**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 07 - 25/08/2016 a 05/09/2016**

Neste laboratório iremos continuar praticando polimorfismo, mas usando **composição com tipos polimórficos**. Além disso, continuaremos utilizando Herança para criar uma **Hierarquia de Exception**. O principal objetivo do laboratório é entender/aplicar o padrão de projeto [Facade](https://pt.wikipedia.org/wiki/Fa%C3%A7ade), [Controller](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)#Controller) e identificar **benefícios e limitações** no uso de Composição com Polimorfismo. Para gerar a **documentação alto-nível** do seu código vamos praticar o uso de **Javadoc**, e teremos duas novidades: a primeira é que estaremos praticando nesse lab a criação de **diagramas de classes** e, a segunda, é que usaremos **testes de aceitação** para avaliar o código implementado.

**Implementação:**

Você foi contratada(o) para implementar o sistema ***P2-CG: Programação 2 - Central de Games***. O P2-CG *armazena e gerencia uma coleção de jogos de um usuário*. Semelhante à plataforma [**Steam**](http://store.steampowered.com/), 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. **Grande parte do sistema já foi desenvolvido: hierarquia de usuários, hierarquia de Jogo e Loja. A loja, no entanto precisa ser ampliada e uma nova forma de premiar jogadores com x2p e puni-los com a diminuição do seu x2p deve ser adicionada na funcionalidade do sistema. Rode os testes de aceitação (crie novos), forneça documentação (javadoc e diagrama de classes) e trate os erros por meio de Exception. Use Herança para fazer um tratamento sofisticado de Exceptions.**

**Testes de aceitação com Easy Accept:**

Neste lab iremos usar **testes de aceitação** para avaliar o código implementado. Teste de aceitação é um tipo de teste que avalia o sistema em relação aos seus requisitos originais, as funcionalidades esperadas e visíveis ao usuário; ajuda a decidir junto ao usuário/cliente se o sistema será “aceito”. Normalmente realizado com o software pronto. Veja um pouco mais sobre testes de aceitação com **Easy Accept** [**aqui**](https://docs.google.com/presentation/d/17a-fn6DM_eXpJ7mTpBWhqW-2rjW78lC1NoezPWLoB20/edit#slide=id.p) (apresentação dos monitores de p2/lp2 2016.1).

Prepare o seu projeto para usar o Easy Accept ([easyaccept.jar](https://drive.google.com/file/d/0B8n9er4GBTlxUEpFZ1V0Zjdoa0U/view?usp=sharing)), [baixe os testes de aceitação para esse lab](https://drive.google.com/folderview?id=0B__VKu7WQvVYNmFQejZkZm4xcVk&usp=sharing) e certifique-se que seu código está de acordo com o testes, ou seja, se todos os testes terão sucesso. (haverá penalização para os testes que falharem).

**Passo 1: A LojaController**

O objetivo dos passos 1 e 2 e do próximo é um exercício do **GRASP** (atribuição de responsabilidades). É comum que em design OO desacoplar a interface gráfica das classes que fazem parte do domínio do problema. Uma forma simplificada de fazer isso é usar dois padrões: controller e façade. Um controller é um objeto que irá conhecer algumas classes do domínio do problema e delegar tarefas para as classes adequadas. Por enquanto, de forma simplificada, teremos um único controller, que chamamos LojaController. Esse controller deve possuir a coleção de usuários e ser capaz de criar jogos, além de delegar para a classe correta uma série de responsabilidades.Em resumo teremos:

* **Façade:** Delegar (***forwarding***) as chamadas da View (camada do *user[[1]](#footnote-0)* do sistema, interface gráfica) para o Controller. Já que não teremos uma camada de Interface Gráfica, a captura de Exceptions e impressões na console 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** (fábricas de objetos)de seu projeto.

Veja a seguir o que deve existir na classe LojaController. (Na verdade você vai renomear o código da Loja para LojaController e modificar o código de acordo com o que vemos a seguir)

