PADRÕES E DESENHO DE SOFTWARE

ANA LOUREIRO 104063 LEI

AVALIAÇÃO: ATP1. 12 DE ABRIL
ATP2. 31 DE MAIO

PROGRAMA:

- PRINCÍPIOS DE DESENHO DE SOFTWARE
- REVISÃO E MELHORIA DO CÓDIGO
- PADRÕES DE DESENHO DE SOFTWARE
- ESTILOS DE ARQUITETURA DE SOFTWARE

CHAPT 1

INTRODUCTION TO SOFTWARE DESIGN

FASES DO DESENHO DE SOFTWARE

1 - REQUISITOS

DEVEM SER DEFINIDOS JUNTO DOS STAKEHOLDERS

- NECESSIDADES A IMPLEMENTAR NO SISTEMA

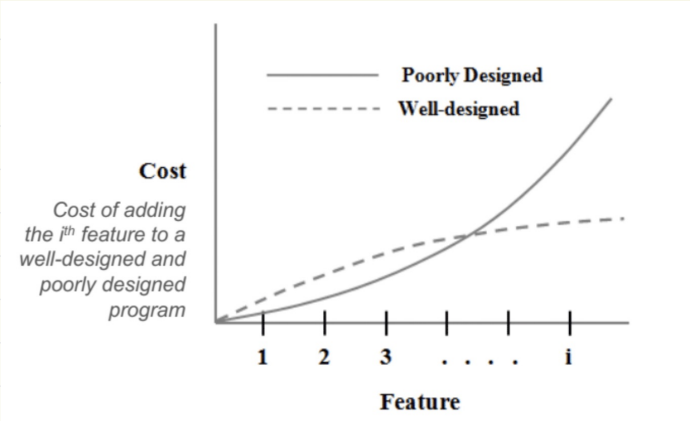
2 - DESENHO

DEFINIR AS CLASSES, MÉTODOS, RELAÇÕES, FLUXOS DE TRABALHO...

3 - CONSTRUÇÃO

ESCRITA DO CÓDIGO QUE IMPLEMENTE O SISTEMA DELINEADO

IMPORTÂNCIA DA MANUTENÇÃO DE COMPLEXIDADE



UM PROGRAMA MAL DESENHADO (DEMASIADO COMPLEXO) :

- RÁPIDO DE DESENVOLVER

PORÉM

- DIFÍCIL DE ENTENDER E MODIFICAR
- INSUSTENTÁVEL A LONGO PRAZO

DESENHO SERVE COMO CONTROLO DA COMPLEXIDADE:

- GERINDO A COMPLEXIDADE ESSENCIAL (ASSOCIADA AO PROBLEMA)
- EVITANDO A COMPLEXIDADE ACIDENTAL (ASSOCIADA À SOLUÇÃO E À FORMA COMO É IMPLEMENTADA).

COMPLEXIDADE TOTAL = COMPLEXIDADE ESSENCIAL + COMPLEXIDADE ACIDENTAL

PARA EVITAR A COMPLEXIDADE NO DESENHO DE SOFTWARE

MODULARIDADE - DIVIDIR O PROBLEMA EM PROBLEMAS MAIS PEQUENOS E FÁCEIS DE RESOLVER.
- DIVIDE AND CONQUER

ABSTRAÇÃO - DE MODO A OMITIR DETALHES ONDE NÃO SÃO NECESSÁRIOS.

OCULTAÇÃO DE INFORMAÇÃO - DESENVOLVER INTERFACES SIMPLES PARA DETALHES COMPLEXOS.

HERANÇA - REUTILIZAÇÃO DE COMPONENTES GENÉRICOS PARA DEFINIR MAIS ESPECÍFICOS

COMPOSIÇÃO - REUTILIZAR COMPONENTES PARA CRIAR UMA NOVA SOLUÇÃO.

CARACTERÍSTICAS DE SOFTWARE DESIGN

NÃO DETERMINÍSTICO - É POUCO PROVÁVEL QUE DOIS DESIGNERS OU PROCESSOS DE DESIGN GEREM O MESMO RESULTADO

HEURÍSTICO - TÉCNICAS DE DESIGN BASEIAM-SE NA HEURÍSTICA E EM REGRAS PRÁTICAS, AO INVÉS DE PROCESSOS REPETITIVOS

EMERGENTE - O RESULTADO FINAL EVOLUI DE ACORDO COM A EXPERIÊNCIA E O FEEDBACK

PROCESSO GENÉRICO DE DESIGN

1. PERCEBER O PROBLEMA (SOFTWARE REQUIREMENTS)
2. CRIAR UM MODELO-SOLUÇÃO (SYSTEM SPECIFICATION) (USE CASES)
3. PROCURAR POR SOLUÇÕES EXISTENTES QUE CUBRAM ALGUNS OU TODOS OS PROBLEMAS IDENTIFICADOS
4. CONSTRUIR PROTÓTIPOS
5. REVER E DOCUMENTAR O DESIGN
6. ITERAR SOBRE A SOLUÇÃO (REFACTOR)

PARA ASSEGURAR TODO ESTE PROCESSO É NECESSÁRIO O CONHECIMENTO DE:

- REQUISITOS DO UTILIZADOR
- DOMÍNIO EM QUE O SISTEMA DE INSERE (TERMOS TÉCNICOS, CONCEITOS)
- AMBIENTE DE EXECUÇÃO EM QUE O PROGRAMA VAI SER IMPLEMENTADO (AS SUAS CAPACIDADES E LIMITAÇÕES).

DESIRABLE INTERNAL DESIGN CHARACTERISTICS

- **Minimal complexity** – Keep it simple. Maybe you don't need high levels of generality.
- **Loose coupling** – minimize dependencies between modules
- **Ease of maintenance** – Your code will be read more often then it is written.
- **Extensibility** – Design for today but with an eye toward the future. Note, this characteristic can be in conflict with "minimize complexity". Engineering is about balancing conflicting objectives.
- **Reusability** – reuse is a hallmark of a mature engineering discipline
- **Portability** – works or can easily be made to work in other environments
- **High fan-in** on a few utility-type modules and low-to-medium fan-out on all modules. High fan-out is typically associated with high complexity.
- **Leanness** – when in doubt, leave it out. The cost of adding another line of code is much more than the few minutes it takes to type.
- **Stratification** – Layered. Even if the whole system doesn't follow the layered architecture style, individual components can.
- **Standard techniques** – sometimes it's good to be a conformist! Boring is good. Production code is not the place to try out experimental techniques.

CHAPT 2

GRASP

GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS

RESPONSABILITY-DRIVEN DESIGN (RDD)

CONSISTE NA DEFINIÇÃO DOS OBJETOS EM TERMOS DE RESPONSABILIDADE, PAPÉIS E COLABORAÇÕES

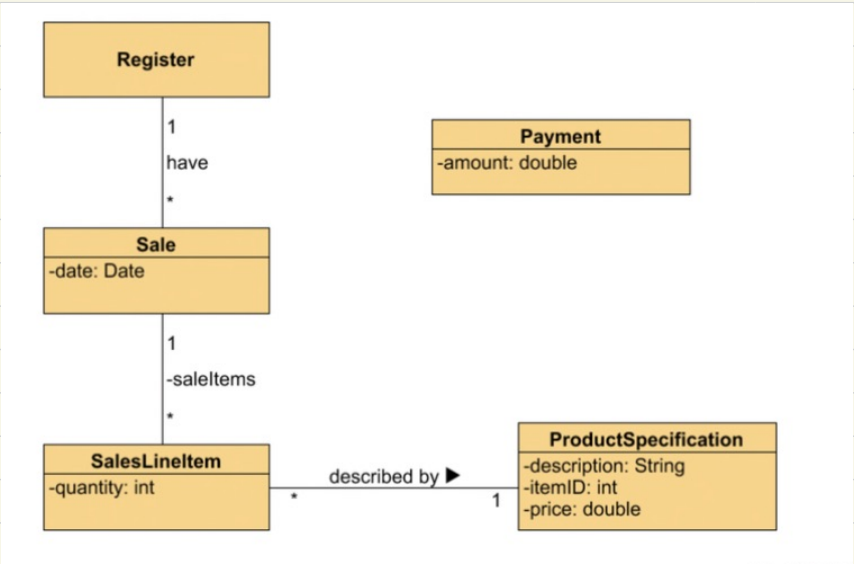
PODEM SER GENERALIZADAS EM
FAZER (EX. CRIAR UM OBJETO),
SABER (EX. OS SEUS DADOS PRIVADOS, OBJETOS RELACIONADOS)

PRINCIPIOS:

- CREATOR
- INFORMATION EXPERT
- LOW COUPLING
- HIGH COHESION
- CONTROLLER
- POLYMORPHISM
- PURE FABRICATION
- INDIRECTION
- PROTECTED VARIATIONS

EXEMPLO A SER USADO:

POS: POINT OF SALE



NOTA: COUPLING VEM DE O QUÃO LIGADO, INFORMADO OU DEPENDENTE ESTÁ DE OUTROS ELEMENTOS DE OUTRAS CLASSES

CLASSES COM STRONG COUPLING (BAD) :

- SOFREM DE MUDANÇAS EM CLASSES RELACIONADAS
- SÃO MAIS DIFÍCEIS DE PERCEBER E MANTER
- SÃO MAIS DIFÍCEIS DE REUTILIZAR

PORÉM, COUPLING É NECESSÁRIO PARA AS CLASSES COMUNICAREM ENTRE SI

- O PROBLEMA É DEMASIADO, E/OU CLASSES DEMASIADO INSTÁVEIS

CREATOR

-> FAZER

PROBLEMA: QUEM CRIA UMA INSTÂNCIA DE 'A'

SOLUÇÃO: DAR A 'B' A RESPONSABILIDADE DE INSTANCIAR 'A' SE PELO MENOS UM DOS SEGUINTES CRITÉRIOS SE APLICA

- 'B' CONTÉM OU CONTÉM UM CONJUNTO DE 'A'
- 'B' REGISTA 'A'
- 'B' CONCLUI UTILIZANDO 'A'
- 'B' TEM A INITIALISING DATA DE 'A'

VANTAGEM: AO FAZER INSTÂNCIAS DE UMA CLASSE RESPONSÁVEL POR CRIAR OBJETOS QUE VÃO REFERENCIAR

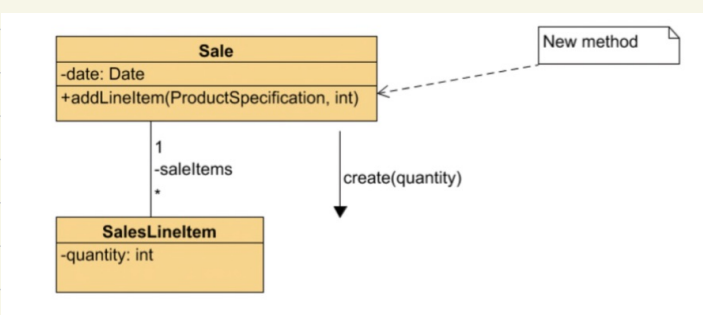
- LOW COUPLING

DESVANTAGEM: CRIAÇÃO PODE REQUERER ALTA COMPLEXIDADE

EXEMPLO:

- QUEM CRIA SALES_LINE_ITEM OBJECTS, POR ID E QUANTIDADE?

SALE AGREGA UMA QUANTIDADE DE SALE_ITEMS



INFORMATION EXPERT

-> SABER
-> FAZER

PROBLEMA: QUAL O PRINCÍPIO PARA ATRIBUIR RESPONSABILIDADES A OBJETOS?

SOLUÇÃO: ATRIBUIR A RESPONSABILIDADE À CLASSE QUE TEM A INFORMAÇÃO NECESSÁRIA

'NÃO FAÇAS NADA QUE PODES ATRIBUIR A OUTRA PESSOA".

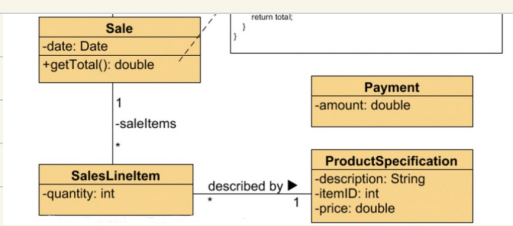
VANTAGENS: CADA CLASSE UTILIZA A INFORMAÇÃO QUE DISPÕE PARA CUMPRIR TAREFAS, NÃO DEPENDENDO DE OUTRAS CLASSES

- COESÃO
- ENCAPSULAMENTO DE INFORMAÇÃO
- LOW COUPLING

DESVANTAGEM: PODE FAZER COM QUE UMA CLASSE SE TORNE DEMASIADO COMPLEXA (EX: CADA EXPERT PRECISARIA DE SER GUARDADO NUMA DATABASE)

EXEMPLO: QUEM É RESPONSÁVEL POR SABER O TOTAL DE UMA VENDA?

- PRECISA DE TODAS AS INSTÂNCIAS DE SALES_LINE_ITEM E OS SEUS SUBTOTALS -> SALE É O INFORMATION EXPERT



PORÉM, OS SUBTOTAIS TAMBÉM SÃO NECESSÁRIOS PARA CADA ITEM

- EXISTINDO ENTÃO 3 EXPERTS

Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

LOW COUPLING PATTERN

PROBLEMA: COMO REDUZIR O IMPACTO DA MUDANÇA E ENCORAJAR A REUTILIZAÇÃO

SOLUÇÃO: ATRIBUIR A RESPONSABILIDADE PARA QUE O ACOPLAMENTO (LINKING CLASSES) SE MANTENHA REDUZIDO. EVITAR QUE UMA CLASSE REFERENCIE MUITAS OUTRAS

VANTAGENS: TORNA O CÓDIGO MAIS FÁCIL DE:

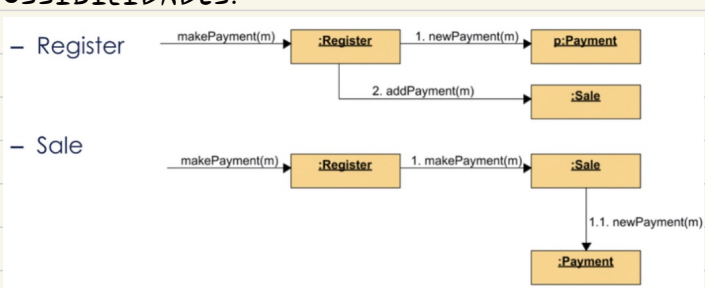
- DE ALTERAR/MANTER
- ENTENDER
- REUTILIZAR

FORMAS DE COUPLING DE 'A' PARA 'B' SÃO:

- 'A' TEM UM ATRIBUTO QUE REFERÊNCIA UMA INSTÂNCIA DE 'B', OU 'B' EM SI
- 'A' TEM UM MÉTODO QUE REFERÊNCIA UMA INSTÂNCIA DE 'B' OU 'B' EM SI
- 'A' É UMA SUBCLASSE (DIRETA OU INDIRETA) DE 'B'
- 'B' É UMA INTERFACE, E 'A' IMPLEMENTA ESSA INTERFACE

EXEMPLO: QUEM TEM A RESPONSABILIDADE DE CRIAR O PAYMENT

DUAS POSSIBILIDADES:



SALE É MELHOR POIS SALE PRECISA DE PAYMENT (SABE O SEU TOTAL)

HIGH COHESION

PROBLEMA: COMO MANTER CLASSES FOCADAS, PERCEPTÍVEIS E MALEÁVEIS

SOLUÇÃO: ATRIBUIR RESPONSABILIDADES DE FORMA A MANTER UMA ALTA COESÃO

> A COESÃO DIMINUI COM O AUMENTO DA QUANTIDADE DE CÓDIGO E DA DIVERSIDADE DE FUNCIONALIDADES

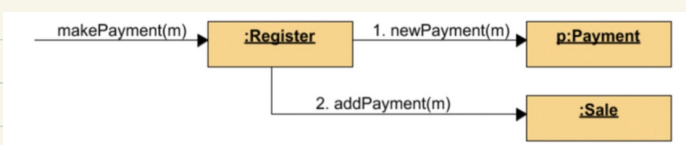
VANTAGENS:

- ALTA COESÃO -> LOW COUPLING
- FACILITA A COMPREENSÃO E MANUTENÇÃO

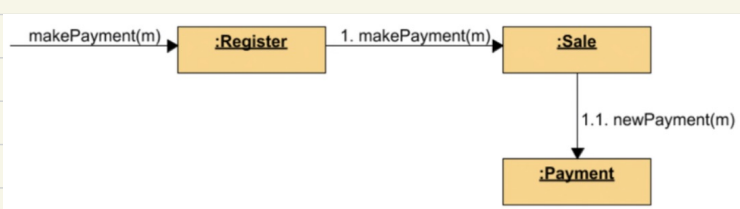
DESVANTAGEM: ÀS VEZES É NECESSÁRIO CRIAR OBJETOS POUCO COESOS, EM COMUNICAÇÕES E OBJETOS REMOTOS, EM QUE É NECESSÁRIO CRIAR UMA INTERFACE PARA VÁRIAS OPERAÇÕES

EXEMPLO:

- REGISTER TOMARIA MAIS E MAIS RESPONSABILIDADES E TORNAR-SE-IA MENOS COESO



- DAR A RESPONSABILIDADE A SALE SUPORTA ALTA COESÃO EM REGISTER, TAL COMO LOW COUPLING



CONTROLLER

PROBLEMA: QUEM É RESPONSÁVEL POR UI EVENTS

SOLUÇÃO: SE RECEBER UM EVENTO QUE NÃO VENHA DO GUI, CRIAR UMA EVENT CLASS QUE SEPARE A ORIGEM DOS EVENTOS DOS OBJETOS QUE LIDAM COM ELES

A RESPONSABILIDADE DE LIDAR COM SYSTEM EVENTS DEVE SER ATRIBUÍDA À CLASSE QUE:

- REPRESENTE O MODELO DE NEGÓCIO OU OVERALL SYSTEM (FAÇADE CONTROLLER)
- SEJA ARTIFICIAL, SEGUINDO O MÉTODO PURE FABRICATION (CASE CONTROLLER)

O CONTROLLER ENCAMINHA:

- OUTPUT REQUESTS
- EVENT HANDLING REQUESTS

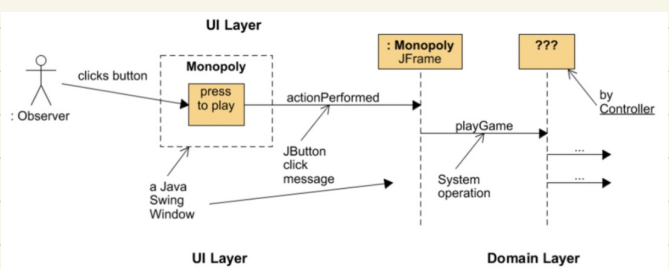
VANTAGENS:

- POTENCIAL ELEVADO DE REUTILIZAÇÃO

(O CONTROLLER FAZ COM QUE O EXTERNAL EVENT SOURCE E INTERNAL EVENT HANDLER SE TORNEM INDEPENDENTES)

- GARANTE QUE AS OPERAÇÕES OCORREM NA ORDEM CERTA

EXEMPLO: NUM JOGO DE MONOPÓLIO COM INTERFACE VISUAL INTERATIVA, QUANDO O UTILIZADOR PRESSIONAR O BOTÃO PARA INICIAR UM NOVO
JOGO, ESTE EVENTO DEVE SER GERIDO POR UMA CLASSE DO TIPO CONTROLLER, E NÃO DIRETAMENTE PELO JOGO.



POLYMORPHISM

PROBLEMA: COMO GERIR COMPORTAMENTO COM BASE NO TIPO (EX. CLASSE) MAS SEM INSTRUÇÕES CONDICIONAIS? (IF/ELSE, SWITCH)

SOLUÇÃO: QUANDO O COMPORTAMENTO ALTERA BASEADO NO TIPO DE OBJETO, UTILIZAR O POLYMORPHIC METHOD PARA SELECIONAR O COMPORTAMENTO, AO INVÉS DE USAR IF/ELSE PARA TESTAR O TIPO DO OBJETO.

> POLYMORPHIC METHOD: ATRIBUIR O MESMO NOME A SERVIÇOS (MÉTODOS) DISTINTOS EM CLASSES DIFERENTES.

VANTAGENS:

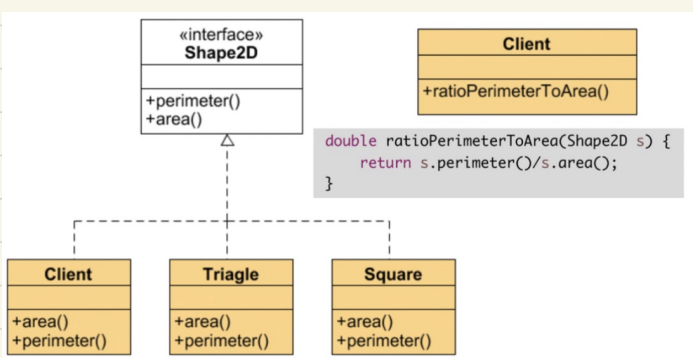
- CÓDIGO MAIS ROBUSTO E FÁCIL DE ESCREVER
- CÓDIGO MAIS ADAPTÁVEL A EXTENSÃO

DESVANTAGENS:

- AUMENTA O NÚMERO DE CLASSES
- MAIOR DIFICULDADE DE COMPREENSÃO DO CÓDIGO

EXEMPLO:

PARA EVITAR QUE UMA FUNÇÃO LIDE COM TODAS A NECESIDADE DE DETERMINAR O TIPO DE FORMA GEOMÉTRICA, UTILIZA-SE O POLIMORFIMO.
ASSIM TODAS AS CLASSES QUE REPRESENTAM UMA FORMA GEOMÉTRICA IMPLEMENTAM UMA INTERFACE COMUM, QUE DETERMINA OS MÉTODOS QUE ESTAS VÃO IMPLEMENTAR.



PURE FABRICATION

PROBLEMA: QUE OBJETO DEVE ASSUMIR UMA RESPOSABILIDADE QUANDO NENHUMA DAS CLASSES DO PROBLEMA A PODE ASSUMIR SEM VIOLAR OS PRINCÍPIOS DO LOW COUPLING E HIGH COESION?

NEM TODAS AS RESPONSABILIDADES SE ENQUADRAM NO DOMÍNIO DAS CLASSES (EX. COMUNICAÇÕES, INTERAÇÃO COM O UTILIZADOR)

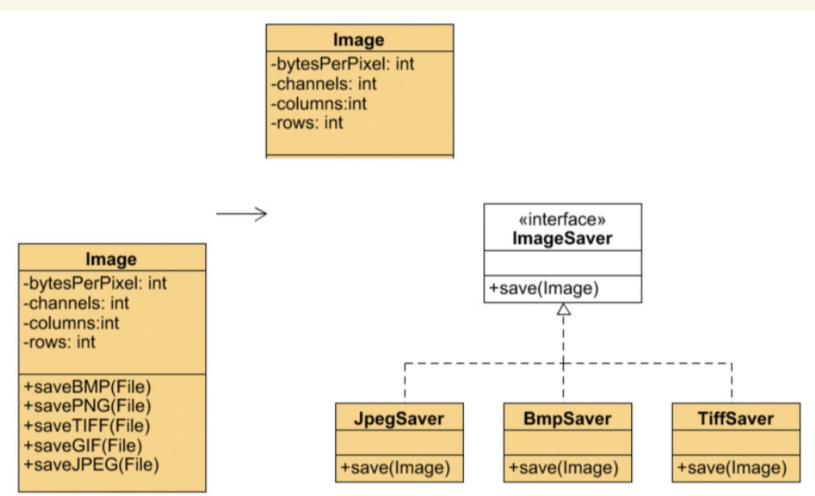
SOLUÇÃO: ATRIBUIR UM CONJUNTO COESO DE RESPONSABILIDADES A UMA CLASSE ARTIFICIAL QUE NÃO REPRESENTA NADA NO DOMÍNIO NO PROBLEMA.

