



FACULDADE PROFESSOR MIGUEL ÂNGELO DA SILVA SANTOS – FeMASS
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Padrões de projeto implementados com Java e Python: aspectos de verbosidade

POR:
RODRIGO GUILHERME CRUZ DAMASCENO

MACAÉ
2021

FACULDADE PROFESSOR MIGUEL ÂNGELO DA SILVA SANTOS – FeMASS
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Rodrigo Guilherme Cruz Damasceno

Padrões de projeto implementados com Java e Python: aspectos de verbosidade

Trabalho Final apresentado ao curso de graduação em
Sistemas de Informação, da Faculdade Professor
Miguel Ângelo da Silva Santos (FeMASS), para
obtenção do grau de BACHAREL em Sistemas de
Informação.

Professor Orientador: Isac Mendes Lacerda, M.Sc.

MACAÉ/RJ

2021

RODRIGO GUILHERME CRUZ DAMASCENO

Padrões de projeto implementados com Java e Python: aspectos de verbosidade

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Sistemas de Informação, da Faculdade Professor Miguel Ângelo da Silva Santos (FeMASS), para obtenção do grau de BACHAREL em Sistemas de Informação.

Aprovada em 01 de julho de 2021

BANCA EXAMINADORA

Isac Mendes Lacerda, M.Sc.

Faculdade Professor Miguel Ângelo da Silva Santos (FeMASS)

1º Examinador

Alan Carvalho Galante, D.Sc.

Faculdade Professor Miguel Ângelo da Silva Santos (FeMASS)

2º Examinador

ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO II

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Data da defesa: 01/07/2021

Aluno (a): RODRIGO GUILHERME CRUZ DAMASCENO

Orientador (a): ISAC MENDES LACERDA

Banca Examinadora:

Presidente (Orientador/a) ISAC MENDES LACERDA

Examinador/a (1) ALAN CARVALHO GALANTE

Título do trabalho: PADRÕES DE PROJETO IMPLEMENTADOS COM JAVA E PYTHON: ASPECTOS DE VERBOSIDADE

Local: GoogleMeet Sala: Virtual

Hora de início: 20:10 Hora do término: 20:40

Em sessão pública, após exposição de 30 minutos, o (a) aluno (a) foi arguido (a) oralmente pelos membros da banca, tendo como resultado para seu Trabalho de Conclusão de Curso II:
(X) APROVADO/A () REPROVADO/A

Nota obtida: 8,00

Na forma regulamentar foi lavrada a presente ata que é abaixo assinada pelos membros da banca, na ordem acima determinada, e pelo aluno:

Macaé, 01 de JULHO de 2021.

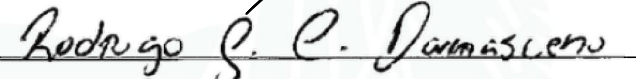
Presidente:



Examinador/a



Aluno (a):



AGRADECIMENTO

A Deus em primeiro lugar, por me dar forças para ultrapassar todos os obstáculos durante a caminhada ao longo do curso. Aos meus pais e irmão, que sempre me incentivaram e acreditaram em mim e estiveram sempre comigo nos bons e maus momentos. A minha amada noiva que está comigo todos os dias, me incentivando e me dando forças. A toda a minha família que, mesmo de longe, sempre depositam toda a confiança em mim. A todos os professores, especialmente ao meu orientador, pelas orientações, correções e ensinamentos para poder chegar até o final deste curso.

EPÍGRAFE

“Os dias prósperos não vêm por acaso; nascem de
muita fadiga e persistência.”
(Henry Ford)

RESUMO

Este trabalho discorre sobre a importância de utilizar padrões de projeto no desenvolvimento de softwares, bem como apresenta detalhadamente seis padrões de projeto altamente utilizados na produção de software (*Factory Method*, *Singleton*, *Adapter*, *Facade*, *State* e *Observer*). Além disso, apresenta os princípios de programação SOLID e discorre sobre a sua utilização.

A pesquisa permeia com a implementação dos seis padrões de projetos em duas linguagens diferentes: Java e Python. A implementação em duas linguagens, que estão entre as linguagens de alto nível mais populares da indústria de software atualmente, tem como propósito comparar aspectos de verbosidade, tomando os seis padrões como referência.

No trabalho também é realizado um estudo de caso sobre a utilização dos padrões de projeto nas linguagens apresentadas, destacando suas diferenças e principais características.

Palavras-chave: Padrões de Projeto; Java; Python; Linguagens.

.

ABSTRACT

This work discusses the importance of using design patterns in software development, as well as presents in detail six design patterns that are highly used in software production (Factory Method, Singleton, Adapt, Facade, State and Observer). In addition, it introduces SOLID programming principles and discusses their use.

The research permeates the implementation of the six design patterns in two different languages: Java and Python. The implementation in two languages, which are among the most popular high-level languages in the software industry today, is intended to compare verbosity, taking the six standards as a reference.

In the work, a case study is also carried out on the use of design patterns in important languages, highlighting their differences and main characteristics.

Keywords: Design Patterns; Java; Python; languages.

LISTA DE FIGURAS

Figura 1: DIAGRAMA DE CLASSE DO PRINCÍPIO SIGLE RESPONSABILITY.....	18
Figura 2: DIAGRAMA DE CLASSE OPEN/CLOSED PRINCIPLE.....	19
Figura 3: DIAGRAMA DE CLASSE OPEN/CLOSED PRINCIPLE.....	19
Figura 4: DIAGRAMA DE CLASSE LISKOV SUBSTITUTION PRINCIPLE.....	20
Figura 5: INTERFACE SEGREGATION PRINCIPLE.....	21
Figura 6: DIAGRAMA DE CLASSE DO PRINCÍPIO DEPENDENCY INVERSION PRINCIPLE.	21
Figura 7: DIAGRAMA DE CLASSE DO PRINCÍPIO DEPENDENCY INVERSION PRINCIPLE.	22
Figura 8: DIAGRAMA DE CLASSE DO PADRÃO FACTORY METHOD.	24
Figura 9: DIAGRAMA DE CLASSE DO PADRÃO SINGLETON.	25
Figura 10: DIAGRAMA DE CLASSE ADAPTER.	26
Figura 11: DIAGRAMA DE CLASSE PADRÃO FACADE.	27
Figura 12: DIAGRAMA DE CLASSE DO PADRÃO STATE	28
Figura 13: DIAGRAMA DE CLASSE DO PADRÃO OBSERVER.....	29
Figura 14: REPRESENTAÇÃO DA INTERFACE "SERVICO" NA LINGUAGEM JAVA.	30
Figura 15: REPRESENTAÇÃO EM JAVA DA CLASSE "RUBBER".	31
Figura 16: REPRESENTAÇÃO EM JAVA DA CLASSE "RIFOOD".	31
Figura 17: CLASSE FACTORY REPRESENTADA EM JAVA.	32
Figura 18: CLASSE CLEINTE CONTENDO O MÉTODO MAIN.....	32
Figura 19: EXEMPLO DO CÓDIGO FACTORY METHOD EM PYTHON.	33
Figura 20: PADRÃO SINGLETON DEMONSTRADO NA LINGUAGEM JAVA.....	34
Figura 21: PADRÃO SINGLETON DEMONSTRADO EM PYTHON.	35
Figura 22: CLASSE HDMI DEMONSTRADA EM JAVA.....	35
Figura 23: CLASSE VGA DEMONSTRADA EM JAVA.....	36
Figura 24: CLASSE ADAPTADOR HDMI EM JAVA.....	36
Figura 25: CLASSE CLIENTE EM JAVA.	37
Figura 26: CÓDIGO COMPLETO DO PADRÃO ADAPTER EM PYTHON.	38
Figura 27: CLASSE LOJA EM JAVA.	39
Figura 28: CLASSE RECEBEPEDIDO EM JAVA.	40
Figura 29: CLASSE CONFIRMAPAGAMENTO EM JAVA.....	40
Figura 30: CLASSE SEPARA PEDIDO EM JAVA.	40
Figura 31: CLASSE ENVIAPEDIDO EM JAVA.....	41
Figura 32: CLASSE LOJA EM PYTHON.	41
Figura 33: CLASSES DO PADRAO FACADE EM PYTHON.....	42
Figura 34: SAIDA DO TERMINAL DO CÓDIGO FACADE EM PYTHON.	43
Figura 35: CLASSE MUDAESTADO PARA O PADRÃO STATE EM JAVA.....	44
Figura 36: INTERFACES, CLASSES QUE A IMPLEMENTAM EM JAVA.	45
Figura 37: CLASSE CLIENTE EM JAVA.	46
Figura 38: PADRÃO STATE EM PYTHON.....	47

Figura 39: INTERFACE OBSERVER EM JAVA	48
Figura 40: CLASSE MATRIZ QUE IMPLEMENTA O SUBJECT EM JAVA	49
Figura 41: CLASSE FILIAL QUE IMPEMENTA A INTERFACE OBSERVER EM JAVA	50
Figura 42: CLASSE CLEINTE EM JAVA	51
Figura 43: ABSTRAÇÕES OBSERVER, SUBJECT E CLASSE MATRIZ EM PYTHON	52
Figura 44: CLASSES OBSERVADORAS E CLIENTE EM PYTHON.....	53
Figura 45: DIFERENÇA NAS LINGUAGENS JAVA E PYTHON	56
Figura 46: DEMONSTRAÇÃO DECLARAÇÃO DE LISTA EM PYTHON.....	56
Figura 47: DEMONSTRAÇÃO DECLARAÇÃO DE LISTA EM JAVA.....	56
Figura 48: DEMONSTRAÇÃO DE INTERFACE EM JAVA	57
Figura 49: DEMONSTRAÇÃO DE ABSTRAÇÃO DE CLASSE EM PYTHON	57
Figura 50: COMPARATIVO JAVA x PYTHON	58

SUMÁRIO

1	INTRODUÇÃO.....	12
1.1.	OBJETIVOS.....	13
1.1.1.	OBJETIVO GERAL.....	13
1.1.2.	OBJETIVOS ESPECÍFICOS.....	13
1.2.	JUSTIFICATIVA	13
1.3.	METODOLOGIA DE PESQUISA.....	14
2	REFERENCIAL TEÓRICO.....	17
2.1.	PRINCÍPIO SOLID.....	17
2.2.	PADRÕES DE PROJETO.....	22
2.2.1.	PADRÕES DE CRIAÇÃO	23
2.2.2.	PADRÕES ESTRUTURAIS	25
2.2.3.	PADRÕES COMPORTAMENTAIS	27
3	CODIFICAÇÃO EM JAVA E PYTHON UTILIZANDO PADRÕES DE PROJETO	30
3.1.	FACTORY METHOD	30
3.2.	SINGLETON	34
3.3.	ADAPTER	35
3.4.	FACADE	38
3.5.	STATE	43
3.6.	OBSERVER	47
4	ANÁLISE COMPARATIVA NA UTILIZAÇÃO DOS PADRÕES DE PROJETO EM JAVA E PYTHON	53
5	CONSIDERAÇÕES FINAIS.....	59
6	REFERÊNCIAS	60
	ANEXO A – DECLARAÇÃO DE REVISÃO ORTOGRÁFICA	61
	ANEXO B – PARECER TÉCNICO: TRABALHO DE CONCLUSÃO DE CURSO (TCC II).....	62

1 INTRODUÇÃO

Num mundo cada vez mais tecnológico em que vivemos, com *smartphones* no bolso, computadores portáteis, entre outros, a cada dia surgem aplicações e ferramentas facilitadoras ao dia a dia das pessoas, para diversas finalidades e comodidades. É difícil imaginar na atualidade um mundo sem aplicativos e *softwares*, porém, para chegarmos nesse mundo cada vez mais conectado e digital, foi preciso criar esse mundo, codificá-lo, e a cada dia esse mundo cresce, com novas ideias e novos códigos que dão origem aos *softwares* que agregam tanto na vida das pessoas.

Mas como esses *softwares* são criados? Existe algum padrão para que nesses montes de linhas de códigos, de “*ifs e elses*”, saiam as telas que vemos nos *smartphones* e computadores?

Primeiramente, *Software* são sequências de instruções a serem interpretadas por um *Hardware*, e para criar essa sequência de instruções existem diversas linguagens de programação. Nos primórdios, não existia um catálogo de padrões para se escrever um código, eram criados com o instinto do programador em colocar uma abstração do imaginário para solucionar um problema específico, em uma sequência de comandos de código. Mas, com o avanço da tecnologia, o aumento e popularidade da programação, os primeiros projetistas de *software* perceberam que algumas soluções de problemas resolvidos por *softwares* seguiam padrões, e viram a necessidade de se catalogar e registrar esses padrões de soluções de projetos.

E o que é um padrão? É definido como algo, um objeto, que serve de modelo para elaboração de outro. E como isso se encaixa na criação de um *software*?

Padrões de projeto, inicialmente, foram propostos na área de arquitetura civil, posteriormente profissionais de software incorporaram essas ideias para a criação dos *softwares*. Antes disso, não havia nenhuma catalogação de soluções de projetos, e com a criação dos padrões, foi possível aos projetistas de *software* que se deparavam com um certo problema, uma ajuda já preexistente que já funcionara na solução de projetos anteriores.

Utilizar os padrões de projeto em *software* traz inúmeras vantagens não só na criação para o programador, mas também ao produto final, pois um outro desenvolvedor que se deparar com o código terá maior facilidade no entendimento das linhas de código, seja para manutenção ou atualização de software.

Conforme Pressman (2010): “reconhecer os problemas e as suas causas e desmascarar os mitos do software são os primeiros passos em direção à solução”.

Existem vários padrões de projeto, e serão abordados alguns deles ao longo deste trabalho, tais como os padrões *State*, *Singleton*, *Method Factory* e *Observer*.

O foco deste trabalho é a avaliação de verbosidade com implementações em *Java* e *Python* de Padrões de Projeto, bem como abordar os principais padrões de projeto e exemplificá-los, destacando suas vantagens, motivação e aplicabilidade.

1.1. OBJETIVOS

1.1.1. OBJETIVO GERAL

Avaliar a verbosidade para implementações em *Java* e *Python* utilizando padrões de Projeto.

