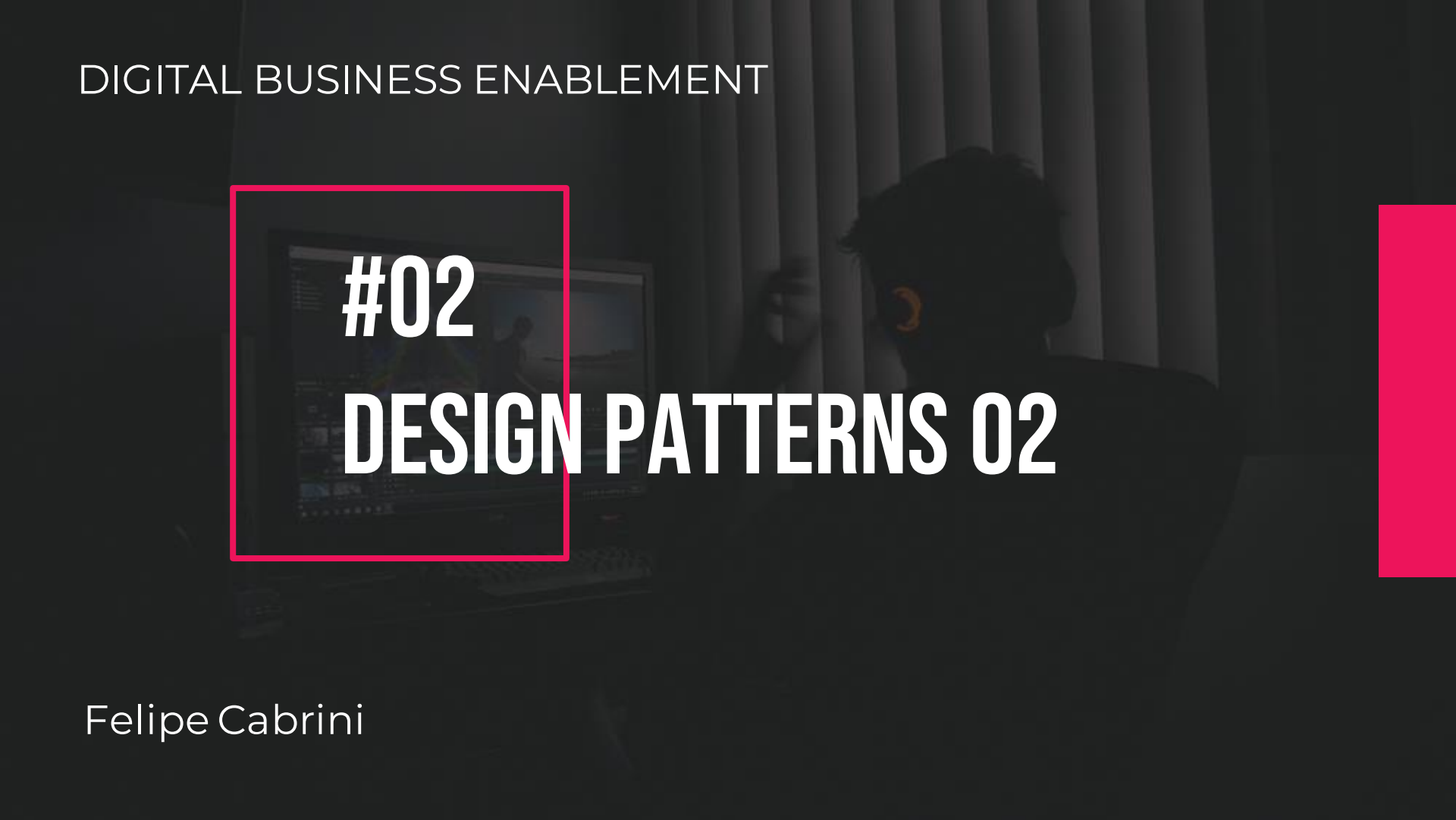


DIGITAL BUSINESS ENABLEMENT

#02
DESIGN PATTERNS 02

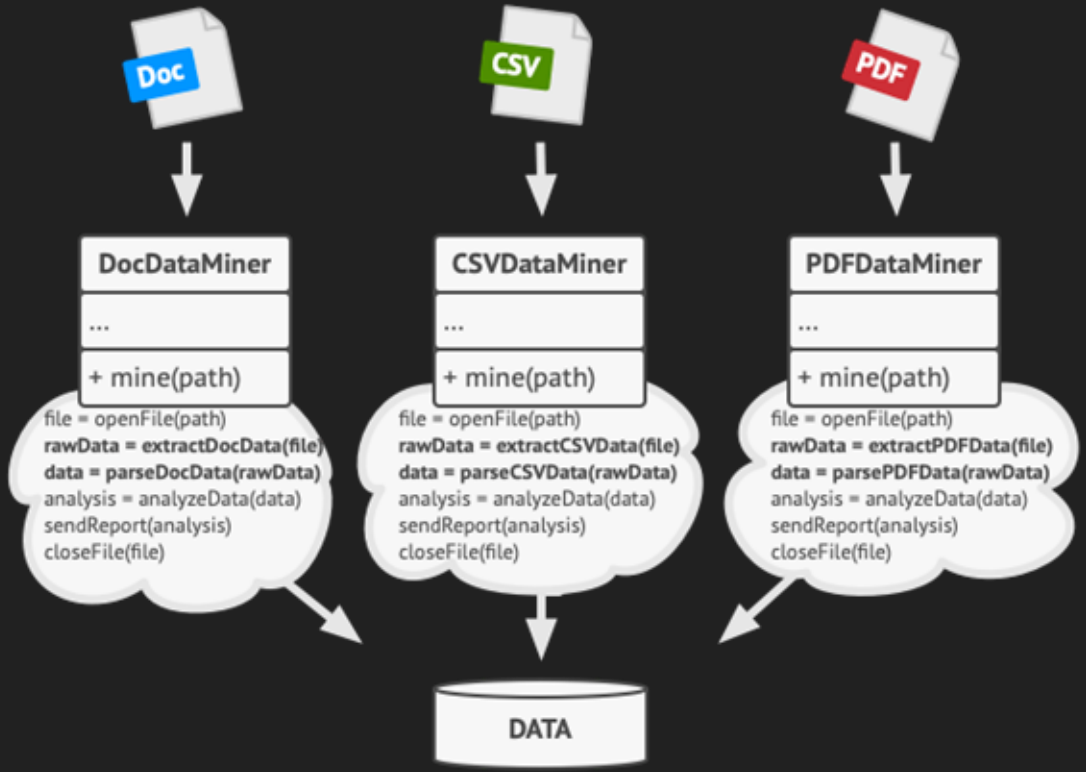
Felipe Cabrini

TEMPLATE METHOD

" Padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura

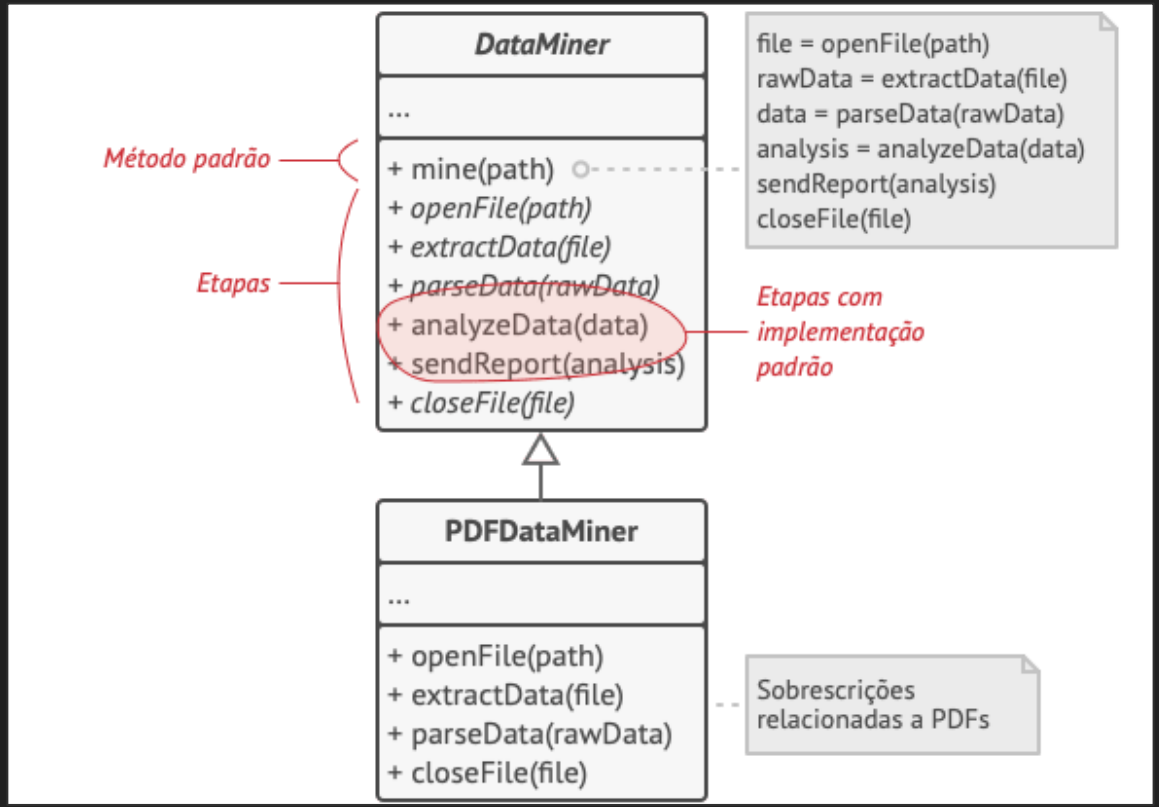
TEMPLATE METHOD

O Problema

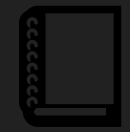


TEMPLATE METHOD

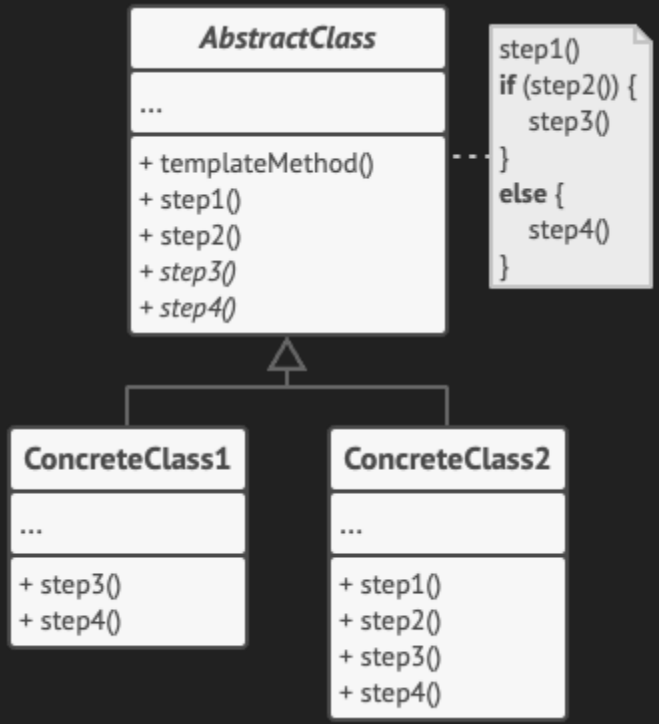
A Solução



