UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ

Universidade Federal do Ceará - *campus* Quixadá

Projeto Detalhado de Software

Documento de Padrões de Projeto

Meu Bolso - Sistema de Gestão Financeira

Docente:
Paulyne Matthews

Discentes:
Antônio Rewelli de Oliveira, 554047;
Giliardy Alves da Silva, 552752;
Lucas Ferreira Nobre, 554590;
Miqueias Bento da Silva, 553972;
Luis Eduardo Vieira de Oliveira, 552375.

Repositório:
https://github.com/LucasFerreira2004/MeuBolso/tree/pds

Av. José de Freitas Queiroz, 5003, Quixadá - CE, 63902-580
14.09.2024

Conteúdo

1. Introdução

Este documento descreve como aplicamos padrões de projeto e princípios que estudamos durante a disciplina de Projeto Detalhado de Software. Dentre o que aplicamos no projeto estão alguns padrões GoF, como Strategy, Observer e Iterator, além de alguns princípios do SOLID, como a Inversão de Dependência, e boas práticas do GRASP.

1.1 Visão geral do documento

Este documento descreve o **sistema Meu Bolso**, um sistema de gestão financeira que apoia o usuário no controle de suas finanças pessoais. Nele, são apresentadas as características gerais do sistema, a organização do projeto e a forma como foram aplicados padrões de projeto na sua implementação.

A estrutura do documento está organizada da seguinte forma:

- **Seção 2 – Descrição geral do sistema**: visão de alto nível, descrição do projeto e escopo.
- **Seção 3 – Estrutura do projeto**: organização dos pacotes, camadas e componentes.
- **Seção 4 – Aplicação de padrões de projeto**: detalha como foram aplicados padrões como Observer, Strategy, entre outros, para garantir um design flexível e de fácil manutenção.
- **Seção 5 - Diagramação**: adicionamos aqui a modelagem da implementação dos padrões de projeto GoF.
- **Seção 6 - Inversão e Injeção de Dependência**: é feito uma breve descrição de como implementamos esse princípio do SOLID no nosso projeto.

2. Descrição geral do sistema

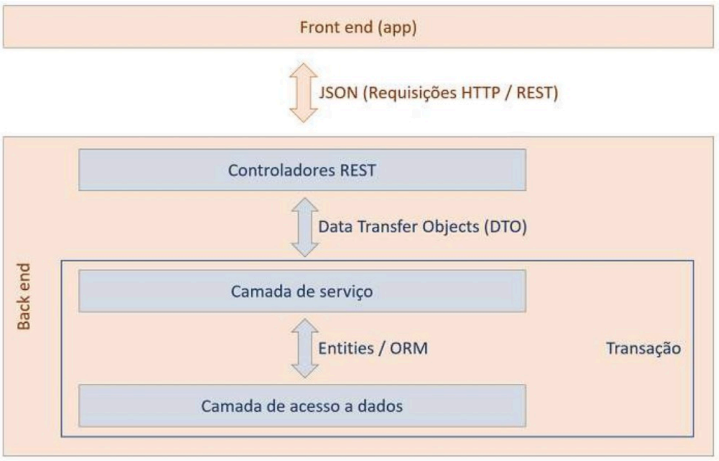
O projeto consiste em uma aplicação desktop para gerenciamento financeiro pessoal. O sistema permitirá que o usuário registre e acompanhe seus ganhos e gastos mensais organizados por categorias, fornecendo também relatórios financeiros e dashboards interativos para facilitar a visualização das finanças. Além disso, incluirá funcionalidades para controle de investimentos pessoais e registro de metas financeiras, como viagens e outros objetivos.

3. Estrutura do projeto

A arquitetura do projeto é organizada por **requisitos** ou **domínios**. Cada pacote (por exemplo, "meta", "transacao", "orcamento") concentra todos os elementos relacionados àquela funcionalidade:

- **Controller**: Gerencia as requisições HTTP.
- **Service**: Implementa as regras de negócio e orquestra as operações.

- **Repository**: Cuida do acesso e persistência dos dados.
- **DTO**: Transfere os dados entre as camadas sem expor as entidades diretamente.
- **Exceções personalizadas**: Tratam erros específicos do domínio.