1.1.2. OBJETIVOS ESPECÍFICOS

- Abordar de modo geral a teoria de padrões de projeto;
- Abordar os princípios de programação SOLID;
- Descrever os padrões de projeto *Factory Method*, *Singleton*, *Adapter*, *Facade*, *State* e *Observer*;
- Implementar os seis padrões do item anterior nas linguagens *Java* e *Python*;
- Comparar a verbosidade das implementações dos seis padrões de projeto.

1.2. JUSTIFICATIVA

A tecnologia sempre foi um marco na sociedade, desde a época dos primeiros descobrimentos até a era atual, no qual vemos um mundo cada vez mais digital. Troca de mensagens, videochamadas, pedidos delivery, está presente na rotina de todos que utilizam aparelhos tecnológicos, para tudo hoje existe um Software que atenda a uma necessidade específica. Porém para que tudo isso exista, primeiro surgiram ideias, depois códigos que se tornaram em uma aplicação final, que é o software.

Padrões de projeto de software são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular” (GAMMA, et al., 2000). Padronizar projetos de software foi algo necessário com o tempo, visto que os primeiros Softwares (décadas de 1940-1950) eram criados com um objetivo

bem específico. Ao passar dos anos, com o advento da tecnologia, percebeu-se que as soluções de projetos seguiam um padrão. Sendo assim, podiam reutilizar soluções em projetos posteriores baseados nesses padrões. A Gangue dos 4 (Gang of Four - GoF), como ficaram conhecidos os autores do livro pioneiro no assunto (GAMMA, et al., 2000), tornou-se a referência principal ao se reunir e criar 23 padrões de projetos, ao qual é abordado alguns destes neste trabalho. Os padrões de projetos definidos pelo GoF são divididos em 3 principais categorias:

- Criacionais: São padrões que se preocupam com o processo de criação dos objetos. Ex: *Factory Method* e *Singleton*;
- Estruturais: São padrões que lidam com composição estrutural de classes ou objetos. Ex: *Adapter* e *Facade*;
- Comportamentais: São padrões que se preocupam com a maneira em que as Classes e objetos interagem e se relacionam. Ex: *State* e *Observer*.

O critério de escolha para utilização destes padrões neste trabalho se deve a familiaridade, aplicabilidade e facilidade de utilizar os padrões em exemplos simples e práticos.

A abordagem apresentada neste estudo se mostra muito atrativa por diversos motivos, o mercado de software cresce constantemente, e a aplicação dos padrões de projeto implica diretamente na qualidade, entendimento, manutenção e funcionamento do software.

Com a exemplificação realizada nas linguagens Java e Python, com duas linguagens das mais populares, a demonstração de padrões nas soluções de problemas, contextualizando a aplicabilidade desses padrões nas respectivas linguagens e sintetização dos resultados.

1.3. METODOLOGIA DE PESQUISA

Pesquisa é a construção de conhecimento original de acordo com certas exigências científicas. Para que seu estudo seja considerado científico você deve obedecer aos critérios de coerência, consistência, originalidade e objetivação. É desejável que uma pesquisa científica preencha os seguintes requisitos: “a) a existência de uma pergunta que se deseja responder; b) a elaboração de um conjunto de passos que permitam chegar à resposta; c) a indicação do grau de confiabilidade na resposta obtida” (GOLDEMBERG, 1999, p.106).

De acordo com SILVA, Edna; MENEZES, as abordagens do problema pesquisado podem ser Quantitativas e Qualitativas.

A pesquisa quantitativa considera que tudo pode ser quantificável, podendo traduzir em números, opiniões e informações, para poder classificá-las e analisá-las. Utiliza recursos e técnicas estatísticas, como porcentagem, média, moda, etc.

Já a pesquisa qualitativa considera que há uma relação dinâmica entre o mundo real e o sujeito. A interpretação dos fenômenos e a atribuição de significados são básicas neste processo de pesquisa. Não requer o uso de recursos estatísticos, o ambiente natural é a fonte direta para coleta de dados e o pesquisador é o instrumento-chave. É descritiva e os pesquisadores tendem a analisar seus dados indutivamente.

A partir das abordagens citadas, este trabalho classifica-se como uma pesquisa qualitativa, pois é escrito a partir de uma linguagem descritiva-argumentativa, utilizando coleta de dados de pesquisas bibliográficas, junto a exemplificação desses padrões que serão interpretados, atribuindo significados para este estudo de caso.

Para Vergara (1998), há várias taxionomias de tipos de pesquisa, que são divididas em dois critérios básicos: quanto aos fins e quanto aos meios.

Quanto aos fins, Vergara (1998), os tipos de pesquisa utilizados neste trabalho são: descritiva e explicativa.

A pesquisa descritiva expõe características de uma determinada população ou de determinado fenômeno. Pode também estabelecer correlações entre variáveis e definir sua natureza. Esta pesquisa não tem compromisso em explicar os fenômenos que descreve, porém serve de base para tal explicação.

Uma pesquisa explicativa tem como principal objetivo tornar algo inteligível, justificar lhe os motivos. Visa, portanto, esclarecer quais fatores contribuem para a ocorrência de determinado fenômeno.

Já quanto aos meios, este trabalho utiliza tais tipos de pesquisa: laboratorial, telematizada e bibliográfica.

Pesquisa de laboratório é experiência realizada em local circunscrito, visto que no campo seria praticamente de ser realizada.

A pesquisa telematizada busca informações em meios que combinem o uso de computador com o de telecomunicações, como internet por exemplo.

Uma pesquisa bibliográfica é o estudo sistematizado desenvolvido com base em material publicado em livros, revistas, jornais, redes eletrônicas, acessível ao público em geral.

Para elaboração deste estudo, quanto aos fins, os tipos de pesquisa abordados podem ser classificados como descritiva, pois apresenta características, exemplos e conceitos, além de descrições práticas de teorias de padrões de projeto. Além disso, também se enquadra no tipo

de pesquisa explicativa, ao qual propõe justificar o porquê, a utilização e aplicabilidade dos padrões de projeto, importantes na produção de um software;

Quanto aos meios, se utiliza dos tipos de pesquisa de laboratório, visto que os conceitos e teorias dos padrões de projeto apresentados, implementados e exemplificados são aplicados em softwares computacionais, e telematizada, porque busca combinar informações através de pesquisas adquiridas na internet, combinando-as com os resultados aplicados nos exemplos dados dos padrões de projeto.

2 REFERENCIAL TEÓRICO

2.1. PRINCÍPIO SOLID

Neste capítulo será apresentado o princípio de programação conhecido como SOLID, definido como uma regra usada como base para se aplicar em vários cenários. “Se você se guiar pelos princípios por trás dos Padrões de Projeto, você saberá quando e como aplicá-los.” (DANIEL, Glaucio, 2021).

O SOLID foi criado no início dos anos 2000 por Robert C. Martin (2008). Já o acrônimo SOLID foi introduzido por Michael Feathers (2004), ao observar que os cinco princípios poderiam se encaixar nesta palavra. Cada letra representa um princípio, são eles:

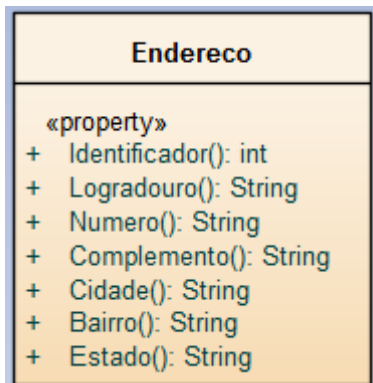
- *Single responsibility Principle;*
- *Open/Closed Principle;*
- *Liskov Substitution Principle;*
- *Interface Segregation Principle;*
- *Dependency Inversion Principle.*

Cada princípio é aplicado em um tipo de situação, de acordo com a necessidade de solução do problema, com o objetivo de se escrever códigos melhores e eficientes.

Single responsibility Principle (Princípio da responsabilidade única):

Este princípio prega que uma classe deve ter um motivo, e somente este motivo, para mudar. Esta classe não pode ter muitos métodos, somente métodos essenciais para sua funcionalidade, para que não seja uma classe inflada que “faz tudo” e seja chamado como uma “*God Class*”, como são conhecidas as classes inchadas de métodos. Isso prejudica na hora de realizar alterações, pois tendo muitos métodos, haverá muito mais trabalho para a alteração ser feita.

Figura 1: DIAGRAMA DE CLASSE DO PRINCÍPIO SIGLE RESPONSABILITY.



Fonte:

<https://www.ateomomento.com.br/wp-content/uploads/2013/12/solid-principio-responsabilidade-unica-classe-endereco.png>

Neste exemplo, é apresentada uma classe com muitos métodos diferentes com muitas especificações, o que torna a classe inflada e com muitas funcionalidades, quando o recomendado pelo princípio é dividir as responsabilidades em classes separadas por essas funcionalidades.

Open/Closed Principle (Princípio Aberto/Fechado):

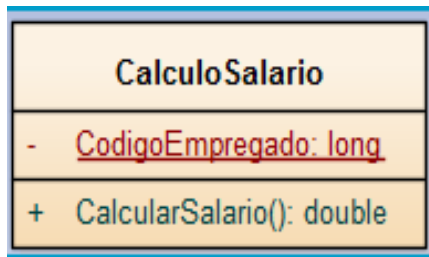
Neste princípio, é considerada a seguinte pergunta: Como é possível realizar alterações em uma implementação de uma aplicação já em andamento, sem quebrar o que já existe? Pois bem, o princípio aberto/fechado consiste em deixar a entidade em questão (pode ser uma classe, função, módulo) aberto para extensão e fechado para modificação. Na prática, significa que pode adicionar novos recursos, porém, nada do que for adicionado pode quebrar o que já foi implementado, e por isso, fechado para modificação.

Dentre as vantagens de se utilizar este princípio é que minimiza os custos da aplicação, pois obtêm-se classes mais objetivas e mais fáceis de serem extensíveis.

No exemplo abaixo é mostrada uma classe responsável por calcular o salário, onde toda a lógica de cálculo é responsabilidade dessa classe.

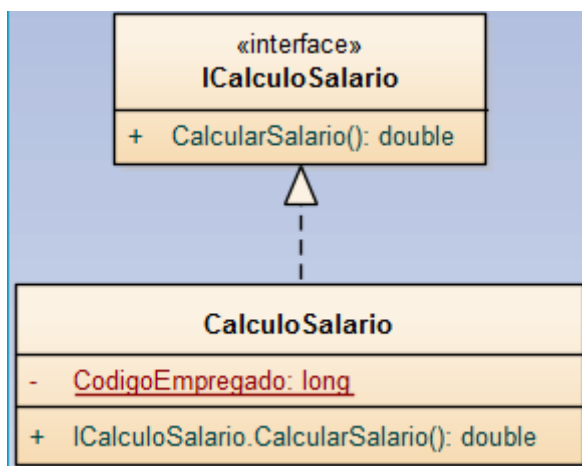
Qualquer alteração na forma de calcular exigirá uma maior alteração nessa e em outras classes que fazem ligações com ela.

Figura 2: DIAGRAMA DE CLASSE OPEN/CLOSED PRINCIPLE.



Fonte: <https://www.ateomomento.com.br/wp-content/uploads/2015/01/solid-open-closed-classe-salario.png>

Figura 3: DIAGRAMA DE CLASSE OPEN/CLOSED PRINCIPLE.



Fonte: <https://www.ateomomento.com.br/wp-content/uploads/2015/01/solid-open-closed-classe-salario-com-interface.png>

Agora, o código acima corrige este problema, onde a Interface “ICalculoSalario” é uma abstração, e os tipos de pagamento estendem essa classe abstrata, implementando seu método “CalcularSalario”.

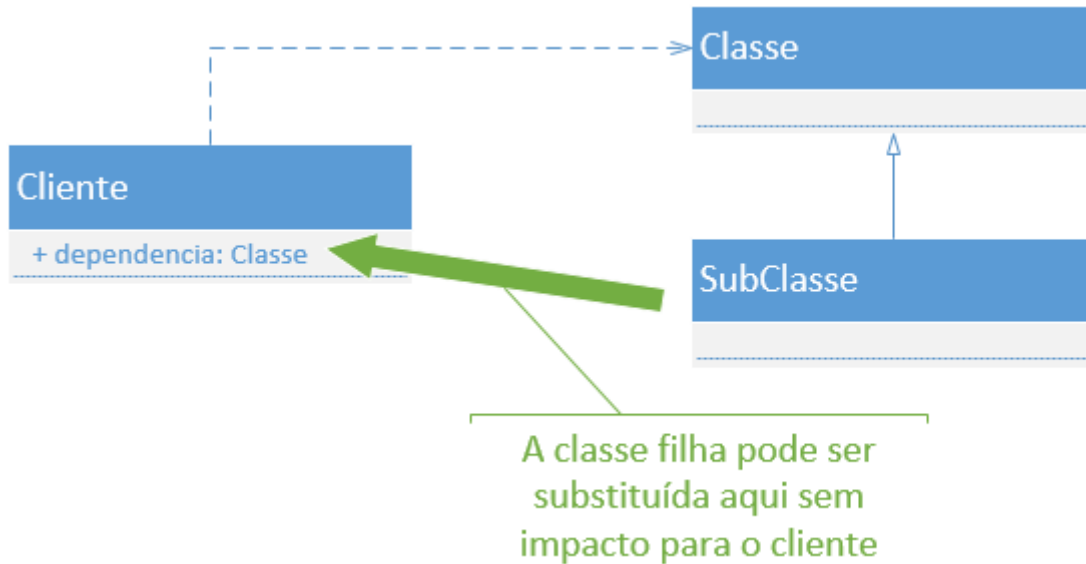
Liskov Substitution Principle (Princípio da Substituição de Liskov):

Este princípio foi definido em 1987 por Bárbara Liskov, e prega que uma classe deve ser substituível por uma classe base. Segundo LISKOV (1988), “o que se deseja aqui é algo como a seguinte propriedade de substituição: se para cada objeto O1 do tipo S existe um objeto O2 do tipo T, tal que, para todos os programas P definidos em termos de T, o comportamento de P fica inalterado quando O1 é substituído por O2, então S é um subtipo de T.”

Em outras palavras, este princípio diz que as instâncias criadas a partir de uma classe derivada, podem ser substituíveis pela classe base, de forma que os objetos não tenham

interferência e a aplicação não será quebrada. Este princípio é uma extensão do *Open/Closed* pois classes derivadas estão estendendo classes base, sem alterar seu comportamento.

