# TÓPICO 16 - REVISÃO GERAL PARA N1

Design Patterns - Professor Ramon Venson - SATC 2026.1

# O que são Design Patterns?

- Soluções recorrentes para problemas recorrentes
- Não são código pronto
- Descrevem intenção, estrutura e colaboração
- Ajudam em:
  - comunicação
  - reutilização
  - manutenção

# Dicionário

| Nome | Descrição |
|---|---|
| `Abstração` | Separação entre ideia e a implementação |
| `Acoplamento` | Grau de dependência entre classes |
| `Coesão` | Grau de foco de uma classe em uma única responsabilidade |
| `Encapsulamento` | Esconder detalhes internos e expor apenas o necessário |
| `Instanciar` | Criar um objeto a partir de uma classe |
| `Cliente` | Código que usa os objetos criados por um padrão |

# Gamma Categorization

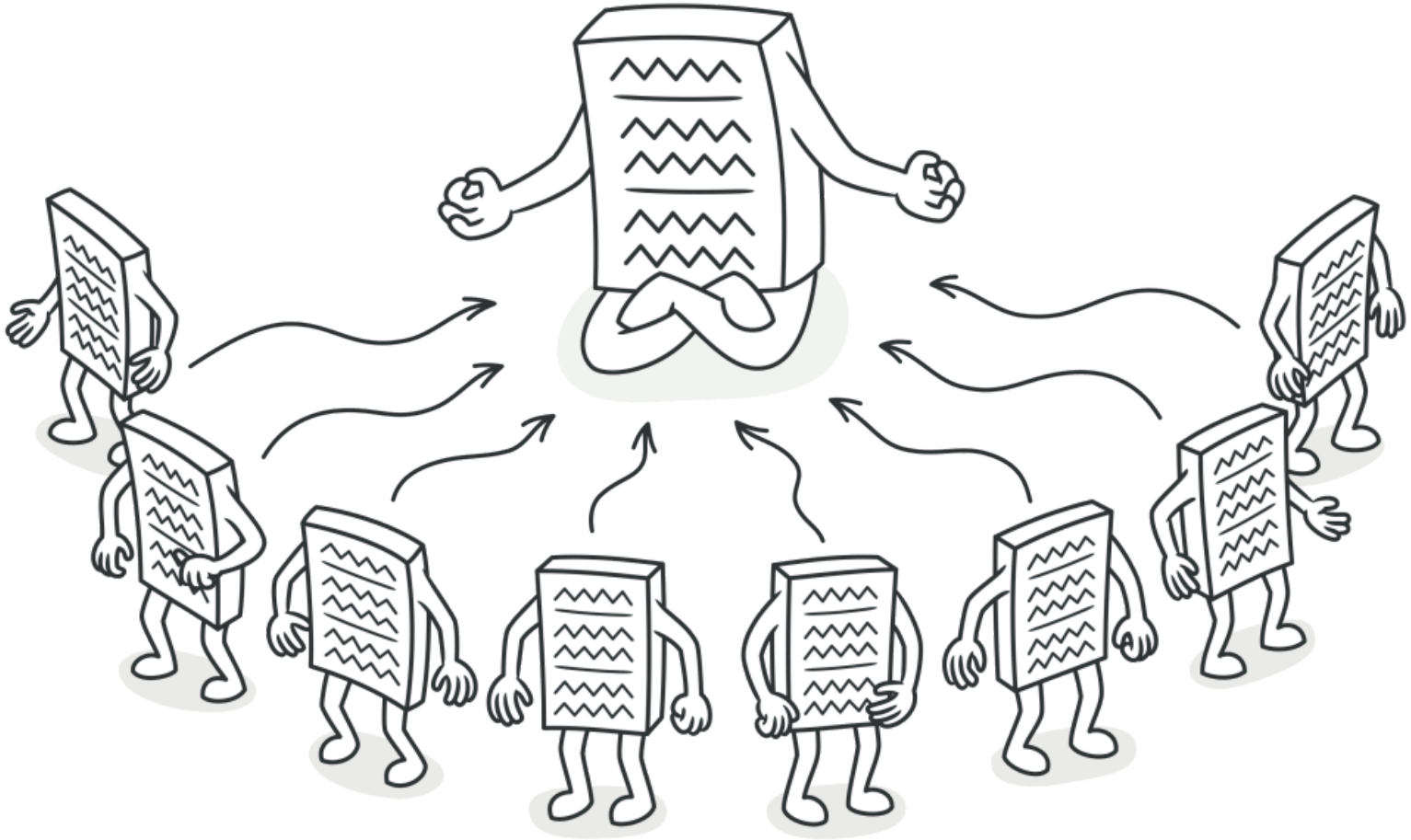
| Categoria | Foco | Resultado esperado |
|---|---|---|
| Criacional | Instânciar objetos | menor acoplamento na instanciação e mais flexibilidade na construção |
| Estrutural | Acoplamento entre as classes | colaboração entre as classes e organização mais claras |
| Comportamental | Diferentes formas de interagir | responsabilidades mais bem distribuídas |

# Padrões Criacionais

| Padrão | Quando pensar nele | Cuidado principal |
|---|---|---|
| Singleton | recurso único | estado global |
| Builder | objeto complexo | excesso de cerimônia |
| Factory Method | subclasses escolhem produto | mais hierarquia |
| Abstract Factory | famílias coerentes | custo para novos produtos |
| Prototype | clonagem de modelos | cópia rasa vs profunda |

# Singleton

Singleton garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela.



