UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JÚLIO CÉSAR BENELLI VARELLA

APLICAÇÃO DE SOLID EM UM SISTEMA WEB PARA INTERMEDIAÇÃO DE COMPRA E VENDA DE COMIDA NA UNESP

BAURU
Novembro/2023

JÚLIO CÉSAR BENELLI VARELLA

APLICAÇÃO DE SOLID EM UM SISTEMA WEB PARA INTERMEDIAÇÃO DE COMPRA E VENDA DE COMIDA NA UNESP

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Prof. Dr. Higor Amario de Souza

BAURU
Novembro/2023

Júlio César Benelli Varella Aplicação de SOLID em um sistema Web para intermediação de compra e venda de comida na UNESP/ Júlio César Benelli Varella. – Bauru, Novembro/2023- 59 p. : il. (algumas color.) ; 30 cm.
Orientador: Prof. Dr. Higor Amario de Souza
Trabalho de Conclusão de Curso – Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Ciência da Computação, Novembro/2023.
1. Sistemas Web 2. Engenharia de Software 3. Permanência Estudantil 4. Princípios SOLID

Júlio César Benelli Varella

Aplicação de SOLID em um sistema Web para intermediação de compra e venda de comida na UNESP

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dr. Higor Amario de Souza
Orientador
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Profª. Drª. Simone das Graças Domingues Prado
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências
Departamento de Computação

Prof. Dr. Kleber Rocha de Oliveira
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Engenharia e Ciências de Rosana
Departamento de Engenharia de Energia

Bauru, 14 de novembro de 2023

Agradecimentos

Agradeço ao meu orientador, que se fez presente e solícito durante todo o desenvolvimento do projeto, me auxiliando sempre que necessário.

Também agradeço à minha família por sempre me apoiar, pela paciência e compreensão ao longo de todos esses anos.

Resumo

A venda de alimentos por estudantes universitários tem se mostrado uma prática recorrente e vital para a manutenção financeira e suporte de projetos acadêmicos. Diante das dificuldades enfrentadas pelos estudantes para sustentar-se durante o curso, somado à necessidade de financiamento para projetos de extensão, o desenvolvimento de um sistema unificado para a comercialização de alimentos emerge como uma solução estratégica. Este trabalho propõe a criação de um sistema Web para facilitar e otimizar a venda de alimentos no campus da UNESP-Bauru, fornecendo um ponto de encontro virtual entre compradores e vendedores. Além disso, o sistema é desenvolvido com ênfase em práticas de codificação limpa e padrões de projeto, utilizando princípios SOLID e o padrão MVC para assegurar um código organizado, manutenível e expansível. A escolha do framework Symfony e do Bootstrap proporciona a funcionalidade e responsividade necessárias ao sistema. A disponibilização do código como open-source reforça a intenção de contribuir para a comunidade acadêmica, permitindo evolução contínua do projeto. Este trabalho, portanto, não só atende uma necessidade prática, mas também promove a manutenibilidade, a extensibilidade do sistema e a prática colaborativa dentro do ambiente universitário.

Palavras-chave: Sistemas Web. Engenharia de Software. Permanência Estudantil. Princípios SOLID.

Abstract

The sale of food by university students has proven to be a recurring and vital practice for financial maintenance and the support of academic projects. Faced with the challenges students encounter in sustaining themselves throughout their academic journey, coupled with the need for funding extension projects, the development of a unified system for food commerce emerges as a strategic solution. This work proposes the creation of a web system designed to facilitate and optimize the sale of food on the UNESP-Bauru campus, providing a virtual meeting point for buyers and sellers. Moreover, the system is developed with emphasis on clean coding practices and design patterns, utilizing SOLID principles and the MVC pattern to ensure organized, maintainable, and expandable code. The selection of the Symfony framework and Bootstrap offers the necessary functionality and responsiveness to the system. Making the code available as open-source reinforces the intention to contribute to the academic community, allowing for the project's continuous evolution. Therefore, this work not only addresses a practical need but also promotes the maintainability, extensibility of the system, and collaborative practice within the university environment.

Keywords: Web Systems, Software Engineering, Student Permanence, SOLID Principles.

Lista de figuras

Lista de códigos

Lista de abreviaturas e siglas

UNESP	Universidade Estadual Paulista "Júlio de Mesquita Filho"
POO	Programação Orientada a Objetos
OO	Design Orientado a Objetos
MVC	Modelo-Visão-Controlador
SRP	Princípio de Responsabilidade Única
OCP	Princípio Aberto-Fechado
LSP	Princípio de Substituição de Liskov
ISP	Princípio da Segregação de Interfaces
DIP	Princípio da Inversão de Dependência
ORM	Object Relational Mapping
MER	Modelo Entidade Relacionamento
CRUD	Create, Read, Update, Delete

Sumário

1 Introdução

Ingressar em uma universidade é uma grande ambição de milhares de brasileiros, inclusive da classe de baixa renda, que veem nesse objetivo uma oportunidade de mudar de vida e têm visto sua presença aumentar na universidade 6 vezes nos últimos 20 anos. No entanto, conseguir uma vaga é apenas a primeira etapa e se manter e terminar o curso pode ser uma jornada difícil por diversas razões como necessidade de mudar de cidade, longas jornadas em cursos integrais, e a instabilidade e dificuldade de obtenção de auxílios de permanência ou bolsas (MORAES, 2022).

Diante desse cenário, a venda de alimentos se tornou uma atividade recorrente nos campi como forma de complementar a renda dos alunos. "Recursos públicos destinados às Universidades são, na maioria das vezes, insuficientes para a demanda de projetos existentes e pela própria disputa entre as diferentes forças existentes em cada uma das Universidades" (ARAÚJO et al., 2011). Nesse contexto, o desenvolvimento de um sistema para a venda de alimentos por alunos nas universidades públicas brasileiras, com foco inicial no campus da UNESP Bauru, surge como uma solução para facilitar essa atividade, proporcionando uma solução estruturada em contraste com os canais dispersos atuais, como grupos de WhatsApp. A solução proposta possibilitaria a organização da venda de alimentos dentro do campus, além de contribuir para a geração de renda dos estudantes.

Ao longo das últimas décadas, o software evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria. Mesmo assim, o desenvolvimento de software de boa qualidade dentro do prazo e orçamento estabelecidos ainda é um problema. O software legado continua a representar desafios especiais àqueles que precisam fazer sua manutenção. A natureza do software é mutante (PRESSMAN; MAXIM, 2016). Dessa forma, a demanda por um software adaptável e escalável orientou o desenvolvimento deste projeto, que se fundamenta nos princípios SOLID e no padrão MVC. A escolha pelo *framework* Symfony e pelo Bootstrap permitiu criar um sistema funcional, responsivo e alinhado com as melhores práticas de engenharia de software.

Este trabalho enfatiza a importância da manutenibilidade e extensibilidade do sistema por meio de refatorações exemplificativas, garantindo uma arquitetura de *software* robusta e flexível. A iniciativa de disponibilizar o sistema como código aberto no GitHub visa promover a contribuição com código-aberto e a evolução contínua do projeto, reforçando seu valor de suporte à comunidade estudantil da UNESP.

1.1 Problemática

A Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP) é uma das maiores universidades públicas do Brasil, com campi em diversas cidades do estado de São Paulo. O campus de Bauru, em particular, comporta mais de 7.000 alunos. Dentre os discentes da UNESP Bauru em 2022 haviam aproximadamente 500 alunos inscritos em algum tipo de auxílio permanência (BAURU, 2022). Com a amenização da pandemia do COVID-19 e a volta das aulas presenciais, estes alunos dependem de auxílios para que seja possível se manter na universidade e muitos decidem por vender alimentos dentro do campus como uma forma de obtenção de renda. A venda de alimentos no campus torna-se uma opção atrativa de renda para aqueles que não são integralmente amparados pelos auxílios estudantis, os quais, por sua vez, demonstram instabilidades como as vivenciadas com o Restaurante Universitário (RU) em abril de 2022, quando o fornecimento de refeições foi drasticamente reduzido. Além disso, a venda de alimentos não se restringe apenas à necessidade individual, sendo também parte integrante da dinâmica de financiamento de projetos de extensão universitária, nos quais estudantes engajam-se na comercialização como meio de obter fundos para suas atividades.

Qualquer programador provavelmente já foi significativamente desacelerado pelo código confuso de outra pessoa. O grau de desaceleração pode ser significativo. Ao longo de um ou dois anos, as equipes que estavam se movendo muito rápido no início de um projeto podem acabar se movendo muito lentamente. Cada alteração feita no código quebra duas ou três outras partes do código. Não há mudança que é trivial. Cada adição ou modificação no sistema exige que os emaranhados, torções e nós sejam “compreendidos” para que mais emaranhados, torções e nós possam ser adicionados. Com o tempo, a bagunça se torna tão grande, tão profunda e tão alta que eles não conseguem limpá-la. À medida que a bagunça aumenta, a produtividade da equipe continua a diminuir, assintoticamente aproximando-se de zero (MARTIN, 2008). Portanto, a importância de práticas de codificação limpa, acompanhada de uma estrutura de código bem organizada e documentada, não pode ser subestimada. Ela é essencial para a sustentabilidade a longo prazo de um projeto de software.

1.2 Justificativa

Como destacado, a comercialização de alimentos por alunos é uma prática significativa na UNESP, essencial para a subsistência de muitos estudantes e para o financiamento de projetos de extensão. O presente trabalho propõe a criação de um sistema unificado que facilite o encontro entre compradores e vendedores de alimentos dentro da universidade, superando as limitações dos métodos atualmente descentralizados e menos eficientes. A plataforma proposta não só conectará a comunidade de maneira mais eficaz, mas também servirá como um modelo de boas práticas de desenvolvimento de software.

Durante o desenvolvimento deste software, serão aplicados conhecimentos de banco de

dados e engenharia de software, com uma aplicação dos princípios SOLID, garantindo assim um código organizado, manutenível e expansível. Isso reforça a robustez e a usabilidade da ferramenta proposta e também estabelece um padrão de qualidade que pode ser seguido e melhorado continuamente pela comunidade acadêmica.

Além disso, este trabalho se propõe a ter uma iniciativa colaborativa ao disponibilizar o sistema como código aberto, incentivando a contribuição de outros membros da comunidade da UNESP.

1.3 Objetivos

Abaixo são apresentados o objetivo geral e objetivos específicos deste trabalho.

1.3.1 Objetivo Geral

O objetivo deste projeto foi desenvolver um sistema Web para facilitar e centralizar o anúncio de venda de alimentos no campus da UNESP e colaborar com a permanência estudantil. Além disso, o código do sistema foi organizado de forma a seguir boas práticas de programção para facilitar futuras extensões e manutenções, uma vez que foi disponibilizado como código-aberto para fomentar a colaboração de outros desenvolvedores.

1.3.2 Objetivos Específicos

- Levantar e analisar requisitos relevantes ao sistema.
- Aplicar metodologias de Engenharia de Software como Padrões de Projeto e Arquitetura de Software.
- Utilizar ferramentas modernas de desenvolvimento como um *framework* Web e MVC.
- Analisar o código desenvolvido e refatorá-lo para que se encaixe nos conceitos de SOLID.
- Disponibilizar o sistema em código-aberto.

2 Fundamentação Teórica

2.1 Programação Orientada a Objetos (POO)

De acordo com Martin (2017), quando mergulha-se no universo do design de software, o *design* orientado a objetos (OO) emerge como um pilar essencial. Em sua essência, OO é frequentemente associado à integração de dados e funções, uma noção que, embora tenha raízes em práticas de programação anteriores a 1966, foi significativamente aprimorada. Muitos defendem que o OO é como um espelho do mundo real no código, proporcionando uma compreensão mais intuitiva do software. No entanto, essa representação vai além das simplificações. Os seguintes conceitos são os pilares dessa abordagem e serão explorados mais adiante:

- **Encapsulamento:** no contexto da programação orientada a objetos, refere-se à prática de restringir o acesso a certos componentes de um objeto, tornando seus atributos privados e acessíveis apenas através de métodos públicos. Essa prática permite proteger a integridade dos dados contidos no objeto, evitando manipulações indesejadas e mantendo o controle sobre como esses dados são acessados ou modificados.
- **Herança:** é um dos pilares da programação orientada a objetos. Ela permite que uma nova classe seja definida com base em uma classe existente, herdando seus atributos e métodos. Isso facilita a reutilização do código, promove uma estrutura hierárquica e representa relacionamentos do tipo "é um".
- **Polimorfismo:** é o conceito que permite que objetos de diferentes classes sejam tratados como objetos de uma mesma superclasse. É um conceito fundamental em programação orientada a objetos, permitindo que diferentes classes tenham métodos com o mesmo nome, mas com comportamentos distintos.