Figura 4: DIAGRAMA DE CLASSE LISKOV SUBSTITUTION PRINCIPLE.



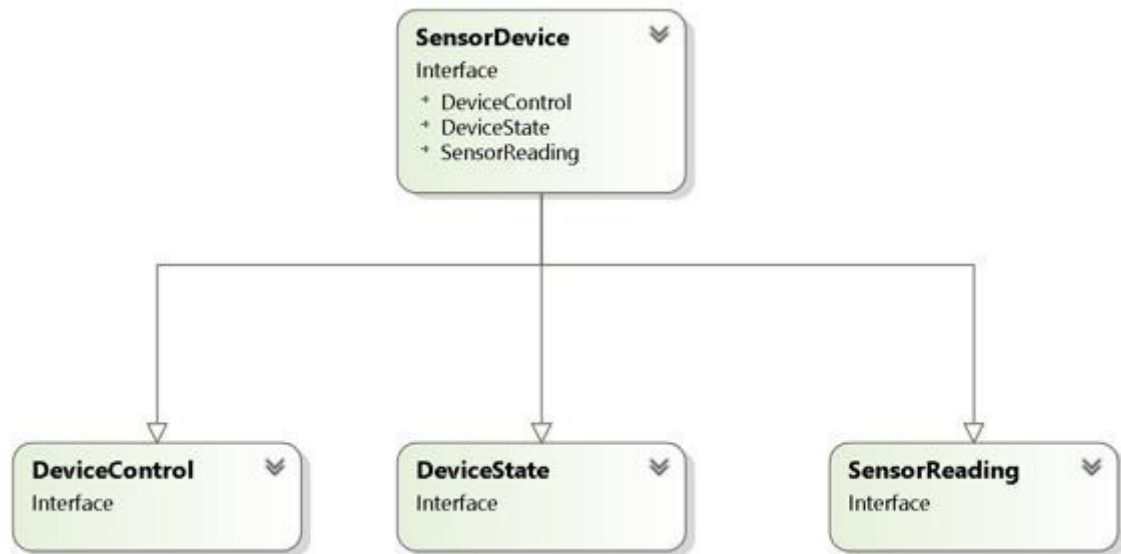
Fonte: [https://4.bp.blogspot.com/-1j-X2eG950w/WL-CBtYj-](https://4.bp.blogspot.com/-1j-X2eG950w/WL-CBtYj-WI/AAAAAAAAAMaY/NbYLBt5x2_E8OfZYEXsxTcTd7zmVVz8BQCLcB/s1600/liskov.png)

[WI/AAAAAAAAAMaY/NbYLBt5x2_E8OfZYEXsxTcTd7zmVVz8BQCLcB/s1600/liskov.png](https://4.bp.blogspot.com/-1j-X2eG950w/WL-CBtYj-WI/AAAAAAAAAMaY/NbYLBt5x2_E8OfZYEXsxTcTd7zmVVz8BQCLcB/s1600/liskov.png)

Interface Segregation Principle (Princípio da Segregação de Interfaces):

Este princípio prega a segregação de interfaces infladas, com muitos métodos, em interfaces menores e mais enxutas. Esta ideia tem a finalidade de quebrar e dividir uma interface que tem muitos métodos implementados, pois as classes que herdam esses métodos muitas vezes não as utilizam. No exemplo abaixo, é mostrado um exemplo do princípio da segregação de interfaces. A interface *SensorDevice* foi subdividida em outras *Interfaces*, com o objetivo de acessar os diferentes métodos por interesse do cliente:

Figura 5: INTERFACE SEGREGATION PRINCIPLE



Fonte: http://stefan-mehnert.de/wp-content/uploads/2016/03/2016-03-06_09-14-35.png

Dependency Inversion Principle (Princípio da Inversão de Dependência):

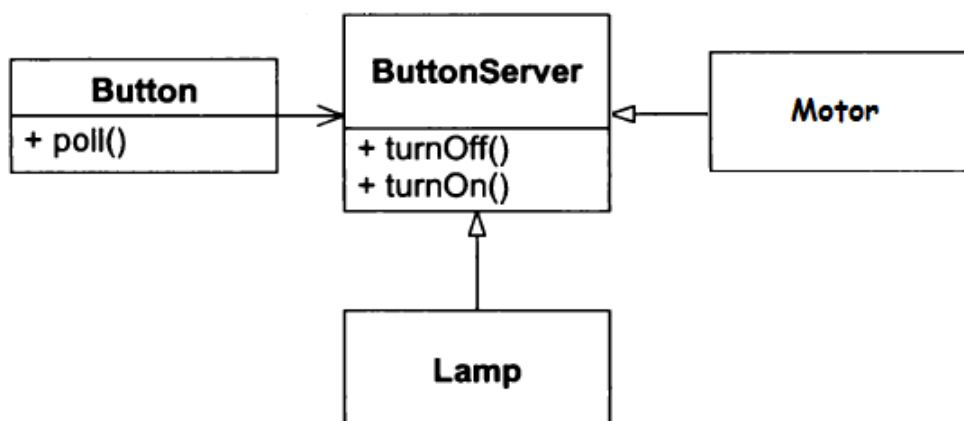
O princípio da inversão de dependência é muito importante para um projeto de arquitetura de *software* flexível. Definição: “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes não devem depender de abstrações.”. Em outras palavras, faz com que a aplicação não fique frágil a mudanças relacionadas a detalhes de implementação.

Figura 6: DIAGRAMA DE CLASSE DO PRINCÍPIO DEPENDENCY INVERSION PRINCIPLE.



Fonte: <https://www.dtidigital.com.br/wp-content/uploads/2015/10/braulio.png>

Figura 7: DIAGRAMA DE CLASSE DO PRINCÍPIO DEPENDENCY INVERSION PRINCIPLE.



Fonte: <https://www.dtidigital.com.br/wp-content/uploads/2015/10/braulio4.png>

No exemplo de diagrama nas figuras acima, o DIP está bem definido, há a inversão de dependência da classe “*Button*” para a classe “*ButtonServer*”, que fará a intermediação de ligar/desligar com as classes “*Lamp*” e “*Motor*”.

2.2. PADRÕES DE PROJETO

Segundo Christopher Alexander (1977), cada padrão descreve um problema no ambiente e o cerne da sua solução, de tal forma que se possa usar essa solução mais de um milhão de vezes, sem nunca o fazer da mesma maneira. Embora Christopher utilize esta definição de padrão para o campo da arquitetura, se aplica perfeitamente na definição de construção de software. Para um desenvolvedor inexperiente que vá começar um projeto do zero, embora tenha conhecimento teórico e domínio das linguagens que utiliza, é muito mais fácil utilizar “cascas” já prontas para solução de um problema, que já foram utilizadas anteriormente. “Uma coisa que os melhores projetistas sabem que não devem fazer, é resolver cada problema a partir de princípios elementares ou do zero” (GAMMA, *et al.*, 2000).

Diante disso, a principal vantagem no uso de padrões de projeto é proporcionar o reuso das soluções propostas para um determinado tipo de problema, assim novos profissionais, mesmo que sem muita experiência, podem projetar *softwares* com a mesma qualidade de um profissional com anos de experiência. Saber utilizar padrões de projeto torna-se, então, boa prática para um projetista de software que deseja tornar-se um profissional melhor.

Segundo Nascimento (2018), existem outros benefícios quanto ao uso dos padrões de projeto, além do reuso de soluções para determinados problemas, tais como o entendimento e manutenção do código-fonte, contribuindo para a redução do acoplamento e para o aumento da coesão das classes, proporcionando a redução de custo e tempo em futuras manutenções.

Um padrão de projeto nomeia, abstrai e identifica aspectos problemáticos comuns e propõe uma solução padrão para esses problemas. O padrão de projeto é usado para identificar as classes e instâncias que participam, seus papéis, colaborações e a distribuição das responsabilidades entre esses participantes. Cada padrão de projeto observa um problema específico ou tópico particular de projeto, orientando o objeto que necessita ser resolvido. Ele descreve em que situação deve ser aplicado, para que tipo de problema resolver, especifica também as consequências, custos e benefícios de sua utilização (GAMMA, HELM, *et al.*, 2000).

Ainda segundo GAMMA, *et al.* (2000), existem três tipos de padrões de projeto, no qual iremos descrever e exemplificar individualmente no estudo. São eles: De criação; Estruturais e Comportamentais. Importante destacar que os padrões de projeto a serem tratados neste trabalho foram definidos arbitrariamente. Deste modo, foram definidos dois de cada uma das categorias mencionadas (criacional, estrutural e comportamental).

2.2.1. PADRÕES DE CRIAÇÃO

Os Padrões de criação, como o nome já diz, são os que criam as instâncias de objetos.

Um exemplo é de uma empresa automobilística, que precisa de pneus para instalar em seus carros. Ao montar o carro ela terceiriza um fornecedor dos pneus, sem se preocupar com o processo de fabricação e envio dos pneus, apenas solicita-os e os instala nos carros. De forma paralela, um sistema de software que usa um padrão de criação, solicita ao método construtor o objeto, ele “fabrica” o objeto, que no exemplo é o pneu, e o envia de volta ao sistema, que o utiliza pronto, sem se preocupar com o processo de instanciação do objeto.

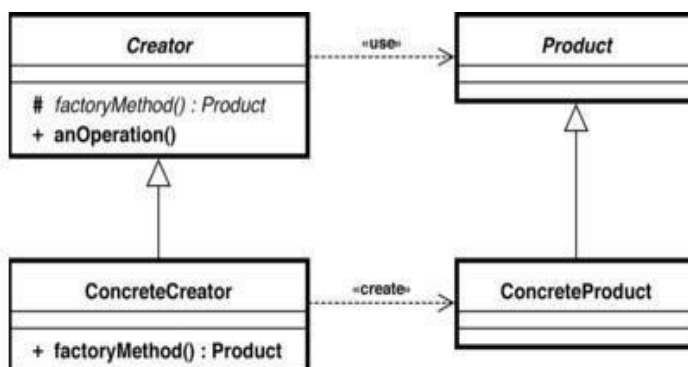
Os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independentemente de como seus objetos são criados, compostos e representados.

Existem diversos padrões de criação, e nesta pesquisa serão exemplificados os padrões *Factory Method* e *Singleton*.

Segundo GAMMA, *et al.* (2000), o padrão *Factory Method* é: “Um padrão que define uma interface para criar um objeto, mas permite às classes decidirem qual classe instanciar. O *Factory Method* permite a uma classe deferir a instanciação para subclasses”.

Em outras palavras, esse padrão encapsula a criação de objetos e deixa as suas subclasses decidirem quais objetos criar.

Figura 8: DIAGRAMA DE CLASSE DO PADRÃO FACTORY METHOD.



Fonte: www.devmedia.com.br/padrao-de-projeto-factory-method-em-java/26348

A principal vantagem da utilização do *Factory Method* é o fato de poder encapsular o código que instancia objetos, desta forma, evita a duplicação e tem-se um único local de manutenção no código.

Uma desvantagem deste padrão é que precisa de uma classe concreta para cada tipo de item produzido, tornando o código mais complexo em alguns casos.

O padrão *Factory Method* tem alguns padrões relacionados, são eles: *Abstract Factory*, *Template Methods* e *Prototype*.

Além deste, de acordo com GAMMA, *et al.* (2000), o padrão *Singleton* assegura que uma classe seja instanciada apenas uma vez, e garante que ela tenha um único ponto de acesso em toda a aplicação.

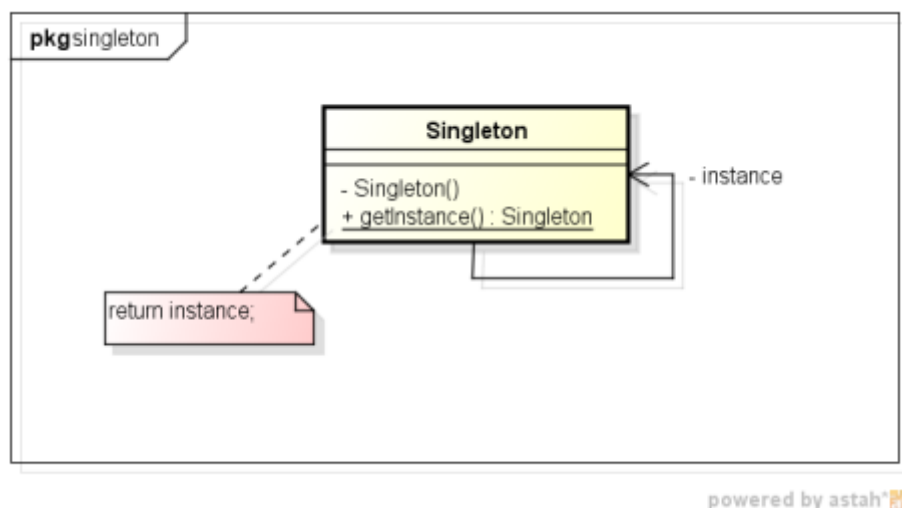
Em outras palavras, garante que se for desejado ter apenas uma instância de determinada classe na aplicação, nenhuma mais, apenas uma, este padrão é o utilizado. Para isso, este padrão precisa ter o seu método construtor sendo privado, ter um método público de acesso e ter um atributo estático da própria classe. Essas regras resolvem o problema de ter várias instâncias na aplicação.

Como vantagem, o uso deste padrão pode ser instanciado e usado somente quando necessário, diferente do caso se usássemos uma variável global, onde o objeto é sempre criado quando o sistema é inicializado, usando recursos não necessários. Assim, facilita o gerenciamento de criação e utilização da instância.

Como desvantagem do padrão *Singleton*, é a impossibilidade de inibir o acesso a classe. Em qualquer parte do código é possível chamar o método de instanciação da classe, pois ele é estático, assim tendo acesso aos dados da classe.

Alguns padrões se relacionam com o padrão *Singleton*, como o *Abstract Factory*, *Builder* e *Prototype*.

Figura 9: DIAGRAMA DE CLASSE DO PADRÃO SINGLETON.



Fonte: www.rafaelsoaresfernandes.com/post/O-padroao-Singleton.aspx

2.2.2. PADRÕES ESTRUTURAIS

Os padrões estruturais têm como objetivo se preocupar na composição de classes e objetos na formação de estruturas maiores.

