Trabalho 2 - UML Design Patterns

Introdução Teórica

Padrões de Design são soluções testadas e aprovadas para problemas comuns de design de software orientado a objetos. Eles fornecem um vocabulário comum para desenvolvedores, facilitando a comunicação, a documentação e a manutenção de código. Ao invés de reinventar a roda, você pode utilizar Padrões de Design para construir software mais robusto, flexível e fácil de entender.

1. Padrões Criacionais:

Os Padrões Criacionais lidam com a criação de objetos, encapsulando a lógica de instanciação e fornecendo flexibilidade e desacoplamento.

1. **Factory Method:** Define uma interface para criar objetos, mas permite que as subclasses decidam qual classe instanciar. Ideal para famílias de produtos relacionados

2. **Abstract Factory:** Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Útil para plataformas cruzadas ou com múltiplas interfaces visuais.

3. **Singleton:** Garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela. Ideal para gerenciar recursos compartilhados como logs ou conexões de banco de dados.

4. **Builder:** Permite a construção de um objeto complexo passo-a-passo, através de uma interface fluente. Ideal para objetos com muitos atributos opcionais.

5. **Prototype:** Permite a criação de novos objetos a partir da cópia de objetos existentes (protótipos), evitando a necessidade de subclasses e construtores complexos. Útil para objetos custosos de criar.

2. Padrões Estruturais:

Os Padrões Estruturais se concentram na composição de classes e objetos, promovendo a reutilização de código e a flexibilidade.

1. **Adapter:** Permite que classes com interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra esperada pelos clientes. Útil para integrar sistemas legados ou APIs de terceiros.

2. **Bridge:** Desacopla uma abstração de sua implementação, permitindo que ambas variem independentemente. Ideal para plataformas com múltiplas implementações.

3. **Composite:** Compõe objetos em estruturas de árvore para representar hierarquias parte-todo. Permite que clientes tratem objetos individuais e composições de forma uniforme.

4. **Decorator:** Adiciona dinamicamente responsabilidades a um objeto, fornecendo uma alternativa flexível à herança. Útil para customização em tempo de execução.

5. **Facade:** Fornece uma interface simplificada para um subsistema complexo, ocultando sua complexidade e tornando-o mais fácil de usar.

6. **Flyweight:** Compartilhar objetos de granularidade fina para suportar a criação eficiente de um grande número de objetos. Útil para otimização de performance em aplicações com alta demanda por objetos.

7. **Mediator:** Define um objeto que encapsula como um conjunto de objetos interagem, promovendo o baixo acoplamento. Ideal para sistemas com alta interdependência entre objetos.

8. **Proxy:** Fornece um substituto ou espaço reservado para outro objeto, controlando o acesso ao objeto real. Útil para controle de acesso, carregamento lento ou operações remotas.

3. Padrões Comportamentais:

Os Padrões Comportamentais lidam com a interação entre objetos, definindo como eles se comunicam e colaboram para realizar tarefas complexas.

1. **Chain of Responsibility:** Evita o acoplamento do remetente de uma solicitação ao seu receptor, dando a múltiplos objetos a chance de tratar a solicitação. A solicitação é passada por uma cadeia de objetos até que um a trate.

2. **Command:** Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar solicitações e suportar operações que podem ser desfeitas.

3. **Iterator:** Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação interna.

4. **Memento:** Captura e externaliza o estado interno de um objeto, sem violar o encapsulamento, para que o objeto possa ser restaurado a este estado posteriormente.

5. **Observer:** Define uma dependência um-para-muitos entre objetos, de forma que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. Útil para implementar notificações e eventos.

6. **State:** Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá mudar de classe. Ideal para modelar máquinas de estado.

7. **Strategy:** Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.

8. **Template Method:** Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. Permite que subclasses redefinam certos passos de um algoritmo sem alterar sua estrutura.

Chain of Responsibility

Problemas:

1. Como posso evitar o acoplamento entre o remetente de uma solicitação e seu receptor?
2. Como permitir que mais de um objeto possa atender alguma requisição?

