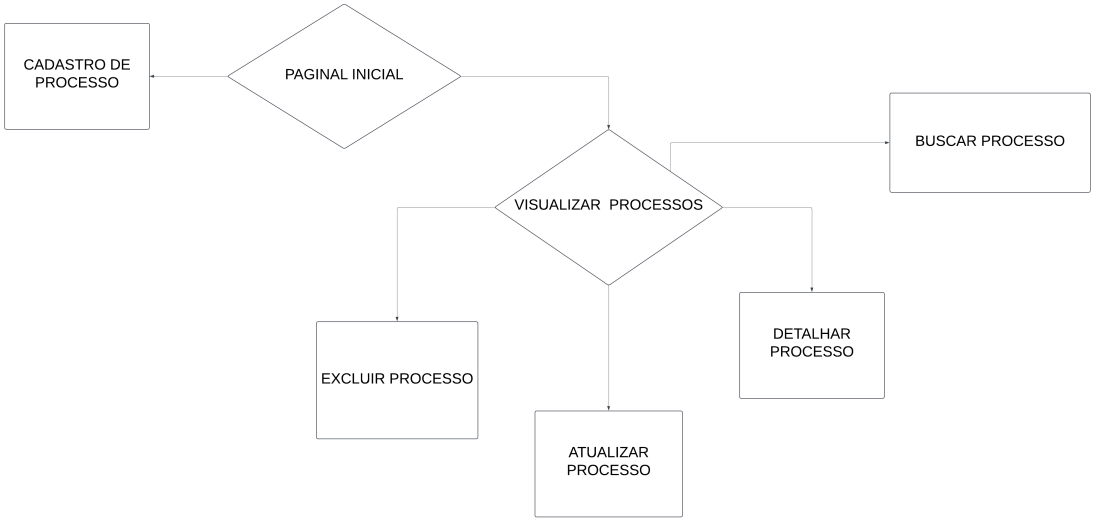
Gestão de Processos

O sistema permitirá adicionar, editar, excluir e visualizar informações sobre os processos de forma prática e eficiente. O sistema integrará quatro padrões de projeto, com pelo menos um de cada categoria (criação, estrutural e comportamental) nas operações CRUD.

Objetivos

1. **Implementar as funções de CRUD** para gerenciar processos.
2. **Integrar os padrões de projeto** de forma eficiente nas operações do sistema.
3. **Melhorar o acompanhamento** de prazos e documentos relacionados aos processos.

FLUXO DO SITE:



1. **Página Inicial** : Esta é a primeira tela que os usuários veem ao acessar o site. É a porta de entrada para as funcionalidades disponíveis, **Visualizar Processos e Cadastrar Novo Processo**.
2. **Visualizar Processos**: Permite ao usuário ver a lista de processos e acessar:
    - **Detalhes do Processo**
    - **Editar Processo**
    - **Excluir Processo**
    - **Pesquisar Processo**
3. **Cadastrar Novo Processo**: Página para criar um novo processo.

- Opção na Página Inicial permite ao usuário adicionar um novo processo.
- Esta ação leva o usuário a um formulário onde ele pode inserir as informações necessárias sobre o novo processo.

Padrões de Projeto Utilizados

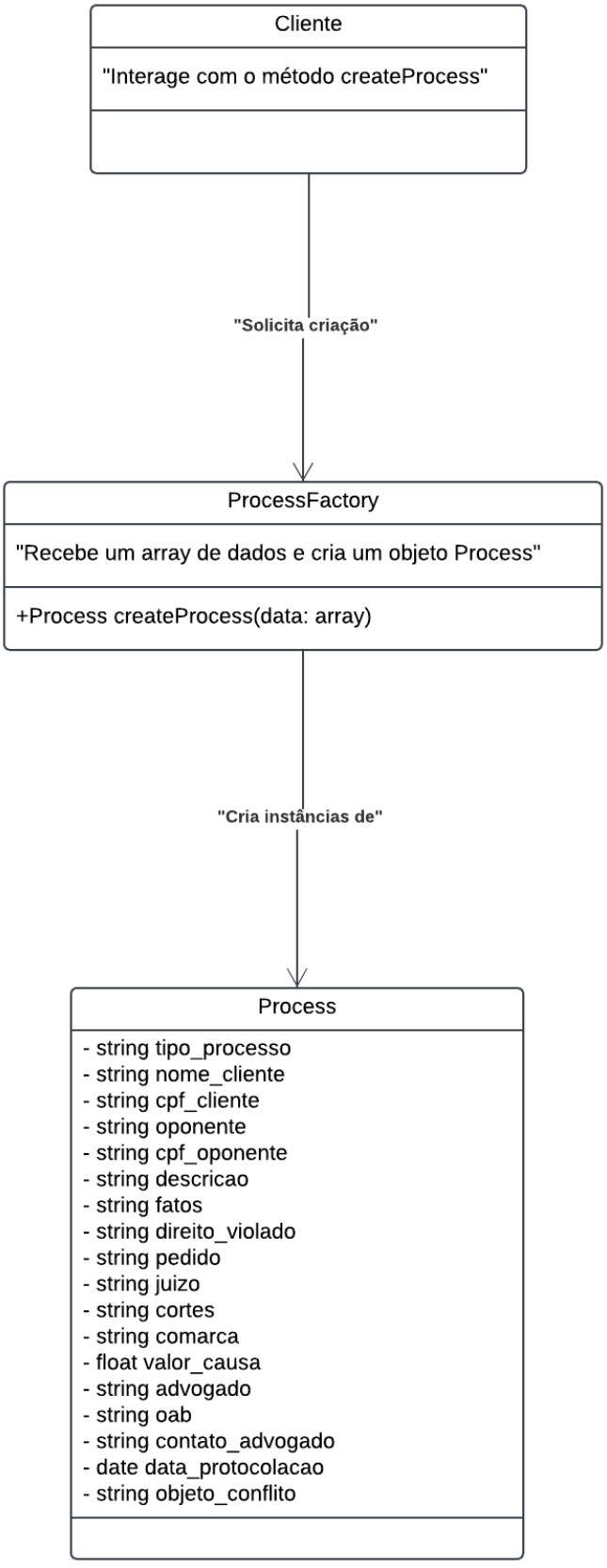
2.1 Factory (Padrão de Criação)

