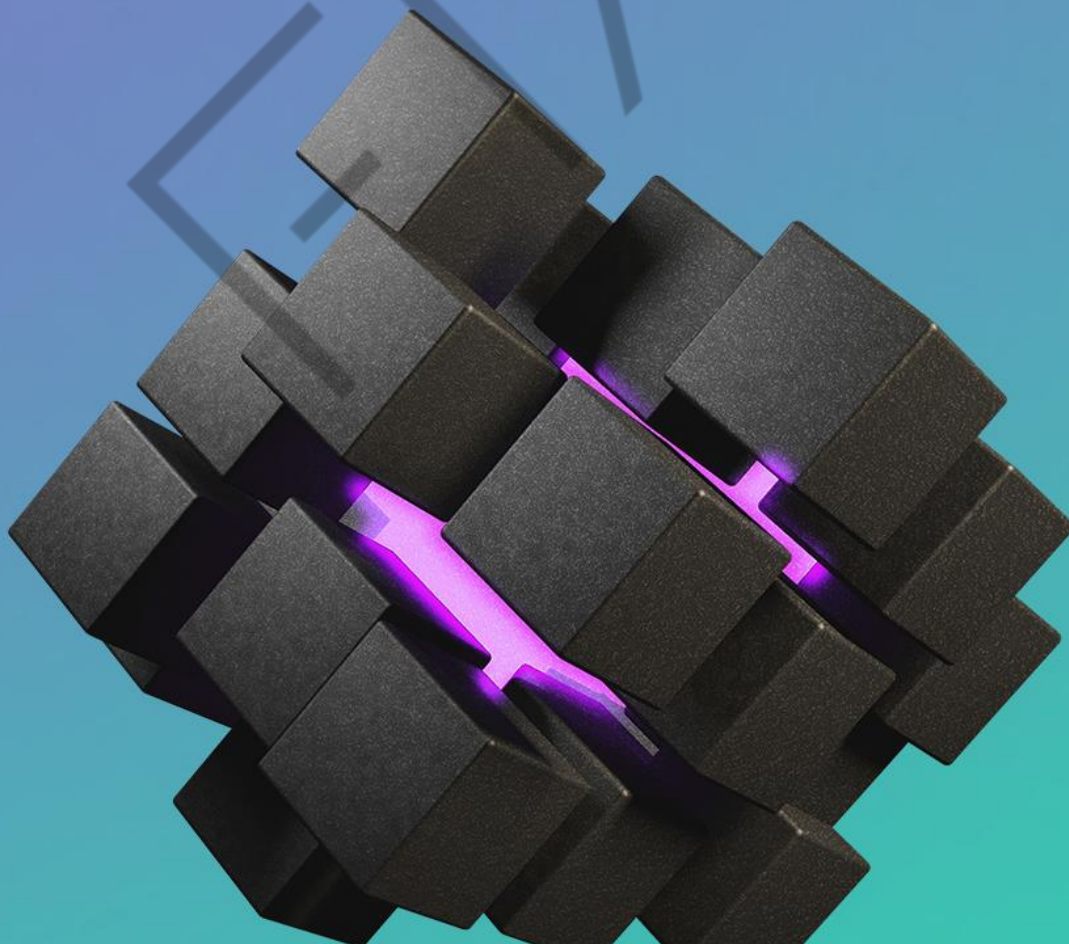


INSPIRING: ORM & JAVA 2.0

OS PADRÕES DE
DESENVOLVIMENTO
MAIS PRESENTES
DO QUE NUNCA



3

LISTA DE FIGURAS

Figura 1 – Construção do software	8
Figura 2 – Overview of major components of MVC	12
Figura 3 – Arquitetura JSP Modelo 2	13
Figura 4 – SAPUI5 is a client UI technology	15
Figura 5 – UML Factory Method	16
Figura 6 – UML Abstract Factory	16
Figura 7 – UML Builder	17
Figura 8 – UML Prototype	18
Figura 9 – UML Singleton	19
Figura 10 – UML Adapter	20
Figura 11 – UML Bridge	21
Figura 12 – UML Composite	22
Figura 13 – UML Decorator	23
Figura 14 – UML Facade	24
Figura 15 – UML Flyweight	25
Figura 16 – UML Proxy	25
Figura 17 – UML Chain of Responsibility	27
Figura 18 – UML Command	28
Figura 19 – UML Interpreter	29
Figura 20 – UML Iterator	30
Figura 21 – UML Mediator	30
Figura 22 – UML Memento	32
Figura 23 – UML Observer	32
Figura 24 – UML State	33
Figura 25 – UML Strategy	34
Figura 26 – UML Template Method	34
Figura 27 – UML Visitor	35
Figura 28 – Página de download, Logging Services Log4j	37
Figura 29 – Usando Log4j	37
Figura 30 – Adicionando as classes ao Build Path	38
Figura 31 – Criando o arquivo log4j2.properties	38
Figura 32 – Pasta e arquivo de log criado pelo Log4j	40
Figura 33 – Conteúdo do log criado pelo Log4j	41
Figura 34 – Trecho do código que grava o log no Eclipse	42
Figura 35 – Diagrama do Exemplo Java – MVC e <i>Design Patterns</i>	49
Figura 36 – Tela de criação do Java Project PISMVC	51
Figura 37 – Tela de criação da interface Imposto na camada <i>Model</i>	51
Figura 38 – Tela de criação da Classe Pis da camada <i>Model</i>	52
Figura 39 – setChanged Method	54
Figura 40 – Tela de criação da interface TelaDeImposto na camada <i>View</i>	54
Figura 41 – Tela de criação da Classe CalculaPis da camada <i>View</i>	55
Figura 42 – Tela de criação da Classe ImpostoController, da camada <i>Controller</i>	58
Figura 43 – Tela de criação da Classe TestaCalculo, da camada <i>View</i>	59
Figura 44 – Tela de cálculo do imposto e msg com o resultado	61
Figura 45 – Fluxo resumido da aplicação	61
Figura 46 – Estrutura atual do Projeto	62
Figura 47 – Estrutura com Singleton	62

Figura 48 – Propriedades alíquota Pis e Confins, definidas em conf.properties no Eclipse	63
Figura 49 – Atributo ALIQUOTA da Classe Pis, recebendo o valor do Singleton.....	63



LISTA DE QUADROS

Quadro 1 – Parametrização <code>append.layout.pattern</code>	41
--	----

EMSE

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Log4j2.properties parametriza o uso do Log4j.....	40
Código-fonte 2 – Treco do arquivo log4j2.properties que parametriza o uso do Log4j	41
Código-fonte 3 – IndexController, instanciando a classe de Log	42
Código-fonte 4 – IndexController, uso do logger.	43
Código-fonte 5 – LoginBusiness, implementação das regras de negócio.....	45
Código-fonte 6 – LoginFacade, abstração das regras de login	46
Código-fonte 7 – LoginController, refatoração para o uso da classe Facade	48
Código-fonte 8 – Interface Imposto.....	51
Código-fonte 9 – Classe Pis, subclasse de <i>Observable</i>	53
Código-fonte 10 – Interface TelaDelImposto	54
Código-fonte 11 – Camada <i>View</i> , Classe CalculaPis implementa a interface <i>Observer</i>	57
Código-fonte 12 – Camada <i>Controller</i> , classe ImpostoController implementa a interface <i>ActionListener</i>	59
Código-fonte 13 – Camada <i>Controller</i> , Classe ImpostoController	60
Código-fonte 14 – Classe AliquotaSingleton implementa o padrão Singleton	63

SUMÁRIO

1 OS PADRÕES DE DESENVOLVIMENTO MAIS PRESENTES DO QUE NUNCA	7
1.1 O que é um <i>Design Pattern</i> ?	7
1.2 Padrão MVC (<i>Model, View, Controller</i>)	10
1.3 GoF Design Patterns	15
1.3.1 Padrões de Criação	15
1.3.2 Padrões Estruturais	19
1.3.3 Padrões Comportamentais	26
2 JAVA LOGGING FRAMEWORK	36
2.1 Facade Pattern hands-on	44
2.2 MVC e os <i>Design Patterns</i>	48
2.3 Frameworks	64
CONSIDERAÇÕES FINAIS	68
REFERÊNCIAS	70
GLOSSÁRIO	72

1 OS PADRÕES DE DESENVOLVIMENTO MAIS PRESENTES DO QUE NUNCA

Nosso aprendizado em Java até o final do ano passado abordou o básico da linguagem Java e, posteriormente, sua integração com o HTML, por meio dos *servlets*, JSP e JSTL. Tivemos a oportunidade de acessar o banco de dados, tomando contato com alguns *Design Patterns* importantes, como o MVC, Singleton, DAO e Factory.

Vamos retomar os estudos exatamente de onde pararam! Neste capítulo, falaremos mais sobre arquitetura e desses *Design Patterns* novamente, além de ter contato com outros que podem ser importantes em seus desenvolvimentos.

Acessar o banco de dados Oracle para o Health Track foi um tremendo sucesso, mas para projetos muito grandes com muitas tabelas e entidades de sistema, você deve ter reparado que seria pouco prático e muito trabalhoso. Vamos abordar nos capítulos seguintes formas diferentes de se acessar o banco de dados.

1.1 O que é um *Design Pattern*?

Vimos os *Design Patterns* de DAO e Factory no ano passado, mas você sabia que existem vários outros *Design Patterns*? Vamos aprender, neste capítulo, como desenvolver códigos-fonte eficientes e dentro das melhores práticas de mercado.

Explicando com mais detalhes o exemplo que utilizamos no resumo deste capítulo, a nossa empresa imaginária do ramo de acessórios femininos se deparou com uma incerteza, em um novo projeto para atender a uma necessidade de suas clientes. Para essa empresa conceituada e de qualidade reconhecida, seria um grande risco errar e ter de lançar várias versões da solução até aprender a melhor forma de fazer. A fim de minimizar tal risco, ela resolveu utilizar uma solução já implementada largamente, segura e amplamente testada pelo mercado. Ou seja, ela utilizou um padrão.



Figura 1 – Construção do software
Fonte: Banco de Imagens Shutterstock (2018)

Esse tipo de situação não é diferente para nós, quando estamos construindo um *software*. Imagine sua equipe no momento do projeto no qual se definirá a separação de camadas de um novo sistema Web, que vocês estão implementando. Tal qual proteger a cliente da chuva, utilizando um padrão que detalha claramente como fazer um guarda-chuva, será que nenhuma outra equipe de projeto de *software* passou pela mesma necessidade de projetar a separação de camadas e as classes que compõem cada camada de um sistema Web?

Como esse é um problema recorrente em projetos de *software*, não seria bom ter uma forma de aproveitarmos a experiência que eles acumularam e absorver rapidamente os conhecimentos que eles consolidaram, sem passar pelas dificuldades e erros que eles cometeram até chegar a uma boa solução?

Uma boa notícia: essa forma existe! No exemplo acima, podemos imaginar que a equipe de *software* tenha escolhido utilizar a arquitetura MVC (*Model*, *View* e *Controller*) para resolver um problema arquitetural (separação de camadas) com o emprego de *Design Patterns* em cada camada do MVC, quando necessário.

Bem, entendemos que MVC é um padrão de arquitetura que orientou a equipe a separar a aplicação Web em três camadas, mas o que é um *Design Pattern* e quando ele será empregado? Podemos dizer que um *Design Pattern* também comumente chamado de padrão de projeto é uma solução consolidada e

reconhecida pela comunidade de desenvolvedores, como uma forma eficiente de se resolver um problema recorrente em projetos e manutenções de *software*.

Ainda utilizando o projeto Web fictício como exemplo, seguindo o padrão arquitetural MVC, classes da camada *Model* do sistema serão responsáveis por lidar com as regras de negócio da aplicação e, dentre outras atividades, gerenciar e prover conexões com o banco de dados.

Mas como a equipe deve fazer isso? Já que este é um problema técnico recorrente, que inúmeros projetos de *software* já resolveram, será que existe um padrão de projeto ou o uso de padrões de projeto em conjunto que ajudem a equipe a desenvolver uma boa solução? Existe, mas, para escolhermos bem, precisamos entender e conhecer melhor os padrões.

Como percebemos, padrões tornam mais fácil o desenvolvimento de *softwares*, reutilizando soluções bem-sucedidas para construir e manter de forma flexível e estável *softwares* orientados a objetos.

Buscando a origem, o conceito de padrão de projeto de *software* foi criado na década de 1970, para resolver problemas de Engenharia, por Christopher Alexander em seus livros: *Notes on the Synthesis of Form*, *The Timeless Way of Building* e *A Pattern Language*. Neles, Alexander estabelece que um padrão deve ter:

- **Encapsulamento:** um padrão encapsula um problema ou uma solução bem definida.
- **Generalidade:** todo padrão deve permitir a construção de outras realizações a partir desse padrão.
- **Equilíbrio:** quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto.
- **Abstração:** os padrões representam abstrações da experiência empírica ou do conhecimento cotidiano.
- **Abertura:** um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.
- **Combinatoriedade:** os padrões são relacionados hierarquicamente.

Existem diversas referências e publicações para padrões de *software*, tais como: *GoF Patterns*, *Architectural Patterns*, *GRASP Patterns*, *Concurrency Patterns*, *JEE Patterns*, entre outras.

A referência mais conhecida relacionada a padrões de projeto de *software* é o livro *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995), dos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Esses quatro autores são conhecidos como “The Gang of Four” (GoF).

Alguns programadores, analistas e arquitetos experientes perceberam que os problemas começavam a se repetir várias vezes e que as soluções eram semelhantes. Sendo assim, passaram a documentá-las, principalmente em projetos de C++ e SmallTalk.

O livro descreve 23 padrões de *design*, obtidos dessas experiências documentadas que comentamos, e os organiza em três tipos de padrões:

- **De criação:** relativos à criação de objetos.
- **Estruturais:** tratam das associações entre classes e objetos.
- **Comportamentais:** referentes às interações (algoritmos) e divisões de responsabilidades entre as classes ou objetos.

Neste capítulo, veremos exemplos de padrões arquiteturais e padrões GoF, começando pelo padrão arquitetural MVC.

1.2 Padrão MVC (*Model, View, Controller*)

No começo do desenvolvimento de *software*, uma solução era codificada para ser executada em uma única máquina, ou seja, o *software* normalmente empregava uma arquitetura formada por apenas uma camada e sem modularidade. Tal camada era responsável pela interface com o usuário, pela execução da regra de negócio e, finalmente, pela persistência das informações. Tudo isso programado em um único código-fonte. Geralmente, aplicações desse tipo possuem linhas e linhas intermináveis de código, muitas vezes, com inúmeros trechos repetidos.

Essas aplicações, também conhecidas como aplicações monolíticas, inviabilizam a reutilização de funcionalidades sem a cópia dos trechos de código e dificultam muito as manutenções corretivas e evolutivas que se façam necessárias.

Para melhorar esse cenário, na década de 1970, surgiu uma arquitetura desenvolvida por Trygve Reenskaug, no centro de pesquisa da Xerox, para projetos de interface visual na linguagem de programação *Smalltalk*. Essa arquitetura recebeu o nome de MVC (*Model, View, Controller*).

Com a evolução da tecnologia, surgiram outras abordagens, como o desenvolvimento de *software desktop* utilizando duas camadas, essa arquitetura ficou conhecida como *Client Server*. Uma solução desse tipo tem o *software* cliente instalado na máquina dos usuários e um servidor central, com o banco de dados. Apesar de ser uma evolução, a interface com o usuário, a regra de negócios e o acesso aos dados ainda estavam em uma aplicação que nem sempre tinha tais responsabilidades bem separadas.

O MVC foi reaproveitado e se fortaleceu com a evolução da plataforma Web. Apesar de criado para resolver questões da década de 1970 em *Smalltalk*, o padrão de arquitetura *Model, View, Controller* foi e ainda é utilizado como padrão de arquitetura que soluciona problemas de separação de responsabilidades na programação para Internet.

No início do desenvolvimento Web, também encontrávamos problemas desse tipo. Se puxarmos pela memória ou realizarmos uma pesquisa, veremos que nos primórdios do desenvolvimento Web, o HTML normalmente era gerado por um programa que também executava regras de negócio e acessava os dados.

O MVC tem como objetivo manter dados e regras de negócios na camada *Model*, separadamente da camada *View*, que conterà as interfaces com o usuário, e da camada de *Controller*, que será responsável pelo fluxo da aplicação.

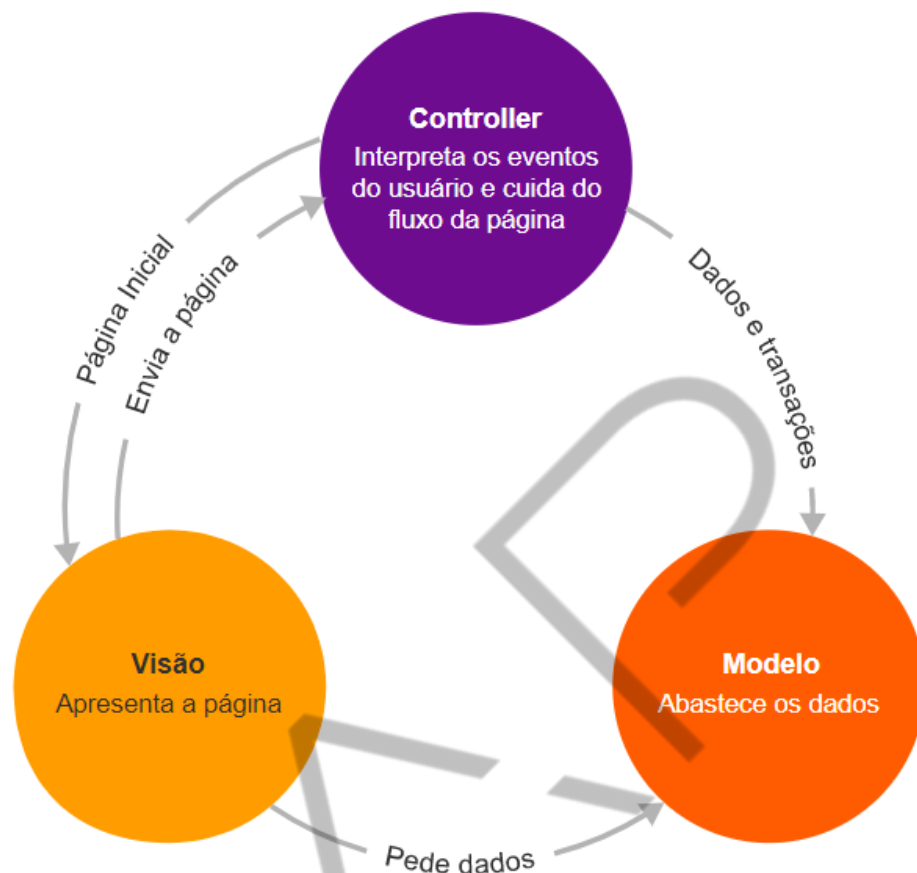


Figura 2 – Overview of major components of MVC
Fonte: Oracle (2017), adaptado por FIAP (2018)

- **Model:**
 - Armazena o estado do aplicativo.
 - Responde aos pedidos de dados.
 - Encapsula a lógica de negócios.
- **View:**
 - Renderiza a interface do usuário (UI).
 - Solicita dados à camada *Model* para a camada *Controller*.
 - Envia eventos à camada *Model* através da camada *Controller*.
 - Permite que o *Controller* selecione a próxima *View*.
- **Controller:**
 - Realiza o roteamento para a página correta.
 - Mapeia as mudanças de dados da UI para o Modelo.