VANTAGENS:

- SUPORTA COESÃO ALTA (AS RESPONSABILIDADES SÃO ATRIBUÍDAS A UMA CLASSE QUE SO SE FOCA NUM SET ESPECIFICO DE FUNÇÕES)
- POTENCIAL DE REUTILIZAÇÃO ELEVADO

EXEMPLO:

UMA CLASSE IMAGEM, SEGUNDO ESTE PRINCÍPIO DEVE SER ABSTRAÍDA DAS OPERAÇÕES DE GUARDAR EM DIFERENTES FORMATOS.



INDIRECTION

PROBLEMA:

- COMO EVITAR DIRECT COUPLING
- COMO DAR DE-COUPLE A OBJETOS PARA QUE SEJA SUPORTADO LOW COUPLING, E AUMENTAR O POTENCIAL DE REUTILIZAÇÃO

SOLUÇÃO: ATRIBUIR A RESPONSABILIDADE A UM OBJETO INTERMÉDIO

VANTAGENS:

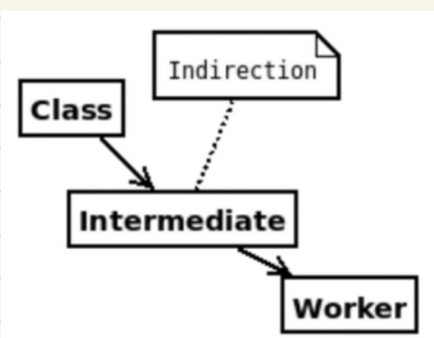
- LOW COUPLING
- PROMOVE REUTILIZAÇÃO

PODE-SE CATEGORIZAR:

- BEHAVIOURAL (EXTENSION)
- INTERFACE (MODIFICATION)
- TECHNOLOGY (ENCAPSULATION)
- COMPLEXITY (ENCAPSULATION)

EXEMPLO:

NECESSIDADE DE ESTABELECER DE UM CANAL DE COMUNICAÇÃO COM UM SISTEMA DE PAGAMENTO PARA VALIDAR UMA TRANSAÇÃO. PARA TAL, DEVE SER CRIADA UMA CLASSE RESPONSÁVEL POR ESTA OPERAÇÃO.



POLYMORPHISM

PROBLEMA: COMO DESENHAR SOFTWARE DE FORMA A QUE AS SUAS VARIAÇÕES NÃO TENHAM UM IMPACTO NEGATIVO NOUTROS ELEMENTOS?

SOLUÇÃO: IDENTIFICAR OS PONTOS DE INSTABILIDADE E ATRIBUIR RESPONSABILIDADES DE FORMA A CRIAR UMA INTERFACE ESTÁVEL EM SUA VOLTA.

É UM PRINCÍPIO DE DESIGN FUNDAMENTAL QUE SERVE DE BASE A MUITOS PADRÕES DE DESIGN

LISKOV SUBSTITUTION PRINCIPLE (LSP)

PRINCÍPIO COM BASE NO PROTECTED VARIATIONS, DEFENDE QUE SENDO 'B' UMA SUBCLASSE DE 'A', OS OBJETOS DE 'A' DEVEM PODER SER SUBSTITUÍDOS POR 'B' SEM ALTERAR A EXECUÇÃO NORMAL DO PROGRAMA:

- 'B' NÃO DEVE REMOVER MÉTODOS IMPLEMENTADOS EM 'A'
- CADA MÉTODO DE 'B' QUE REESCREVA UM DEFINIDO EM 'A' DEVE SER MAIS ESPECIFICADO DO QUE O MÉTODO DE 'A'

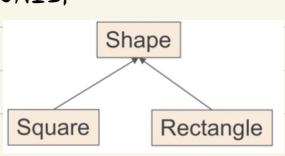
EXEMPLO:
TODOS OS QUADRADOS SÃO RETÂNGULOS? CLASS SQUARE EXTENDS RECTANGLE

MAS SERÁ QUE É UM VERDADEIRO SUBTYPE DE RETÂNGULO?
(CONSEGUE SUBSTITUIR QUANDO SE PEDE UM RETÂNGULO?)

NÃO -> RETÂNGULOS TEM COMPRIMENTO E LARGURA INDEPENDENTES
O QUADRADO VIOLA ESSA EXPECTATIVA (SURPRISE CLIENTS)

E O OPOSTO?

NÃO -> QUADRADOS TEM COMPRIMENTO E LARGURA IGUAIS,
O RETÂNGULO VIOLA ESSA EXPECTATIVA



SOLUÇÃO: NÃO OS RELACIONAR

LAW OF DEMETER (DON'T TALK TO STRANGERS)

PROBLEMA: COMO EVITAR SABER SOBRE A ESTRUTURA DE OBJETOS INDIRETOS

SOLUÇÃO:

- SE DUAS CLASSES NÃO TEM RAZÃO PARA ESTAR LIGADOS, ENTÃO NÃO DEVEM INTERAGIR DIRETAMENTE

- DENTRO DE UM MÉTODO, MENSAGENS SÓ DEVEM SER MANDADOS AOS SEGUINTES OBJETOS:
  - .THIS OBJECT (OU SELF)
  - UM PARÂMETRO DO MÉTODO
  - UM ATRIBUTO DE SELF
  - UM ELEMENTO DE UMA COLEÇÃO QUE É UM ATRIBUTO DE SELF
  - UM OBJETO CRIADO DENTRO DO MÉTODO

VANTAGENS:

- MANTÉM LOW COUPLING ENTRE CLASSES
- TORNA O DESIGN MAIS ROBUSTO

DESVANTAGEM: OS MÉTODOS INDIRETOS PODEM LEVAR AO AUMENTO DE OVERHEAD

EXEMPLO: NUMA EMPRESA DIVIDIDA EM DEPARTAMENTOS, CADA UM COM UM GESTOR, QUE É UMA INSTÂNCIA DA CLASSE EMPREGADO, SE A EMPRESA QUISER SABER O TOTAL DE DINHEIRO QUE PAGA AOS GESTORES NÃO DEVE INVOCAR UM MÉTODO NO EMPEGADO, MAS SIM NO DEPARTAMENTO, UMA VEZ QUE ESTE É SEU ATRIBUTO (O EMPREGADO NÃO!)

CHAPT 3

DESIGN PATTERNS

GENERAL CONCEPTS

DESIGN PATTERN

SÃO PRINCÍPIOS E SOLUÇÕES TÍPICAS PARA PROBLEMAS COMUNS EM SOFTWARE DESIGN

FUNCIONAM COMO PLANTAS PRÉ-FEITAS QUE SE PODEM ADAPTAR PARA RESOLVER PROBLEMAS RECORRENTES EM PROGRAMAÇÃO

CARACTERÍSTICAS DE UM BOM PADRÃO:

- RESOLVE UM PROBLEMA
- É UM CONCEITO TESTADO
- A SOLUÇÃO NÃO É OBVIA
- DESCREVE UMA RELAÇÃO
- O PADRÃO TEM UMA COMPONENTE HUMANA SIGNIFICATIVA

TIPOS DE PADRÃO:

- ARQUITETURAIS - EXPRESSAM UMA ESTRUTURA ORGANIZACIONAL DO SISTEMA

- DESIGN - ESTABELECE UM ESQUEMA PARA OS SUBSISTEMAS OU COMPONENTES DE UM SISTEMA (OU RELAÇÕES)

- IDIOMÁTICOS - DESCREVEM COMO IMPLEMENTAR ASPETOS PARTICULARES DOS COMPONENTES (OU RELAÇÕES ENTRE ELES) UTILIZANDO FUNÇÕES DA DADA LINGUA DE PROGRAMAÇÃO

GANG OF FOUR (GOF) PATTERNS

TEMPLATE DE PADRÕES :

- NOME
- INTENÇÃO (O QUE FAZ E VANTAGENS)
- MOTIVAÇÃO (EXEMPLO)
- ESTRUTURA (TEMPLATE DE UM DIAGRAMA DE CLASSES)
- APLICABILIDADE (QUANDO USAR)
- CONSEQUÊNCIAS (VANTAGENS E DESVANTAGENS)
- IMPLEMENTAÇÃO

3 GRUPOS DE PADRÕES:

- CREATIONAL - PROCESSO DE CRIAÇÃO DE UM OBJETO
- STRUCTURAL - COMPOSIÇÃO DE CLASSES OU OBJETOS
- BEHAVIOURAL - A FORMA COMO CLASSES E OBJETOS INTERAGEM E DISTRIBUEM RESPONSABILIDADES

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

CHAPT 4

DESIGN PATTERNS

CREATIONAL

SOURCE: REFACTORING.GURU

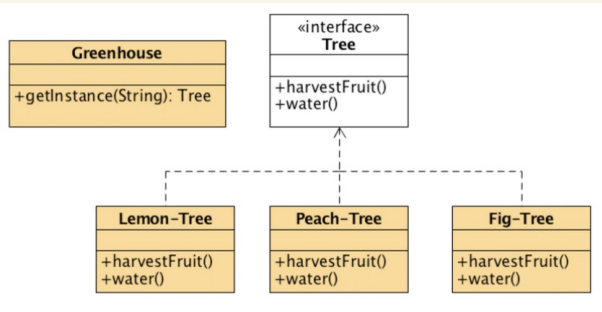
CLASS: FACTORY METHOD

INTENÇÃO: DEFINIR UMA INTERFACE PARA CRIAR UM OBJETO NUMA SUPERCLASSE, PERMITINDO ÀS SUBCLASSES ALTERAR O TIPO DE OBJETO A SER CRIADO, ATRAVÉS DE UM CONSTRUTOR VIRTUAL

PROBLEMA: EXPANSÃO DO CÓDIGO PARA OBJETOS DE TIPO DIFERENTE

SOLUÇÃO: SUBSTITUIR AS CHAMADAS AOS CONSTRUTORES DE CADA UM DOS OBJETOS A CRIAR, POR CHAMADAS AO MÉTODO DE FÁBRICA (ESTÁTICO), PASSANDO UM ARGUMENTO QUE DEFINA O TIPO DE OBJETO A CONSTRUIR, SENDO DEVOLVIDO UM OBJETO DESTE TIPO (PRODUTO)

EXEMPLO:

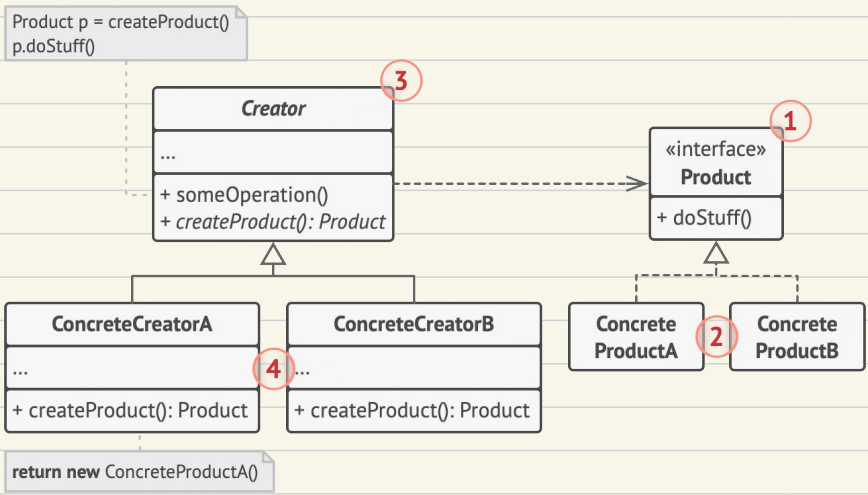


USE CASES:

- QUANDO NÃO SE SABE DE ANTEMÃO OS TIPOS E DEPENDÊNCIAS EXATAS DOS OBJETOS QUE IRÃO EXISTIR

- QUANDO SE PRETENDE ECONOMIZAR RECURSOS DO SISTEMA REUTILIZANDO OBJETOS EXISTENTES EM VEZ DE OS RECRIAR SEMPRE.

ESTRUTURA:



1 O PRODUTO DECLARA A INTERFACE, QUE É COMUM A TODOS OS OBJETOS QUE PODEM SER PRODUZIDOS PELO CREATOR E SUAS SUBCLASSES.

2 PRODUTOS CONCRETOS SÃO IMPLEMENTAÇÕES DIFERENTES DA INTERFACE DO PRODUTO.

3 A CLASSE CREATOR DECLARA O MÉTODO FÁBRICA QUE RETORNA NOVOS OBJETOS PRODUTO. É IMPORTANTE QUE O TIPO DE RETORNO DESSE MÉTODO CORRESPONDA À INTERFACE DO PRODUTO.
PODE SE DECLARAR O MÉTODO FÁBRICA COMO ABSTRATO PARA FORÇAR TODAS AS SUBCLASSES A IMPLEMENTAR AS SUAS VERSÕES DO MÉTODO. COMO ALTERNATIVA, O MÉTODO FÁBRICA BASE PODE RETORNAR ALGUM TIPO DE PRODUTO PADRÃO.

4 CRIADORES CONCRETOS SOBREPÕE-SE AO MÉTODO FÁBRICA BASE PARA RETORNAR UM TIPO DIFERENTE DE PRODUTO.

OBJECT: ABSTRACT FACTORY

INTENÇÃO: CRIAR UMA INTERFACE PARA CRIAR FAMÍLIAS DE OBJETOS RELACIONADOS SEM ESPECIFICAR A SUA CLASSE, ATRAVÉS DE UMA HIERARQUIA QUE ABRANGE VÁRIAS PLATAFORMAS E A CONSTRUÇÃO DE VÁRIOS PRODUTOS, EVITANDO A UTILIZAÇÃO DO OPERADOR NEW

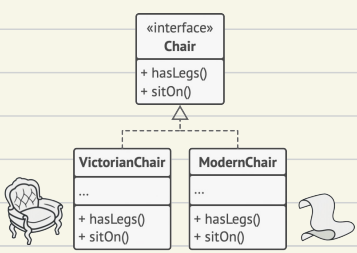
SOLUÇÃO:

1. CRIAR EXPLICITAMENTE INTERFACES PARA CADA PRODUTO DISTINTO DA FAMÍLIA DE PRODUTOS (EX. CADEIRA, SOFÁ, MESA). TODAS AS VARIANTES DOS PRODUTOS SEGUEM ESTAS INTERFACES

2. DECLARAR A ABSTRACT FACTORY - UMA INTERFACE COM UMA LISTA DE MÉTODOS DE CRIAÇÃO PARA TODOS OS PRODUTOS QUE FAZEM PARTE DA FAMÍLIA DE PRODUTOS (EX. CRIAR_CADEIRA, CRIAR_SOFÁ, CRIAR_MESA).

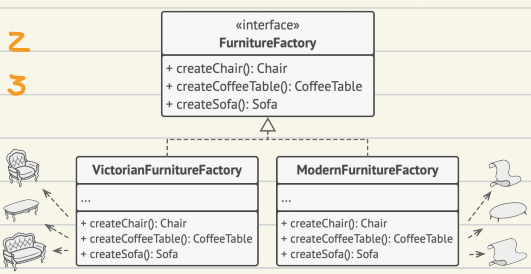
3. CRIAR UMA FACTORY PARA CADA VARIANTE DA FAMILIA DE PRODUTOS, BASEADA NA INTERFACE

1

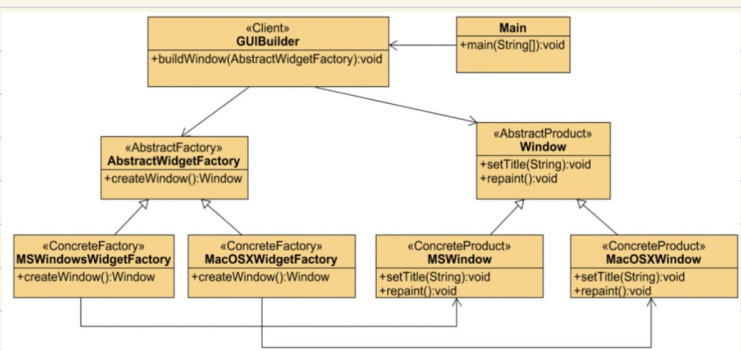


2

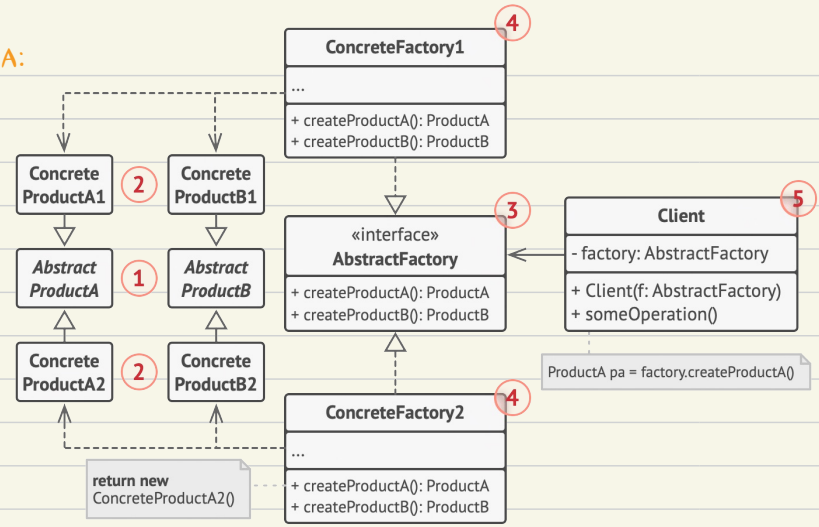
3



EXEMPLO:



ESTRUTURA:



1 PRODUTOS ABSTRATOS DECLARAM INTERFACES PARA UM CONJUNTO DE PRODUTOS DISTINTOS MAS RELACIONADOS QUE FAZEM PARTE DE UMA FAMÍLIA DE PRODUTOS.

2 PRODUTOS CONCRETOS SÃO VÁRIAS IMPLEMENTAÇÕES DE PRODUTOS ABSTRATOS, AGRUPADOS POR VARIANTES. CADA PRODUTO ABSTRATO (CADEIRA/SOFÁ) DEVE SER IMPLEMENTADO EM TODAS AS VARIANTES DADAS (VICTORIAN/MODERN).

3 A INTERFACE FÁBRICA ABSTRATA DECLARA UM CONJUNTO DE MÉTODOS PARA CRIAÇÃO DE CADA UM DOS PRODUTOS ABSTRATOS.

4 FÁBRICAS CONCRETAS IMPLEMENTAM MÉTODOS DA FABRICA ABSTRATA. CADA FÁBRICA CONCRETA CORRESPONDE A UMA VARIANTE ESPECÍFICA DE PRODUTOS E CRIA APENAS AQUELAS VARIANTES DE PRODUTO.

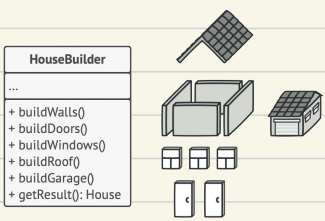
5 EMBORA FÁBRICAS CONCRETAS INSTANCIAM PRODUTOS CONCRETOS, OS SEUS MÉTODOS DE CRIAÇÃO DEVEM RETORNAR PRODUTOS ABSTRATOS CORRESPONDENTES. ASSIM O CLIENTE QUE USA UMA FÁBRICA NÃO FICA LIGADO À VARIANTE ESPECÍFICA DO PRODUTO QUE FOI SELECIONADO. O CLIENTE PODE TRABALHAR COM QUALQUER VARIANTE DE PRODUTO/FÁBRICA CONCRETO, DESDE QUE A LIGAÇÃO FEITA COM OS OBJETOS SEJA VIA INTERFACES ABSTRATAS.

OBJECT: BUILDER

INTENÇÃO: SEPARAR A CONSTRUÇÃO DE UM OBJETO COMPLEXO DA SUA REPRESENTAÇÃO, PODENDO ASSIM CRIAR DIFERENTES REPRESENTAÇÕES

SOLUÇÃO: ESTENDER A BASE DO OBJETO E CRIAR UM CONJUNTO DE SUBCLASSES PARA COBRIR TODAS AS COMBINAÇÕES DE PARÂMETROS

- EXTRAIR O CÓDIGO DE CONSTRUÇÃO DO OBJETO PARA FORA DA CLASSE E TRANSFORMAR EM OBJETOS SEPARADOS (BUILDERS)

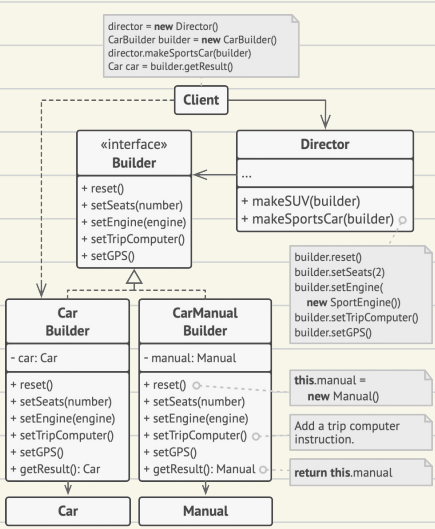


PARA CRIAR UM OBJETO BASTA EXECUTAR CERTAS FUNÇÕES NUM OBJETO BUILDER.

PODE-SE EXTRAIR UMA SÉRIE DE CHAMADAS DO BUILDER PARA CONSTRUIR UM PRODUTO NUMA CLASSE SEPARADA CHAMADA DIRECTOR. ESTA DEFINE A ORDEM NA QUAL EXECUTAR AS ETAPAS DE CONSTRUÇÃO, ENQUANTO QUE O BUILDER EXECUTA A IMPLEMENTAÇÃO DESSAS ETAPAS.

O DIRECTOR ESCONDE COMPLETAMENTE OS DETALHES DA CONSTRUÇÃO DO PRODUTO DO CÓDIGO CLIENTE.

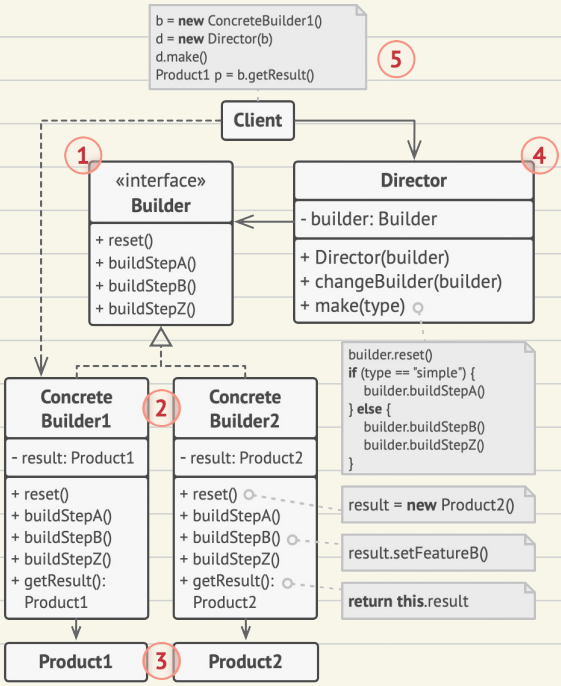
EXEMPLO:



ESTRUTURA:

1. A INTERFACE BUILDER DECLARA ETAPAS DE CONSTRUÇÃO DO PRODUTO QUE SÃO COMUNS A TODOS OS TIPOS DE BUILDERS.

2. BUILDERS CONCRETOS FAZEM DIFERENTES IMPLEMENTAÇÕES DAS ETAPAS DE CONSTRUÇÃO. PODEM PRODUZIR PRODUTOS QUE NÃO SEGUEM A INTERFACE COMUM.



3 PRODUTOS SÃO OS OBJETOS RESULTANTES. PRODUTOS CONSTRUÍDOS POR DIFERENTES BUILDERS NÃO PRECISAM DE PERTENCER A MESMA INTERFACE OU HIERARQUIA DE CLASSE.