O código implementa uma versão simplificada do padrão **Factory**. Centraliza a criação de objetos da classe `Process`, permitindo instanciar processos de maneira consistente e reutilizável. A simplicidade no uso desse padrão esta na ausência de diferenciação entre tipos de processos e na utilização de uma única classe base (`Process`) para todos os casos. O método `createProcess` na classe `ProcessFactory` é responsável por encapsular a lógica de construção do objeto `Process,` utiliza os dados fornecidos como entrada e retorna um objeto configurado, reduzindo a complexidade do código cliente que consome essa funcionalidade.

Onde está implementado:

- Arquivo: `ProcessFactory.php`

- Diretório: `Models`


Cliente
"Interage com o método createProcess"
"Solicita criação"
ProcessFactory
"Recebe um array de dados e cria um objeto Process"
+Process createProcess(data: array)
"Cria instâncias de"
Process
- string tipo_processo
- string nome_cliente
- string cpf_cliente
- string oponente
- string cpf_oponente
- string descricao
- string fatos
- string direito_violado
- string pedido
- string juizo
- string cortes
- string comarca
- float valor_causa
- string advogado
- string oab
- string contato_advogado
- date data_protocolacao
- string objeto_conflito

Estrutura

A estrutura segue os princípios do padrão Factory, com foco em simplicidade:

1. `ProcessFactory` **(Creator):**
   - Classe responsável por instanciar objetos da classe `Process`.
   - Método principal: `createProcess($data)`.
2. `Process` **(Product):**
   - Classe que representa o processo.
   - Contém atributos como `nome_cliente`, `cpf_cliente`, `tipo_processo`, e outros.

Participantes

1. **Fábrica (`ProcessFactory`):**
   - Atuação: Centraliza a lógica de criação de objetos.
   - Responsabilidade: Receber os dados necessários e retornar um objeto da classe `Process`.
2. **Produto (`Process`):**
   - Atuação: Representa o objeto criado pela fábrica.
   - Responsabilidade: Armazenar informações do processo, como detalhes do cliente, descrição do caso, e informações adicionais.
3. **Cliente do sistema:**
   - Interage com a fábrica para criar objetos `Process`.
   - Trabalha diretamente com a instância retornada, sem conhecer detalhes da criação (cliente - solicitação feita pela controller/facade).

Colaborações

1. `ProcessFactory` **e** `Process`**:**
   - A classe `ProcessFactory` cria e retorna objetos `Process` com base nos dados fornecidos.
   - A classe `Process` encapsula todos os atributos do processo, permitindo que o cliente do sistema manipule o objeto de forma direta.
2. **Cliente do sistema e** `ProcessFactory`**:**

- O cliente utiliza o método `createProcess` para criar um processo sem precisar instanciar diretamente a classe `Process`.
- A interação reduz o acoplamento, já que o cliente trabalha apenas com a interface fornecida pela fábrica.

3. **Armazenamento de dados:**
   - O cliente fornece os dados necessários para criar o objeto, que são encapsulados pelo `Process`.

2.2 Strategy (Padrão Comportamental)

Descrição:

O padrão Strategy organiza uma coleção de comportamentos ou lógicas, encapsulando cada um de forma independente e permitindo que sejam trocados dinamicamente. No sistema, ele é utilizado para adaptar as regras de validação de processos conforme o seu tipo.