## Exemplo de Singleton

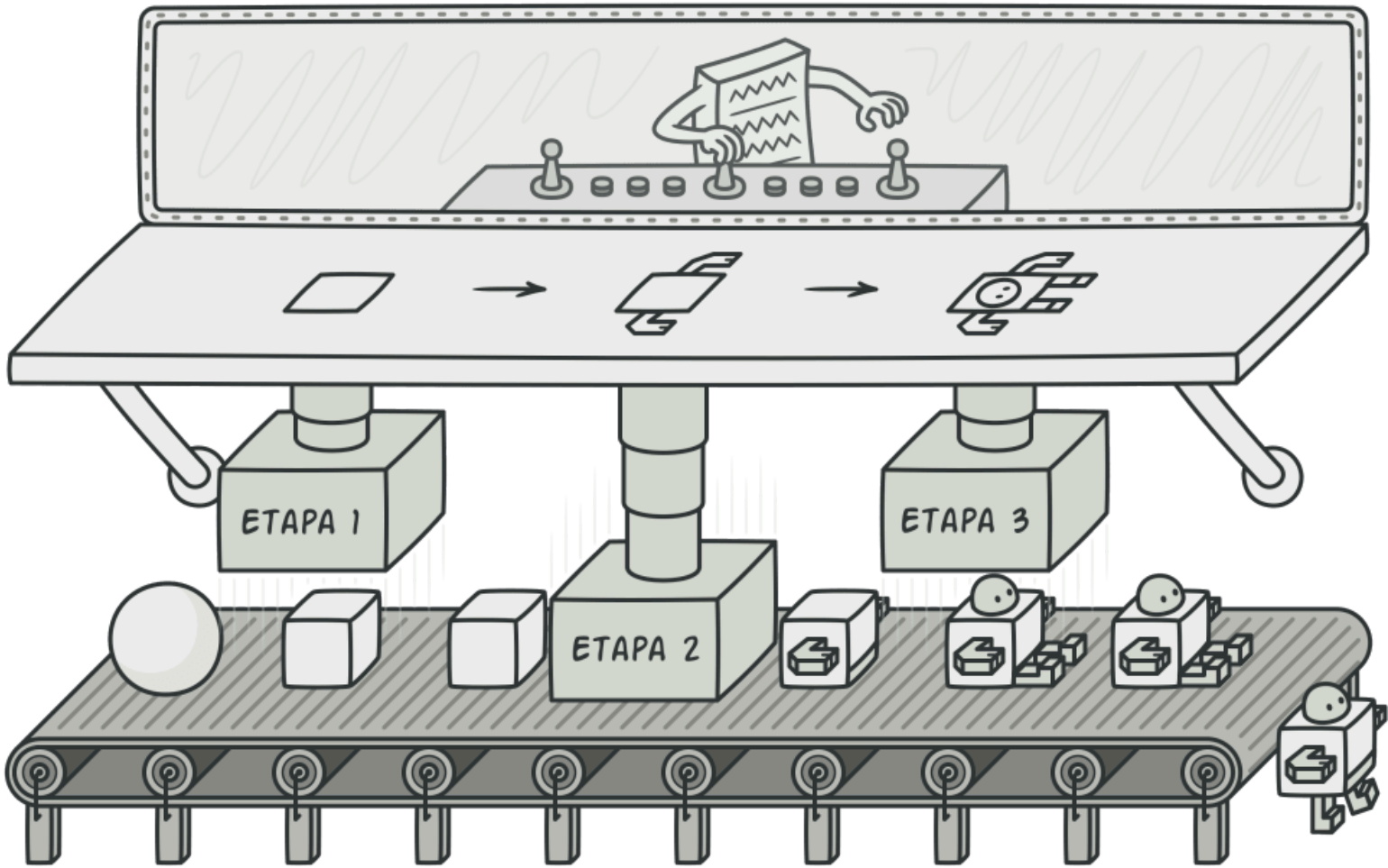
```
public class Logger {
    private static Logger instance;

    private Logger() {
        // Construtor privado para evitar instância externa
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
}
```

## Uso do Singleton

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Garantir uma única instância | Necessidade de controle global | Acesso fácil e consistente | Dificuldade de teste e acoplamento global |
| Gerenciar recurso compartilhado | Recurso que deve ser único (ex: conexão de banco) | Evita conflitos e inconsistências | Pode esconder dependências e dificultar manutenção |
| Controlar acesso a recurso | Recurso que precisa de controle de acesso (ex: logger) | Centraliza controle e configuração | Pode se tornar um "deus objeto" e acumular responsabilidades |

# Builder

Builder separa a construção de um objeto complexo de sua representação, permitindo criar diferentes representações usando o mesmo processo de construção.



# Exemplo de Builder

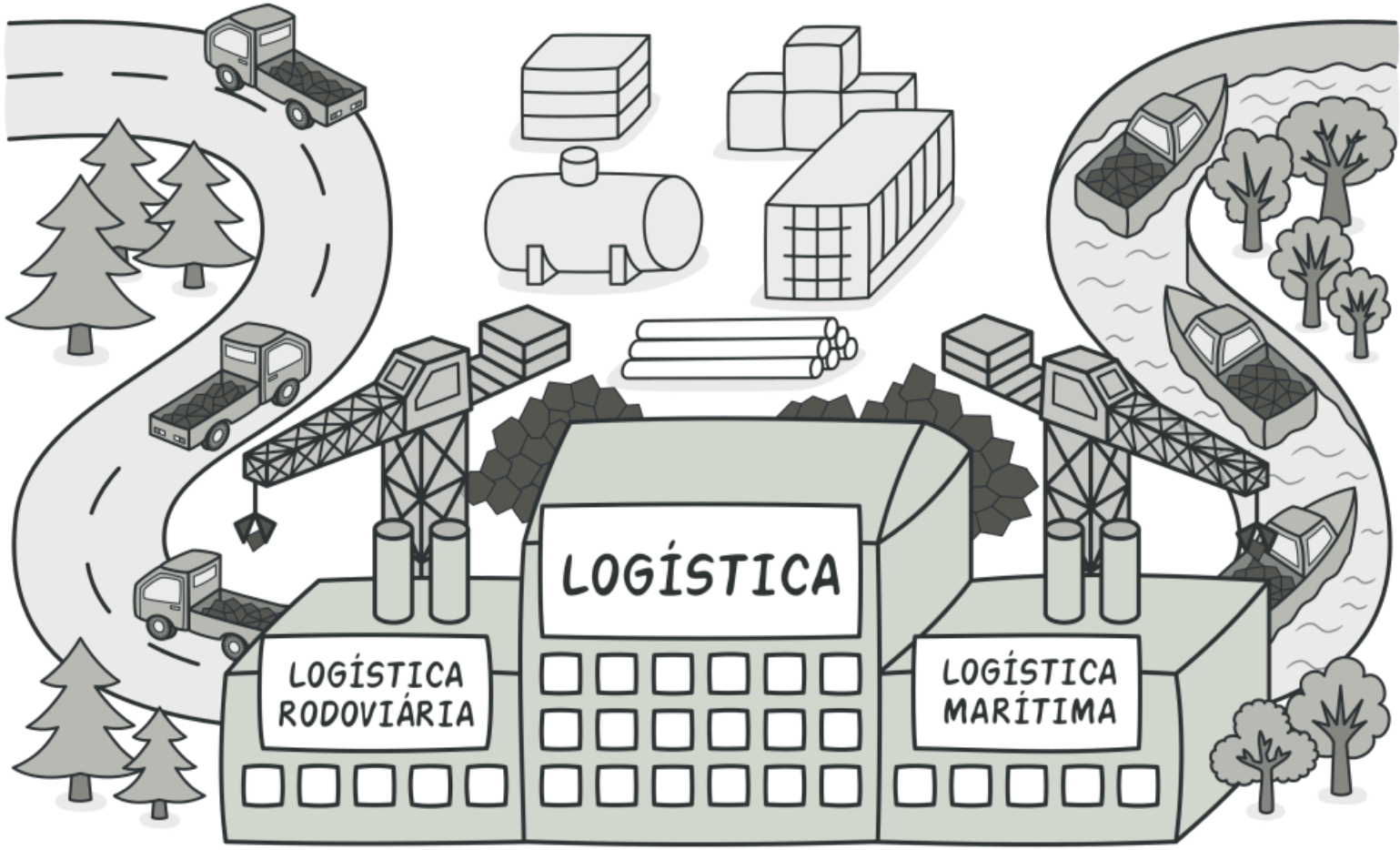
```
public class UserBuilder {
    private String name;
    private String email;

    public UserBuilder setName(String name) {
        this.name = name;
        return this;
    }

    public UserBuilder setEmail(String email) {
        this.email = email;
        return this;
    }

    public User build() {
        return new User(name, age, email);
    }
}
```

## Uso do Builder

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Construção de objetos complexos | Muitos parâmetros ou passos para criar um objeto | Separação clara entre construção e representação | Pode ser verboso para objetos simples |
| Variação na construção | Diferentes formas de construir o mesmo objeto | Flexibilidade na criação de objetos | Pode introduzir complexidade desnecessária |
| Imutabilidade | Necessidade de objetos imutáveis | Facilita a criação de objetos imutáveis | Pode ser excessivo para objetos mutáveis ou simples |

# Factory Method

Factory Method define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar.