Essa separação facilita a manutenção, aumenta a coesão e reduz o acoplamento entre os módulos, tornando o projeto mais escalável e organizado.

Também como os membros que trabalharam com o back-end separadamente por requisitos, essa estrutura facilitou o trabalho em conjunto.

4. Aplicação de padrões de projeto

Padrão Observer - Notificação de Metas

1. Contextualização

No projeto **MeuBolso**, há a funcionalidade de **notificar o usuário** sempre que uma meta atinge certos marcos de progresso (ex.: 50%, 90%, 100%). Para evitar o acoplamento direto entre a classe que representa a meta e o mecanismo de notificação, aplicamos o **padrão Observer** do catálogo GoF (Gang of Four). Dessa forma, quando a meta sofre alterações, **observadores** específicos são informados e podem executar ações (como enviar uma mensagem via WebSocket).

2. Objetivo e Benefícios da Implementação

- **Objetivo**:
    - Desacoplar a **lógica de notificação** (ex.: envio de Toast Notification) da **lógica de cálculo** e **estado** da meta.

- Evitar condicionais do tipo “se for 50%, faça X, se for 90%, faça Y…” espalhadas pelo código, concentrando-as em Observadores especializados.

- **Benefícios**:
  - **Baixo Acoplamento**: A meta (Sujeito) não conhece detalhes do serviço de notificação; apenas chama o método de atualização dos Observadores.
  - **Extensibilidade**: Adicionar um novo tipo de notificação (por exemplo, email, SMS) não requer alterar a classe da meta, apenas criar outro Observador.
  - **Separação de Responsabilidades**: A meta cuida do estado (valor investido, progresso), enquanto cada Observador cuida de como notificar o usuário.

3. Implementação no Projeto

3.1. Interface Observer

Criamos uma interface que define um método de atualização, por exemplo atualizar(Meta meta, int threshold).

```
// Interface Observer
public interface MetaObserver {  6 usages  1 implementation  Miqueias Bento
    void atualizar(Meta meta, int threshold);  2 usages  1 implementation  Miqueias Bento
}
    Ctrl⇧L to Chat, Ctrl+I to Command
```

3.2. Sujeito: a Classe da Meta

A entidade que representa a meta possui métodos para calcular o progresso, verificar se um threshold foi atingido e, ao detectar um marco inédito (ex.: 50%), **notifica** os Observadores chamando seu método atualizar(...).

- A classe Meta possui uma lista de observers que serão notificados;
- E o método de interesse na qual ao atualizá-lo, os observers serão notificados de atualização é o método **verificarThresholds();**

3.3. Observador Concreto: Serviço de Notificação

Um exemplo é o **ToastNotificationService**, que implementa a interface MetaObserver. Quando recebe a notificação (atualizar(Meta meta, int threshold)), o serviço envia uma mensagem para o frontend via WebSocket (ou outro canal), exibindo um toast.

3.4. Ligação com o Service de Transações

No TransacaoMetaService, depois de atualizar o valor investido na meta, chamamos meta.verificarThresholds(...) e passamos o Observador (por exemplo, ToastNotificationService) para que seja notificado quando um threshold for atingido.

4. Resultados / Vantagens Conseguidas

1. **Menos Dependências**: A classe Meta não precisa saber se a notificação é por email, toast ou outro meio.
2. **Fácil Inclusão de Novas Notificações**: Basta criar outra implementação de MetaObserver.
3. **Organização e Manutenção**: A lógica de notificação ficou centralizada em serviços especializados, facilitando mudanças.

5. Possíveis Extensões ou Evoluções

- **Novos Observadores**: Incluir notificação por email ou na área de notificação do sistema do usuário sem alterar a classe Meta.
- **Conjunto de Observadores**: Em vez de um único observer, a meta poderia manter uma lista de MetaObserver e notificar todos de uma vez.
- **Configurações de Notificação**: Permitir que cada usuário escolha quais thresholds deseja ser notificado (ex.: apenas 100%). Isso poderia ser configurado em cada Observador.

