Universidade Federal de Campina Grande – UFCG
Centro de Engenharia Elétrica e Informática – CEEI
Departamento de Sistemas e Computação – DSC
Disciplina: Laboratório de Programação 2

Laboratório 06 - 11/06/2015 - 24/06/2015

Neste laboratório iremos praticar **Composição** e o uso de interfaces como **List**, **Comparable** e **Comparator**. Para praticar a mudança dinâmica de comportamento na composição, iremos utilizar Herança. O objetivo é resolver **o mesmo problema do Lab 05**, porém usando a Composição em conjunto com Herança para reduzir o acoplamento no design. Dessa forma, utilize a oportunidade para observar as consequências das modificações no seu código (impacto das modificações). Por exemplo, verifique quais as classes que começam a apresentar erros (de compilação e funcionamento).

Iremos também praticar mais o uso de Enumerations para facilitar a manipulação de constantes, e manter a preocupação com a hierarquia de Exceptions, javadoc e diagrama de classes.

Implementação:

Você foi contratada(o) para **melhorar** o sistema ***P2-CG: Programação 2 - Central de Games***. Como discutido no Lab 05, o P2-CG *armazena e gerencia uma coleção de jogos de um usuário*. Semelhante à plataforma **Steam**, os usuários do sistema podem comprar diferentes jogos da loja, e com isso acumular diversos pontos que fornecem benefícios e reconhecimento em meio à comunidade de jogadores da plataforma. **Algumas funcionalidades foram adicionadas** na manipulação de jogos, como por exemplo, o uso de um catálogo de jogos no Usuário. **Forneça, testes, especificação e trate os erros por meio de Exception. Use Herança para fazer um tratamento sofisticado de Exceptions.**

Passo 1: Refatoramento do Lab 05: Façade e Controller

O objetivo desse passo é um exercício do **GRASP** (atribuição de responsabilidades). A classe **Loja** estava com um design de **baixa coesão**, já que exercia diversas funções além de sua responsabilidade de armazenar e manipular os usuários. "*Quebre*" a classe Loja em **3 classes distintas**: **Loja**, **Façade** e **Controller**. Cada classe tem as respectivas responsabilidades:

- **Façade:** Delegar (***forwarding***) as chamadas da View (camada do *user*[1] do sistema) para o Controller. Já que não teremos uma camada de Interface Gráfica, a captura de Exceptions e impressões serão feitas na Façade.
- **Controller:** Gerenciar as comunicações entre as classes da lógica de negócios (Jogo, Loja e Usuário). Será a responsabilidade do Controller gerenciar os **Factory** de seu projeto.
- **Loja:** Armazenar e pesquisar os Usuários (Veteranos e Noobs).

Fique atenta(o) para as **modificações no seu diagrama de classes** para refletir esses novos relacionamentos entre as classes. **Importante:** Observe como o **acoplamento** entre Loja e o restante do projeto Usuário é **reduzido** e a **coesão** das classes **aumenta**.

Passo 2: Catálogo de Jogos

O objetivo desse passo é exercitar o uso de **Composição independente do polimorfismo** no Usuário. Vamos **encapsular** a lista de jogos de usuário em um objeto com funcionalidades mais complexas que reflitam um catálogo de jogos.

Crie um ***wrapper*** CatalogoJogos cuja função seja adicionar, remover e pesquisar os jogos do usuário (use o forwarding para isso). Tire vantagem do encapsulamento e **torne os métodos da lista (List) mais intuitivos**. Por exemplo: 'adicionaJogo' é mais intuitivo para um catálogo de jogos do que 'add' ou 'adiciona'.

- Crie **métodos para retornar os jogos** com o **maior score**, que foi **jogado mais vezes**, e que foi **"zerado" mais vezes**. **No caso de empates, retorne o mais recente na lista.**
- Também crie um método para **retornar jogos com uma jogabilidade específica**.

[1] Nos referimos ao User como um usuário sentado na frente do Computador utilizando o Sistema. Esse nome foi escolhido para evitar ambiguidades com a classe Usuario do seu sistema.

Passo 3: Organização do Catálogo de Jogos

O objetivo desse passo é exercitar o uso de **Interfaces independente de comportamento polimórfico** no Usuário. Para isso será usado um conjunto de constantes denominados **TiposOrdenacao**. Esse conjunto será utilizado para **decidir** qual o critério de ordenação utilizado para organizar o catálogo de Jogos. Esses tipos são: **Vício**, **Desempenho**, **Experiência** e ***Default***. O comportamento Default é a ordenação de jogos (de forma crescente) pelo título do jogo. Se for requisitada uma ordenação por Vicio, Desempenho ou Experiência, os jogos do catálogo devem ser ordenados, respectivamente, pela quantidade de **vezes que o jogo foi jogado**, **pelos maiores scores** e pela **quantidade de vezes que o jogo foi zerado**. Utilize as Interfaces **Comparable** e **Comparator** para atingir isso.