## Exemplo de Factory Method

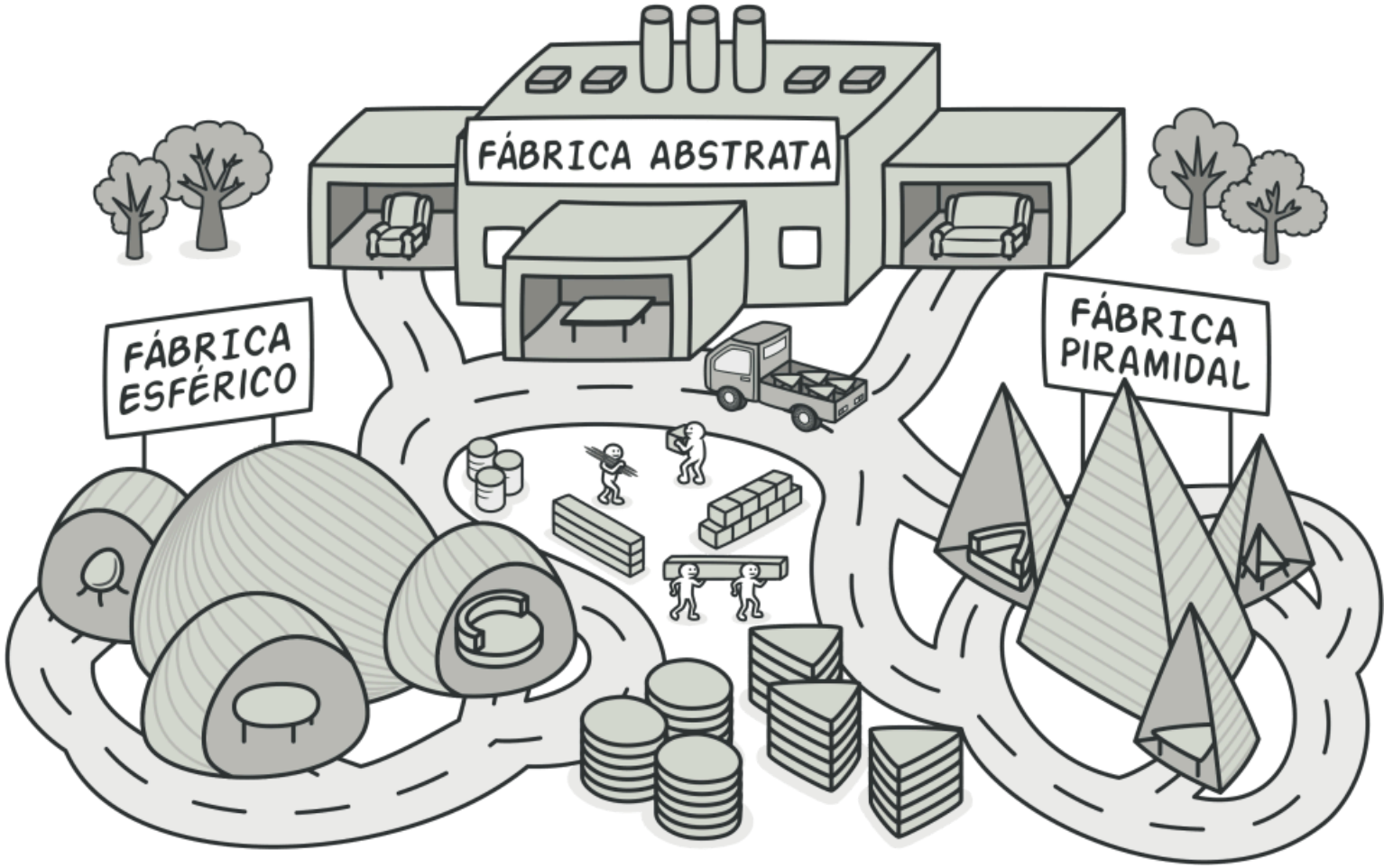
```
public class UserFactory {
  public User createUser(String name) {
    String email = name.toLowerCase() + "@example.com";
    User user = new User(
      name,
      email
    );
  }
}
```

## Uso do Factory Method

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Delegar criação para subclasses | Variação na criação de objetos | Flexibilidade para escolher a classe concreta | Pode levar a hierarquias complexas |
| Encapsular lógica de criação | Lógica de criação complexa ou variável | Centraliza a lógica de criação | Pode ser difícil de entender para iniciantes |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Abstract Factory

Abstract Factory fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.



## Exemplo de Abstract Factory

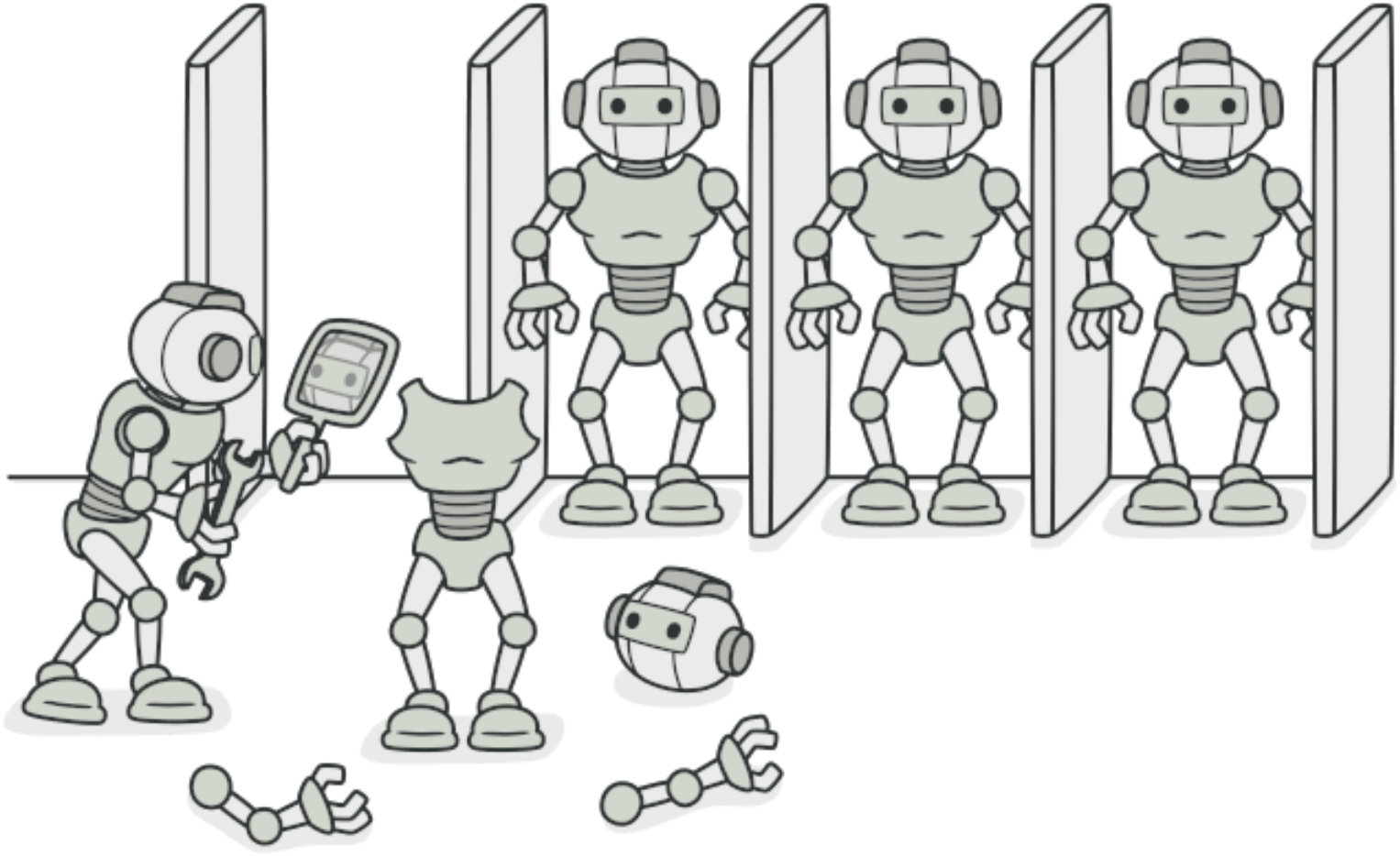
```
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

public class AndroidFactory implements GUIFactory {
    public Button createButton() {
        return new AndroidButton();
    }
    public Checkbox createCheckbox() {
        return new AndroidCheckbox();
    }
}
```

