Lab3 Creational Patterns CES28 2017

Leitura Prévia

Estudem os DPs indicados abaixo. Indico o livro original de DP da GoF, pelos bons comentarios (especialmente o referenciado no exercício 4), mas também são válidas outras referências como os catálogos que utilizaram hoje.

O trabalho para a dupla será discutir e chegar em diagramas da solução, garantido que não haverá falhas de entendimento do problema e que estão no caminho correto. Entrega em 2 semanas depois. Os 20% extra é uma chance para quem precisa de nota.

- Builder
- Fluent Builder (jlordiales blog)
- Factory Method (atenção, é diferente de Static Factory Method)
- Abstract Factory
- Variação de Abstract Factory: [GoF 94], pp 90, item (2. Creating the Products)
- Bridge

Problema

Rationale: Os primeiros exercícios são aplicações diretas dos padrões de projeto recomendados. Os últimos exercícios exigem adaptação e combinação de padrões de projeto.

Problema: Gerar automaticamente modelos de cartas. Exemplo, CartaComercial, CartaPessoal, ModeloRelatorio.

Uma carta deve ter no mínimo as seguintes partes: Cabeçalho, Corpo, Conclusão e Assinatura. Cada parte pode ter subpartes. O cabeçalho normalmente é a parte mais complexa e variável.

Simplificação: Uma carta é apenas uma string, com "\n" para separar linhas. Por exemplo, se o usuário deseja gerar um modelo de Carta Comercial, deverá fornecer os dados necessários, e as suas classes deverão gerar uma string que contém o conteúdo da carta.

- Não percam tempo excessivo com formatação. Por exemplo, se quiser alinhar um nome à direita, simplesmente acrescentem alguns espaços no começo da linha ao invés de fazer um cálculo exato. Não se preocupem com fontes, cores, títulos. Suponha apenas um .txt. Não percam tempo com nomes por extenso de datas, pode apenas imprimir os numeros "03/30/2015" (mas a ordem MM-DD-YYYY é importante para o inglês, por exemplo). A idéia é mostrar os padrões funcionando com exemplos reais (invente modelos de cartas apropriados) mas sem gastar tempo com detalhes de formatação.

Regras

1. **Duplas de 2 indivíduos. Não dupla de 1, nem de 3, etc.**
2. **CADA EXERCÍCIO DEVE SER RESPONDIDO COM JUnit @Tests APROPRIADOS, MOSTRANDO COMO UM CLIENTE USA AS CLASSES DESENVOLVIDAS PARA GERAR O RESULTADO FINAL (CARTAS COMPLETAS)**.
    a. Como as cartas são strings, o conteúdo correto delas pode ser colocado hardcoded no código dos testes, comparando com a string gerada efetivamente[1]. Isto deixa claro no código do teste qual o formato exato esperado, sem precisar analisar saída em arquivos ou outras dificuldades, e **sem precisar ler e comparar cartas manualmente**.
    b. É ***preciso submeter todo o código***, inclusive com os @Test que geram as respostas de cada exercício.
    c. Pode apresentar um conjunto de classes completo implementando o pedido em todos os exercícios, ou pode separar em packages que representam versões intermediárias que resolvem apenas os exercícios iniciais. Deve haver @Tests demonstrando cada exercício em separado, indicando claramente a qual exercício se refere cada @Test. O propósito é garantir que ainda é possível fazer o mais simples mesmo depois de implementar o mais complicado.
3. Os últimos exercícios requerem apresentar um diagrama de classes.
    a. Vale foto de diagrama desenhado a mão. **Mas garanta que é legivel**!
    b. Não precisa diagramar todas as classes e todos os detalhes, apenas aquelas relevantes para a solução e os DPs usados.

[1] Lembre: compara-se Strings com o método .equals() e nao com '=='