- Passa dados da camada View para a camada *Model*.
- Essa arquitetura serviu como uma luva para o desenvolvimento Web.

Segundo a Oracle, quando olhamos para aplicações centradas em banco de dados no geral ou para aplicativos *Web-based thin-client* em particular, podemos ver que essas aplicações devem realizar várias tarefas distintas, tais como interagir com os usuários, acessar dados, executar regras de negócio, apresentar dados e controlar fluxos.

A arquitetura MVC fornece uma maneira de compartimentalizar essas tarefas, trabalhando a partir da premissa de que atividades, como apresentação de dados, devem ser separadas do acesso a dados. Dessa forma, podemos facilmente fazer alterações na fonte de dados, sem ter que reescrever a UI.

A Figura “Arquitetura JSP Modelo 2” ilustra a arquitetura MVC sendo utilizada pela Oracle no JSP *Model 2 Architecture*. Essa arquitetura Oracle foi implementada em Java, tendo os Servlets como responsáveis por controlar o fluxo da aplicação Web (*Controller*), delegando a lógica de negócios para componentes, geralmente *JavaBeans* ou EJBs (*Model*). Enquanto as páginas JSP geram o HTML, interface com usuários, através de navegadores Web (*View*).

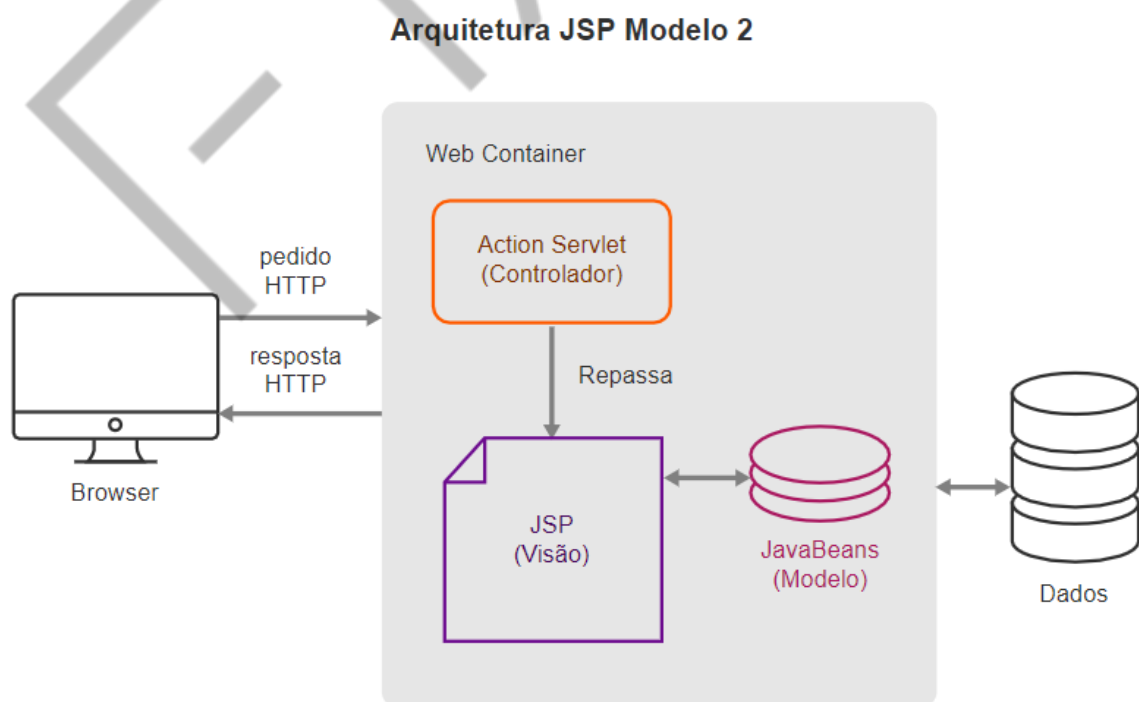


Figura 3 – Arquitetura JSP Modelo 2
Fonte: Oracle (2017), adaptado por FIAP (2018)

Vamos entender um pouco o fluxo do MVC. Quando o usuário manipula uma *View*, um evento é gerado. Os eventos, geralmente, fazem com que um *Controller* mude um *Model*, ou uma *View*, ou ambos. Sempre que um *Controller* alterar as propriedades ou dados de um *Model*, todas as *Views* dependentes serão automaticamente atualizadas. De forma semelhante, sempre que um *Controller* muda uma *View*, dados do *Model* subjacente são obtidos pela *View*, para se atualizar.

Mas o que o MVC traz de benefícios reais para essa arquitetura Java?

O MVC estruturou os componentes do *Model 2* de acordo com as respectivas responsabilidades em tempo de execução. Dessa forma, a Oracle obteve os seguintes benefícios:

- Redução da complexidade.
- Eliminação do acoplamento entre as camadas Model e View.
- Elementos mais gerenciáveis e flexíveis que compõem a arquitetura.
- Fácil adição de novas fontes de dados, sem impacto na View.
- Adição fácil de novos tipos de clientes, sem impacto em Model.
- Aumento da reutilização de componentes.
- Uma forma de associar habilidades específicas de desenvolvedores com componentes dedicados. Ou seja, por exemplo, programadores front se especializam e só codificam a camada View.
- Rastreamento mais fácil de erros.

Essa abordagem arquitetural facilita que a implementação de uma regra de negócios possa ser acessada e atenda a diversas solicitações de usuários, vindas de diferentes tipos de dispositivos. Veja outra gigante de TI, a SAP, utilizando o MVC. A figura a seguir demonstra o SAPUI5, na arquitetura MVC, com HTML5 adaptado à camada *View* da solução para cada tipo de dispositivo.

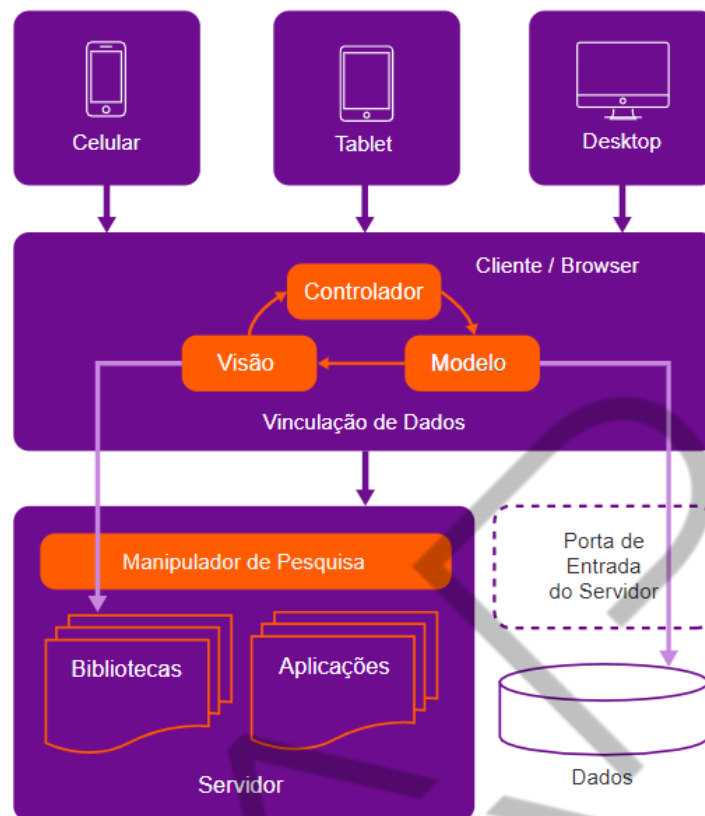


Figura 4 – SAPUI5 is a client UI technology
Fonte: SAP (2017), adaptado por FIAP (2018)

1.3 GoF Design Patterns

Como dissemos, a referência mais conhecida relacionada a padrões de projeto de *software* é o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, da Gang of Four ou GoF.

Os 23 padrões de *design* documentados no livro estão segregados em três tipos de padrões:

1.3.1 Padrões de Criação

Em aplicações Orientadas a Objeto, criar objetos é uma necessidade muito comum, que, seguindo o princípio do encapsulamento, deve ter a complexidade isolada para se resolver alguns problemas:

- Como podemos abstrair e simplificar o processo de criação de objetos?
- Como devemos encapsular quais classes concretas a aplicação usa?

- Como podemos encapsular quais instâncias dessas classes concretas são criadas e como se relacionam com outras?

Os 23 *Design Patterns* GoF catalogados, para resolver tais questões, são:

Factory Method

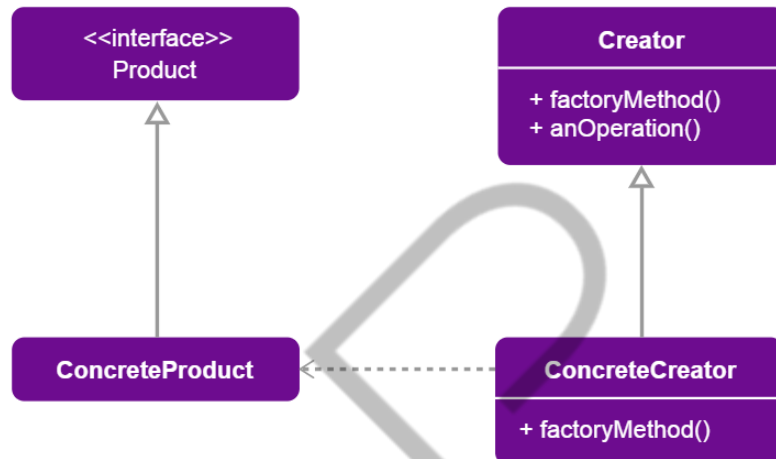


Figura 5 – UML Factory Method
Fonte: Elaborado pelo autor (2017)

- Define uma interface para a criação de um objeto, mas delega para as subclasses a decisão de qual classe será instanciada, sem que o restante da aplicação tenha de conhecer tais subclasses.
- Encapsula a escolha da classe concreta que será utilizada.
- Resolve a situação na qual o cliente não quer ou não pode saber qual implementação concreta está usando.

Abstract Factory

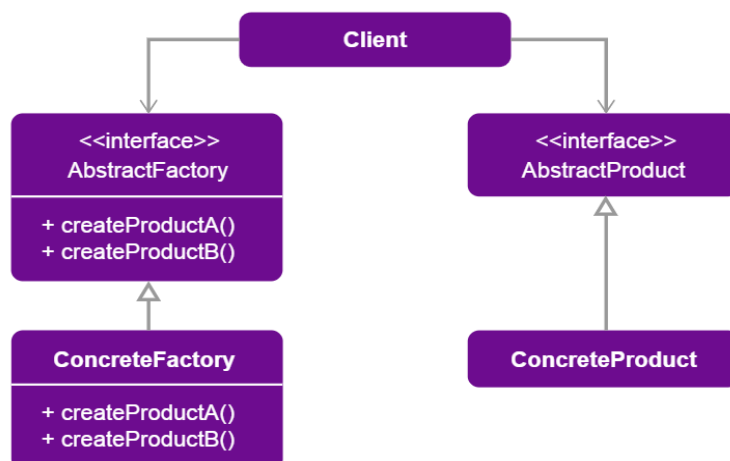


Figura 6 – UML Abstract Factory
Fonte: Elaborado pelo autor (2017)

- Define uma interface para a criação de uma família de objetos relacionados ou dependentes, sem especificar as implementações das classes concretas.
- Encapsula a escolha das classes concretas que serão utilizadas na criação dos objetos de diversas famílias.
- O código do cliente não terá conhecimento algum do tipo concreto. Os objetos concretos serão criados pela fábrica e o código do cliente acessará tais objetos através da interface.

Builder

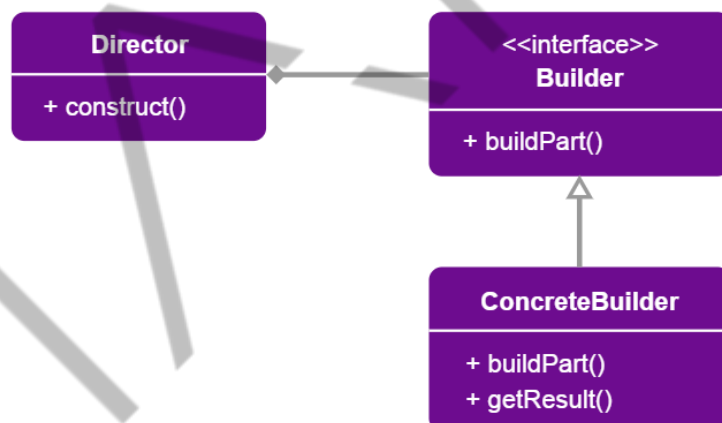


Figura 7 – UML Builder
Fonte: Elaborado pelo autor (2017)

- Separa a construção de um objeto complexo de sua representação, para que o mesmo processo de construção possa criar representações diferentes.
- Diferentes tipos de objetos podem ser criados em cada passo do processo de construção.
- Imagine o processo de criação de um objeto bicicleta, com diversos componentes que podem variar de mais simples e baratos a mais complexos e caros. Ainda criamos uma bicicleta,

independentemente dos componentes que escolhermos na criação. Concorde? O padrão Builder visa organizar esse tipo de criação quando temos objetos complexos.

Prototype

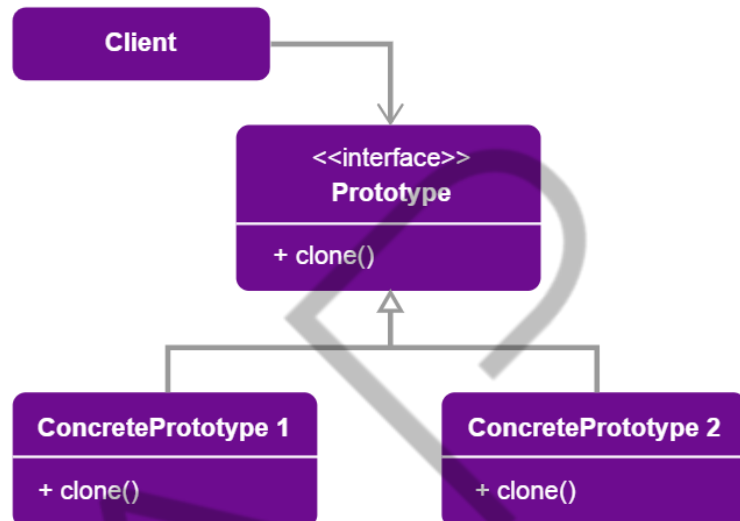


Figura 8 – UML Prototype
Fonte: Elaborado pelo autor (2017)

- Especifica tipos de objetos a serem criados usando uma instância como protótipo e criando novos objetos a partir da cópia desse protótipo.
- Possibilita a criação de novos objetos a partir da clonagem de um objeto existente, aproveitando o seu estado.
- Imagine a Receita Federal fazendo cinco tipos diferentes de análise em declarações de um contribuinte, com décadas de contribuição. A melhor solução seria criar o objeto com todas as declarações cinco vezes, uma a cada tipo de análise, ou criá-lo uma única vez, contendo todas as declarações e cloná-lo para realizar cada tipo de análise?

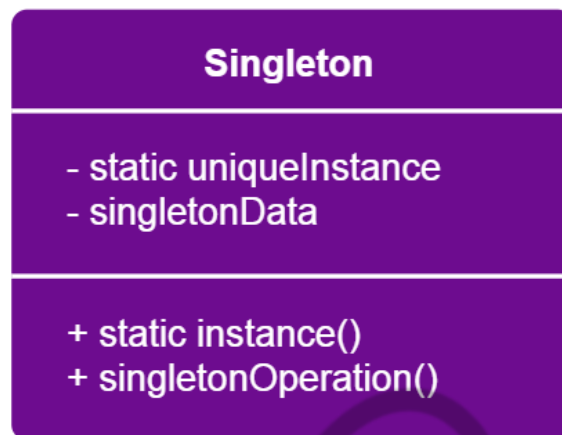
Singleton

Figura 9 – UML Singleton
Fonte: Elaborado pelo autor (2017)

- Garante que uma classe só tenha uma única instância, com um ponto de acesso global a ela.
- Uma classe nesse padrão oferece apenas uma instância de objeto dela, independentemente do número de requisições de criação que receber. A quantidade de instâncias será sempre controlada por ela mesma.
- Situações comuns de uso:
 - Um único objeto para uma conexão JDBC.
 - Um único objeto para representar as configurações do aplicativo.
 - Um único objeto para acesso ao log do aplicativo.

1.3.2 Padrões Estruturais

Em um aplicativo Orientado a Objetos, as relações entre objetos podem gerar forte dependência entre eles. Um aplicativo muito acoplado tem sua manutenção bem custosa, pois uma alteração em uma classe pode afetar todas as que se relacionam com ela.

Os padrões estruturais tratam dessas associações, identificando a melhor forma de realizar o relacionamento entre as entidades e de combiná-las em estruturas maiores, tornando-as menos complexas e acopladas.

Os padrões de classes estruturais usam herança para compor classes. Enquanto os padrões de objetos estruturais descrevem maneiras de conectar objetos para atingir um determinado fim.