TEMPLATE METHOD



A Solução



TEMPLATE METHOD

Vamos ao código

Imagine uma cafeteria que oferece várias bebidas, como café, chá e chocolate quente. Cada tipo de bebida requer um processo específico de preparação, mas o fluxo geral de preparação é o mesmo: aquecer a água, adicionar o ingrediente principal, mexer e servir. Vamos aplicar o design pattern Template Method para esse cenário.

TEMPLATE METHOD

Vamos ao código

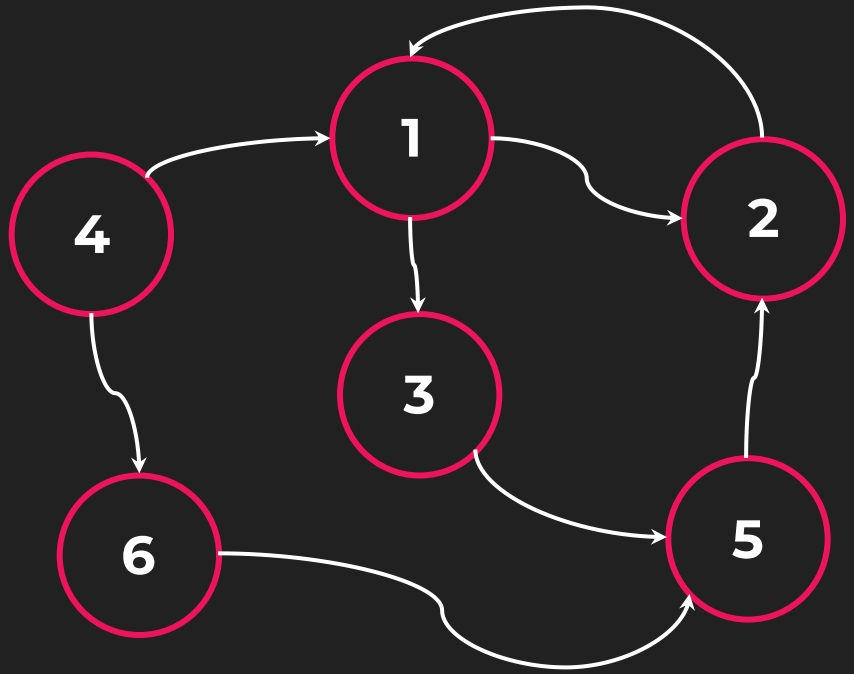
Neste exemplo, o design pattern Template Method é aplicado na classe Beverage. A classe abstrata Beverage define o fluxo geral de preparação das bebidas usando o método prepareBeverage(). As subclasses concretas Coffee e Tea substituem os métodos abstratos brew() e addCondiments() para implementar o processo específico de preparação de cada bebida.