## Uso do Abstract Factory

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Criar famílias de objetos relacionados | Necessidade de criar objetos em conjunto | Garante consistência entre os objetos criados | Pode ser complexo para famílias pequenas |
| Isolar código cliente de classes concretas | Código cliente não deve conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Prototype

Prototype permite criar novos objetos clonando um modelo existente, evitando a necessidade de criar objetos do zero.



## Exemplo de Prototype

```
public class User implements Cloneable {
    private String name;
    private String email;
    private Setor setor;

    public User clone() {
        try {
            User clone = (User) super.clone();
            clone.setor = this.setor.clone(); // Clonagem profunda
            return clone;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

## Uso do Prototype

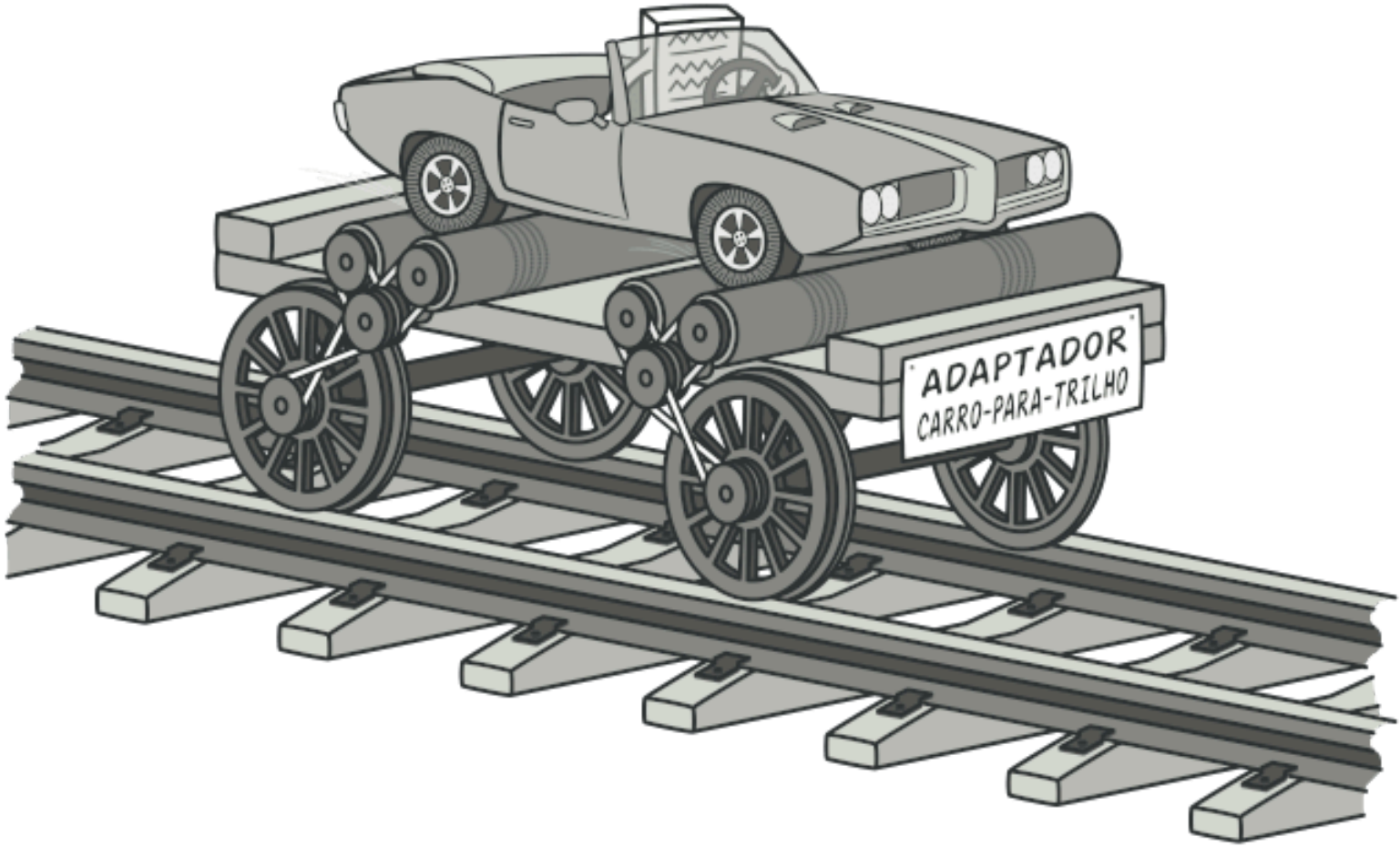
| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Criar objetos a partir de um modelo | Necessidade de criar objetos semelhantes | Permite criar objetos rapidamente a partir de um modelo | Clonagem correta pode ser difícil |
| Reduzir acoplamento com classes concretas | Código cliente não deve conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a criação de variáveis | Variáveis pré-configuradas para diferentes cenários | Permite criar variáveis a partir de modelos | Pode resultar em código mais verboso |

# Padrões Estruturais

| Padrão | Quando pensar nele | Cuidado principal |
|---|---|---|
| Adapter | interfaces incompatíveis | virar remendo permanente |
| Bridge | duas dimensões de variação | abstração desnecessária |
| Composite | árvore parte-todo | interface genérica demais |
| Decorator | camadas combináveis | ordem e depuração |
| Facade | subsistema complexo | fachada inchada |
| Proxy | controlar acesso | esconder custo real |
| Flyweight | muitos objetos parecidos | otimização prematura |

# Adapter

Adapter permite que classes com interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra que o cliente espera.



## Exemplo de Adapter

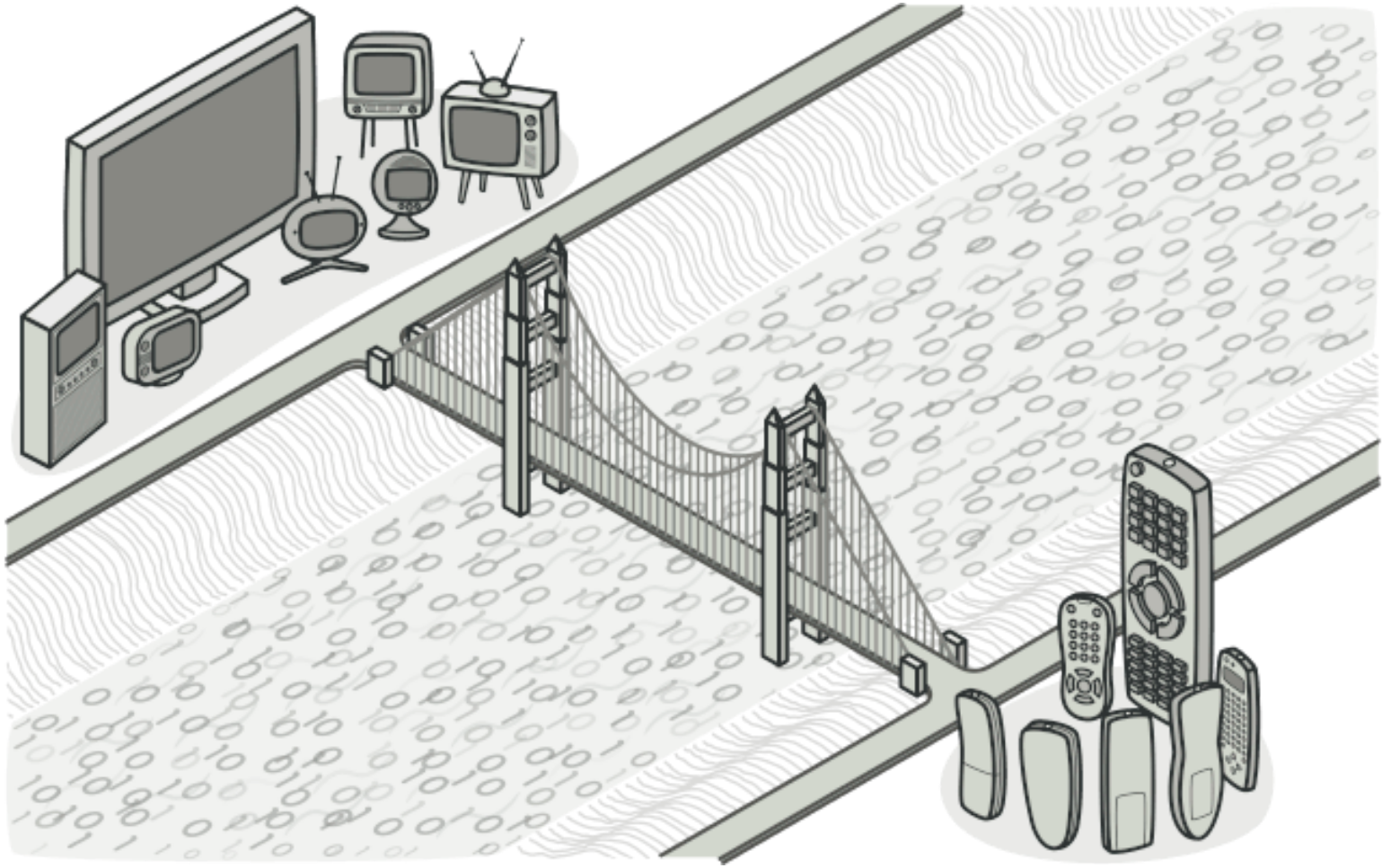
```
public interface API {
    void request();
}