Padrão Iterator - Atualização de valores de Orçamentos

1. Contextualização

No projeto **MeuBolso**, tivemos a necessidade de iterar sobre coleções de orçamentos para atualizá-los, verificar thresholds de notificação, recalcular valores, etc. Embora o Java já forneça iterações via forEach e for (Orcamento o : orcamentos), optamos por **demonstrar o Padrão Iterator** manualmente para fins didáticos. Assim, criamos uma classe **OrcamentoAgregado** que expõe um Iterator específico, mostrando explicitamente como separar a lógica de iteração do restante do código.

2. Objetivo e Benefícios da Implementação

- **Objetivo**:
  - Demonstrar o padrão GoF **Iterator**, criando uma forma consistente de percorrer coleções de orçamentos sem expor detalhes internos (por exemplo, a lista subjacente).
- **Benefícios**:
  - **Baixo Acoplamento**: A classe que usa o iterator não precisa saber se a coleção é um List, um Set ou outra estrutura.
  - **Uniformidade**: Qualquer outro lugar que precise iterar sobre orçamentos pode reutilizar o mesmo Iterator, mantendo um padrão de navegação.
  - **Demonstração Didática**: Embora o Java ofereça iteradores prontos, esse exercício deixa claro como o padrão Iterator funciona internamente.

3. Implementação no Projeto

3.1. Estrutura do Iterator

Aggregate (Coleção): Criamos uma classe OrcamentoAgregado que armazena uma lista de Orcamento, ele retorna nosso **Iterator** customizado.

Iterator Concreto: A classe OrcamentoIterator implementa Iterator<Orcamento> e controla a posição atual na lista de orçamentos, expondo métodos hasNext() e next().

3.2. Utilização na Atualização de Orçamentos

No método de atualização de orçamentos, em vez de usar forEach, passamos a usar nosso Iterator explicitamente.

4. Resultados / Vantagens Conseguidas

1. **Código mais Didático**: Fica claro como o padrão Iterator separa a lógica de "próximo elemento" da lógica de atualização.
2. **Ocultação de Detalhes**: A classe que atualiza os orçamentos não precisa saber se a coleção interna é um ArrayList, LinkedList ou outra estrutura.
3. **Extensibilidade**: Se no futuro quisermos um iterador diferente (por exemplo, filtrando alguns orçamentos), basta criar outra implementação de Iterator<Orcamento>.

5. Possíveis Extensões ou Evoluções

- **Iteradores Personalizados**: Se quisermos iterar do mais recente ao mais antigo, ou em outra ordem, basta implementar outro Iterator.

Padrão Strategy - Criação de datas fixas/parceladas e avanço de datas

1. Contextualização

No *MeuBolso*, temos a funcionalidade de criar transações normais a partir do cadastro de transações fixas ou parceladas.

Em nosso banco de dados temos uma tabela de **transações recorrentes** que mapeiam registros de transações **fixas** e **parceldas**, pois ambas têm os mesmos atributos, exceto pelos atributos tipo, data_final e qtd_parcelas.

A partir dos registros na tabela de **transações recorrentes** temos algoritmos responsáveis por cadastrar novas **transações** conforme o usuário avança datas em nosso sistema.

As **transações recorrentes** podem ter diferentes periodicidades (DIARIA, SEMANAL, MENSAL, ULTIMO_DIA_MES) que definem de quanto em quanto tempo devem ser criadas transações a partir dela. As **transações recorrentes** ainda podem ser do tipo **fixa** ou **parcelada** sendo que temos um algoritmo diferente de criação de transações para cada um desses tipos.

No futuro desejamos acrescentar mais periodicidades e também mais tipos de transações recorrentes.

Solução: implementamos strategies para os algoritmos de criação a partir de uma transação recorrente e strategies para cada um dos tipos de avanço de datas com base na periodicidade, a fim de evitar o uso de if repetitivos, diminuir o acoplamento, separar as responsabilidades e assim tornar o código mais escalável para mudanças.

2. Objetivo e Benefícios da implementação

Objetivo: A implementação do padrão Strategy tem como objetivo permitir a criação flexível e desacoplada de transações recorrentes a partir da periodicidade e do tipo da transação. Isso evita a necessidade de estruturas condicionais extensas no código e facilita a adição de novos tipos de transações e periodicidades no futuro.