STATE

" Padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. É como se o objeto mudasse de classe.

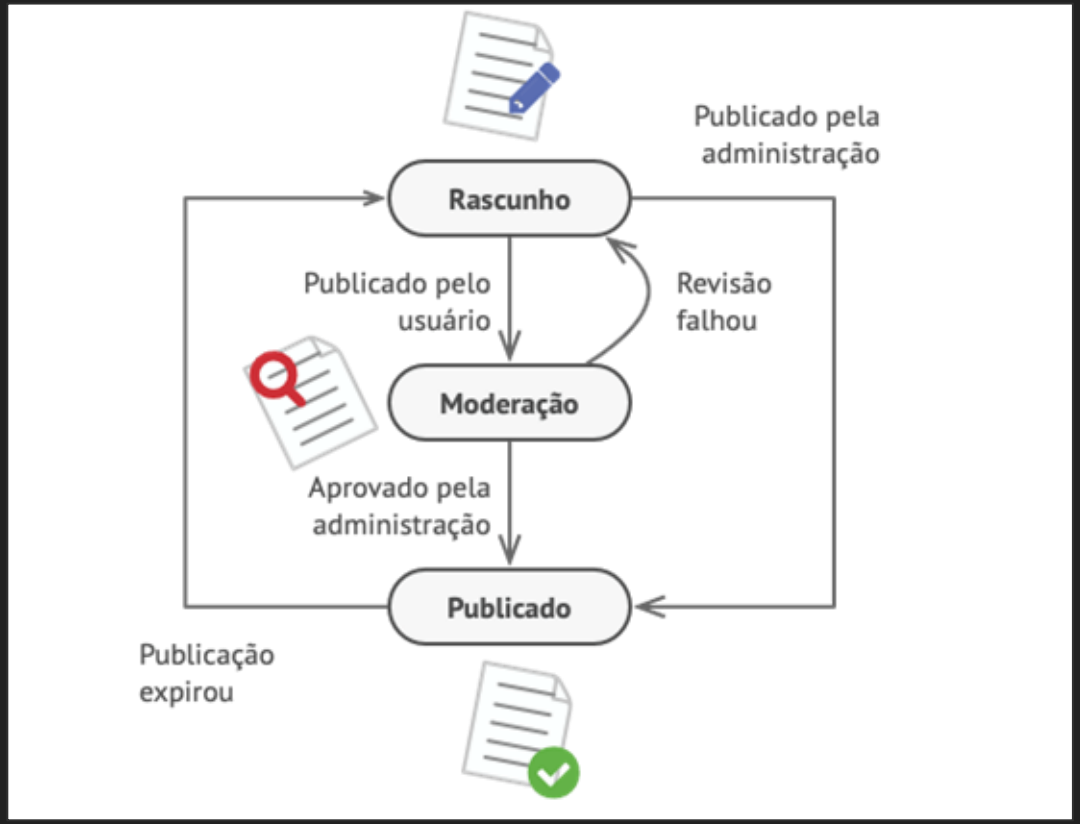
STATE

O problema



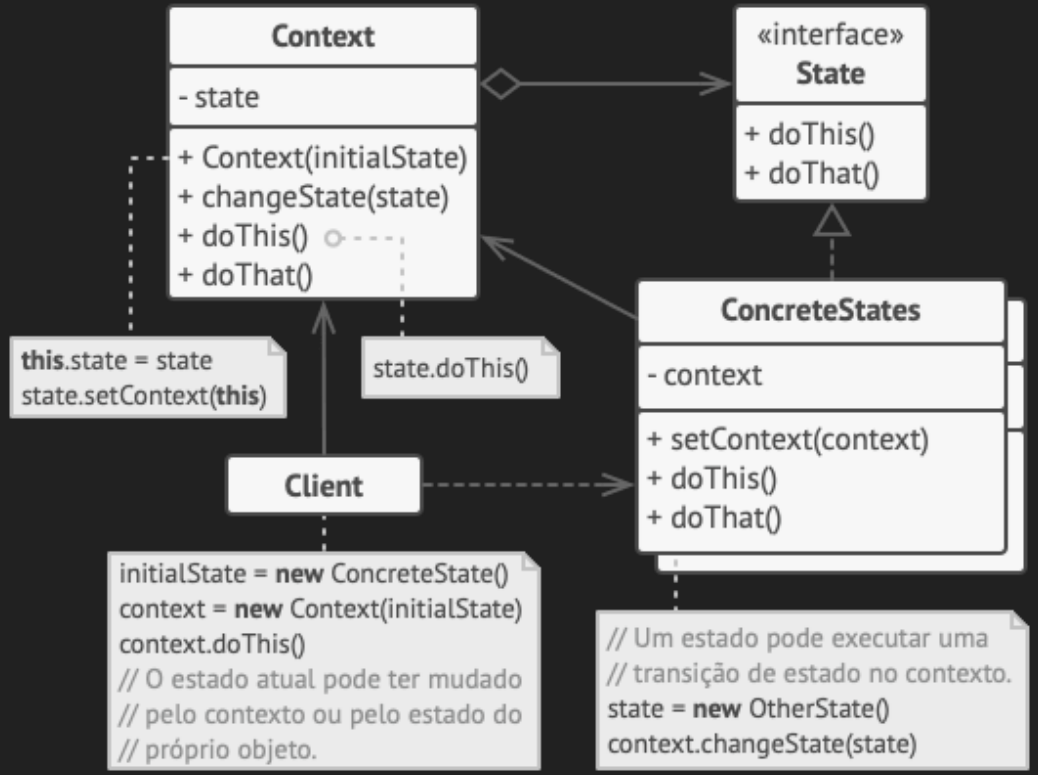
STATE

O problema



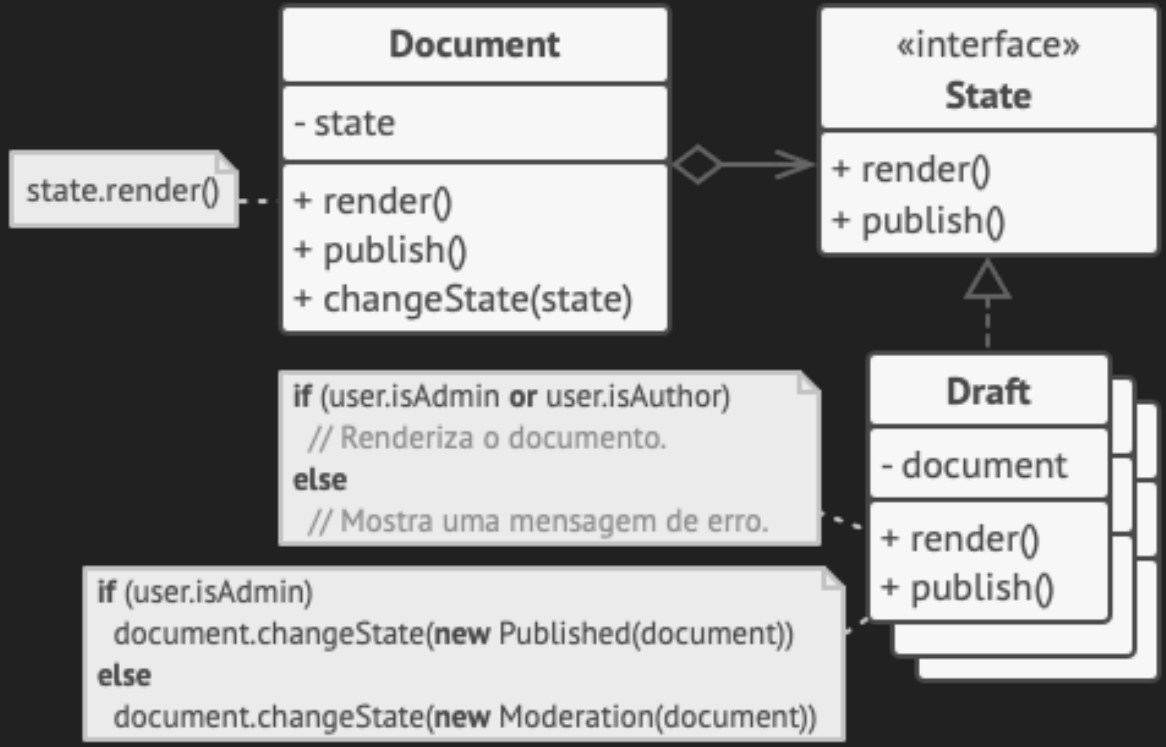
STATE

A Solução



STATE

A Solução



STATE

Vamos ao código

Vamos considerar um sistema de autenticação que possui diferentes estados de autenticação: Não Autenticado, Autenticado e Bloqueado. Dependendo do estado atual, diferentes ações podem ser realizadas, como fazer login, fazer logout e verificar o status da conta. Vamos aplicar o design pattern State para modelar esse cenário.