4 A CLASSE DIRETOR DEFINE A ORDEM NA QUAL AS ETAPAS DE CONSTRUÇÃO SÃO CHAMADAS, PODENDO CRIAR E REUTILIZAR CONFIGURAÇÕES ESPECÍFICAS DE PRODUTOS.

5 O CLIENTE DEVE ASSOCIAR UM DOS OBJETOS BUILDERS COM O DIRETOR. NORMALMENTE FEITO APENAS UMA VEZ, ATRAVÉS DE PARÂMETROS DO CONSTRUTOR DO DIRETOR. O DIRETOR ENTÃO USA AQUELE OBJETO BUILDER PARA TODAS AS FUTURAS CONSTRUÇÕES. CONTUDO, HÁ UMA ABORDAGEM ALTERNATIVA PARA QUANDO O CLIENTE PASSA O OBJETO BUILDER AO MÉTODO DE PRODUÇÃO DO DIRETOR. NESSE CASO, PODE-SE USAR UM BUILDER DIFERENTE CADA VEZ QUE PRODUZIR ALGO COM O DIRETOR.

OBJECT: SINGLETON

INTENÇÃO: GARANTIR QUE UMA CLASSE TEM UMA ÚNICA INSTÂNCIA E FORNECER UM PONTO DE ACESSO GLOBAL

SOLUÇÃO:

- DEFINIR O CONSTRUTOR COMO PRIVADO (OU PROTECTED)
  PRIVATE SINGLETON(STRING NAME)

  PREVINE QUE OUTROS OBJETOS USEM 'NEW' COM A CLASSE SINGLETON

- DEFINIR UM ATRIBUTO ESTÁTICO DO SEU TIPO NA CLASSE
  STATIC PRIVATE SINGLETON INSTANCE

- DEFINIR UM MÉTODO PÚBLICO PARA LHE ACEDER
  STATIC PUBLIC SINGLETON GET_INSTANCE()

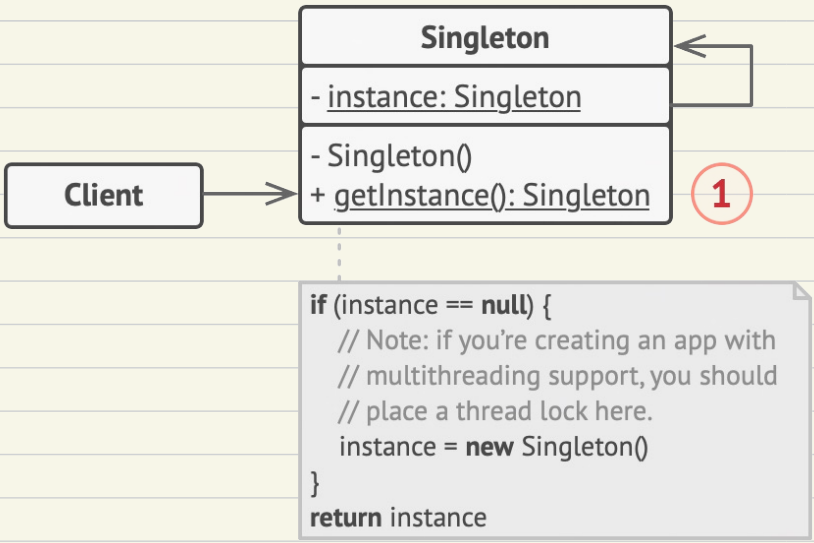
  ÚNICO MÉTODO UTILIZADO PELOS CLIENTES PARA LHE ACEDER

EXEMPLO:

```
class Singleton {

    private String name;

    static private Singleton instance = new Singleton("Ermita");

    private Singleton(String name) {
        this.name = name;
    }

    static public Singleton getInstance() {
        return instance;
    }
    @Override
    public String toString() {
        return name;
    }
}
```

```
class LazySingleton {

    private String name;

    static private LazySingleton instance=null;

    private LazySingleton(String name) {
        this.name = name;
    }
    static public synchronized LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton("Ermita");
        }
        return instance;
    }
    @Override
    public String toString() {
        return name;
    }
```

ESTRUTURA



1. A CLASSE SINGLETON DECLARA O MÉTODO ESTÁTICO GETINSTANCE QUE RETORNA A MESMA INSTÂNCIA DE SUA PRÓPRIA CLASSE.

O CONSTRUTOR DO SINGLETON DEVE SER ESCONDIDO DO CÓDIGO CLIENTE. O MÉTODO GETINSTANCE DEVE SER O ÚNICO PONTO DE ACESSO DO SINGLETON

OBJECT: OBJECT POOL

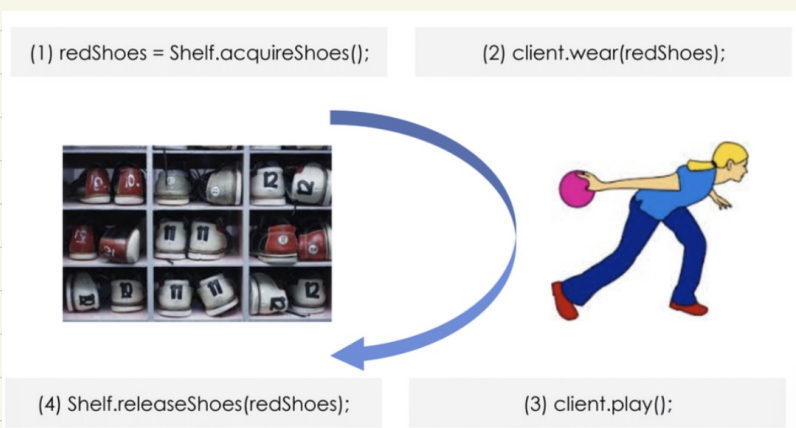
INTENÇÃO: AUMENTAR A EFICIÊNCIA NA UTILIZAÇÃO DE OBJETOS COM CUSTOS ELEVADOS DE INICIALIZAÇÃO QUE SÃO UTILIZADOS FREQUENTEMENTE MAS CUJA INSTANCIAÇÃO SIMULTÂNEA É REDUZIDA.

(DB / NETWORK CONNECTIONS, THREADS)

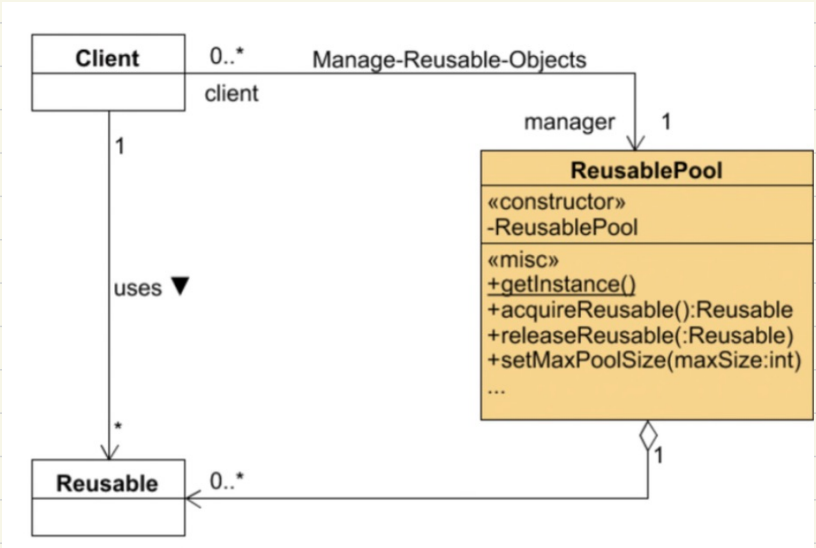
SOLUÇÃO: CRIAR UMA CLASSE RESPONSÁVEL POR GERIR OS OBJETOS REUTILIZÁVEIS.

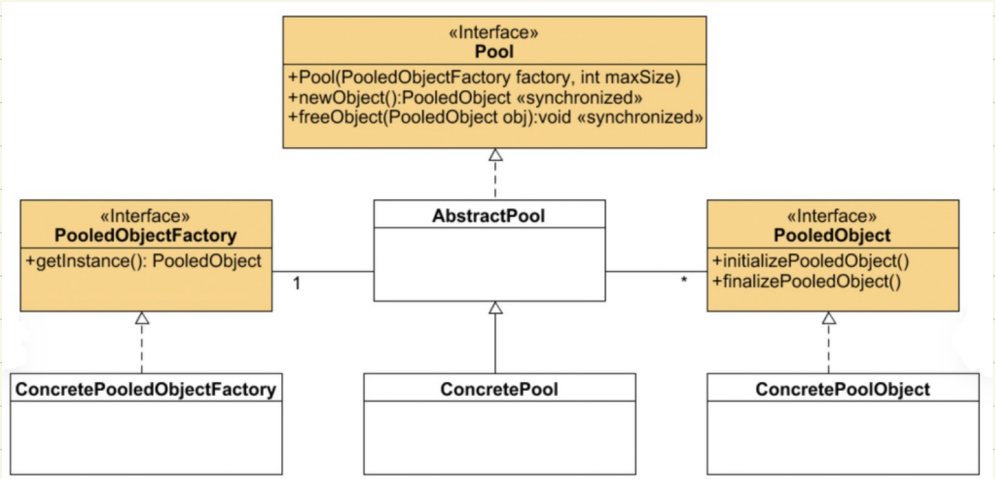
1. CRIAR UMA CLASSE DO TIPO POOL COM UMA COLEÇÃO DE POOL OBJECTS
2. CRIAR MÉTODOS ACQUIRE() E RELEASE() NESTA CLASSE.

EXEMPLO:



ESTRUTURA


Client
0..*
Manage-Reusable-Objects
client
manager
1
1
uses
ReusablePool
«constructor»
-ReusablePool
«misc»
+getInstance()
+acquireReusable():Reusable
+releaseReusable(:Reusable)
+setMaxPoolSize(maxSize:int)
...
*
1
Reusable
0..*


«Interface»
Pool
+Pool(PooledObjectFactory factory, int maxSize)
+newObject():PooledObject «synchronized»
+freeObject(PooledObject obj):void «synchronized»
«Interface»
PooledObjectFactory
+getInstance(): PooledObject
1
AbstractPool
«Interface»
PooledObject
+initializePooledObject()
+finalizePooledObject()
*
ConcretePooledObjectFactory
ConcretePool
ConcretePoolObject

OBJECT: PROTOTYPE

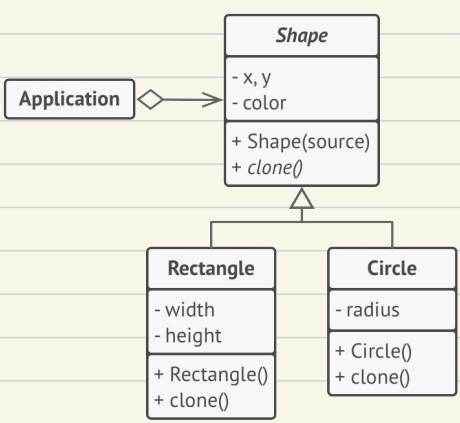
INTENÇÃO: CRIAR OBJETOS NOVOS A PARTIR DE EXISTENTES, COMO PROTÓTIPOS, SEM ESTAR DEPENDENTE DAS SUAS CLASSES.

SOLUÇÃO: DELEGAR O PROCESSO DE CLONAGEM NO OBJETO A SER COPIADO ATRAVÉS DE UM MÉTODO DECLARADO NUMA INTERFACE COMUM A TODOS OS OBJETOS PASSÍVEIS DE CLONAGEM

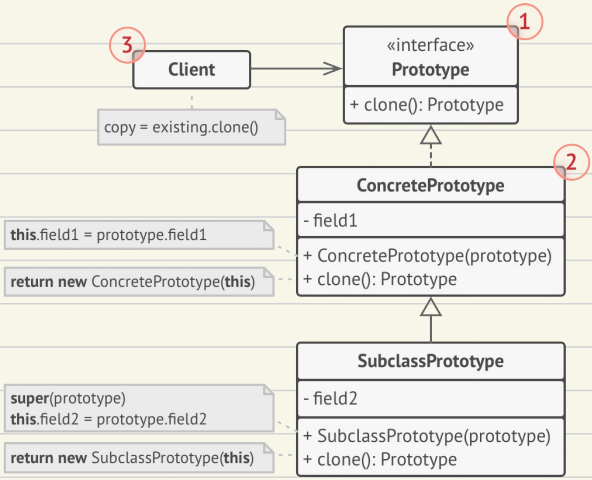
1. ADICIONAR UM MÉTODO CLONE() AO PRODUTO;
2. DESENHAR UM REGISTO QUE MANTÉM A CACHE DOS OBJETOS PROTOTIPÁVEIS;

- PODE ESTAR ENCAPSULADO NUMA CLASSE DO TIPO FACTORY, OU NA CLASSE BASE DA HIERARQUIA DO PRODUTO;
- PODE OU NÃO ACEITAR ARGUMENTOS, DESCOBRE O PROTÓTIPO CORRETO A CLONAR E INVOCA O MÉTODO CLONE(), RETORNANDO O SEU RESULTADO;
- O CLIENTE SUBSTITUI TODAS AS REFERÊNCIAS AO OPERADOR NEW E SUBSTITUI-LAS POR INVOCAÇÕES AO MÉTODO DE FÁBRICA.

EXEMPLO:



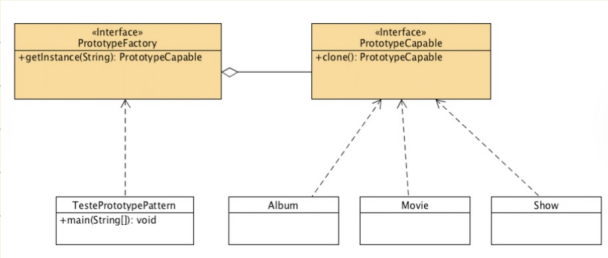
ESTRUTURA:



1 A INTERFACE PROTÓTIPO DECLARA OS MÉTODOS DE CLONAGEM. NA MAIORIA DOS CASOS É APENAS UM MÉTODO CLONAR.

2 A CLASSE PROTÓTIPO CONCRETA IMPLEMENTA O MÉTODO DE CLONAGEM. ALÉM DE COPIAR OS DADOS DO OBJETO ORIGINAL PARA O CLONE, O MÉTODO TAMBÉM PODE LIDAR COM ALGUNS CASOS ESPECÍFICOS DO PROCESSO DE CLONAGEM RELACIONADOS A CLONAR OBJETOS LIGADOS, DESFAZENDO DEPENDÊNCIAS RECURSIVAS, ETC.

3 O CLIENTE PODE PRODUZIR UMA CÓPIA DE QUALQUER OBJETO QUE SEGUE A INTERFACE DO PROTÓTIPO.



RESUMO:

ABSTRACT FACTORY

CRIAR UMA INSTÂNCIA DE VÁRIAS FAMÍLIAS DE CLASSES.

BUILDER

SEPARAR A CONSTRUÇÃO DO OBJETO DE SUA REPRESENTAÇÃO.

FACTORY METHOD

CRIAR UMA INSTÂNCIA DE VÁRIAS CLASSES DERIVADAS.

SINGLETON

UMA CLASSE NA QUAL APENAS UMA INSTÂNCIA PODE EXISTIR.

OBJECT POOL

EVITAR AQUISIÇÕES E LIBERTAÇÕES CARAS DE RECURSOS AO RECICLAR OBJETOS QUE JÁ NÃO ESTÃO EM USO.

PROTOTYPE

UMA INSTÂNCIA INICIALIZADA PARA SER COPIADA OU CLONADA.

CHAPT 5

DESIGN PATTERNS

STRUCTURAL

SOURCE: REFACTORING.GURU

CLASS: ADAPTER

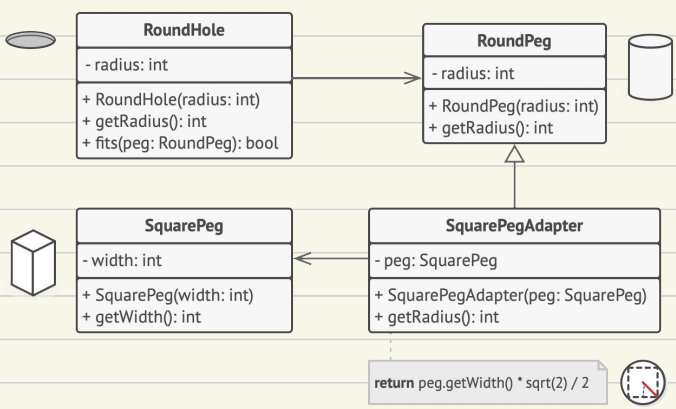
INTENÇÃO: PERMITIR OBJETOS COM INTERFACES INCOMPATÍVEIS COLABORAREM ENTRE SI.

SOLUÇÃO:
A CLASS ADAPTER É RESPONSÁVEL POR:

- FORNECER UMA INTERFACE, COMPATÍVEL COM UM DOS OBJETOS;
- ESTE OBJETO ACEDE AOS MÉTODOS DESTA INTERFACE;
- A CADA CHAMADA, O ADAPTER PASSA O PEDIDO AO SEGUNDO OBJETO, MAS COM A FORMATAÇÃO ADEQUADA (PROCESSO DE ADAPTAÇÃO).

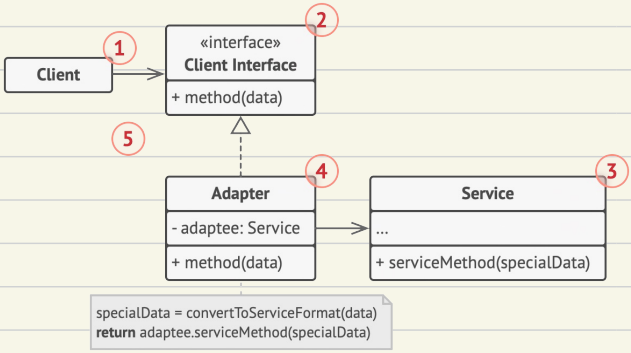
O OBJETO "ADAPTADO" NÃO SE APERCEBE DA EXISTÊNCIA DO ADAPTER.

EXEMPLO:



JAVA EXAMPLE: HTTPS://REFACTORING.GURU/DESIGN-PATTERNS/ADAPTER/JAVA/EXAMPLE

ESTRUTURA:



1. CLIENTE É A CLASSE QUE CONTÉM A LÓGICA DO PROGRAMA.

2. A INTERFACE DO CLIENTE DESCREVE UM PROTOCOLO QUE AS CLASSES DEVEM SEGUIR PARA PODEREM COLABORAR COM O CÓDIGO CLIENTE.

3. O SERVIÇO É UMA CLASSE ÚTIL (3RD PARTY OU LEGACY). O CLIENTE NÃO A PODE USAR DIRETAMENTE POIS TÊM INTERFACES INCOMPATÍVEIS.

4. O ADAPTADOR É A CLASSE QUE É CAPAZ DE TRABALHAR COM O CLIENTE E O SERVIÇO: ELA IMPLEMENTA A INTERFACE DO CLIENTE ENQUANTO ENCOBRE O OBJETO DO SERVIÇO. O ADAPTADOR RECEBE CHAMADAS DO CLIENTE ATRAVÉS DA INTERFACE DO ADAPTADOR E TRADUZ EM CHAMADAS PARA O OBJETO ENCOBRIDO DO SERVIÇO EM UM FORMATO QUE ELE POSSA ENTENDER.

5. O CÓDIGO CLIENTE NÃO É ACOPLADO À CLASSE CONCRETA DO ADAPTADOR DESDE QUE ELE TRABALHE COM O ADAPTADOR ATRAVÉS DA INTERFACE DO CLIENTE.

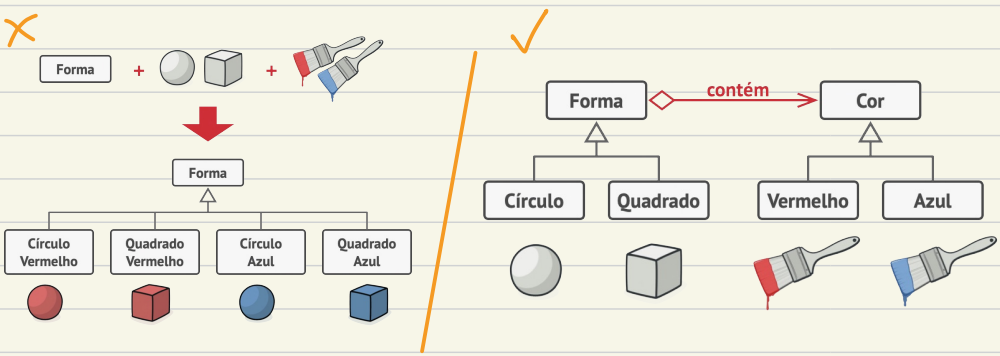
```
public class SquarePegAdapter extends RoundPeg {
    private SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        double result;
        // Calculate a minimum circle radius, which can fit this peg.
        result = (Math.sqrt(Math.pow((peg.getWidth() / 2), 2) * 2));
        return result;
```

OBJECT: BRIDGE

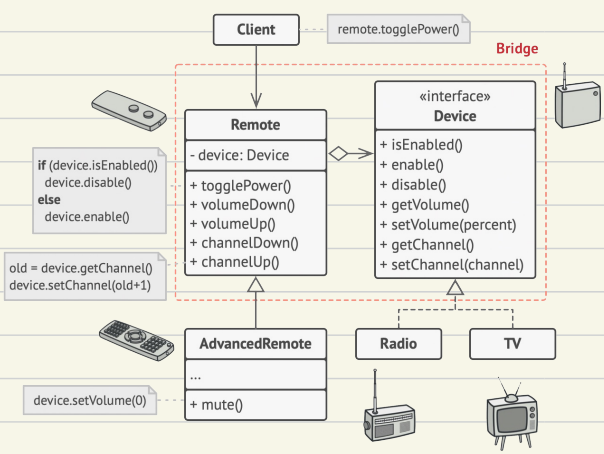
INTENÇÃO: PERMITIR A DIVISÃO DE UMA CLASSE OU UM CONJUNTO DE CLASSES RELACIONADAS EM DUAS HIERARQUIAS: ABSTRAÇÃO E IMPLEMENTAÇÃO, PARA QUE AMBAS POSSAM SER DESENVOLVIDAS DE FORMA INDEPENDENTE.

SOLUÇÃO: TROCAR DE HERANÇA PARA COMPOSIÇÃO DO OBJETO

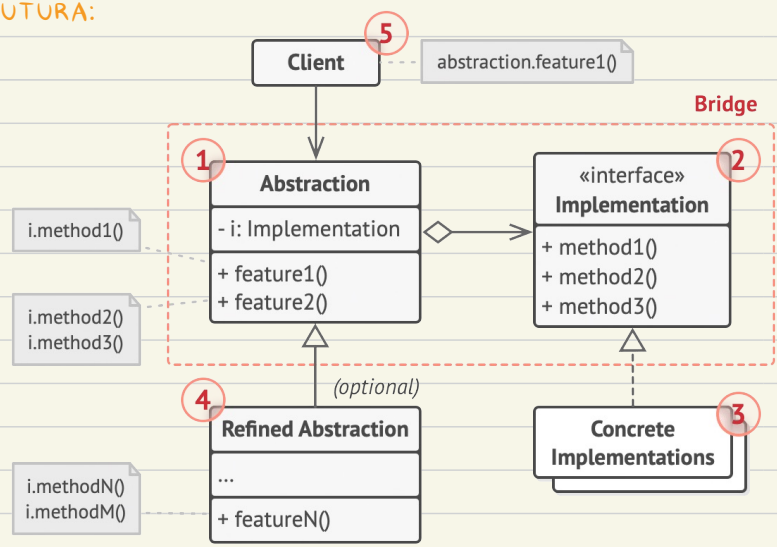
- EXTRAIR UMA DAS DIMENSÕES EM UMA HIERARQUIA DE CLASSE SEPARADA, PARA QUE AS CLASSES ORIGINAIS REFERENCIEM UM OBJETO DA NOVA HIERARQUIA, AO INVÉS DE TER TODOS OS SEUS ESTADOS E COMPORTAMENTOS DENTRO DE UMA CLASSE.