Onde está implementado:

- Arquivo principal: `ProcessValidator.php`
- Arquivos específicos: `CivilValidator.php`, `CriminalValidator.php`, `FamilyValidator.php`, `LaborValidator.php`
- Diretório: `Validators`

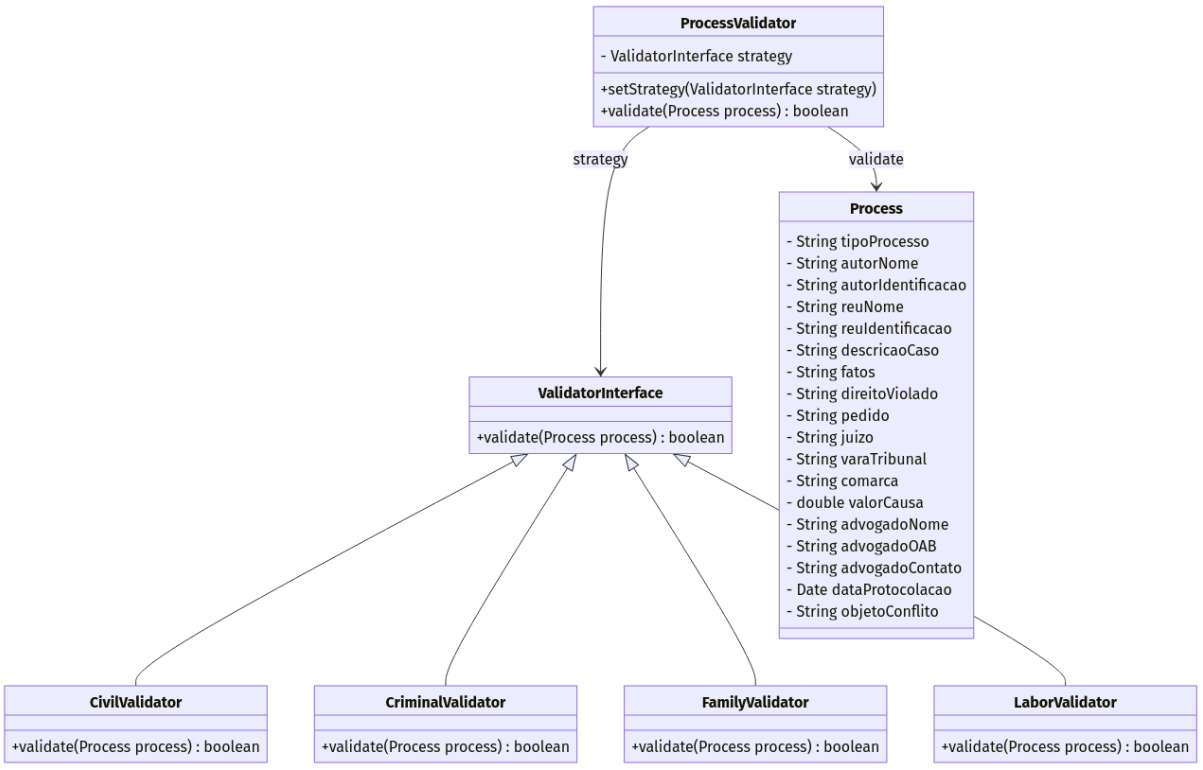
Uso no sistema:

O `ProcessValidator` permite que a validação de um processo seja feita por classes específicas, como `CivilValidator` ou `CriminalValidator`, dependendo do tipo. Isso oferece flexibilidade e organização na implementação de regras de validação.

Benefícios:

- Redução da complexidade da validação.
- Fácil troca ou adição de novas regras de validação.
- Promove coesão, pois cada classe foca em um tipo específico de validação.

Estrutura, Participantes e Colaborações



Estrutura

Implementação do padrão **Strategy**, com foco na separação das regras de validação de processos. Ele organiza as classes de forma que a lógica de validação seja flexível e intercambiável, permitindo atender diferentes tipos de processos (Civil, Criminal, Familiar, Trabalhista) sem modificar o código principal.

- A classe `ProcessValidator` é o **Contexto**, que utiliza uma instância de uma estratégia de validação (implementação da interface `ValidatorInterface`).
- A interface `ValidatorInterface` define o contrato para todas as estratégias de validação.
- As classes concretas (`CivilValidator`, `CriminalValidator`, `FamilyValidator`, `LaborValidator`) implementam as regras específicas de validação de processos.
- A classe `Process` representa o objeto que será validado pelas estratégias.

---

2. Participantes do Sistema

1. `ProcessValidator` (Contexto):
   - Classe principal que coordena a validação de um processo.
   - Depende de uma implementação de `ValidatorInterface`, que é configurada em tempo de execução.
   - **Colaboração**:
     - Chama o método `validate(Process process)` da estratégia configurada para validar um processo.
2. `ValidatorInterface` (Estratégia Abstrata):
   - Define a interface que todas as estratégias de validação devem implementar.
   - Método:
     - `validate(Process process): boolean` – Realiza a validação do processo.
3. `CivilValidator`, `CriminalValidator`, `FamilyValidator`, `LaborValidator` (Estratégias Concretas):
   - Implementações concretas de `ValidatorInterface`, que encapsulam regras específicas para cada tipo de processo.
   - Cada classe possui uma lógica de validação diferente baseada nos dados do processo.
   - **Colaboração**:
     - São usadas pelo `ProcessValidator` para validar um processo específico.
4. `Process` (Elemento de Domínio):
   - Representa o objeto a ser validado.
   - Contém os dados do processo, como tipo, autor, réu, descrição do caso, e outros atributos.

---

3. Colaborações entre as Classes

1. `ProcessValidator` e `ValidatorInterface`:
   - O `ProcessValidator` utiliza uma instância de `ValidatorInterface` para delegar a validação. Isso permite que diferentes estratégias sejam usadas de forma intercambiável.