Solução:

1. Definir uma cadeia de objetos onde cada um poderá responder àquela solicitação ou enviar para o próximo objeto tratá-la.
2. Quem faz a solicitação não precisa saber o tamanho da cadeia, ou mesmo como (por quem) ela será resolvida.

Segundo Gamma et al. (1995, p. 223):

> *"Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate."*

Em outras palavras, o padrão de design ***Chain of Responsibility*** é um padrão comportamental que permite passar uma requisição por uma cadeia de objetos processadores até que um deles a trate, ou até que a cadeia termine. Isso evita o acoplamento do remetente da requisição ao receptor, pois o remetente não precisa saber qual objeto irá processá-la.

Componentes do padrão:

1. **Handler:** Define a interface comum para todos os objetos que podem tratar a requisição. Essa interface geralmente inclui um método para processar a requisição e outro para definir o próximo objeto na cadeia (se houver).

2. **ConcreteHandlers:** São implementações concretas da interface Handler. Cada ConcreteHandler é responsável por lidar com um tipo específico de requisição ou por realizar uma etapa específica do processamento.

3. **Client:** Cria e configura a cadeia de objetos Handler, definindo a ordem em que eles devem processar a requisição. O Client então, envia a requisição para o primeiro Handler na cadeia.

Problema: Seu desafio é construir uma API RestFul em Java para gerenciar agendamentos de consultas em uma clínica particular. Sua responsabilidade principal é modelar e implementar o sistema de agendamento, que deve atender a 6 regras de validação definidas pelo cliente.

Solução: Para agilizar o processo de aprovação e garantir a qualidade da solução, vamos utilizar o padrão Chain of Responsibility (CoR) em uma versão inicial com apenas duas regras de validação. Após a aprovação do cliente, a implementação completa com as demais validações será realizada.

Diagrama UML 'Sistema de Agendamento de Consultas' (simplificado)

