P3 – CES28 – Dicksiano Melo

DPs são tijolos para construir Frameworks

Primeiro DP: Facade
a) Imagine que a Framework passe por uma grande mudança e os seus sistemas internos sejam alterados. É razoável alterar uma Framework internamente, porém, espera-se que a Framework continue a fornerce as mesmas funcionas e, em certos casos, apenas aumente o número de serviços providos. Imagine uma Framework utiliza para renderizar figuras. É razoável que seus criadores troquem os algoritmos de Computação Gráfica utilizados por métodos mais eficazes. Contudo, espera-se que a mesma Framework continue renderizando figuras, afinal essa é a função dela. Assim, mesmo que o sistema sofra uma grande alteração, a sua fachada com o usuário pode permanecer a mesma, aproveitando todo trabalho feito anteriormente.

Imagine outro caso: eu possuo uma aplicação que utiliza uma determinada Framework. Essa Framework sofre uma drástica alteração dos seus subsistemas internos. Contudo, minha aplicação não é afetada, pois ela só enxerga a fachada da Framework e, desse modo, utiliza apenas o método daquela fachada. A Framework mudou drásticamente, e eu atualizei minha aplicação para utilizar essa nova versão, porém o meu código não foi alterado me poupando uma grande carga de trabalho.

b) Observe que dado uma Framework (que pode ter grande complexidade) e um programador-usuário, o programador, em boa parte dos casos, não precisa saber como a Framework funciona internamente. O programador, em geral, deseja apenas utilizar a Framework para atingir o seu objetivo de forma mais simples, efiente e segura. Assim, ele precisa apenas acessar aquilo que lhe interessa, sem perder tempo com os detalhes de funcionamento da Framework. Assim, se um programador utiliza uma Framework para renderizar gráficos em sua aplicação, ele não precisa saber quais algoritmos de Computação Gráfica a Framework utiliza, ou se ela utiliza mais de uma Thread ou como ela acessa o hardware, ele precisa apenas de uma interface para chamar os métodos daquela Framework (por exemplo, renderizarLinha() ou redenrizarCircunferencia). Desse modo, é muito importante que os métodos de interesse dos programadores-usuarios sejam fáceis de ser utilizados e estejam acessíveis da maneira mais simples possível. Por isso o padrão Facade é tão interessante. Utilizando esse padrão, os criadores de uma Framework isolam todos os subsistemas complexos e expões os métodos que serão utilizados pelo programador-usuario. Imagine um sistema bem complexo com vários métodos utéis para o programador-usuario espalhados em diversas classes, o programador-usuario terá que buscar esses métodos dentro das classes e depois disso terá que memorizar onde está cada método. Observe que a curva de aprendizado dessa Framework é muito ruim. Já com o padrão Facade, é possível colocar os métodos úteis para o programador-usuarios em uma única classe e ele terá consciência de que precisará apenas buscar o método que deseja em uma única classe, melhorando significativamente a curva de aprendizado dessa ferramenta.

Segundo DP: Template Method

a) Imagine uma Framework que implemente um algoritmo bastante complexo formado por diversos passos menores. A minha aplicação utiliza esse algoritmo, porém ela gera resultados diferentes dependendo do contexto. Em vez de implementar tudo 2 vezes, eu utilizaria o template method escrito

dentro da Framework e me preocuparia apenas em escrever o passo final da minha aplicação sem perder tempo voltando para o início.

Um exemplo simples seria um Framework que gerasse um modelo preditivo produzido através de Machine Learning. Uma vez gerado esse modelo, minha aplicação específica poderia testar a confiabilidade desse modelo utilizando um novo conjunto de entradas ou guardar esse modelo preditivo em algum banco de dados seguro. Em ambos os contextos, todo o processamento para gerar o medelo preditivo seria o mesmo e utilizaria o mesmo template method.

b) O Template Method é muito utilizado em Frameworks exatamente porque ele pode ser utilizado para semparar os interesses do Framework e do programador-usuario. O Framework ficaria responsável pelos métodos mais gerais e o código do programador usuário ficaria responsável por expadir aqueles métodos para o contexto da sua aplicação sem entrar nas camadas internas do Framework.

Abusus non tollit Usum

a)
Singleton DP – 2. Acoplamento excessivo e código difícil de entender devido à proliferação de Dependências e conflitos de nomes

Dependency Injection – 1. Excessiva quantidade de código e classes auxiliares para inicializar objetos

Getters and Setters – 3. Confusão semântica dependendo da ordem de chamada de métodos, resultando em objetos com estado inválido.

b)

Dependency Injection – 1. Excessiva quantidade de código e classes auxiliares para inicializar objetos

O uso exagerado de Injeção de Dependências pode levar o código a se tornar desnecessariamente extenso, uma vez que haverão diversas partes em que ocorre a injeção da dependência na classe dependente. Além disso, uma classe em que se injetam diversas depêndencias é difícil de ser construída, pois se ela depende de diversas outras, é preciso injetar essas depêndencias para que ela funcione. Imagine um classe que dependa de outras 30 classes. E imagine que essas depêndencias são injetadas pelo contrutor. O programador deveria, pelo menos, criar uma classe builders para facilitar a chamada desse construtor e mesmo assim esse código ficaria muito mal-cheiroso.