public class APIAdapter implements API {
    private LegacyAPI legacyAPI;

    public APIAdapter(LegacyAPI legacyAPI) {
        this.legacyAPI = legacyAPI;
    }

    public void request() {
        legacyAPI.oldRequest();
    }
}
```

## Uso do Adapter

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Integrar interfaces incompatíveis | Necessidade de usar uma classe com uma interface diferente | Permite reutilizar código existente | Pode se tornar um "remendo" permanente e dificultar manutenção |
| Isolar código cliente de mudanças | Mudanças em uma classe não devem afetar o cliente | Facilita a manutenção e extensão | Pode introduzir complexidade desnecessária |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Bridge

Bridge separa a abstração da implementação, permitindo que ambas variem independentemente.



## Exemplo de Bridge

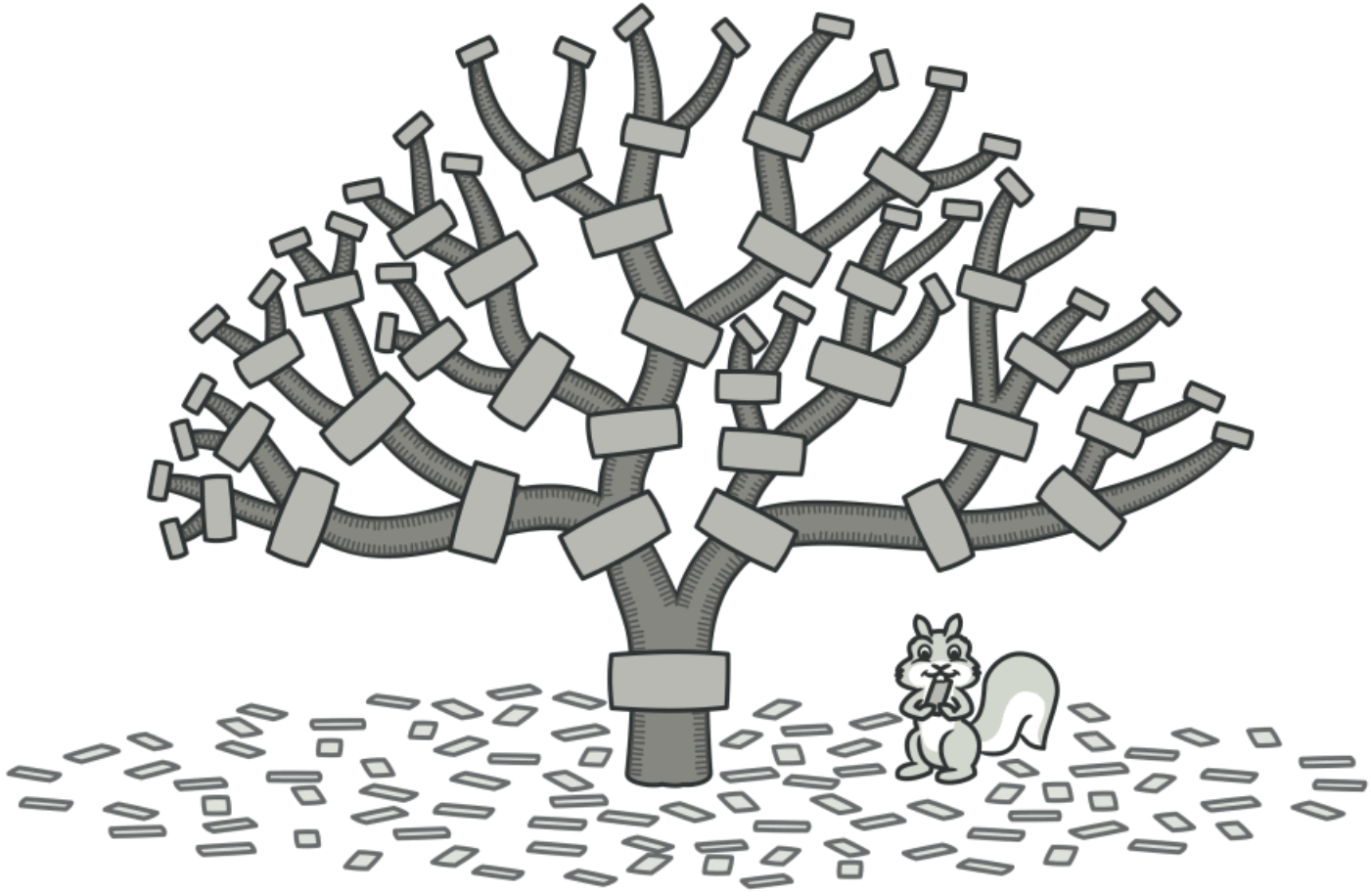
```
public interface GeradorFotos {
    void gerarFoto();
}