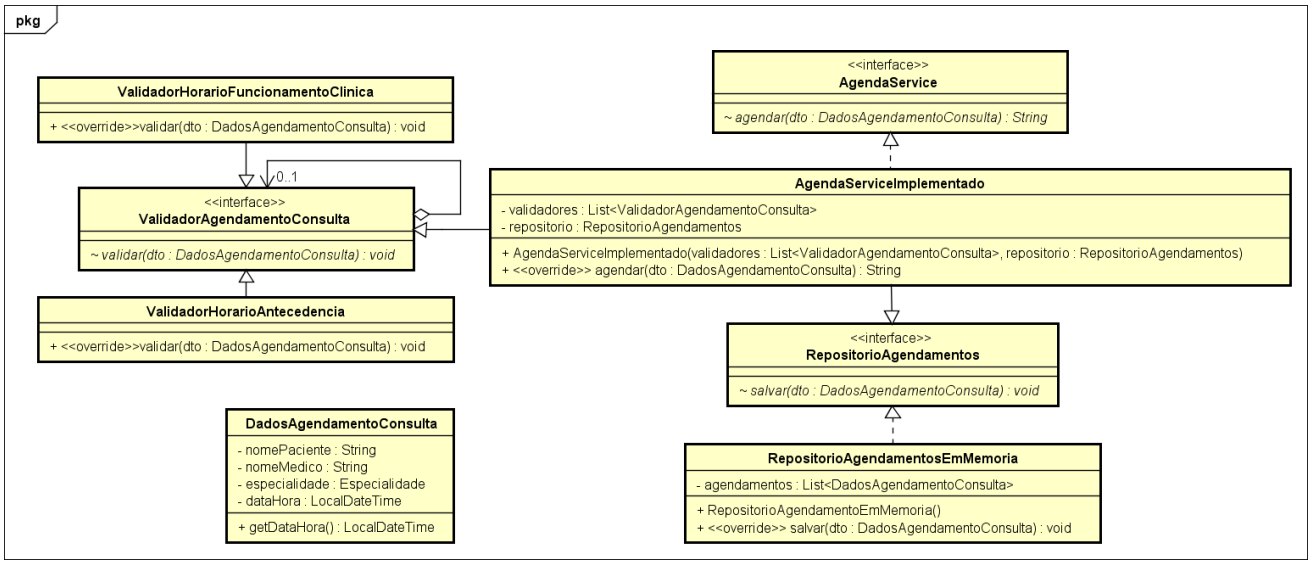
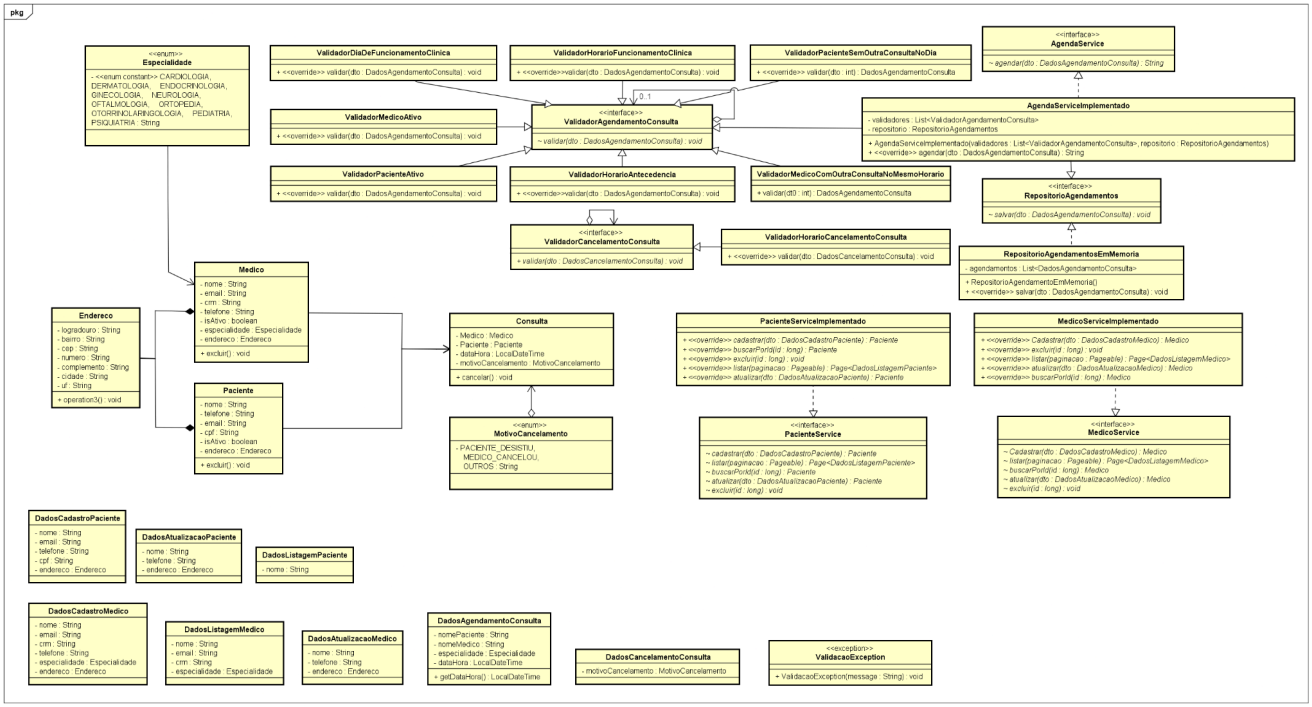


Diagrama UML 'Sistema de Agendamento de Consultas' (Completo)



Vamos analisar a versão simplificada da aplicação 'Sistema de Agendamento de Consultas' e entender como o padrão '***Chain of Responsibility***' resolveu nosso problema.

Observe o código abaixo:

```
public interface ValidadorAgendamentoConsulta {
   void validar(DadosAgendamentoConsulta dados);
}
```

Dentre os componentes do padrão ***Chain of Responsibility***, a interface '`ValidadorAgendamentoConsulta`' representa o ***Handler.***

A Interface '`ValidadorAgendamentoConsulta`' define um contrato '`validar`' sem retorno (void) que recebe como parâmetro '`DadosAgendamentoConsulta`'.