* Vender jogos (jogos são associados a usuários - parcialmente desenvolvido no lab anterior). Como entrada do método **você não pode receber o objeto jogo** a ser vendido ou o usuário que vai comprar o jogo. Você vai receber o login do ususário (e assim poder recuperar o usuário) e as informações do jogo a ser vendido, que **deve ser criado antes de ser associado** ao usuário.
* Adicionar dinheiro na conta de um usuário (parcialmente desenvolvido no lab anteiror). Mais uma vez **você não pode receber um objeto Usuario**, mas o login do usuário).
* Cadastrar (criar) novos usuários na loja (é uma melhoria do adicionar usuário que tínhamos no lab anterior). No lab anterior você desenvolveu a funcionalidade de adicionar usuários. (**Dica:** pense numa forma **desacoplada e coesa** de gerenciar essa funcionalidade… pense em ter uma **classe cuja única responsabilidade é criar os diferentes tipos de usuários.** Tem um exemplo bem simples [aqui](http://www.devmedia.com.br/padrao-de-projeto-factory-method-em-java/26348). Chamamos essa classe de **FactoryDeUsuario** - fábrica de usuários - e ela será um atributo de LojaController. A loja deve ser capaz de criar Usuários Noob. O método da classe FactoryDeUsuario que deve retornar a respectiva instância criada de Usuario Noob vai se chamar criaUsuario(). O usuário noob deve ser criado com uma lista de jogos vazia, e com valores de quantia de dinheiro, x2p e score máximo zerados. Apesar do usuário ser criado sempre como Noob, este método deve declarar o retorno como sendo do tipo Usuario. Assim o código fica preparado para, em um futuro próximo, ser aumentado para criar outros tipos de usuários também. Na loja um método que delega a criação do usuário ao Factory vai existir.
  + Use Strings na LojaController para passar as informações do usuário a ser criado. Mais uma vez, para manter o padrão, o método deve receber também o tipo de usuário a ser criado (apesar de por enquanto apenas "Noob" ser aceito). Exemplo: lojacontroller.criaUsuario("Raquel Lopes", "quellogs", "Noob"); Essa responsabilidade deve ser delegada ao objeto FactoryDeUsuario que é um atributo da loja. Então note que você terá dois métodos chamados criaUsuario, um em loja e um na FactoryDeUsuario.
* Agora você precisa ser capaz de Criar Jogos. (**Dica:** novamente, reusamos a mesma solução Factory para chegar a uma forma **desacoplada e coesa** de gerenciar a criação dos diferentes tipos de jogos. Escreva uma **classe cuja única responsabilidade é criar os diferentes tipos de jogos.** Como no item anterior, essa classe é uma **FactoryDeJogo** - fábrica de jogos - e um objeto dessa classe será um atributo de LojaController. Note que a loja tem 2 atributos factory: a FactoryDeUsuario e FactoryDeJogo. A FactoryDeJogo possuirá apenas o método público criaJogo e internamente os métodos criarJogoRPG, criarJogoLuta, criarJogoPlataforma, retornando a respectiva instância criada de Jogo.) O método criaJogo da FactoryDeJogo deve declarar Jogo como o tipo de retorno. Mas na prática o jogo retornado sera de um dos 3 tipos possíveis. Lembre-se que usuários não podem ter jogos repetidos, isto é, com com nome repetidos.
  + Na loja, mais uma vez, você vai criar um método chamado criaJogo que vai delegar ao FactoryDeJogo a responsabilidade de criar o jogo. Use Strings na LojaController para definir o tipo de jogo a ser criado. Exemplo: lojacontroller.criaJogo("Super Mario", 40.00, "Plataforma", listaEstilo); Atenção: essa responsabilidade deve ser delegada ao objeto FactoryDeJogos que é um atributo da loja. Então note que você terá dois métodos chamados criaJogo, um em loja e um na FactoryDeJogos.
* Imprimir as Informações de todos os Usuários e seus respectivos jogos (toString()). Na impressão de usuários não precisa levar em consideração o desconto dos jogos. Lembre de **imprimir** também a quantidade de **x2p** de cada usuário ao imprimir as informações dos usuários da loja. Use a seguinte formatação:

|  |
| --- |
| === Central P2-CG ===  Jogador Noob: francisco.neto Francisco Oliveira Neto - 300 x2p Lista de Jogos: + Magicka - RPG: ==> Jogou 5 vez(es) ==> Zerou 0 vez(es) ==> Maior score: 65478  Total de preço dos jogos: R$ 25,00  -------------------------------------------- |

**Passo 2: LojaFacade**

Em conjunto com o controller temos uma classe que chamamos de Façade (ou fachada). **Essa classe não conhece as classes do domínio do problema** (Usuario, Jogo, etc.). A LojaFacade tem *apenas um objeto do tipo LojaController*, e delega a este objeto responsabilidades. A interface da LojaFacade deve ser correspondente à interface da LojaController, isto é, expor métodos semelhantes.

**O que ganhamos com essas duas classes?** A interface gráfica conhece apenas a LojaFacade, que conhece os controllers (nesse caso, apenas 1 controller, LojaController). Se as classes do domínio do problema mudarem a fachada não precisa mudar, apenas o controller precisa mudar, dessa forma desacoplamos o código da interface gráfica do resto do código do programa. Nesse passo você deve criar a LojaFacade que tem um objeto do tipo LojaController como atributo e que expõe os mesmos métodos públicos de LojaController, delegando as tarefas ao objeto lojaController que tem como atributo.