Singleton DP – 2. Acoplamento excessivo e código difícil de entender devido à proliferação de Dependências e conflitos de nomes

Imagine um código em que há várias classes dependentes de um objeto Singleton Esse código tenderá a gerar erros uma vez que o Singleton iria funcionar como uma Variável Global da qual vários objetos dependem. Como existe uma única instância, várias instâncias de mesma classe iriam ter que acessar esse singleton e isso poderia causar uma grande confusão.

A propriedade do Singleton de ser único acaba tornando alguns programadores desatentos, pois eles evitam se dar ao trabalho de pensar melhor sobre a visibilidade daquele objeto e adotam essas solução como o caminho mais fácil sem se preocupar com os possíveis problemas.

c) O Singleton é útil desde que seja usado com cautela. No lab04, por exemplo, seria uma boa solução a ser adotada para os bancos de dados, pois eles deveriam ser únicos. Em GameDevelopment costuma-se usar singleton para classes que guardam propriedades do jogo que são acessados por diversos métodos. Vale ressaltar que o Singleton causa grandes discussões e alguns preferir abolir o seu uso.

Observação: Achei complicado definir uma única consequência para cada um dos 3 conceitos. Seria possível trocar as consequências escolhidos, visto que um abuso de conceito pode ter mais de uma consequência dependendo da forma que esse conceito é usado. Porém, optei pela consequência que me parece mais comum.

Joãozinho programa Interpolação

	M	V	P
1. RESPONSABILITY: DEFINIR PONTO DE INTERPOLACAO (LEITURA ENTRADA DE USUARIO HUMANO)		X	
2. RESPONSABILITY: DEFINIR QUAL EH O ARQUIVO COM DADOS DE PONTOS DA FUNCAO (LEITURA ENTRADA DE USUARIO HUMANO)		X	
3. RESPONSABILITY: ABRIR E LER ARQUIVO DE DADOS			X
4. RESPONSABILITY: IMPRIMIR RESULTADOS		X	
5. RESPONSABILITY: DADO O VALOR DE X, EFETIVAMENTE LER O ARQUIVO			X
6. RESPONSABILITY: DADO O VALOR DE X, EFETIVAMENTE CHAMAR O CALCULO			X
7. RESPONSABILITY: CRIAR O OBJETO CORRESPONDENTE AO METODO DE INTERPOLACAO DESEJADO	X		
8. RESPONSABILIDADE: EFETIVAMENTE IMPLEMENTAR UM METODO DE INTERPOLACAO	X		

1. Observer que temos um package para cada módulo de MVP. Além disso, View enxerga apenas Presenter e Model também enxerga apenas Presenter, obedecendo ao modelo MVP.

2. ViewAbstract é uma classe abstrata. Para gerar novos Views bastaria criar novas classes derivadas de ViewAbstract e utilizar o setter de Presenter para trocar a View.

Além disso, eu fiz Presenter ser Singleton, pois toda vez que o cliente criar um novo tipo de View, ele sempre vai acessar a mesma instância de Presenter sem modificar o código de prensenter.

Espero ter interpretado o problema de maneira correta. Inicialmente, pensei em criar um setter para o view de presenter (eu fazia **presenter.setView(view2)**). Porém, nesse caso o cliente teria que conhecer o objeto de Presenter. Da forma que eu fiz, a Main só instancia os objetos de View e esses objetos de View automaticamente de conectam o Singleton de Presenter. Essa me parece ser a solução desejada, pois o cliente (que nesse caso é a Main) só trabalha com View e Presenter/Model ficam camuflados, além de garantir que Presenter/Model possua instância única. Salvei a primeira solução (que me parece mais simples) no meu computador, mas imagino que o desejado seja a solução final.

3. Para isso, eu adicionei uma Hashtable do tipo **<String,InterpolationMethod>**. Assim, sempre que o usuário quiser adicionar um novo método, passa adicionar o novo método e uma String utilizando o método **addMethod()** que o representa e **getMethod()** utilizará o método adequado através da Hashtable.

GRASP x SOLID

1. **Open-Closed Principle**

Esse princípio afirma que o software deve ser aberto para extensões e fechado para modificações.

Perceba que adotando o novo modelo MVP o código está aberto para extensões, uma vez que é possível utilizar diferentes instâncias de View. E isso ocorre sem alterar o código de Model e Presenter, ou seja, o software está fechado para modificações.

Novamente, é possível implementar e escolher outros algoritmos de interpolação e, dessa forma, expandir a camada Model sem quebar o código feito para as camadas View e Presenter, ou seja, mais uma vez o conteúdo está fechado para modificações.

2. **Liskov Substitution Principle**

Observe que existe a exigência de usar várias instâncias de Views. A solução adotada é fazer com que Presenter dependa de uma classe abstrata View. Desse modo, caso se deseje novos tivos de View basta criar novas classes herdeiras dessa classe abstrata. Assim, cumpri-se o Princípio da Substituição de Liskov, uma vez que é possível substituir a classe pai pr qualquer um de seus derivados.