Segundo GAMMA, et al (2000), os padrões estruturais podem ser aplicados a classes e objetos. Os padrões estruturais de classe utilizam a herança para compor interfaces ou implementações e descrevem maneiras de compor objetos para obter novas funcionalidades.

A partir disso, serão exemplificados dois padrões do tipo estrutural. O *Adapter* e o *Facade*.

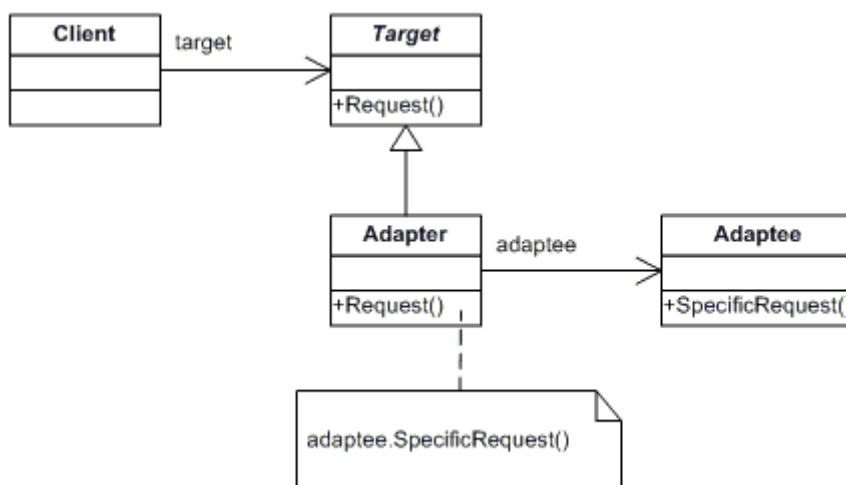
A intenção do padrão *Adapter* segundo GAMMA, et al. (2000), é converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível.

Portanto, o *Adapter* utiliza-se, quando é necessária, a criação de uma nova interface para uma determinada instância, que funciona de forma semelhante, mas possui uma interface incompatível.

O padrão *Adapter* traz alguns benefícios, como tornar o código altamente reutilizável, pois torna possível classes trabalharem juntas, o que não seria possível fazer devido a interfaces incompatíveis.

Existem alguns padrões relacionados ao padrão *Adapter*, que são: O padrão *Bridge*, *Decorator* e o *Proxy*.

Figura 10: DIAGRAMA DE CLASSE ADAPTER.



Fonte: www.devmedia.com.br/padrao-de-projeto-adapter-em-java/26467

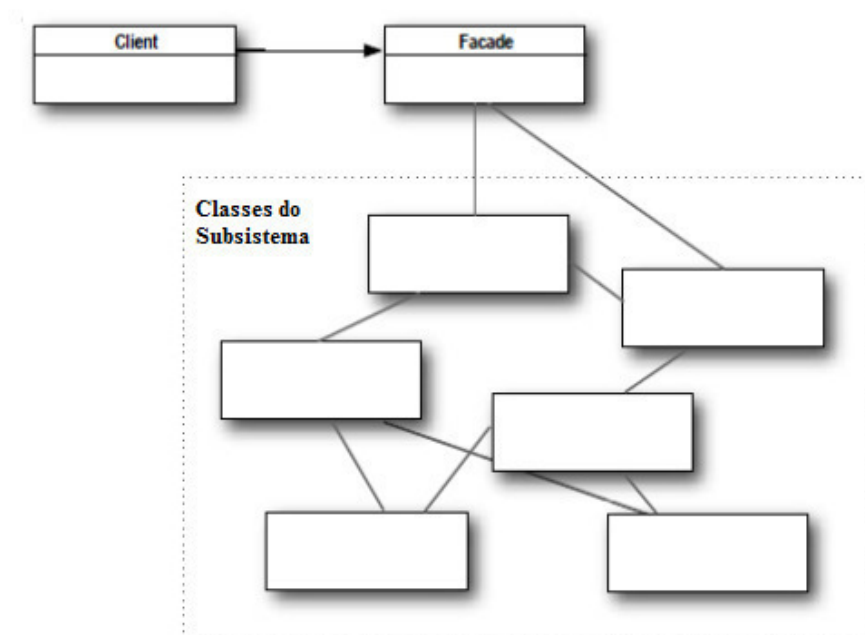
Já o padrão de projeto *Facade* tem como objetivo simplificar uma Interface, ocultando a complexidade de uma ou mais classes. Para GAMMA, *et al.* (2000), a intenção do padrão *Facade* consiste em fornecer uma interface unificada que represente um conjunto de interfaces em um subsistema, facilitando a sua utilização.

Em outras palavras, o uso do padrão *Facade*, permite que um sistema interaja com todos os objetos necessários, através de uma única interface, sem necessidade de comunicar-se com todas as classes que se relacionam para a operação.

Como vantagem, o padrão *Facade* permite desconectar a implementação do cliente de qualquer subsistema, tornando possível acrescentar novas funcionalidades ao subsistema alterando apenas o *Facade* ao invés de vários pontos do sistema, além de simplificar a interface.

Uma das possíveis desvantagens desse padrão é tornar a classe muito inchada, com muitos métodos.

Figura 11: DIAGRAMA DE CLASSE PADRÃO FACADE.



Fonte: www.devmedia.com.br/padroao-de-projeto-facade-em-java/26476

2.2.3. PADRÕES COMPORTAMENTAIS

Segundo GAMMA, *et al.* (2000), os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles.

Em outras palavras, os padrões comportamentais caracterizam o modo em que as classes e objetos interagem e distribuem responsabilidades.

Dentre os padrões comportamentais, serão citados os padrões *State* e *Observer*.

O padrão *State* se caracteriza por encapsular o estado em que o objeto se encontra no sistema, delegando tarefas para o objeto que representa o estado atual.

Para Gamma, *et al.* (2000), o padrão *State* permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe.

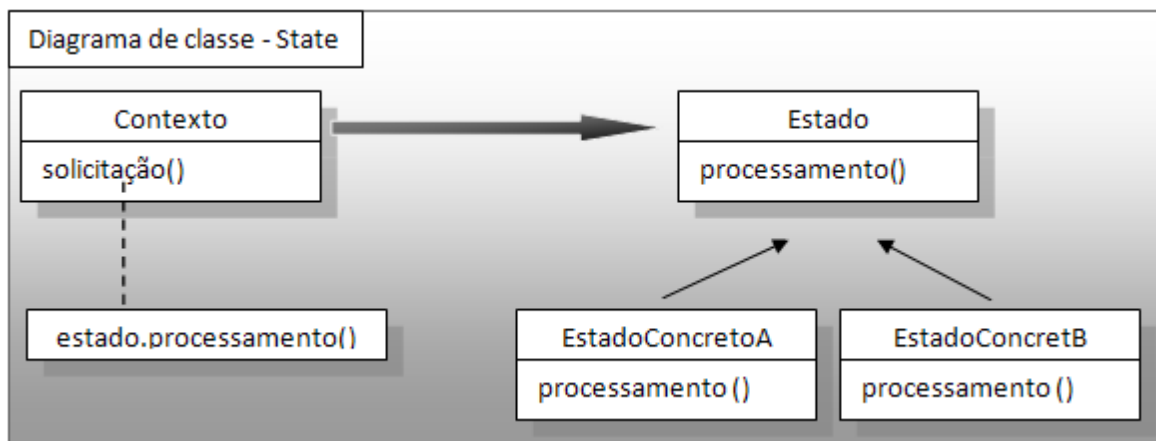
Como exemplo, é mostrado um sistema que rastreie uma entrega, desde o fornecedor até o cliente final. Os estados poderiam ser: “em separação”, “enviado”, “entregue”. Note que o objeto é o mesmo, mas assumiria três estados diferentes na aplicação.

A grande vantagem do padrão *State* é deixar bem definido os estados nos quais os objetos se encontram no sistema e quais suas possíveis transições. Uma desvantagem é que uma

subclasse *State* terá conhecimento de pelo menos uma outra, trazendo dependências de implementação entre subclasses.

Um padrão relacionado ao padrão *State* é o padrão *Singleton*.

Figura 12: DIAGRAMA DE CLASSE DO PADRÃO STATE



Fonte: www.devmedia.com.br/design-patterns-state-parte-4/16783

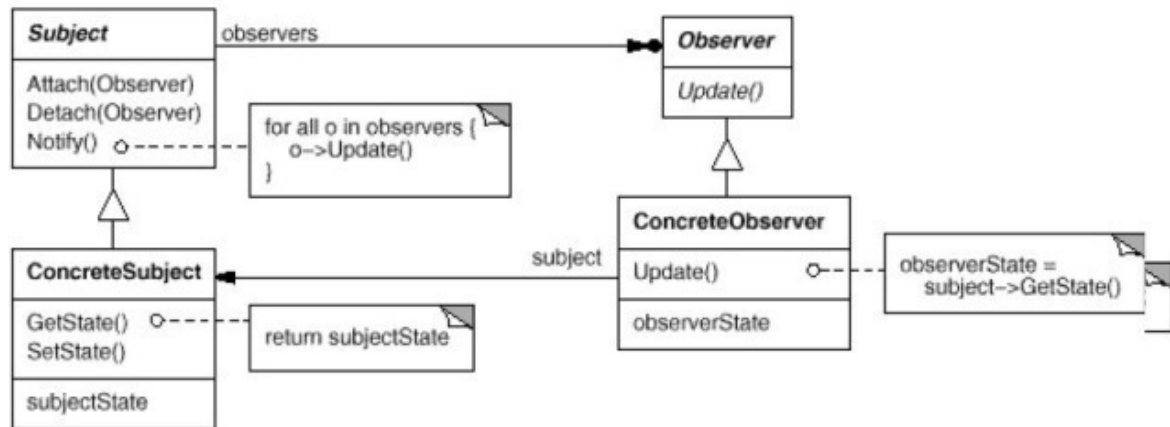
Conforme Gamma, *et al.* (2000), o padrão *Observer* tem a intenção de definir uma dependência um-para-muitos objeto, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

Em outras palavras, o padrão *Observer* é usado quando é preciso manter atualizados todos os objetos de uma determinada relação entre objetos, quando um determinado evento acontece.

Como exemplo, um sistema de rede *fast food*, que ao atualizar seu cardápio na matriz, objeto observado, envia uma notificação a todos os afiliados atualizarem respectivamente seus cardápios.

Uma vantagem do padrão *Observer* é a reutilização de código, visto que tanto os objetos observados quanto os observadores podem ser reutilizados, pois existe um acoplamento entre eles devido ao uso de interfaces e classes abstratas. Porém, o uso indiscriminado deste padrão pode gerar sério problema de performance do sistema, gerando muitas requisições ao notificar os objetos observadores.

Figura 13: DIAGRAMA DE CLASSE DO PADRÃO OBSERVER.



Fonte: www.devmedia.com.br/padrao-de-projeto-observer-em-java/26163

3 CODIFICAÇÃO EM JAVA E PYTHON UTILIZANDO PADRÕES DE PROJETO

Será apresentado neste capítulo exemplos práticos de códigos implementados nas linguagens Java e Python, utilizando os padrões de projeto já apresentados. Serão utilizados exemplos que mostram com clareza a utilização dos Padrões e como eles tornam a programação mais limpa, objetiva e eficaz.

3.1. FACTORY METHOD

No código que será apresentado abaixo, é mostrado o exemplo da utilização do *Factory Method* em uma aplicação de aplicativo de serviços, de locomoção e de pedidos de comida. Primeiramente será apresentado na linguagem Java e subsequentemente na linguagem Python.

Na figura abaixo, é apresentada a *Interface* “Servico”, que tem dois métodos a serem implementados, “iniciaRota()” e “getCarga()”.

Figura 14: REPRESENTAÇÃO DA INTERFACE “SERVICO” NA LINGUAGEM JAVA.

```
c > Factory > Servicos > Servico.java > Se
1 package Factory.Servicos;
2
3 public interface Servico{
4     void iniciaRota();
5     void getCarga();
6 }
```

Fonte: Elaboração própria

Depois, as classes “Ruber” (de locomoção) e “Rifood” (de pedidos de comida), que implementam a *Interface* “Servico” e seus métodos:

Figura 15: REPRESENTAÇÃO EM JAVA DA CLASSE “RUBBER”.

```

rc > Factory > Servicos > Ruber.java > ...
1  package Factory.Servicos;
2  import javax.swing.JOptionPane;
3  public class Ruber implements Servico{
4      @Override
5      public void iniciaRota(){
6          getCarga();
7          JOptionPane.showMessageDialog(null,"Inicia trajeto");
8      }
9      @Override
10     public void getCarga(){
11         JOptionPane.showMessageDialog(null,"Pegou os passageiros e esta pronto");
12     }
13
14 }
15

```

Fonte: Elaboração própria.

Figura 16: REPRESENTAÇÃO EM JAVA DA CLASSE “RIFOOD”.

```

src > Factory > Servicos > Rifood.java > Rifood > getCarga()
1  package Factory.Servicos;
2  import javax.swing.JOptionPane;
3  public class Rifood implements Servico{
4
5      @Override
6      public void iniciaRota() {
7          getCarga();
8          JOptionPane.showMessageDialog(null,"Inicia Entrega");
9      }
10     @Override
11     public void getCarga() {
12         JOptionPane.showMessageDialog(null,"Encomenda retirada e esta pronto");
13     }
14
15 }
16

```

Fonte: Elaboração própria.

Com as classes “Ruber” e “Rifood” criadas, os serviços da aplicação estão prontos. “Rifood” para entregas de comida e “Ruber” para transporte de pessoas.

Agora, será apresentada a fábrica de métodos, o *Factory Method*. Primeiro com uma classe abstrata “Pedido”. Essa classe iniciará efetivamente o serviço prestado, com o método “criaPedido()”:

Figura 17: CLASSE FACTORY REPRESENTADA EM JAVA.

```

rc > Factory > Pedido.java > ...
1  package Factory;
2  import Factory.Servicos.Servico;
3
4  public abstract class Pedido {
5      void iniciaTransporte(){
6          Servico pedido = criaPedido();
7          pedido.iniciaRota();
8      }
9      protected abstract Servico criaPedido();
10
11 }
12

```

Fonte: Elaboração própria.