2. `ValidatorInterface` **e Estratégias Concretas:**
    - As classes concretas (`CivilValidator`, `CriminalValidator`, etc.) implementam `ValidatorInterface`, o que garante que todas tenham o mesmo método `validate()`.
3. `ProcessValidator` **e Estratégias Concretas:**
    - O `ProcessValidator` não precisa conhecer detalhes das estratégias concretas, apenas chama o método `validate()` da estratégia configurada, promovendo baixo acoplamento.
4. `Process` **e Estratégias:**
    - As estratégias concretas recebem uma instância de `Process` e realizam a validação com base nos dados fornecidos.

2.3 Facade (Padrão Estrutural)

Descrição:

O padrão Facade fornece uma interface simplificada para acessar operações complexas. No sistema, ele centraliza as operações relacionadas aos processos, como **CRUD** (criar, consultar, atualizar, excluir).

Onde está implementado:

- Arquivo: `ProcessFacade.php`
- Diretório: `Controller`

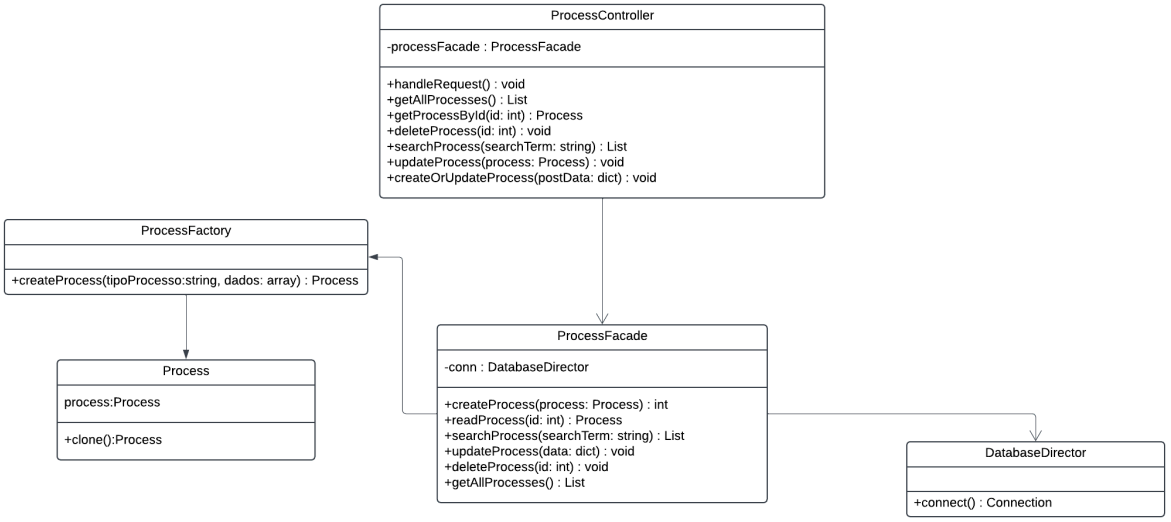
Uso no sistema:

O `ProcessFacade` centraliza as operações do sistema, ocultando detalhes de implementação dos controladores e fornecendo uma interface única para gerenciar os processos.

Benefícios:

- Simplifica a interação com subsistemas complexos.
- Reduz o acoplamento entre componentes.

- Facilita a manutenção e teste do sistema.



Estrutura

O **ProcessFacade** é o ponto de entrada principal para gerenciar as operações relacionadas aos processos. Ele atua como uma camada intermediária que encapsula a complexidade de subsistemas, como validações e interações com o banco de dados. A principal vantagem dessa abordagem é a simplificação e redução de acoplamento entre a camada de controle (`ProcessController`) e os subsistemas.

Participantes

1. ProcessFacade (Central)

- **Função principal**: Fornece uma interface simplificada para gerenciar processos.
- **Métodos principais**:
  - **CRUD**:
    - `createProcess()`: Adiciona um novo processo ao sistema.
    - `readProcess()`: Recupera informações de um processo específico.
    - `updateProcess()`: Atualiza informações de um processo existente.
    - `deleteProcess()`: Remove um processo do banco de dados.

        - `getAllProcesses()`: Retorna todos os processos.
    - **Busca**:
        - `searchProcess()`: Realiza buscas específicas por id de processos.

2. ProcessController

- **Função**: Atua como mediador, repassando as solicitações do usuário ao `ProcessFacade` e retornando os resultados.
- **Colaboração**: Depende completamente do `ProcessFacade` para executar operações.

3. Subsistemas que colaboram com o ProcessFacade

- `DatabaseConnection`: Fornece a conexão para operações no banco de dados.
- `Process`: Modelo de domínio representando os dados de um processo.
- `ProcessFactory`: Cria instâncias de `Process` com base nos dados de entrada.
- `ProcessValidator` **e validadores específicos**: Garante a consistência e validade dos dados antes de persistir.

4. Validadores específicos

- `FamilyValidator`, `LaborValidator`, `CivilValidator`, `CriminalValidator`: São estratégias usadas pelo `ProcessValidator` para validações específicas do tipo de processo.

Colaborações com o ProcessFacade

1. Recebendo comandos do ProcessController

O `ProcessController` delega suas operações ao `ProcessFacade`, como:

- Criar ou atualizar processos.
- Buscar dados de processos específicos ou listar todos.