EXEMPLO:



ESTRUTURA:



1 A ABSTRACTION FORNECE A LÓGICA DE HIGH-LEVEL CONTROL. ELA DEPENDE DA IMPLEMENTAÇÃO PARA FAZER O LOW-LEVEL WORK.

2 A IMPLEMENTATION DECLARA A INTERFACE COMUM PARA TODAS AS IMPLEMENTAÇÕES CONCRETAS. UMA ABSTRAÇÃO SÓ PODE COMUNICAR COM UM OBJETO DE IMPLEMENTAÇÃO ATRAVÉS DE MÉTODOS QUE SÃO DECLARADOS AQUI.
ABSTRAÇÃO PODE TER OS MESMOS MÉTODOS QUE A IMPLEMENTAÇÃO, MAS GERALMENTE DECLARA COMPORTAMENTOS COMPLEXOS QUE DEPENDEM DE UMA VARIEDADE DE OPERAÇÕES DECLARADAS PELA IMPLEMENTAÇÃO.

3 CONCRETE IMPLEMENTATIONS CONTÉM CÓDIGO ESPECÍFICO DA PLATAFORMA .

4 REFINED ABSTRACTIONS FORNECEM VARIANTES DE LOGIC CONTROL. COMO A CLASSE DE ABSTRAÇÃO, TRABALHAM COM DIFERENTES IMPLEMENTAÇÕES ATRAVÉS DA INTERFACE GERAL DE IMPLEMENTAÇÃO.

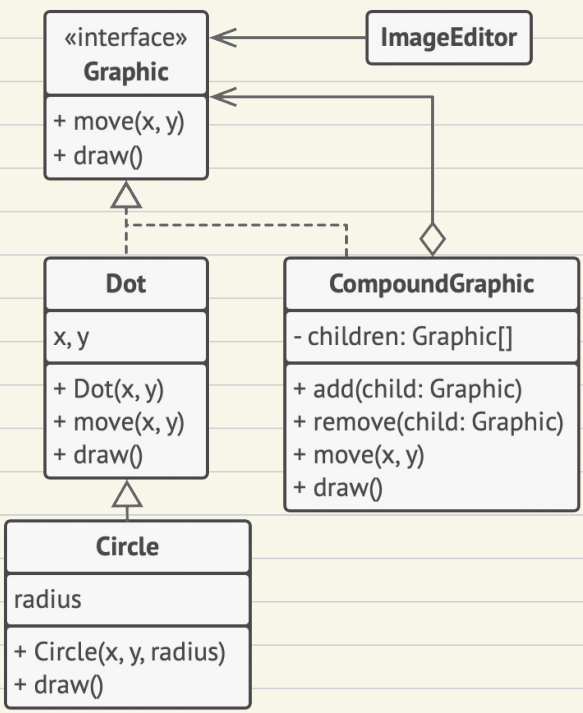
5 GERALMENTE O CLIENTE TRABALHA SÓ COM A ABSTRAÇÃO - TEM COMO TRABALHO LIGAR O OBJETO DE ABSTRAÇÃO COM UM DOS OBJETOS DE IMPLEMENTAÇÃO.

OBJECT: COMPOSITE

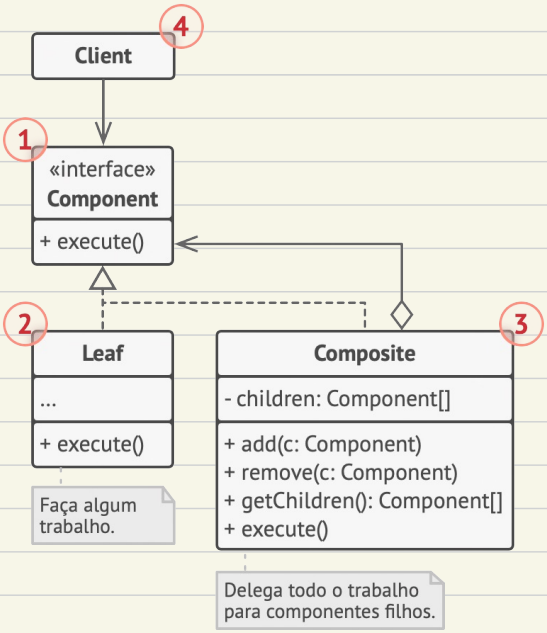
INTENÇÃO: TER COMPONENTES DE PRODUTOS MAS DE FORMA A QUE UM COMPONENTE EM SI TAMBÉM SEJA UM PRODUTO (COMPOSIÇÃO RECURSIVA)

SOLUÇÃO: CRIAR UMA INTERFACE COMUM AO PRODUTO E À COMPOSIÇÃO DE PRODUTOS, SENDO QUE ESTA ÚLTIMA CONTÉM UM ATRIBUTO QUE É UMA COLEÇÃO DE ELEMENTOS DESTA INTERFACE.

EXEMPLO:



ESTRUTURA:



1 A INTERFACE COMPONENT DESCREVE OPERAÇÕES QUE SÃO COMUNS PARA OS ELEMENTOS SIMPLES E PARA OS ELEMENTOS COMPLEXOS DA ÁRVORE.

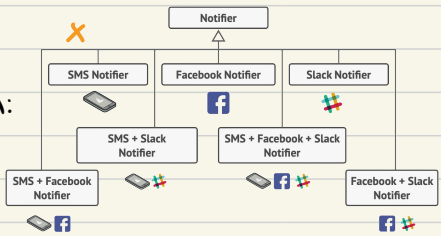
2 A FOLHA É UM ELEMENTO BÁSICO DE UMA ÁRVORE QUE NÃO TEM SUB-ELEMENTOS. ACABAM POR FAZER MAIOR PARTE DO TRABALHO POIS NÃO TÊM NINGUÉM PARA O DELEGAR.

3 O CONTAINER (OU COMPOSITE) É O ELEMENTO QUE TEM SUB-ELEMENTOS: FOLHAS OU OUTROS CONTAINERS. TRABALHA COM TODOS OS SUB-ELEMENTOS APENAS ATRAVÉS DA INTERFACE COMPONENTE.
AO RECEBER UM REQUEST, UM CONTAINER DELEGA O TRABALHO PARA OS SEUS SUB-ELEMENTOS, PROCESSA OS RESULTADOS INTERMEDIÁRIOS, RETORNA O RESULTADO FINAL PARA O CLIENTE.

4 O CLIENTE TRABALHA COM TODOS OS ELEMENTOS ATRAVÉS DA INTERFACE COMPONENTE. COMO RESULTADO, O CLIENTE PODE TRABALHAR DA MESMA FORMA TANTO COM ELEMENTOS SIMPLES COMO ELEMENTOS COMPLEXOS DA ÁRVORE.

OBJECT: DECORATOR

INTENÇÃO: ADICIONAR NOVAS RESPONSABILIDADES A UM OBJETO DE FORMA DINÂMICA, COLOCANDO-O DENTRO DE UM WRAPPER QUE CONTÉM ESSE COMPORTAMENTO



APESAR QUE A HERANÇA É UMA OPÇÃO, ELA:

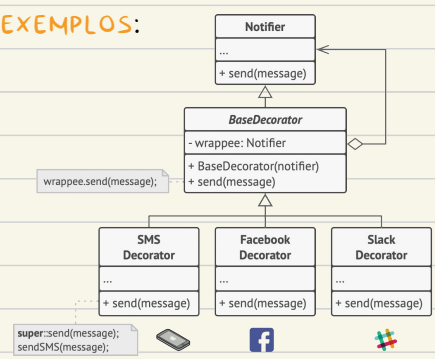
- É ESTÁTICA
- NÃO TEM HERANÇA MÚLTIPLA
- CRIARIA DEMASIADAS SUBCLASSES

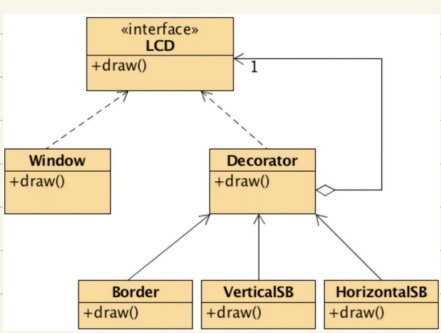
SOLUÇÃO: AGREGAÇÃO/COMPOSIÇÃO
CRIAR UM OBJETO (WRAPPER) QUE REFERÊNCIA OUTRO E ATRIBUIR ALGUM DO SEU TRABALHO À CLASSE

O WRAPPER IMPLEMENTA OS MESMOS MÉTODOS QUE O ALVO, NO QUAL DELEGA OS PEDIDOS QUE RECEBE, PODENDO, NO ENTANTO, ALTERAR OS DADOS ANTES OU DEPOIS DE OS PASSAR.

- WRAPPER E WRAPPED OBJECT IMPLEMENTAM A MESMA INTERFACE

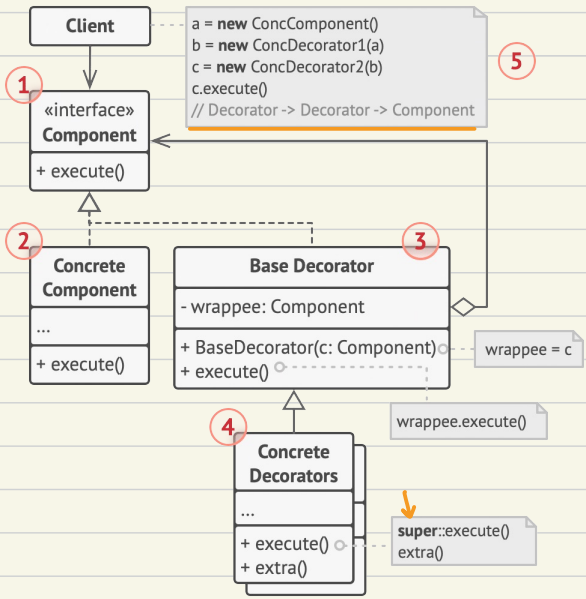
EXEMPLOS:





! JAVA CODE: HTTPS://REFACTORING.GURU/DESIGN-PATTERNS/DECORATOR/JAVA/EXAMPLE

ESTRUTURA:



1 O COMPONENTE DECLARA A INTERFACE COMUM TANTO PARA OS WRAPPER COMO PARA OS WRAPPED OBJETOS .

2 O COMPONENTE CONCRETO É UMA CLASSE DE OBJETOS QUE VAI SER WRAPPED . ELA DEFINE O COMPORTAMENTO BÁSICO, QUE PODE SER ALTERADO POR DECORADORES.

3 A CLASSE DECORADOR BASE TEM UM CAMPO PARA REFERENCIAR UM WRAPPED OBJECT. O TIPO DO OBJETO DEVE SER DECLARADO IGUAL À INTERFACE DO COMPONENTE PARA QUE AMBOS POSSAM CONTER OS COMPONENTES CONCRETOS E OS DECORADORES. O DECORADOR BASE DELEGA TODAS AS OPERAÇÕES PARA O OBJETO ENVOLVIDO.

4 OS DECORADORES CONCRETOS DEFINEM OS COMPORTAMENTOS ADICIONAIS QUE PODEM SER ADICIONADOS AOS COMPONENTES DINAMICAMENTE. OS MÉTODOS DOS DECORADORES CONCRETOS DÃO OVERRIDE AOS DO DECORADOR BASE E EXECUTAM O CÓDIGO TANTO ANTES COMO DEPOIS DE CHAMAREM O MÉTODO BASE.

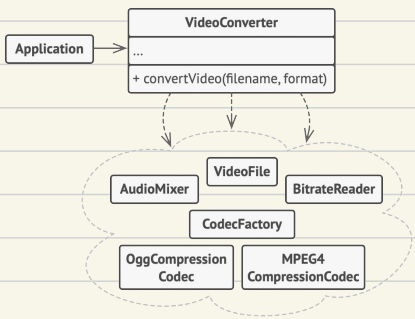
5 O CLIENTE PODE ENVOLVER COMPONENTES EM MÚLTIPLAS CAMADAS DE DECORADORS, DESDE QUE TRABALHE COM TODOS OS OBJETOS ATRAVÉS DA INTERFACE DO COMPONENTE.

OBJECT: FACADE

INTENÇÃO: CRIAR UMA INTERFACE COMUM (E MAIS SIMPLES) A UM CONJUNTO (COMPLEXO) DE INTERFACES NUM SUBSISTEMA

SOLUÇÃO: IMPLEMENTAR A INTERFACE QUE UTILIZA APENAS OS MÉTODOS QUE O CLIENTE NECESSITA

EXEMPLO:




```java
public class VideoFile {
    private String name;
    private String codecType;

    public VideoFile(String name) {
        this.name = name;
        this.codecType = name.substring(name.indexOf(".") + 1);
    }

    public String getCodecType() {
        return codecType;
    }

    public String getName() {
        return name;
    }
}
```

```java
public interface Codec {
}
```

```java
public class MPEG4CompressionCodec implements Codec {
    public String type = "mp4";

}
```

```java
public class OggCompressionCodec implements Codec {
    public String type = "ogg";
}
```

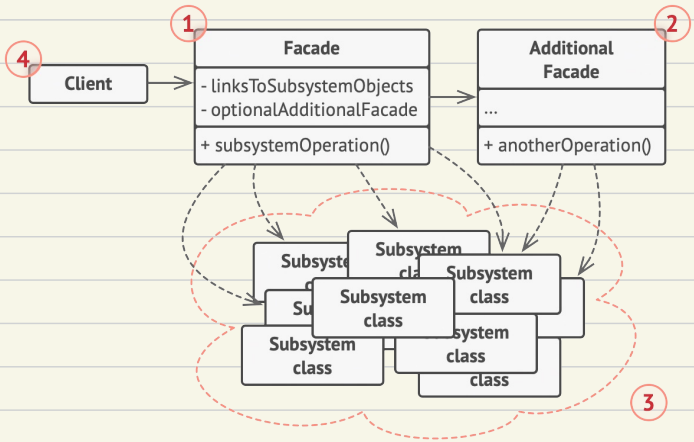
(. . .)

CLIENT

```java
public class Demo {
    public static void main(String[] args) {
        VideoConversionFacade converter = new VideoConversionFacade();
        File mp4Video = converter.convertVideo("youtubevideo.ogg", "mp4");
        // ...
    }
}
```

```java
public class VideoConversionFacade {
    public File convertVideo(String fileName, String format) {
        System.out.println("VideoConversionFacade: conversion started.");
        VideoFile file = new VideoFile(fileName);
        Codec sourceCodec = CodecFactory.extract(file);
        Codec destinationCodec;
        if (format.equals("mp4")) {
            destinationCodec = new MPEG4CompressionCodec();
        } else {
            destinationCodec = new OggCompressionCodec();
        }
        VideoFile buffer = BitrateReader.read(file, sourceCodec);
        VideoFile intermediateResult = BitrateReader.convert(buffer, destinationCodec);
        File result = (new AudioMixer()).fix(intermediateResult);
        System.out.println("VideoConversionFacade: conversion completed.");
        return result;
    }
}
```

ESTRUTURA:



1. A FACHADA FORNECE UM ACESSO CONVENIENTE PARA UMA PARTE PARTICULAR DA FUNCIONALIDADE DO SUBSISTEMA. ELA SABE ONDE DIRECIONAR O PEDIDO DO CLIENTE E COMO OPERAR TODAS AS PARTES MÓVEIS.

2. UMA CLASSE FACHADA ADICIONAL PODE SER CRIADA PARA PREVENIR A POLUIÇÃO DE UMA ÚNICA FACHADA COM FUNCIONALIDADES NÃO RELEVANTES QUE PODEM TORNÁ-LA MAIS COMPLEXA. FACHADAS ADICIONAIS PODEM SER USADAS TANTO POR CLIENTES COMO POR OUTRAS FACHADAS.

3. O SUBSISTEMA COMPLEXO CONSISTE EM VÁRIOS OBJETOS VARIADOS. DE MODO A FUNCIONAR, É PRECISO ENTRAR NOS DETALHES DA IMPLEMENTAÇÃO DO SUBSISTEMA, TAIS COMO A INICIALIZAÇÃO DOS OBJETOS POR ORDEM CORRETA E INSTÂNCIA-LOS COM DADOS NO FORMATO CORRETO.
AS CLASSES DO SUBSISTEMA NÃO SABEM DA EXISTÊNCIA DA FACHADA. ELAS OPERAM DENTRO DO SISTEMA E TRABALHAM ENTRE SI DIRETAMENTE.

4. O CLIENTE USA A FACHADA EM VEZ DE CHAMAR OS OBJETOS DO SUBSISTEMA DIRETAMENTE.

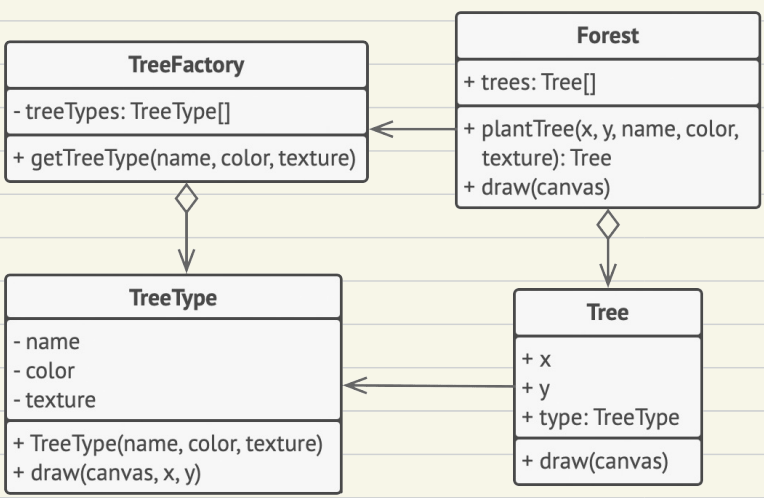
OBJETO: FLYWEIGHT

INTENÇÃO: UTILIZAR A PARTILHA DE FORMA A SUPORTAR MUITOS OBJETOS COMPACTOS DE FORMA EFICIENTE.

SOLUÇÃO: SEPARAR OS ATRIBUTOS COMUNS A TODOS OS OBJETOS DESSA CLASSE (ESTADO INTRÍNSECO) DOS ESPECÍFICOS IMUTÁVEIS EM CADA UM (ESTADO EXTRÍNSECO) EM DUAS CLASSES SENDO QUE A QUE TEM O ESTADO EXTRÍNSECO DERIVA DA QUE TEM O INTRÍNSECO

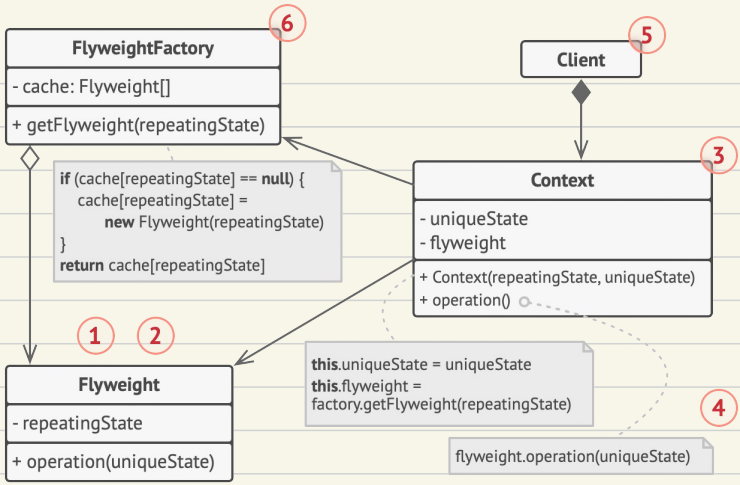
O CLIENTE DEVE UTILIZAR UMA FÁBRICA SEGUNDO O PADRÃO OBJECT POOL EM VEZ DO OPERADOR NEW()

EXEMPLO:



NOTA: O PADRÃO FLYWEIGHT É APENAS UMA OPTIMIZAÇÃO. ANTES DE O APLICAR, VERIFIQUE SE O PROGRAMA TEM UM PROBLEMA DE CONSUMO DE RAM RELACIONADO A EXISTÊNCIA DE MÚLTIPLOS OBJETOS SEMELHANTES NA MEMÓRIA AO MESMO TEMPO

ESTRUTURA:



2. A CLASSE FLYWEIGHT CONTÉM A PORÇÃO DO ESTADO DO OBJETO ORIGINAL QUE PODE SER PARTILHADA PELOS OBJETOS. O ESTADO ARMAZENADO DENTRO DE UM FLYWEIGHT É CHAMADO "INTRÍNSECO". O ESTADO PASSADO PELOS MÉTODOS FLYWEIGHT É CHAMADO "EXTRÍNSECO".

3. A CLASSE CONTEXTO CONTÉM O ESTADO EXTRÍNSECO, ÚNICO PARA TODOS OS OBJETOS ORIGINAIS. QUANDO UM CONTEXTO É COMBINADO COM UM DOS OBJETOS FLYWEIGHT, FICA A REPRESENTAR O ESTADO COMPLETO DO OBJETO ORIGINAL.

4. GERALMENTE, O COMPORTAMENTO DO OBJETO ORIGINAL PERMANECE NA CLASSE FLYWEIGHT. NESSE CASO, QUEM CHAMAR O MÉTODO DO FLYWEIGHT DEVE TAMBÉM PASSAR OS DADOS APROPRIADOS DO ESTADO EXTRÍNSECO NOS PARÂMETROS DO MÉTODO. POR OUTRO LADO, O COMPORTAMENTO PODE SER MOVIDO PARA A CLASSE CONTEXTO, QUE USARIA O FLYWEIGHT MERAMENTE COMO UM OBJETO DE DADOS.

5. O CLIENTE CALCULA OU ARMAZENA O ESTADO EXTRÍNSECO DOS FLYWEIGHTS. DA PERSPECTIVA DO CLIENTE, UM FLYWEIGHT É UM OBJETO MODELO QUE PODE SER CONFIGURADO NO MOMENTO DA EXECUÇÃO AO PASSAR ALGUNS DADOS DE CONTEXTO NOS PARÂMETROS DE SEUS MÉTODOS.

6. A FÁBRICA FLYWEIGHT GERE UM CONJUNTO DE FLYWEIGHTS EXISTENTES. COM A FÁBRICA OS CLIENTES NÃO PRECISAM CRIAR FLYWEIGHTS DIRETAMENTE.

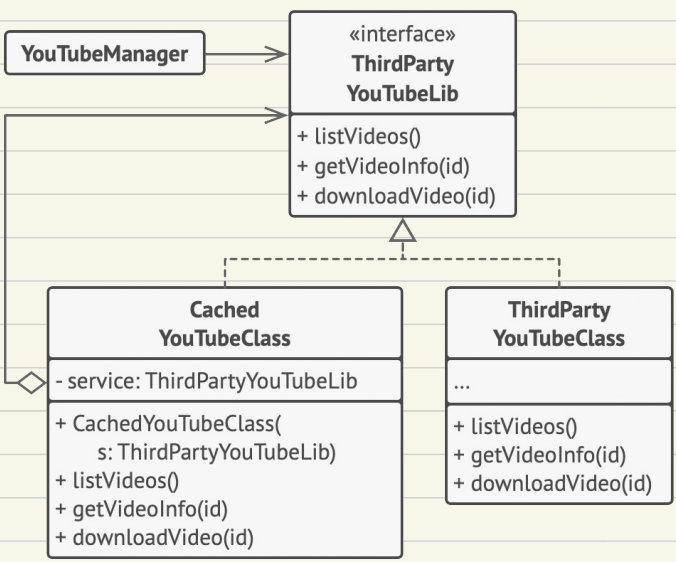
OBJETO: PROXY

INTENÇÃO: CONTROLAR O ACESSO AO OBJETO ORIGINAL, PERMITINDO QUE SE FAÇA ALGO ANTES OU DEPOIS DE CHEGAR AO MESMO.