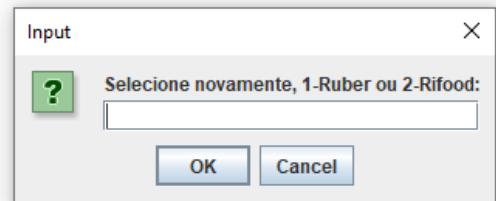
E a seguir a classe “Cliente”, com o método “main()”:

Figura 18: CLASSE CLIENTE CONTENDO O MÉTODO MAIN.

```

rc > Factory > Cliente.java > Cliente > main(String[])
1  package Factory;
2  import javax.swing.JOptionPane;
3
4  public class Cliente {
5      private static Pedido pedido;
6
7      public static void main(String[] args) throws Exception {
8          int type = Integer.parseInt (JOptionPane.showInputDialog("Selecione novamente, 1-Ruber ou 2-Rifood:"));
9          switch(type){
10             case 1:
11                 setPedido(new PedidoRuber());
12                 pedido.iniciaTransporte();
13                 break;
14             case 2:
15                 setPedido(new PedidoRifood());
16                 pedido.iniciaTransporte();
17                 break;
18             default :
19                 JOptionPane.showMessageDialog(null,"Selecione novamente, 1-Ruber ou 2-Rifood:");
20         }
21     }
22     public static Pedido getPedido() {
23         return pedido;
24     }
25     public static void setPedido(Pedido pedido) {
26         Cliente.pedido = pedido;
27     }
28 }

```



Fonte: Elaboração própria.

Agora, será apresentado o código na linguagem em Python que resolve o mesmo exemplo. Uma classe de abstração chamada “Servico”, classes concretas “Ruber” e “Rifood”

que implementam os métodos da classe “Servico”, cada uma com sua especificação, a classe *Factory* “Pedido”, responsável por “fabricar” as instâncias.

Figura 19: EXEMPLO DO CÓDIGO FACTORY METHOD EM PYTHON.

```
factory_method.py > ...
1  from abc import ABCMeta, abstractmethod
2
3  class Servico (metaclass=ABCMeta):
4
5      @abstractmethod
6      def iniciaRota(self):
7          pass
8
9      @abstractmethod
10     def pegaCarga(self):
11         pass
12
13 class Ruber(Servico):
14     def iniciaRota(self):
15         print(f'Iniciando corrida')
16     def pegaCarga(self):
17         print(f'Passageiros embarcados e está pronto para iniciar')
18
19 class Rifood(Servico):
20     def iniciaRota(self):
21         print(f'Iniciando entrega')
22     def pegaCarga(self):
23         print(f'Encomenda retirada pronto para iniciar')
24
25 class Pedido:
26     def cria_pedido(self, tipo):
27         if tipo == 'Ruber':
28             return Ruber()
29         elif tipo == 'Rifood':
30             return Rifood()
31
32 #CLIENTE
33 if __name__ == '__main__':
34     pedido_tipo = Pedido()
35     pedido_nome = input("Ruber ou Rifood: ")
36     pedido = pedido_tipo.cria_pedido(pedido_nome)
37     pedido.pegaCarga()
38     pedido.iniciaRota()
```

Fonte: Elaboração própria.

Percebe-se que o Python é uma linguagem muito mais enxuta, com menos linhas de códigos que a linguagem Java para realizar o mesmo objetivo. Isso é um assunto para a seção 3.7.

3.2. SINGLETON

O padrão *Singleton* é um padrão simples, como já descrito na seção 2.2.1, abaixo segue exemplo do padrão na linguagem Java:

Figura 20: PADRÃO SINGLETON DEMONSTRADO NA LINGUAGEM JAVA.

```

1  public class Universo {
2      private static Universo instanciaUnica;
3      private Universo() {
4      }
5      public static synchronized Universo getInstance() {
6          if (instanciaUnica == null)
7              instanciaUnica = new Universo();
8          return instanciaUnica;
9      }
10
11     Run | Debug
12     public static void main(String[] args) {
13         Universo universo1 = Universo.getInstance();
14         Universo universo2 = Universo.getInstance();
15         System.out.println(universo1);
16         System.out.println(universo2);
17     }

```

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPUÇÃO

```

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\Singleton> & '
a\jdk-16.0.1\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsIn
paceStorage\ea0b288a5d63ed83dc33841512ec5c22\redhat.java\jdt_ws\Singl
Universo@4617c264
Universo@4617c264
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\Singleton>

```

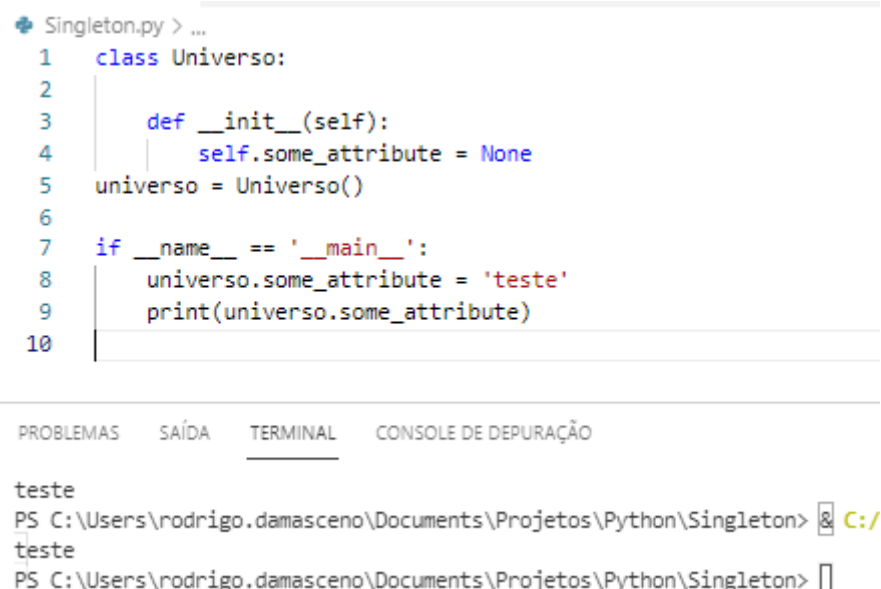
Fonte: Elaboração própria.

No código mostrado, um exemplo de uma classe “Universo”, que pode pertencer a um jogo de batalha espacial por exemplo. Repare que “Universo” é um só, neste exemplo, não tem como existir dois universos no jogo. Logo, o sistema precisa de apenas uma instância da classe “Universo”, e é demonstrada no código com um atributo estático chamado “instanciaUnica”, um método “*getInstance()*” que é onde serão instanciados o objeto e o método construtor privado, que garante que somente esta classe tem o poder de instanciar o objeto de “Universo”.

Na parte final do código, o método “main”, com a criação de duas instâncias, ou tentativas de criação, universo1 e universo2. Observe que na saída do terminal, imprime o mesmo endereço de memória para as duas instancias, isso significa que o padrão *Singleton* está funcionando, garantindo que apenas uma instancia da classe exista.

Agora, segue o mesmo exemplo de aplicação na linguagem Python:

Figura 21: PADRÃO SINGLETON DEMONSTRADO EM PYTHON.



```

Singleton.py > ...
1  class Universo:
2
3      def __init__(self):
4          self.some_attribute = None
5  universo = Universo()
6
7  if __name__ == '__main__':
8      universo.some_attribute = 'teste'
9      print(universo.some_attribute)
10

```

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

teste
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Singleton> C:/
teste
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Singleton>

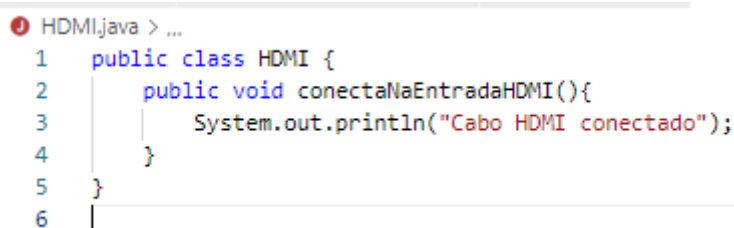
Fonte: Elaboração própria.

O Python permite uma implementação bem mais simples de um programa que faz a mesma coisa. Neste exemplo não foi necessária a criação de um método “getInstance()”, pois foi simplificada através da variável universo.

3.3. ADAPTER

No padrão *Adapter*, como o nome diz, adapta uma instância diferente para um formato reconhecível e direciona a chamada para os métodos do objeto envolvido. No exemplo a seguir, resolvemos o caso de um adaptador de cabo VGA para HDMI. Abaixo, duas classes “VGA” e “HDMI”, e uma classe “AdaptadorHDMI”, que adapta o objeto VGA para funcionar o método “conectaNaEntradaHDMI()”, da classe “HDMI”:

Figura 22: CLASSE HDMI DEMONSTRADA EM JAVA.



```

HDMI.java > ...
1  public class HDMI {
2      public void conectaNaEntradaHDMI(){
3          System.out.println("Cabo HDMI conectado");
4      }
5  }
6

```

Fonte: Elaboração Própria.

Figura 23: CLASSE VGA DEMONSTRADA EM JAVA

```

VGA.java > $ VGA > conectaNaEntradaVGA()
1 public class VGA {
2     public void conectaNaEntradaVGA(){
3         System.out.println("Cabo VGA conectado!");
4     }
5 }
6

```

Fonte: Elaboração Própria.

Figura 24: CLASSE ADAPTADOR HDMI EM JAVA.

```

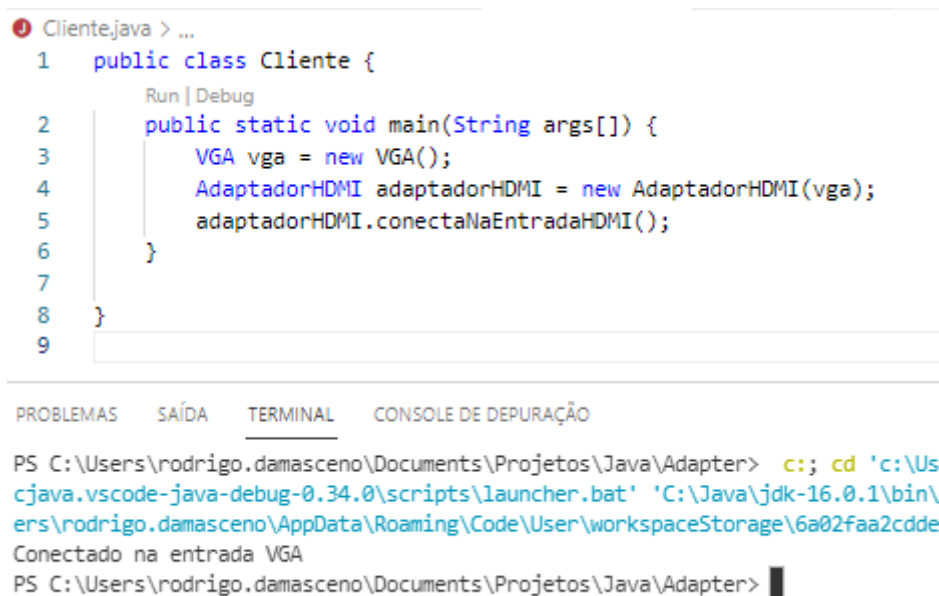
AdaptadorHDMI.java > $ AdaptadorHDMI
1 public class AdaptadorHDMI extends HDMI{
2     private VGA caboVGA;
3
4     public AdaptadorHDMI(VGA caboVga){
5         this.caboVGA=caboVga;
6     }
7     public void conectaNaEntradaHDMI(){
8         caboVGA.conectaNaEntradaVGA();
9     }
10

```

Fonte: Elaboração Própria.

E abaixo, a classe “Cliente”, com a chamada da criação de um objeto vga, outro objeto “adaptadorHDMI”, que na verdade é criado com o vga sendo passado como referência, e chamando o método “conectaNaEntradaHDMI()”. Porém, na saída do Terminal, repare que o que é escrito na tela é que foi conectado com a entrada VGA:

Figura 25: CLASSE CLIENTE EM JAVA.



```
1 public class Cliente {  
    Run | Debug  
2     public static void main(String args[]) {  
3         VGA vga = new VGA();  
4         AdaptadorHDMI adaptadorHDMI = new AdaptadorHDMI(vga);  
5         adaptadorHDMI.conectaNaEntradaHDMI();  
6     }  
7  
8 }  
9
```

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPURACÃO

```
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\Adapter> c::; cd 'c:\Us  
cjava.vscode-java-debug-0.34.0\scripts\launcher.bat' 'C:\Java\jdk-16.0.1\bin\  
ers\rodrigo.damasceno\AppData\Roaming\Code\User\workspaceStorage\6a02faa2cdde  
Conectado na entrada VGA  
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\Adapter>
```

Fonte: Elaboração Própria.

Sendo assim, o padrão *Adapter* está cumprindo seu papel, adaptando a entrada VGA para HDMI.

