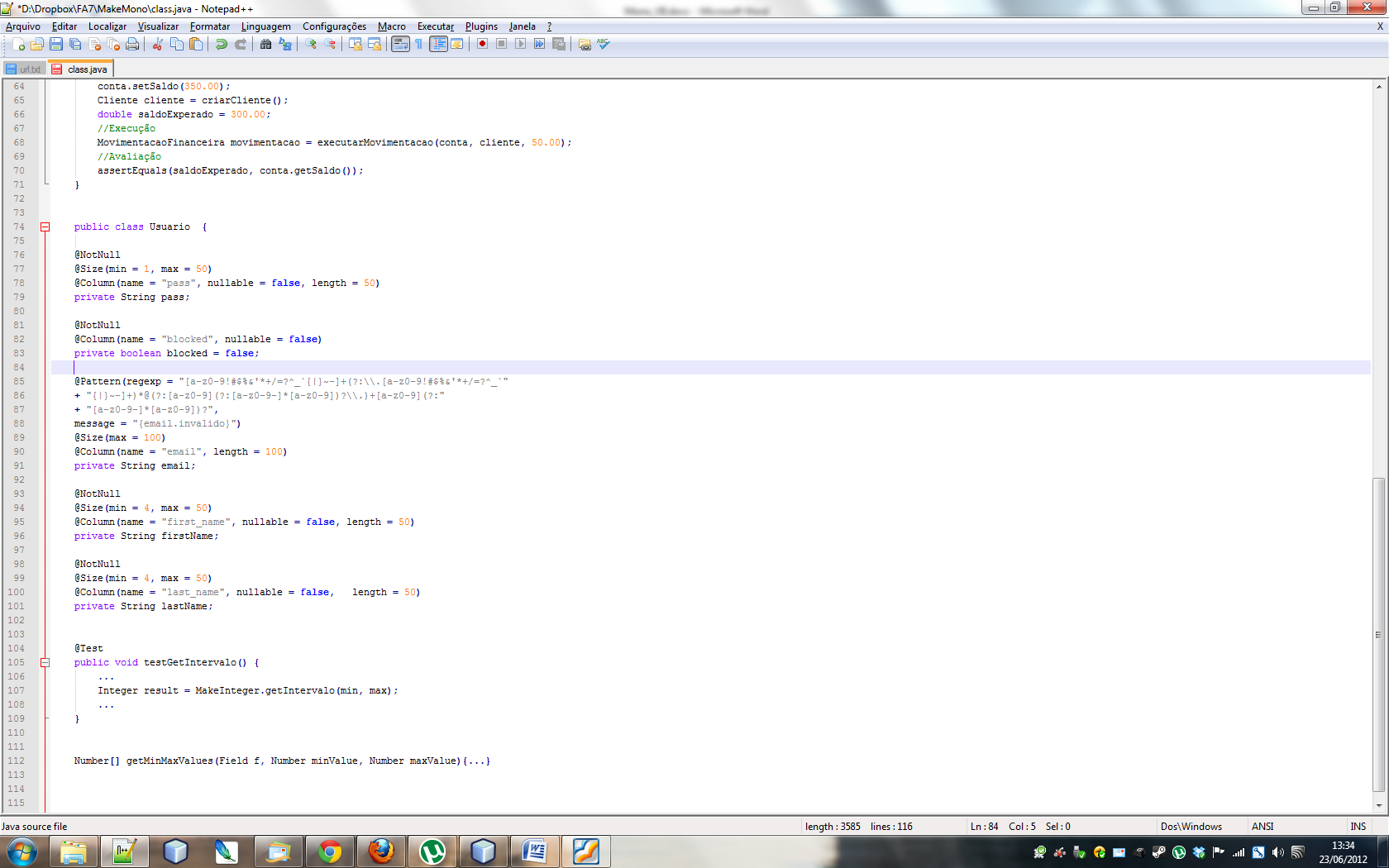
As fábricas foram divididas em tipos que trabalham para uma ou um grupo de anotações da especificação JSR-303:

|  |  |
| --- | --- |
| Anotação JSR-303 | Fábrica Make |
| @AssertFalse, @AssertTrue | TrueFalseFactory |
| @DecimalMax, @DecimalMin, @Digits, @Max, @Min | NumberFactory |
| @Future, @Past | DateFactory |
| @Size | SizeFactory\* |
| @NotNull | EnumFactory ou DefaultFactory |
| Fábrica que gerencia fábricas especializadas, não verifica as anotações da JSR-303. | MakeWorksFactory |

Tabela 2 - Fabricas Make para anotações JSR-303   
\*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-303 determina que anotações como @NotNll podem ser inseridas juntamente com outras anotações, por exemplo:

  
 Figura 29 - Duas anotações da JSR-303 em um mesmo Field

Veja que o Field do tipo String com nome de pass possui duas anotações da JSR-303, @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.