Esse contrato pode ser implementado em outras classes concretas. Como no dois exemplos abaixo:

```
public class ValidadorHorarioAntecedencia implements ValidadorAgendamentoConsulta {

   @Override
   public void validar(DadosAgendamentoConsulta dados) {
       LocalDateTime dataConsulta = dados.dataHora();
       LocalDateTime agora = LocalDateTime.now();
       long diferencaEmMinutos = Duration.between(agora, dataConsulta).toMinutes();

       if (diferencaEmMinutos < 30) {
           throw new ValidacaoException("A consulta deve ser marcada com pelo menos 30 minutos de
antecedência.");
       }
   }
}
```

```
public class ValidadorHorarioFuncionamentoClinica implements ValidadorAgendamentoConsulta {

   private static final LocalTime HORA_ABERTURA = LocalTime.of(8, 0);
   private static final LocalTime HORA_FECHAMENTO = LocalTime.of(18, 0);

   @Override
   public void validar(DadosAgendamentoConsulta dados) throws ValidacaoException {
       LocalDateTime dataHoraConsulta = dados.getDataHora();

       if (dataHoraConsulta.toLocalTime().isBefore(HORA_ABERTURA) ||
               dataHoraConsulta.toLocalTime().isAfter(HORA_FECHAMENTO)) {
           throw new ValidacaoException("A clínica não funciona neste horário.");
     }
   }
}
```

Dentre os componentes do padrão ***Chain of Responsibility***, as classes concretas que implementam (assinam) o contrato da interface '`ValidadorAgendamentoConsulta` ' são chamados de ***ConcreteHandlers***

Nesse dois exemplos, temos as classes concretas '`ValidadorHorarioAntecedencia`' e '`ValidadorHorarioFuncionamentoClinica`'

Essas classes implementam a interface '`ValidadorAgendamentoConsulta`', portanto assinam o seu contrato 'validar'. Podemos observar que, embora o contrato seja o mesmo, a maneira como ele é implementado é diferente entre as duas classes.

A classe `ValidadorHorarioAntecedencia` é responsável por validar se uma determinada consulta passada como parâmetro respeita a regra de ser agendada com no mínimo 30 minutos de antecedência. Para realizar esse cálculo, o método utiliza o `LocalDateTime.now()` que captura o instante atual e compara com o tempo do agendamento que foi passado como parâmetro. Caso ele seja inferior a 30 minutos, é lançada uma exceção personalizada indicando que o agendamento é invalidado, portanto não foi concluído. Caso contrário não é lançada nenhuma exceção e o código continua.

A classe `ValidadorHorarioFuncionamentoClinica` é responsável por validar se uma determinada consulta passada como parâmetro respeita a regra de ser agendada no horário de funcionamento do hospital/clínica. O intervalo de horário permitido é entre as 8h00 da manhã e 18h00 da tarde. Qualquer agendamento realizado fora desse intervalo implica no lançamento de uma exceção personalizada, portanto não foi concluído. Caso contrário, não é lançada nenhuma exceção e o código continua.

Dentre os componentes do padrão ***Chain of Responsibility***, as classe concreta '`AgendaServiceImplementado`' representa o **Client**

```
public class AgendaServiceImplementado implements AgendaService {

   private final List<ValidadorAgendamentoConsulta> validadores;
   private final RepositorioAgendamentos repositorio;

    public AgendaServiceImplementado(
        List<ValidadorAgendamentoConsulta> validadores,
        RepositorioAgendamentos repositorio
    )   {
          this.validadores = validadores;
          this.repositorio = repositorio;
    }

   @Override
   public String agendar(DadosAgendamentoConsulta dados) throws ValidacaoException {
       validadores.forEach(validador → validador.validar(dados));
       repositorio.salvar(dados);
       return "Agendamento realizado com sucesso!";
   }
}

public interface RepositorioAgendamentos {
   void salvar(DadosAgendamentoConsulta dados);
}
```

```java
public class RepositorioAgendamentosEmMemoria implements RepositorioAgendamentos {
   private final List<DadosAgendamentoConsulta> agendamentos = new ArrayList<>();

   @Override
   public void salvar(DadosAgendamentoConsulta dados) {
       agendamentos.add(dados);
   }
}
```

A classe acima é uma espécie de gerenciador. No padrão arquitetural RestFul, costumamos dizer que essa é uma classe de serviço. Ela recebe a injeção de dependência dos validadores e do repositório de agendamentos através do construtor. Isso permite que a classe `AgendaServiceImplementado` seja flexível e extensível, pois ela não está acoplada a implementações específicas de validadores e repositórios.