2.2 Arquitetura de Software

A arquitetura de um sistema de software é a forma dada a esse sistema por aqueles que o constroem. Essa forma é feita na divisão desse sistema em componentes, a disposição desses componentes e as maneiras pelas quais esses componentes comunicam-se uns com os outros.

De acordo com Martin, o objetivo principal da arquitetura é apoiar o ciclo de vida do sistema. Uma boa arquitetura torna o sistema fácil de entender, fácil de desenvolver, fácil de manter, e fácil de implantar. O objetivo final é minimizar o custo da vida útil do sistema e maximizar a produtividade do programador (MARTIN, 2017).

2.2.1 Arquitetura Modelo-Visão-Controlador (MVC)

A arquitetura MVC (do inglês *Model-View-Controller*) separa as camadas do programa em diferentes arquivos com funções bem definidas. A divisão facilita o entendimento do código e, consequentemente, diminui a complexidade e aumenta a produtividade no desenvolvimento de novas funcionalidades e na manutenção do sistema.

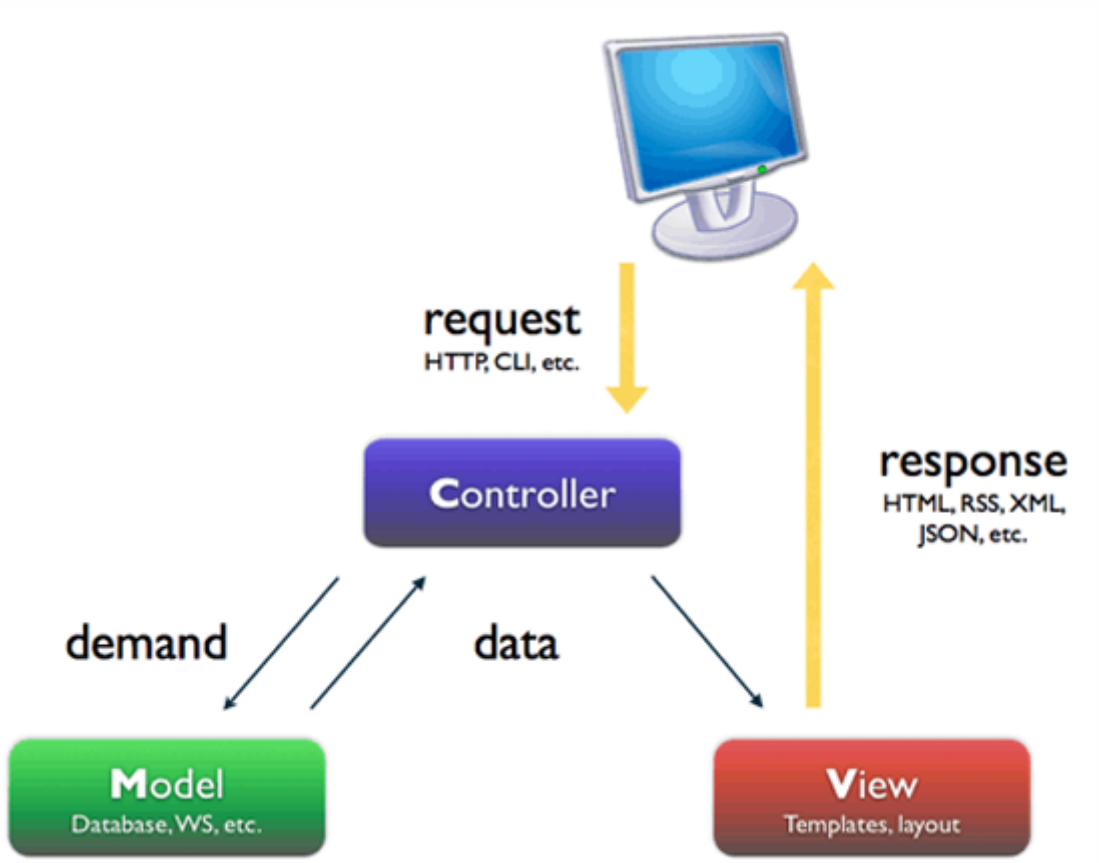
As diferentes camadas dessa arquitetura são:

- *Model* - A camada Modelo é responsável pela lógica de negócios de uma aplicação. Ele irá encapsular métodos para acessar dados (bancos de dados, arquivos, etc.) e disponibilizará uma biblioteca de classes reutilizável. Normalmente um modelo é construído com abstração de dados em mente, validação e autenticação.
- *Controller* - O Controlador é responsável pelo tratamento de eventos. Esses eventos podem ser acionados por um usuário interagindo com aplicativo ou por um processo do sistema. Um controlador aceita solicitações e prepara os dados para uma resposta. O Controlador interage com o Modelo para recuperar os dados necessários e gera a Visualização. Também é responsável por estabelecer o formato da resposta.
- *View* - A *View* é responsável pelo gerenciamento da interface gráfica do usuário. Isso significa todos os formulários, botões, gráficos elementos e todos os outros elementos HTML que estão dentro do aplicativo.

A arquitetura MVC pode ser descrita como um ciclo natural que ocorre entre o software e o usuário. Nesse processo, o usuário inicia uma ação, e como resposta, o aplicativo modifica seu modelo de dados e apresenta uma visão atualizada ao usuário (POP; ALTAR, 2014).

Este ciclo está representado na Figura 1. O usuário acessa a View e gera requisições que são tratadas pelo Controller que, por sua vez, se comunica com a camada Model para fazer as alterações requisitadas pelo usuário e transferir a resposta dessa alteração para a View atualizada.

Figura 1 – Diagrama da Arquitetura MVC para uma aplicação Web



Fonte: *StackOverflow*, 2011 [1]

2.3 SOLID

Os princípios SOLID nos orientam sobre a melhor maneira de organizar nossas funções e estruturas de dados em agrupamentos, normalmente referidos como classes, e como esses agrupamentos devem se relacionar entre si. Essas orientações não são exclusivas para a programação orientada a objetos. Uma classe pode ser vista simplesmente como um conjunto de funções e dados. Todo sistema de software possui tais conjuntos, independentemente de serem chamados de classes ou não. Esses princípios visam criar estruturas de software que se adaptam às mudanças, são claras e compreensíveis e servem como base para componentes reutilizáveis em diversos sistemas (MARTIN, 2017).

2.3.1 Princípio de Responsabilidade Única

O Princípio de Responsabilidade Única (SRP, do inglês *Single Responsibility Principle*) estabelece que uma classe ou módulo deve ter um, e apenas um, motivo para mudar. Códigos

[1] https://stackoverflow.com/questions/5966905/what-is-the-right-mvc-diagram-for-a-web-application

que servem diferentes propósitos, como interface gráfica, consultas SQL ou regras de negócio, não devem se misturar. Todo código que é alterado por diferentes razões deve ser separado, para que as alterações em uma parte não quebrem as demais (MARTIN, 2020).

Essa separação de responsabilidades se faz importante porque frequentemente os requisitos de um programa mudam e isso é refletido nas responsabilidades das classes. Portanto, se uma classe assume mais de uma responsabilidade, terá mais de uma razão para ser alterada, gerando acoplamento. Essa estrutura leva a um programa propenso a falhas quando modificado (MARTIN, 2003).

Em "Clean Architecture", Martin apresenta um exemplo ilustrando o Princípio da Responsabilidade Única com base na classe Employee de um sistema de pagamentos. Esta classe possui três métodos distintos: calculatePay(), reportHours() e save():

Código 1 – Exemplo de código PHP que não segue o SRP

```
class Employee {

    private $id;
    private $name;

    public function __construct($id, $name) {
        $this->id = $id;
        $this->name = $name;
    }

    public function calculatePay() {
        // Logica para calcular o salario do empregado
    }

    public function reportHours() {
        // Logica para reportar as horas trabalhadas
    }

    public function save() {
        // Logica para salvar informacoes do empregado
        // no banco de dados
    }
}
```

Fonte: (MARTIN, 2017)

Ao incluir esses três métodos na mesma classe, houve uma junção de responsabilidades destinadas a diferentes atores. O método calculatePay() seria de interesse do setor financeiro,

reportHours() do setor de operações e save() seria de responsabilidade dos administradores de banco de dados. Essa estrutura resulta em um acoplamento, pois qualquer modificação solicitada por um setor poderia afetar o outro.

Refatorando o código para adequar-se ao SRP:

Código 2 – Exemplo de código refatorado PHP seguindo o SRP

```
class Employee {
    private $id;
    private $name;

    public function __construct($id, $name) {
        $this->id = $id;
        $this->name = $name;
    }

    public function calculatePay() {
        // Logica para calcular o salario do empregado
    }
}

class EmployeeReport {
    public function reportHours(Employee $employee) {
        // Logica para reportar as horas trabalhadas
    }
}

class EmployeeRepository {
    public function save(Employee $employee) {
        // Logica para salvar informacoes do empregado
        // no banco de dados
    }
}
```

Fonte: elaborado pelo autor

Dessa forma, cada classe tem uma única responsabilidade, evitando acoplamento (MARTIN, 2017).

2.3.2 Princípio Aberto-Fechado

O Princípio Aberto-Fechado (OCP, do inglês *The Open-Closed Principle*) sintetiza que entidades de software, como classes ou módulos, devem estar abertas para extensão,

mas fechadas para modificação. Em outras palavras, o sistema deve permitir alterações como inclusões de novas funcionalidades ou comportamentos, que deverão ser inseridas sem alterar o código previamente desenvolvido e testado. O objetivo é fazer o sistema passível de modificação sem que as alterações dos módulos de nível inferior impactem os módulos de nível superior, promovendo a modularidade e robustez do software (MARTIN, 2017).

Em termos práticos, esse princípio encoraja o uso de interfaces para evitar a modificação de classes existentes.

Uma situação comumente utilizada para ilustrar o OCP é a de um programa responsável por desenhar formas geométricas. Abaixo, o código em C++ que não segue o princípio:

Código 3 – Exemplo de código que não segue o OCP

```
// shape.h
enum ShapeType {circle, square};

struct Shape {
  ShapeType itsType;
};

// circle.h
struct Circle {
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};

void DrawCircle(struct Circle*);

// square.h
struct Square {
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

void DrawSquare(struct Square*);

// drawAllShapes.cc
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n) {
  int i;
```

```
  for (i=0; i<n; i++) {
    struct Shape* s = list[i];
    switch (s->itsType) {
    case square:
      DrawSquare((struct Square*)s);
      break;
    case circle:
      DrawCircle((struct Circle*)s);
      break;
    }
  }
}
```

Fonte: (MARTIN, 2017)

No exemplo do código 3, a função *DrawAllShapes()* confere a forma requisitada para a ser desenhada dentro da estrutura de repetição *switch*, isso faz com que ela não se encaixe no OCP, pois para qualquer nova forma a ser adicionada no programa, um novo caso (*case*) seria inserido à estrutura *switch*. Isso resulta na alteração do código existente toda vez que uma nova forma é introduzida, contrariando o OCP (MARTIN, 2003).

Para adequar o código ao OCP, a refatoração a ser feita aplicaria o conceito de polimorfismo. A proposta seria criar uma interface *Shape* com o método *draw()*, que seria implementado por todas as formas. Esse método seria diferente para cada classe que a implementasse, seguindo o cálculo específico da respectiva forma geométrica. As estruturas condicionais também seriam removidas, e seria possível percorrer uma lista das formas invocando apenas o método *draw()*.