Adapter

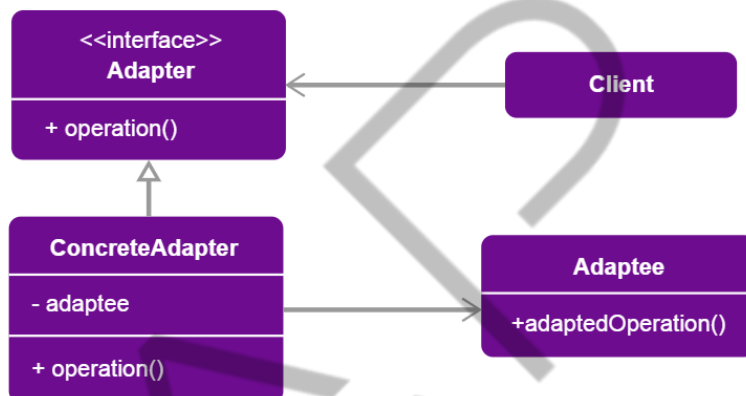


Figura 10 – UML Adapter
Fonte: Elaborado pelo autor (2017)

- Converte a interface de uma classe em outra interface esperada pelos clientes. Adapter, também conhecido como Wrapper, cria uma adaptação para permitir a comunicação entre classes que não poderiam trabalhar juntas, devido à incompatibilidade de suas interfaces.
- Um exemplo largamente utilizado nas literaturas técnicas é o emprego de um adaptador entre uma tomada de três pinos e um plug com dois pinos, de um eletrodoméstico mais antigo. Uma classe Adapter faz o papel desse adaptador.
- Em software, essa abordagem é muito utilizada em integrações de um sistema novo ou camada de apresentação nova, com um sistema existente, evitando a alteração do que já foi codificado no sistema existente. Ou seja, uma estratégia utilizando Adapter será desenvolvida para adaptar a integração do novo sistema, ou da nova

camada de visualização, ao código e aos comportamentos já existentes.

Bridge

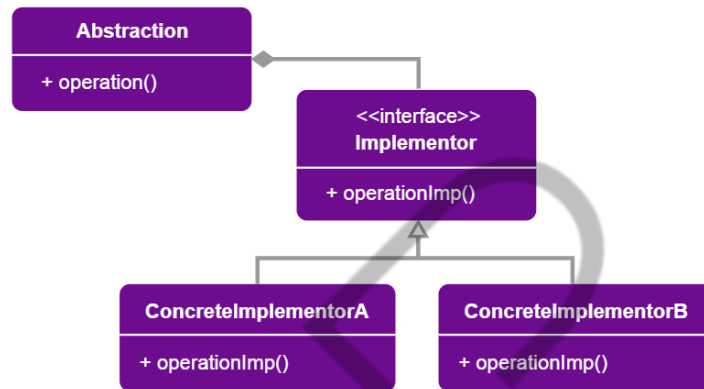


Figura 11 – UML Bridge
Fonte: Elaborado pelo autor (2017)

- Este padrão desacopla uma abstração de sua implementação, para que os dois possam variar independentemente e assim produzir tipos de objetos diferentes.
- A abstração, neste caso, significa a representação de um elemento com o qual um cliente tem interesse em interagir, e a implementação, a representação de operações que a abstração deve realizar. Esse padrão separa o que varia na solução e encapsula as variações separadamente na abstração e na implementação.
- Imagine aplicações clientes que realizem logs de erro, em arquivos ou em banco de dados. Elas terão relação com a superclasse Log, que será especializada pelas subclasses LogArq e LogBd. Agora, imagine que a formatação do que será gravado no log pode acontecer por tipo de erro ou por data ou por hora. Sem utilizar Bridge, você teria três heranças em cada subclasse para implementar cada tipo de formatação, pior, heranças com códigos repetidos.

- Utilizando Bridge, as subclasses não serão especializadas e a superclasse Log terá um relacionamento de agregação com uma interface chamada Formato. A interface Formato terá três implementações, uma classe para formatar por tipo de erro, outra por data e outra por hora. Evitando, assim, as heranças nas subclasses para tratar a implementação de formatação.
- Com essa solução, evitamos códigos duplicados, e se outros locais de gravação forem solicitados ou novas formas de formatação surgirem, teremos baixo impacto nas classes e relacionamentos criados.

Composite

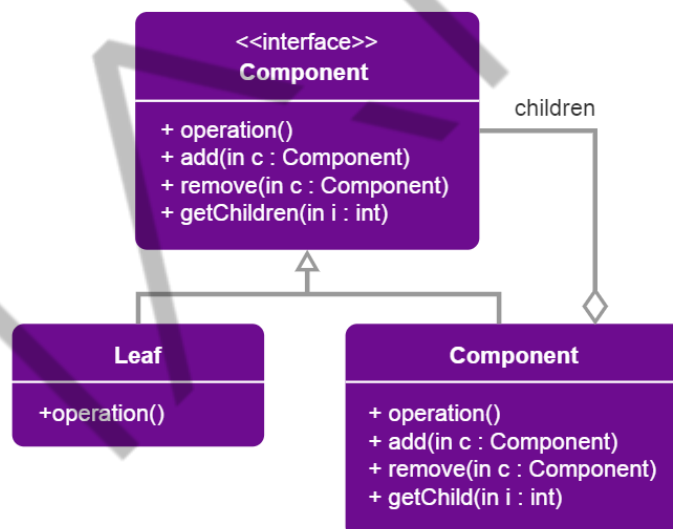


Figura 12 – UML Composite
Fonte: Elaborado pelo autor (2017)

- Compor objetos em estruturas de árvore para representar hierarquias todo-partes. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
- Ainda utilizando o exemplo da bicicleta, imagine uma loja que vende bicicletas montadas de acordo com o desejo do cliente. Seria uma boa solução ter todos os possíveis opcionais como atributos da nossa classe Bicicleta? E se o

cliente não quiser um jogo de marchas? E se ele pedir apenas um sistema de freio para a roda traseira? Se a loja começar a trabalhar com um novo opcional? Teremos de criar mais um atributo na classe e alterar o método que calcula preços? Não parece uma boa solução, né?

- O padrão Composite resolverá esse problema, tratando o conjunto que forma a bicicleta como um indivíduo.

Decorator

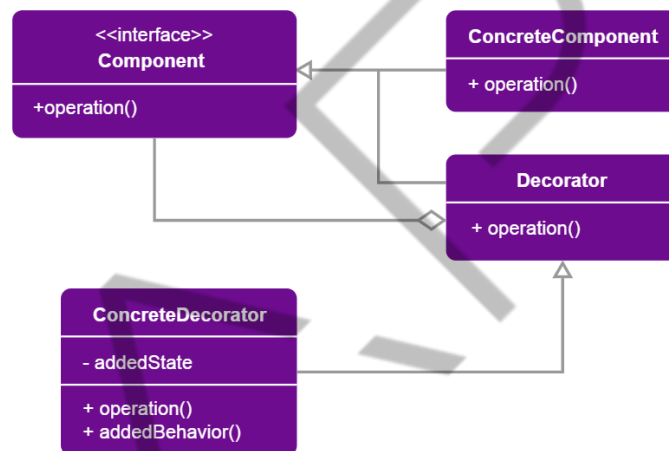


Figura 13 – UML Decorator
Fonte: Elaborado pelo autor (2017)

- Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade. Os métodos comuns a uma família de subclasses podem ser codificados em classes separadas para serem utilizados quando necessário.
- O Decorator foi utilizado nas classes do pacote Java IO. `InputStream` foi codificado em uma classe abstrata, `BufferedInputStream`, como uma implementação concreta de `InputStream` e `GzipInputStream` também, para tratar de compactação. Ambas possuem construtores que recebem como parâmetro um `InputStream`. Dessa forma, podemos pegar qualquer objeto `InputStream` como um `FileInputStream` e colocá-lo em um Buffer em memória e

depois passar esse Buffer para o construtor de um objeto GzipInputStream descompactá-lo.

Facade

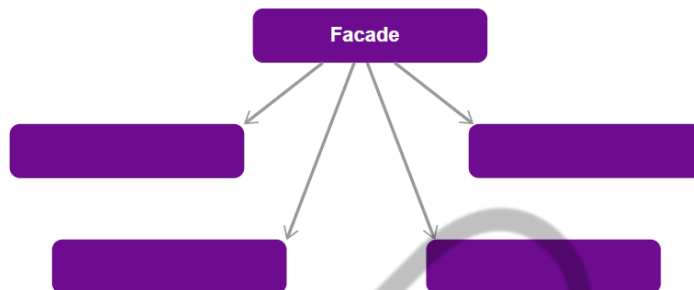


Figura 14 – UML Facade
Fonte: Elaborado pelo autor (2017)

- Oferece uma interface única para um conjunto de interfaces de um subsistema. O padrão Facade define uma interface de nível mais elevado, que torna o subsistema mais fácil de usar.
- Imagine que para efetivar um pedido da nossa bicicleta customizada, o sistema deva verificar o estoque, reservar os componentes no estoque, calcular os impostos da venda e programar a entrega. Como serão diversas classes responsáveis por fazer tudo isso, empregando o padrão Facade, codificaremos uma classe que atuará como uma fachada para o cliente. A classe receberá o pedido do cliente, chamará todos os objetos envolvidos e devolverá uma resposta ao cliente, simplificando a interface para o subsistema que realiza o pedido, controlando a transação (todos os passos do subsistema devem ser executados com sucesso) e reduzindo o acoplamento. O cliente se relaciona com o Facade, e não com todas as classes que atendem ao processo de pedido.

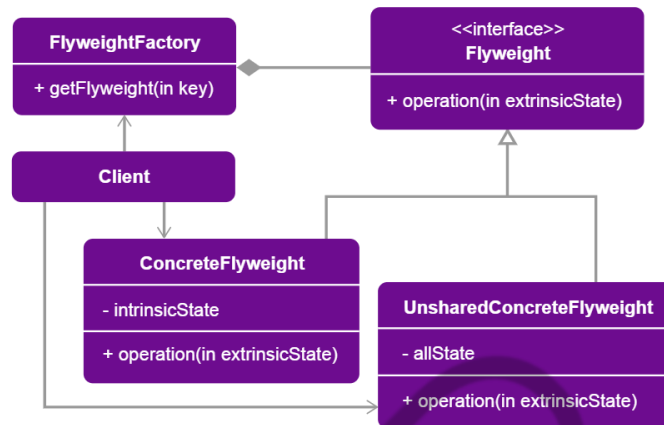
Flyweight

Figura 15 – UML Flyweight
Fonte: Elaborado pelo autor (2017)

- Compartilhar, de forma eficiente, objetos que são usados em grande quantidade.
- Flyweight é um padrão que propõe a técnica de usar compartilhamento para dar suporte a grandes quantidades de objetos refinados eficientemente. Para aplicar esse padrão, precisamos criar uma Factory a fim de gerenciar a criação de objetos, reaproveitando instâncias e só criando novos objetos quando eles não tiverem sido instanciados. A classe String emprega o padrão Flyweight.

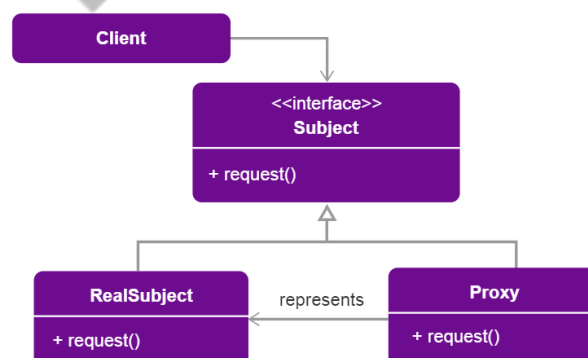
Proxy

Figura 16 – UML Proxy
Fonte: Elaborado pelo autor (2017)

- O padrão Proxy fornece um objeto representante com a mesma interface de outro objeto, de forma a controlar o acesso ao objeto por intermédio do representante.

- O objeto representante contém uma referência ao objeto real e repassa chamadas, geralmente adicionando funcionalidades e acrescentando ou filtrando dados na comunicação.
- Esse padrão é comumente utilizado em objetos distribuídos, o cliente envia mensagens para o Proxy e o Proxy se comunica com o objeto real.

1.3.3 Padrões Comportamentais

Os *Design Patterns* comportamentais identificam padrões comuns de comunicação entre objetos e determinam formas de realizar esses padrões. Ao fazê-las, aumentam a flexibilidade e diminuem o acoplamento na realização da comunicação.

Podemos dizer que os padrões comportamentais descrevem a forma como os objetos e as classes interagem e dividem as responsabilidades entre eles, usam composição em vez da herança.

Para entendermos um pouco melhor, alguns desses padrões descrevem como um grupo de objetos pares coopera para executar uma tarefa que nenhum objeto pode realizar sozinho.

Uma questão importante aqui é como os objetos pares se conhecem. Os pares poderiam manter referências explícitas entre si, mas isso aumentaria o seu acoplamento. No extremo, todos os objetos saberiam sobre todos os outros.

Os padrões comportamentais resolvem esse tipo de problema, facilitam a comunicação e garantem um baixo acoplamento entre os pares.

Chain of Responsibility

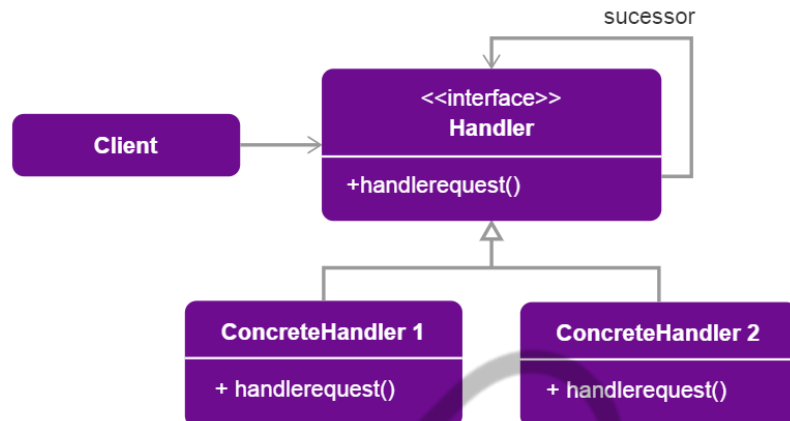


Figura 17 – UML Chain of Responsibility
Fonte: Elaborado pelo autor (2017)

- Evita acoplar o remetente de uma requisição ao seu destinatário, ao dar a mais de um objeto a chance de servir à requisição. Encadeia os objetos receptores e transmite a solicitação através da cadeia até que um objeto a trate.
- Imagine uma aplicação que controla a aprovação de solicitação de descontos, em uma loja de departamentos, em que o vendedor tem um limite inferior ao limite do responsável pelo departamento e o responsável tem o limite de aprovação inferior ao do gerente da loja. O padrão Chain of Responsibility organiza essa situação em uma cadeia, na qual, toda a vez que um colaborador tem sua alçada de aprovação de desconto excedida, a solicitação de desconto é passada para o próximo nível da cadeia de responsabilidade.
- Vendedor > Responsável Depto. > Gerente.

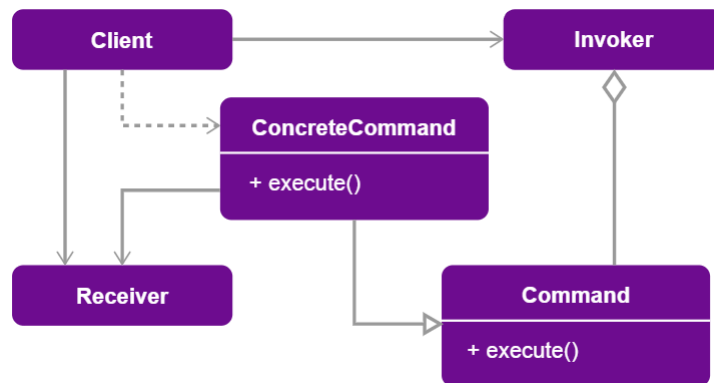
Command

Figura 18 – UML Command
Fonte: Elaborado pelo autor (2017)

- O padrão command encapsula uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições enfileirar ou registrar solicitações e implementar recursos de cancelamento de operações.
- O command controla as chamadas a um determinado componente, modelando cada requisição como um objeto. Isso permite emitir chamados para objetos sem saber nada sobre a operação que está sendo solicitada ou sobre o destinatário da solicitação.
- O comando desacopla o objeto que invoca a operação daquela que a executa. Para alcançar essa separação, devemos criar uma classe abstrata que mapeia um objeto com uma ação. A classe-base contém um `execute()`, método que simplesmente chama a ação no receptor.
- Todos os clientes tratam cada objeto command como uma “caixa preta”, simplesmente invocando o `execute()` método virtual do objeto sempre que o cliente precisa da operação do objeto.
- O command é muito aplicado para encapsular operações associadas a um objeto de interface gráfica, como um botão.

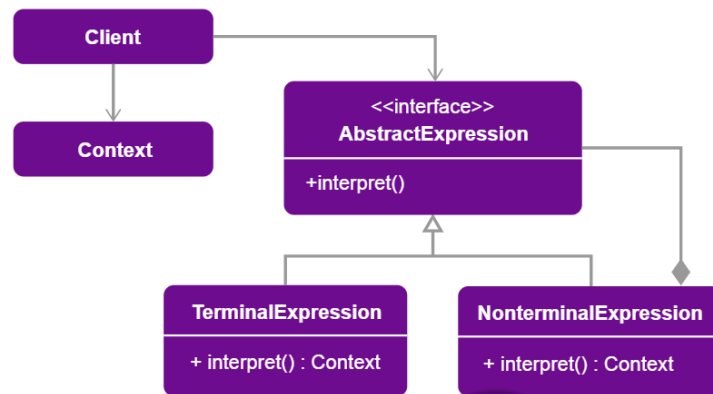
Interpreter

Figura 19 – UML Interpreter
Fonte: Elaborado pelo autor (2017)

- Dada uma linguagem, definir uma representação para sua gramática com um interpretador que usa a representação para interpretar sentenças na linguagem.
- Se tipos de problemas ocorrem com frequência em situações bem definidas e bem compreendidas, podemos descrever esses problemas com uma linguagem simples e, então, eles poderiam ser facilmente resolvidos, com uma interpretação dessa linguagem por um programa.
- O padrão Interpreter sugere modelar o domínio com uma gramática recursiva. Cada regra na gramática é, portanto, um “composite” (uma regra que referencia outras regras) ou um “terminal” (um ponto final na estrutura de árvore de regras). O Interpreter baseia-se na execução recursiva do padrão Composite para interpretar as “sentenças” que ele deve processar.
- Um exemplo de emprego do Interpreter é utilizá-lo para realizar parsing de comandos SQL. Fazer um parse de um comando Select é um problema que ocorre com muita frequência, em situações bem definidas e conhecidas. O comando é descrito em uma linguagem simples e é estruturado em uma árvore de regras que

descrevem os campos a ser selecionados, as tabelas de origem, as condições de seleção, agrupamentos e ordenações, por exemplo.