    c. Os primeiros exercícios não precisam de diagrama, já que são uma aplicação mais direta dos DPs.
4. Não é preciso seguir DPs à risca como se estivessem escritos em pedra. Eles podem (e talvez devam) ser adaptados às necessidades do projeto.
5. ***Mau cheiro excessivo perde ponto.***
6. Vale procurar por qualquer explicação de DP na internet, livros etc.
7. Vale copiar (copy-paste) um padrão de projeto pronto de um exemplo de livro / catálogo e adaptar para o seu projeto. Mas cuidado com os nomes, nomes apropriados fazem parte do "cheiro" do código! Cuidado, em não usar uma solução mais complexa ou mais simples do que o necessário.
8. ***OS @Testes NÃO DEVEM PARAR PARA PEDIR ENTRADA DO USUÁRIO***. Entrada de dados deve ser Mockada. Vale criar classes que respondam dados predefinidos; Vale ter uma lista de Pessoas já preenchida, hardcoded no teste; Vale mocks que gerem dados em sequência, e.g. {"nome1", "rua dos bobos, no 1", "telefone (01) 0001-0001"} e na próxima chamada {"nome2", "rua dos bobos, no 2", "telefone (02) 0002-0002"}; Vale mocks que respondam dados lidos de arquivos. MAS NAO VALE PARAR OS TESTES PARA PEDIR QUE O HUMANO DIGITE UMA ENTRADA. TODOS OS DEMOS DEVEM EXECUTAR INSTANTANEAMENTE ATÉ O FIM!!!!!!! Portanto o demo segue uma sequência programada de entradas.
9. Vale implementar outros testes para testar ou debugar resultados intermediários, e pode entregar junto com o código, com nome apropriado, mas **deve ser claro quais são os testes que efetivamente respondem cada exercício**.

Ex 1: Gerar vários modelos de carta

O objetivo é um projeto de classes que permita gerar vários modelos de carta com as mesmas partes básicas, que podem ser divididos em subpartes se necessário. Um modelo de carta comercial poderia ser gerado assim:

```
class ComercialLetter {
      ComercialLetter(Person sender, Person destinatary,
                        ,Address          addressSender,          Address
                        addressDestiny, Date date ) {
            // armazena todos os dados necessarios
      }
      public String model() {
            return header() + body() + conclusion() + signature();
      }
      protected String header() {
            return date_.toPrint() + "\n\n" + sender_.name() + "\n" +
            addressSender_.toPrint() + "\n" + destinatary_.name() +
            addressDestiny_.toPrint() + "\n";
      }
      protected String body() {
            return "Dear "+destinatary_.name() + "\n" ;
```

```
	}
	protected String conclusion() {
		return “\nSincerely,\n”;
	}
	protected String signature() {
		return  “\n\n                                    _____________\n
”+ sender_.name() + “\n                ” + sender.phone().toPrint() +
“\n            email:” + sender.email();
		//espaços representam justificação à direita.keep it
simple!
	}

}//class Commercial Letter

// o cliente desta classe apenas precisa chamar:
String lettermodel = CommercialLetter.model();
// agora lettermodel pode ser trabalhada por um editor de texto.
```

Note acima como CommercialLetter é responsável pelo layout da carta, mas não pelo formato da impressão de cada campo (nome, fone, email)

Implemente classes que permitam ao programador usuário escolher facilmente um modelo de carta entre os modelos existentes e gerar a string correspondente. Implemente no mínimo 3 modelos (2 deles podem ser parecidos, mas diferentes; pelo menos um deles deve ser bem diferente dos demais)

Deve ser fácil adicionar novo código para incluir um novo modelo de carta.

Dica: Estude o DP Builder.
Dica: Estude o pdf sobre Builder Fluente! É opcional, mas não é interessante?

Ex 2: Várias línguas para as subpartes

O formato em que se escreve o nome, endereço, telefone, etc., pode mudar, por exemplo, com a língua escolhida para a carta. Note que aqui não mudamos o layout ou formatação geral da carta. Mudamos apenas o formato em que cada campo (nome, endereço, telefone) é escrito.

Implemente uma solução que torne fácil para o programador usuário das suas classes escolher uma língua e gerar de forma consistente, formatos de endereço, nome (Person.nome() pode incluir títulos como “Mr”, “Sr”, “Messier”), telefone, data, etc. Deve ser fácil adicionar novo código para suportar uma nova língua.

Dica: Estude os DP AbstractFactory e FactoryMethods, e implemente subclasses que tratem de imprimir nos formatos corretos e nas diversas línguas.

Ex 3: Vários Modelos, subpartes e línguas.

NECESSÁRIO APRESENTAR DIAGRAMA DE CLASSES

Estenda o Ex 1 para que o usuário-programador possa escolher uma língua. Ao escolher uma língua, automaticamente, sem nenhuma outra configuração, todas as partes do modelo serão geradas já traduzidas e no formato correto para a língua escolhida.