OBSERVER

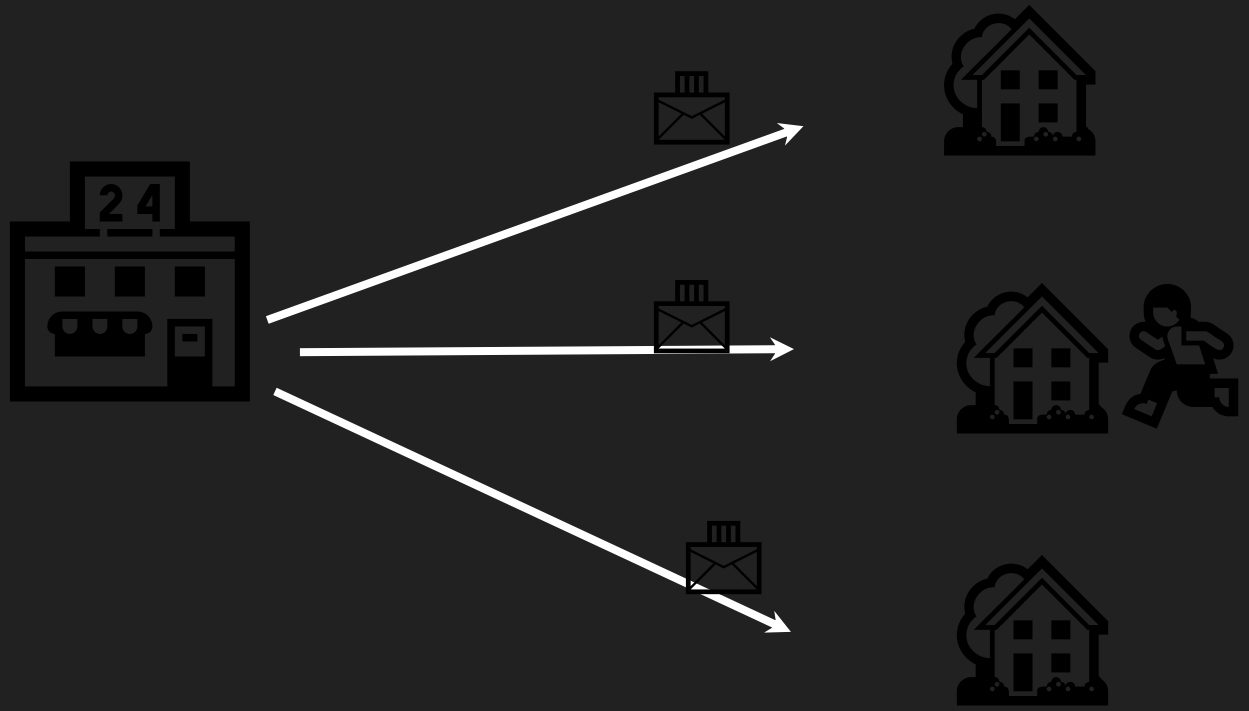
" Padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