Iterator

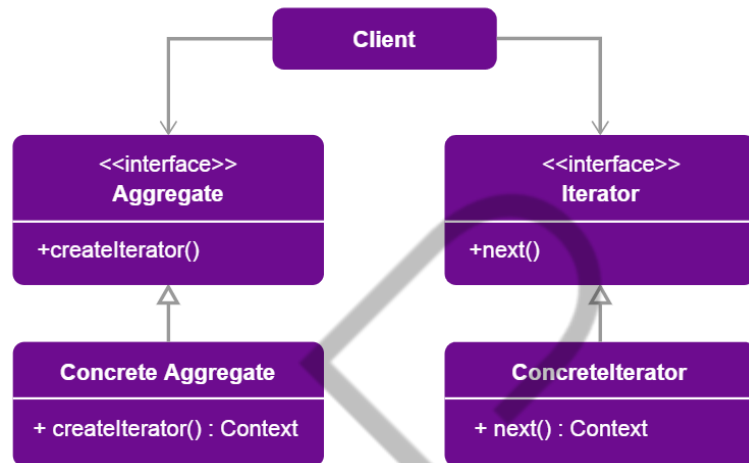


Figura 20 – UML Iterator
Fonte: Elaborado pelo autor (2017)

- O padrão Iterator fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação. Além disso, provê um modo eficiente para percorrer sequencialmente os elementos de uma coleção, em geral, uma lista, como se ele fosse um ponteiro, referenciando um elemento da coleção e se modificando para apontar para o próximo elemento. Em Java, esse padrão é muito utilizado para percorrer listas, conjuntos e mapas.

Mediator

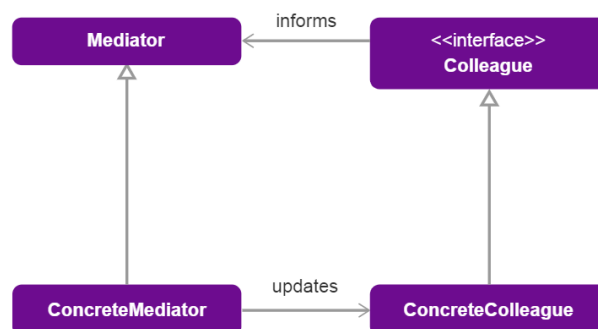


Figura 21 – UML Mediator
Fonte: Elaborado pelo autor (2017)

- Em aplicativos Orientados a Objeto, utilizar vários objetos coesos aumenta o reúso, mas também aumenta as comunicações, problema que, se não for tratado adequadamente, pode reduzir os benefícios de ter vários objetos, diminuindo o reúso em face do aumento do acoplamento.
- O padrão Mediator define um objeto que encapsula como um conjunto de objetos interage; promove acoplamento fraco, ao intermediar a comunicação multidirecional entre objetos, para que não se refiram um ao outro explicitamente, permitindo, assim, variar as interações de forma independente.
- Na implementação deste padrão, cada objeto participante conhece o Mediator, sem conhecer os demais. O Mediator recebe as requisições do remetente e repassa ao objeto destinatário.
- Um exemplo de Mediator seria o seu emprego para mediar passageiros clientes do Uber e motoristas que atendem via Uber. Uma classe para cada um deles e uma classe Mediator para gerir a comunicação entre eles. Uma vez que o passageiro não deve ter uma comunicação direta com o motorista e vice-versa.

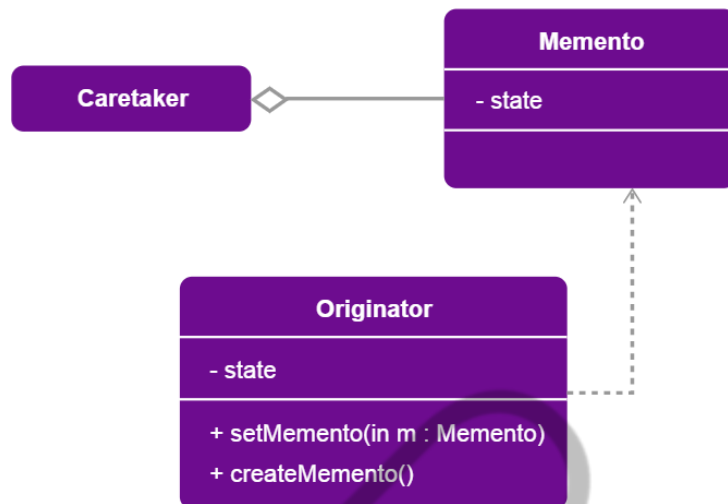
Memento

Figura 22 – UML Memento
Fonte: Elaborado pelo autor (2017)

- Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, para que ele possa ter esse estado restaurado posteriormente. Um exemplo de utilização do padrão Memento é empregá-lo quando você precisar implementar um mecanismo de desfazer em uma aplicação, ou seja, usando serialização com esse padrão, é fácil preservar um instantâneo do objeto antes de uma alteração e trazê-lo novamente mais tarde, se você optar por desfazer a alteração.

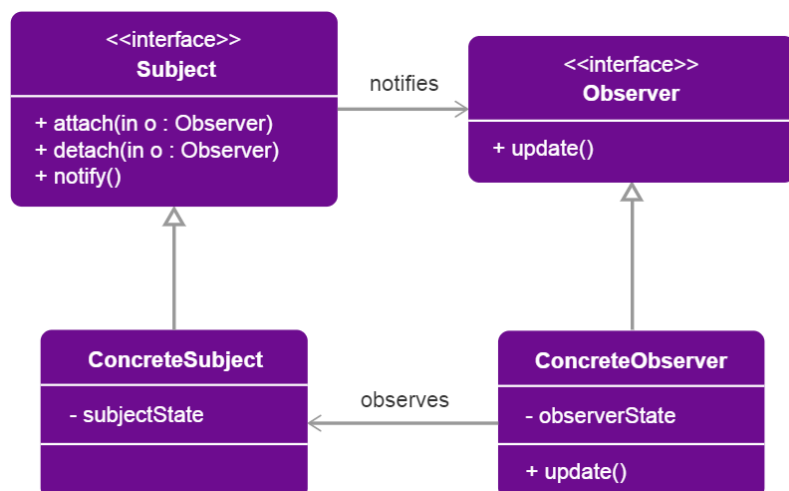
Observer

Figura 23 – UML Observer
Fonte: Elaborado pelo autor (2017)

- Define uma dependência um-para-muitos entre objetos, para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente.
- Imagine a central de monitoramento com um software distribuído em um conjunto de sensores. A melhor prática seria ter um objeto que fica perguntando aos objetos executados em cada sensor se houve alguma alteração, ou receber a informação do objeto/sensor que sofrer alguma alteração?
- O padrão Observer orienta uma solução para a segunda prática.

State

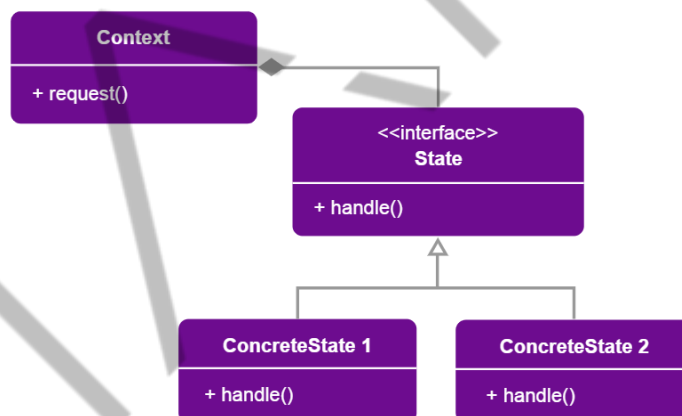


Figura 24 – UML State
Fonte: Elaborado pelo autor (2017)

- Permite a um objeto alterar seu comportamento quando seu estado interno mudar. O objeto vai aparentar mudar de classe.
- Imagine o cancelamento de uma conta em um restaurante, com três estados diferentes: não atendida, atendida parcialmente e atendida. O comportamento do método cancelar conta será diferente, de acordo com cada estado.

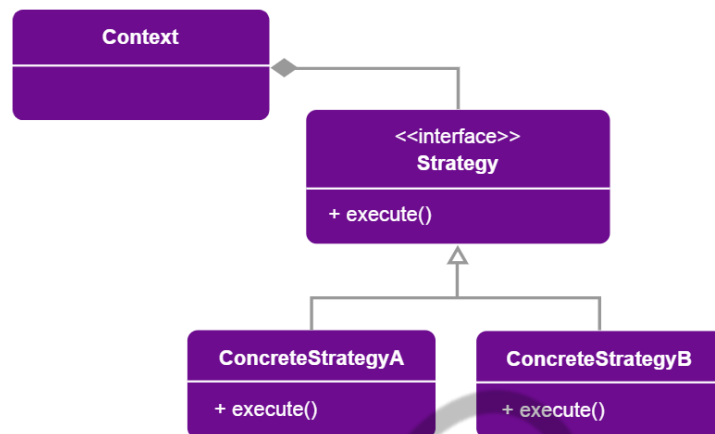
Strategy

Figura 25 – UML Strategy
Fonte: Elaborado pelo autor (2017)

- Este Design Pattern define uma família de algoritmos, encapsula cada algoritmo em uma classe e torna-os intercambiáveis. Strategy permite que algoritmos mudem, para prover comportamentos mais adequados de um objeto diante de uma situação, independentemente dos clientes que os utilizam.
- Imagine um sistema para a nossa loja de bicicletas que permite diversas regras de cálculo de desconto, de acordo com o valor da bicicleta e a forma de pagamento. O Strategy resolve esse tipo de problema.

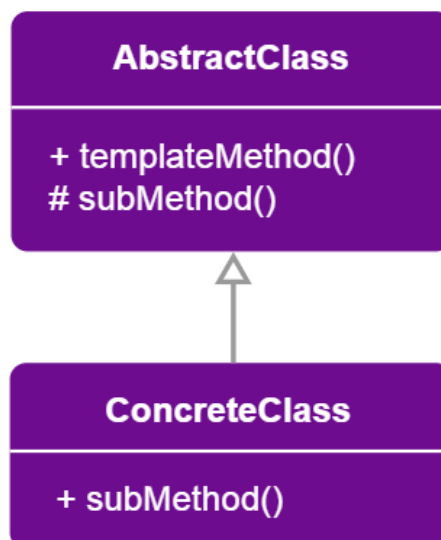
**Template
Method**

Figura 26 – UML Template Method
Fonte: Elaborado pelo autor (2017)

- Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
- Nós já vimos uma utilização desse padrão em outra fase deste curso, nos exemplos bancários que modelaram os tipos de conta a partir da herança de uma classe abstrata Conta. Como o cálculo da taxa vai variar de acordo com o tipo da conta, subclasses de conta, o método calcularTaxa() deve ser abstrato em Conta, para que cada subclasse o implemente de acordo com a especificidade que ela define.

Visitor

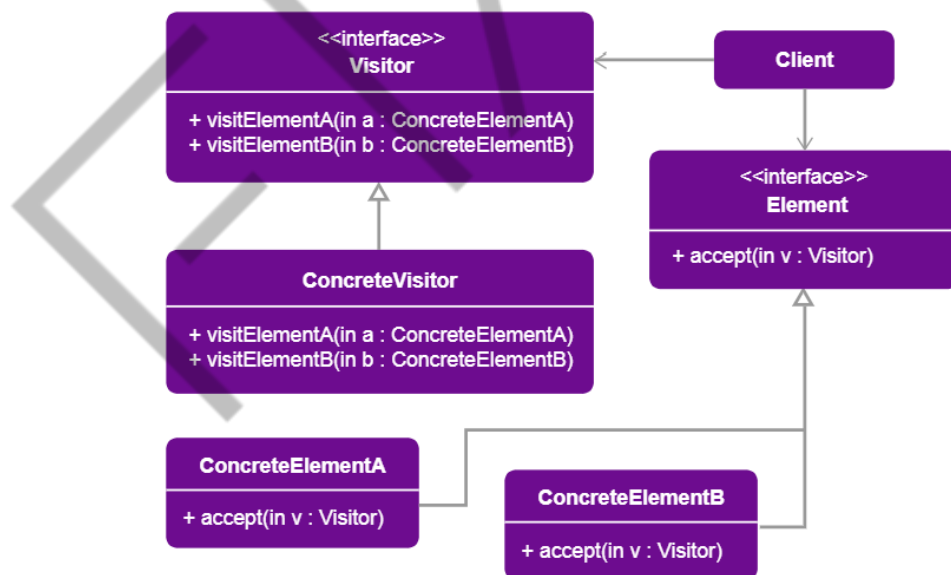


Figura 27 – UML Visitor
Fonte: Elaborado pelo autor (2017)

- Representa uma operação a ser realizada sobre os elementos de uma estrutura de objetos. Esse padrão permite que uma nova operação seja

definida, sem mudar as classes dos elementos sobre os quais opera.

- Imagine que vamos despachar as nossas bicicletas, mas os nossos pedidos são especializados por pedido local, com entrega na cidade, pedido fora, com entregas fora da cidade e pedidos de exportação. Todos são subclasses de Pedido, portanto, elementos de uma mesma estrutura de objetos. Na cidade, entregaremos com uma van própria; fora da cidade, a entrega será feita com uma transportadora, e para exportar, a entrega será realizada com um grupo de transporte. Esse padrão criará uma interface com uma assinatura diferente para cada método. Cada método tratará uma especialização de pedido e uma classe que implementará essa interface, com um método codificado, que, para cada tipo de pedido, utilizará a forma adequada de entrega. Como exemplo, para PedidoLocal setEntrega PedidoVan. Este padrão facilita a adição de novas funcionalidades em objetos sem o uso de ifs e sem mexer na estrutura de herança, mas, se uma nova subclasse for adicionada, um novo método deverá ser acrescentado na Interface Visitor e na classe que a implementa.

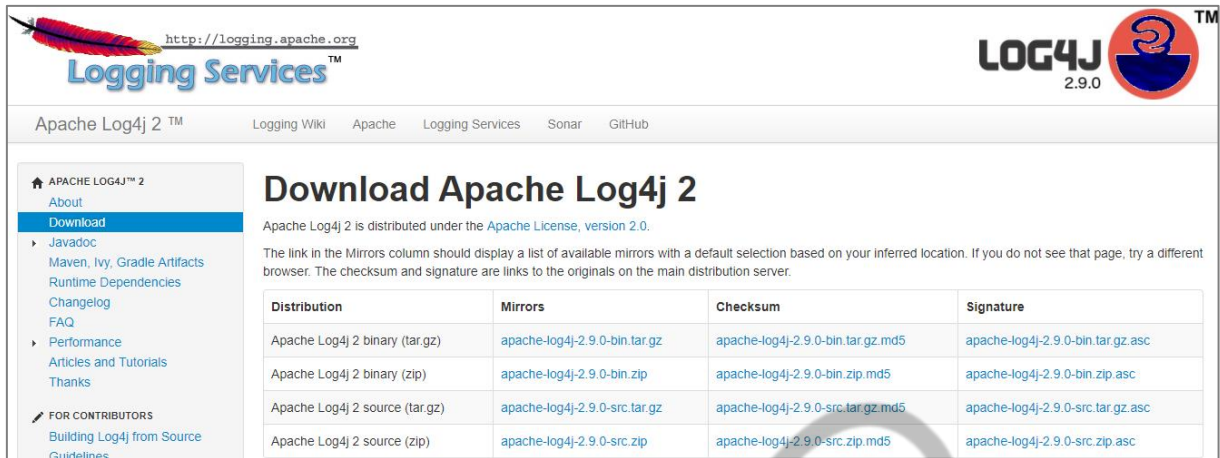
2 JAVA LOGGING FRAMEWORK

Esse é um caso típico de emprego de *framework*, ou seja, a melhor saída seria escolher um conjunto de códigos-fonte, bem implementados e largamente testados, que atendam ao propósito genérico de gravar logs. Sendo assim, vamos escolher entre as opções de mercado o Framework Java Apache Log4j e, com ele, resolver o problema específico de logs do nosso projeto.

Antes de iniciar o download da biblioteca Log4j, faça o download do projeto-base que se encontra no final da fase (fiap-tds2-fase1-base.zip) e importe o projeto na IDE que você está mais ambientado (**neste projeto, usamos o Eclipse IDE for Enterprise Java Developers – Versão 2019-03 (4.11.0) com a versão Java JDK 11.0.7, Dynamic Web Module 4.0 e Apache Tomcat 9.0**).

Com o projeto aberto na IDE, baixe a versão .zip binária do Log4j em:

<<http://logging.apache.org/log4j/2.x/download.html>>.



Apache Log4j 2™

Logging Wiki Apache Logging Services Sonar GitHub

APACHE LOG4J™ 2

- About
- Download
- Javadoc
- Maven, Ivy, Gradle Artifacts
- Runtime Dependencies
- Changelog
- FAQ
- Performance
- Articles and Tutorials
- Thanks

FOR CONTRIBUTORS

- Building Log4j from Source
- Guidelines

Download Apache Log4j 2

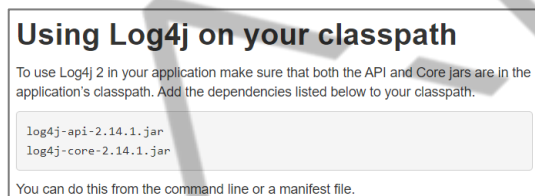
Apache Log4j 2 is distributed under the [Apache License, version 2.0](#).