2.3.3 Princípio de Substituição de Liskov

O Princípio de Substituição de Liskov (LSP, do inglês "*Liskov Substitution Principle*") se refere à capacidade de objetos de uma superclasse serem substituídos por objetos de uma subclasse sem afetar a correção do programa. Em termos formais, Barbara Liskov definiu o princípio da seguinte forma: 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 permanece inalterado quando o1 é substituído por o2. Assim, S é considerado um subtipo de T. O LSP enfatiza a importância das relações de herança e polimorfismo para garantir a integridade e extensibilidade de sistemas de software.

Um exemplo frequentemente citado para ilustrar o LSP é o problema do retângulo e do quadrado. Imagine uma classe Retangulo e uma subclasse derivada chamada Quadrado. Embora, na geometria, um quadrado seja um tipo especial de retângulo, no contexto da programação

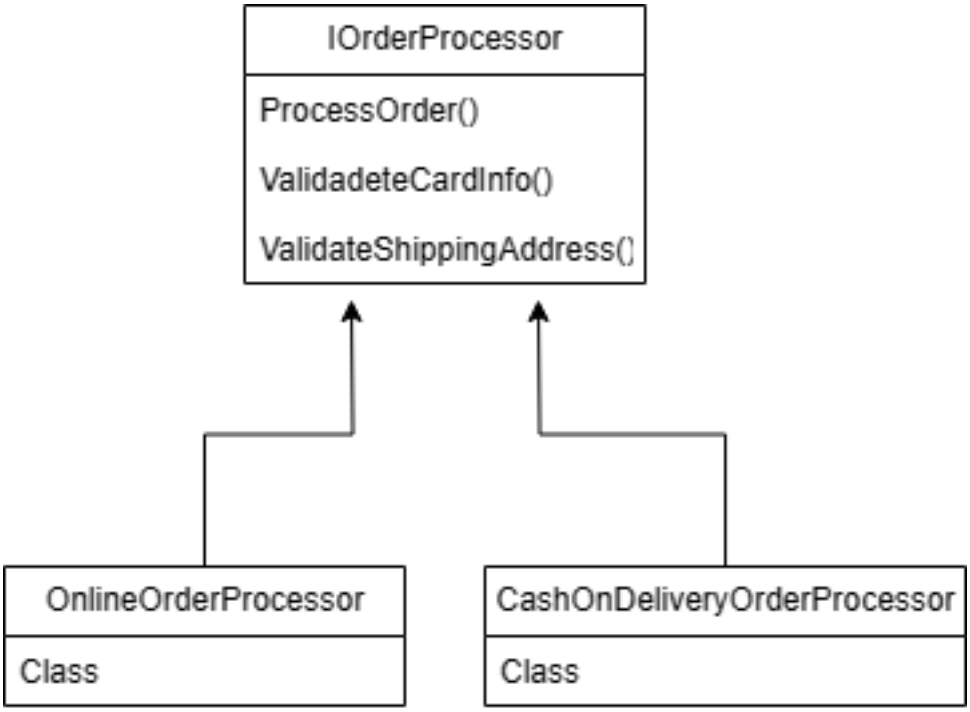
orientada a objetos, considerar um Quadrado como subtipo de Retangulo pode ser problemático. Isso ocorre porque, em um retângulo, a altura e a largura são independentes, enquanto no quadrado elas são iguais. Se uma classe Usuario espera trabalhar com um Retangulo e alterar sua altura e largura independentemente, essa expectativa seria violada ao lidar com um Quadrado. Esta discrepância pode levar a erros e confusões no programa (MARTIN, 2017). Portanto, neste caso, o Quadrado não seria um substituto adequado para Retangulo, indicando uma violação do LSP.

2.3.4 Princípio da Segregação de Interfaces

A principal ideia do Princípio da Segregação de Interfaces (ISP, do inglês "*Interface Segregation Principle*") é garantir que uma classe não seja forçada a implementar métodos que não utilizará.

O princípio pode ser exemplificado utilizando o contexto de um *website* de um *e-commerce*. Inicialmente, imagina-se que o *site* aceitará apenas pagamentos com cartão, portanto uma interface "IOrderProcessor" é criada com 3 métodos: "ValidateCardInfo()" para validar informações do cartão; "ValidateShippingAddress()" para confirmar o destino da entrega; e "ProcessOrder()" para processar do pedido. Uma nova classe "OnlineOrderProcessor" implementa estes métodos para lidar com pedidos online. Posteriormente, pagamento em dinheiro começa a ser aceito pelo *site* e então uma classe "CashOnDeliveryProcessor" é criada implementando a interface "IOrderProcessor". No entanto, por não precisar validar os dados do cartão de crédito, essa classe apenas lança uma exceção no método "ValidateCardInfo()". Essa disposição de classes é ilustrada na Figura 2:

Figura 2 – Diagrama de classes que violam o ISP

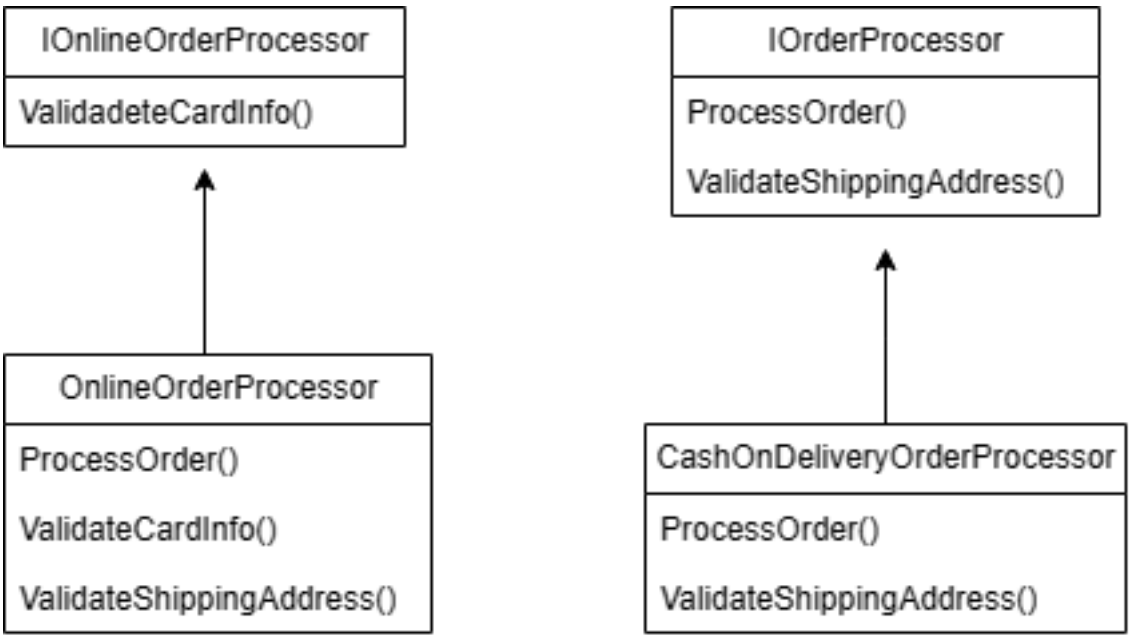


Fonte: *Medium*, 2023[2]

Dessa forma, se algum passo adicional seja necessário nos pagamentos online, novos métodos serão criados na interface "IOrderProcessor" e implementadas por "OnlineOrderProcessor", mas a classe "CashOnDeliveryOrderProcessor" também teria que ser alterada para implementar os novos métodos e lançar exceções, pois não serão necessários, violando o ISP (JOSHI, 2016). A disposição das classes refatorada é ilustrada na Figura 3:

[2] https://medium.com/@heshan.qiu/reposting-a-primer-on-the-practical-usage-of-solid-principles-895f039eb374

Figura 3 – Diagrama das classes refatoradas de acordo com o ISP



Fonte: *Medium*, 2023[3]

2.3.5 Princípio da Inversão de Dependência

O Princípio da Inversão de Dependência, conhecido também pela sigla DIP (do inglês "*Dependency Inversion Principle*") é essencial para promover a desacoplagem em sistemas, tornando-os mais flexíveis, fáceis de modificar e extensíveis.

O DIP pode ser sintetizado em duas sentenças:

- As classes de alto nível não devem depender das classes de baixo nível. Ambas deve depender de abstrações.
- As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.

Uma classe de alto nível é uma classe que faz algo significativo na aplicação, enquanto uma classe de baixo nível é uma classe que faz algum trabalho auxiliar. Usando de exemplo uma aplicação *web* contendo usuários, a classe *EmailNotifier* que envia um *email* para recuperação da conta é de baixo nível, enquanto a classe *UserManager* que gerencia o usuário é de alto nível.

O código 4 mostra como a classe UserManager pode depender de EmailNotifier:

[3] https://medium.com/@heshan.qiu/reposting-a-primer-on-the-practical-usage-of-solid-principles-895f039eb374

Código 4 – Exemplo de código que não segue o DIP

```
public void ChangePassword(string username, string oldpwd, string newpwd)
{
    EmailNotifier notifier = new EmailNotifier();
    //change password here
    notifier.Notify("Password was changed on " + DateTime.Now);
}
```

Fonte: Medium, 2023.[4]

Antes de invocar o método Notify(), o método ChangePassword instancia EmailNotifier, evidenciando o alto acoplamento entre as duas classes e violando o DIP. Sempre que ocorrem mudanças no EmailNotifier, pode ser necessário fazer ajustes no UserManager. Assim sendo, as classes mais básicas devem ser terminadas antes escrever as classes mais complexas. Adicionalmente, se decidirmos fazer alterações no método de notificação no futuro, como trocar notificações por e-mail por notificações via SMS, isso exigirá mudanças no código do UserManager para acomodar a nova classe de notificação.

A solução para essa implementação que viola o DIP seria criar uma interface INotifier, que seria implementada pela classe de notificação desejada. Essa classe seria inserida no construtor da classe UserManager, e essa instância seria usada para chamar o método Notify(). Dessa forma, ao trocar o método de notificação de *email* para SMS, a classe UserNotifier não seria alterada, pois a instância utilizada para enviar a notificação foi passada no construtor da classe, invertendo a direção da dependência (JOSHI, 2016).

2.4 *Framework*

Frameworks são conjuntos de classes que colaboram para formar um *design* reutilizável para um tipo específico de software. Eles são customizados para aplicações específicas através da criação de subclasses de classes abstratas do próprio *framework* e definem a arquitetura da aplicação, incluindo estrutura, divisão em classes e objetos, responsabilidades chave, colaboração entre classes e objetos, e o fluxo de controle. Além disso, predetermina esses parâmetros de *design*, permitindo que os desenvolvedores se concentrem nos aspectos específicos de suas aplicações. *Frameworks* enfatizam a reutilização de *design* mais do que a reutilização de código, embora normalmente incluam subclasses concretas para uso imediato.

O uso de *frameworks* leva a uma inversão de controle entre a aplicação e o software base, isso reduz as decisões de *design* que você precisa fazer. As aplicações construídas com *frameworks* tendem a ser mais rápidas de desenvolver, manter e parecem mais consistentes

[4] https://medium.com/@heshan.qiu/reposting-a-primer-on-the-practical-usage-of-solid-principles-895f039eb374

para os usuários. No entanto, isso pode limitar a liberdade criativa, já que muitas decisões de *design* são pré definidas (GAMMA et al., 1994).

A essência de um *framework* é evitar a repetição de tarefas e eliminar atividades de baixo valor agregado. Por exemplo, em vez de passar dias criando um formulário de autenticação, os desenvolvedores podem usar funcionalidades prontas do *framework* e focar em componentes mais específicos, garantindo código de alta qualidade (SYMFONY, 2023b).