public class GeradorFotosAltaQualidade implements GeradorFotos {
    public void gerarFoto() {
        // implementação para alta qualidade
    }
}
```

## Uso do Bridge

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Separar abstração da implementação | Duas dimensões de variação (ex: formato e plataforma) | Permite variar ambas as dimensões independentemente | Pode introduzir complexidade desnecessária |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a manutenção | Mudanças em uma dimensão não afetam a | Facilita a manutenção e extensão | Pode resultar em código mais |

# Composite

Composite permite compor objetos em estruturas de árvore para representar hierarquias parte-todo, permitindo que clientes tratem objetos individuais e composições de forma uniforme.



# Exemplo de Composite

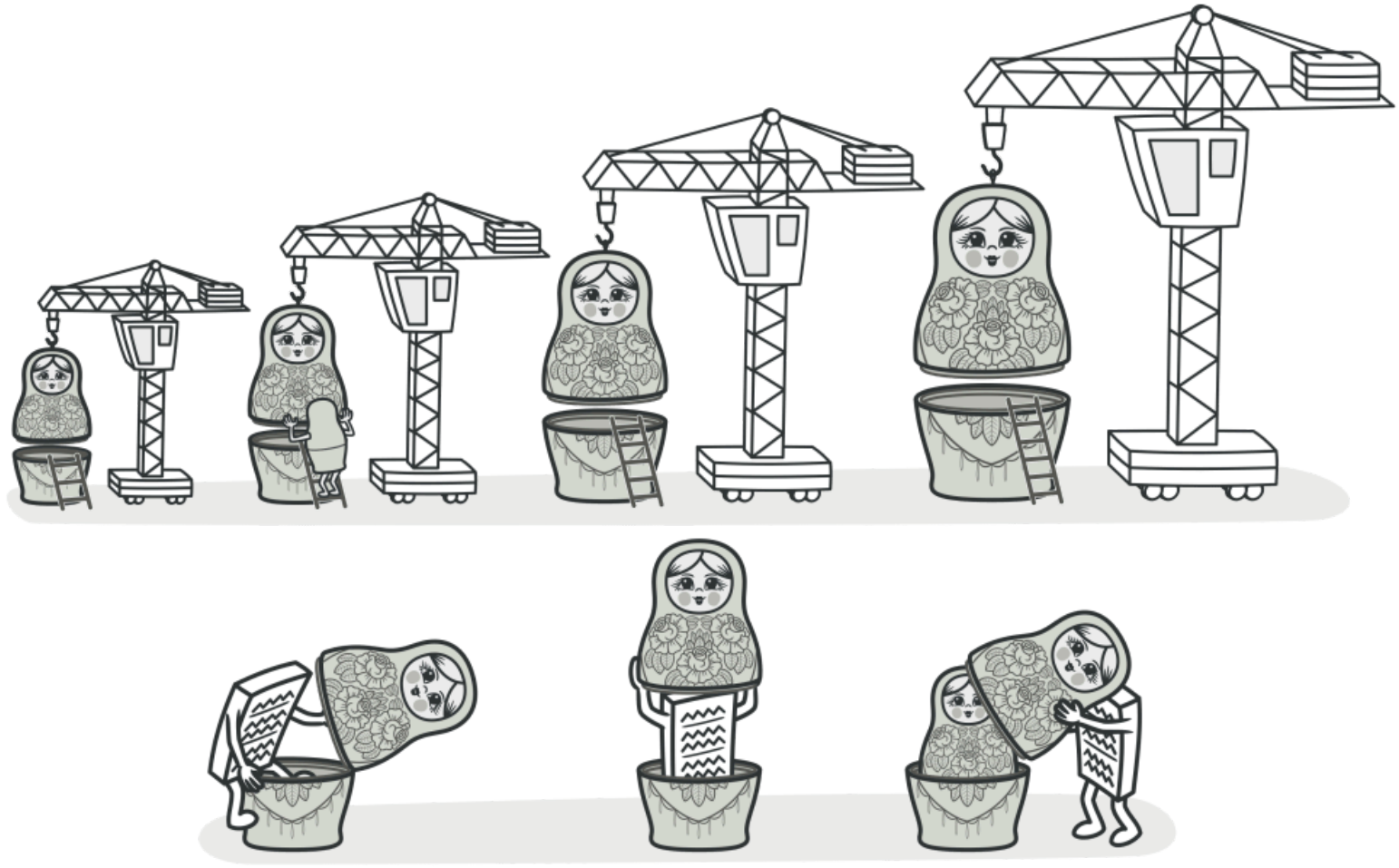
```
public interface Pedido {
    void calculaPreco();
}

public class Caixa implements Pedido {
    public void calculaPreco() {
        // calcular preço dos itens
    }
}