The link in the Mirrors column should display a list of available mirrors with a default selection based on your inferred location. If you do not see that page, try a different browser. The checksum and signature are links to the originals on the main distribution server.

Distribution	Mirrors	Checksum	Signature
Apache Log4j 2 binary (tar.gz)	apache-log4j-2.9.0-bin.tar.gz	apache-log4j-2.9.0-bin.tar.gz.md5	apache-log4j-2.9.0-bin.tar.gz.asc
Apache Log4j 2 binary (zip)	apache-log4j-2.9.0-bin.zip	apache-log4j-2.9.0-bin.zip.md5	apache-log4j-2.9.0-bin.zip.asc
Apache Log4j 2 source (tar.gz)	apache-log4j-2.9.0-src.tar.gz	apache-log4j-2.9.0-src.tar.gz.md5	apache-log4j-2.9.0-src.tar.gz.asc
Apache Log4j 2 source (zip)	apache-log4j-2.9.0-src.zip	apache-log4j-2.9.0-src.zip.md5	apache-log4j-2.9.0-src.zip.asc

Figura 28 – Página de download, Logging Services Log4j
Fonte: Logging Services (2017)

Para usar o Log4j, precisamos descompactar o zip e adicionar os dois *jars* ao *classpath* do projeto.



Using Log4j on your classpath

To use Log4j 2 in your application make sure that both the API and Core jars are in the application's classpath. Add the dependencies listed below to your classpath.

```
log4j-api-2.14.1.jar  
log4j-core-2.14.1.jar
```

You can do this from the command line or a manifest file.

Figura 29 – Usando Log4j
Fonte: Logging Services (2017)

Para usar o Log4j, copie e cole no diretório lib os arquivos (log4j-api-2.14.0.jar e log4j-core-2.14.0.jar) e, com o botão direito, adicione-os ao Build Path do projeto.

Esses arquivos contêm os códigos implementados e amplamente testados por equipes da Apache e estão disponíveis para reaproveitarmos na resolução do nosso problema específico, que é gravar os cálculos de impostos em arquivos de log.

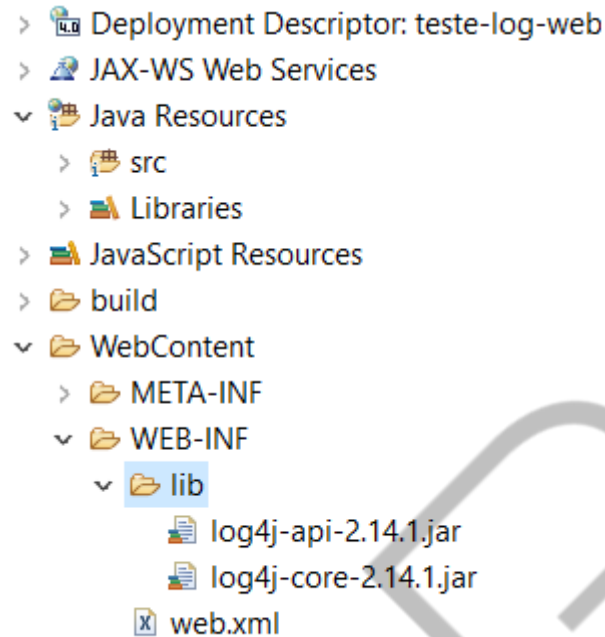


Figura 30 – Adicionando as classes ao Build Path
Fonte: Elaborado pelo autor (2017)

Para configurar o Log4j, crie um arquivo de nome **log4j2.properties** na pasta src do projeto.

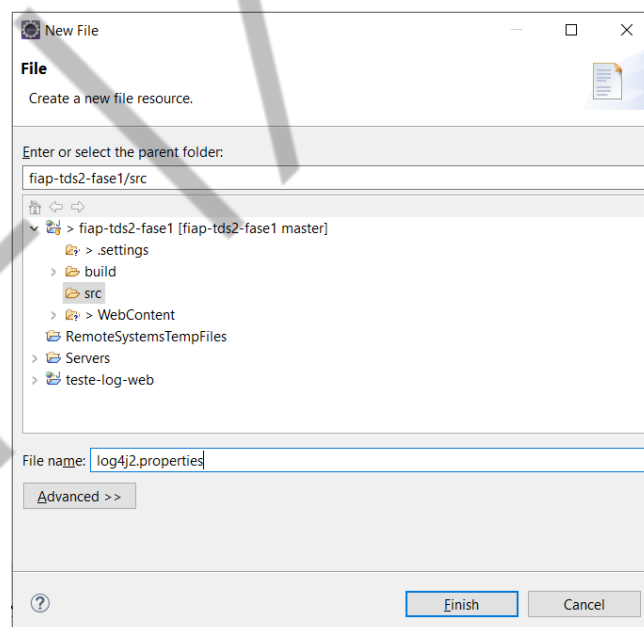


Figura 31 – Criando o arquivo log4j2.properties
Fonte: Elaborado pelo autor (2017)

No arquivo log4j2.properties, parametrize:

```
name = PropertiesConfigPis
```

```
#Diretório que vamos salvar o log e o nome do arquivo
property.filename = c:/temp/logs/fiap-tds2-fase1.log

filters = threshold

filter.threshold.type = ThresholdFilter
filter.threshold.level = info

#Especifica o appender
appenders = rolling

#RollingFile é um appender que cria um novo arquivo de
acordo com as políticas
appender.rolling.type = RollingFile
appender.rolling.name = RollingFile

appender.rolling.fileName = ${filename}
#Define onde e com que nome arquivar os logs antigos
appender.rolling.filePattern = logs/${date:yyyy-MM}/fiap-%d{MM-dd-yyyy}-%i.log.gz
#Define como cada linha de log será escrita
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = %d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

#Determina as políticas para a quebra
appender.rolling.policies.type = Policies
#Quebra porque o padrão de data / hora não se aplica
mais ao arquivo
appender.rolling.policies.time.type = TimeBasedTriggeringPolicy
appender.rolling.policies.time.interval = 1
appender.rolling.policies.time.modulate = true
#Quebra quando o arquivo atingir 10MB
appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
appender.rolling.policies.size.size=10MB

#Exclui os arquivos de log mais antigos quando
chegarmos a 20
appender.rolling.strategy.type = DefaultRolloverStrategy
appender.rolling.strategy.max = 20

vloggers = rolling

logger.rolling.name = br.com.fiap
logger.rolling.level = debug
logger.rolling.additivity = false
logger.rolling.appenderRef.rolling.ref = RollingFile
```

Código-fonte 1 – Log4j2.properties parametriza o uso do Log4j
Fonte: Elaborado pelo autor (2017)

Com esse arquivo, configuramos o Log4j para gravar em um arquivo chamado `fiap-tds2-fase1.log`, na pasta logs que será criada com os diretórios `c:/temp`. Configuramos o Log4j para trabalhar com o *appender* RollingFile. Esse *appender* é específico para gravar em arquivos existentes e para criar novos arquivos de log, quando uma determinada política for aplicável. Veja os comentários para melhor entendimento.

O Log4j possui diversos *appenders*, para as mais diferentes funções, entre eles, podemos citar: *appenders* para JDBC, JPA, Cassandra, SMTP e Console.

Agora que configuramos o Log4j, vamos validar se tudo está correto. Primeiro, verifique o código do método `doGet` da classe `br.com.fiap.tds2.fase1.controller.IndexController`. Em seguida, observe as linhas das chamadas dos métodos `debug`, `info` e `erro` da instância da classe `Logger`. Depois, execute a aplicação no servidor Apache Tomcat e, pelo navegador, faça uma chamada ao `IndexController` (url: <http://localhost:8080/fiap-tds2-fase1/Index>).

Ao executarmos o servlet, veremos que o Log4j criou a pasta e o arquivo de logs em `c:/temp/logs/fiap-tds2-fase1.log`.

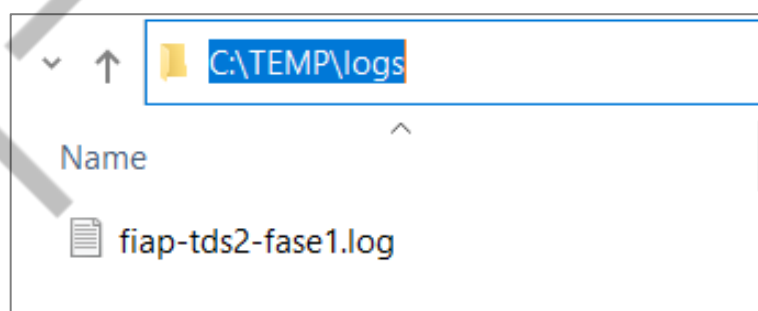


Figura 32 – Pasta e arquivo de log criado pelo Log4j
Fonte: Elaborado pelo autor (2017)

E o resultado do arquivo de log depois da primeira execução contendo os valores das 3 linhas descritas na classe `IndexController`.


```

fiap-tds2-fase1.log X
C: > TEMP > logs > fiap-tds2-fase1.log
1 2021-12-6PM 22:14:39 DEBUG IndexController:27 - Debug Log4j
2 2021-12-6PM 22:14:39 INFO IndexController:28 - Info Log4j
3 2021-12-6PM 22:14:39 ERROR IndexController:29 - Error Log4j
4 |

```

Figura 33 – Conteúdo do log criado pelo Log4j
Fonte: Elaborado pelo autor (2017)

Você pode estar se perguntando, mas de onde surgiram as demais informações além do texto descrito nos comandos de log?

Para responder a essa pergunta, precisamos retornar às configurações do Log4j, o arquivo `log4j2.properties` responderá através das seguintes linhas:

```

#Define como cada linha de log será escrita
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = %d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n

```

Código-fonte 2 – Treco do arquivo `log4j2.properties` que parametriza o uso do Log4j
Fonte: Elaborado pelo autor (2017)

O responsável por formatar a linha do log foi a parametrização `appender.layout.pattern`, da seguinte forma:

Padrão de formatação	Resultado esperado	Resultado obtido no arquivo de log
<code>%d{yyyy-MM-dd HH:mm:ss}</code>	Data e hora formatadas	2017-09-14 17:18:45
<code>%-5p</code>	Nível do log	INFO
<code>%c{1}</code>	Nome da classe	Pis
<code>%L</code>	Linha do log	39
<code>%m</code>	Mensagem	Valor: 1000.0 Alíquota: 0.0065 Valor do Pis: 6.5
<code>%n</code>	Quebra de linha	

Quadro 1 – Parametrização `appender.layout.pattern`
Fonte: Elaborado pelo autor (2017)

O nível do log `%-5p` foi informado em `LOGGER.info` e a mensagem foi o parâmetro `LOGGER.info(%m)`.

```
LOGGER.debug("Debug Log4j");  
LOGGER.info("Info Log4j");  
LOGGER.error("Error Log4j");
```

Figura 34 – Trecho do código que grava o log no Eclipse
Fonte: Elaborado pelo autor (2017)

Se desejarmos formatar de outras maneiras, é só alterarmos o **appender.layout.pattern**. Se quisermos mudar o nível do log, é mais simples ainda. Podemos alterar, como exemplo, para **LOGGER.error**.

Agora ficou mais claro como a utilização de um *framework* pode nos ajudar? Você consegue imaginar o esforço para desenvolver todas essas funcionalidades dentro do seu projeto?

Nada fácil! Concorda?

Para mais detalhes sobre a utilização do Log4j, pesquise em:

<<https://logging.apache.org/log4j/2.x/manual/index.html>>.

Para praticar, abra a classe LoginController.java, no método doPost, verifique os comandos System.out.println e faça uma análise de como refatorar com o uso biblioteca Log4j. O primeiro passo para refatoração é criar a instância da classe Logger na LoginController. Veja o trecho de código abaixo:

```
@WebServlet("/Login")  
public class LoginController extends HttpServlet {  
  
    // INSTANCIA DA CLASSE LOGGER  
    private static final Logger LOGGER =  
    LogManager.getLogger(LoginController.class.getName());  
  
    public LoginController() {  
        super();  
    }  
}
```

Código-fonte 3 – IndexController, instanciando a classe de Log
Fonte: Elaborado pelo autor (2017)

Depois de criada a instância da classe de Logger, troque os comandos

System.out.println pelos comandos de Log que achar mais aderentes à necessidade. Veja o exemplo abaixo:

```
//System.out.println("Iniciando a execução do  
IndexController");  
LOGGER.debug("Iniciando a execução do IndexController");
```

Código-fonte 4 – IndexController, uso do logger.
Fonte: Elaborado pelo autor (2017)

Execute o projeto e faça alguns testes na funcionalidade de Login. Depois, verifique como ficou o conteúdo do arquivo C:\TEMP\logs\fiap-tds2-fase1.log.

Para aprofundar ainda mais no Log4j, no arquivo log4j2.properties, altere o parâmetro de configuração **logger.rolling.level**, mudando o valor de **debug** para **info** e depois **error**. Faça o restart da aplicação, execute a funcionalidade de login novamente e verifique como ficou o arquivo de log.

Segue o link da solução completa:

https://drive.google.com/drive/folders/16Oafc4saxqi53L-9NJDOjf7qIMUx7N_T?usp=sharing

2.1 Facade Pattern hands-on

Vimos, anteriormente, a teoria do padrão Facade e agora vamos colocar em prática uma simples refatoração do projeto para deixar nosso código mais organizado e limpo com o uso desse padrão.

O primeiro passo é criar uma classe `br.com.fiap.tds2.fase1.business.LoginBusiness` e transferir os métodos que foram implementados na classe `LoginController`. Veja o código abaixo:

```
package br.com.fiap.tds2.fase1.business;

import br.com.fiap.tds2.fase1.exceptions.LoginAtivoException;
import br.com.fiap.tds2.fase1.exceptions.LoginExcedeuTentativasException;
import br.com.fiap.tds2.fase1.exceptions.LoginExpiradoException;
import br.com.fiap.tds2.fase1.exceptions.LoginInvalidoException;
import br.com.fiap.tds2.fase1.exceptions.LoginIpInvalidoException;
import br.com.fiap.tds2.fase1.model.UsuarioModel;

public class LoginBusiness {

    public void validarLogin(UsuarioModel usuarioModel) throws LoginInvalidoException {
        if ( ! usuarioModel.getLogin().equals("ok@mail.com")
        || !usuarioModel.getSenha().equals("123") ) {
            throw new LoginInvalidoException("Usuário ou senha incorreta");
        }
    }

    public void validarUsuarioAtivo(UsuarioModel usuarioModel) throws LoginAtivoException {
        if (
        usuarioModel.getLogin().equals("login.ativo@mail.com") ) {
            throw new LoginAtivoException("Usuário logado em outro equipamento");
        }
    }

    public void validarTentativasdeLogin(UsuarioModel usuarioModel) throws LoginExcedeuTentativasException {
        if (
```

```
usuarioModel.getLogin().equals("tentativas@mail.com") ) {  
    throw new  
    LoginExcedeuTentativasException("Tentativas excedidas");  
}  
  
}  
  
    public void validarUsuarioExpirado(UsuarioModel  
usuarioModel) throws LoginExpiradoException {  
        if (  
usuarioModel.getLogin().equals("expirado@mail.com") ) {  
            throw new LoginExpiradoException("Login  
expirado");  
        }  
    }  
  
    public void validarIp(UsuarioModel usuarioModel) throws  
LoginIpInvalidoException {  
        if ( usuarioModel.getLogin().equals("ip@mail.com") )  
{  
            throw new LoginIpInvalidoException("Ip  
suspeito");  
        }  
    }  
  
}
```

Código-fonte 5 – LoginBusiness, implementação das regras de negócio
Fonte: Elaborado pelo autor (2021)

Agora vamos pensar em um seguinte cenário de negócio. Precisamos criar duas regras de login para o nosso sistema. A primeira será usada por todos que acessam o serviço pela web e a segunda será aplicada aos usuários do serviço pelo aplicativo móvel. A diferença dos dois tipos de acessos é que a forma pelo aplicativo móvel não precisa fazer a validação de IP.

Para a nossa solução, vamos criar a classe Facade que irá tratar os dois tipos de acessos, deixando para uma forma bem simples e clara de decidir qual mecanismo de login executar. Crie a classe `br.com.fiap.tds2.fase1.business.LoginFacade` com o código abaixo:

```
package br.com.fiap.tds2.fase1.business;  
  
import br.com.fiap.tds2.fase1.exceptions.LoginAtivoException;
```

```
import
br.com.fiap.tds2.fase1.exceptions.LoginExcedeuTentativasExcept
ion;
import
br.com.fiap.tds2.fase1.exceptions.LoginExpiradoException;
import
br.com.fiap.tds2.fase1.exceptions.LoginInvalidoException;
import
br.com.fiap.tds2.fase1.exceptions.LoginIpInvalidoException;
import br.com.fiap.tds2.fase1.model.UsuarioModel;

public class LoginFacade {

    private LoginBusiness loginBusiness = new LoginBusiness();

    public void loginWeb(UsuarioModel usuarioModel) throws
        LoginAtivoException,
        LoginExcedeuTentativasException,
        LoginExpiradoException,
        LoginIpInvalidoException,
        LoginInvalidoException {

        loginBusiness.validarUsuarioAtivo(usuarioModel);
        loginBusiness.validarTentativasdeLogin(usuarioModel);
        loginBusiness.validarUsuarioExpirado(usuarioModel);
        loginBusiness.validarIp(usuarioModel);
        loginBusiness.validarLogin(usuarioModel);

    }

    public void loginMobile(UsuarioModel usuarioModel) throws
        LoginAtivoException,
        LoginExcedeuTentativasException,
        LoginExpiradoException,
        LoginInvalidoException {

        loginBusiness.validarUsuarioAtivo(usuarioModel);
        loginBusiness.validarTentativasdeLogin(usuarioModel);
        loginBusiness.validarUsuarioExpirado(usuarioModel);
        loginBusiness.validarLogin(usuarioModel);

    }

}
```

Código-fonte 6 – LoginFacade, abstração das regras de login
Fonte: Elaborado pelo autor (2021)

E por último, vamos adaptar nosso Servlet LoginController para o uso da nossa classe Facade. Veja abaixo:

```
package br.com.fiap.tds2.fase1.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import br.com.fiap.tds2.fase1.business.LoginFacade;
import br.com.fiap.tds2.fase1.exceptions.LoginAtivoException;
import br.com.fiap.tds2.fase1.exceptions.LoginExcedeuTentativasException;
import br.com.fiap.tds2.fase1.exceptions.LoginExpiradoException;
import br.com.fiap.tds2.fase1.exceptions.LoginInvalidoException;
import br.com.fiap.tds2.fase1.exceptions.LoginIpInvalidoException;
import br.com.fiap.tds2.fase1.model.UsuarioModel;

@SuppressWarnings("serial")
@WebServlet("/Login")
public class LoginController extends HttpServlet {

    // INSTANCIA DA CLASSE LOGGER
    private static final Logger LOGGER =
        LogManager.getLogger(LoginController.class.getName());
    private final LoginFacade loginFacade = new LoginFacade();

    public LoginController() {
        super();
    }

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        try {

            UsuarioModel usuarioModel = new UsuarioModel();
```

```
usuarioModel.setLogin(request.getParameter("login"));

usuarioModel.setSenha(request.getParameter("senha"));

loginFacade.loginWeb(usuarioModel);

RequestDispatcher despachar =
request.getRequestDispatcher("home.jsp");
despachar.forward(request, response);

    } catch ( LoginAtivoException |
LoginExcedeuTentativasException | LoginExpiradoException|
LoginInvalidoException | LoginIpInvalidoException e) {
        System.out.println("Erro ");
        response.sendRedirect("erro.jsp?msg=" +
e.getMessage());
    } catch (Exception e) {
        System.out.println("Ocorreu um erro muito
crítico, desconhecido pela aplicação");
        response.sendRedirect("erro.jsp?msg=Erro
crítico");
    }
}
```

Código-fonte 7 – LoginController, refatoração para o uso da classe Facade
Fonte: Elaborado pelo autor (2021)

Apenas de termos um exemplo simples, fica fácil de entender a utilidade de *pattern* Facade e de notar que, para qualquer alteração de regra de login, o ponto de alteração seria centralizado. Esse padrão é muito simples e de fácil uso, porém, se utilizado muitas vezes em cenários simples, pode gerar grandes pontos de códigos redundantes.