OBSERVER

O problema

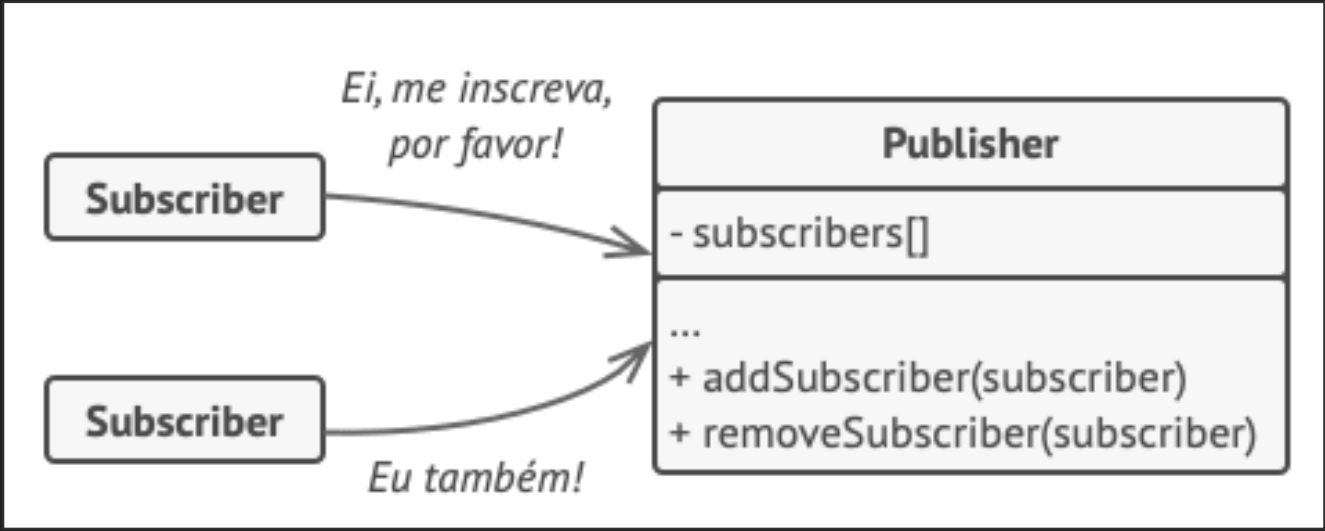
OBSERVER

O problema



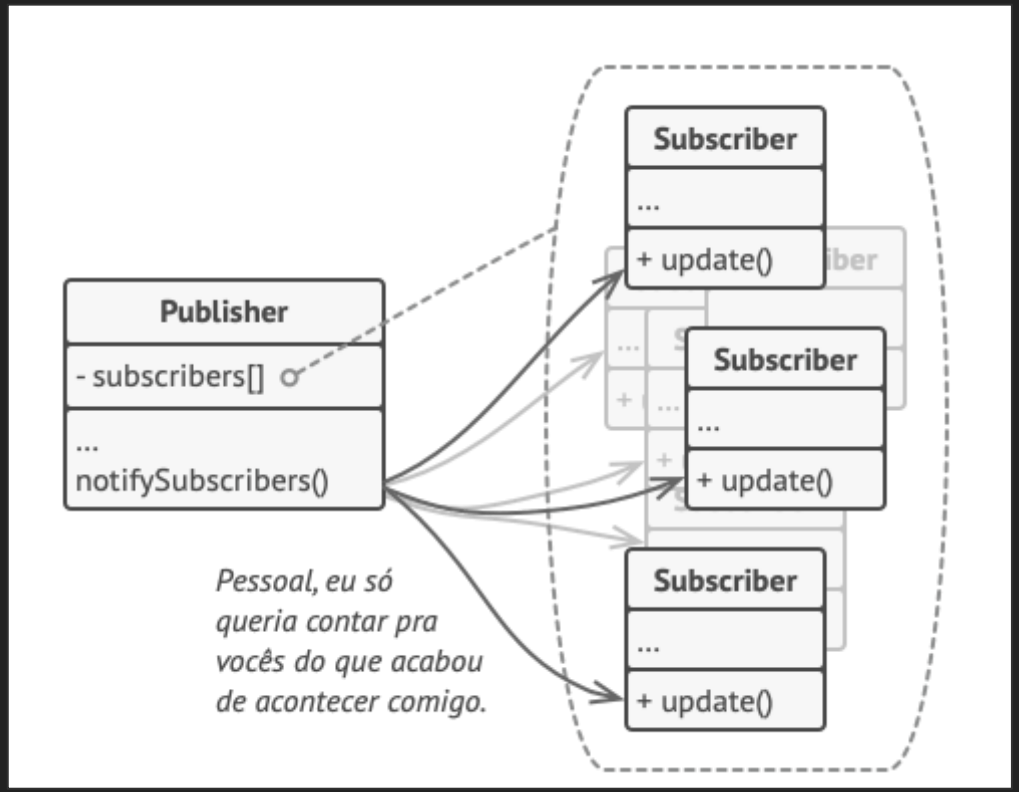
OBSERVER

A Solução



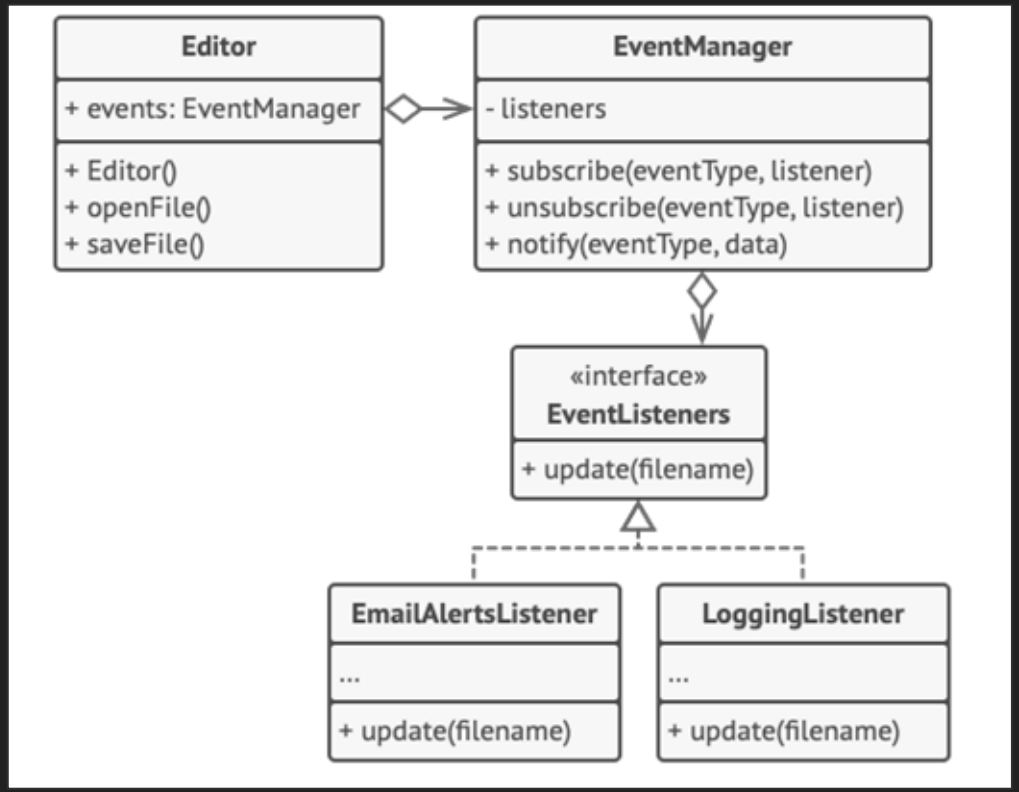
OBSERVER

A Solução



OBSERVER

A Solução



OBSERVER

Vamos ao código

O problema que estamos resolvendo é como criar um sistema de monitoramento de temperatura que seja eficiente e flexível, permitindo a adição e remoção de **observadores** (partes interessadas) que precisam ser notificadas sobre alterações na temperatura. Além disso, queremos garantir que o sistema possa alertar automaticamente quando a temperatura atingir um nível crítico.

Para resolver esse problema, estamos aplicando o design pattern Observer. Ele nos permite criar uma estrutura onde o sensor de temperatura (sujeito) pode notificar automaticamente os **observadores** (monitoramento e alerta) sempre que a temperatura exceder um limite crítico. Isso torna o sistema mais modular e flexível, permitindo a inclusão de novos observadores no futuro, se necessário.

A implementação utiliza uma abordagem de separação de responsabilidades, onde o sensor de temperatura é responsável apenas por rastrear a temperatura e notificar os **observadores**, enquanto os observadores são responsáveis por reagir às notificações de acordo com suas funções específicas (monitoramento ou alerta).

Atividade

- **Entrega dia 04/09/2023 23:59**
- **Valendo 5 pontos**
- individual
- Entregar github.
  Conter codigo fonte.
  README.md explicando quais foram os designs patterns foram usados e onde foram.
- Não precisa ter api/spring boot/gradle somente java com os designs patterns
  Precisa compilar e rodar e no console.
- Mandar link no **teams**.

Atividade

- Uma biblioteca universitária deseja modernizar seu sistema de gerenciamento. O novo sistema deve permitir o registro de livros, o empréstimo de livros aos alunos e a devolução dos mesmos.
- **Requisitos:**

1. **Registro de Livros**: A biblioteca deve ser capaz de adicionar novos livros ao sistema, cada um com um título, autor, ISBN e quantidade disponível.
