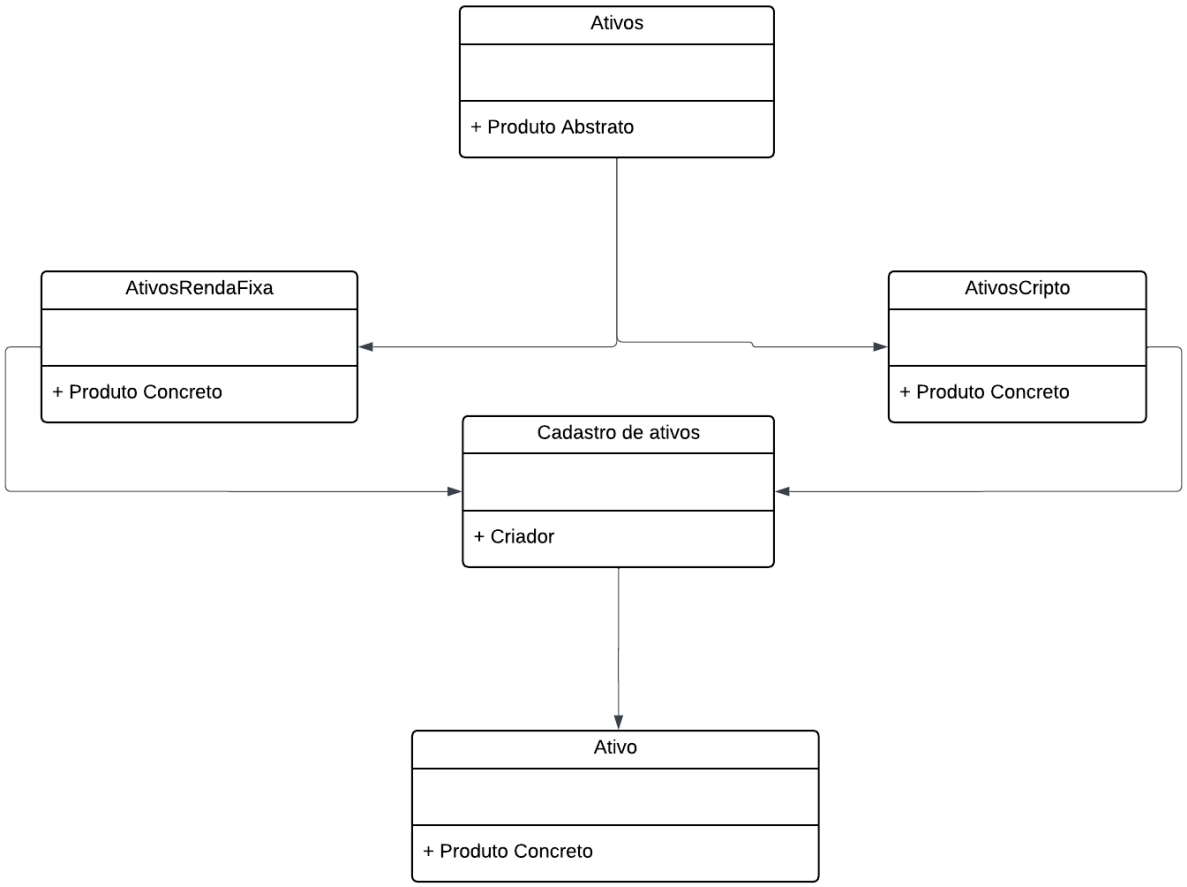
## Padrões de projeto

**1. Factory Method Descrição:** O Factory Method permite a criação dinâmica de diferentes tipos de ativos (fundos, renda fixa, ações, cripto) sem que o código principal precise saber as classes específicas. Isso facilita a inclusão de novos tipos de ativos no sistema, sem modificar a lógica principal. Com isso, o código fica mais flexível, permitindo futuras expansões e personalizações sem impactar outras partes do sistema. Esse padrão também centraliza a criação de objetos, tornando o código mais organizado e de fácil manutenção.