public class Container implements Pedido {
    private List<Pedido> caixas;
    public void calculaPreco() {
        for (Caixa caixa : caixas) {
            caixa.calculaPreco();
        }
    }
}
```

## Uso do Composite

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Representar hierarquias parte-todo | Estrutura hierárquica de objetos (ex: arquivos e pastas) | Permite tratar objetos individuais e composições de forma uniforme | Pode resultar em uma interface genérica demais e difícil de usar |
| Promover baixo acoplamento | Cliente não conhece classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a manutenção | Mudanças em um componente afetam outros | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Decorator

Decorator permite adicionar responsabilidades a objetos de forma dinâmica, sem alterar sua estrutura.



# Exemplo de Decorator

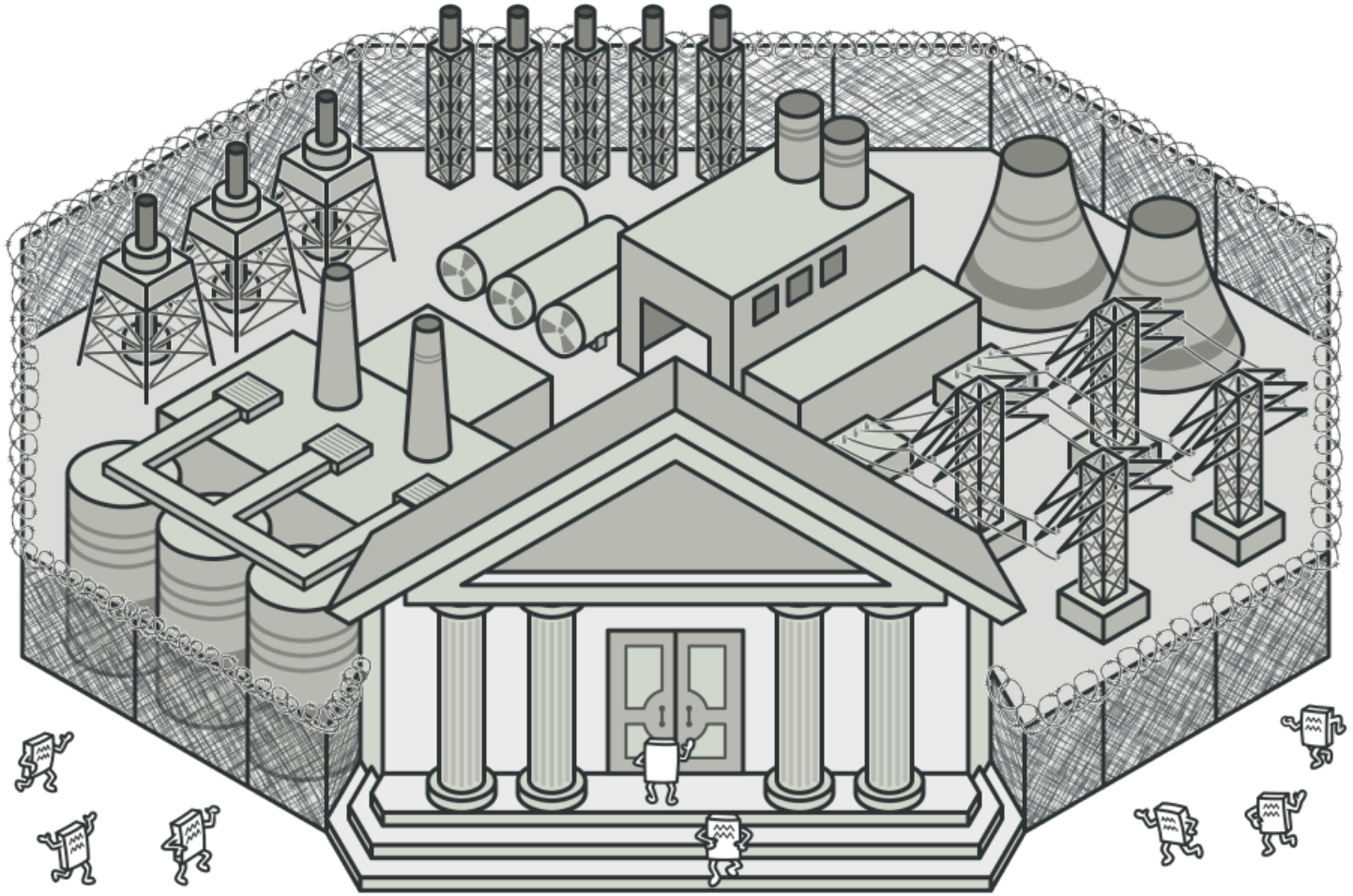
```
public class PedidoSimples implements Pedido {
    public void calculaPreco() {
        // calcular preço básico
    }
}

public class PedidoComDesconto implements Pedido {
    private Pedido pedido;
    public PedidoComDesconto(Pedido pedido) {
        this.pedido = pedido;
    }
    public void calculaPreco() {
        pedido.calculaPreco();
        // aplicar desconto
    }
}
```

## Uso do Decorator

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Adicionar responsabilidades dinamicamente | Necessidade de adicionar funcionalidades a objetos | Permite adicionar funcionalidades de forma flexível e combinável | Pode resultar em uma grande quantidade de classes |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a manutenção | Mudanças em um decorador não afetam outros | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Facade

Facade fornece uma interface unificada para um conjunto de interfaces em um subsistema, facilitando o uso do subsistema.

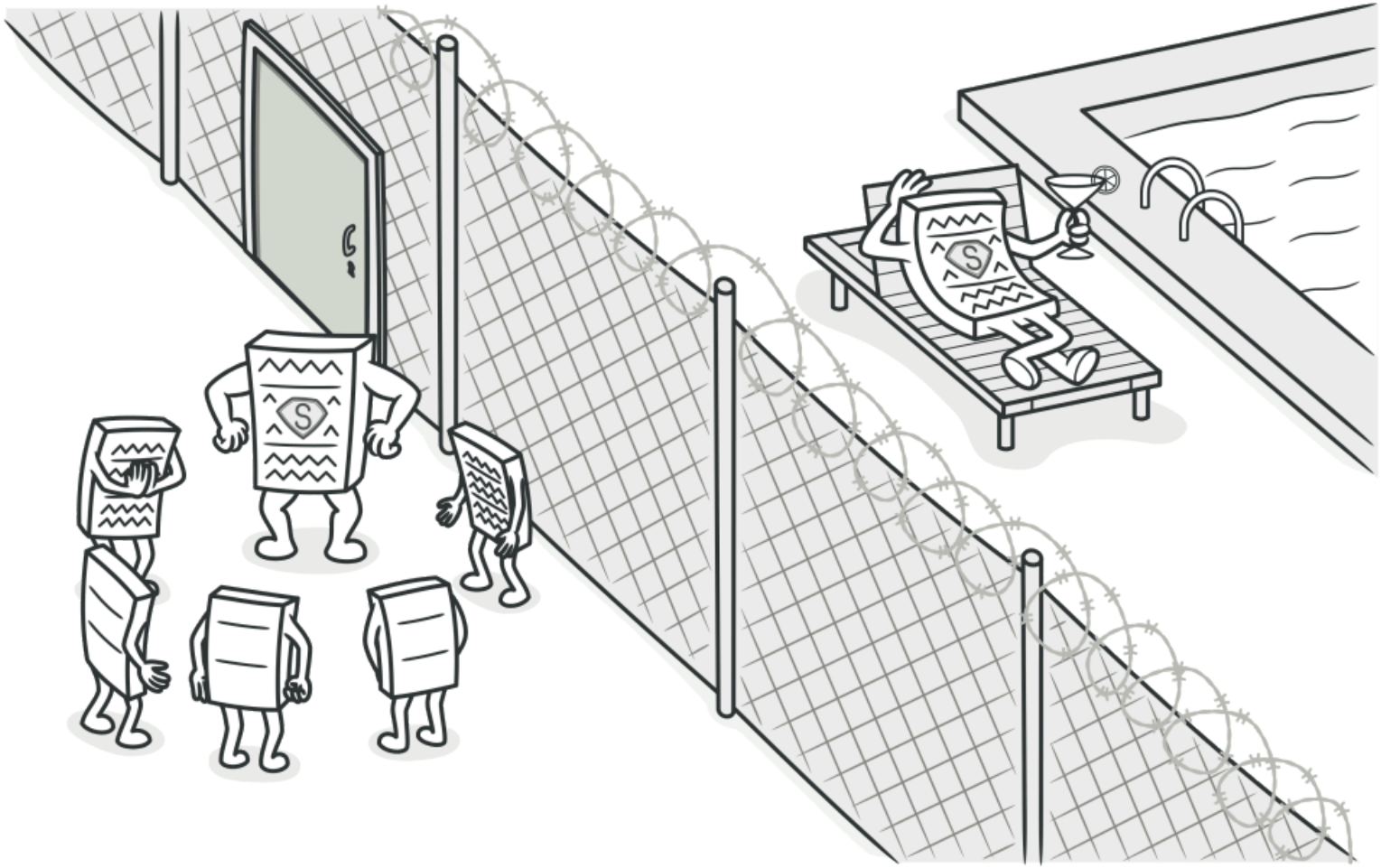


## Exemplo de Facade