Agora, segue a mesma resolução de problema na linguagem Python:

Figura 26: CÓDIGO COMPLETO DO PADRÃO ADAPTER EM PYTHON.

```

Adapter.py > ...
1  class VGA:
2      def conectaNaEntradaVGA(self):
3          print(f"Cabo VGA conectado!")
4      pass
5  class HDMI:
6      def conectaNaEntradaHDMI(self):
7          print(f"Cabo HDMI conectado!")
8      pass
9  class AdaptaHDMI(HDMI):
10     def __init__(self, vga: VGA):
11         self.adaptadorVGA = vga
12     def conectaNaEntradaHDMI(self):
13         self.adaptadorVGA.conectaNaEntradaVGA(self)
14     pass
15     pass
16
17 if __name__ == "__main__":
18     caboVGA = VGA
19     adaptadorA = AdaptaHDMI(caboVGA)
20     adaptadorA.conectaNaEntradaHDMI()

```

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

```

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Adapter> & C:/Pyt
Cabo VGA conectado!
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Adapter>

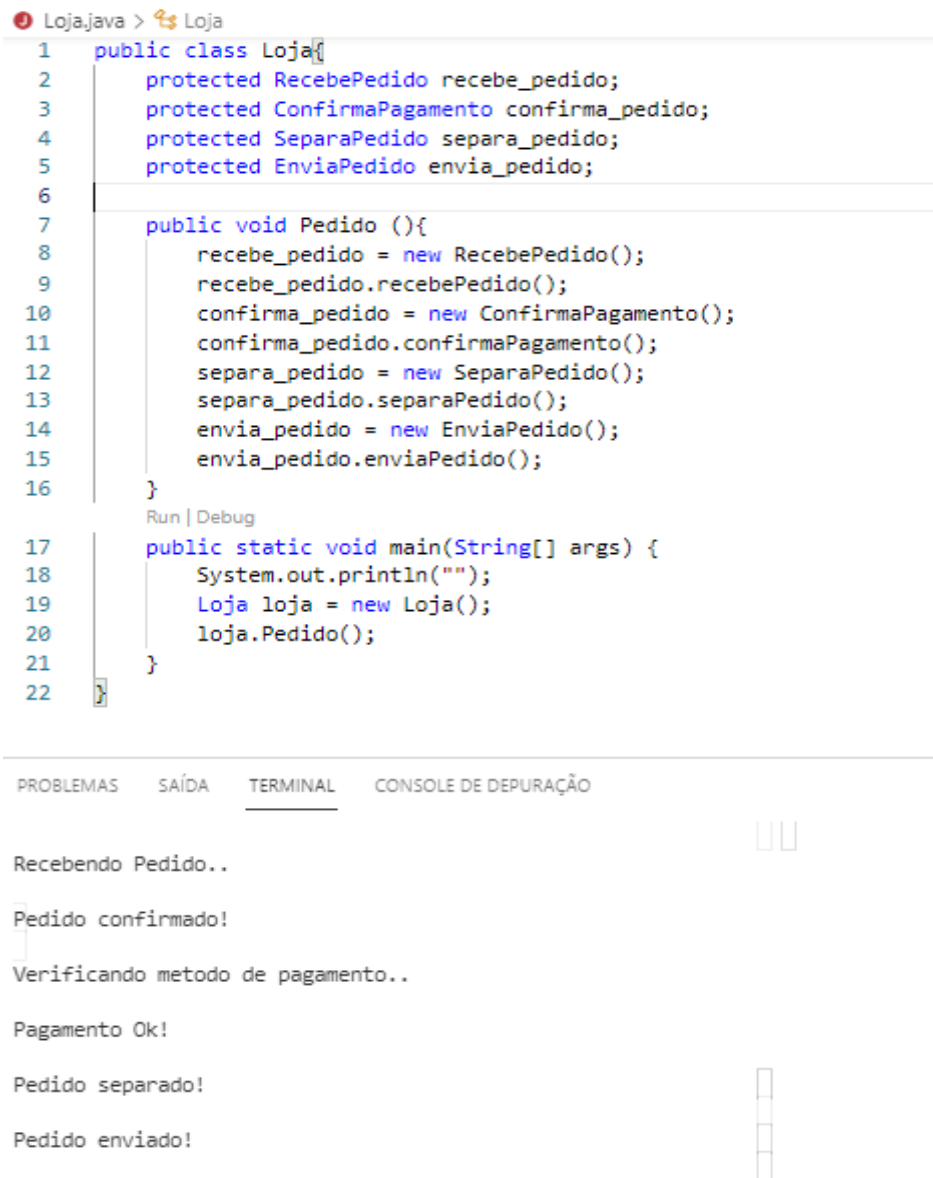
```

Fonte: Elaboração Própria

3.4. FACADE

O exemplo de código utilizado para o padrão *Facade* (Fachada) se trata de um sistema de loja para vendas *online*, onde o pedido é realizado através desse sistema. No código em Java descrito a seguir, é mostrada a classe “Loja”, que é a classe Fachada, e dá acesso a todas as funcionalidades do sistema, que são: “RecebePedido()”, “ConfirmaPagamento()”, “SeparaPedido()” e “EnviaPedido()”. Ao final da Figura 27, o método “main” que mostra a saída das funcionalidades da aplicação.

Figura 27: CLASSE LOJA EM JAVA.



```

Loja.java > Loja
1 public class Loja{
2     protected RecebePedido recebe_pedido;
3     protected ConfirmaPagamento confirma_pedido;
4     protected SeparaPedido separa_pedido;
5     protected EnviaPedido envia_pedido;
6
7     public void Pedido (){
8         recebe_pedido = new RecebePedido();
9         recebe_pedido.recebePedido();
10        confirma_pedido = new ConfirmaPagamento();
11        confirma_pedido.confirmaPagamento();
12        separa_pedido = new SeparaPedido();
13        separa_pedido.separaPedido();
14        envia_pedido = new EnviaPedido();
15        envia_pedido.enviaPedido();
16    }
17
18    public static void main(String[] args) {
19        System.out.println("");
20        Loja loja = new Loja();
21        loja.Pedido();
22    }

```

Run | Debug

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

Recebendo Pedido..
 Pedido confirmado!
 Verificando metodo de pagamento..
 Pagamento Ok!
 Pedido separado!
 Pedido enviado!

Fonte: Elaboração própria.

Figura 28: CLASSE RECEBEPEDIDO EM JAVA.

```

1 public class RecebePedido {
2     public boolean pedido_ok(){
3         return true;
4     }
5     public void recebePedido(){
6         System.out.println("Recebendo Pedido.. \n");
7         if(this.pedido_ok()==true){
8             System.out.println("Pedido confirmado!\n");
9         }
10    }
11
12 }
13

```

Fonte: Elaboração própria

Figura 29: CLASSE CONFIRMAPAGAMENTO EM JAVA.

```

1 public class ConfirmaPagamento {
2     public boolean pagamento_ok(){
3         return true;
4     }
5     public void confirmaPagamento(){
6         System.out.println("Verificando metodo de pagamento.. \n");
7         if(this.pagamento_ok()==true){
8             System.out.println("Pagamento Ok!\n");
9         }
10    }
11
12 }
13

```

Fonte: Elaboração própria

Figura 30: CLASSE SEPARA PEDIDO EM JAVA.

```

1 public class SeparaPedido {
2     public void separaPedido(){
3         System.out.println("Pedido separado! \n");
4     }
5
6 }
7

```

Fonte: Elaboração própria.

Figura 31: CLASSE ENVIAPEDIDO EM JAVA.

```

EnviaPedido.java > EnviaPedido > enviaPedido()
1 public class EnviaPedido {
2     public void enviaPedido(){
3         System.out.println("Pedido enviado! \n");
4     }
5 }
6

```

Fonte: Elaboração própria.

Agora abaixo o mesmo exemplo de aplicação na linguagem Python:

Figura 32: CLASSE LOJA EM PYTHON.

```

Loja.py > Site > recebeCompra
1 class Loja:
2     def __init__(self):
3         print('Loja Online: Seja bem vindo! \n')
4
5     def pedido(self):
6         self.receber_pedido = RecebePedido()
7         self.receber_pedido.recebePedido()
8         self.confirmar_pagamento = ConfirmaPagamento()
9         self.confirmar_pagamento.confirmaPagamento()
10        self.separar_pedido = SeparaPedido()
11        self.enviar_pedido = EnviaPedido()
12

```

Fonte: Elaboração própria.

Figura 33: CLASSES DO PADRAO FACADE EM PYTHON.

```

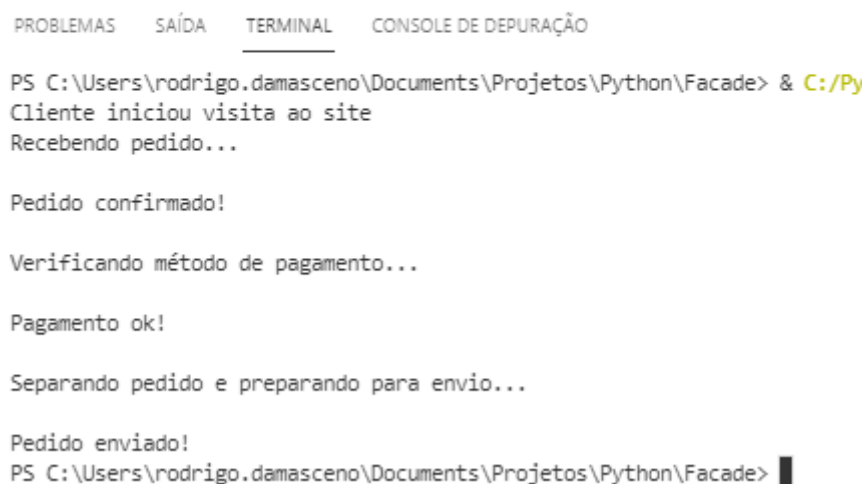
13 class RecebePedido:
14     def __init__(self):
15         print('Recebendo pedido... \n')
16
17     def pedido_ok(self):
18         return True
19
20     def recebePedido(self):
21         if self.pedido_ok():
22             print('Pedido confirmado!\n')
23
24 class ConfirmaPagamento:
25     def __init__(self):
26         print('Verificando método de pagamento...\n')
27
28     def pagamento_ok(self):
29         return True
30
31     def confirmaPagamento(self):
32         if self.pagamento_ok():
33             print('Pagamento ok!\n')
34
35 class SeparaPedido:
36     def __init__(self):
37         print('Separando pedido e preparando para envio...\n')
38
39 class EnviaPedido:
40     def __init__(self):
41         print('Pedido enviado!')
42
43 class Site:
44     def __init__(self):
45         print('Cliente iniciou visita ao site')
46
47     def recebeCompra(self):
48         print('Cliente confirmou carrinho de compras e iniciará pedido...')
49         pe = Loja()
50         pe.pedido()
51

```

Fonte: Elaboração própria.

Figura 34: SAIDA DO TERMINAL DO CÓDIGO FACADE EM PYTHON.

```
52 if __name__ == '__main__':
53     compra = Site()
54     compra.recebeCompra()
```



```
PROBLEMAS  SAÍDA  TERMINAL  CONSOLE DE DEPURAÇÃO

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Facade> & C:/Py
Cliente iniciou visita ao site
Recebendo pedido...

Pedido confirmado!

Verificando método de pagamento...

Pagamento ok!

Separando pedido e preparando para envio...

Pedido enviado!
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\Facade> █
```

Fonte: Elaboração própria.

3.5. STATE

Para o padrão *State*, um mesmo objeto pode ter seu comportamento alterado na aplicação. Nos códigos mostrados abaixo, foi utilizado Clima como exemplo, podendo variar de Sol, Chuvoso e Nublado.

É utilizada uma interface “Clima”, e as classes “Sol”, “Chuvoso” e “Nublado”, que as implementam. Também é usada no exemplo em Java, a classe “MudaEstado”, responsável pela mudança de estados do objeto clima. Segue o exemplo em Java:

Figura 35: CLASSE MUDAESTADO PARA O PADRÃO STATE EM JAVA.

```
MudaEstado.java > ...
1  import java.util.Random;
2  public class MudaEstado {
3      public enum Estado {Sol, Chuvoso, Nublado;};
4      Clima clima;
5      Estado estado;
6      public MudaEstado(){
7      }
8      public Clima mudaEstado(){
9          Random random = new Random();
10         int indice = random.nextInt(2);
11         this.estado = estado.values()[indice];
12         if (estado.toString() == "Sol"){
13             clima = new Sol();
14             return clima;
15         }
16         else if (estado.toString() == "Chuvoso"){
17             clima = new Chuvoso();
18             return clima;
19         }
20         else if (estado.toString() == "Nublado"){
21             clima = new Nublado();
22             return clima;
23         }
24         return clima;
25     }
26 }
27
```

Fonte: Elaboração própria.

Figura 36: INTERFACES, CLASSES QUE A IMPLEMENTAM EM JAVA.

```
Clima.java > ...
1  public interface Clima {
2      public default String getTempo(){
3          return null;
4      }
5  }
6

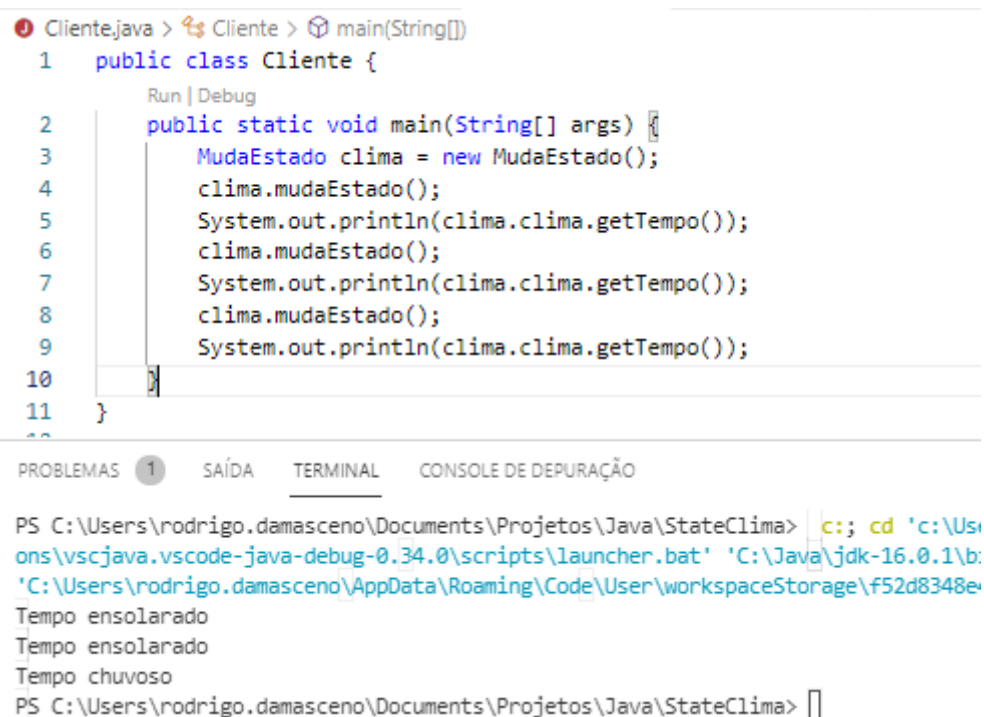
Sol.java > Sol
1  public class Sol implements Clima {
2      @Override
3      public String getTempo() {
4          return "Tempo ensolarado";
5      }
6  }
7
8

Chuvoso.java > ...
1  public class Chuvoso implements Clima {
2      @Override
3      public String getTempo() {
4          return "Tempo chuvoso";
5      }
6  }

Nublado.java > ...
1  public class Nublado implements Clima{
2      @Override
3      public String getTempo() {
4          return "Tempo nublado";
5      }
6  }
7  }
8
```

Fonte: Elaboração própria.