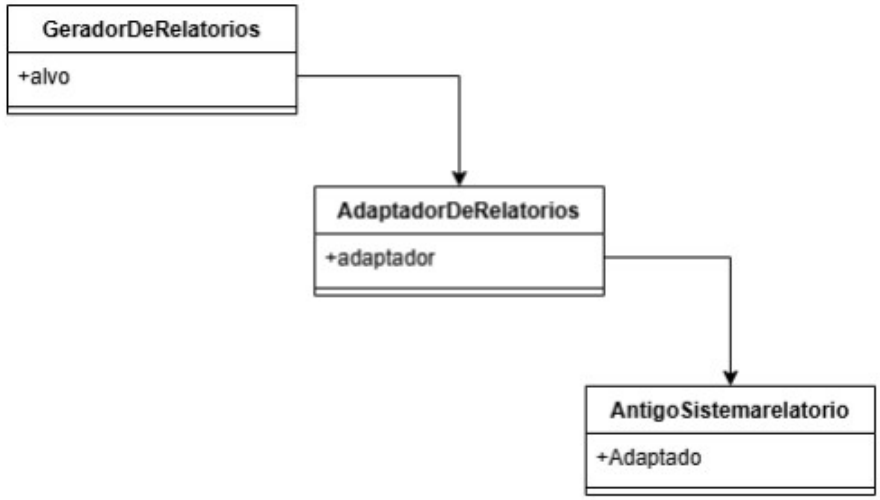
```
# Produto Abstrato
class Ativo:
    def exibir_informacoes(self):
        pass

# Produto Concreto 1
class Fundo(Ativo):
    def exibir_informacoes(self):
        return "Ativo: Fundo"

# Produto Concreto 2
class RendaFixa(Ativo):
    def exibir_informacoes(self):
        return "Ativo: Renda Fixa"

# Produto Concreto 3
class Acao(Ativo):
    def exibir_informacoes(self):
        return "Ativo: Ação"

# Produto Concreto 4
class Cripto(Ativo):
    def exibir_informacoes(self):
        return "Ativo: Cripto"

# Fábrica
class FabricaDeAtivo:
    def criar_ativo(self, tipo):
        if tipo == 'fundo':
            return Fundo()
        elif tipo == 'renda_fixa':
            return RendaFixa()
        elif tipo == 'acao':
            return Acao()
        elif tipo == 'cripto':
            return Cripto()
        else:
            raise ValueError("Tipo de ativo inválido")
```




**2. Adapter Descrição:** O Adapter é um padrão de design estrutural que permite que interfaces incompatíveis trabalhem juntas. No seu dashboard financeiro, ele pode ser utilizado para integrar um sistema de relatórios antigo com a nova interface moderna.

```
# Alvo
class GeradorDeRelatorio:
    def gerar_relatorio(self):
        pass

# Adaptee
class SistemaAntigoDeRelatorio:
    def gerar_relatorio_antigo(self):
        return "Relatório no formato antigo"

# Adaptador
class AdaptadorDeRelatorio(GeradorDeRelatorio):
    def __init__(self, sistema_antigo):
        self.sistema_antigo = sistema_antigo

    def gerar_relatorio(self):
        return self.sistema_antigo.gerar_relatorio_antigo()
```

```
# Código do Cliente
sistema_antigo = SistemaAntigoDeRelatorio()
adaptador = AdaptadorDeRelatorio(sistema_antigo)
print(adaptador.gerar_relatorio())  # Saída: Relatório no formato antigo
```




**3. Strategy Descrição:** O Strategy é um padrão comportamental que permite que o comportamento de uma classe seja alterado em tempo de execução. Pode ser usado no seu dashboard financeiro para definir diferentes modos de visualização dos dados (como gráfico de pizza ou tabela).

```
from abc import ABC, abstractmethod

# Estratégia
class ModoDeVisualizacao(ABC):
    @abstractmethod
    def exibir(self):
        pass

# Estratégia Concreta 1
class VisualizacaoGraficoPizza(ModoDeVisualizacao):
    def exibir(self):
        return "Exibindo gráfico de pizza"
```

```
# Estratégia Concreta 2
class VisualizacaoTabela(ModoDeVisualizacao):
    def exibir(self):
        return "Exibindo tabela"

# Contexto
class Dashboard:
    def __init__(self, modo: ModoDeVisualizacao):
        self._modo = modo

    def definir_modo(self, modo: ModoDeVisualizacao):
        self._modo = modo

    def exibir_dados(self):
        print(self._modo.exibir())

# Código do Cliente
dashboard = Dashboard(VisualizacaoGraficoPizza())
dashboard.exibir_dados()  # Saída: Exibindo gráfico de pizza
dashboard.definir_modo(VisualizacaoTabela())
dashboard.exibir_dados()  # Saída: Exibindo tabela
```