Implemente uma outra ordenação na **Loja** para ordenar os usuários. Implemente seu código de forma que os usuários sejam **comparáveis por meio da quantidade de x2p** que eles possuem. Depois, **crie um método no Controller** para imprimir os Usuarios ***top 5*** do P2-CG. Ou seja, os 5 usuários que possuem **MAIS** x2p.

Extra (1 ponto):

Utilize 'Composição com Polimorfismo' junto com o Comparator para fornecer dois métodos de ordenação: crescente e decrescente. A troca entre o Crescente e Decrescente (para cada um dos tipos de ordenação) deve ser dinâmica.

Dica: Verifique se o **design pattern Adapter** pode ajudar a implementação.

Passo 4: Usuário e Jogador

Para exercitar a **composição com o polimorfismo** vamos **separar as responsabilidades** do nosso objeto Usuario. Note que temos **dois papéis distintos**. O papel de um Usuário é ter e manipular as informações de sua conta como nome, login, seus jogos e pontos (x2p). O Jogador, por sua vez, contempla as responsabilidades de jogar um Jogo.

Remova a Herança entre Usuario, Veterano e Noob, e **transforme seu Usuario em uma Composição com uma Herança**. Note que o ***wrapper*** será o Usuario, e o objeto composto será o Jogador. Faça o ***forwarding*** entre Usuario e Jogador para manter os métodos 'punir' e 'recompensar' funcionando de forma polimórfica. Porém, para melhorar a legibilidade, **renomeie os métodos** punir e recompensar para, respectivamente, os métodos **'perdeuPartida'** e **'ganhouPartida'**.

Passo 5: Upgrade e Downgrade de Usuário

Este passo exercita a **troca dinâmica de comportamento** por meio de Composição. Chegou o momento de **recompensar aqueles usuários Noob (upgrade)** que acumularam pontos e mostraram **excelência** durante sua experiência de jogos e, semelhantemente, **punir aqueles Veteranos (downgrade)** que não estão à altura de seu título. Cada usuário poderá, a partir de agora, mudar de tipo de acordo com a quantidade de pontos atingida. **O limiar de pontos para upgrade e downgrade é: 1000 x2p**. Então se um usuário Noob atingir a meta de 1000 x2p (ou seja, seus pontos são **maiores ou iguais** a 1000 x2p), ela(e) será promovida(o) para um usuário do tipo Veterano. Semelhantemente, se um usuário Veterano reduzir sua quantidade de pontos para abaixo de (ou seja, **menor que**) 1000 x2p, então ela(e) será considerada(o) um usuário Noob. Note que essa transformação deve ser **dinâmica**, aonde os tipos de usuários devem refletir os seus respectivos pontos de experiência.

Ao contrário do Lab 05 ela **deve ser automática**, ou seja, ao modificar os pontos do Usuario, o comportamento do Usuario deve mudar se o limiar foi atingido (seja para acima ou abaixo de 1000 x2p). **Para isso, use a composição adicionada no Passo 4**.

Note também que ao separar o Usuario do Jogador, a atualização afeta apenas o Jogador, uma vez que o Jogador é uma classe que possui apenas comportamento polimórfico. As **informações importantes** (x2p, jogos, login, etc.) estão no Usuario. Diante disso, **o Usuario** terá apenas um método (que pode ser **private** já que a atualização é automática e depende apenas da alteração nos x2p) que irá trocar a sua instância de Jogador de acordo com a quantidade de x2p. Consequentemente, não é necessário fazer verificação com **instanceof** nem lançamento de Exceptions por tipo de Usuario inválido.

Passo 6: Diagrama UML de classes

Documente o seu projeto OO por meio de um diagrama. Junto com o projeto você deve entregar um diagrama. Existem várias ferramentas para a criação de Diagramas. Uma delas é a **Astah** que é simples de usar.

Considerações importantes para o seu projeto

- Escreva o **Javadoc** para os seus métodos. Procure ser objetivo e expressivo. Pratique suas habilidades de comunicação mencionando as funcionalidades do método e da classe de acordo com seus parâmetros e atributos. **Não seja óbvio, nem verborrágico**… seja **assertivo**. :)
- **Cuidado ao tratar as Exceptions.** Realize o lançamento e captura de forma adequada

para não quebrar o funcionamento de seu código devido ao mau gerenciamento de Exceções. Agrupe funções semelhantes usando uma Heirarquia de Exceptions por meio de Herança. Isso facilita a legibilidade do código e o processo de captura de Exception por `try/catch.`

- **Escreva os testes em JUnit para o seu código.** Para desenvolver o projeto em TDD, comece pelos testes da Façade. Ela é o que chamamos de Fachada (Façade) do seu projeto e é o **ponto de entrada** para a lógica de negócios (as demais classes de seu projeto).
- Faça a implementação do Lab em um **projeto no Eclipse.** Nomeie o seu projeto da seguinte forma: **Matricula_PrimeiroNome_Lab05**. Por exemplo:

  `114210216_Gerson_Lab05`

  A nomeação de pacotes e classes fica a seu critério. Porém, use nomes intuitivos e curtos, isso é o primeiro passo para evitar um código 'seboso'. Legibilidade é um dos critérios básicos para a avaliação.

Boa sorte e boa implementação!