A principal responsabilidade dessa classe é orquestrar o processo de agendamento de consultas. Ela faz isso iterando sobre a lista de validadores e aplicando cada um deles aos dados de agendamento fornecidos. Se algum validador lançar uma `ValidacaoException`, o processo de agendamento é interrompido e a exceção é propagada para cima. Se todas as validações forem bem-sucedidas, a classe `AgendaServiceImplementado` salva os dados de agendamento no repositório ‘`RepositorioAgendamentos`‘ que é responsável por salvar a string contendo o status do processamento da operação com a mensagem: `“Agendamento realizado com sucesso!”.` Os casos negativos podem ser:

1. `A consulta deve ser marcada com pelo menos 30 minutos de antecedência.`
2. `A clínica não funciona neste horário.`

Em resumo, da maneira que a aplicação foi modelada, fica muito simples acrescentar, modificar ou remover os validadores. isso ocorre pois a classe `AgendaServiceImplementado` foi projetada seguindo o princípio da Inversão de Controle (IoC) e Injeção de Dependência (DI). Ao receber os validadores através do construtor, a classe se torna altamente flexível e desacoplada de implementações específicas.

Vantagens do Padrão Chain of Responsibility:

1. **Facilidade de manutenção:** Novos validadores podem ser adicionados, existentes podem ser modificados ou removidos sem a necessidade de alterar o código da classe `AgendaServiceImplementado` . Basta ajustar a configuração da injeção de dependência.

2. **Código mais limpo e coeso:** A classe `AgendaServiceImplementado` foca em sua responsabilidade principal, que é orquestrar o processo de agendamento, delegando a validação para classes especializadas.

3. **Maior testabilidade:** Ao utilizar interfaces para os validadores, é possível criar mocks e stubs para facilitar a escrita de testes unitários específicos e direcionados.

4. **Extensibilidade:** A adição de novos validadores não requer alterações no código existente, promovendo a escalabilidade da aplicação.

Desvantagens do Padrão Chain of Responsibility

1. **Complexidade:** A adição de muitos validadores pode tornar a cadeia complexa e difícil de gerenciar, especialmente se as regras de validação forem interdependentes.

2. **Depuração:** Em cadeias longas, identificar a causa de uma falha na validação pode ser desafiador, pois a exceção pode ser lançada em qualquer ponto da cadeia.

3. **Performance:** Em cenários com alta demanda, a execução sequencial dos validadores pode impactar o desempenho, especialmente se as validações forem custosas em termos de tempo de processamento.

4. **Tratamento de erros:** A propagação de exceções através da cadeia pode dificultar o tratamento de erros específicos, exigindo mecanismos adicionais para identificar o validador que falhou.

Conclusão:

O padrão Chain of Responsibility (CoR) oferece uma abordagem eficaz para lidar com validações em cadeia, promovendo flexibilidade, manutenibilidade e testabilidade do código. A capacidade de adicionar, modificar ou remover validadores sem alterar a classe principal do serviço é um grande benefício, assim como a possibilidade de focar a responsabilidade de cada classe em uma tarefa específica.

No entanto, é importante estar ciente das desvantagens do CoR. A complexidade pode aumentar com o crescimento da cadeia de validadores, especialmente se as regras forem interdependentes. A depuração pode se tornar desafiadora, pois a origem de um erro pode estar em qualquer ponto da cadeia. Além disso, a performance pode ser afetada em cenários de alta demanda devido à execução sequencial dos validadores.