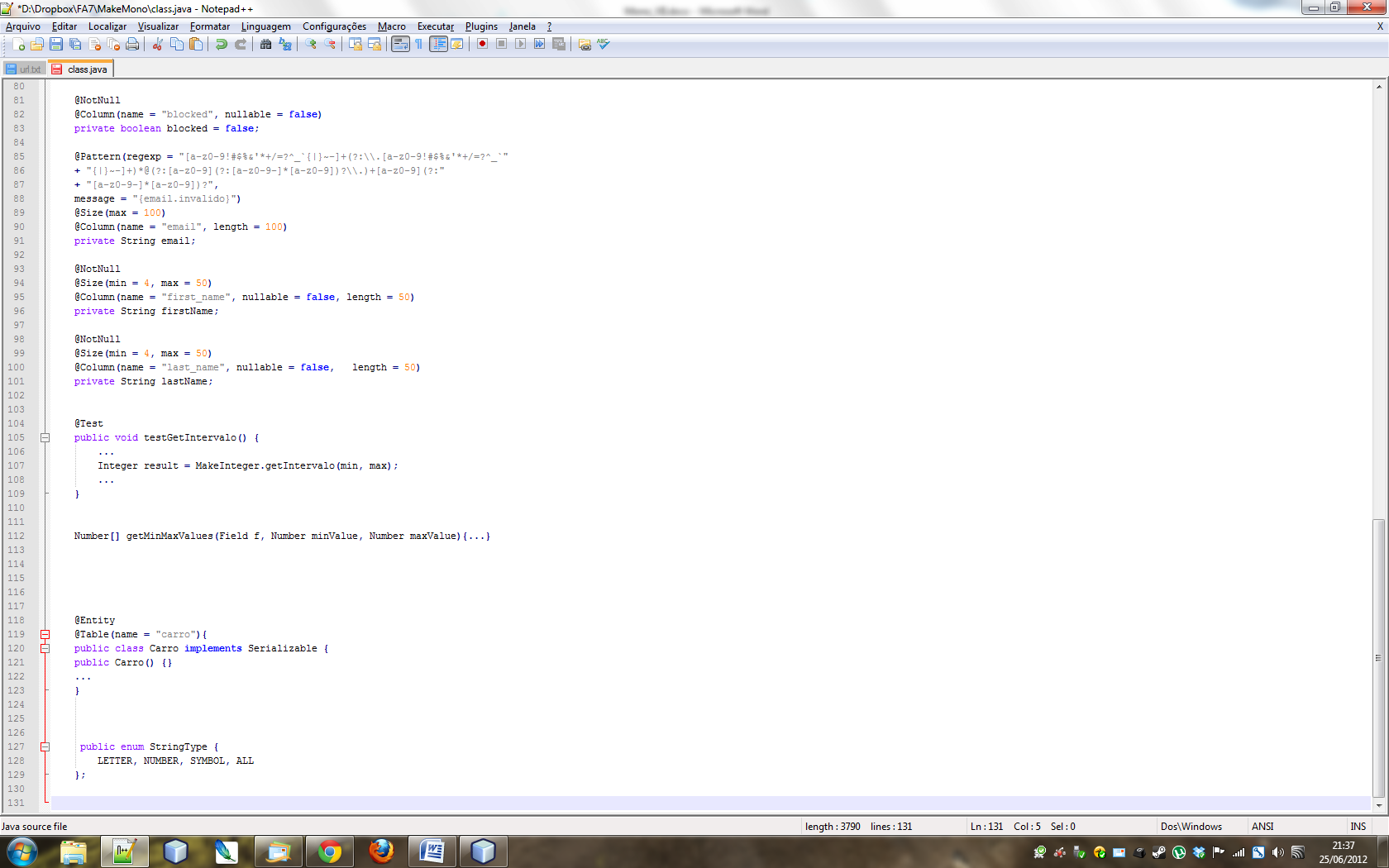
Duas anotações da JSR-303 não foram mencionadas na tabela 2, @Null e @Pattern.

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 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 possui 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.

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.

* + 1. Fábricas especializadas:

As fábricas pré-existentes no Make criam valores válidos para as validações padrão da JSR-303, 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.

As fábricas especializadas inclusas no Make são:

|  |  |  |  |
| --- | --- | --- | --- |
| 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 3 - Fábricas especializadas Make

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.

Figura - Ordem de preferência na busca por uma fábrica que gere um valor para um Field

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 JSR-303, 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.

* 1. 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 JSR-303, 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 JSR-303.

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.