Benefícios:

- **Baixo acoplamento:** Cada tipo de transação e periodicidade é encapsulado em uma classe específica, eliminando dependências diretas entre os componentes.
- **Facilidade de manutenção:** A separação das estratégias facilita a compreensão e manutenção do código.
- **Extensibilidade:** Novos tipos de transações recorrentes ou periodicidades podem ser adicionados sem modificar o código existente, bastando criar uma nova estratégia.
- **Reutilização de código:** Estratégias podem ser reaproveitadas em diferentes contextos, garantindo um comportamento consistente.

3. Implementação no Projeto

Os padrões **Strategy** e **Factory** foram implementados no módulo **repetirTransacoes**, fizemos um diagrama para o módulo que mostra a estrutura dos padrões.

3.1. Strategy GerarTransações

- **Strategy:** IGerarTransações, que possui o método gerarTransacoes
- **Concrete Strategies:** FixasGerarTransacoes, ParceladasGerarTransacoes
- **Context:** TransacaoRepeticaoService

3.2. Strategy GerarTransações

- **Strategy:** IAvancoDataStrategys, que possui o método avancarData
- **Concrete Strategies:** AvancoDiarioStrategy, AvancoSemanalStrategy, AvancoMensalStrategy, AvancoUltimoDiaMesStrategy.
- **Context:** FixasGerarTransacoes e ParceladasGerarTransacoes.

4. Resultados / Vantagens Conseguidas

- **Menos dependências:** O serviço de transações recorrentes não precisa conhecer detalhes de como cada periodicidade ou tipo de transação deve ser gerenciado.

- **Facilidade na inclusão de novas lógicas:** Se no futuro for necessário adicionar novas periodicidades ou tipos de transação recorrente, basta implementar uma nova estratégia sem modificar o código existente.
- **Código mais limpo e modular:** A separação das responsabilidades tornou o código mais organizado e de fácil manutenção.
- **Reutilização de código:** permite que utilizemos as estratégias em outras partes do código de nossa aplicação.

5. Possíveis Extensões ou Evoluções

- Poderíamos adicionar novas periodicidades, como transações trimestrais, bimestrais ou personalizadas pelo usuário.
- Poderíamos adicionar novos tipos de transações recorrentes, como transações com uma quantidade fixa de repetições ou outras características.

Padrão Factory - Retorno de Strategies Concretas

1. Contextualização

A retorno de cada strategy depende de atributos do tipo enum da entidade **TransacaoRecorrente**, sendo esses atributos: TipoRepeticao (FIXO, PARCELAMENTO), para a strategy IGerarTransacoes, e o atributo Periodicidade (DIARIO, SEMANAL, MENSAL, ULTIMO_DIA_MES), para a strategy IAvancoDataStrategy.

Criou-se, portanto, a necessidade de implementar factories para cada uma das strategies para retornar para a classe cliente a strategy correta de acordo com o tipo passado pela classe cliente.

2. Objetivo e Benefícios da Implementação

Objetivo: Facilitar a obtenção da implementação correta das strategies (IGerarTransacoesStrategy e IAvancoDataStrategy), garantindo que o código cliente não precise lidar diretamente com a lógica de decisão sobre qual estratégia utilizar.

Benefícios:

- **Desacoplamento:** O código cliente não precisa conhecer os detalhes das implementações concretas das strategies. Ele apenas solicita a strategy correta à factory.
- **Facilidade de manutenção:** Caso novos tipos de transações ou periodicidades sejam adicionados, basta modificar a factory sem alterar o código cliente.
- **Código mais limpo e organizado:** Evita grandes estruturas condicionais (if ou switch) espalhadas pelo código, centralizando a lógica de decisão em um único ponto.

3. Implementação no Projeto

Os padrõe **Strategy** e **Factory** foram implementados no módulo **repetirTransacoes**, fizemos um diagrama para o módulo que mostra a estrutura dos padrões.