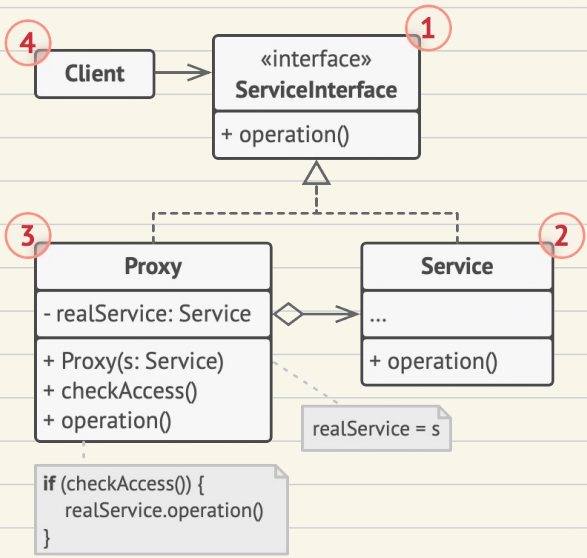
SOLUÇÃO:

- CRIAR UMA CLASSE PROXY COM A MESMA INTERFACE DO OBJETO ORIGINAL
- É DADO AO CLIENTE A REFERÊNCIA AO PROXY, EM VEZ DO ORIGINAL
- TODAS AS OPERAÇÕES DO CLIENTE PASSAM PELO PROXY, PERMITINDO O PROXY REALIZAR PROCESSAMENTO ADICIONAL

EXEMPLO:



ESTRUTURA:



1. A INTERFACE DE SERVIÇO É A INTERFACE QUE O PROXY DEVE SEGUIR PARA SER CAPAZ DE SE DISFARÇAR COMO UM OBJETO DO SERVIÇO.

2. O SERVIÇO É UMA CLASSE QUE FORNECE ALGUMA LÓGICA ÚTIL.

3. A CLASSE PROXY TEM UM CAMPO DE REFERÊNCIA QUE APONTA PARA UM OBJETO DO SERVIÇO. APÓS O PROXY ACABAR O SEU PROCESSAMENTO (EX: LAZY INITIALISATION , LOGGING, ACCESS CONTROL, CACHING, ETC.), ELE PASSA O PEDIDO PARA O OBJETO DO SERVIÇO.
   GERALMENTE OS PROXIES DÃO MANAGE A TODO O CICLO DE VIDA DOS SEUS OBJETOS DE SERVIÇO.

4. O CLIENTE DEVE TRABALHAR COM OS SERVIÇOS E OS PROXIES ATRAVÉS DA MESMA INTERFACE. ASSIM PODE-SE PASSAR UMA PROXY PARA QUALQUER CÓDIGO QUE ESPERA UM OBJETO DO SERVIÇO.

USOS DO PROXY:

DISTRIBUTED OBJECTS

O CLIENTE E O OBJETO ESTÃO EM PROCESSOS OU MÁQUINAS DIFERENTES, ENTÃO UMA CHAMADA DIRETA NÃO FUNCIONARÁ. O PAPEL DO PROXY É PASSAR A CHAMADA DO MÉTODO ATRAVÉS DOS LIMITES DE PROCESSO OU MÁQUINA E RETORNAR OS RESULTADOS PARA O CLIENTE.

SECURE OBJECTS

DIFERENTES CLIENTES TÊM NÍVEIS DIFERENTES DE PRIVILÉGIOS DE ACESSO A UM OBJETO.
OS CLIENTES ACEDEM AO OBJETO POR MEIO DE UM PROXY.
O PROXY PERMITE OU REJEITA ESTA CHAMADA DEPENDENDO DO MÉTODO QUE ESTÁ A SEE CHAMADO E DE QUEM ESTÁ A FAZER A CHAMADA.

LAZY LOADING

ALGUNS OBJETOS SÃO CAROS PARA INSTANCIAR (EX. CONSOMEM MUITOS RECURSOS OU LEVAM MUITO TEMPO PARA INICIALIZAR).
EM VEZ DISSO, CRIA-SE UM PROXY E DÁ-SE O PROXY AO CLIENTE.
O PROXY CRIA O OBJETO SOB DEMANDA QUANDO O CLIENTE O USA PELA PRIMEIRA VEZ.
OS PROXIES DEVEM ARMAZENAR AS INFORMAÇÕES NECESSÁRIAS PARA CRIAR O OBJETO DINAMICAMENTE

COPY-ON-WRITE

MÚLTIPLOS CLIENTES PODEM ACEDER O MESMO OBJETO, DESDE QUE NINGUÉM O TENTE ALTERAR.
QUANDO UM CLIENTE TENTA ALTERAR O OBJETO, ELE RECEBE SUA PRÓPRIA CÓPIA PRIVADA DO OBJETO. CLIENTES APENAS DE LEITURA CONTINUAM A VER O OBJETO ORIGINAL, ENQUANTO CLIENTES ESCRITORES RECEBEM AS SUAS PRÓPRIAS CÓPIAS. ISTO PERMITE A PARTILHA DE RECURSOS, FAZENDO PARECER QUE CADA UM TEM O SEU PRÓPRIO OBJETO.
QUANDO OCORRE UMA OPERAÇÃO DE ESCRITA, O PROXY FAZ UMA CÓPIA PRIVADA DO OBJETO DINAMICAMENTE PARA ISOLAR OS OUTROS CLIENTES DAS MUDANÇAS.

RESUMO

ADAPTER

CORRESPONDÊNCIA DE INTERFACES DE DIFERENTES CLASSES.

BRIDGE

SEPARA A INTERFACE DE UM OBJETO DA SUA IMPLEMENTAÇÃO.

COMPOSITE

UMA ESTRUTURA DE ÁRVORE DE OBJETOS SIMPLES E COMPOSTOS.

DECORATOR

ADICIONAR RESPONSABILIDADES A OBJETOS DINAMICAMENTE.

FACADE

UMA CLASSE ÚNICA QUE REPRESENTA UM SUBSISTEMA INTEIRO.

FLYWEIGHT

INSTÂNCIA USADA PARA PARTILHA EFICIENTE.

PROXY

UM OBJETO QUE REPRESENTA OUTRO OBJETO.

CHAPT. 6

DESIGN PATTERNS

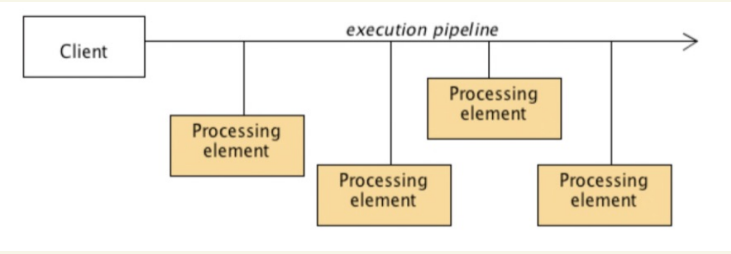
BEHAVIOUR

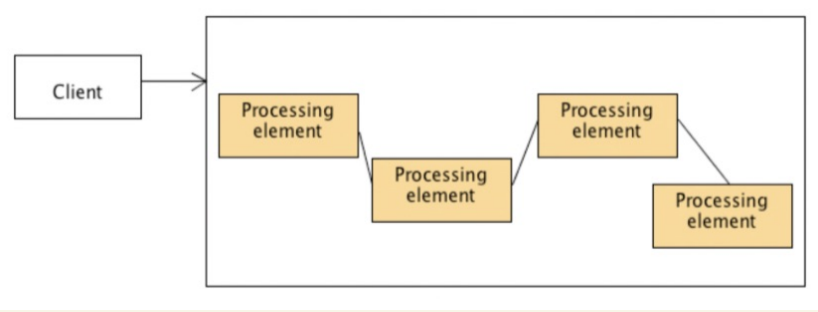
CHAIN OF RESPONSIBILITY

INTENÇÃO PASSAR PEDIDOS POR UMA CORRENTE DE HANDLERS

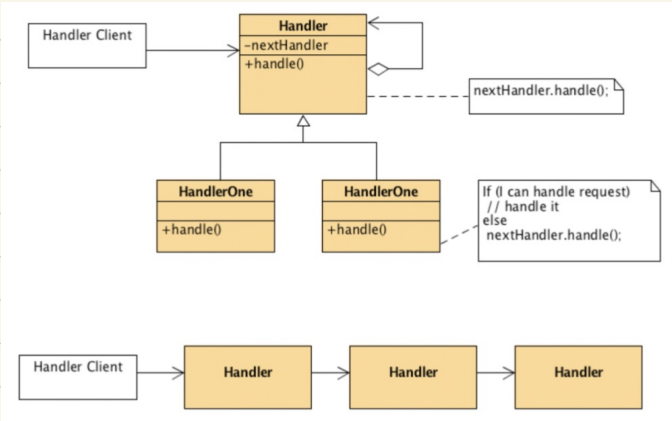
PROBLEMA PROCESSAR OS PEDIDOS DE FORMA EFICIENTE, SEM MAPEAMENTO OU RELAÇÕES DE DEPENDÊNCIA

SOLUÇÃO AO RECEBER UM PEDIDO, CADA HANDLER DECIDE SE PROCESSA O PEDIDO OU O PASSA ADIANTE PARA O PRÓXIMO HANDLER NA CORRENTE.





ESTRUTURA



CHECKLIST

- A CLASSE BASE TEM UM PONTEIRO PARA O NEXT
- CADA CLASSE DERIVADA DÁ O SEU CONTRIBUTO PARA O PROCESSAMENTO DO PEDIDO
- CASO O PEDIDO TENHA DE SER PASSADO PARA OUTRO, A CLASSE CHAMA O HANDLER BASE, QUE APONTA PARA O NEXT HANDLER
- O CLIENTE CRIA E LIGA A CORRENTE
- O CLIENTE PASSA CADA PEDIDO PELA RAIZ DA CORRENTE

```
abstract class Parser {
    private Parser successor = null;

    public void parse(String fileName) {
        if (successor != null)
            successor.parse(fileName);
        else
            System.out.println("No parser for the file: " + fileName);
    }

    protected boolean canHandleFile(String fileName, String format) {
        return (fileName == null) || (fileName.endsWith(format));
    }

    public Parser setSuccessor(Parser successor) {
        this.successor = successor;
        return this;
    }
}
```

COMMAND

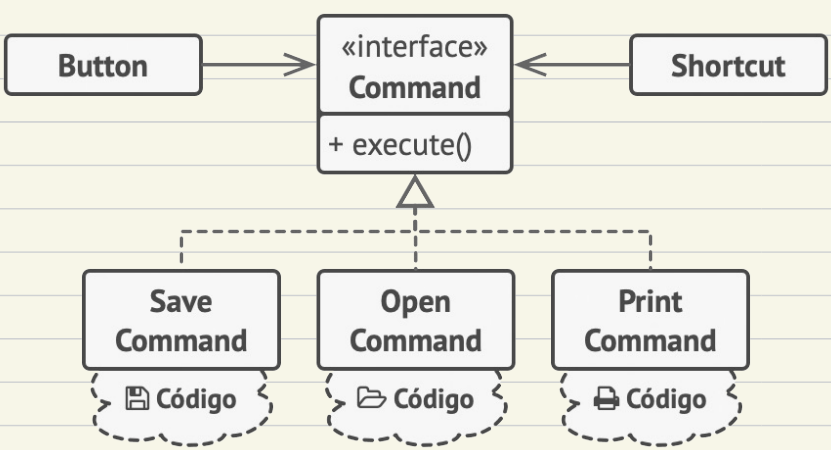
INTENÇÃO TRANSFORMAR UM PEDIDO NUM OBJETO INDEPENDENTE

PROBLEMA FAZER PEDIDOS A OBJETOS SEM SABER SOBRE A OPERAÇÃO QUE ESTÁ A SER PEDIDA OU O RECETOR DO PEDIDO

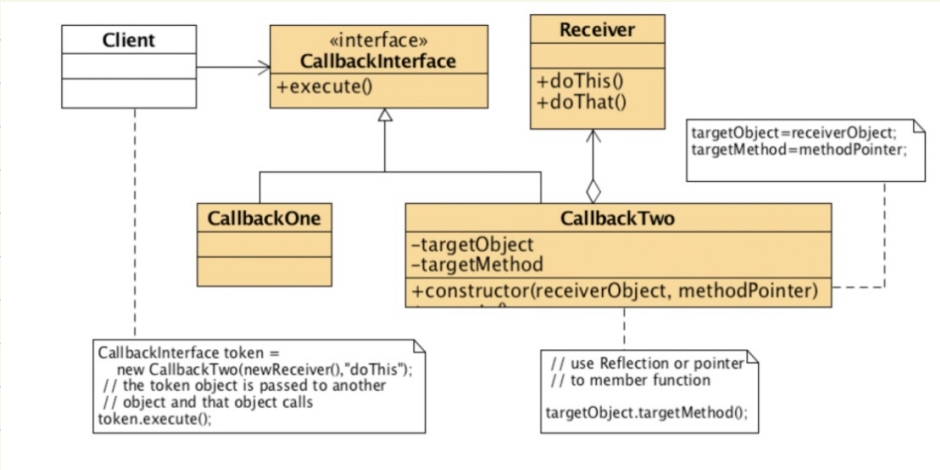
SOLUÇÃO DIVIDIR A APLICAÇÃO EM CAMADAS, DISTINGUINDO A INTERFACE INVOCADORA (ACTIONLISTENER) DOS OBJETOS QUE VÃO IMPLEMENTAR O PEDIDO

QUANDO USAR: SE OS PEDIDOS TIVEREM DE SER PROCESSADOS EM VÁRIOS MOMENTOS DIFERENTES OU POR ORDENS DIVERSAS

EXEMPLO:



ESTRUTURA

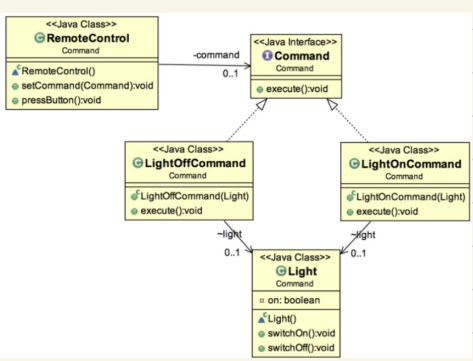



```
// Receiver
class Light {
    private boolean on;
    public void switchOn()  { on = true; }
    public void switchOff() { on = false; }
}


// Invoker
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
```

```
//Command
interface Command {
    public void execute();
}
```

```
// Concrete Command
class LightOnCommand implements Command {
    // reference to the light
    Light light;
    public LightOnCommand(Light light)    { this.light = light;  }
    public void execute()                 { light.switchOn();  }
}

// Concrete Command
class LightOffCommand implements Command {
    // reference to the light
    Light light;
    public LightOffCommand(Light light)   { this.light = light;  }
    public void execute()                 { light.switchOff();   }
}
```

```
public class Client {
  public static void main(String[] args)  {
      RemoteControl control = new RemoteControl();

      Light light = new Light();

      Command lightsOn = new LightOnCommand(light);
      Command lightsOff = new LightOffCommand(light);

      //switch on
      control.setCommand(lightsOn);
      control.pressButton();

      //switch off
      control.setCommand(lightsOff);
      control.pressButton();
```



ITERATOR

INTENÇÃO PERCORRER OBJETOS DE UMA COLLECTION SEM EXPOR A SUA ORGANIZAÇÃO (LISTA, ÁRVORE...)

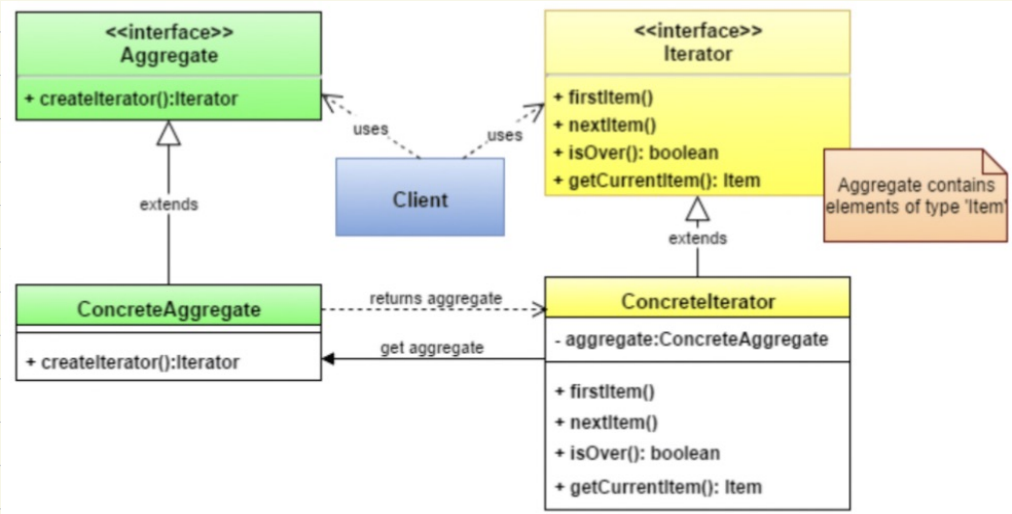
PROBLEMA EM COLEÇÕES MAIS COMPLEXAS PODEM HAVER VÁRIAS FORMAS DE PERCORRER A COLEÇÃO (É CONTRA INTUITIVO IMPLEMENTA-LAS TODAS)

SOLUÇÃO EXTRAIR O COMPORTAMENTO DE TRAVESSIA DA COLEÇÃO (TRAVERSAL BEHAVIOUR) PARA UM OBJETO SEPARADO (ITERADOR)

CHECK LIST

- ADICIONAR UM MÉTODO ITERATOR() NA CLASSE DA COLEÇÃO, DANDO À CLASSE ITERATOR ACESSO.
- CONSTRUIR A CLASSE ITERATOR QUE PERMITA PERCORRER A COLEÇÃO
- O CLIENTE PEDE AO OBJETO DA COLEÇÃO PARA CRIAR O OBJETO ITERATOR
- O CLIENTE USA HASNEXT(), NEXT() PARA ACEDER AOS ELEMENTOS DA COLEÇÃO.

ESTRUTURA




```java
// For a set or list
for (Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next();
}
// For keys of a map
for (Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next();
}
// For values of a map
for (Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next();
}
// For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
}
```

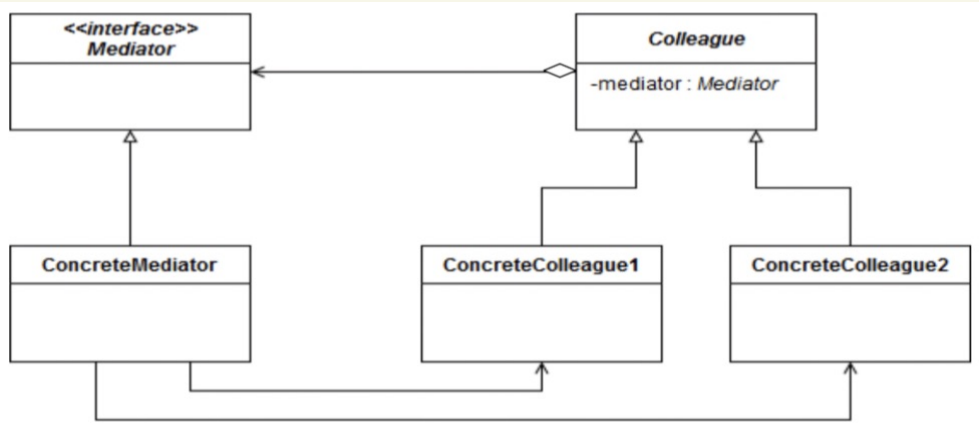
MEDIATOR

INTENÇÃO REDUZIR AS DEPENDÊNCIAS E COMUNICAÇÕES DIRETAS ENTRE OBJETOS (USAR UM OBJETO MEDIADOR)

PROBLEMA AS DEPENDÊNCIAS ENTRE OBJETOS TORNA OS DIFÍCIL DE REUTILIZAR

SOLUÇÃO COLABORAÇÃO DIRETA, ATRAVÉS DE UM MEDIADOR

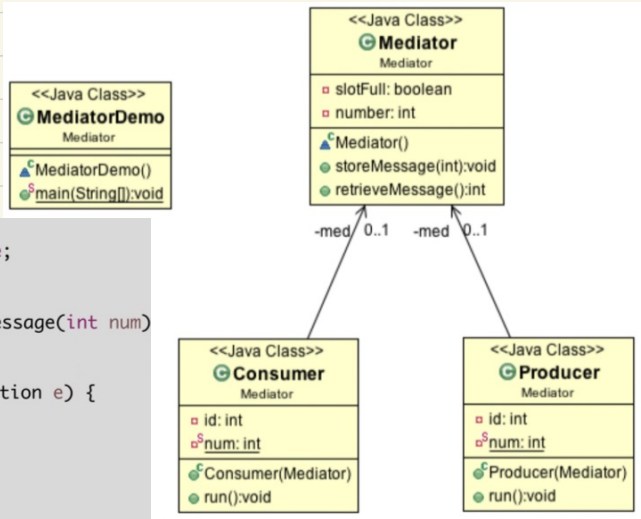
EXEMPLO:



CHECK LIST

- ENCAPSULAR AS INTERAÇÕES NUMA MEDIATOR
- CRIAR UMA INSTÂNCIA DO MEDIATOR EM TODAS AS CLASSES QUE INTERAGEM
- EQUILIBRAR O PRINCIPIO DE DECOUPLING COM A DE DISTRIBUIÇÃO DE RESPONSABILIDADES

ESTRUTURA




```
class Mediator {
    private boolean slotFull = false;
    private int number;

    public synchronized void storeMessage(int num)
        while (slotFull == true) {
            try {  wait();
            } catch (InterruptedException e) {
                // ...
            }
        }
        slotFull = true;
        number = num;
        notifyAll();
    }

    public synchronized int retrieveMessage() {
        // ...
    }
}
```

```
class Producer extends Thread {
    // 2. Producers are coupled only to the Mediator
    private Mediator med;
    private int id;
    private static int num = 1;

    public Producer(Mediator m) {
        med = m;
        id = num++;
    }

    public void run() {
        int num;
        while (true) {
            med.storeMessage(num = (int) (Math.random() * 100));
            System.out.println("p" + id + "-" + num + "   ");
        }
    }
}
```

```
class MediatorDemo {
    public static void main(String[] args) {
        Mediator mb = new Mediator();
        new Producer(mb).start();
        new Producer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
    }
```

```
class Consumer extends Thread {
    // 3. Consumers are coupled only to the Mediator
    private Mediator med;
    private int id;
    private static int num = 1;

    public Consumer(Mediator m) {
        med = m;
        id = num++;
    }

    public void run() {
        while (true) {
            System.out.println("\tc"+ id + "-" + med.retrieveMessage()+" ");
```

MEMENTO

INTENÇÃO SALVAR E RESTAURAR O ESTADO ANTERIOR DE UM OBJETO (UNDO/ROLLBACK)

PROBLEMA ACESSO A TODOS OS ATRIBUTOS NEM SEMPRE É POSSÍVEL

SOLUÇÃO DELEGAR A CRIAÇÃO DOS 'SNAPSHOTS' (RETRATOS) AO DONO DO ESTADO

OS 3 PAPEIS:

ORIGINATOR:

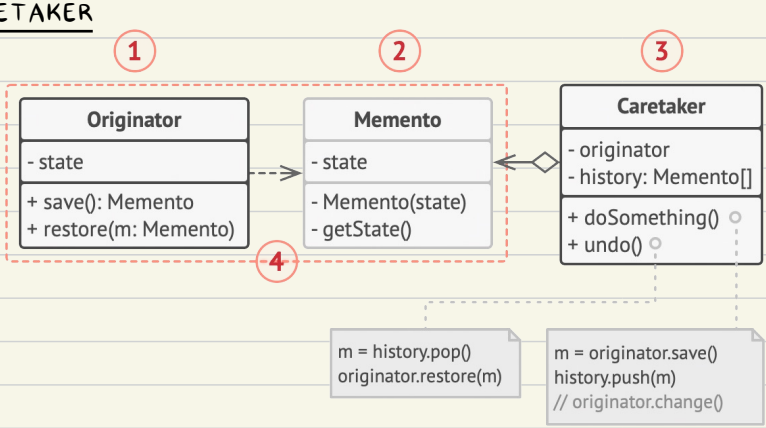
- O OBJETO QUE TEM AS FUNÇÕES DE SAVE() E RESTORE() PARA SI MESMO

CARETAKER:

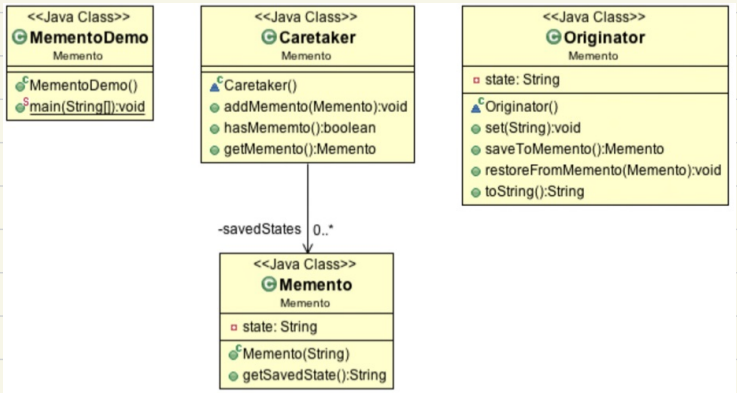
- O OBJETO QUE SABE PORQUE E QUANDO É QUE O ORIGINATOR PRECISA DE DAR SAVE OU RESTAURAR

MEMENTO:

- SNAPSHOT QUE É ESCRITO E LIDO PELO ORIGINATOR E GERIDO PELO CARETAKER



ESTRUTURA



CHECKLIST

- IDENTIFICAR OS PAPÉIS DO CARETAKER E DO ORIGINATOR
- CRIAR A CLASSE MEMENTO
- O CARETAKER SABE QUANDO GUARDAR O ESTADO DO ORIGINATOR
- O ORIGINATOR CRIA UM MEMENTO E COPIA O SEU ESTADO PARA EST
- O CARETAKER RECEBE O MEMENTO E ARMAZENA-O (NÃO O LÊ)
- O CARETAKER SABE QUANDO DAR ROLL BACK AO ORIGINATOR
- O ORIGINATOR RETOMA O ESTADO ANTERIOR COM RECURSO AO ESTADO GUARDADO NO MEMENTO

```
class Memento {
   private String state;
   public Memento(String stateToSave) { state = stateToSave;
   public String getSavedState() { return state; }
}

class Originator {
   private String state; // simple example
   public void set(String state) {
       this.state = state;
   }
   public Memento saveToMemento() {
       return new Memento(state);
   }
   public void restoreFromMemento(Memento m) {
       state = m.getSavedState();
   }
   @Override public String toString() { return state; }
}
```

```
class Caretaker {
   private Stack<Memento> savedStates = new Stack<Memento>();

   public void addMemento(Memento m) {
       savedStates.push(m);
   }
   public boolean hasMemento() {
       return !savedStates.isEmpty();
   }
   public Memento getMemento() {
       return savedStates.pop();
```

```
public class MementoDemo {
   public static void main(String[] args) {
       Caretaker caretaker = new Caretaker();

       Originator originator = new Originator();
       for (int i= 1; i<=5; i++) {
          originator.set("State " + i);
          System.out.println("Originator: state set to "+ originator);
          caretaker.addMemento( originator.saveToMemento() );
          System.out.println("Memento saved");
       }

       while (caretaker.hasMemento()) {
          originator.restoreFromMemento( caretaker.getMemento() );
          System.out.println("Originator: after restore: "+originator);
```

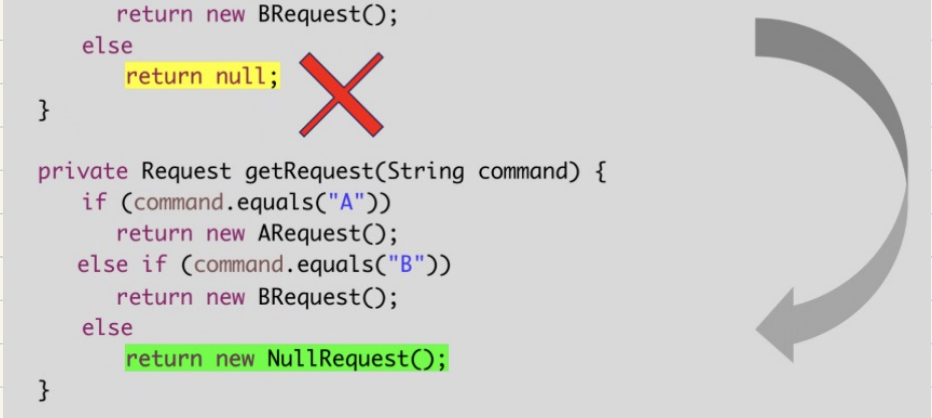
NULL OBJECT

INTENÇÃO TRATAR DA AUSÊNCIA DE UM OBJETO, FORNECENDO UMA ALTERNATIVA ADEQUADA AO COMPORTAMENTO DE NÃO FAZER NADA

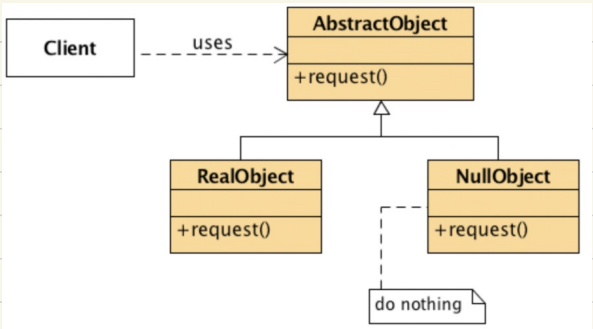
PROBLEMA TRATAR A AUSÊNCIA DE UM OBJETO, ABSTRAINDO O CLIENTE

SOLUÇÃO DEVOLVER UM OBJETO COMO RESPOSTA A NÃO ENCONTRAR A RESPOSTA AO REQUEST DADO

```
private Request getRequest(String command) {
    if (command.equals("A"))
       return new ARequest();
    else if (command.equals("B"))
       return new BRequest();
    else
        return null;
}

private Request getRequest(String command) {
    if (command.equals("A"))
       return new ARequest();
    else if (command.equals("B"))
       return new BRequest();
    else
        return new NullRequest();
}
```



ESTRUTURA




```
abstract class Emp
{
    protected String name;
    public abstract boolean isNull();
    public abstract String getName();
}
```

```
class NoClient extends Emp
{
    @Override
    public String getName()
    {
        return "Not Available";
    }

    @Override
    public boolean isNull()
    {
        return true;
    }
}
```

```
class Coder extends Emp
{
    public Coder(String name)
    {
        this.name = name;
    }
    @Override
    public String getName()
    {
        return name;
    }
    @Override
    public boolean isNull()
    {
        return false;
    }
}
```

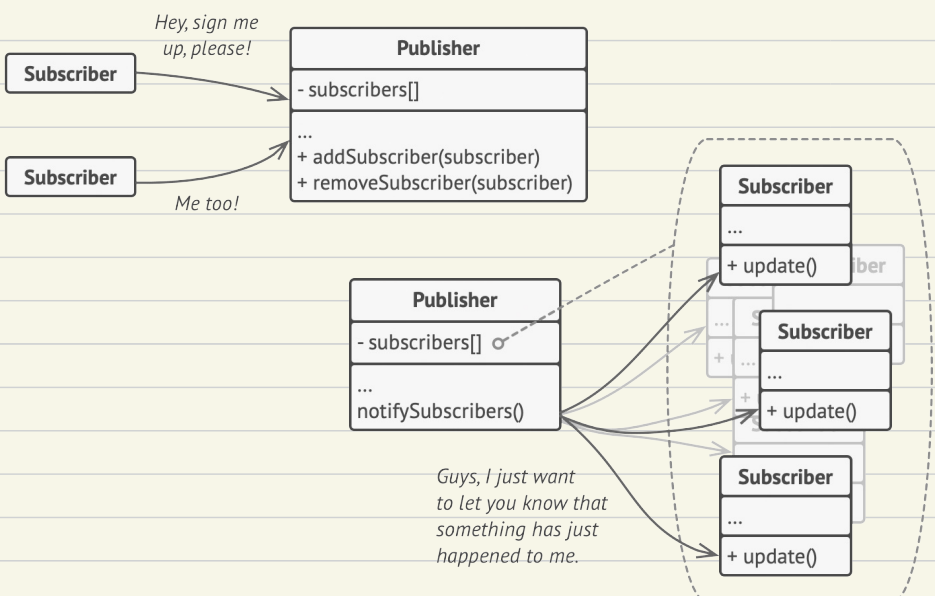
```
class EmpData
{

    public static final String[] names = {"Lokesh", "Kushagra",
    public static Emp getClient(String name)
    {
        for (int i = 0; i < names.length; i++)
        {
            if (names[i].equalsIgnoreCase(name))
            {
                return new Coder(name);
            }
        }
        return new NoClient();
    }
```

OBSERVER

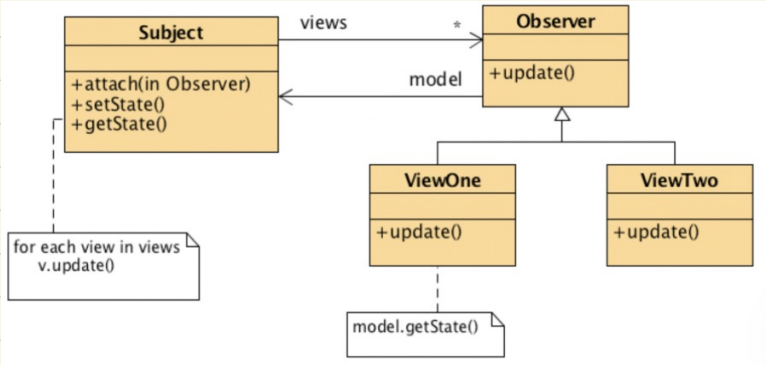
INTENÇÃO DEFINIR UMA RELAÇÃO ONE-TO-MANY ENTRE DOIS OBJETOS, DE FORMA A QUE QUANDO O ONE MUDA DE ESTADO, TODOS OS SEUS DEPENDENTES SÃO NOTIFICADOS AUTOMATICAMENTE

PROBLEMA OS CLIENTES NÃO QUEREM ESTAR CONSTANTEMENTE A VERIFICAR SE O ESTADO MUDOU

SOLUÇÃO CRIAR UM MECANISMO DE SUBSCRIÇÃO NO PUBLICADOR (ONE), QUE OFERECE MÉTODOS PARA NOTIFICAR OS SUBSCRITORES (MANY)

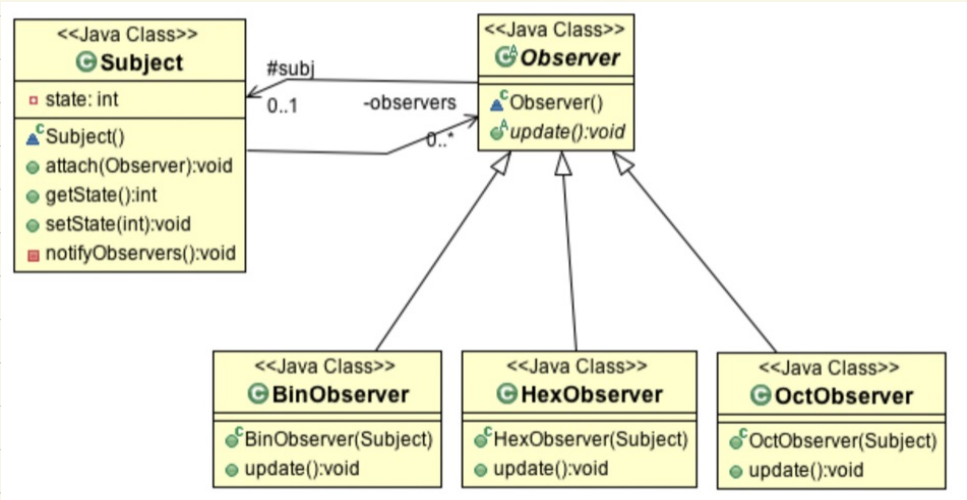


ESTRUTURA



CHECK LIST

- DISTINGUIR AS FUNCIONALIDADES PRINCIPAIS INDEPENDENTES DAS OPCIONAIS E DEPENDENTES
- MODELAR AS INDEPENDENTES NOS PUBLICADORES E AS DEPENDENTES NOS OBSERVADORES
- O PUBLICADOR TEM UMA REFERÊNCIA (LISTA) DOS OBSERVADORES
- OS OBSERVADORES REGISTAM-SE NO PUBLICADOR
- O PUBLICADOR NOTIFICA OS OBSERVADORES DE ALTERAÇÕES



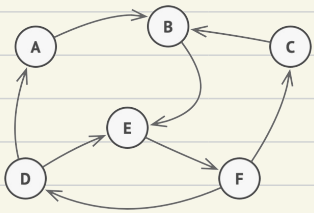
STATE

INTENÇÃO: PERMITIR QUE UM OBJETO ALTERE O SEU COMPORTAMENTO DE ACORDO COM ALTERAÇÕES NO SEU ESTADO INTERNO

PROBLEMA: COMO O PRÓXIMO ESTADO É DEPENDENTE DO ESTADO ATUAL, À MEDIDA QUE ADICIONAMOS MAIS ESTADOS É NECESSÁRIO ADICIONAR MAIS CONDIÇÕES, TORNANDO-SE DIFÍCIL MANTER A LONGO PRAZO COM EVENTUAIS ALTERAÇÕES

SOLUÇÃO: CRIAR NOVAS CLASSES PARA CADA ESTADO E EXTRAIR O COMPORTAMENTO ESPECÍFICO DE CADA ESTADO PARA DENTRO DESSAS CLASSES

```
class CeilingFanPullChain {
    private State currentState;

    public CeilingFanPullChain() {
        currentState = new Off();
    }

    public void setState(State s) {
        currentState = s;
    }

    public void pull() {
        currentState.pull(this);
    }
}

interface State {
    void pull(CeilingFanPullChain wrapper);
}
```

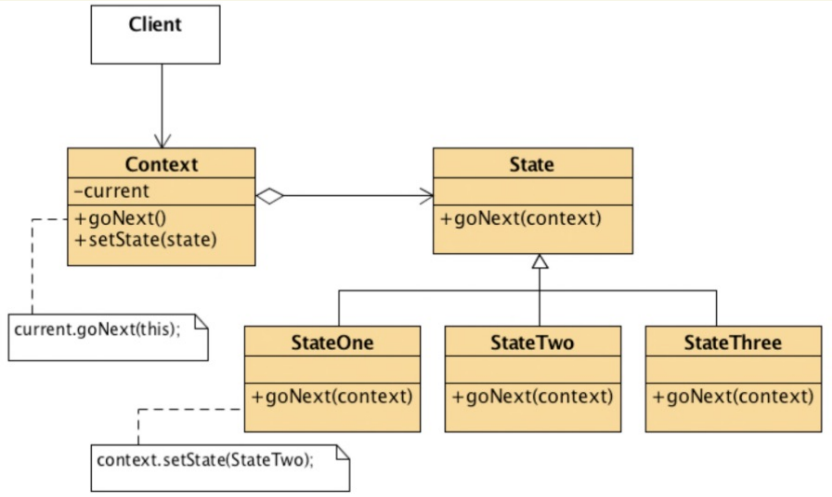



```
public class StateDemo {
    public static void main(String[] args) {
        CeilingFanPullChain1 chain = new CeilingFanPullChain1();
        while (true) {
            System.out.print("Enter.. ");
            Scanner scan = new Scanner(System.in);
            scan.nextLine();
            chain.pull();
```

```
class Off implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Low()); System.out.println(" low speed");
} }

class Low implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Medium()); System.out.println(" medium speed");
} }

class Medium implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new High()); System.out.println(" high speed");
} }

class High implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Off()); System.out.println(" turning off");
} }
```

```
Enter..
   low speed
Enter..
   medium speed
Enter..
   high speed
Enter..
   turning off
Enter..
   low speed
Enter..
   medium speed
```

ESTRUTURA



CHECK LIST

- IDENTIFICAR OU CRIAR A CLASSE STATE MACHINE QUE SERÁ O WRAPPER DOS ESTADOS (CONTEXT)
- CRIAR A CLASSE BASE QUE REPLICA OS MÉTODOS DA INTERFACE DEFINIDA ANTERIORMENTE, CADA UM COM UMA INSTÂNCIA DA CLASSE WRAPPER COMO ARGUMENTO EXTRA EXTRA
- CRIAR UMA CLASSE DERIVADA PARA CADA ESTADO
- A CLASSE WRAPPER MANTÉM O ESTADO ATUAL DO OBJETO
- TODOS OS PEDIDOS DO CLIENTE SÃO DELEGADOS NO ESTADO ATUAL, SENDO PASSADO AO WRAPPER O ESTADO QUE RESULTOU DA EXECUÇÃO DO MÉTODO INVOCADO
- -> OS MÉTODOS DA CLASSE STATE FAZEM AS ALTERAÇÕES DE ESTADO

STRATEGY

INTENÇÃO DEFINIR UMA FAMÍLIA DE ALGORITMOS EM CLASSES SEPARADAS, COM OBJETOS INTERCAMBIÁVEIS

PROBLEMA A COMPLEXIDADE AUMENTA COM A INTRODUÇÃO DE NOVOS ALGORITMOS

SOLUÇÃO EXTRAIR OS ALGORITMOS PARA CLASSES SEPARADAS (STRATEGIES), ONDE CADA UMA OFERECE UMA ESTRATÉGIA PARA RESOLVER O PROBLEMA.

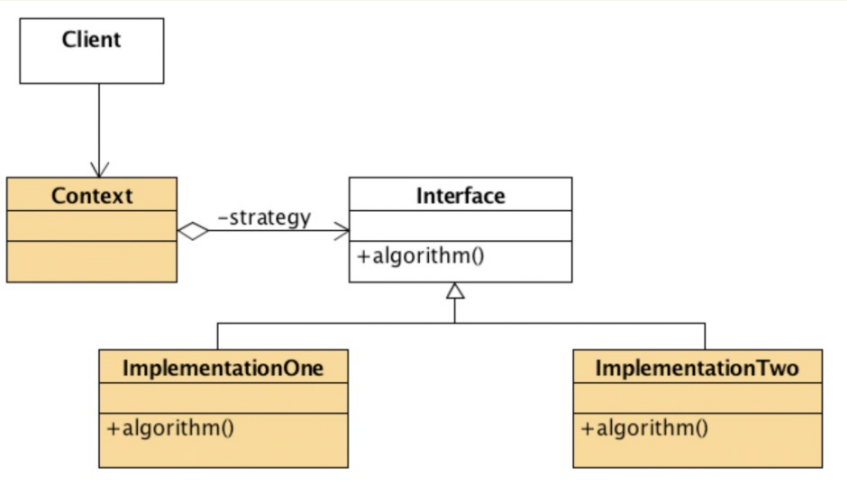
EXEMPLO

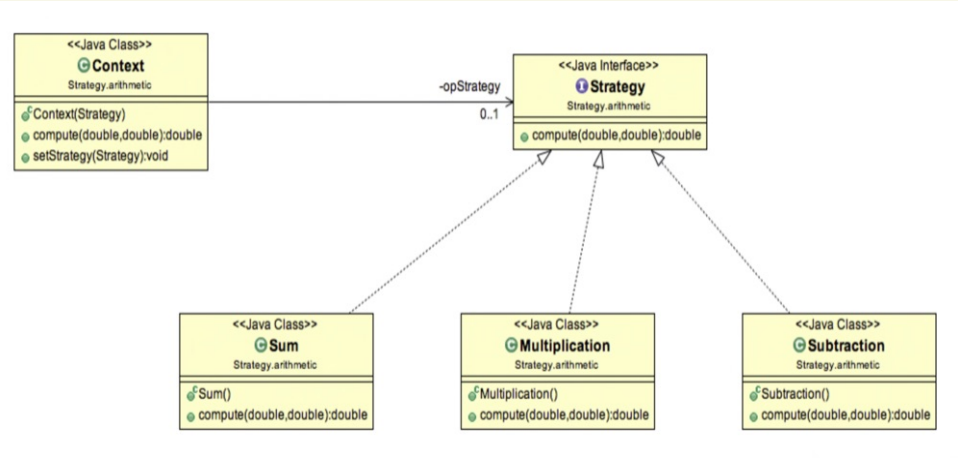
```java
public interface Strategy {  double compute(double elem1, double elem2); }

public class Sum implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 + elem2;
    }
}

public class Multiplication implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 * elem2;
    }
}

public class Subtraction implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 - elem2;
    }
}
```

```java
public class Context {
    private Strategy opStrategy;
    public Context(Strategy operation) {
        this.opStrategy = operation; }

    public double compute(double firstNumber, double secondNumber){
        return opStrategy.compute(firstNumber, secondNumber); }

    public void setStrategy(Strategy strategy){
        opStrategy = strategy;}
}

public class StrategyDemo {
    public static void main(String[] args) {
        double e1 = 5, e2 = 33;
        Context c = new Context(new Sum());
        System.out.println("Result: " + c.compute(e1, e2));
        c.setStrategy(new Subtraction());
        System.out.println("Result: " + c.compute(e1, e2));
        c.setStrategy(new Multiplication());
        System.out.println("Result: " + c.compute(e1, e2));
    }
}
```

```
Result: 38.0
Result: -28.0
Result: 165.0
```

ESTRUTURA


Client
Context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()


<<Java Class>>
Context
Strategy.arithmetic
Context(Strategy)
compute(double,double):double
setStrategy(Strategy):void
-opStrategy
0..1
<<Java Interface>>
Strategy
Strategy.arithmetic
compute(double,double):double
<<Java Class>>
Sum
Strategy.arithmetic
Sum()
compute(double,double):double
<<Java Class>>
Multiplication
Strategy.arithmetic
Multiplication()
compute(double,double):double
<<Java Class>>
Subtraction
Strategy.arithmetic
Subtraction()
compute(double,double):double

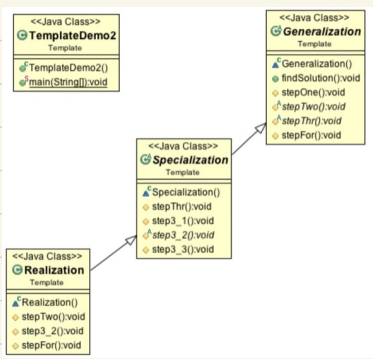
TEMPLATE METHOD

INTENÇÃO PROGRAMAR CLASSES QUE PARTILHAM MÉTODOS E ATRIBUTOS

PROBLEMA AS CLASSES SÃO SEMELHANTES HAVENDO DUPLICAÇÃO NA IMPLEMENTAÇÃO

SOLUÇÃO CRIAR UMA SUPERCLASSE QUE IMPLEMENTE OS MÉTODOS E ATRIBUTOS COMUNS, QUE PODEM SER OVERRIDEN PELAS SUBCLASSES (SUBSCREVEM ETAPAS ESPECÍFICAS, SEM MODIFICAR A ESTRUTURA)