Segue o link da solução com a implementação do padrão Facade:
<https://drive.google.com/drive/folders/18DbRzgTzzhBdzmMqW-amLu0AuXFeGHW1?usp=sharing>

2.2 MVC e os *Design Patterns*

O padrão de arquitetura MVC pode e deve utilizar *Design Patterns* em suas camadas. Podemos dizer que os principais relacionamentos entre esses dois padrões são realizados com *Observer*, *Composite* e *Strategy*. Essa composição de padrões foi muito implementada em aplicações desktop antes da evolução das

aplicações web. Engana-se quem acha que esse padrão é obsoleto pela razão da migração de aplicativos desktop para aplicativos web. Esse padrão é a base dos *design patterns* usados em aplicações mobile.

Faremos um exemplo em Java; para tanto, vamos entender um pouco melhor as relações entre os padrões:

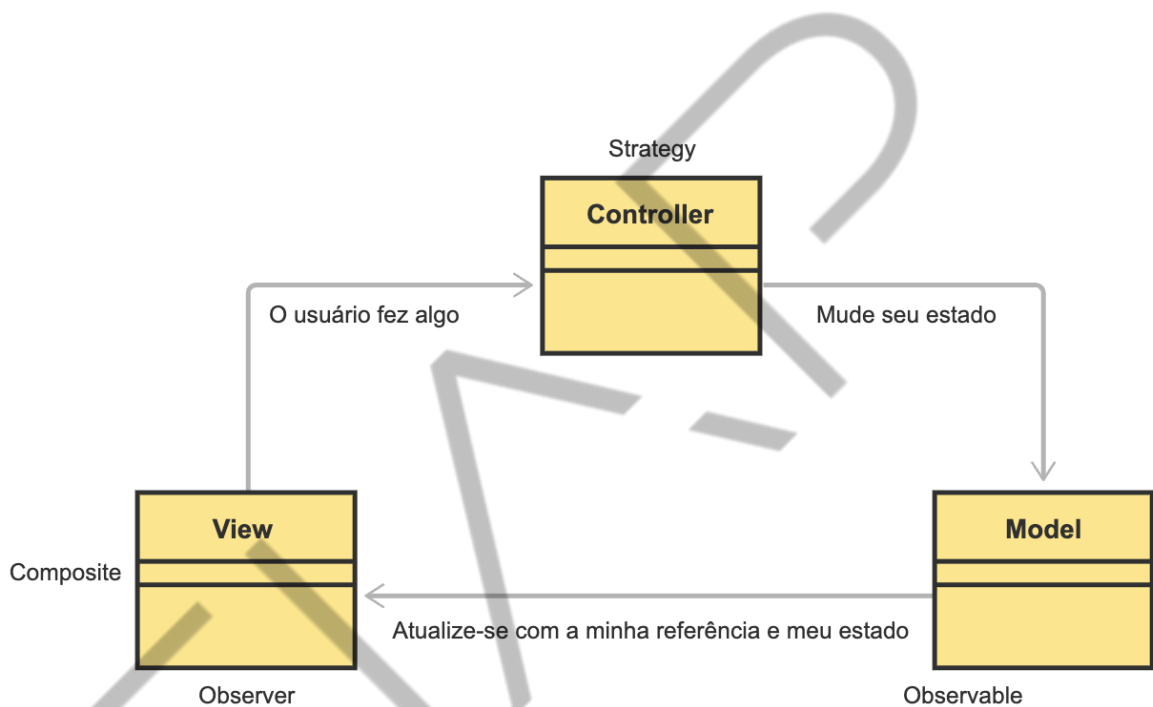


Figura 35 – Diagrama do Exemplo Java – MVC e *Design Patterns*
Fonte: Elaborado pelo autor (2017)

- A camada View, em nosso exemplo, implementará a interface Observer para obter o estado de um Observable (Model) e utilizará internamente o padrão Composite para gerenciar os componentes da tela.
- O Model será uma subclasse de Observable para que possa manter a camada View atualizada em relação às suas mudanças de estado, sem saber nada sobre ela em tempo de codificação.
- A relação entre as camadas View e Controller será um exemplo do padrão Strategy, já que a arquitetura MVC permite que você modifique a forma como uma View responde à entrada do usuário, utilizando controladores diferentes, sem alterar a apresentação visual dela. Para que isso ocorra, o

MVC encapsula cada mecanismo possível de resposta ao usuário em um objeto Controller diferente.

Sendo assim, o próprio *Controller* é uma estratégia, visto que um controlador é um objeto que encapsula um algoritmo. Objeto este que pode ser trocado facilmente por outro objeto controlador, caso se deseje um comportamento diferente da *View*.

Agora que entendemos um pouco melhor as relações entre os padrões, vamos ao nosso programa Java. O programa que faremos realizará o cálculo do imposto PIS (Programa de Integração Social) cumulativo sobre um valor informado na Tela. O exemplo que faremos tem o objetivo de se manter muito simples, focando apenas na utilização do MVC e de *Design Patterns*. Desconsidere quaisquer outras necessidades da aplicação, como validações, tratamento de exceções, menus etc.

O objetivo deste exemplo é criar uma forma simples de apresentar a separação de camadas proposta pelo MVC, com *Composite*, *Strategy* e *Observer*, e Java.

Vamos lá?

Crie um Java Project de nome PISMVC.

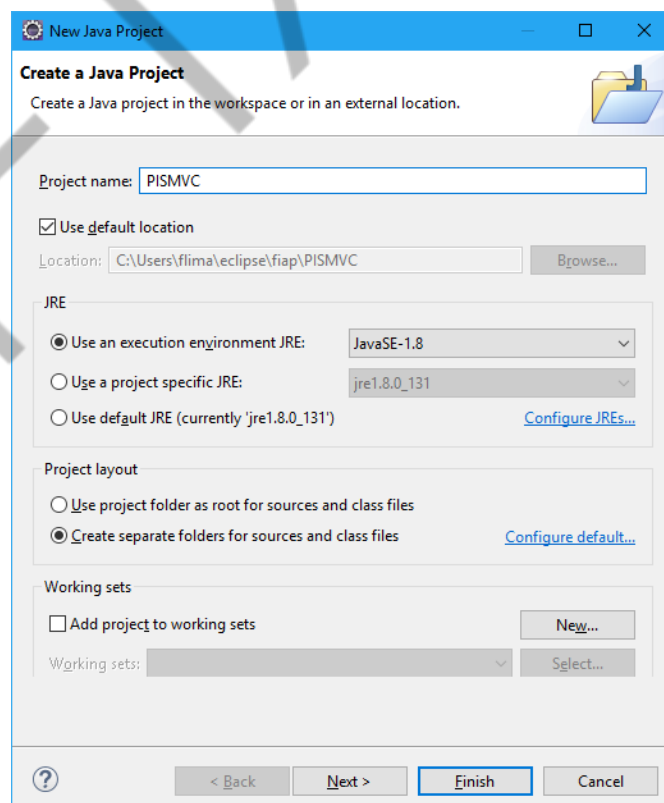


Figura 36 – Tela de criação do Java Project PISMVC
Fonte: Elaborado pelo autor (2017)

Vamos começar pela camada *Model*. Crie uma interface *Imposto* no pacote `br.com.fiap.imposto.model`, na pasta `src` do projeto PISMVC.

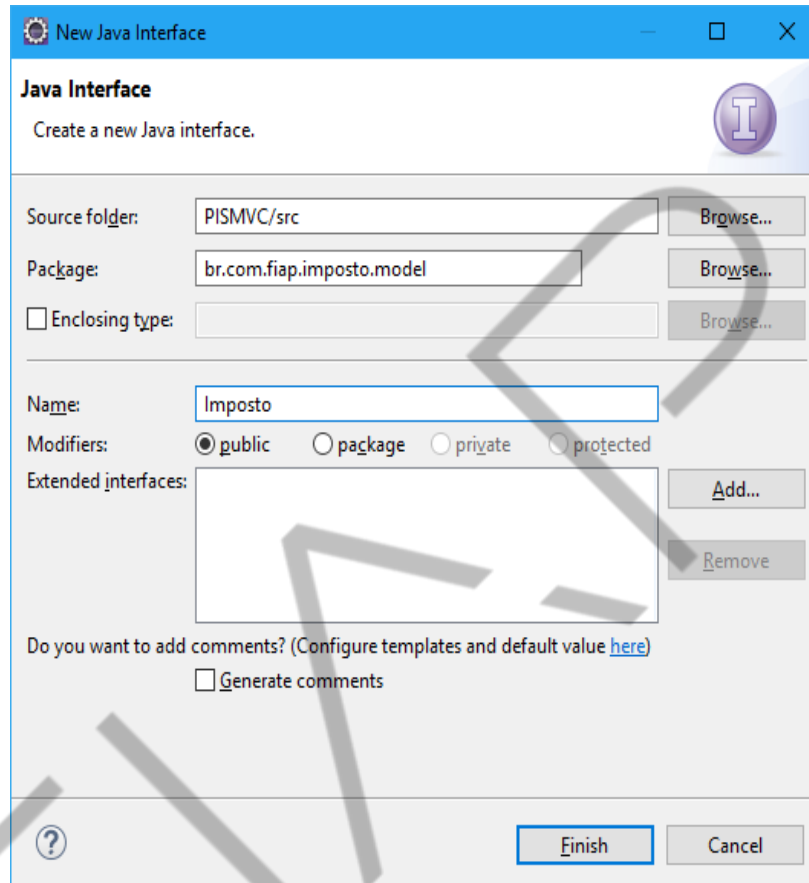


Figura 37 – Tela de criação da interface *Imposto* na camada *Model*
Fonte: Elaborado pelo autor (2017)

Implemente o código da Interface:

```
package br.com.fiap.imposto.model;

public interface Imposto {

    public void calcularImposto(float valor);

}
```

Código-fonte 8 – Interface *Imposto*
Fonte: Elaborado pelo autor (2017)

Crie uma classe *Pis* no pacote `br.com.fiap.imposto.model`.

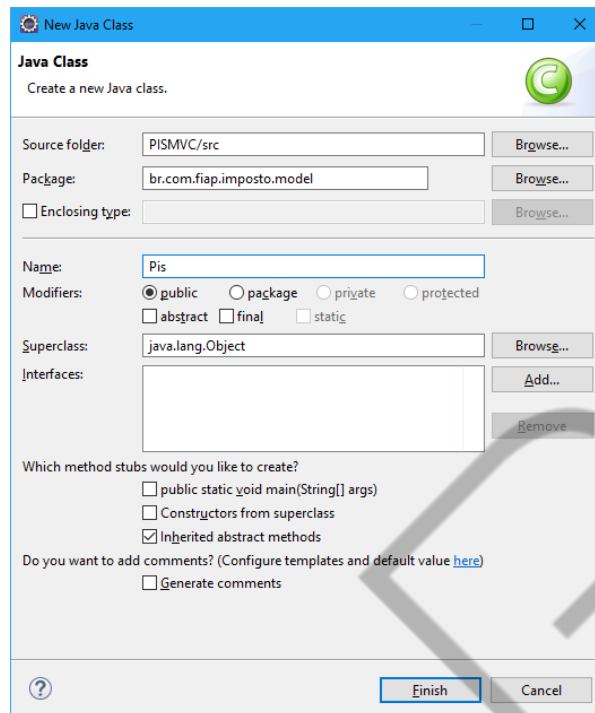


Figura 38 – Tela de criação da Classe Pis da camada *Model*
Fonte: Elaborado pelo autor (2017)

Implemente o código da camada *Model* com a Classe Pis:

```
package br.com.fiap.imposto.model;

import java.util.Observable;

// O Model é um Observable
// A Classe Pis da camada Model não possui referência a View ou ao //
Controller
// Ao implementar Imposto, teremos maior flexibilidade no Controller

public class Pis extends Observable implements Imposto{

    private final float ALIQUOTA = 0.65f;
    private float valorDoPis = 0;

    public Pis() {
        System.out.println("Construtor do Model chamado");
    }

    public float getAliquota() {
        return ALIQUOTA;
    }
}
```

```
public float getValorDoPis() {  
    return valorDoPis;  
}  
  
public void calcularImposto(float valor) {  
    valorDoPis = valor * ALIQUOTA;  
    // setChanged Altera o estado interno do objeto  
    // para true, pois houve alteração no estado valorDoPis  
    setChanged();  
    // Os observadores devem ser notificados  
    // Envia o valor do PIS como parte da mensagem de  
    // notificação para a View que é um Observer  
    notifyObservers(valorDoPis);  
}  
  
@Override  
public String toString() {  
    return "Pis [ALIQUOTA=" + ALIQUOTA  
        + ", valorDoPis=" + valorDoPis + "];"  
}  
}
```

Código-fonte 9 – Classe Pis, subclasse de *Observable*
Fonte: Elaborado pelo autor (2017)

No código da Classe Pis, o método `notifyObservers(valorDoPis)` enviará o estado do modelo alterado para a *View*, mas, antes disso, o método `setChanged()` marca que o objeto observado (Pis subclasse de *Observable*) foi alterado (mudou seu estado).

O método **`notifyObservers()`** verificará se o objeto foi alterado, ou seja, o atributo **`private boolean changed`** da superclasse **`Observable`** foi alterado para *true* pelo método **`setChanged()`**, para depois localizar todos os *Observers* da Classe Pis e executar o método **`update()`** de cada um deles. No nosso exemplo, teremos apenas um *Observer*, a Classe *CalculaPis* da camada *View* da nossa aplicação.

Dessa forma, o *Model* Pis é amplamente reutilizável, pois faz o cálculo do Pis e pode atender a mais de um observador, enviando o resultado sem ter nenhuma referência de qualquer observador em seu código.

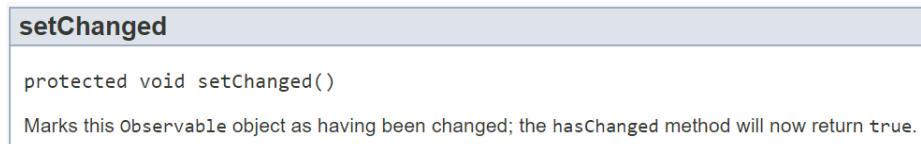


Figura 39 – setChanged Method
Fonte: Oracle (2017)

Agora, vamos implementar a camada *View*. Para tanto, crie a interface *TelaDeImposto* no pacote `br.com.fiap.imposto.view`.

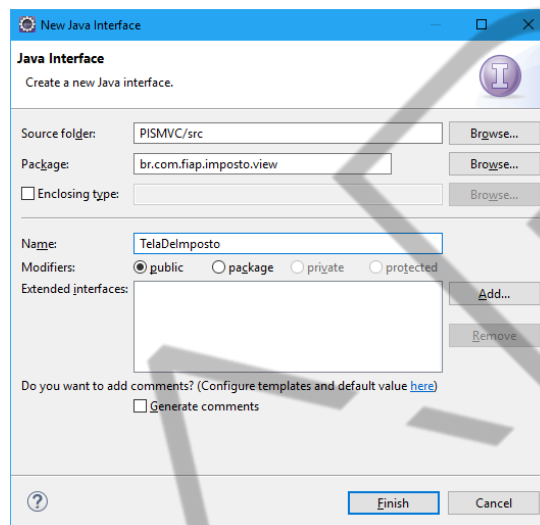


Figura 40 – Tela de criação da interface *TelaDeImposto* na camada *View*
Fonte: Elaborado pelo autor (2017)

Implemente o código da Interface:

```
package br.com.fiap.imposto.view;

public interface TelaDeImposto {

    public float getValor();

}
```

Código-fonte 10 – Interface *TelaDeImposto*
Fonte: Elaborado pelo autor (2017)

Crie a Classe *CalculaPis* também no pacote `br.com.fiap.imposto.view`.