2. Gerenciando operações CRUD e busca

O `ProcessFacade` encapsula toda a complexidade de interagir com o banco de dados. Ele:

- Usa `DatabaseConnection` para executar consultas SQL.

- Retorna objetos `Process` ou listas de objetos para o controlador.

3. Criando objetos de domínio

Para garantir consistência, o `ProcessFacade` delega ao `ProcessFactory` a criação de objetos `Process` com base nos dados fornecidos.

4. Validação dos dados

O `ProcessFacade`:

- Configura o `ProcessValidator` com a estratégia apropriada (ex.: `FamilyValidator`).
- Valida os dados antes de persistir ou atualizar no banco.

2.5 Prototype (Padrão de Criação)

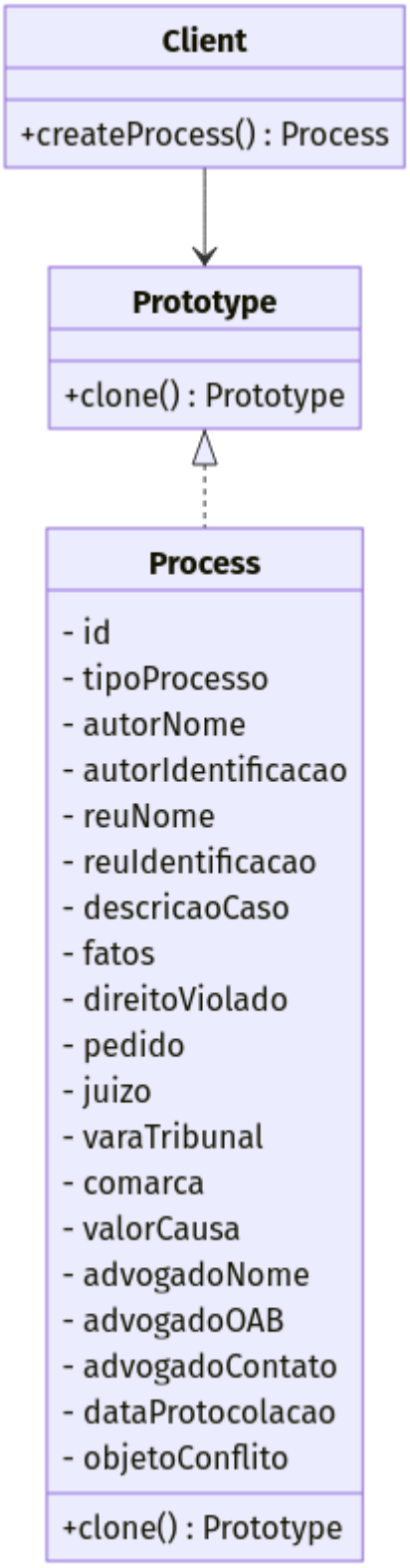
O padrão Prototype permite a criação de novos objetos como cópias de objetos existentes, reduzindo a necessidade de criar instâncias do zero.

Uso no sistema:

Ainda não implementado, mas poderia ser usado para clonar processos já existentes, criando cópias com base em processos previamente configurados.

Benefícios:

- Reduz a complexidade na criação de novos objetos.
- Reutilização de objetos existentes como base.
- Facilita a manipulação de objetos semelhantes.


Client
+createProcess() : Process
Prototype
+clone() : Prototype
Process
- id
- tipoProcesso
- autorNome
- autorIdentificacao
- reuNome
- reuIdentificacao
- descricaoCaso
- fatos
- direitoViolado
- pedido
- juizo
- varaTribunal
- comarca
- valorCausa
- advogadoNome
- advogadoOAB
- advogadoContato
- dataProtocolacao
- objetoConflito
+clone() : Prototype

Estrutura

O padrão **Prototype** permite criar novos objetos como cópias de objetos já existentes. Em vez de construir um objeto do zero, ele utiliza um protótipo preexistente para simplificar o processo de criação.

No sistema, a classe `Process` implementa o padrão Prototype. O método `clone()` da classe permite criar uma cópia de um objeto `Process` com o mesmo estado inicial.

Estrutura do sistema:

1. **Interface Prototype**: Define o método `clone()` que será implementado pelas classes concretas.
2. **Classe Concreta (Process)**:
   - Contém os atributos que representam os dados de um processo.
   - Implementa o método `clone()` para retornar uma nova instância do objeto com os mesmos atributos.
3. **Cliente (Client)**:
   - A parte do sistema que solicita cópias de objetos utilizando o método `clone()`.

Participantes

1. Prototype (Interface)

- **Função**: Define o contrato que as classes concretas devem seguir para implementar o padrão Prototype.
- **No código**: Representada pela interface `Prototype`, que contém o método `clone()`.

2. Concrete Prototype (Process)

- **Função**: Implementa o método `clone()` para permitir que objetos sejam clonados.
- **No código**:
  - A classe `Process` implementa a interface `Prototype`.
  - Seu método `clone()` cria uma nova instância da classe `Process`, copiando os valores dos atributos do objeto original.