3.1. Factory GerarTransacoes

- **Product:** IGerarTransacoesStrategy
- **Concrete Products:** FixasGerarTransacoesStrategy, ParceladasGerarTransacoesStrategy
- **Factory:** GerarTransacoesFactory

3.2. Factory AvancoData

- **Product:** IAvancoDataStrategy
- **Concrete Products:** AvancoDiarioStrategy, AvancoSemanalStrategy,AvancoMensalStrategy, AvancoUltimoDiaMesStrategy
- **Factory:** AvancoDataFactory.

4. Resultados / Vantagens Conseguidas

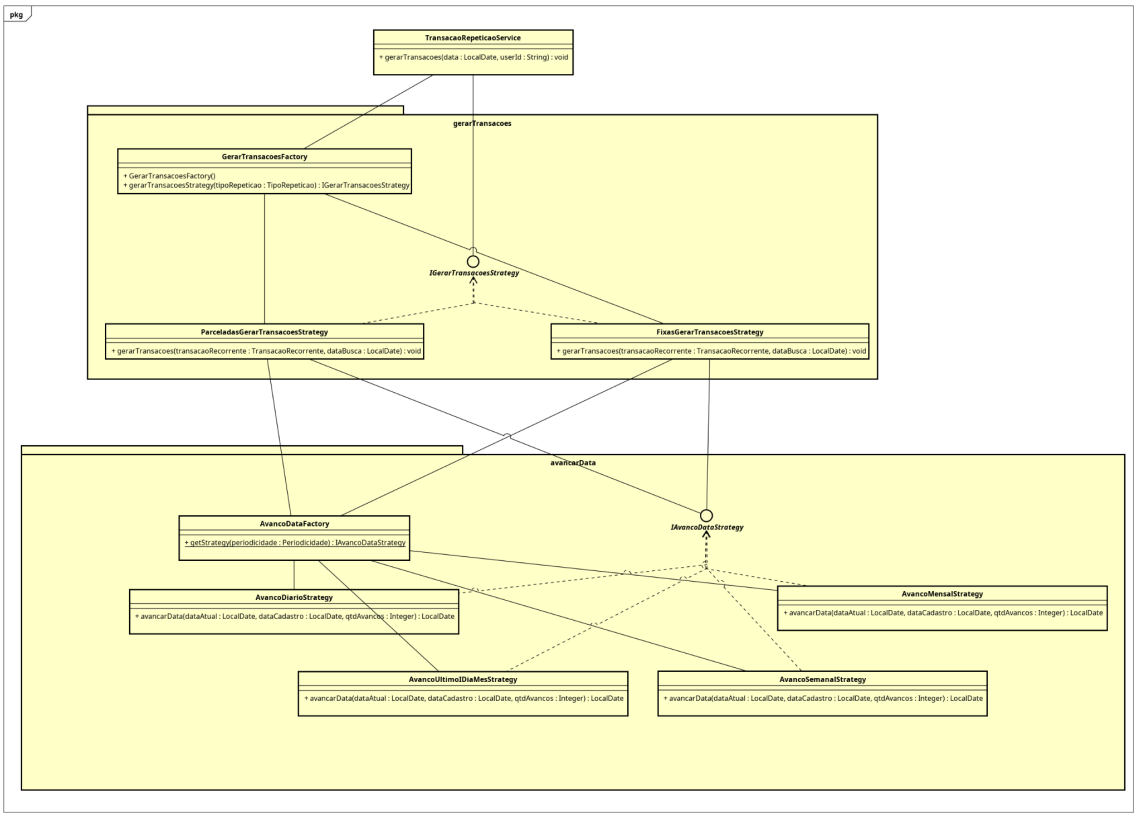
- **Código cliente simplificado:** As classes que precisam de uma strategy não precisam implementar lógica de decisão para saber qual instância usar.
- **Facilidade de adição de novas strategies:** Se surgir um novo tipo de transação ou periodicidade, basta criar uma nova implementação e ajustá-la na factory.
- **Melhor organização do código:** Toda a lógica de decisão foi encapsulada nas factories, tornando a estrutura mais modular e legível.