EXEMPLO:




```
abstract class Generalization {
// 1. Standardize the skeleton of an algorithm in a "template" method
   public void findSolution() {
         stepOne();
         stepTwo();
         stepThr();
         stepFor();
   }
// 2. Common implementations of individual steps are defined in base class
   protected void stepOne()
       { System.out.println("Generalization.stepOne" ); }
 // 3. Steps requiring peculiar impls are ";placeholders" in the base class
   abstract protected void stepTwo();
   abstract protected void stepThr();
   protected void stepFor()
       { System.out.println( "Generalization.stepFor" ); }
}
```

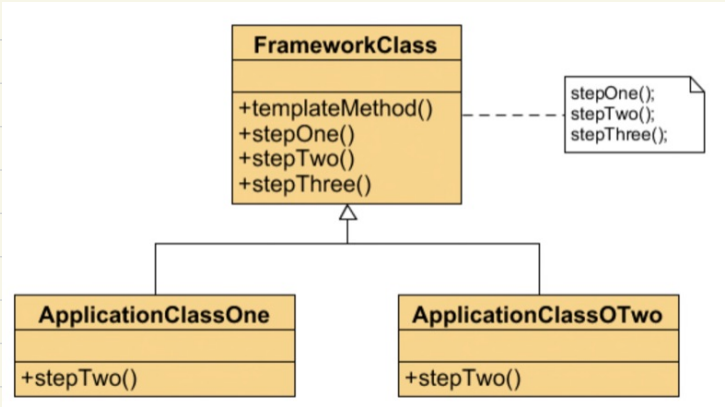
```
abstract class Specialization extends Generalization {
// 4. Derived classes can override placeholder methods
// 1. Standardize the skeleton of an algorithm in a "template" method
   protected void stepThr() {
      step3_1();
      step3_2();
      step3_3();
   }
// 2. Common implementations of individual steps are defined
   protected void step3_1()
       { System.out.println( "Specialization.step3_1" ); }
// 3. Steps requiring peculiar impls are "placeholders" in t
   abstract protected void step3_2();
   protected void step3_3()
       { System.out.println( "Specialization.step3_3" ); }
}
```

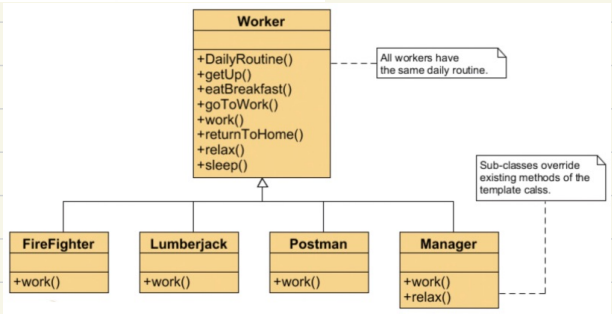
```
class Realization extends Specialization {
// 4. Derived classes can override placeholder methods
   protected void stepTwo(){ System.out.println("Realization.stepTwo" ); }
   protected void step3_2(){ System.out.println("Realization.step3_2" ); }
// 5. Derived classes can override implemented methods
// 6. Derived classes can override and "call back to" base class methods
   protected void stepFor() {
      System.out.println( "Realization.stepFor" );
      super.stepFor();
   }
}
```

```
public class TemplateDemo2 {
      public static void main( String[] args ) {
         Generalization algorithm =
             new Realization();
         algorithm.findSolution();
      }
```

```
Generalization.stepOne
Realization.stepTwo
Specialization.step3_1
Realization.step3_2
Specialization.step3_3
Realization.stepFor
Generalization.stepFor
```

ESTRUTURA





CHECK LIST

- CRIAR UMA CLASSE BASE TEMPLATE COM O ESQUELETO DAS CLASSES (COMMON IMPLEMENTATIONS DE PAÇOS INDIVIDUAIS)
- CRIAR UMA SUBCLASSE PARA CADA IMPLEMENTAÇÃO ESPECIFICA (FAZEM OVERRIDE DOS MÉTODOS)

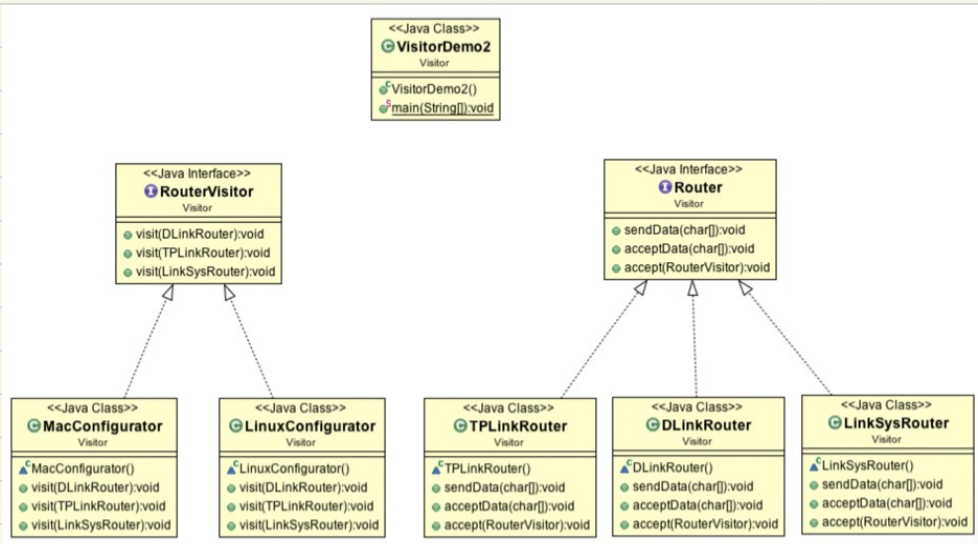
VISITOR

INTENÇÃO DEFINIR NOVAS OPERAÇÕES SEM MUDAR AS CLASSES DOS OBJETOS EM QUE OPERA

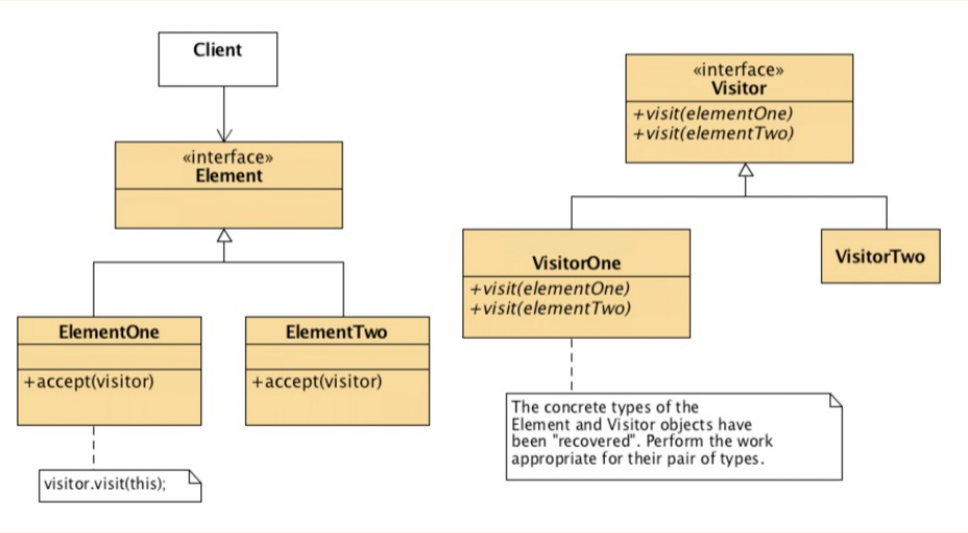
PROBLEMA TER VÁRIAS OPERAÇÕES NUM OBJETO POLUI A CLASSE

SOLUÇÃO CRIAR O VISITOR PARA CADA OPERAÇÃO, QUE DEFINE UM MÉTODO PARA CADA OBJETO, SENDO O OBJETO QUE INVOCA O VISITOR (AS ARGUMENT)

EXEMPLO:



ESTRUTURA



CHECK LIST

- CRIAR UMA INTERFACE VISITOR COM MÉTODO VISIT() PARA CADA ELEMENTO
- ADICIONAR O MÉTODO ACCEPT(VISITOR) AO ELEMENTOS
- CRIAR UMA CLASSE DERIVADA DO VISITOR PARA CADA OPERAÇÃO

RESUMO

CHAIN OF RESPONSIBILITY

UMA MANEIRA DE PASSAR UM REQUEST POR UMA CADEIA DE OBJETOS

COMMAND ECAPSULA UM REQUEST COMMAND COMO UM OBJETO

ITERATOR ACEDER SEQUENCIALMENTE ELEMENTOS DE UMA COLEÇÃO

MEDIATOR SIMPLIFICA COMUNICAÇÃO ENTRE CLASSES

MEMENTO GUARDA E RESTORA UM OBJECT'S INTERNAL STATE

NULL OBJECT ATUA COMO UM DEFAULT VALUE DE UM OBJETO

OBSERVER MANEIRA DE NOTIFICAR MUDANÇAS A VÁRIAS CLASSES

STATE

ALTERA O COMPORTAMENTO DE UM OBJETO QUANDO MUDA DE ESTADO

STRATEGY ENCAPSULA UM ALGORITMO DENTRO DE UMA CLASSE

TEMPLATE METHOD

PASSA ETAPAS EXATAS DE UM ALGORITMO PARA UA SUBCLASSE

VISITOR DEFINE NOVAS OPERAÇÕES A UMA CLASSE, SEM A MUDAR

CHAPT 7

PADRÕES DE ARQUITETURA DE SOFTWARE

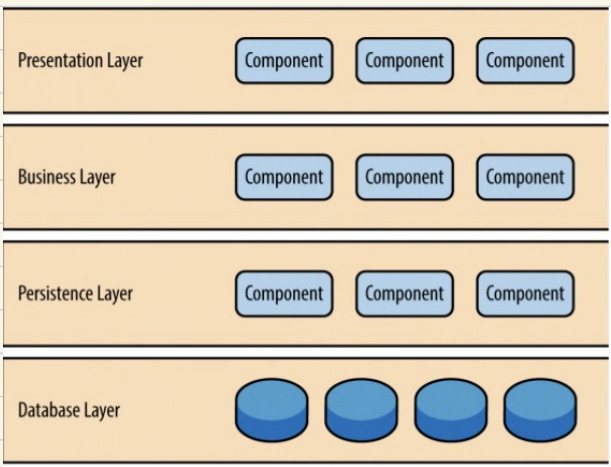
- DEFINEM CARACTERÍSTICAS BÁSICAS E COMPORTAMENTO DE APLICAÇÕES. MODOS A COMO RESPONDER A PROBLEMAS COMUNS À PROGRAMAÇÃO

LAYERED ARCHITECTURE

- MAIS CONHECIDO (ALSO N-TIER ARCHITECTURE PATTERN)

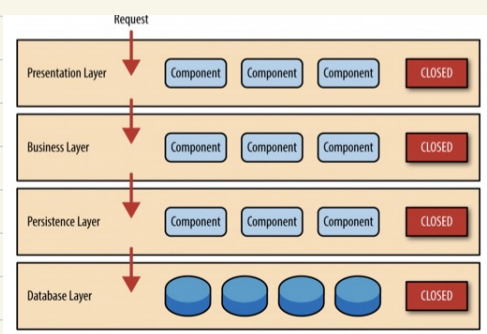
CARACTERÍSTICA PRINCIPAL

SEPARAÇÃO DE PAPEIS - CADA CAMADA É UMA CAMADA DE ABSTRAÇÃO



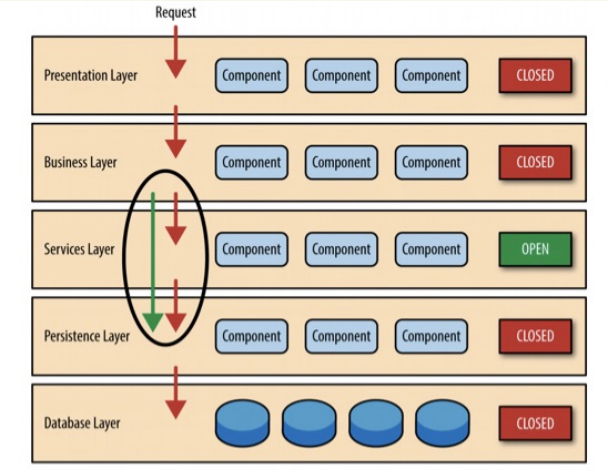
CLOSED LAYERS

- UM REQUEST SÓ MOVE PARA A LAYER INFERIOR À MESMA
- LAYERS OF ISOLATION - MUDANÇAS NUMA CAMADA NÃO AFETAM OS COMPONENTES DAS RESTANTES



OPEN LAYERS

- QUANDO CERTOS SERVIÇOS NÃO SÃO NECESSÁRIOS NA LAYER SEGUINTE MAS SIM NAS RESTANTES. ASSIM É "IGNORADO" PELA LAYER EM QUE NÃO É PRECISO



- PODE OCORRER O SINKHOLE ANTI-PATTERN (PEDIDOS PASSAM POR MUITAS CAMADAS SEM SEREM PROCESSADOS), ONDE SE DEVE APLICAR A 80-20 RULE, ONDE APENAS 20% DOS PEDIDOS SÃO ENCAMINHADOS

Agilidade Facilidade de responder a mudanças constantes no ambiente	Baixa	Apesar das mudanças nas camadas serem isoladas, continua a haver um acoplamento elevado dos componentes.
Facilidade de *deploy*	Baixa	Particularmente complicado em aplicações maiores, pois uma pequena alteração num componente pode criar a necessidade de *deploy* de toda a aplicação.
Facilidade de testes	Alta	Devido à independência entre as camadas, estas podem ser testadas individualmente.
Performance	Baixa	O facto dos pedidos terem de percorrer várias camadas pode levar a quebras no desempenho.
Escalabilidade	Baixa	Devido ao acoplamento elevado entre os seus componentes, geralmente a escalabilidade é reduzida. Uma possibilidade é a divisão das camadas em implementações físicas distintas, o que no entanto aumenta a granularidade, elevando o custo da mesma.
Facilidade de desenvolvimento	Alta	Devido à sua popularidade e simplicidade de implementação. Tem uma ligação forte à maneira como as empresas comunicam e se organizam (ver Google "Conway's law").

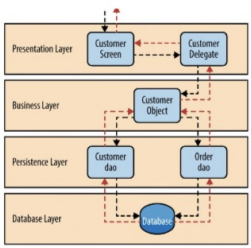
Ex1:

Para obter informação de um cliente na aplicação de uma empresa, o monitor do cliente é resopinsável por receber o pedido e mostrar a informação associada. Este não sabe onde os dados são armazenados, nem como são obtidos, limitando-se a encaminhar o pedido para o módulo *costumer delegate*, que contacta um módulo na camada inferior.

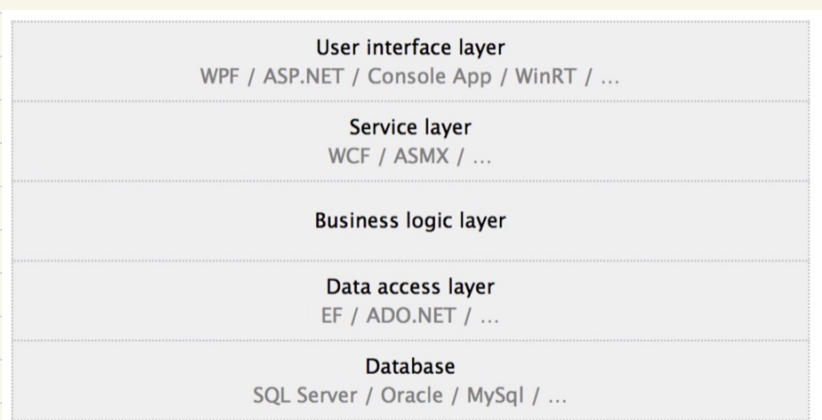
O *costumer object*, por sua vez, agrega toda a informação necessária para responder ao pedido, chamando dois módulos da camada inferior para a obter.

Finalmente os módulos *costumer dao (data acess object)* e *order dao*, vão executar *queries* de consulta dos dados da base de dados, na camada inferior.

As respostas são depois enviadas no sentido inverso, até chegar à camada de apresentação, onde a informação é então apresentada.



Ex2:



EVENT-DRIVEN ARCHITECTURE

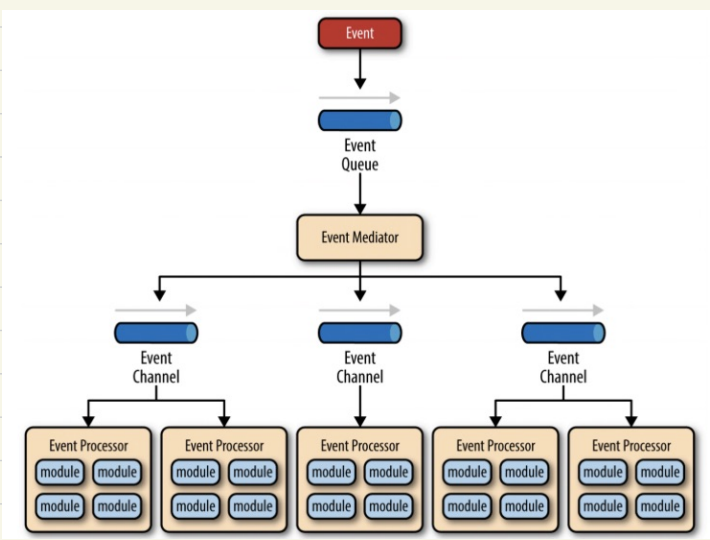
- MAIS CONHECIDO POR MESSAGE-DRIVEN ARCHITECTURE OU STREAM PROCESSING ARCHITECTURE

- ARQUITETURA DISTRIBUÍDA ASSÍNCRONA, USADA PARA CRIAR APLICAÇÕES ESCALÁVEIS , EXTREMAMENTE ADAPTÁVEL

MEDIATOR TOPOLOGY

- PARA EVENTOS QUE TÊM MÚLTIPLOS PASSOS, É PRECISO UM MEDIADOR PARA OS ORGANIZAR

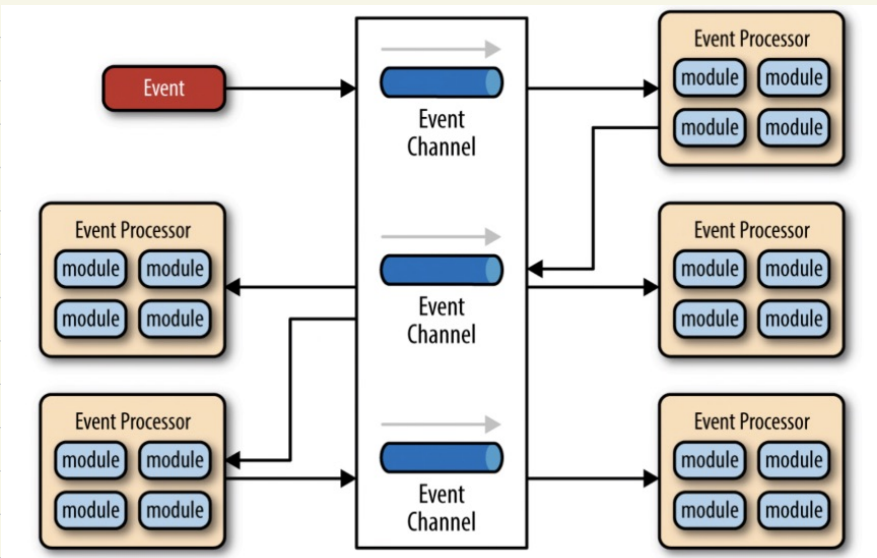
- TEM COMPONENTES INDIVIDUAIS DESACOPLADOS, DIVIDIDOS EM QUATRO TIPOS:
  - EVENT QUEUES
  - EVENT MEDIATOR
  - EVENT CHANNELS
  - EVENT PROCESSORS



BROKER TOPOLOGY

- NÃO HÁ MEDIADOR CENTRAL, USADO PARA FLUXO DE PROCESSAMENTO SIMPLES

- DIVIDIDO EM DUAS CATEGORIAS:
  - BROKER COMPONENT
  - EVENT PROCESSOR COMPONENT



- PADRÃO COMPLEXO, DEVIDO AO DESACOPLAMENTO E INDEPENDÊNCIA DOS COMPONENTES É DIFÍCIL MANTER UMA UNIDADE DE TRABALHO TRANSACIONAL

KEY ASPECTS DOS COMPONENTES QUE PROCESSAM EVENTOS

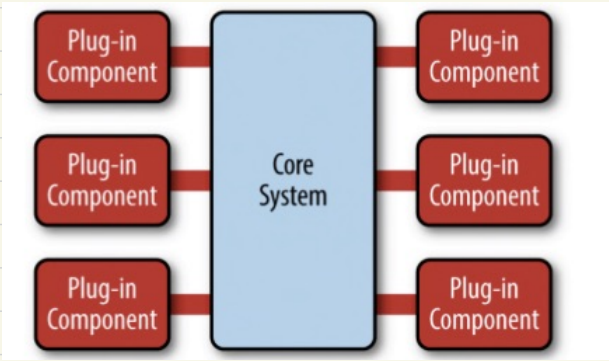
- CREATION
- MAINTENANCE
- GOVERNANCE

Agilidade Facilidade de responder a mudanças constantes no ambiente	Alta	Uma vez que os componentes de processamento têm um único propósito e estão desacoplados, as modificações são isoladas a um número restrito de processadores.
Facilidade de *deploy*	Alta	Novamente a natureza desacoplada dos componentes permite um *deploy* simples. A topologia do *broker* é mais simples do que a do *mediator*, pois no último os mediadores e os processadores estão de alguma forma conectados, pelo que uma alteração num componente implica a alteração no mediador.
Facilidade de testes	Baixa	Testes dos componentes individuais são relativamente simples. No entanto, são necessários testes especializados para gerar eventos.
Performance	Alta	A possibilidade de realizar operações assíncronas, desacopladas e em paralelo permite uma alta performance.
Escalabilidade	Alta	Novamente o desacoplamento e a independência dos componentes permitem uma escalabilidade alta e específica para certos componentes.
Facilidade de desenvolvimento	Baixa	A natureza assíncrona cria algumas dificuldades no desenvolvimento, nomeadamente devido à necessidade da gestão de erros como processadores de eventos não responsivos ou *brokers* indisponíveis.