Portanto, a decisão de utilizar o padrão CoR deve ser baseada em uma análise cuidadosa dos requisitos do sistema. Em aplicações onde a flexibilidade e a manutenibilidade são cruciais e o número de validações é moderado, o CoR pode ser uma excelente escolha. No entanto, em sistemas complexos com alta demanda e validações custosas, é importante considerar as desvantagens e avaliar se o padrão é a melhor solução para o problema em questão.

Proxy

Problemas:

1. Como controlar o acesso a objetos e garantir a sua utilização apenas para usuários autorizados?

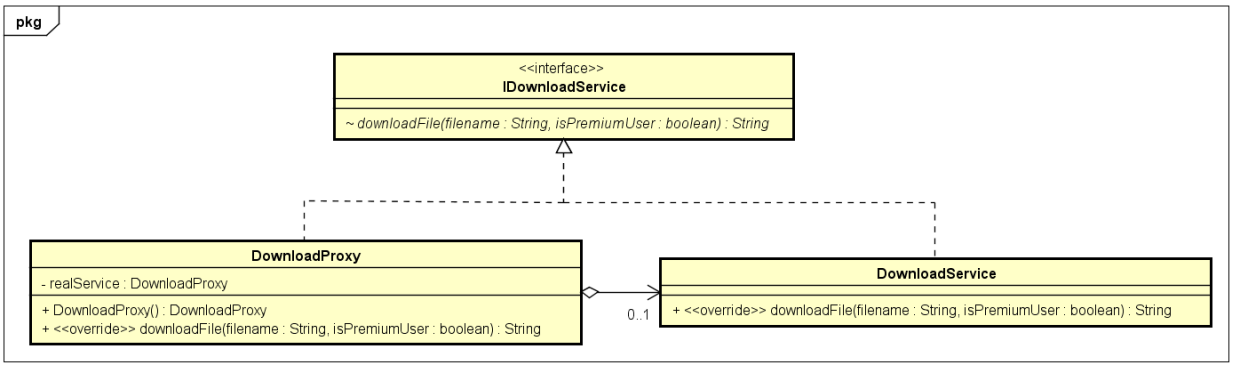
2. Como tornar o código mais flexível, permitindo modificações e testes sem que afete o sistema do cliente?

Soluções:

1. O Proxy consegue controlar restritamente o acesso a objetos, por meio de mecanismos de autenticação e autorização nos quais as regras de acesso são definidas detalhadamente, aumentando a segurança do sistema.

2. Isolando o objeto, o Proxy, cria algo como um substituto, assim permitindo que o desenvolvedor possa alterar linhas de código do objeto sem que atrapalhe o seu funcionamento. Também, por meio do encapsulamento mostra uma interface mais simples, escondendo a complexidade da implementação e permitindo que a interface do cliente se mantenha inalterada.

Diagrama UML Padrão Proxy



O padrão Proxy faz parte da categoria de padrões estruturais, alterando como objetos e classes são estruturados e manipulados. Sua principal característica está em atuar como um mediador em sistemas, atuando como um substituto do objeto em questão.

Diversas aplicações práticas são possíveis utilizando este padrão, como escalar um sistema de forma mais flexível, diminuir a complexidade na manipulação de objetos por meio de implementações, trazer uma maior segurança e autenticação como dito anteriormente, armazenar um cache local temporariamente para dados frequentemente acessados, entre outros.

Singleton

Problemas:

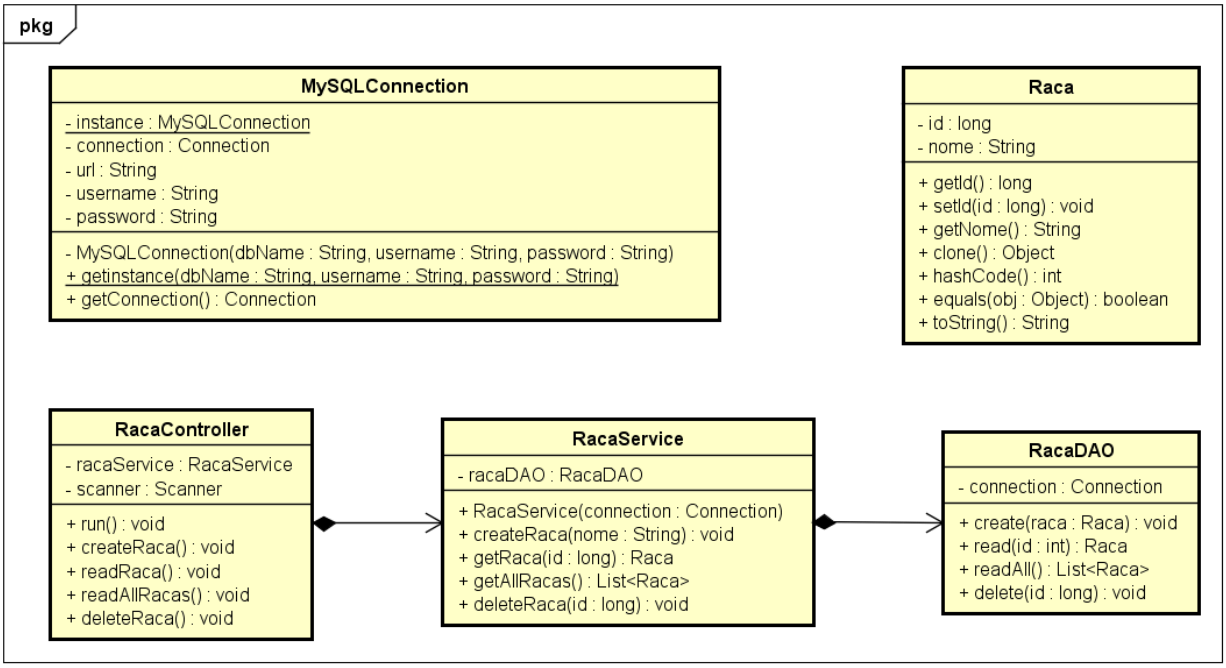
1. Como garantir que uma classe tenha apenas uma única instância em todo sistema, prevenindo a criação acidental de múltiplas instâncias?

2. Como fornecer um ponto global de acesso a essa instância única, garantindo que todos os componentes do sistema utilizem a mesma instância?

Soluções:

1. O padrão Singleton garante que uma classe tenha apenas uma única instância ao assegurar que, uma vez criada, a instância será reutilizada em vez de ser criada novamente. Isso é feito controlando a criação do objeto dentro da própria classe.

2. O Singleton fornece um ponto de acesso global à sua instância, permitindo que outros componentes do sistema possam acessar e utilizar essa instância de maneira consistente. Isso é útil para recursos que devem ser compartilhados globalmente, como configurações de aplicativos, gerenciadores de log, ou conexões de banco de dados.



O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

O Singleton é utilizável quando uma classe em seu programa deve ter apenas uma instância disponível para todos seus clientes, por exemplo, um objeto de base de dados único compartilhado por diferentes partes do programa. Desabilitando todos os outros meios de criar objetos de uma classe exceto pelo método especial de criação. Esse método tanto cria um novo objeto ou retorna um objeto existente se ele já tiver sido criado.

Vantagens do Padrão Singleton

1. Você pode ter certeza que uma classe só terá uma única instância;

2. Você ganha um ponto de acesso global para aquela instância;

3. O objeto singleton é inicializado somente quando for pedido pela primeira vez.

Desvantagens do Padrão Singleton

1. Viola o princípio da responsabilidade única. O padrão resolve dois problemas de uma só vez;

2. Pode mascarar um design ruim, por exemplo, quando os componentes do programa sabem muito sobre cada um;

3. O padrão requer tratamento especial em um ambiente multi-thread para que múltiplas threads não possam criar um objeto singleton várias vezes;

4. Pode ser difícil realizar testes unitários do código cliente do Singleton porque muitos frameworks de teste dependem de herança quando produzem objetos simulados. Já que o construtor da classe singleton é privado e sobrescrever métodos estáticos é impossível na maioria das linguagens.

Referências

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos. 1. ed. Porto Alegre: Bookman, 1995.