**Passo 3: Pontos Extras pela Jogabilidade**

Depois de várias **reclamações** de usuários jogando como Noobs e dando *‘ragequit’* nas partidas de alguns jogos, você foi chamada para **melhorar o sistema de pontuação do P2-CG**. Agora o tipo de usuário vai **ganhar ou perder** pontuação extra de acordo com o tipo do usuário jogando e a respectiva jogabilidade do jogo. Substitua o método registraJogada(...) em Usuario pelos seguintes métodos:

* **recompensar(String nomeJogo,int scoreObtido,boolean zerou)**: Indica que um usuário teve um bom desempenho no jogo especificado. O objetivo é **recompensar** o usuário com x2p.
* **punir(String nomeJogo, int scoreObtido, boolean zerou):** Semelhante ao método anterior porém o comportamento desse método irá **diminuir** os x2p do usuário.

**O comportamento de punir/recompensar muda de acordo com o tipo do usuário.** Portanto, fique atenta(o) para o polimorfismo nas subclasses e na superclasse. Para o usuário **Noob** considere as seguintes estratégias:

* **Punir:**
  + -10 pontos se o jogo for **Online**.
  + -20 pontos se o jogo for **Competitivo**.
  + -50 pontos se o jogo for **Cooperativo** (ninguém merece Noob ‘*feedando*’).
* **Recompensar:**
  + +30 pontos se o jogo for **Offline** (é melhor praticar offline primeiro).
  + +10 pontos se o jogo for **Multiplayer**.

Para o usuário **Veterano** considere as seguintes estratégias:

* **Punir:**
  + -20 pontos se o jogo for **Competitivo**.
  + -20 pontos se o jogo for **Offline**.
* **Recompensar:**
  + +10 pontos se o jogo for **Online**.
  + +20 pontos se o jogo for **Cooperativo**.

**Note que um jogo pode possuir mais de um desses estilos.** Se for o caso, os pontos devem ser somados/subtraídos de forma **cumulativa**. Então, se um Usuário for punido em um Jogo Online e Cooperativo, então o total de pontos perdido é 60 (-10 - 50). Da mesma forma, se um Veterano for recompensado em um Jogo Online e Multiplayer ele ganha apenas os 10 pontos do Online.

Note também que esses dois métodos (punir/recompensar) estão ligados com o método **registraJogada(...)** da classe **Jogo** que também fornece pontos extras. Fique atenta(o) para **manter a integridade** no cálculo de seus x2p e de armazená-los no local correto.

**Passo 4: Implementação do padrão Strategy para permitir mudança de tipo de usuário**

Assim como usuários Noob são recompensados ao acumular pontos (upgrade), é preciso **punir aqueles Veteranos (downgrade)** que não estão à altura de seu título. Um usuário Veterano poderá, a partir de agora passar para Noob de acordo com a quantidade de pontos atingida. **O limiar de pontos para downgrade é: 1000 x2p**. Então se um usuário Veterano perder pontos e ficar aquém da meta de 1000 x2p (ou seja, seus pontos são **menores ou iguais** a 1000 x2p), ela(e) será rebaixada(o) para um usuário do tipo Noob. No lab anterior a funcionalidade oposta foi implementada: se uma usuária(o) Noob atingir a meta de 1000 x2p, isto é, se tiver x2p maior ou igual a 1000, ela(e) é promovida(o) a Veterano. No lab anterior, essa transformação **dinâmica**, foi feita de forma manual, onde um novo usuário Veterano seria criado e inicializado com os mesmos dados do usuário Noob sendo promovido.

**Neste lab iremos usar o padrão Strategy** para realizar essa mudança dinâmica. Você deve ter percebido que mudar o tipo do objeto em tempo de execução da forma realizada foi bem estranho. Deve existir algo melhor. Sim, existe, o padrão de projeto Strategy. Leia sobre o padrão de projeto [Strategy](https://pt.wikipedia.org/wiki/Strategy) aqui.

O usuário precisa mudar de tipo durante a sua vida, mas não é bonito mudar a classe de um objeto importante durante sua vida, em tempo de execução. Como lidar com isso? A dica é: **separe o que é igual daquilo que muda**. Olhe para a sua hierarquia de classes e identifique o que é igual para todos os usuários e o que muda se o usuário for de um tipo ou de outro. Depois use interfaces para definir o comportamento que muda e encapsule os diferentes comportamentos em objetos de classes diferentes que implementam a mesma interface.