5. Possíveis Extensões ou Evoluções

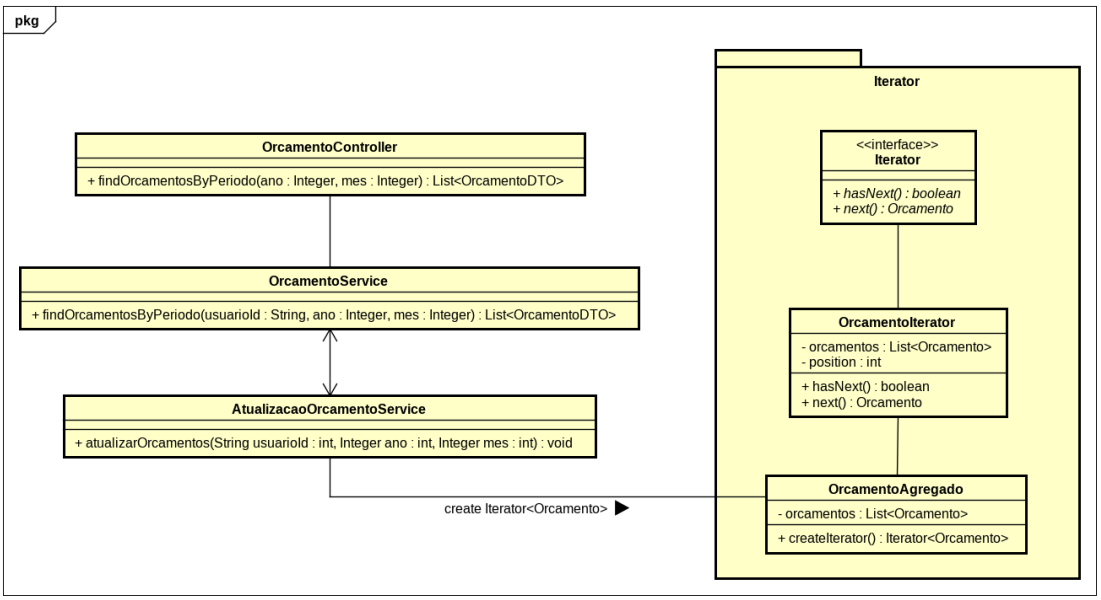
- **Suporte a novos tipos de transações recorrentes:** Caso novas categorias de transações sejam adicionadas, a factory pode ser expandida para incluí-las sem impacto no restante do sistema.

5. Diagramação

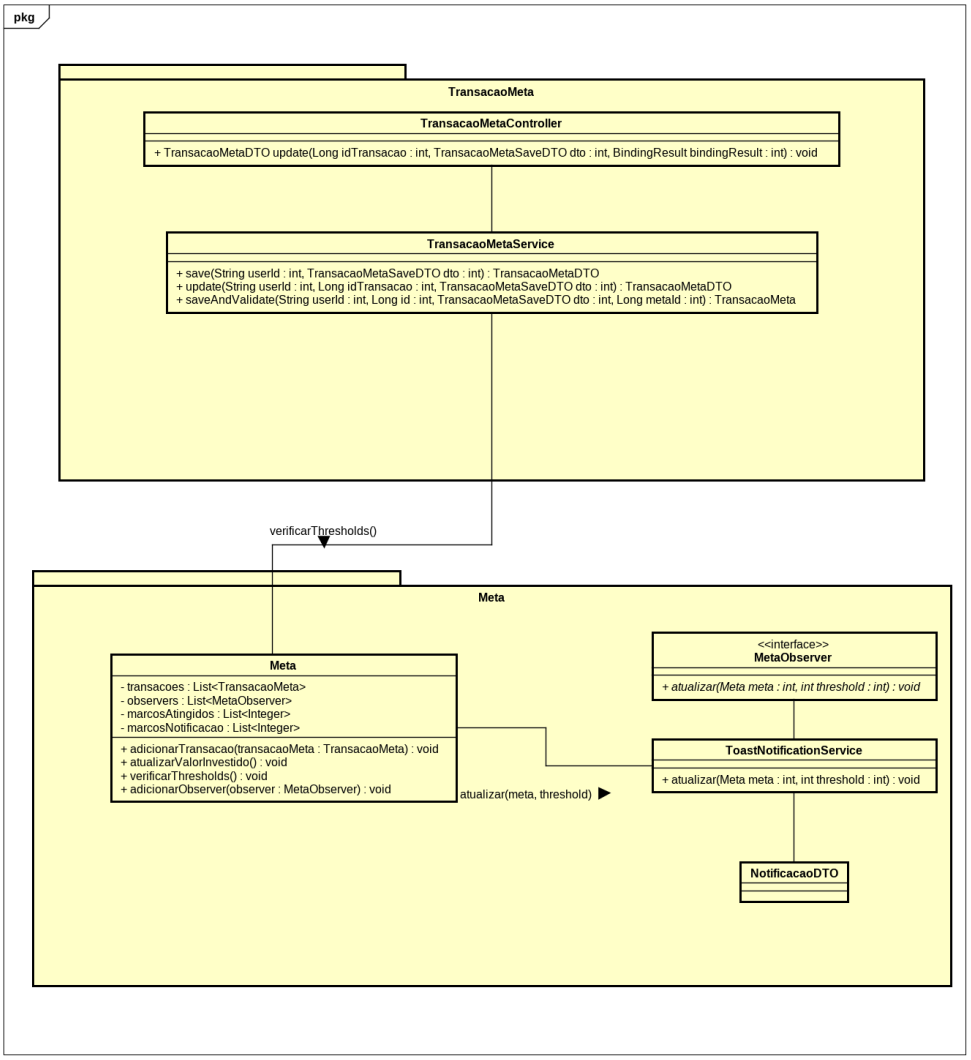
Factory e Strategy (pacote repetirTransacao)