2.5 *Marketplace*

Os *Marketplaces* representam espaços que conectam diversos vendedores, tanto em ambientes físicos quanto online. Com a evolução da tecnologia, especialmente da internet, surgiram os *e-marketplaces*. Para fornecedores, esses ambientes online representam uma forma eficiente de publicidade e redução de custos operacionais e financeiros. Já para os compradores, eles simplificam o processo de seleção de fornecedores, proporcionando mais opções, facilitando o acesso a informações sobre produtos e eliminando intermediários tradicionais no processo de compra e venda (SMITH; REIS, 2023).

3 Ferramentas

Nesta seção são apresentadas as ferramentas escolhidas para o desenvolvimento do projeto.

3.1 PHP

O PHP (um acrônimo recursivo para PHP: Hypertext Preprocessor) é uma linguagem de *script open source* de uso geral, muito utilizada, e especialmente adequada para o desenvolvimento web e que pode ser embutida dentro do HTML. O PHP é focado principalmente nos scripts do lado do servidor, portanto, você pode fazer qualquer coisa que outro programa CGI pode fazer, como coletar dados de formulários, gerar páginas com conteúdo dinâmico ou enviar e receber *cookies* (PHP, 2023).

3.2 Symfony

Symfony é um *framework* PHP de código aberto para desenvolvimento web. Ele foi projetado para otimizar a criação e manutenção de aplicativos web, permitindo aos desenvolvedores reciclar o código e evitar a reescrita de funções comuns (SYMFONY, 2023a).

Pode-se destacar algumas vantagens desse *framework*:

- Desenvolvimento rápido: A maior vantagem de usar um *framework* como o Symfony é a aceleração do processo de desenvolvimento. Componentes pré-criados, como formulários, autenticação e roteamento, permitem aos desenvolvedores não perderem tempo reescrevendo funcionalidades comuns.
- Extensibilidade: O Symfony opera no princípio de *bundles*, que são semelhantes a *plugins* ou pacotes. Cada *bundle* pode ser reutilizado em diferentes projetos ou compartilhado com a comunidade.
- Desempenho: O Symfony é construído com desempenho em mente, utilizando várias técnicas, como carregamento lento (*lazy-loading*), para garantir que os aplicativos construídos com ele sejam rápidos.
- Manutenibilidade: O Symfony segue os padrões de *design* de *software* e princípios SOLID, garantindo que o código seja modular e fácil de manter.
- Segurança: O *framework* tem uma série de medidas integradas para proteger os aplicativos contra ameaças comuns na web.

3.3 Twig

Twig é um moderno sistema de *templates* para PHP. É flexível, rápido e seguro. Desenvolvido como um projeto independente, o Twig é usado por muitos sistemas PHP, incluindo o Symfony, para facilitar a renderização de visões e *templates* (TWIG, 2023). Dentre sua principais características estão:

- Flexibilidade: Twig tem uma sintaxe concisa e limpa que permite expressar *templates* de maneira clara. Com ele, é possível ter controle total sobre a marcação, garantindo que seu HTML/CSS/JS não seja afetado de maneira adversa.
- Extensibilidade: O Twig é altamente extensível, permitindo que os desenvolvedores definam funções, filtros e *tags* personalizados, expandindo assim suas capacidades de acordo com as necessidades específicas do projeto.
- Desempenho: O Twig compila *templates* em classes PHP otimizadas, garantindo uma execução rápida e reduzindo a sobrecarga no tempo de execução. Uma vez compilado, o *template* é armazenado em cache, melhorando ainda mais o desempenho.
- Herança de *Templates*: Uma característica poderosa do Twig é a herança de *templates*. Isso permite que você defina um *layout* base e depois estenda ou substitua partes dele em *templates* derivados. Isso facilita a manutenção e a reutilização do código.

3.4 Doctrine

Doctrine ORM é um mapeador objeto-relacional (ORM) para PHP 7.1+ que fornece persistência transparente para objetos PHP. Ele utiliza o padrão Data Mapper em sua essência, visando uma completa separação da sua lógica de domínio/negócios da persistência em um sistema de gerenciamento de banco de dados relacional.

O benefício do Doctrine para o programador é a capacidade de se concentrar na lógica de negócios orientada a objetos e se preocupar com a persistência apenas como um problema secundário. Isso não significa que a persistência é menosprezada pelo Doctrine 2; no entanto, acreditamos que existem benefícios consideráveis para a programação orientada a objetos se a persistência e as entidades forem mantidas separadas.

3.5 Git

Git é um sistema de controle de versão distribuído gratuito e de código aberto projetado para gerenciar desde pequenos até projetos muito grandes com rapidez e eficiência. Esse sistema permite que desenvolvedores rastreiem e gerenciem as alterações no código ao longo do tempo, colaborando com outros profissionais de maneira mais ágil e organizada. Graças

à sua capacidade distribuída, o Git também oferece a vantagem de permitir múltiplas cópias locais de um projeto, o que facilita a integração de diferentes alterações e contribuições em um repositório central (GIT, 2023).

3.6 MySQL

MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto amplamente utilizado em todo o mundo. Reconhecido por sua rapidez e confiabilidade, ele é utilizado para armazenar dados para uma variedade de aplicações, desde websites simples até aplicativos corporativos complexos. Sua estrutura permite que os usuários criem, atualizem, leiam e excluam dados em tabelas usando a linguagem de consulta SQL. Além disso, o MySQL tem uma reputação de ser seguro e robusto, com capacidades que garantem a integridade dos dados e otimizam o desempenho. Muitas organizações escolhem o MySQL devido à sua flexibilidade, seu licenciamento de código aberto e sua capacidade de se integrar facilmente com uma variedade de linguagens de programação e sistemas operacionais (MYSQL, 2023).

4 Desenvolvimento

Neste capítulo, será detalhado o processo de desenvolvimento do projeto, abrangendo desde a modelagem inicial até a refinação do sistema seguindo os princípios SOLID. O planejamento prévio foi crucial para estabelecer uma fundação sólida para o sistema. Estas fases preparatórias garantiram a clara definição dos requisitos e a estruturação adequada do banco de dados, minimizando potenciais obstáculos e retrabalhos no decorrer do desenvolvimento. Tais etapas são fundamentais e se mantêm constantes, independentemente das ferramentas tecnológicas selecionadas.

4.1 Requisitos

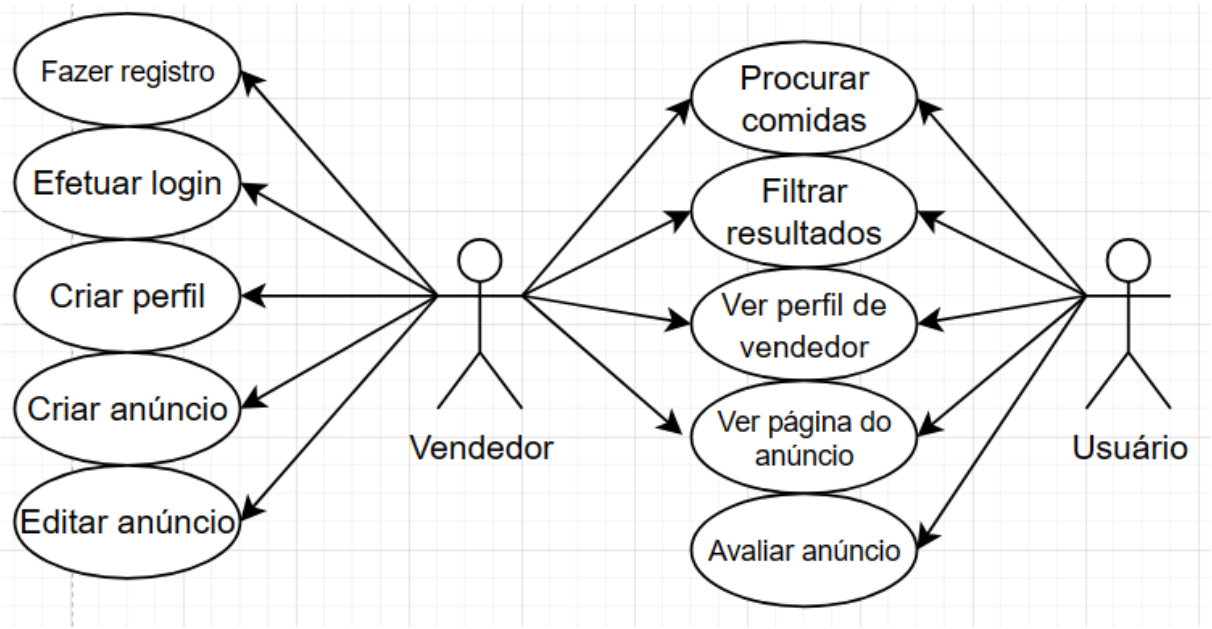
A primeira etapa no desenvolvimento foi a definição dos requisitos. Mais do que somente avaliar as ferramentas tecnológicas e o cronograma disponível, foi importante compreender as demandas do público que o projeto visa atender. Assim, um questionário foi desenvolvido e distribuído à comunidade da UNESP, incluindo alunos, docentes e funcionários, por meio do Grupo Administrativo do Campus da UNESP Bauru. O objetivo era entender as dificuldades atuais na comercialização de alimentos na instituição, bem como coletar sugestões e antecipar expectativas para o novo sistema.

Analisando os principais canais de anúncios existentes, como grupos do WhatsApp e cartazes espalhados pelo campus, foram identificadas características predominantes sobre os produtos ofertados. Com base nestas observações, estabeleceram-se os requisitos essenciais:

- Sistema de autenticação para criar anúncios;
- Criação de um perfil de vendedor;
- Capacidades de inserção, edição e remoção de itens no perfil de vendedor;
- Apresentação de itens disponíveis com imagens e descrições detalhadas;
- Filtragem por categorias de anúncios;
- Indicação da localização dos pontos de venda;
- Interface intuitiva e adaptável a diferentes dispositivos;
- Sistema de avaliação dos produtos oferecidos pelos vendedores.

Assim, a partir dos requisitos apresentados, os casos de usos se estabeleceram como demonstrado na Figura 4:

Figura 4 – Casos de Uso



Fonte : Elaborado pelo autor.

4.2 Modelagem do banco de dados

Após a definição dos requisitos, deu-se início à etapa de modelagem do banco de dados. Nessa fase, interpretaram-se os requisitos, transpondo-os para uma estrutura relacional, e optou-se pelo MySQL como sistema de gerenciamento do banco de dados para efetuar essa modelagem. As entidades foram mapeadas em tabelas, e seus atributos e propriedades traduzidos em colunas, resultando na seguinte configuração:

- Vendedor: id, nome, email, senha, localização, horário;
- Item: id, vendedor_id, nome, descrição, caminho_foto;
- Categoria: id, item_id, nome;
- Usuário: id, nome, email, senha;
- Avaliações: id, item_id, user_id, nota;

Para representar os principais atores e objetos do sistema, foram estabelecidas entidades para o Vendedor e para o Usuário. A entidade Vendedor mantém uma relação de 1 para N com a entidade Item, indicando que um único vendedor pode criar e gerenciar múltiplos anúncios. Esta relação é estabelecida por meio de chaves estrangeiras na entidade Item, que referenciam o vendedor que criou o anúncio.