MICROKERNEL ARCHITECTURE

- MAIS CONHECIDO POR PLUG-IN ARCHITECTURE

- IMPLEMENTA FUNCIONALIDADES EXTRA PARA UMA APLICAÇÃO DE FORMA ISOLADA, DANDO EXTENSIBILIDADE E FEATURE SEPARATION



- SEPARADO EM DOIS COMPONENTES:
  - CORE SYSTEM
  - PLUG-IN MODULES

CORE SYSTEM CONTÉM APENAS AS FUNCIONALIDADES BÁSICAS E ESTRITAMENTE NECESSÁRIAS AO FUNCIONAMENTO DO SISTEMA

PLUG-IN MODULES PODEM SER CONECTADOS AO CORE POR VÁRIAS MANEIRAS:

- OSGI (OPEN SERVICE GATEWAY INITIATIVE
- MESSAGING
- WEB SERVICES
- POINT-TO-POINT BINDING

- O BENEFÍCIO DESTE PADRÃO É A SUA GRANDE ABERTURA AO DESENVOLVIMENTO INCREMENTAL

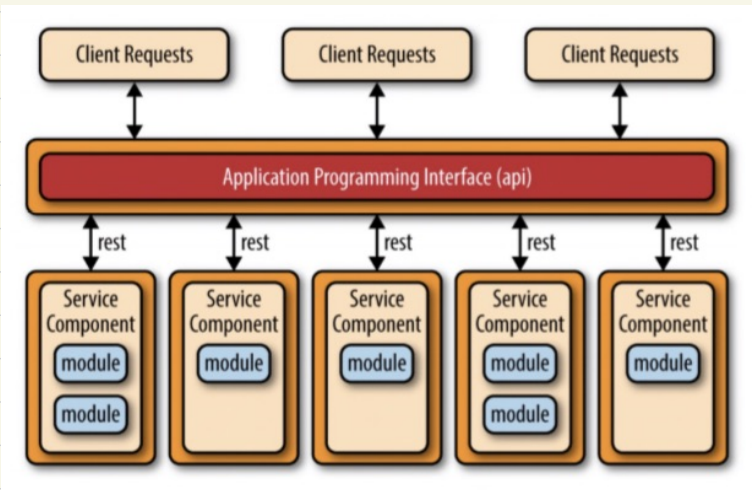
- É A MELHOR ESCOLHA PARA PRODUCT-BASED SOFTWARE, ONDE HÁ PREVISÃO DE LANÇAMENTO DE FUNCIONALIDADES ADICIONAIS MAIS TARDE, COM SELEÇÃO DE USERS

Agilidade Facilidade de responder a mudanças constantes no ambiente	Alta	Devido à natureza modular das soluções desenvolvidas com este padrão, as mudanças são isoladas e rapidamente implementáveis. Devido à sua simplicidade, o *core system* tem tendência a ter um desenvolvimento estabilizado rapidamente, com alterações pontuais ao longo do tempo.
Facilidade de *deploy*	Alta	Novamente a natureza desacoplada dos componentes permite um *deploy* simples e em tempo de execução.
Facilidade de testes	Alta	Os testes podem ser isolados.
Performance	Alta	A modularidade permite a criação de aplicações com as funcionalidades estritamente necessárias, dispensando a implementação de código desnecessário.
Escalabilidade	Baixa	Geralmente as implementações são pequenas, implementáveis em unidades individuais e por isso pouco escaláveis.
Facilidade de desenvolvimento	Baixa	O desenvolvimento destas aplicações implica um planeamento e desenho cuidado dos contratos, nomeadamente quanto ao registo dos *plug-ins*, à sua granularidade e diversas possibilidades de comunicação interna.

MICROSERVICES ARCHITECTURE PATTERN

- SIMPLIFICA O DESENVOLVIMENTO ATRAVÉS DA DIVISÃO EM PEQUENOS SERVIÇOS (DEPLOYED COMO SEPARATE UNITS, PERMITINDO EASIER DEPLOYMENT E DECOUPLING)
- COMUNICAM POR JMS, AMQQ, REST, ETC

- ALTERNATIVA A APLICAÇÕES MONOLÍTICAS (QUE RECORREM À LAYERED ARCHITECTURE) OU A ORIENTADAS A SERVIÇOS.

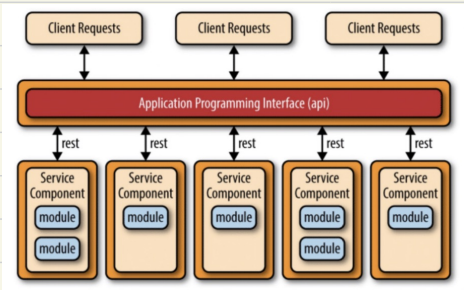


MAIN CHARACTERISTICS

- FACILMENTE MANTIDO E TESTADO
- LOOSELY COUPLED COM OUTROS SERVIÇOS
- DEPLOYED INDEPENDENTEMENTE
- PODE SER DESENVOLVIDO POR UMA EQUIPA PEQUENA

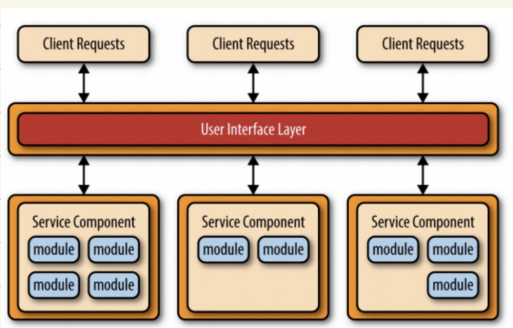
API REST-BASED TOPOLOGY

- DISPONIBILIZAR UMA API PARA ACEDER UM NÚMERO REDUZIDO DE SERVIÇOS INDIVIDUAIS E INDEPENDENTES



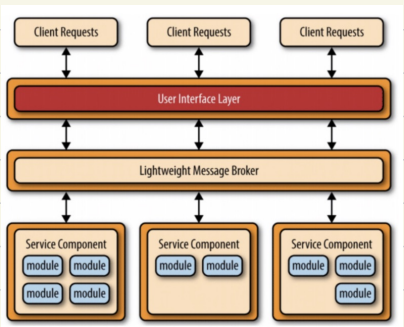
APLICATION REST-BASED TOPOLOGY

- DISPONIBILIZAR UMA INTERFACE (DEPLOYED SEPARATELY)



CENTRALIZED MESSAGING TOPOLOGY

- DISPONIBILIZA UM MESSAGE BROKER PARA ROTEADOR



- APLICAÇÕES MAIS ROBUSTAS, COM MELHOR SCALABILITY E SUPORTE PARA ENTREGA CONTINUA
- CAPAZ DE DAR REAL-TIME PRODUCTION DEPLOYMENTS
- SENDO UMA DISTRIBUTED ARCHITECTURE, TEM OS MESMOS PROBLEMAS QUE O EVENT-DRIVEN ARCHITECTURE

Agilidade Facilidade de responder a mudanças constantes no ambiente	Alta	Uma vez que os componentes de processamento têm um único propósito e estão desacoplados, as modificações são isoladas a um número restrito de processadores.
Facilidade de *deploy*	Alta	Novamente a natureza desacoplada dos componentes permite um *deploy* simples.
Facilidade de testes	Alta	Testes dos componentes individuais são muito mais simples que numa aplicação monolítica, sendo ainda excluida a possibilidade de uma alteração num componente inteferir com o funcionamento de outro.
Performance	Baixa	Devido à sua natureza distribuída as aplicações desenvolvidas segundo este padrão tendem a ter uma performance mais reduzida quando comparadas com outras.
Escalabilidade	Alta	Novamente o desacoplamento e a independência dos componentes permitem uma escalabilidade alta e específica para certos componentes.
Facilidade de desenvolvimento	Alta	Devido à funcionalidade estar isolada em cada componente, o desenvolvimento é focado em unidades funcionais e por isso mais simples.

SPACE-BASED ARCHITECTURE

- TAMBÉM CHAMADO DE CLOUD-BASED ARCHITECTURE

- DESENHADO PARA RESOLVER OS PROBLEMAS DE SCALABILITY E CONCURRENCY DE WEB-BASED SERVICES (EXISTE BOTTLENECK QUANDO A USER LOAD AUMENTA)

- APROVEITA DE DISTRIBUTED SHARED MEMORY
- HIGH SCALABILITY -> AO SUBSTITUIR A DATABASE CENTRAL POR IN-MEMORY DATA GRIDS REPLICADAS (ONDE SE GUARDA APPLICATION DATA)

    -> UNIDADES DE PROCESSAMENTO SÃOS INICIADAS E DESLIGADAS DINAMICAMENTE COM A USER LOAD

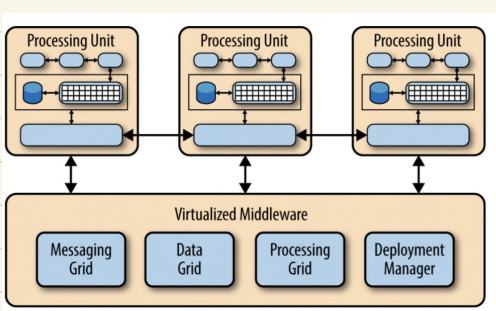
COMPOSTO POR:

PROCESSING-UNIT COMPONENT

- CONTÉM OS COMPONENTES DA APLICAÇÃO (WEB-BASED, BACKEND BUSINESS LOGIC)

VIRTUALIZED-MIDDLEWARE

- FUNCIONA COMO UM CONTROLADOR, GERINDO AS COMUNICAÇÕES, PROCESSAMENTO DE PEDIDOS,...



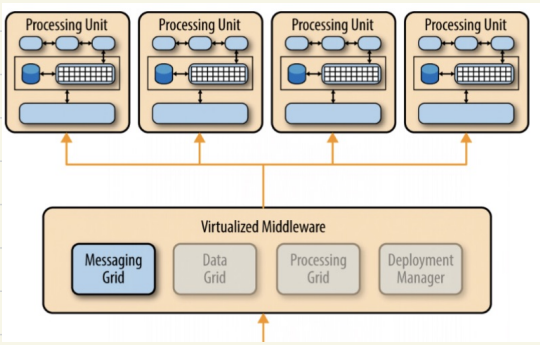
DEEPER DIVE EM VIRTUALIZED-MIDDLEWARE

- GERE MESSAGE REQUESTS, SESSIONS, DATA REPLICATION, DISTRIBUTED REQUEST PROCESSING E PROCESS-UNIT DEPLOYMENT

QUATRO COMPONENTES PRINCIPAIS:

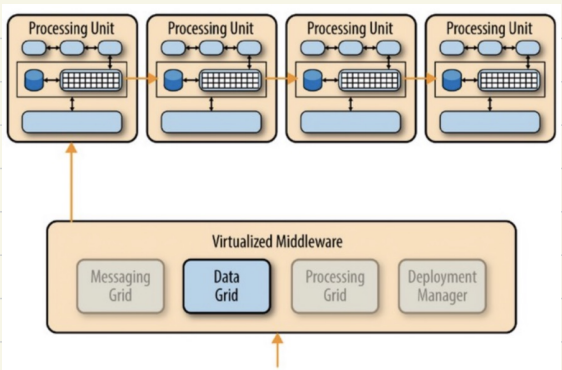
MESSAGING GRID

- ENCAMINHA OS PEDIDOS PARA A UNIDADE DE PROCESSAMENTO QUE A PODE RECEBER NO MOMENTO



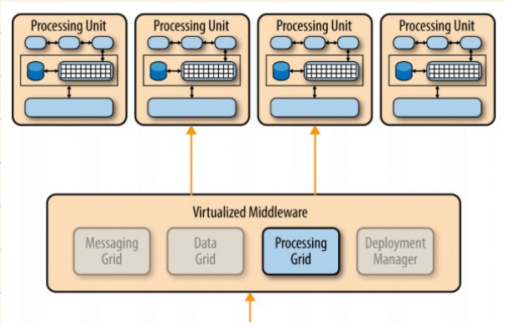
DATA GRID

- GERE A REPLICAÇÃO DE DATA ENTRE UNIDADES DE PROCESSAMENTO QUANDO HÁ UPDATES



PROCESSING GRID (OPCIONAL)

- GERE O PROCESSAMENTO DISTRIBUÍDO ENTRE AS UNIDADE, SERVINDO DE MEDIATOR



DEPLOYMENT MANAGER

- GERE O SET UP DINÂMICO DAS UNIDADES DE PROCESSAMENTO (MONITORIZA TEMPOS DE RESPOSTA E CARGA DE DADOS)

- É UM PADRÃO COMPLEXO QUE REQUERE MUITOS RECURSOS,
- BOA PARA PEQUENAS APLICAÇÕES COM VARIAÇÕES DE FLUXO
- MÁ PARA APLICAÇÕES GRANDES COM FLUXO DE DADOS CONSTANTEMENTE GRANDE

Agilidade Facilidade de responder a mudanças constantes no ambiente	Alta	Devido ao *deplyment manager*, as aplicações conseguem responder muito bem a alterações no fluxo de acessos. Também constumam responder bem a alterações no código devido ao tamanho reduzido das aplicações e à sua natureza dinâmica.
Facilidade de *deploy*	Alta	Apesar de a sua natureza não ser desacoplada, o seu dinamismo e as ferramentas dos sistemas *cloud-based* permitem fazer um *deploy* relativamente simples.
Facilidade de testes	Baixa	Atingir um fluxo de acessos elevado em ambiente de testes é bastante dispendioso e *time consuming*.
Performance	Alta	Devido ao acesso *in-memory* e a mecanismos de *caching* este padrão consegue oferecer soluções de alta performance.
Escalabilidade	Alta	Devido à inexistência de dependências a uma base de dados central, este padrão é facilmente escalável.
Facilidade de desenvolvimento	Baixa	A memória partilhada replicada em todas as unidades de processamento e os mecanismos de *caching* são funcionalidades complexas de desenvolver.

RESUMINDO

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

CHAPT 8

JAVA REFLECTION

JAVA REFLECTION

PERMITE QUE UM PROGRAMA SE EXAMINE A SI MESMO

CONSEGUE:

- DETERMINAR A CLASSE DE UM OBJETO
- OBTER INFORMAÇÃO SOBRE UMA CLASSE, COMO SUPERCLASSES, FIELDS, CONSTRUCTORS
- OBTER INFORMAÇÃO SOBRE UMA INTERFACE
- CRIAR E MANIPULAR VECTORES DE OBJETOS

SEM SABER O NOME DA CLASSE OU MÉTODOS CONSEGUE:

- CRIAR UMA INSTÂNCIA DE UMA CLASSE
- LER/MODIFICAR VARIÁVEIS
- INVOCAR MÉTODOS

CONSEGUE OBTER A CLASSE POR DUAS MANEIRAS:

```
Class<?> cl1 = Class.forName("java.util.Properties");
ou
Object obj = ... // e.g. new StringBuffer("Teste");
Class<?> cl2 = obj.getClass();
```

OBJ.GETCLASS().GETNAME() PARA O NOME

ENCODING SCHEME UTILIZADO POR CLASS.FORNAME()

B → byte; C → char; D → double; F → float; I → int; J → long;
Lclass-name → class-name[]; S → short; Z → boolean
Use as many "["s as there are dimensions in the array

CLASS

```
public final class Class<T>
extends Object
implements Serializable, GenericDeclaration,
   Type, AnnotatedElement
```

```
static Class<?> forName(String className);
T newInstance();
Field[] getFields();
Method[] getMethods();
boolean isInstance(Object obj);
String getName();

getInterfaces(), getSuperclass(),
getModifiers(), getField(), getMethod(),...
```

```
void printClassName(Object obj) {
    System.out.println("The class of " + obj +
                       " is " + obj.getClass().getName());
}
```

FIELD

```
public final class Field
extends AccessibleObject
implements Member
```

```
Object get(Object obj);
void   set(Object obj, Object val);

getType(), getDeclaringClass(),

setDouble(…), setInt(…), .....
```

```
Field[] flds = someObject.getClass().getFields();
for (Field f: flds)
       System.out.println(f.getName());
```

METHOD

```
public final class Method
extends AccessibleObject
implements GenericDeclaration, Member
```

```
Object invoke(Object obj, Object... args);

Class<?> getReturnType();

Class<?>[] getParameterTypes(),

getExceptionTypes(), getDeclaringClass(),..
```

```
Method methods[] = someClass.getMethods();
for (Method m: methods)
       System.out.println(m);
```

MÉTODOS

PUBLIC STATIC CLASS<?> FORNAME (STRING CLASSNAME)

- RETORNA UM OBJ CLASSE QUE REPRESENTA A CLASSE DADA

PUBLIC STRING GETNAME()

- RETORNA O NOME DO OBJETO (EX. JAVA.LANG.STRING)

PUBLIC INT GETMODIFIERS()

- RETORNA UM INT QUE REPRESENTA: PUBLIC, FINAL OU ABSTRACT

PUBLIC T NEWINSTANCE()

- RETORNA A INSTANCIA DA CLASSE AT RUNTIME

EXEMPLO NEW INSTANCE

```
public class ReflectionNew {
   public static void main(String[] args) throws Exception {
      Class<?> sc = Class.forName("aula5_1.Circulo");

      System.out.println("Name = " + sc.getName());
      System.out.println("SimpleName = " + sc.getSimpleName());

      Class<?>[] paramTypes = { Double.TYPE, Double.TYPE, Double.TYPE };
      Constructor<?> cons = sc.getConstructor(paramTypes);
      Object ar[] = { 2, 4, 10 };
      Object theObject = cons.newInstance(ar);
      System.out.println("New object: " + theObject);

      Constructor<?> cs = sc.getConstructor(new Class<?>[]{Double.TYPE});
      System.out.println("New object: " + cs.newInstance(new Object[]{20}));
   }
}
```

```
Name = aula5_1.Circulo
SimpleName = Circulo
New object: Circulo de Centro (2.0,4.0) e de raio 10.0
New object: Circulo de Centro (0.0,0.0) e de raio 20.0
```

MÉTODOS (2)

PUBLIC CLASS[] GETCLASSES()

- RETORNA UM ARRAY DE TODAS AS INNER CLASSES DA CLASS

PUBLIC CONSTRUCTOR GETCONSTRUCTOR(CLASS[] PARAMS)

- RETORNA TODOS OS PUBLIC CONSTRUCTORS DA CLASS QUE TÊM OS PARÂMETROS ESPECIFICADOS

PUBLIC CONSTRUCTOR[] GETCONSTRUCTORS()

- RETORNA TODOS OS PUBLIC CONSTRUCTORS DA CLASS

EXEMPLO

```
public class Reflection2 {
   public static void main(String[] args) throws InstantiationException,
                        IllegalAccessException {
         String s="Mar";

         Class<?> sc = s.getClass();
         System.out.println("******** Construtores ***********");
         Constructor<?> contrs[] = sc.getConstructors();
         for (Constructor<?> c: contrs)
                  System.out.println(c);
   }
}
```

```
******** Construtores **********
public java.lang.String()
public java.lang.String(java.lang.String)
public java.lang.String(char[])
public java.lang.String(char[],int,int)
public java.lang.String(int[],int,int)
public java.lang.String(byte[],int,int,int)
public java.lang.String(byte[],int)
...
```

MÉTODOS (3)

(NOT SURE IF RIGHT)

PUBLIC FIELD GETFIELD(STRING NAME)

- RETORNA UM OBJETO DA CLASS FIELD QUE CORRESPONDE À INSTÂNCIA DA CLASS ("NAME")

PUBLIC FIELD[] GETFIELDS()

- RETORNA TODAS AS VARIÁVEIS PÚBLICAS DA CLASSE

PUBLIC FIELD[] GETDECLAREDFIELDS()

- RETORNA TODAS AS VARIÁVEIS DECLARADAS DA CLASSE

EXEMPLOS

```
public static void main(String[] args) throws Exception {
    Class<?> sc = Class.forName("aula5_1.Circulo");
    System.out.println("\n******** Fields ******************\n");
    Field fields[] = sc.getFields();
    for (Field f: fields)
        System.out.println(f);
    System.out.println("\n******** Declared Fields **************\n");
    Field dfields[] = sc.getDeclaredFields();
    for (Field f: dfields)
        System.out.println(f);
    System.out.println("\n******** raio Field  ******************\n");
    Field field = sc.getField("raio");  // deve usar-se getDeclaredField
    System.out.println(field);
}
```

```
******** Fields ******************

******** Declared Fields *************
private double aula5_1.Circulo.raio
******** raio Field  ******************
    Exception in thread "main" java.lang.NoSuchFieldException:
    aula5_1.Circulo.raio
    at java.lang.Class.getField(Class.java:1520)
    at reflection.Reflection2.main(Reflection2.java:39)
```

```java
class SampleGet {
   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 325);
      printHeight(r);
   }
   static void printHeight(Object r) {
      Field heightField; // declares a field
      Integer heightValue;
      Class<?> c = r.getClass(); // get the Class object
      try {
         heightField = c.getField("height"); // get the field object
         heightValue = (Integer) heightField.get(r); // get the value
         System.out.println("Height: " + heightValue.toString());
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

```
Height: 325
```

```java
class SampleSet {
   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
      System.out.println("original: " + r.toString());
      modifyWidth(r, 300);
      System.out.println("modified: " + r.toString());
   }
   public static void modifyWidth(Object r, Integer widthParam ) {
      Field widthField; // declare a field
      Integer widthValue;
      Class<?> c = r.getClass(); // get the Class object
      try {
          widthField = c.getField("width"); //get the field object
          widthField.set(r, widthParam); //set the field to widthParam =300
      } catch (Exception e ) {
          // . . .
      }
   }
}
```

```
original: java.awt.Rectangle[x=0,y=0,width=100,height=20]
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]
```

MÉTODOS (4)

PUBLIC METHOD GETMETHOD(STRING NAME, CLASS[] PARAMS)

- RETORNA UM OBJETO METHOD QUE CORRESPONDE O MÉTODO "NOME", COM CERTOS PARÂMETROS

PUBLIC METHOD[] GETMETHODS()

- RETORNA TODOS OS MÉTODOS PÚBLICOS DA CLASSE

PUBLIC METHOD[] GETDECLAREDMETHODS()

- RETORNA OS MÉTODOS DECLARADOS DA CLASSE

PUBLIC PACKAGE GETPACKAGE()

- RETORNA A PACKAGE QUE CONTÉM A CLASSE

PUBLIC CLASSE GETSUPERCLASSE()

- RETORNA A SUPERCLASSE DA CLASSE

EXEMPLOS:

```
public class Reflection2 {
    public static void main(String[] args) throws InstantiationException,
                                                   IllegalAccessException {
          String s="Mar";
          Class<?> sc = s.getClass();
          System.out.println("******** Métodos  ***************\n");
          Method methods[] = sc.getMethods();
          for (Method m: methods)
                  System.out.println(m);
    }
}
```

```
******** Métodos    ***************

public boolean java.lang.String.equals(java.lang.Object)
public java.lang.String java.lang.String.toString()
public int java.lang.String.hashCode()

…..
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
```

```java
public static void main(final String[] args) {
   try {
      String[] z = new String[] { "Jim", "John", "Joe" };
      final Class<?> type = z.getClass();
      if (!type.isArray()) {
         throw new IllegalArgumentException();
      } else {
         System.out.println("Name = "+ type.getName() +
               "\nType = "+type.getComponentType());
      }
   } catch (final Exception ex) {
      ex.printStackTrace();
   }
}
```

```
Name = [Ljava.lang.String;
Type = class java.lang.String
```

```java
public class ArrayNew {
   public static void main(String[] args) throws ClassNotFoundException {
         System.out.println(createNativeArray("int", 12).getClass());
         System.out.println(createNativeArray("boolean", 10, 10).getClass());
         System.out.println(createNativeArray("double", 5, 5, 5).getClass());
   }
   public static Object createNativeArray(String typeName, int... dim)
   throws ClassNotFoundException {
         Class<?> clazz = null;
         if ("int".equals(typeName)) {
                  clazz = Integer.TYPE;
         } else if ("boolean".equals(typeName)) {
                  clazz = Boolean.TYPE;
         } else if ("double".equals(typeName)) {
                  clazz = Double.class;
                  // All other native types: short, long, float ......
         } else {
                  throw new ClassNotFoundException(typeName);
         }
         return Array.newInstance(clazz, dim);
   }
}
```

```
class [I
class [[Z
class [[[Ljava.lang.Double;
```

PLUG-INS?

IPlugin.java

```java
public interface IPlugin {
    public void metodo();
}
```

Plugin1.java

```java
public class Plugin1 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin1: metodo invocado");
    }
}
```

Plugin2.java

```java
public class Plugin2 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin2: metodo invocado");
    }
}
```

Plugin3.java

```java
public class Plugin3 implements IPlugin {
    public void metodo() {
          System.out.println("Plugin3: metodo invocado");
    }
}
```

Plugin.java

```java
package reflection;
import java.io.File;

abstract class PluginManager {
    public static IPlugin load(String name) throws Exception {
      Class<?> c = Class.forName(name);
      return (IPlugin) c.newInstance();
    }
}

public class Plugin {
    public static void main(String[] args) throws Exception {
      File proxyList = new File("reflection/plugins");
      for (String f: proxyList.list()) {
         try {
            IPlugin obj =
                 PluginManager.load("reflection."+f.substring(0,f.lastIndexOf('.')));
            obj.metodo();
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
    }
```

```
Plugin1: metodo invocado
Plugin2: metodo invocado
Plugin3: metodo invocado
```