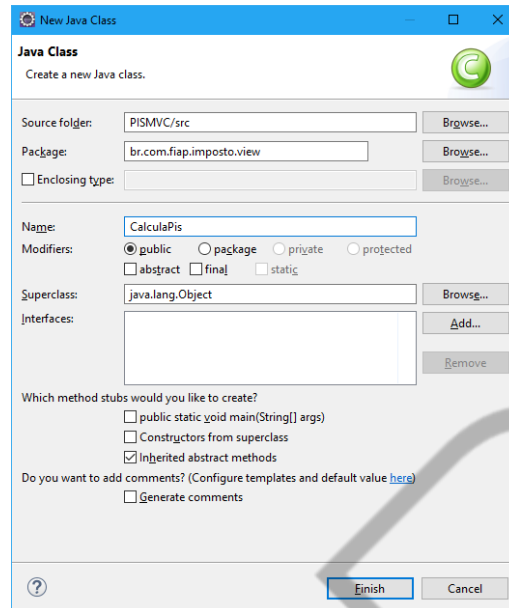


Figura 41 – Tela de criação da Classe CalculaPis da camada View
Fonte: Elaborado pelo autor (2017)

Implemente a tela da aplicação na camada View, com a Classe CalculaPis:

```
package br.com.fiap.imposto.view;

import java.awt.Button;
import java.awt.Frame;
import java.awt.Label;
import java.awt.Panel;
import java.awt.TextField;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Observable;
import java.util.Observer;
import javax.swing.JOptionPane;

public class CalculaPis implements Observer, TelaDeImposto {

    private TextField txtValorFaturado;
    private Button btnCalcular;

    // Construtor que compõe a tela
    public CalculaPis() {
        System.out.println("Construtor da View chamado");

        // O Frame é uma subclasse de Window que estende Container
```

```
// Essa hierarquia de classes utiliza o padrão Composite
// para compor a tela, veja os métodos .add
Frame frame =
    new Frame("Cálculo do PIS MVC e Design Patterns");
frame.add("North", new Label("Valor Faturado"));

txtValorFaturado = new TextField();
frame.add("Center", txtValorFaturado);

Panel panel = new Panel();
btnCalcular = new Button("Calcular PIS");
panel.add(btnCalcular);
frame.add("South", panel);

frame.addWindowListener(new CloseListener());
frame.setSize(200, 150);
frame.setLocation(100, 100);
frame.setVisible(true);
}

// Método que retorna o valor para cálculo de imposto
public float getValor() {
    return Float.parseFloat(txtValorFaturado.getText());
}

// Método que possibilita a View enviar a ação de calcular
// para o Controller chamar o Model
public void addController(ActionListener controller) {
    System.out.println("A View adicionou o Controller");
    btnCalcular.addActionListener(controller);
}

/* update exibe uma mensagem na View contendo:
 * A classe Model
 * O toString sobrescrito
 * O valor do estado, atributo valorDoPis da classe Pis
 */

// O método update será chamado pelo Model por notifyObservers()
public void update(Observable objModel, Object estadoDoModel)
{
    String msg = objModel.getClass()
```



```
        + "\n" + objModel.toString()
        + "\n" + ((Float)estadoDoModel).floatValue();

        JOptionPane.showMessageDialog(null, msg);
    }

    // Encerra o programa
    public static class CloseListener extends WindowAdapter {

        public void windowClosing(WindowEvent e) {
            e.getWindow().setVisible(false);
            System.exit(0);
        }
    }
}
```

Código-fonte 11 – Camada View, Classe CalculaPis implementa a interface Observer
Fonte: Elaborado pelo autor (2017)

No código da classe CalculaPis, o construtor da classe cria a tela da aplicação, utilizando as classes do *Abstract Window Toolkit* (AWT). A classe *Frame* desse pacote é uma subclasse de *Window* e, por sua vez, *Window* é uma subclasse de *Container*. *Panel* também é uma subclasse de *Container* e, nela, bem como no *Frame*, adicionamos componentes através do método `add()`. Esse é um uso clássico do *Design Pattern Composite*. Uma árvore de componentes, adicionados uns aos outros, utilizados como uma composição de componentes e grupos para formar a tela da nossa aplicação. Bem, e o padrão *Observer* na camada *View*?

A classe CalculaPis implementa a interface *Observer*. Portanto, é obrigada a codificar o método `update(Observable objModel, Object estadoDoModel)` e será por meio desse método que um objeto *Observable*, que CalculaPis, estiver observando, poderá executar um código na *View*. CalculaPis também implementa a interface *TelaDeImposto*. Sendo assim, deve implementar o método `getValor()`, possibilitando um *Controller* menos acoplado.

Resumindo, em nosso exemplo, o Observador é a *View* CalculaPis, o Observado é o *Model* Pis. Neste ponto da codificação, falta implementar o *Controller* e ligar as três camadas na nossa aplicação por meio de uma classe executável.

Vamos ao *Controller*? Crie a Classe `ImpostoController` no pacote `br.com.fiap.imposto.control`, conforme imagem abaixo:

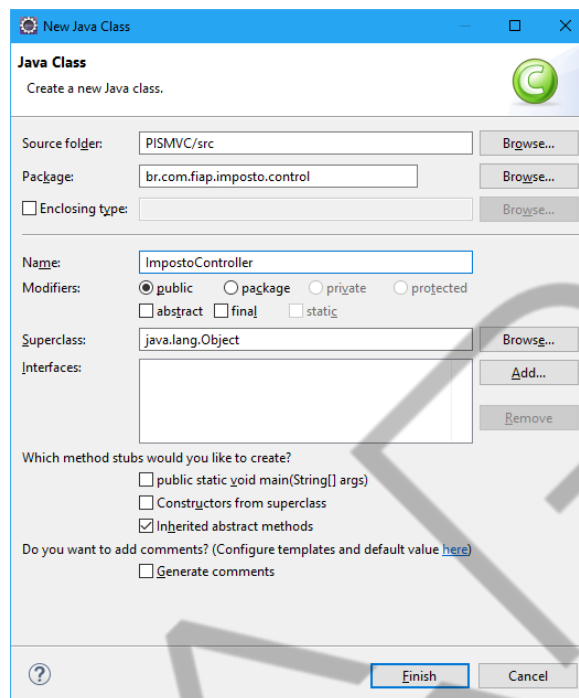


Figura 42 – Tela de criação da Classe `ImpostoController`, da camada *Controller*
Fonte: Elaborado pelo autor (2017)

Implemente o código da camada *Controller*, com a Classe `ImpostoController`:

```
package br.com.fiap.imposto.control;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import br.com.fiap.imposto.model.Imposto;
import br.com.fiap.imposto.view.TelaDeImposto;

// ImpostoController implementa ActionListener para capturar
// o evento do botão da View
public class ImpostoController implements ActionListener {

    // O Controller possui referências a Imposto e a Tela
    // É reutilizável para Telas que calculem Impostos
    // Ou seja, classes que implementem estas interfaces
    private Imposto model;
    private TelaDeImposto view;

    // O construtor recebe as referências das demais camadas
```

```
public ImpostoController(Imposto model, TelaDeImposto view) {  
    System.out.println("Construtor do Controller chamado");  
    this.model = model;  
    this.view = view;  
}  
  
// Método invocado quando o botão da View é invocado  
public void actionPerformed(ActionEvent e) {  
    model.calcularImposto(view.getValor());  
}  
}
```

Código-fonte 12 – Camada *Controller*, classe *ImpostoController* implementa a interface *ActionListener*
Fonte: Elaborado pelo autor (2017)

O nosso *Controller* implementa *ActionListener* para que possa capturar a ação do botão através do método *actionPerformed*. *ImpostoController* possui referências às interfaces *TelaDeImposto* e *Imposto*. Sendo assim, é capaz de controlar qualquer *View* que implemente *TelaDeImposto* e qualquer *Model* que implemente um *Imposto*.

Agora, precisamos de uma classe executável em nosso projeto para testar a aplicação.

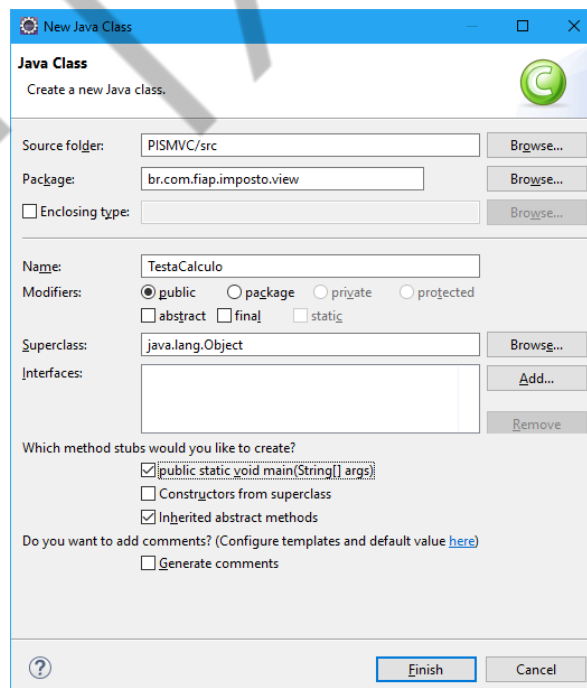


Figura 43 – Tela de criação da Classe *TestaCalculo*, da camada *View*
Fonte: Elaborado pelo autor (2017)

Implemente a Classe TestaCalculo, no pacote *View*:

```
package br.com.fiap.imposto.view;

import br.com.fiap.imposto.control.ImpostoController;
import br.com.fiap.imposto.model.Pis;

public class TestaCalculo {

    public static void main(String[] args) {

        Pis modelPis = new Pis();
        CalculaPis viewCalculaPis = new CalculaPis();
        // Adiciona um observador para o objeto observado Pis
        modelPis.addObserver(viewCalculaPis);
        // Instancia um Controller e informa quem ele controlará
        ImpostoController controller =
            new ImpostoController(modelPis, viewCalculaPis);
        // Envia o Controller criado para a View
        viewCalculaPis.addController(controller);
    }
}
```

Código-fonte 13 – Camada *Controller*, Classe *ImpostoController*
implementa a interface *ActionListener*
Fonte: Elaborado pelo autor (2017)

Ao executarmos a classe *TestaCalculo*, a tela para digitarmos o Valor faturado será exibida. Digitando o valor 1000 e pressionando o botão, o método *actionPerformed* de *ImpostoController* será acionado. Por sua vez, ele acionará o método *calcularImposto* da *Model* *Pis*, passando como parâmetro o valor digitado, obtido pelo método *getValor* da *View* *CalculaImposto*. O método *calcularImposto* da *Model* marca o *Pis* com o estado de alterado, através do método *setChanged* e atualiza a *View*, a partir do método *notifyObservers*. O método *update* da *View* recebe uma referência da *Model* e o valor calculado, exibindo, na sequência, uma mensagem com as informações recebidas.

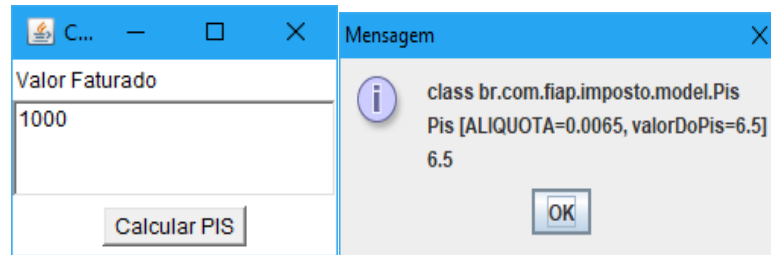


Figura 44 – Tela de cálculo do imposto e msg com o resultado
Fonte: Elaborado pelo autor (2017)

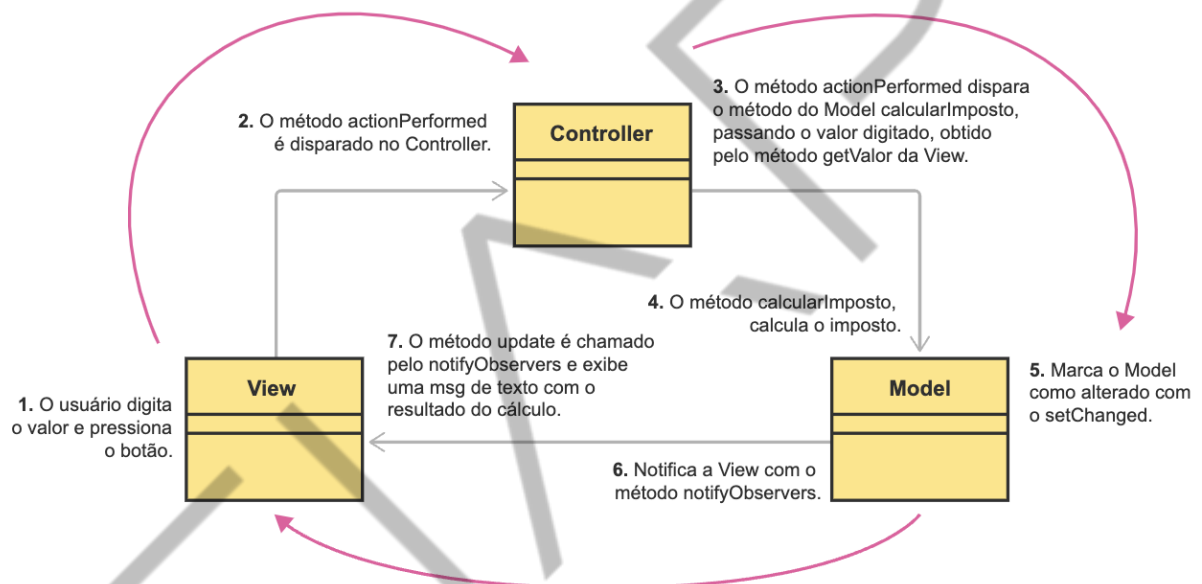


Figura 45 – Fluxo resumido da aplicação
Fonte: Elaborado pelo autor (2017)

Como já comentamos, esta aplicação foi codificada de forma muito simples, apenas para apresentar o MVC em conjunto com os *Design Patterns* mais utilizados em suas camadas. Independentemente disso, há um ponto da aplicação que ficou muito ruim. O valor da Alíquota do PIS está *hard coded*, ou seja, está com o valor fixo, cravado no código-fonte.

Se quisermos alterar tal valor, somente um programador poderá fazê-lo. Não seria legal implementarmos uma solução flexível, envolvendo outro padrão? Então, vamos codificar uma classe *singleton*, para ser um ponto único que buscará o valor da alíquota em um arquivo de propriedades para toda a aplicação.

Crie a classe AliquotaSingleton no pacote útil e um novo Folder de nome resources no src, conforme imagem abaixo. Para ambos, no Eclipse, botão direito New > Class e New > Folder. Em Resources, novamente botão direito, New > File para você criar o arquivo conf.properties.

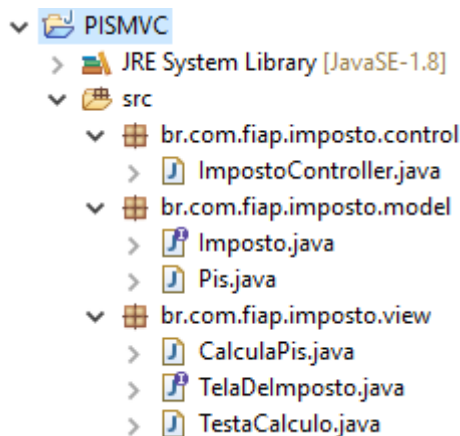


Figura 46 – Estrutura atual do Projeto
Fonte: Elaborado pelo autor (2017)

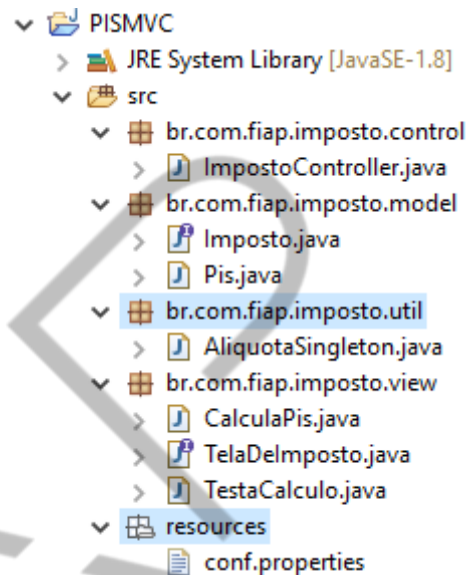


Figura 47 – Estrutura com Singleton
Fonte: Elaborado pelo autor (2017)

Implemente a Classe AliquotaSingleton no pacote util:

```
package br.com.fiap.imposto.util;

import java.io.IOException;
import java.util.Properties;

public class AliquotaSingleton {

    // Atributo privado static pertence à classe e não ao objeto, será
    // compartilhado por todos os objetos que o acessarem
    private static Properties properties;

    private static final String ARQ = "/resources/conf.properties";
    // O construtor da classe é privado
    // outros objetos não podem instanciar AliquotaSingleton
    private AliquotaSingleton() {}
```

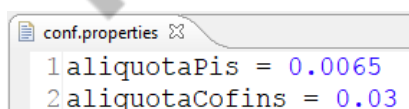
```
// A única maneira de obter um objeto Properties
// é através do método público e estático getInstance()
// que sempre retornará uma única instância dessa classe

public static Properties getInstance() {

    // Se properties é nulo, instancia um para retornar
    // desta forma, instanciamos apenas um properties
    if(properties == null) {
        properties = new Properties();
        try{
            properties.load(AliquotaSingleton
                .class.getResourceAsStream("ARQ"));
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
    return properties;
}
}
```

Código-fonte 14 – Classe AliquotaSingleton implementa o padrão Singleton
Fonte: Elaborado pelo autor (2017)

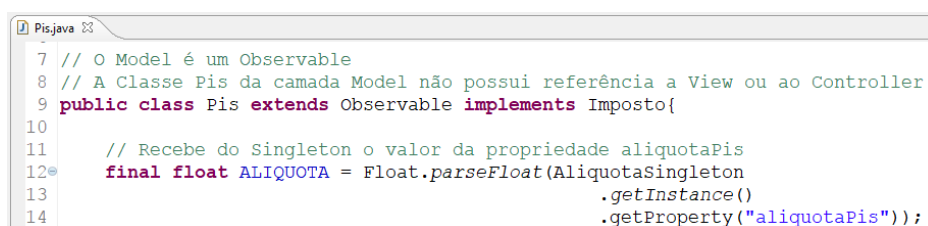
No arquivo conf.properties, crie as seguintes propriedades:



```
conf.properties
1aliquotaPis = 0.0065
2aliquotaCofins = 0.03
```

Figura 48 – Propriedades alíquota Pis e Confins, definidas em conf.properties no Eclipse
Fonte: Elaborado pelo autor (2017)

Agora, vamos tirar a alíquota *hard coded* no *Model*. Para tanto, altere a classe Pis da seguinte forma:



```
Pis.java
7 // O Model é um Observable
8 // A Classe Pis da camada Model não possui referência a View ou ao Controller
9 public class Pis extends Observable implements Imposto{
10
11     // Recebe do Singleton o valor da propriedade aliquotaPis
12     final float ALIQUOTA = Float.parseFloat(AliquotaSingleton
13         .getInstance()
14         .getProperty("aliquotaPis"));
```

Figura 49 – Atributo ALIQUOTA da Classe Pis, recebendo o valor do Singleton
Fonte: Elaborado pelo autor (2017)

Feita a alteração, execute o programa novamente e você verá que tudo funcionará, mas com a flexibilidade de ter o valor da alíquota parametrizado no arquivo `conf.properties`.