Figura 37: CLASSE CLIENTE EM JAVA.



```

1  public class Cliente {
2      public static void main(String[] args) {
3          MudaEstado clima = new MudaEstado();
4          clima.mudaEstado();
5          System.out.println(clima.clima.getTempo());
6          clima.mudaEstado();
7          System.out.println(clima.clima.getTempo());
8          clima.mudaEstado();
9          System.out.println(clima.clima.getTempo());
10     }
11 }

```

PROBLEMAS 1 SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

```

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\StateClima> c::; cd 'c:\Users\rodrigo.damasceno\Documents\Projetos\Java\StateClima' & .\vscjava.vscode-java-debug-0.34.0\scripts\launcher.bat 'C:\Java\jdk-16.0.1\bin\java.exe' -cp 'C:\Users\rodrigo.damasceno\AppData\Roaming\Code\User\workspaceStorage\f52d8348e\workspace\StateClima\out\production\StateClima'
Tempo ensolarado
Tempo ensolarado
Tempo chuvoso
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\StateClima>

```

Fonte: Elaboração própria.

Agora, o exemplo na linguagem Python:

Figura 38: PADRÃO STATE EM PYTHON

```

ClimaTempo.py > ...
1  from __future__ import annotations
2  from abc import ABC, abstractmethod
3  import random
4  class Clima:
5      def __init__(self):
6          self.estados = [Sol(), Chuva(), Nublado()]
7          self.estado = None
8      def mudaEstado(self):
9          self.estado=self.estados[random.randint(0, 2)]
10         return self.estado
11  class Estado(ABC):
12      @staticmethod
13      @abstractmethod
14      def __str__(self):
15          pass
16  class Sol(Estado):
17      def __str__(self):
18          return "Tempo ensolarado."
19  class Chuva(Estado):
20      def __str__(self):
21          return "Tempo chuvoso."
22
23  class Nublado(Estado):
24      def __str__(self):
25          return "Tempo Nublado."
26  if __name__ == "__main__":
27      clima = Clima()
28      print(clima.mudaEstado())
29      print(clima.mudaEstado())
30      print(clima.mudaEstado())

```

PROBLEMAS SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

Experimente a nova plataforma cruzada PowerShell <https://aka.ms/pscore6>

```

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\State> & C:/Python39/python
Tempo Nublado.
Tempo Nublado.
Tempo ensolarado.
PS C:\Users\rodrigo.damasceno\Documents\Projetos\Python\State>

```

Fonte: Elaboração própria.

3.6. OBSERVER

No padrão *Observer*, como já explicado na seção 2.2.3, existe uma instância observada que possui dependentes (*observers*), e essa instância notifica a todos os seus dependentes acerca de qualquer mudança sofrida por ela. No código que é exemplificado abaixo em Java, foi demonstrada uma rede de restaurantes. Essa rede possui uma Matriz e filiais no estado do Rio e SP. A Matriz possui um cardápio que é tabelado, isto é, as filiais devem oferecer o mesmo

cardápio. Ao realizar qualquer mudança no cardápio da Matriz, o sistema deve notificar a todas as filiais da alteração sofrida. Segue exemplo na linguagem Java:

Figura 39: INTERFACE OBSERVER EM JAVA

```

1  package Observer;
2  import java.util.List;
3
4  public interface Observer {
5      public void update(List<String> cardapio);
6  }
7  public interface SubjectFilial{
8      public void addObserver( Observer observer );
9      public void removeObserver( Observer observer );
10     public void notifyObservers();
11
12
13
14 }
15

```

Fonte: Elaboração própria.

A figura acima mostra a demonstração das *Interfaces Observer* e *Subject*. O objeto instanciado a partir da classe que implementa a *Interface Subject* será o objeto observado, que no caso é a Matriz. O *Subject* é responsável por conter as referências das instâncias observadoras, assim notificar quando houver mudanças.

A interface *Object* é responsável por abstrair as classes observadores, no caso são “FilialRio” e “FilialSP”.

Abaixo, as imagens que demonstram a codificação da “Matriz” e “FilialRio”. A “FilialSP” tem a mesma codificação da “FilialRio”, mudando apenas o nome da classe:

Figura 40: CLASSE MATRIZ QUE IMPLEMENTA O SUBJECT EM JAVA.

```

1  package Observer;
2  import java.util.ArrayList;
3  import java.util.List;
4  public class Matriz implements Subject{
5      private List<String> cardapio = new ArrayList();
6      private List<Observer> filiais;
7      public Matriz(){
8          filiais = new ArrayList<>();
9      }
10     public void adicionaItemCardapio(String item){
11         cardapio.add(item);
12         notifyObservers();
13         System.out.println("Item adicionado ao cardápio. Atualizado!\n");
14     }
15     public void removeItemCardapio(String item){
16         cardapio.remove(item);
17         notifyObservers();
18     }
19     @Override
20     public void addObserver(Observer observer) {
21         filiais.add(observer);
22     }
23
24     @Override
25     public void removeObserver(Observer observer) {
26         for( Observer filial :filiais ){
27             if(filial==observer){
28                 filiais.remove(filial);
29                 break;
30             };
31         }
32     }
33     @Override
34     public void notifyObservers() {
35         for( Observer filial :filiais ){
36             filial.update(cardapio);
37         }
38     }
39 }

```

Fonte: Elaboração própria.

Figura 41: CLASSE FILIAL QUE IMPEMENTA A INTERFACE OBSERVER EM JAVA

```

1  package Observer;
2  import java.util.ArrayList;
3  import java.util.List;
4
5  public class FilialRio implements Observer{
6      private List<String> cardapio = new ArrayList();
7      private Subject filial;
8
9      public FilialRio(Subject filial){
10         this.filial = filial;
11         this.filial.addObserver(this);
12     }
13     public void exibirCardapio(){
14         System.out.println("Cardapio Filial Rio:");
15         for( String exibecardapio :cardapio){
16             System.out.println(exibecardapio);
17         }
18         System.out.println("\n");
19     }
20
21     @Override
22     public void update(List<String> cardapio) {
23         this.cardapio = cardapio;
24     }
25 }
26

```

Fonte: Elaboração própria.

A seguir, a classe Cliente com o método “main()”, com as chamadas dos objetos filial Matriz, “filialRJ” e “filialSP”. Nas linhas 8, 11 e 14 são adicionados itens ao cardápio da filial Matriz, posteriormente atualizados nas filiais RJ e SP, e depois mostrador através do terminal:

Figura 42: CLASSE CLEINTE EM JAVA

```

Observer > Cliente.java > Cliente > main(String[])
1  package Observer;
2
3  public class Cliente {
    Run | Debug
4  public static void main( String[] args ){
5      Matriz filial = new Matriz();
6      FilialRio filialRJ = new FilialRio(filial);
7      FilialSP filialSP = new FilialSP(filial);
8      filial.adicionaItemCardapio("Pizza");
9      filialRJ.exibirCardapio();
10     filialSP.exibirCardapio();
11     filial.adicionaItemCardapio("Hamburgue");
12     filialRJ.exibirCardapio();
13     filialSP.exibirCardapio();
14     filial.adicionaItemCardapio("Salada");
15     filialRJ.exibirCardapio();
16     filialSP.exibirCardapio();
17 }
18 }

```

PROBLEMAS 7 SAÍDA TERMINAL CONSOLE DE DEPURAÇÃO

Item adicionado ao cardápio. Atualizado!

Cardapio Filial Rio:

Pizza
Hamburgue
Salada

Cardapio Filial SP:

Pizza
Hamburgue
Salada

PS C:\Users\rodrigo.damasceno\Documents\Projetos\Java\Observer> █

Fonte: Elaboração própria.

Agora, nas Figura 43 e 44, segue o mesmo problema na linguagem Python:

Figura 43: ABSTRAÇÕES OBSERVER, SUBJECT E CLASSE MATRIZ EM PYTHON

```

RedeRestaurantes.py > ...
1  from abc import ABCMeta, abstractmethod
2  from typing import List
3
4  class Observer(metaclass=ABCMeta):
5      @abstractmethod
6      def atualizar(cardapio):
7          pass
8
9  class Subject(metaclass=ABCMeta):
10     @abstractmethod
11     def adicionaFilial(self,observer: Observer):
12         pass
13     @abstractmethod
14     def retiraFilial(self,observer: Observer):
15         pass
16     @abstractmethod
17     def notificarFiliais(self):
18         pass
19
20 class Matriz (Subject):
21     filiais: List[Observer] = []
22     cardapio = []
23     def __init__(self):
24         pass
25     def adicionaFilial(self, filial: Observer):
26         self.filiais.append(filial)
27         self.notificarFiliais
28     def retiraFilial(self, filial: Observer):
29         if not filial:
30             return self.filiais.pop()
31         else:
32             return self.filiais.remove(filial)
33     def notificarFiliais(self):
34         for filial in self.filiais:
35             filial.atualizar(self.cardapio)
36     def adicionaItemCardapio(self, item):
37         self.cardapio.append(item)
38         print('Item adicionado:',item ,'. Cardápio atualizado!\n\n')
39     def removeItemCardapio(self, item):
40         self.cardapio.remove(item)

```

Fonte: Elaboração própria.

Figura 44: CLASSES OBSERVADORAS E CLIENTE EM PYTHON

```

43  #Observadores
44  class FilialRio(Observer):
45      cardapio = []
46      def __init__(self, filial_restaurante: Subject):
47          self.filial_restaurante = filial_restaurante
48          self.filial_restaurante.adicionaFilial(self)
49
50      def atualizar(cardapio):
51          self.cardapio = cardapio
52
53      def exibircardapio(self):
54          print('Filial Rio cardapio:')
55          print(self.cardapio)
56
57  class FilialSP(Observer):
58      cardapio = []
59      def __init__(self, filial_restaurante: Subject):
60          self.filial_restaurante = filial_restaurante
61          self.filial_restaurante.adicionaFilial(self)
62
63      def atualizar(cardapio):
64          self.cardapio = cardapio
65
66      def exibircardapio(self):
67          print('Filial Rio cardapio:')
68          print(self.cardapio)
69
70  #Cliente
71  if __name__ == '__main__':
72
73      filial = Matriz()
74      filialRJ = FilialRio(filial)
75      filialSP = FilialSP(filial)
76      filial.adicionaItemCardapio('Hamburguer')
77      filialRJ.exibircardapio

```

Fonte: Elaboração própria.

4 ANÁLISE COMPARATIVA NA UTILIZAÇÃO DOS PADRÕES DE PROJETO EM JAVA E PYTHON

Como todas as linguagens faladas, as linguagens de programação também evoluem, e existem linguagens para diferentes tipos de realidades e necessidades. As duas linguagens abordadas deste trabalho são orientadas a objetos e de alto nível. A linguagem Java, lançada em 1995, é de largo espectro. Ou seja, envolve sistemas móveis, *desktop*, embarcados e empresariais. Python por sua vez, lançada em 1991, originalmente foi uma linguagem baseada para o desenvolvimento Linux e posteriormente tornou-se uma das linguagens mais utilizadas

nas tendências de inteligência artificial e análise de dados (*Big Data*), além de também ser utilizada para sistemas *web*.

A análise comparativa das linguagens Java e Python, que é o estudo de caso mais relevante deste trabalho, onde é desenvolvida as principais diferenças características dessas linguagens, pôde ser realizada com base nos códigos implementados utilizando os padrões de projeto. Visualmente, conforme visto nos exemplos dos problemas e das soluções apresentadas nas duas linguagens, percebemos que em relação ao Python, o Java necessita de mais linhas de código e comandos maiores para resolver o mesmo problema.

No padrão apresentado *Factory Method*, algumas diferenças utilizadas nas abordagens de Java e Python serão destacadas. As Figuras 16 e 19 mostram as implementações da classe Rubber nas linguagens Java e Python respectivamente. Em Java, quando é implementado uma interface, é utilizada a palavra *implements* antes do nome da abstração a ser implementada (Figura 16). Em Python é diferente, não existe a interface de Java como apenas uma coleção de métodos abstratos. Ao invés disso, deve-se utilizar uma classe abstrata através do módulo o *Abstract Base Class (ABC)*. Para importar módulos em Python, é usado o *import*, como ilustrado na Figura 49, onde é importado os módulos “ABCMeta” e “abstractmethod” da biblioteca “abc”.

Já no padrão *Singleton*, nas Figuras 20 e 21, respectivamente, são demonstradas nas linguagens Java e Python como este padrão é utilizado, onde somente uma instância de determinado objeto pode existir na aplicação. Na Figura 20, em Java, é mostrado que apesar de dois objetos serem criados no método “*main*”, ambos retornam o mesmo endereço de memória, mostrando que só uma instância desse objeto existe, através do método “*getInstance*”, que verifica com um *if* se a “*instanciaUnica*” (atributo estático da classe Universo) é *Null* e, se caso for, retorne uma nova instância criada, se não for, retorna a instancia já criada anteriormente.

Na Figura 21 é mostrado um exemplo em Python muito mais simplificado, graças a utilização de variáveis de módulos, pois no Python os módulos são carregados apenas uma vez na aplicação. Demonstrado na Figura 21, o padrão foi simplificado com a variável universo, declarada no escopo do módulo Singleton.py. Este modo simplificado é possível devido ao fato de que módulos em Python funcionem como um objeto *Singleton*, carregados apenas uma vez.

Agora, no padrão *Adapter*, foi demonstrado nas Figuras 22, 23, 24 e 25 um exemplo em Java da implementação deste *Pattern*. É utilizado o conceito de herança neste exemplo, na classe “AdaptadorHDMI” (Figura 24) que estende a classe HDMI (Figura 22).

O conceito de herança, que permite a criação de subclasses a partir de outras classes na linguagem Java, onde as subclasses herdam métodos e atributos da superclasse, utilizando a palavra reservada *extends* seguida do nome da superclasse para definir a subclasse.

Em Python a sintaxe deste conceito é diferente, basta acrescentar o nome da superclasse entre parênteses logo após definir o nome da classe, como no exemplo da Figura 26, onde “AdaptadorHDMI” é subclasse da classe HDMI.