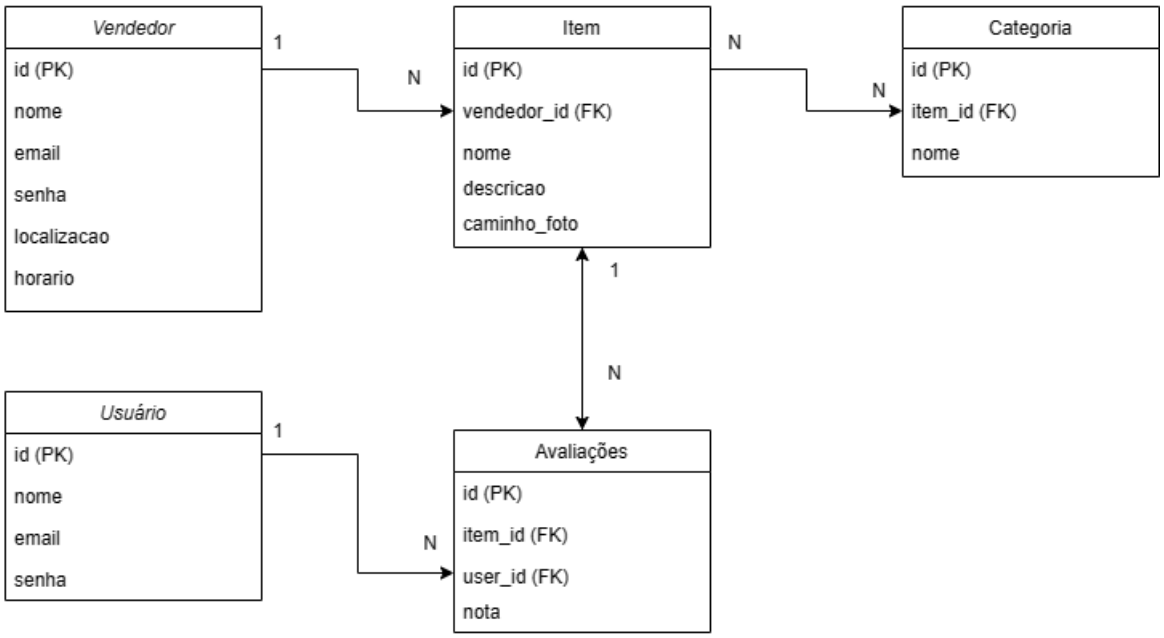
Por outro lado, a entidade Item estabelece uma relação de N para N com a entidade Categoria. Esta relação bidirecional sugere que um item pode ser associado a múltiplas

categorias, e uma categoria pode agrupar vários itens. Para implementar essa relação, é típico usar uma tabela de junção que contém chaves estrangeiras tanto para Item quanto para Categoria.

Além disso, a entidade Usuário mantém uma relação de 1 para N com a entidade Avaliações, o que sugere que um único usuário pode fazer várias avaliações, mas cada avaliação é única para um usuário e um item. Esta relação é implementada com chaves estrangeiras na entidade Avaliações referenciando o Usuário que a fez. Da mesma forma, a entidade Item tem uma relação de 1 para N com Avaliações, onde cada item pode acumular múltiplas avaliações dos usuários, novamente evidenciado pela presença de chaves estrangeiras na tabela Avaliações apontando para o Item avaliado

O Modelo Entidade Relacionamento (MER) da Figura 5 representa como as entidades, relacionamentos e atributos ficaram definidos:

Figura 5 – MER do projeto



Fonte : Elaborado pelo autor.

Em seguida, antes de começar o desenvolvimento do sistema, definiu-se a ferramenta utilizada.

4.3 Escolha das Tecnologias e Justificativa

Na etapa inicial de planejamento e desenvolvimento, a escolha de uma tecnologia apropriada é fundamental para o sucesso do projeto. Dentre as opções disponíveis, optou-se pelo Symfony como a ferramenta principal para o desenvolvimento do sistema.

4.3.1 Por que Symfony?

Várias razões influenciaram esta decisão:

- **Arquitetura MVC**: O Symfony é amplamente reconhecido por sua robusta implementação do padrão Modelo-Visão-Controlador (MVC), que promove uma separação clara das lógicas de negócios, apresentação e controle. Esta característica facilita a manutenção, evolução e compreensão do sistema.
- **Conhecimento Prévio**: A familiaridade do autor com o Symfony proporciona uma curva de aprendizado reduzida, permitindo um desenvolvimento mais ágil e confiável.
- **Adesão ao SOLID**: A estrutura do Symfony naturalmente adere aos princípios SOLID, garantindo que o código seja modular, extensível e de fácil manutenção.
- **Geração de Código**: O Symfony oferece ferramentas que permitem a geração de código através de comandos, agilizando e padronizando o desenvolvimento.
- ***Open Source* e Comunidade Ativa**: Sendo uma ferramenta de código aberto, o Symfony possui uma comunidade ativa. Esta comunidade contribui com melhorias constantes, *plugins*, bibliotecas e uma documentação extensa e detalhada.

4.3.2 Symfony vs Laravel

Durante o processo de seleção de tecnologia, o Laravel também foi considerado, dada sua popularidade e robustez. Entretanto, algumas distinções entre as duas ferramentas culminaram na preferência pelo Symfony:

- **Flexibilidade vs Conveniência**: Enquanto o Symfony é frequentemente elogiado por sua flexibilidade e modularidade, permitindo que os desenvolvedores tenham mais controle sobre a estrutura e componentes do projeto, o Laravel é muitas vezes apontado por sua conveniência e rapidez no desenvolvimento, com muitas funcionalidades "prontas para uso".
- **Componentes**: O Symfony é conhecido por seus componentes reutilizáveis, que podem ser usados não apenas dentro do próprio Symfony, mas em outros projetos PHP. O Laravel, por outro lado, é mais coeso e integrado, oferecendo uma experiência mais uniforme, mas potencialmente menos modular.
- **Configuração**: O Symfony adota uma abordagem de configuração explícita, onde os desenvolvedores devem especificar suas intenções de forma clara. O Laravel, por outro lado, se inclina mais para as convenções, reduzindo a necessidade de configuração em muitos casos.

Dadas estas considerações e o contexto específico deste projeto, o Symfony foi a escolha mais alinhada com os objetivos e requisitos estabelecidos.

4.4 Configuração do Ambiente de Desenvolvimento

A etapa inicial do desenvolvimento prático do sistema foi a configuração do ambiente de desenvolvimento. Este processo é essencial para garantir que todas as ferramentas necessárias estejam devidamente instaladas e configuradas para facilitar o desenvolvimento, testes e posterior manutenção do software.

4.4.1 Instalação e Configuração do Symfony

O Symfony foi instalado através do Composer, um gerenciador de dependências para PHP. Com o Composer, foi possível instalar o Symfony e suas dependências no projeto com um único comando.

4.4.2 Criação do Banco de Dados

Após a instalação, o arquivo .env do projeto foi atualizado para configurar a conexão com o banco de dados, inserindo as credenciais e o nome do banco de dados desejado. Neste arquivo, foram especificadas as credenciais e o nome do banco de dados utilizado pelo projeto. A linha relevante do arquivo .env foi configurada da seguinte forma:

```
DATABASE_URL="mysql://root:arca7@127.0.0.1:3306/food?serverVersion=8"
```

A *string* de conexão acima informa ao Symfony os detalhes para conectar-se ao MySQL, indicando o usuário (root), a senha (arca7), o endereço do servidor de banco de dados (127.0.0.1), a porta (3306), o nome do banco de dados (food) e a versão do servidor MySQL (8).

Com a estrutura de configuração pronta, o próximo passo foi criar o banco de dados. O Doctrine ORM foi utilizado, que facilita a gestão da base de dados através de comandos. O comando para criar o banco de dados foi o seguinte:

```
php bin/console doctrine:database:create
```

Este comando instrui o Doctrine a criar um novo banco de dados conforme as configurações definidas no arquivo .env. Com o banco de dados criado, procedeu-se para definir o esquema de banco de dados baseado nas entidades mapeadas pelo Doctrine:

```
php bin/console doctrine:schema:update --force
```

Assim, o Doctrine gerou e executou as instruções SQL necessárias para estruturar o banco de dados em conformidade com as entidades do projeto.

Com o ambiente de desenvolvimento devidamente configurado e o banco de dados criado, iniciou-se a implementação das funcionalidades do sistema.

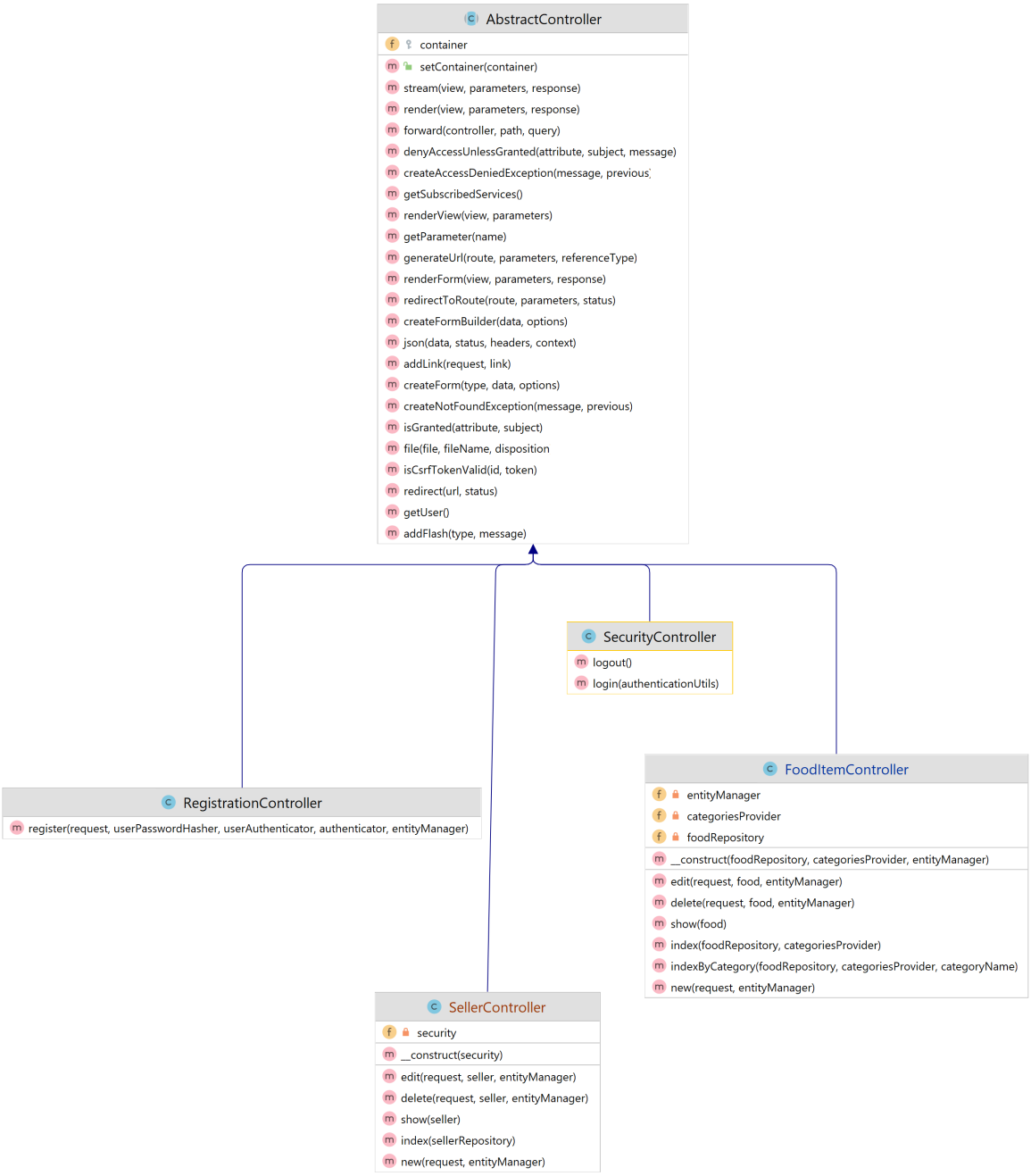
4.4.3 Estrutura de Diretórios e Arquitetura

Nessa etapa de desenvolvimento do trabalho, foi explorada a organização e estruturação do projeto dentro do *framework* Symfony, que segue uma arquitetura bem definida e organizada, facilitando o desenvolvimento e manutenção do código.