2. **Empréstimo de Livros**: Os alunos podem pegar livros emprestados. Cada livro só pode ser emprestado a um aluno por vez.
3. **Devolução de Livros**: Após o empréstimo, os alunos devem devolver os livros.

Sugestão de implementação

1. Crie um design pattern comportamental que permita que as tarefas notifiquem automaticamente os observadores quando seu status mudar.
2. Implemente outro design pattern comportamental para permitir que diferentes estratégias de priorização de tarefas sejam escolhidas em tempo de execução.
3. Utilize um design pattern criacional para garantir que apenas uma única instância do sistema de gerenciamento de tarefas seja criada.
4. Utilize um design pattern estrutural para permitir que as tarefas sejam compostas em uma estrutura de árvore, onde uma tarefa pode conter outras tarefas.

Sugestão de implementação

- Observer Pattern (Comportamental):
  - Crie uma classe Task que atua como o sujeito observado.
  - Implemente uma interface TaskObserver para que as classes interessadas possam se inscrever como observadores.
  - Implemente o mecanismo de notificação para atualizar automaticamente os observadores quando o status da tarefa mudar.
- Strategy Pattern (Comportamental):
  - Crie uma interface PriorityStrategy com métodos para definir e obter prioridades.
  - Implemente diferentes estratégias de priorização, como LowPriorityStrategy, MediumPriorityStrategy e HighPriorityStrategy, que implementam a interface PriorityStrategy.
  - Na classe Task, inclua um atributo do tipo PriorityStrategy que permita a troca dinâmica da estratégia de priorização.

Sugestão de implementação

- Singleton Pattern (Criacional):
  - Implemente a classe TaskManager como um singleton para garantir que haja apenas uma instância do gerenciador de tarefas.
  - O TaskManager deve conter métodos para adicionar tarefas, listar tarefas e buscar tarefas por status.

- Composite Pattern (Estrutural):
  - Crie uma classe abstrata TaskComponent que represente tanto tarefas individuais quanto tarefas compostas.
  - Implemente classes concretas SingleTask e CompositeTask que herdam de TaskComponent, onde CompositeTask pode conter várias tarefas, incluindo outras CompositeTask.