Iterator para atualização de Orcamentos



Observer na notificação de metas

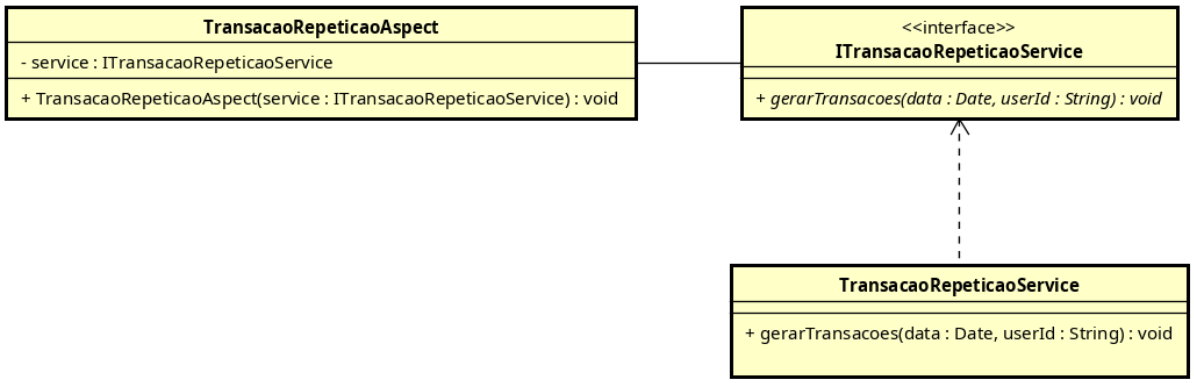


6. Inversão e Injeção de dependências

1. Contextualização

Por padrão o Spring Data JPA realiza a inversão de dependência dos repositories, logo a inversão e dependências é aplicada e gerenciada implicitamente pelo próprio spring.
Em nosso projeto implementamos a inversão e injeção de dependência também nos services.
Vamos pegar usar como **exemplo** a inversão de dependência

2. Diagrama



A injeção de dependências é declarada na **InjecaoDependenciasConfig** no pacote **config**, nela declaramos qual classe concreta deve ser provida quando determinada interface é injetada em nosso sistema, seja por construtor ou pela notação @Autowired.

3. Exemplo de injeção via classe de configuração gerenciada pelo Spring

```
@Configuration
public class InjecaoDependeciaConfig {
  //indicando a classe que deve ser injetada quando a interface ITransacaoRepeticao for
declarada
  @Bean
  public ITransacaoRepeticaoService TransacaoRepeticaoService(){return new
TransacaoRepeticaoService();}
 // . . .
}
```