A pasta src/ contém o principal código da aplicação Symfony, onde fica o código-fonte PHP, dividido em pastas que representam os componentes do padrão MVC:

- Controller/: Armazena os controladores, responsáveis pela lógica de interação com o usuário. As classes contidas nesse diretório estão representadas no diagrama da Figura 6:

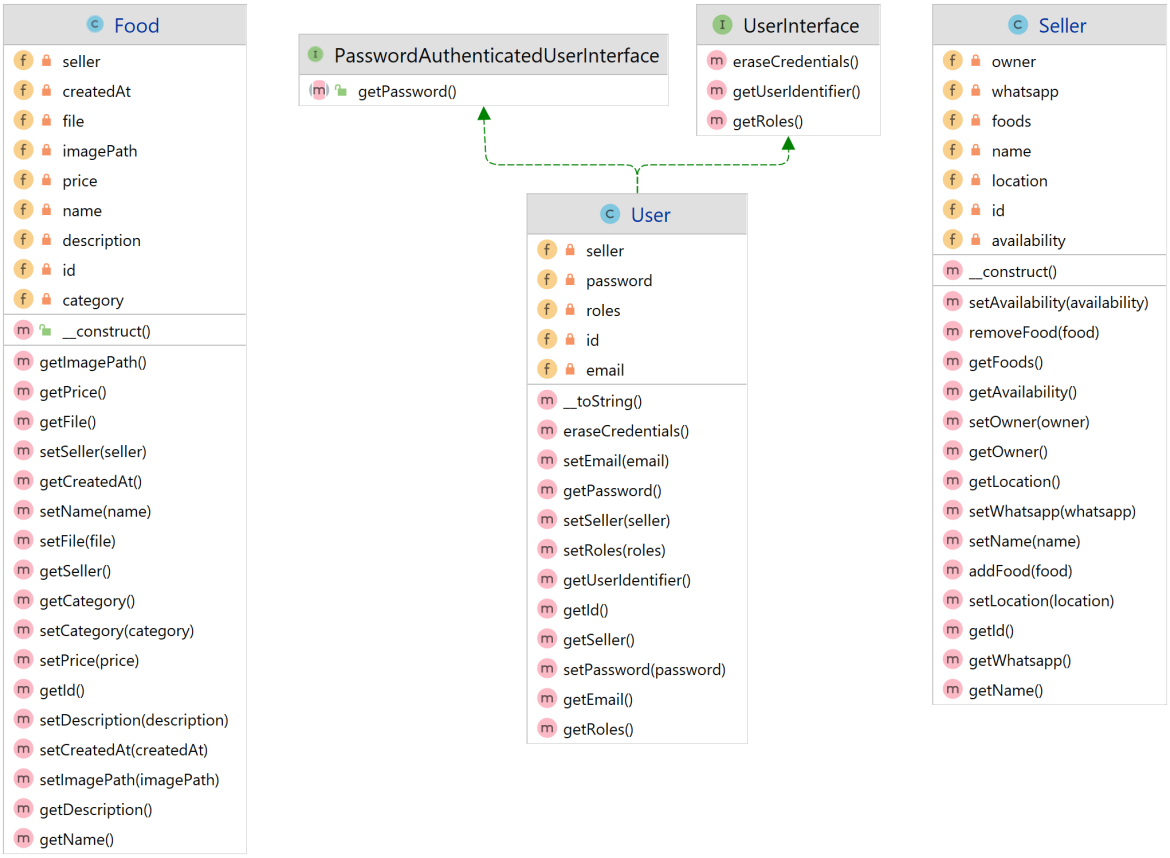
Figura 6 – Diagrama de classes de Controller



Fonte: Elaborado pelo autor.

- Entity/: Define as entidades do sistema, que são usados para gerar as migrations, que executam os comandos SQL para alterar o banco de dados. Suas classes estão ilustradas na Figura 7:

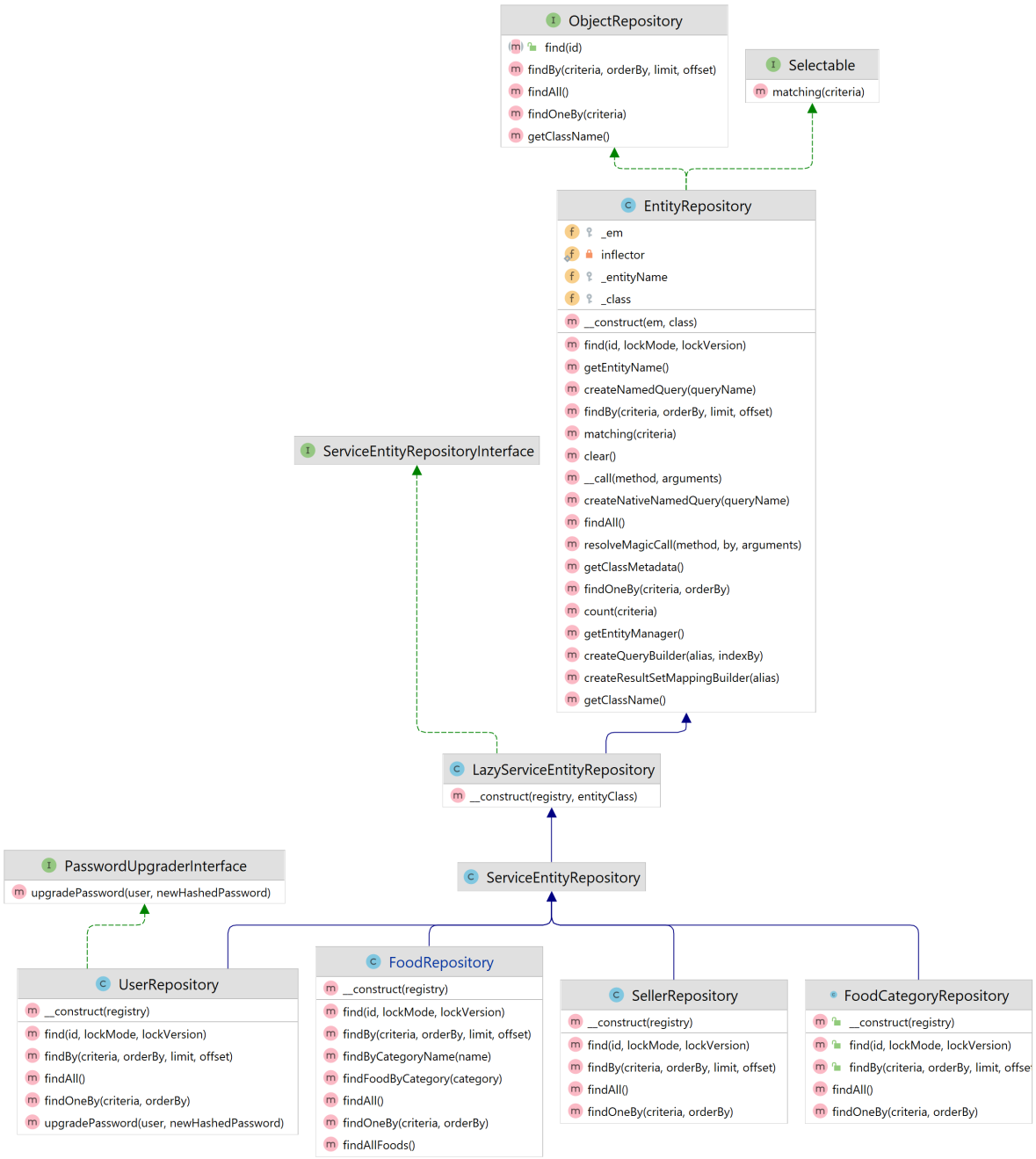
Figura 7 – Diagrama de classes de Entity



Fonte: Elaborado pelo autor.

- Repository/: Contém os repositórios, que servem para manipular os dados das entidades, destacados na Figura 8:

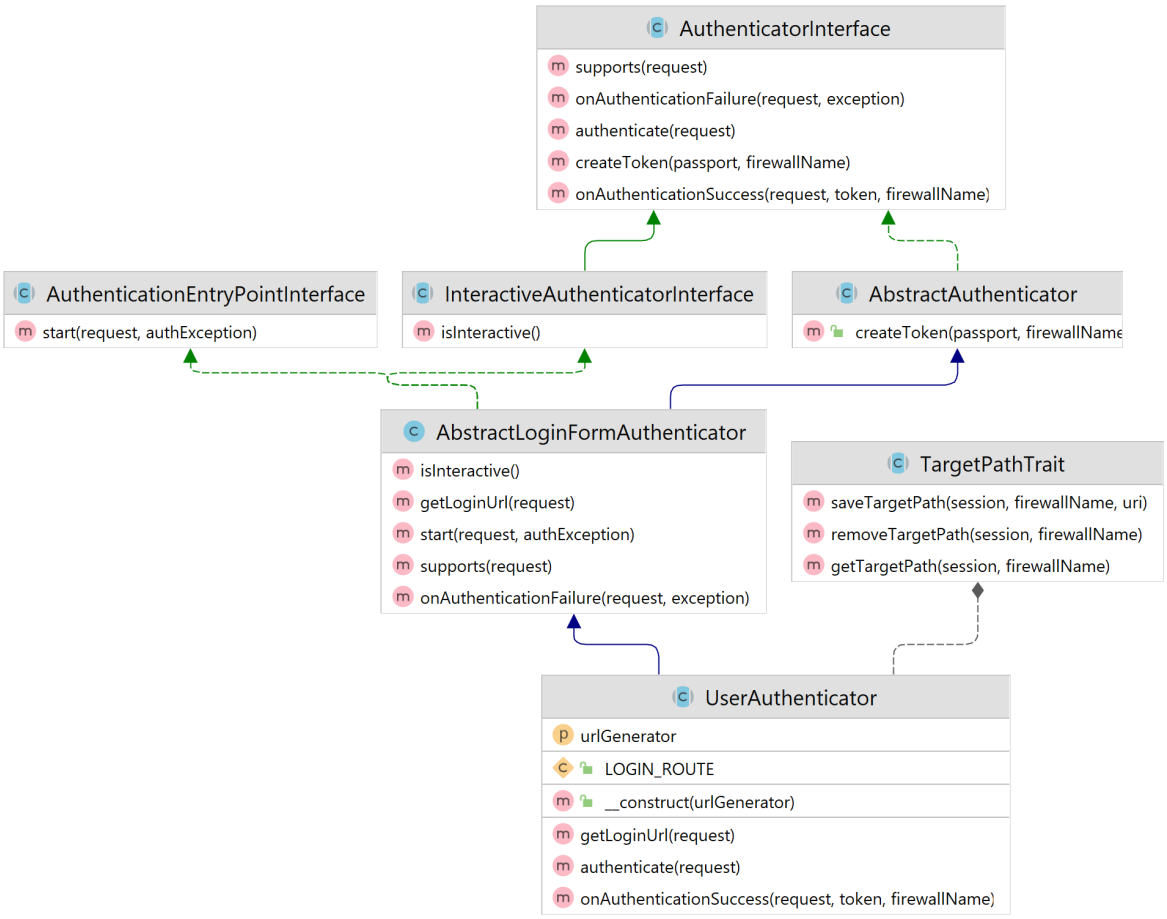
Figura 8 – Diagrama de classes de Repository



Fonte: Elaborado pelo autor.