3. Client

- **Função**: É o responsável por solicitar a criação de novos objetos utilizando o método `clone()`.
- **Exemplo no sistema**:
  - Um módulo de gerenciamento de processos pode solicitar a clonagem de um processo para criar uma nova entrada com base em um processo já existente.
  - Isso facilita o reaproveitamento de informações sem a necessidade de repetir manualmente a configuração.

Colaborações

1. **Interação entre Prototype e Cliente**:
   - O cliente utiliza um objeto existente como protótipo.
   - Chama o método `clone()` para criar uma nova instância com as mesmas informações.
   - Após a clonagem, o cliente pode realizar modificações nos atributos do novo objeto, sem afetar o protótipo original.
2. **Reaproveitamento de Objetos**:
   - Objetos complexos, como `Process`, com muitos atributos, podem ser clonados rapidamente.
   - Isso elimina a necessidade de recriar e configurar todos os atributos manualmente.
3. **Flexibilidade na Personalização**:
   - Após a clonagem, o novo objeto pode ser personalizado para atender a requisitos específicos.
   - Por exemplo, um advogado pode criar um processo idêntico ao anterior, mas com alterações nos nomes do autor e do réu.

Colaborações

1. **Criação de Objetos com Base em Protótipos**
   - O cliente chama o método `clone()` de um objeto `Process` existente.

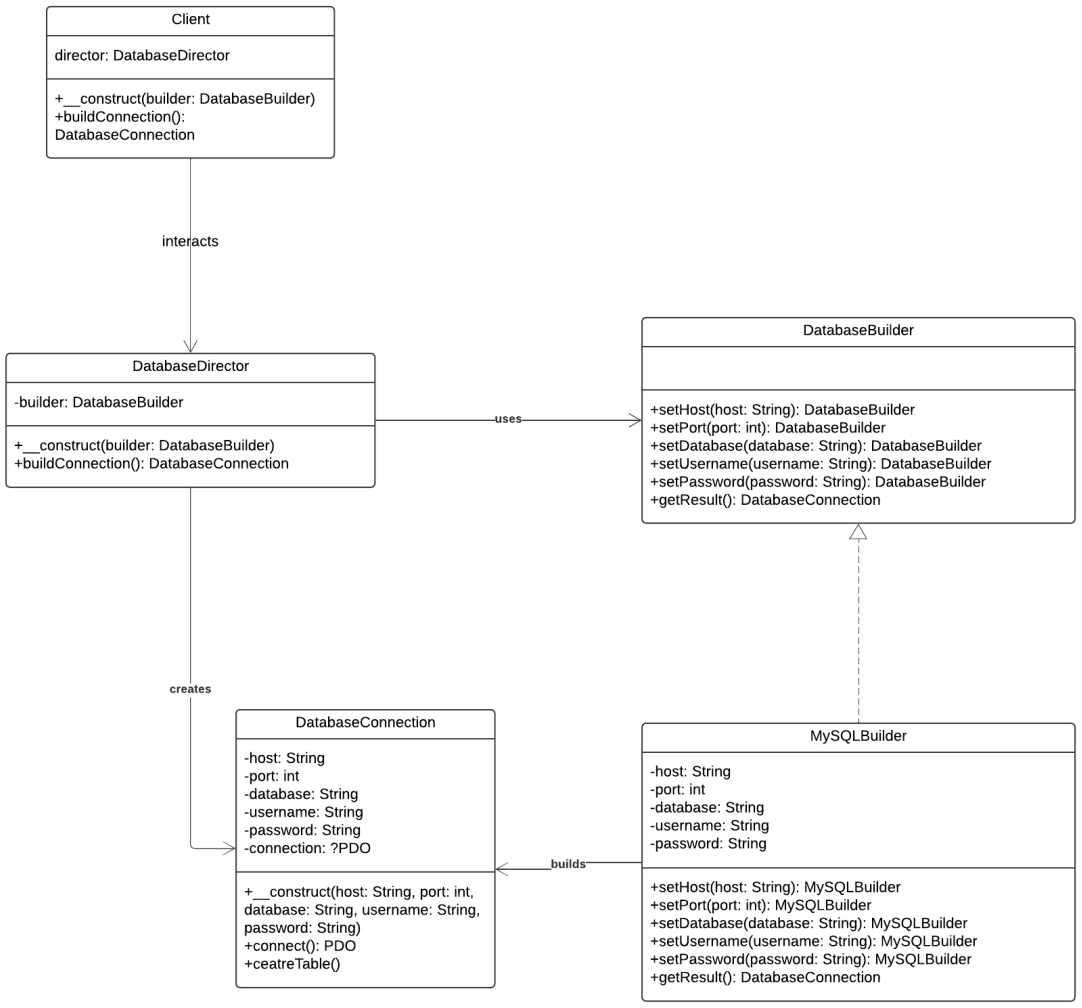
- O método `clone()` retorna uma nova instância com os mesmos atributos.

2. **Facilidade de Configuração**
   - O objeto clonado pode ser modificado conforme necessário após a cópia, sem afetar o protótipo original.
3. **Exemplo Prático no Sistema**
   - Imagine um cenário onde um usuário precisa duplicar um processo com pequenas alterações (ex.: mudar o autor, mas manter o restante das informações). Com o método `clone()`, o novo processo pode ser criado com apenas uma modificação mínima no objeto clonado.