```
public class SistemaPagamentoFacade {
    public void processarPagamento() {
        banco.conectar();
        banco.autenticar();
        banco.transferir();
        banco.desconectar();
    }
}
```

# Proxy

Proxy fornece um substituto ou representante de outro objeto para controlar o acesso a ele, podendo adicionar lógica adicional antes ou depois de acessar o objeto real.



# Exemplo de Proxy

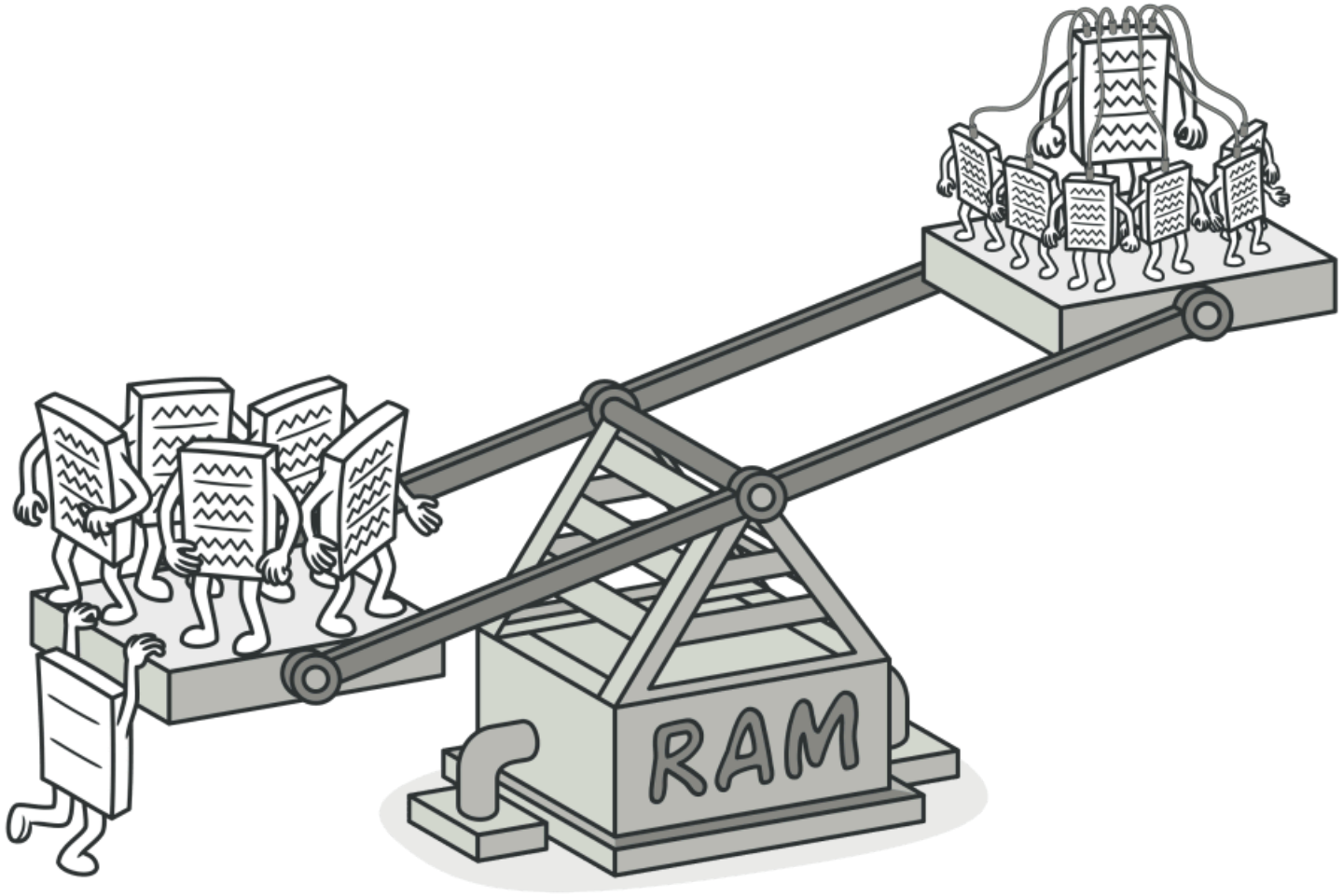
```
public class UserAPI implements API {
    public void request() {
        // implementação real
    }
}

public class UserAPIProxy implements API {
    private UserAPI userAPI;

    public UserAPIProxy(UserAPI userAPI) {
        this.userAPI = userAPI;
    }

    public void request() {
        // controle de acesso ou lógica adicional
        userAPI.request();
    }
}
```

## Uso do Proxy

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Controlar acesso a um recurso | Necessidade lógica adicional antes de acessar um recurso | Permite adicionar lógica adicional sem alterar a classe | Pode esconder o custo real da operação e dificultar a depuração |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a manutenção | Mudanças no proxy não afetam a classe real | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Flyweight

Flyweight usa compartilhamento para suportar grandes quantidades de objetos semelhantes de forma eficiente, minimizando o uso de memória.



## Exemplo de Flyweight

```
public class Mesh {
    private String tipo;
    private Texture textura;
}
```

## Uso do Flyweight

| Resolve | Causa | Vantagem | Desvantagem |
|---|---|---|---|
| Reduzir uso de memória | Muitos objetos semelhantes com dados compartilháveis | Permite compartilhar dados comuns entre objetos para economizar memória | Pode introduzir complexidade adicional para gerenciar o compartilhamento e a imutabilidade dos objetos |
| Promover baixo acoplamento | Código cliente não precisa conhecer classes concretas | Facilita a manutenção e extensão | Pode resultar em código mais verboso |
| Facilitar a manutenção | Mudanças em um flyweight não afetam outros | Facilita a manutenção e extensão | Pode resultar em código mais verboso |

# Conclusão

- Padrões são repertório, não receita fixa
- Nome do padrão importa menos que a intenção
- Escolha depende do problema real