- Security/: Gerencia as questões de autenticação e autorização, sua estrutura é exibida na Figura 9:

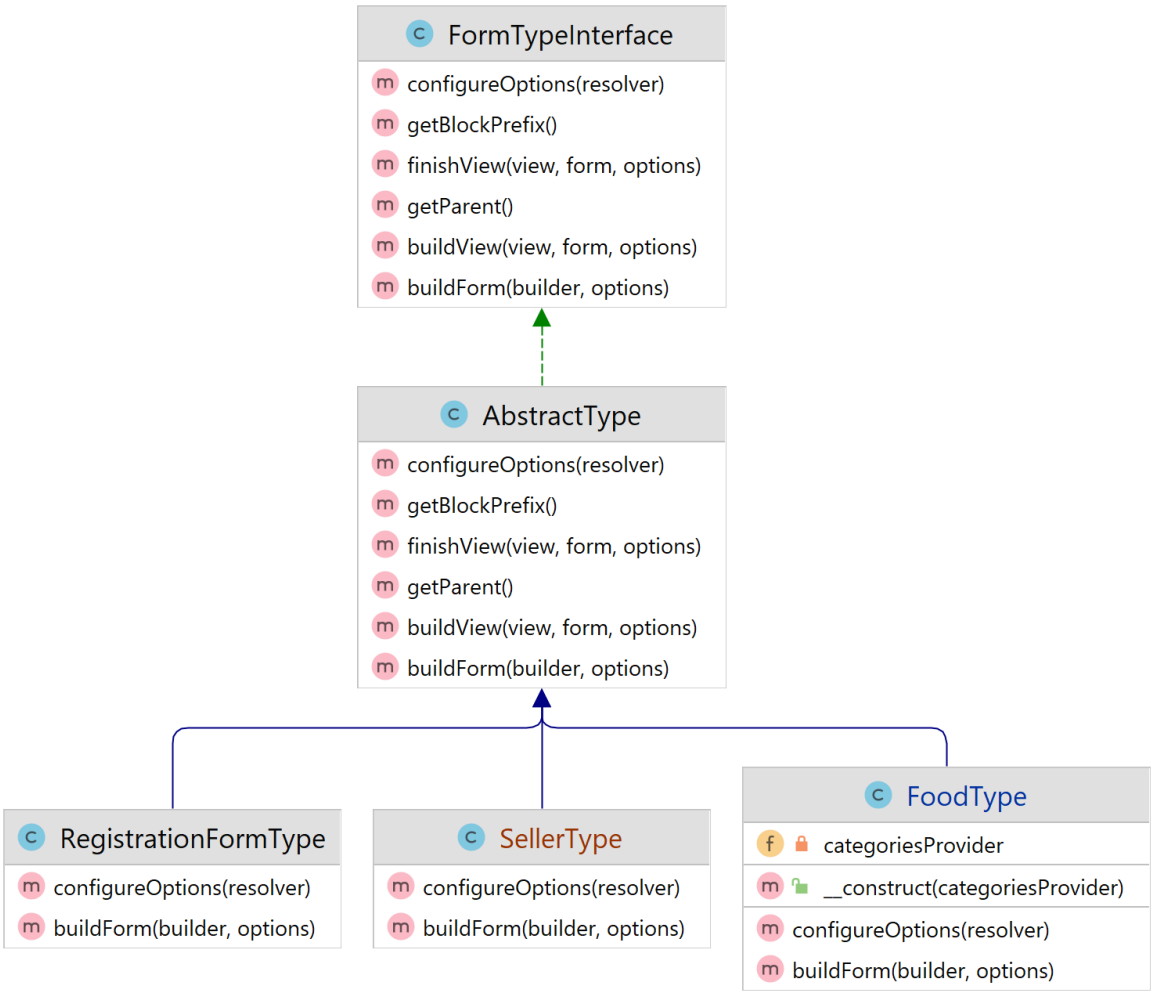
Figura 9 – Diagrama de classes de Security



Fonte: Elaborado pelo autor.

- Form/: Guarda as classes que constroem os formulários. Suas classes são representadas pelo diagrama da Figura 10:

Figura 10 – Diagrama de classes de Form



Fonte: Elaborado pelo autor.

- Service/: Onde estão os serviços, que são componentes reutilizáveis contendo a lógica de negócios.

Além do diretório src/, outros diretórios que possuem um papel importante são:

- public/: Serve como o diretório raiz do documento para o servidor web e abriga todos os arquivos acessíveis publicamente, como imagens, folhas de estilo e scripts JavaScript.
- config/: Contém os arquivos de configuração do projeto.
- templates/: Contém os arquivos de template Twig, que é a *view*, a camada de interação com o usuário.

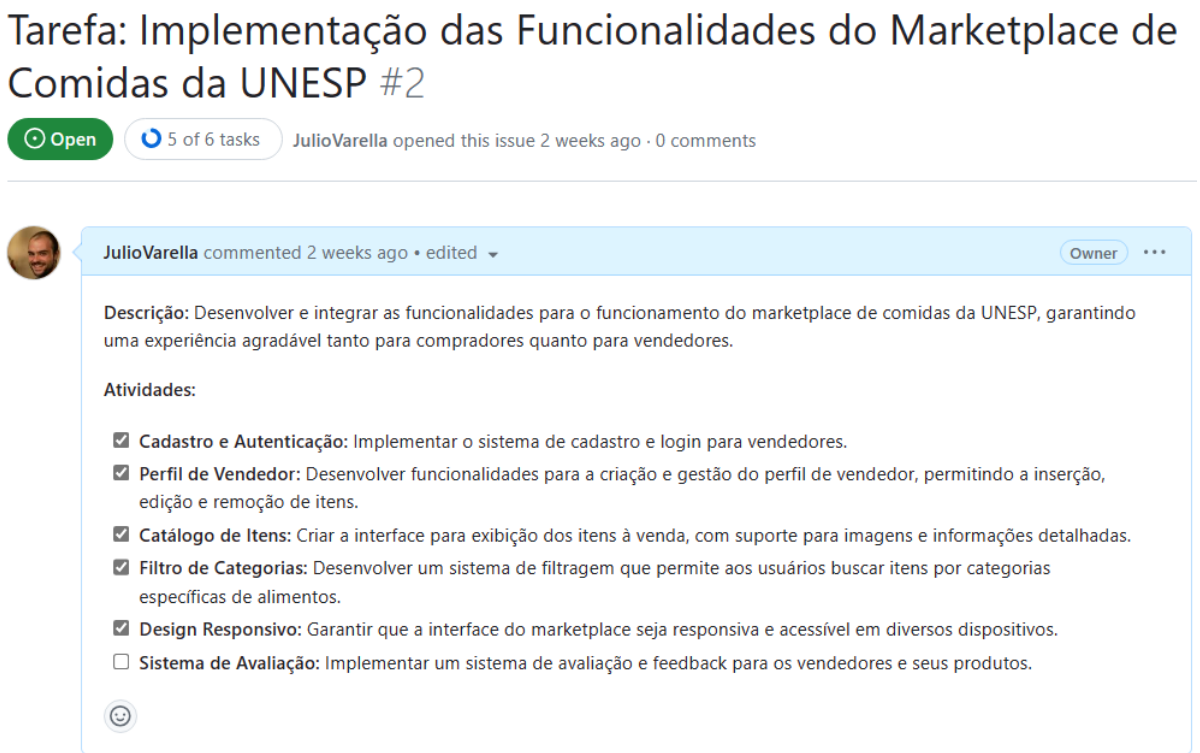
4.4.4 Controle de Versão e Contribuição com código-aberto

Para o controle de versão e da facilitação da colaboração, um repositório público foi criado no GitHub, permitindo a hospedagem do código-fonte do projeto. A aplicação de princípios SOLID e da arquitetura MVC, assim como o uso de boas práticas de desenvolvimento, são estratégias que melhoraram a modularidade e a extensibilidade do software. Esses princípios garantem a clareza no entendimento do código e simplificam a integração de novas funcionalidades, dando ao projeto uma base sólida para evolução contínua.

A criação de uma *issue* central no GitHub serve como um *roadmap* dinâmico, detalhado na Figura 11, que define o desenvolvimento atual e futuro. Esta *issue*, mantida e atualizada de forma contínua, é onde ocorre a adição e o acompanhamento de funcionalidades. Ela garante que o progresso do projeto seja transparente e aberto à comunidade, refletindo a adaptabilidade do sistema às demandas.

Contribuições são encorajadas através da branch dev, onde novas funcionalidade são propostas, testadas e revisadas. Através deste mecanismo de *pull requests*, a *branch main* é preservada, assegurando a estabilidade do código principal. A colaboração aberta e o uso de práticas de desenvolvimento avançadas solidificam o compromisso do projeto com a qualidade e a manutenção a longo prazo, impulsionadas pela comunidade de usuários e desenvolvedores.

Figura 11 – *Issue* do github para acompanhamento do *roadmap*



Fonte: GitHub, 2023.[1]

[1] https://github.com/JulioVarella/unesp-marketplace/issues/2

O acesso ao repositório é disponibilizado através do seguinte endereço: https://github.com/JulioVarella/unesp-marketplace

4.5 Implementação das funcionalidades

A primeira implementação foi das entidades, pois é a partir delas que os comandos do Doctrine são capazes de transformar o mapeamento do código em comandos SQL para a criação das tabelas do banco de dados.

As entidades "Food", "Seller" e "User" foram criadas para representar os vendedores e os anúncios, assim como para definir a conta de usuário necessária para criar anúncios.

O código 5 é uma migração, uma classe que contém os código SQL para criação das entidades definidas:

Código 5 – Migração das entidades Seller e Food

```
final class Version20231014141850 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to
            your needs
        $this->addSql('CREATE TABLE food (id INT AUTO_INCREMENT NOT
            NULL, name VARCHAR(255) NOT NULL, description LONGTEXT
            DEFAULT NULL, image_path VARCHAR(255) DEFAULT NULL,
            created_at DATETIME NOT NULL COMMENT \'(DC2Type:
            datetime_immutable)\', price DOUBLE PRECISION NOT NULL,
            category VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT
            CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE
            = InnoDB');
        $this->addSql('CREATE TABLE seller (id INT AUTO_INCREMENT NOT
            NULL, name VARCHAR(255) NOT NULL, email VARCHAR(255) NOT
            NULL, password VARCHAR(255) NOT NULL, location VARCHAR
            (255) NOT NULL, availability VARCHAR(255) NOT NULL,
            PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `
            utf8mb4_unicode_ci` ENGINE = InnoDB');
    }
```

```
    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it
           to your needs
        $this->addSql('DROP TABLE food');
        $this->addSql('DROP TABLE seller');
    }
}
```

Fonte: elaborado pelo autor

Para restringir o acesso a determinadas funcionalidades, foi criado um sistema de autenticação para que aqueles que desejam criar anúncios sejam registrados e tenham um perfil. Para implementar o sistema de autenticação, outro comando disponível no Symfony foi usado para gerar a página de cadastro e *login*, também do MakerBundle:

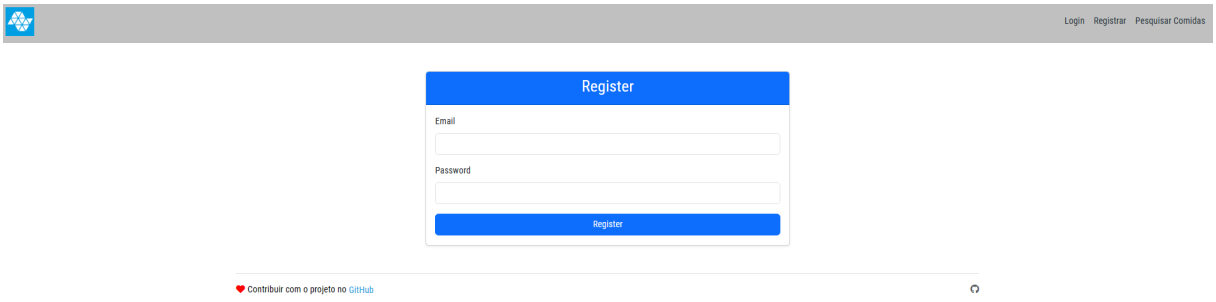
```
php bin/console make:auth
```

Com esse comando, o Symfony gera automaticamente:

- Uma classe de autenticador, que contém a lógica de como os usuários são autenticados no sistema.
- Um *controller* com a rota e a lógica para exibir e lidar com o formulário de *login*.
- Um *template* Twig para a página de *login*, que pode ser personalizado para se adequar ao visual desejado pelo projeto.

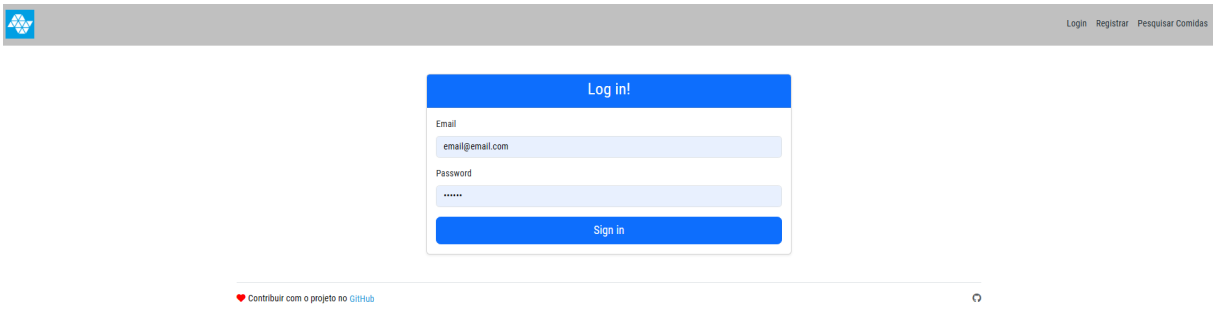
As figuras 12 e 13 mostram as telas criadas e devidamente ajustadas com componentes do Bootstrap:

Figura 12 – Tela de cadastro



Fonte: Elaborada pelo autor.