TODOS os modelos implementados anteriormente devem ser gerados em inglês e português, no mínimo, incluindo formatação dos campos (data MM-DD-YYYY versus DD-MM-YYYY) e tradução apropriada para cada língua.

Dica: Combine os 2 exercícios anteriores (que eu saiba não existe a solução completa em nenhum livro. Um DP que pode ajudar a inspirar é o Bridge). Os modelos de carta devem saber qual língua deve ser usada, e usar Factories para gerar subpartes na língua correta. Outra dica: leia, pense, discuta com o parceiro e diagrame antes de codificar !!

Nota: O formato (layout) da carta em si independe da língua. Nada impede que se use um formato CommercialLetterUSA americano em português, com “31/03/2016” “Sr. Smith” e “(12) 99999-9999”, mas com a ordem e disposição dos campos usual em inglês. Portanto, se o usuário quiser mudar o layout da carta também, em vez de apenas traduzir, deve criar outro modelo independente ComercialLetterBrasil.

Ex 4: Subpartes Dinâmicas

NECESSÁRIO APRESENTAR DIAGRAMA DE CLASSES

Um programador com acesso às classes do Ex 2 e Ex 3 pode escolher uma língua, gerar um modelo e em seguida mudar a representação de uma das subpartes sem reimplementar todo o Ex 2. Ele programa a sua nova representação da impressão de uma subparte em uma língua específica e redefine, dinamicamente, as classes do ex2 para gerar a sua nova representação.
Por exemplo:

- Antes, em portugues, os telefones eram impressos na forma “(99) 99999-9999”
- Agora o programador quer redefinir a impressão de telefone em português para “DDD: 99 Tel: 99999.9999”. Ele pode escrever uma nova subclasse para tratar de imprimir da nova forma desejada, mas deve modificar dinamicamente a solução existente para utilizar a sua nova subclasse.
- Na próxima vez que forem geradas cartas em português, o formato do telefone deverá ser o alterado.

Dica: [GoF 94], pp 90, item (2. Creating the Products) propõe uma forma de alterar dinamicamente uma AbstractFactory. Outra dica: leia, pense, discuta com o parceiro e diagrame antes de codificar!!
Update: o que resolve o problema é a factory com dicionário interno, não usar prototypes. O objetivo é uma factory onde a associação entre um nome e uma classe pode ser facilmente

modificada. Antes pedir “english” retornava um objeto da classe X, agora retorna um objeto da classe Y. É possível, talvez até interessante, usar Prototypes, mas não é o ponto da questão.

Ex 5: Opcional – Modelos Associados às Línguas

Opcional: Vale 20% pontos extra (deste exercício de Creational Patterns).
NECESSÁRIO APRESENTAR DIAGRAMA DE CLASSES

Ao escolher uma língua, o usuário das classes automaticamente receberá uma carta comercial no formato correspondente à língua escolhida, não apenas considerando o formato dos campos, mas também a ordenação e disposição dos campos na página.

Deve existir então um formato Carta Comercial ING e Carta Comercial BR, que diferem não na tradução em si de cada campo (isso é um problema das subpartes) mas no layout, ordenação e disposição dos campos na página.

Por exemplo, vejam em http://www.alunosonline.com.br/portugues/carta-comercial.html, um exemplo de carta comercial brasileira: inclui o nome da cidade na primeira linha, o que não é usual nos EUA (http://www.wikihow.com/Sample/Business-Letter)!

Portanto, ao escolher a língua portuguesa para uma carta comercial, por default, automaticamente o modelo comercial brasileiro será o usado para gerar uma carta comercial com todas as subpartes traduzidas para o portugues.

MAS, se o programador usuário das classes escolher explicitamente uma língua para o modelo e/ou para uma ou várias subpartes, ou para todas as subpartes de uma vez, ainda deve ser possível, por exemplo, gerar o modelo comercial americano traduzido para português. Ou seja, ao escolher a língua, além de traduzir as subpartes, um modelo associado a língua é produzido por default, mas isso não impede que se produzam outras combinações de modelos de layout e subpartes associados a outras línguas, se indicado explicitamente.

Implemente uma solução elegante para permitir isso.

Dica: Aberto! Pense em uma forma elegante de fazer!!!

Referência

[GoF 94]
https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=sr_1_1?ie=UTF8&qid=1475697934&sr=8-1&keywords=design+patterns+GoF