Arquivo este que pode ser alterado pelo usuário para mudar a alíquota sem a necessidade de alterar códigos e recompilar a aplicação. O `conf.properties` é acessado em toda a aplicação, apenas pela classe Singleton `AliquotaSingleton`, para garantir o uso da alíquota certa, obtida em um ponto único. A propriedade `alíquota Cofins` foi criada apenas para mostrar a versatilidade desse tipo de parametrização.

2.3 Frameworks

Existem relatos que apontam os primeiros anos da década de 1980 como o início da utilização de *frameworks*, com o crescimento do uso da linguagem *Smalltalk*. Nesse período, bibliotecas de classes que resolviam determinados domínios começaram a ser agrupadas para compor *frameworks*.

Um *framework* foi definido por Coad (1992) como um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas. Antes disso, Johnson e Foote (1988) definiram *framework* como um conjunto de classes abstratas e concretas que provê uma infraestrutura genérica de soluções para um conjunto de problemas.

Segundo Wolfgang (1994), um *framework* é uma coleção de classes abstratas e concretas e a interface entre elas, representando o projeto de um subsistema.

Na óptica de Johnson (1997), um *framework* é um esqueleto de uma aplicação que pode ser instanciado por um desenvolvedor de aplicações. Trata-se de uma aplicação parcialmente completa, reutilizável que, quando especializada, produz aplicações personalizadas (JOHNSON; FOOTE, 1988).

Podemos dizer que *framework* é uma estrutura de código desenvolvida com um propósito. Um programa incompleto que servirá de base para diversos outros programas. Isso significa que o *framework* deve ser geral o suficiente para abranger diversos programas, mas também específico o suficiente para atender às necessidades de cada um.

Benefícios do uso de um *framework* no desenvolvimento de *software*:

- Redução do esforço de programação = maior produtividade.
- Interoperabilidade.
- Redução do esforço de aprendizado.
- Redução do esforço de projetar, implementar e testar.
- Aumento do reúso de software.
- Estar de acordo com as boas práticas de mercado, entre outros.

Para exemplificarmos o uso de *frameworks*, vamos pensar em uma necessidade muito comum em soluções corporativas e escolher um *framework* de mercado para resolvê-la. Uma situação muito comum é precisarmos logar as ações realizadas em uma aplicação. Imagine que temos a necessidade de saber detalhes sobre todos os cálculos de Pis que são realizados na nossa aplicação. Detalhes como Valor Faturado, Pis Calculado, Alíquota utilizada, data e hora do cálculo.

Pense comigo: será que vale a pena desenvolver uma solução, uma aplicação à parte, somente para tratar logs? Vamos ter o esforço de projetar, implementar e testar a solução de log, dentro de um projeto que atende ao negócio calcular imposto? Não parece bom, concorda?

2.4 Padrões arquiteturais de microsserviços

Nos anos mais recentes, em especial a partir de 2018, a arquitetura de microsserviços vem ganhando cada vez mais força e popularidade por influência de cases de sucesso de empresas, como Amazon, Netflix, Spotify, entre muitas outras.

Tida como uma evolução de SOA e MVC, a arquitetura de microsserviços estrutura a aplicação como uma série de serviços desacoplados que podem ser deployados independentemente.

Como principais vantagens dessa abordagem, é possível estruturar times multidisciplinares pequenos e ágeis, responsáveis por contextos de produtos, escalabilidade horizontal, maior disponibilidade e resiliência.

Grandes grupos de desenvolvedores trabalhando em uma mesma base de código podem ser caóticos e contraproducentes. Com microsserviços, uma organização consegue dispor de diversas pequenas equipes independentes, cada uma com sua base de código, o que acelera o processo de desenvolvimento e a frequência que deploys são realizados.

A arquitetura de microsserviços, entretanto, traz uma série de outros desafios técnicos, que não existiam nas aplicações monolíticas. Cada microsserviço possui seu próprio banco de dados. Como manter consistência de dados entre diferentes bancos que não devem ter acoplamento ou relação entre eles? Caso um serviço falhe, como fazer para que os outros serviços que dependem dele parem de encaminhar requisições que não poderão ser processadas? Esses e outros problemas são resolvidos com padrões arquiteturais de microsserviços.

A seguir falaremos de alguns desses padrões:

Saga

Considerando que uma transação pode se estender por inúmeros serviços diferentes, caso um problema ocorra em um dos serviços, como assegurar de que os outros serviços que já processaram a transação façam os devidos ajustes nos seus bancos de dados?

Por exemplo, suponha que uma loja virtual utilize 3 microsserviços para fazer o *checkout* de uma compra:

Serviço 1: Responsável por dar a baixa no estoque.

Serviço 2: Integrado com a transportadora que fará a entrega do cliente.

Serviço 3: Faz a cobrança no cartão de crédito do cliente.

No nosso exemplo, supondo que os Serviços 1 e 2 foram executados com sucesso, mas ao tentar realizar a cobrança no cartão não havia limite disponível para realizar a compra. O serviço 2 precisa cancelar a entrega e o serviço 1 precisa reintegrar o produto ao estoque.

O padrão Saga vem para resolver essas questões. Existem duas abordagens:

1) Coreografada – Cada serviço conhece o próximo a ser executado e, em casa de falha, “avisa” o serviço passado de que um ajuste precisa ser executado. A

forma de comunicação entre os serviços costuma ser assíncrona, utilizando filas de um message broker, como RabbitMQ, Kafka, entre outros, ou Event Sourcing, que veremos a seguir.

2) Orquestrada – Existe um serviço orquestrador que possui conhecimento de todos os passos que precisam ser executados. Em caso de uma falha, cabe ao orquestrador encaminhar requisições para todos os demais serviços fazerem os devidos ajustes a seus bancos de dados.

Event Sourcing

Esse padrão trata da comunicação de fatos relevantes ocorridos em um serviço, que podem ser consumidos por outros microsserviços.

Um evento pode ser, por exemplo, um novo usuário que acaba de fazer registro em um site. O serviço de registro cria o usuário no seu banco de dados e emite um evento com os dados relevantes do novo usuário. Outros serviços, como, por exemplo, o serviço que envia um e-mail de boas-vindas para o cliente pode consumir esse evento e realizar sua tarefa.

Plataformas de streaming, como Apache Kafka e Nats, costumam ser amplamente utilizadas para construir soluções de Event Sourcing nas organizações.

CQRS

Uma característica importante da arquitetura de microsserviços é que cada serviço deve ter um Banco de Dados próprio. Dois microsserviços não devem compartilhar um mesmo banco.

Podemos nos ver em uma situação onde muitas requisições de escrita ocorrem em um banco. Isso pode levar a uma demora na resposta de consultas nesse banco, levando à degradação do microsserviço.

Para resolver essa situação – sem solução em uma aplicação monolítica – utiliza-se o padrão CQRS (Command Query Responsibility Segregation, ou Segregação das Responsabilidades de Comandos e Consultas).

Uma command, como vimos anteriormente no capítulo dos *Design Patterns* GoF, é uma ação que geralmente faz uma mudança no estado do serviço, ou seja, no fundo, se traduz com uma inserção ou atualização no banco de dados.

Esse padrão separa em diferentes serviços – e, por consequência, diferentes bancos de dados – as responsabilidades de escrita e leitura das informações.

O desafio, nesse caso, é manter ambos os bancos consistentes entre si. Para isso, pode ser utilizada autorreplicação do banco de dados (se suportado pelo SGBD em questão) ou até mesmo o padrão Event Sourcing. Nele, cada escrita gera um evento, consumido pelo serviço de leitura, e faz os ajustes em seu banco.

Circuit Breaker

Software pode falhar. Com microsserviços, não é diferente. Um serviço pode se tornar indisponível por uma infinidade de razões. Em um caso como esse, os outros serviços que dependem dele para realizar algumas ações não conseguirão completar suas requisições e podem continuar bombardeando o serviço indisponível, agravando ainda mais a situação de indisponibilidade.

Circuit breaker é o padrão utilizado para resolver esses problemas. Ao ter uma certa quantidade de requisições negadas por um serviço do qual depende, o microsserviço identifica que aquele serviço-alvo está indisponível e para temporariamente de aceitar requisições que o fariam contatar o serviço que está passando por indisponibilidade.

Ao se recuperar, o serviço-alvo emite um evento, que é consumido pelos serviços que dependem dele. A partir desse momento, os serviços voltam a aceitar requisições normalmente.

O Hystrix, da Netflix, é um exemplo de biblioteca *open source* usada amplamente para a implementação de *circuit breakers*.

Esses são alguns dos principais padrões arquiteturais de microsserviços. O site <https://microservices.io/> possui uma lista abrangente deles e muitos outros padrões que endereçam as principais questões que essa nova abordagem nos trouxe.

CONSIDERAÇÕES FINAIS

O objetivo deste conteúdo foi abordar o básico e essencial sobre os padrões MVC e GoF, além dos conceitos de *framework*. Desenvolvemos um exemplo de

projeto em Java que empregou o padrão de arquitetura MVC e os padrões GoF: Composite, Strategy, Observer e Singleton. Na sequência, evoluímos a aplicação, utilizando um *framework* Java para *Logging*.

Lembro que, além dos padrões MVC e GoF, existem diversos outros padrões, tais como Architectural Patterns, GRASP Patterns, Concurrency Patterns, JEE Patterns, entre outros. Da mesma forma, podemos afirmar que existem diversos *frameworks* reconhecidos pelo mercado, implementados com os mais diversos tipos de propósito.

Existem inúmeros conteúdos que são abordados nos livros e URLs de referência deste capítulo. Para mais informações, recomendamos a leitura.

REFERÊNCIAS

- ALEXANDER, C. **Notes on the synthesis of form**. Nova York. Harvard University Press, 1964.
- ALEXANDER, C. **The timeless way of building**. Nova York. Oxford University Press, 1979.
- ALEXANDER, C. **A pattern language**. Nova York. Gustavo Gili, 1977.
- ALUR, Deepak et al. **Core J2EE Patterns (Core Design Series)**: best practices and design strategies. Mountain View, CA, USA. Sun Microsystems, Inc., 2003.
- APACHE. **Welcome to Log4j 2! - Introduction**. [s.d.]. Disponível em: <<https://logging.apache.org/log4j/2.0/manual/index.html>>. Acesso em: 15 dez. 2020.
- APACHE. **Download Apache Log4j 2**. [s.d.]. Disponível em: <<https://logging.apache.org/log4j/2.0/download.html>>. Acesso em: 15 dez. 2020.
- APACHE. **Articles and tutorials**. [s.d.]. Disponível em: <<https://logging.apache.org/log4j/2.0/articles.html>>. Acesso em: 15 dez. 2020.
- BORAJI. **Log4j 2 – RollingFileAppender example**. 26 jul. 2017. Disponível em: <<https://www.boraji.com/log4j-2-rollingfileappender-example>>. Acesso em: 15 dez. 2020.
- BOWMAN, M; BRIAND, L. C.; LABICHE, Y. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. **IEEE Transactions on Software Engineering**, v. 36, n. 6, p. 817-837, 2010.
- COAD, P. Object-oriented patterns. **Communications of the ACM**, v. 35, n. 9, p. 152-159, 1992.
- FADERMAN, A.; KOLETZKE, P.; DORSEY, P. **Oracle JDeveloper 10g Handbook**. Nova York. McGraw-Hill, Inc., 2004.
- GAMMA, Erich et al. **Design Patterns**: elements of reusable object-oriented languages and systems. Boston. Cover Art, 1994.
- HORSTMANN, C. S.; CORNELL, G. **Core Java 2**: Volume I, Fundamentals. Londres. Pearson Education, 2002.
- JOHNSON, R. E. Frameworks = (Components + Patterns). **Communications of the ACM**, v. 40, n. 10, p. 39-42, 1997.
- JOHNSON, R. E; FOOTE, B. Designing reusable classes. **Journal of object-oriented programming**, v. 1, n. 2, p. 22-35, 1988.
- LARMAN, C. **Applying UML and Patterns**: an introduction to object oriented analysis and design and iterative development. Índia: Pearson Education, 2012.

METSKER, S. J. **The Design Patterns Java Workbook**. Londres. Addison-Wesley Longman Publishing Co., Inc., 2002.

ORACLE. **Introduction to the Oracle Application Development Framework**. [s.d.]. Disponível em: <http://oln.oracle.com/static/J14341GC10/lesson_1.htm>. Acesso em: 15 dez. 2020.

ORACLE. **About the Model 2 Versus Model 1 Architecture**. [s.d.]. Disponível em: <http://download.oracle.com/otn_hosted_doc/jdeveloper/1012/developing_mvc_applications/adf_aboutmvc2.html>. Acesso em: 15 dez. 2020.

REENSKAUG, T. Models-views-controllers. **Technical note, Xerox PARC**, v. 32, n. 55, p. 6.2, 1979.

RICHARDSON, C. **Microservices Patterns**. Nova York. Manning Publications, 2019.

SAP. **UI Development toolkit for HTML5 (SAPUI5)**. Disponível em: <https://help.sap.com/saphelp_uiaddon10/helpdata/en/95/d113be50ae40d5b0b562b84d715227/frameset.htm>. Acesso em: 15 dez. 2020.

VLISSIDES, John et al. Design Patterns: elements of reusable object-oriented software. Reading, Massachusetts: Addison-Wesley, v. 49, n. 120, p. 11, 1995.

WOLFGANG, Pree. Design patterns for object-oriented software development. Reading, Massachusetts: Addison-Wesley, 1994.

GLOSSÁRIO

<i>Design Pattern</i>	Solução consolidada e reconhecida pela comunidade de desenvolvedores, como uma forma eficiente de se resolver um problema recorrente em projetos e manutenções de <i>software</i> .
MVC	Padrão de arquitetura que orientou a equipe a separar a aplicação Web em três camadas.
Abstração	Habilidade de se concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais.
GoF	Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides são conhecidos como Gang of Four (GoF).
<i>Architectural Patterns</i>	Solução geral, reconhecida pela comunidade de <i>software</i> , documentada e reutilizável, que resolve um problema recorrente na arquitetura de <i>software</i> dentro de um determinado contexto.
GRASP Patterns	<i>General Responsibility Assignment Software Patterns</i> ou GRASP é um conjunto de diretrizes para atribuir responsabilidade a classes e objetos em projeto orientado a objetos.
<i>Concurrency Patterns</i>	Os padrões de concorrência ou <i>Concurrency Patterns</i> são padrões de <i>design</i> que lidam com o paradigma de programação <i>multithreaded</i> ou processamento paralelo.
JEE Patterns	Padrões de projeto para aplicações Java Platform, Enterprise Edition. São padrões mais complexos, utilizados para resolver necessidades de maior escala em ambientes geralmente distribuídos.

SmallTalk	Linguagem de programação orientada a objeto.
C++	Linguagem de programação compilada multiparadigma.
UI	Interface do usuário na qual ocorre a interação homem-computador.
JSP Java Server Pages	É uma tecnologia Java, atualmente mantida pela Oracle, criada para os desenvolvedores de <i>software</i> programarem páginas web.
JavaBeans	Segundo a especificação da Sun Microsystems, os JavaBeans são: “Componentes reutilizáveis de <i>software</i> que podem ser manipulados visualmente com a ajuda de uma ferramenta de desenvolvimento.”
EJBs Enterprise JavaBeans	Componente da plataforma JEE (Java Platform, Enterprise Edition) executado em um container de um servidor de aplicação web.
SAP	É uma das maiores empresas do mundo no setor de <i>software</i> empresarial. “A SAP provê soluções que ajudam as organizações a enfrentarem a complexidade do negócio e conseguirem gerar novas oportunidades de inovação e crescimento para se manterem à frente da concorrência.”
SAPUI5	Conjunto de bibliotecas JavaScript que desenvolvedores podem usar para criar aplicativos responsivos que rodam em diversos browsers.
PIS	O Programa de Integração Social – PIS, em nosso contexto, são contribuições que passaram a financiar o programa de seguro-desemprego e o abono de um salário mínimo anual aos empregados que recebam até dois salários mínimos mensais de empregadores contribuintes do programa.
COFINS	Contribuição para Financiamento da Seguridade Social, instituída pela Lei Complementar n. 70/1991.

AWT Abstract Window Toolkit	É <i>toolkit</i> gráfico original da linguagem de programação Java.
ActionListener	Interface ouvinte da plataforma Java para receber e tratar eventos de ação.
Framework	Entre outras definições, um <i>framework</i> é um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas.
Log4j	<i>Framework Open Source</i> desenvolvido pela Apache Software Foundation para realizar log de dados em uma aplicação.
Apache	Apache Software Foundation (ASF) é uma fundação que tem como missão fornecer <i>software</i> para o bem público. São disponibilizados pela instituição serviços e suporte para muitas comunidades de projetos de <i>software</i> de mentalidade semelhante de indivíduos que optam por aderir ao ASF.
JAR (Java ARchive) ou (.jar)	Arquivo compactado usado para distribuir um conjunto de classes Java, um aplicativo Java ou outros itens, como artefatos, XMLs etc.
JDBC Java Database Connectivity	Conjunto de classes e interfaces (API) escritas em Java para se trabalhar com banco de dados relacionais.
JPA Java Persistence API	É uma API Java que descreve uma interface comum para <i>frameworks</i> de persistência de dados.
Cassandra	Projeto da Apache de SGDB de segunda geração, distribuído e altamente escalável.
SMTP Simple Mail Transfer Protocol	Protocolo-padrão para envio de e-mails pela Internet.

EMAP