Destaca-se no padrão *Facade*, demonstrado nas Figuras 27, 28, 29, 30 e 31 na linguagem Java e nas Figuras 32, 33 e 34 na linguagem Python, formas diferentes de se referenciar o próprio objeto, quando for utilizar métodos ou atributos que pertencem a este objeto. Na linguagem Java a forma de se referenciar é demonstrada com a palavra reservada *this*, como mostrado na Figura 29, onde “this.pagamento_ok” faz referência o atributo booleano “pagamento_ok” da classe “ConfirmaPagamento.java”.


Em Python, essa referência é feita de outra forma, com a palavra reservada *self*. Como demonstrado na Figura 33, onde “self.pedido_ok” faz referência ao método “def pedido_ok (self)”.


No padrão *State*, a instanciação de um novo objeto em Java demonstrado na Figura 35, onde é comparado o nome retornado do *enum* estado e assim criado um objeto com a palavra reservada *new*. O *new* em Java representa o comando para criação de um novo objeto, seguido pelo nome da classe que tipifica este objeto. Em Python, como demonstrado na Figura 38, é apresentado o método especial, “__init__”. Este método é como se fosse o “inicializador” da instância, e o *self*, como explicado anteriormente, inicializa os atributos deste objeto.

Por fim, no padrão *Observer*, que é demonstrado nas Figuras 39, 40, 41 e 42 na linguagem Java, é destacado na Figura 47, linha 5, a criação de um *ArrayList* para guardar os itens do cardápio, da classe “Matriz”. Em Java, para utilizar *ArrayList*, deve-se importar a biblioteca “java.util.ArrayList;”. Na declaração, coloca-se: “ArrayList<Objeto> nomeDoArrayList = new ArrayList<Objeto>”.

Na linguagem Python essa declaração é bem mais simples, levando em conta que Python é uma linguagem com tipagem dinâmica. Na Figura 46, nas linhas 21 e 22, é mostrado o exemplo de duas listas sendo criadas de formas diferentes e muito mais simplistas.

Figura 45: DIFERENÇA NAS LINGUAGENS JAVA E PYTHON


Java


Python

```
public class Main
{
    public static void main(String[] args) {
        System.out.println("This is java");
        System.out.println("with strict syntax");
    }
}
```

```
print('This is a python program')

print('with indentation')
```

Output

This is java
with strict syntax

This is a python program
with indentation

Fonte: <https://dz2cdn1.dzone.com/storage/temp/14102905-14-10-2020-wednesday-1-4-1.png>

O Python é bem mais simples em seus comandos comparado ao Java, como nas declarações de variáveis por exemplo, onde são dinamicamente tipadas, sem necessidade de declarar o tipo da variável, diferentemente do Java.

Figura 46: DEMONSTRAÇÃO DECLARAÇÃO DE LISTA EM PYTHON

```
20 class Matriz (Subject):
21     filiais: List[Observer] = []
22     cardapio = []
23     def __init__(self):
24         pass
```

Fonte: Elaboração própria.

Figura 47: DEMONSTRAÇÃO DECLARAÇÃO DE LISTA EM JAVA

```
1 package Observer;
2 import java.util.ArrayList;
3 import java.util.List;
4 public class Matriz implements Subject{
5     private List<String> cardapio = new ArrayList();
6     private List<Observer> filiais;
7     public Matriz(){
8         filiais = new ArrayList<>();
9     }
10 }
```

Fonte: Elaboração própria.

Outra diferença é na implementação de abstrações. Enquanto no Java, ao criar uma Classe Abstrata ou *Interface*, é preciso tipificar na criação da abstração o que ela é, com *Abstract* ou *Interface*, como no exemplo abaixo:

Figura 48: DEMONSTRAÇÃO DE INTERFACE EM JAVA

```
1 package Observer;
2 import java.util.List;
3
4 public interface Observer {
5     public void update(List<String> cardapio);
6 }
```

Fonte: Elaboração própria.

Enquanto no Python essa criação é diferente, como mencionado anteriormente, incorpora-se o módulo *Abstract Base Class* (ABC) que fornece a infraestrutura para definir as Classes Abstratas. No Python não existe uma sintaxe exclusiva para a criação de *Interfaces*, mas pode-se criar um objeto que funcione com as mesmas características.

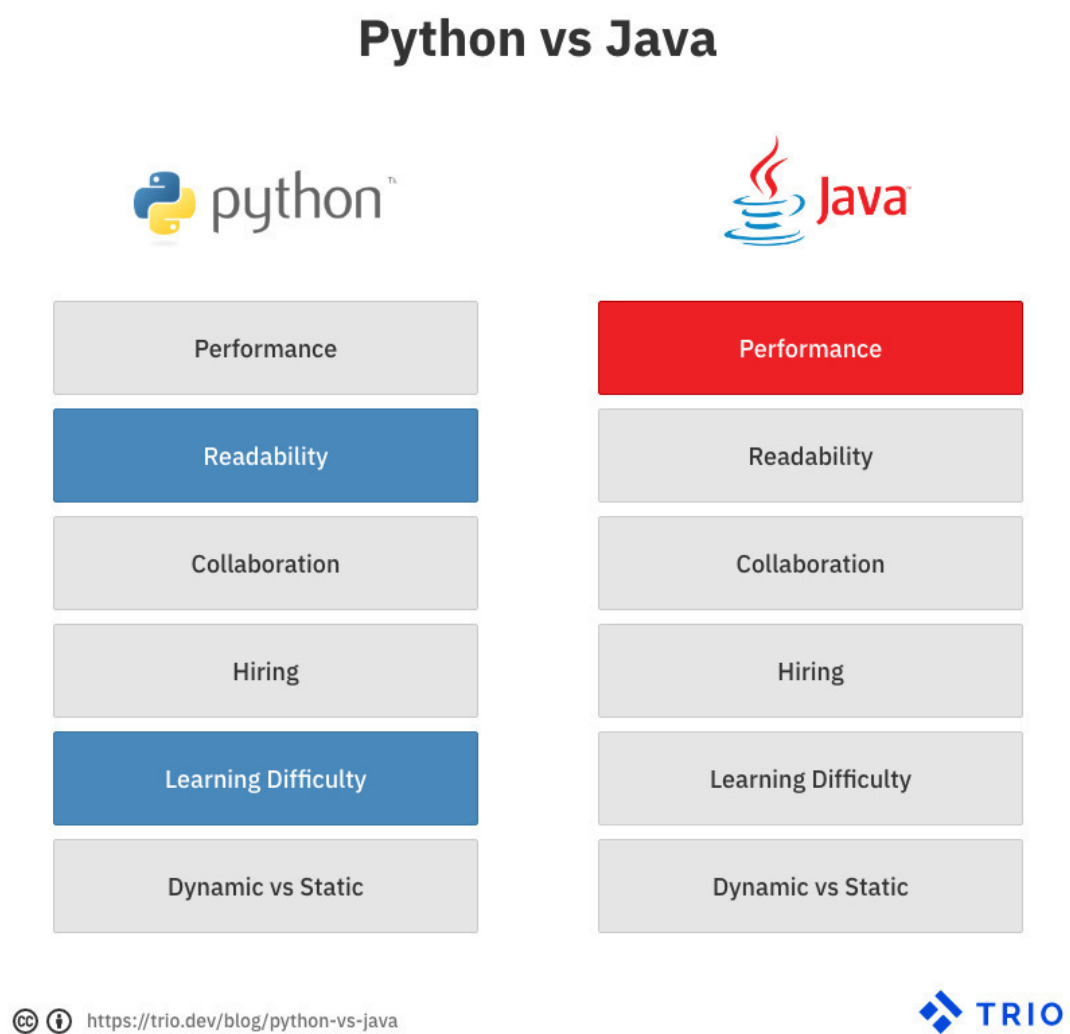
Figura 49: DEMONSTRAÇÃO DE ABSTRAÇÃO DE CLASSE EM PYTHON

```
1 from abc import ABCMeta, abstractmethod
2 from typing import List
3
4 class Observer(metaclass=ABCMeta):
5     @abstractmethod
6     def atualizar(cardapio):
7         pass
8
```

Fonte: Elaboração própria.

Quanto a desempenho, imagina-se que Python por exigir menos linhas de código que Java para implementar um mesmo problema, tem mais desempenho. Mas não é bem assim, tanto Python como Java não possuem regras inerentes a velocidade e desempenho, isso está muito mais ligado a característica das implementações e especificação do problema no qual a linguagem mais se encaixa. Porém, pelo fato de Java ser codificado estaticamente, enquanto Python é codificado dinamicamente, Java pode ser considerado mais rápido pois isso afeta a velocidade de processamento.

Figura 50: COMPARATIVO JAVA x PYTHON



Fonte: <https://res.cloudinary.com/usetrio/image/upload/v1609860835/mngxxqnxkq4l9paaqkq4.png>

5 CONSIDERAÇÕES FINAIS

Todos os conceitos apresentados no capítulo 2 mostram a importância de se utilizar os padrões de projeto na criação de *softwares* e sistemas de informação. Com todas as etapas de padronização de um código, o resultado gerado é um sistema eficiente e de fácil manutenção, seja qual for a linguagem de programação utilizada.

Para chegar a essa conclusão, foi necessário analisar os exemplos apresentados, de códigos que não utilizam um padrão de projeto e os que utilizam algum padrão e assim mostrar que, um sistema que utiliza padrões de projeto é o melhor a se fazer para qualquer desenvolvedor de sistemas.

Foram utilizados vários exemplos no capítulo 3 de códigos, de diferentes áreas de aplicabilidade, e que mostram que qualquer sistema que se depare com qualquer tipo de problema pode ser resolvido com algum tipo de padrão de projeto. Se um novo sistema, seja qual for, estiver sendo concebido, tendo a certeza de que alguém em algum momento já implementou uma solução parecida e que pode ser aplicada neste novo projeto.

As linguagens de programação apresentadas neste trabalho, Java e Python são duas das mais utilizadas no mundo, e mostram que apesar de suas diferenças, podem adaptar-se a utilizar qualquer padrão de projeto em sua implementação. Os aspectos de verbosidade entre elas realizadas no capítulo 3 nos mostram que são linguagens com finalidades e aplicabilidades diferentes, focos diferentes de utilização, sintaxes diferentes e que ainda assim podem se utilizar os mesmos padrões de projeto para solucionar um mesmo problema, cada uma de sua forma.

Em um mundo onde, cada vez mais, sistemas de informação aparecem para solucionar novos problemas, onde surgem novas tecnologias, novas linguagens de programação, novos dispositivos, enfim, um novo mundo que se reinventa a cada dia, existem padrões de projeto de *software* para solucionar esses problemas, padrões esses que foram estudados e cunhados para facilitar a vida dos desenvolvedores deste novo mundo que o futuro nos reserva.

6 REFERÊNCIAS

ALEXANDER, Christopher, A Pattern Language. Oxford University Press, New York, 1977.

DANIEL, Glaucio. Design Patterns com Java-Entendendo Padrões de Projetos, 2021. Disponível em: <<https://www.udemy.com/course/curso-design-patterns-java/learn/lecture/23215232#questions/13732310>>.

FEATHERS, Michael C., Working Effectively with Legacy Code. Pearson; 1ª edição, 2004.

GAMMA, Erich et al. Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

GOLDENBERG, Mirian. A arte de pesquisar. Rio de Janeiro: Record, 1999.

MARTIN, Robert C., Código Limpo: Habilidades Práticas do Ágile Software. Alta Books, 2008.

NASCIMENTO, João. Uma introdução aos Padrões de Projeto, 2018. Disponível em: <<https://www.igti.com.br/blog/uma-introducao-aos-padroes-de-projeto/>>. Acessado em 19/07/2020.

PRESSMAN, Roger S ; PENTADO, Rosângela Ap. D. Engenharia de Software. Porto Alegre: AMGH, 2010.

BREWSTER, Cordenne. Python vs. Java in 2021: Side-by-Side Comparison. Disponível em: <<https://trio.dev/blog/python-vs-java>> Acesso em: 04 de abril de 2021.

SILVA, Edna; MENEZES, Estera. Metodologia da Pesquisa e Elaboração de Dissertação. 3ª ed. Universidade Federal de Santa Catarina Programa de Pós-Graduação em Engenharia de Produção Laboratório de Ensino a Distância, Florianópolis, 2001.

VERGARA, Sylvia C., Projetos e Relatórios de Pesquisa em Administração. 2ª ed. São Paulo: Atlas, 1998.

ANEXO A – Declaração de revisão ortográfica

Eu, Cassia Celeste Alvino Cruz Damasceno, professor(a) licenciado(a) em História – pela Universidade Veiga de Almeida, declaro para os devidos fins de direito que fiz a **revisão ortográfica do artigo/monografia** de Rodrigo Guilherme Cruz Damasceno intitulado(a) Padrões de projeto implementados com Java e Python: aspectos de verbosidade, apresentado(a) ao curso de Sistemas de Informação como requisito para obtenção do título de Bacharel em Sistemas de Informação.

Macaé, 08 de julho de 2021.



Assinatura do (a) Professor(a)
Licenciado(a) em História.

**PARECER TÉCNICO: TRABALHO DE CONCLUSÃO DE CURSO
(TCC II)**

CURSO: () ADMINISTRAÇÃO () ENGENHARIA DE PRODUÇÃO
(X) SISTEMAS DE INFORMAÇÃO () MATEMÁTICA

ALUNO (A): RODRIGO GUILHERME CRUZ DAMASCENO
MATR. 1701130042 DATA DA DEFESA: 01/07/2021
PROFESSOR ORIENTADOR: ISAC MENDES LACERDA
BANCA: ISAC MENDES LACERDA; ALAN CARVALHO GALANTE

TÍTULO:

PADRÕES DE PROJETO IMPLEMENTADOS COM JAVA E PYTHON: ASPECTOS DE VERBOSIDADE.

PARECER FINAL:

Em cumprimento ao Art. 9º, §5º da Deliberação nº 04/17, atesto que o (a) aluno (a) acima referido (a):

(X) atendeu às solicitações/ajustes encaminhados pela banca para validação da versão final do TCC.

() não atendeu às solicitações/ajustes encaminhados pela banca para validação da versão final do TCC.

Encaminhe-se à Secretaria Acadêmica da FeMASS, para os registros, considerando o

(a) aluno(a): (X) APROVADO () REPROVADO

Macaé, 10 de Julho de 2021

ASSINATURA DO(A) PROFESSOR(A) ORIENTADOR(A)