2.4 Builder (Padrão de Criação)

O padrão **Builder** permite criar objetos complexos de forma incremental e reutilizável. Ele separa o processo de construção de um objeto de sua representação, facilitando a criação de diferentes configurações de objetos de forma organizada.

No sistema, o **Builder** é usado para configurar e criar conexões com bancos de dados. A classe `DatabaseDirector` coordena o processo de construção, enquanto `DatabaseBuilder` (e suas implementações como `MySQLBuilder`) define os passos para configurar uma conexão.



Estrutura do sistema:

1. **Interface Builder (`DatabaseBuilder`)**: Define os métodos para configurar os parâmetros de uma conexão com banco de dados e retorna o objeto final.
2. **Classe Concreta Builder (`MySQLBuilder`)**: Implementa a interface `DatabaseBuilder` para criar conexões específicas para MySQL.
3. **Diretor (`DatabaseDirector`)**: Controla o processo de construção chamando os métodos definidos no Builder.
4. **Produto (`DatabaseConnection`)**: Representa o objeto final configurado, que é retornado ao cliente.

Participantes

1. Builder (Interface `DatabaseBuilder`)

- **Função**: Define o contrato para configurar os atributos necessários à construção de um objeto de conexão.
- **No código**:
  - Métodos como `setHost(host: String)`, `setPort(port: int)`, e `getResult()` permitem configurar e recuperar o produto final.
  - Garante uma interface consistente para diferentes tipos de conexões.

2. Concrete Builder (`MySQLBuilder`)

- **Função**: Implementa os métodos definidos no Builder para configurar e criar conexões específicas de bancos MySQL.
- **No código**:
  - Armazena os atributos necessários (`host`, `port`, `database`, etc.).
  - Cria e retorna o objeto `DatabaseConnection` configurado no método `getResult()`.

3. Director (`DatabaseDirector`)

- **Função**: Coordena o processo de construção chamando os métodos do Builder em uma sequência predefinida.
- **No código**:
  - O método `buildConnection()` configura o builder passo a passo, definindo o host, porta, banco de dados, usuário e senha.
  - Retorna o objeto `DatabaseConnection` pronto.

4. Produto (`DatabaseConnection`)

- **Função**: Representa o objeto final que contém as configurações e pode ser usado para interagir com o banco de dados.
- **No código**:
  - Contém atributos como `host`, `port`, `database`, `username` e `password`.
  - Inclui métodos como `connect()` para estabelecer a conexão e `createTable()` para executar operações no banco.