**Crie uma interface** chamada, por exemplo, TipoDeUsuarioIF, que deve expor o comportamento que muda de um usuário para o outro. Basicamente três métodos estarão ali: **comprar** jogo (já que usuários diferentes recebem quantidades diferentes de x2p ao comprar jogos), **punir** e **recompensar** (já que usuários diferentes são punidos/recompensados de forma diferente).

Em seguida, você deve **escrever duas classes que implementam a interface criada**: a classe Noob e a classe Veterano. Na classe Noob o comportamento dos métodos deve seguir o que foi especificado para usuários Noob, enquanto na classe Veterano o comportamento é como especificado para usuários veteranos.

Finalmente, você deve **modificar a classe Usuario**. Para termos a troca dinâmica pelo Strategy, precisamos **criar uma Composição com o Polimorfismo criado** (Interface).

Esta classe *não mais será abstrata*. Ela deve ter um **atributo** chamado, por exemplo, statusDoUsuario, que é do tipo TipoDeUsuarioIF. Esse atributo pode ser do tipo Noob ou Veterano, dependendo do status atual do usuário em questão. Implemente/modifique os métodos comprar, recompensar e punir de forma que eles **delegam ao objeto statusDoUsuario** a responsabilidade de comprar, recompensar e punir. Dependendo do tipo do objeto esses métodos delegados irão se comportar de forma diferente. Além disso, você deve atualizar a classe usuário (ou alguma outra classe dependendo do seu design) para que **o downgrade e upgrade sejam realizados de forma automática** sempre que a quantidade de x2p diminuir para um valor inferior a 1000 ou aumentar para um valor superior a 1000, respectivamente. No **downgrade** deve ocorrer a troca do tipo associado ao atributo statusDoUsuario que antes deveria ser Veterano e deve passar a ser Noob. No **upgrade** deve ocorrer a troca do tipo associado ao atributo statusDoUsuario que antes deveria ser Noob e deve passar a ser Veterano. Essa funcionalidade é fundamental para que o padrão Strategy funcione, pois é ela que muda em tempo de execução o status do usuário.

**Passo 5: Diagrama UML de classes**

Está na hora de começar a **documentar seu projeto OO por meio de um diagrama.** Nesta disciplina usaremos o [diagrama de classes UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language). É uma linguagem utilizada para representar **classes**, seus respectivos **componentes** e os **diversos relacionamentos** (tipos polimórficos, dependências, acoplamentos, etc.) A princípio vamos aprender o básico da representação como alguns relacionamentos (Herança), a definição dos membros de classes e suas respectivas visibilidades.

**Junto com o projeto você entregará um diagrama.** Existem várias ferramentas para a criação de Diagramas. Uma delas é a [**Astah**](http://astah.net/editions/community)que é simples de usar.

**Passo 7: Marcação de chamadas polimórficas**

O seu código está cheio de chamadas polimórficas. Marque as chamadas polimórficas com o comentário "//chamada polimórfica".

**Considerações importantes para o seu projeto:**

* Escreva o **Javadoc** para os seus métodos. Procure ser objetivo e comunicativo. 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 (extra).** Lembre-se que teremos testes de aceitação com Easy Accept para a LojaFacade, então, você pode fazer testes de unidade para as demais classes do projeto. **Nesse lab iremos recompensar como extra a implementação de testes de unidade**.
* Faça a implementação do Lab em um **projeto no Eclipse.** Nomeie o seu projeto da seguinte forma: **Matricula\_PrimeiroNome\_Lab07**. Por exemplo:

**114210216\_Gerson\_Lab07**

**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.

## Como o seu lab será avaliado:

Serão considerados os seguintes critérios com suas respectivas notas:

* (6.0) Funcionalidade e os tipos de dados associados: observar se foram implementadas as funcionalidades pedidas, associadas aos seus respectivos tipos de dados.
  + (1.0) Façade
  + (1.0) Controller e Factories
  + (2.0) Punir e Recompensar
  + (2.0) Troca dinâmica de tipo de usuário (strategy)
* (1.0) Testes de aceitação: testes com sucesso.
* (1.0) Identificação de chamadas polimórficas
* (1.0) Javadoc e Diagrama UML
* (0.5) Uso especializado de exceções: procure usar suas próprias exceções
* (0.5) Legibilidade: inclui organização do código, estilo e lógica clara
* **(extra 0.5)** Testes de unidade: observar a cobertura e qualidade dos testes de unidade (tem testes para todas as classes e esses testes cobrem condições normais e limites de uso dos objetos)

***Boa sorte e boa implementação!***

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. [↑](#footnote-ref-0)