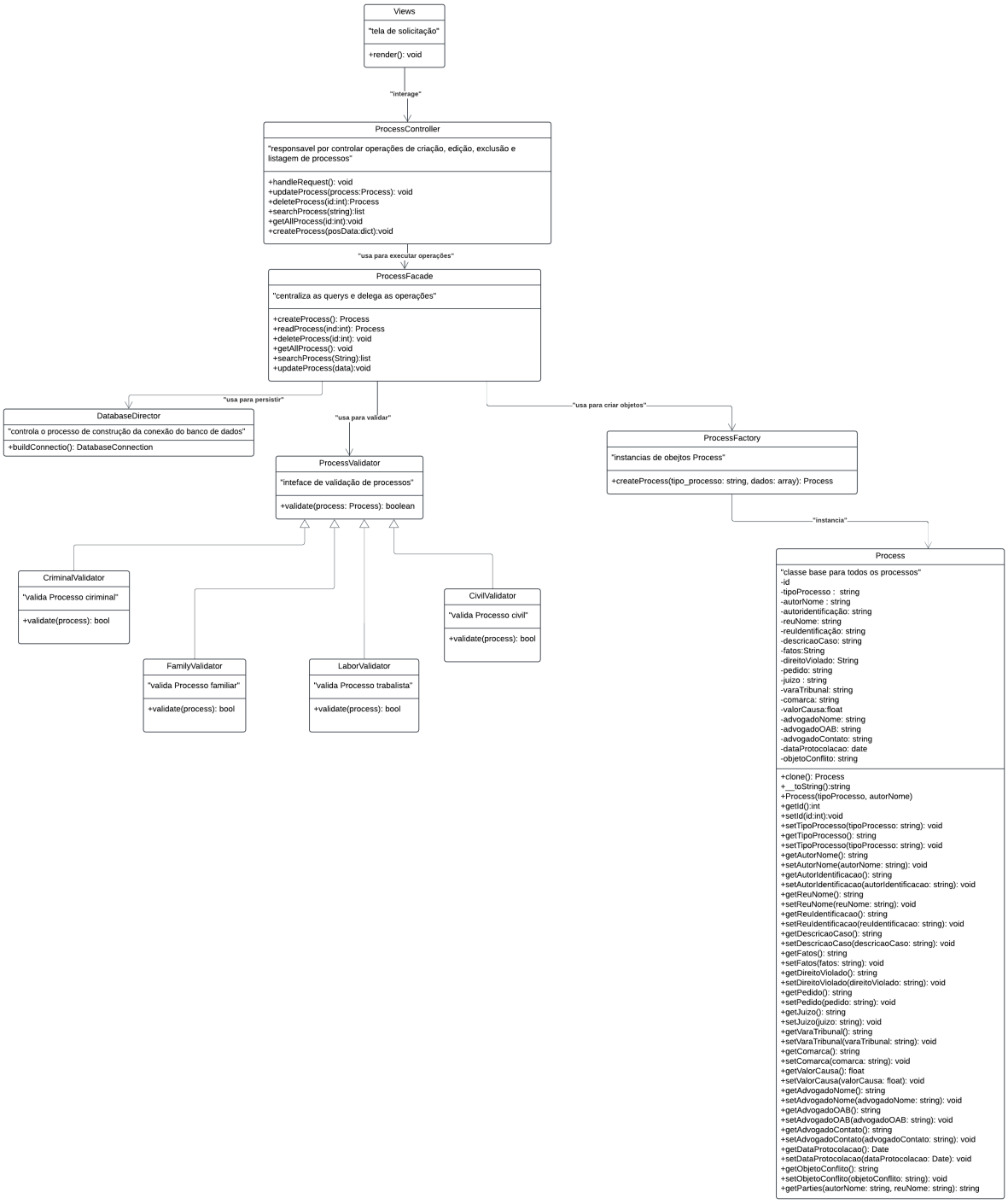
Colaborações

1. **Interação entre Cliente e Diretor**:
   - O cliente cria uma instância de `DatabaseDirector`, passando um Builder específico (por exemplo, `MySQLBuilder`).
   - O diretor coordena a construção, chamando métodos como `setHost()` e `setPort()` no Builder.
2. **Processo de Construção**:
   - O Builder armazena os parâmetros fornecidos pelo Diretor.
   - Após todas as configurações, o Builder retorna o objeto `DatabaseConnection` configurado.
3. **Configuração e Retorno do Objeto Final**:
   - O cliente utiliza o objeto `DatabaseConnection` retornado pelo diretor para realizar operações no banco de dados, como conexões ou consultas.

Exemplo Prático no Sistema

1. O cliente deseja criar uma conexão com um banco de dados MySQL.
2. Ele cria uma instância de `DatabaseDirector` e passa um `MySQLBuilder` como parâmetro.
3. O `DatabaseDirector` chama métodos do Builder, como `setHost("localhost")` e `setPort(3306)`, para configurar os parâmetros.
4. Após configurar todos os parâmetros, o método `getResult()` do Builder retorna um objeto `DatabaseConnection` configurado.
5. O cliente utiliza o objeto retornado para se conectar ao banco e executar operações.

Estruturando os Relacionamentos



principais relacionamentos que organizam projeto:

1. **Configuração (Config):**
    - **DatabaseDirector (Builder):**
        - Classe responsável pela construção da conexão com o banco de dados.

    - Relaciona-se com `ProcessFacade` para fornecer acesso ao banco.

2. **Controladores (Controller):**
    - **ProcessController:**
        - Coordena a interação entre:
            - **Views** (camada de apresentação): Atualizar, cadastro, detalhar, etc.
            - **ProcessFacade**: Centraliza operações do domínio de processos.
        - Relacionamento:
            - Usa `ProcessFacade` para executar as operações CRUD.
    - **ProcessFacade (Facade):**
        - Centraliza a lógica de negócios de operações com processos.
        - Depende de:
            - **ProcessFactory** para criar objetos `Process`.
            - **ProcessValidator** para validar objetos de acordo com regras específicas.
            - **DatabaseConnection** para persistência.
3. **Modelos (Models):**
    - **Process:**
        - Classe central que representa os dados e a lógica do processo.
        - Usa o padrão **Prototype** por meio do método `clone()`.
        - Relacionamento:
            - Criada por `ProcessFactory`.
            - Validada por `ProcessValidator` (ou suas subclasses especializadas).
            - Manipulada diretamente por `ProcessFacade`.
    - **ProcessFactory (Factory):**
        - Cria objetos `Process` com base no tipo de processo.
        - Relacionamento:

        - Instancia `Process` para ser usado no sistema.

4. **Validações (Validators):**
    - **ProcessValidator:**
        - Define a interface para validação de processos.
        - Implementa o padrão **Strategy** para permitir troca de regras de validação.
        - Subclasses especializadas:
            - **CivilValidator, CriminalValidator, FamilyValidator, LaborValidator**:
                - Estendem `ProcessValidator` e implementam validações específicas.
        - Relacionamento:
            - Usado pelo `ProcessFacade` para validar processos com base no tipo.
5. **Apresentação (Views):**
    - **Views (PHP Pages):**
        - Incluem `index.php`, `atualizar_processo.php`, `cadastro.php`, etc.
        - Relacionamento:
            - Dependem de `ProcessController` para interagir com o backend.

Fluxo do sistema

- Um **usuário interage com as Views** (ex.: cadastro ou atualização).
- As **Views chamam o ProcessController**, que coordena as operações.
- O **ProcessController chama o ProcessFacade** para centralizar a lógica.
- O **ProcessFacade**:
    - Cria objetos com `ProcessFactory`.
    - Valida processos com `ProcessValidator` .
    - Executa operações de banco via `DatabaseConnection`.