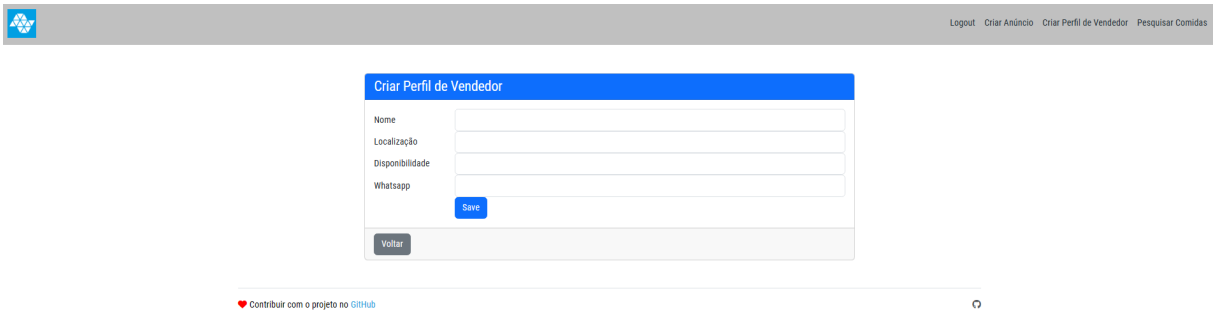
Figura 13 – Tela de *login*



Fonte: Elaborada pelo autor.

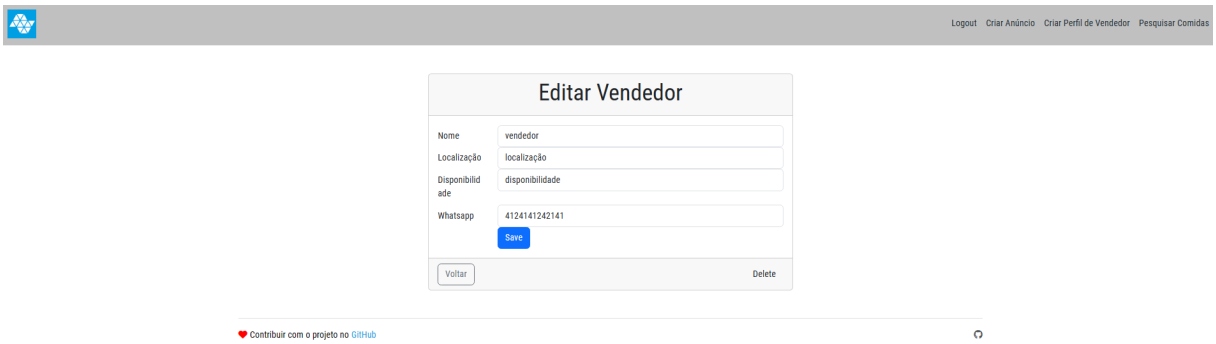
Ao se cadastrar e realizar o *login*, o vendedor precisa criar um perfil para que os compradores possam ver suas informações e contactá-lo ou saber onde procurá-lo pelo campus, uma vez que o sistema é apenas um meio de conectar as duas partes dessa relação e não processa nenhuma forma de pagamento. Nas figuras 14 e 15, temos a visualização das telas de criação e edição do perfil de vendedor:

Figura 14 – Tela de criação do perfil de vendedor



Fonte: Elaborada pelo autor.

Figura 15 – Tela de edição do perfil de vendedor



Fonte: Elaborada pelo autor.

A partir da criação das entidades, foi possível utilizar uma ferramenta do Symfony para acelerar o processo de criação das operações CRUD (*Create*, *Read*, *Update*, *Delete*)

com o comando "make:crud" do componente MakerBundle. Sendo assim, a partir da linha de comando, cada entidade teve seu CRUD gerado:

```
php bin/console make:crud Food
php bin/console make:crud Seller
```

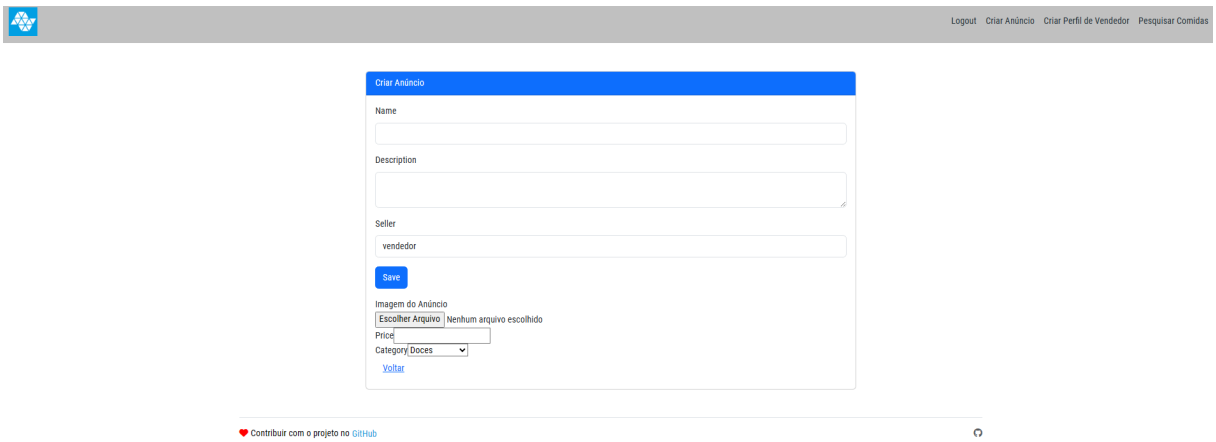
Dessa forma, os arquivos para manipular essas entidades foram gerados, sendo eles:

- Um controlador com métodos para listar, criar, editar e deletar os registros.
- Templates do Twig para as visualizações de cada operação CRUD.
- Um formulário PHP para cada entidade, utilizado para as operações de criação e edição.

Além de agilizar o desenvolvimento, os códigos gerados seguem boas práticas padronizadas pelo Symfony, incluindo aderência aos princípios SOLID e ao *design* MVC que serão mostrados posteriormente.

As próximas figuras ilustram as interfaces utilizadas para interação com os anúncios. A Figura 16 mostra a tela de criação de um anúncio, onde o vendedor irá preencher os dados do seu anúncio que será vinculado ao seu perfil. Até o momento, a imagem deve ser colocada diretamente no diretório /public e ser referenciada através do campo Image path. O *upload* de imagem é uma funcionalidade que deve ser implementada futuramente através da continuação do desenvolvimento pelo repositório aberto.

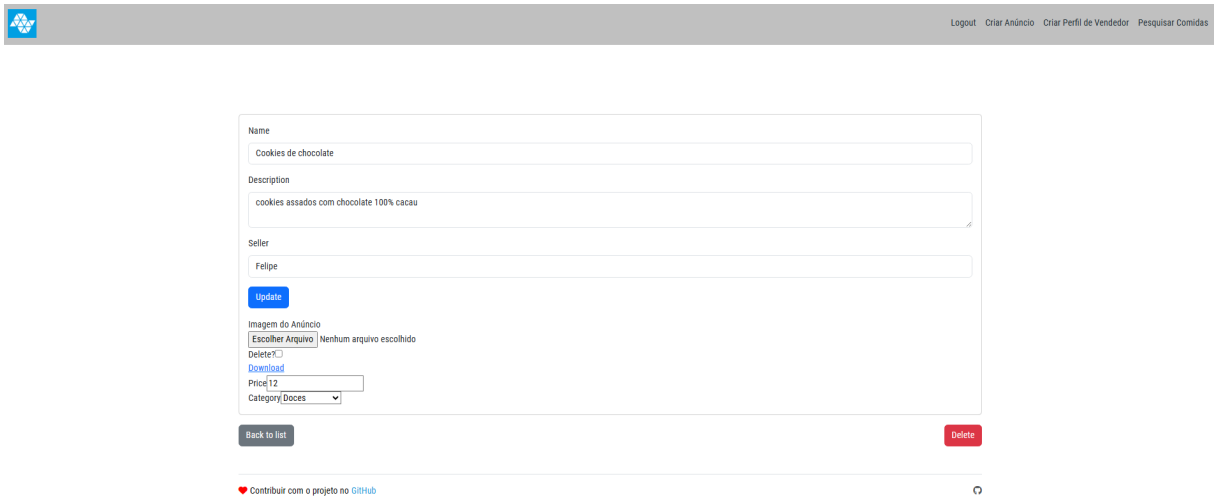
Figura 16 – Tela de criação de anúncio



Fonte: Elaborada pelo autor.

A Figura 17 mostra a tela de edição do anúncio, onde o vendedor também pode excluí-lo.

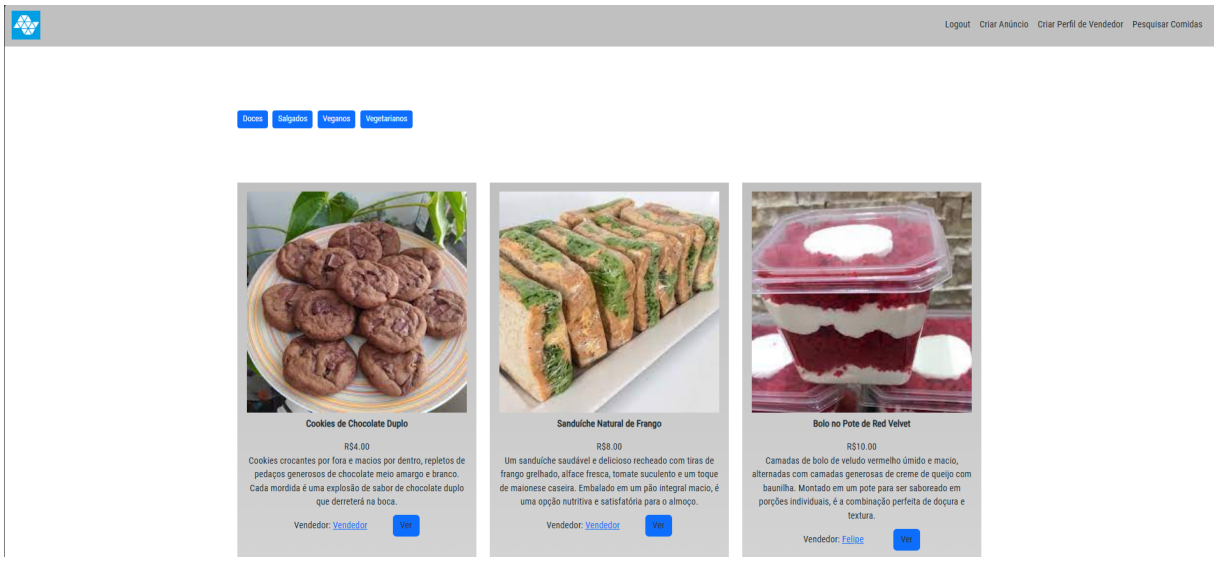
Figura 17 – Tela de edição de um anúncio



Fonte: Elaborada pelo autor.

Ao navegar pelo menu e clicar no *link* para "Pesquisar Comidas", o usuário é redirecionado para a principal tela do site, mostrada na Figura 18, onde os anúncios são organizados em uma lista com nome, foto, descrição, vendedor e um *link* para a página própria do anúncio. Além disso, também existe um filtro de anúncios por categoria.

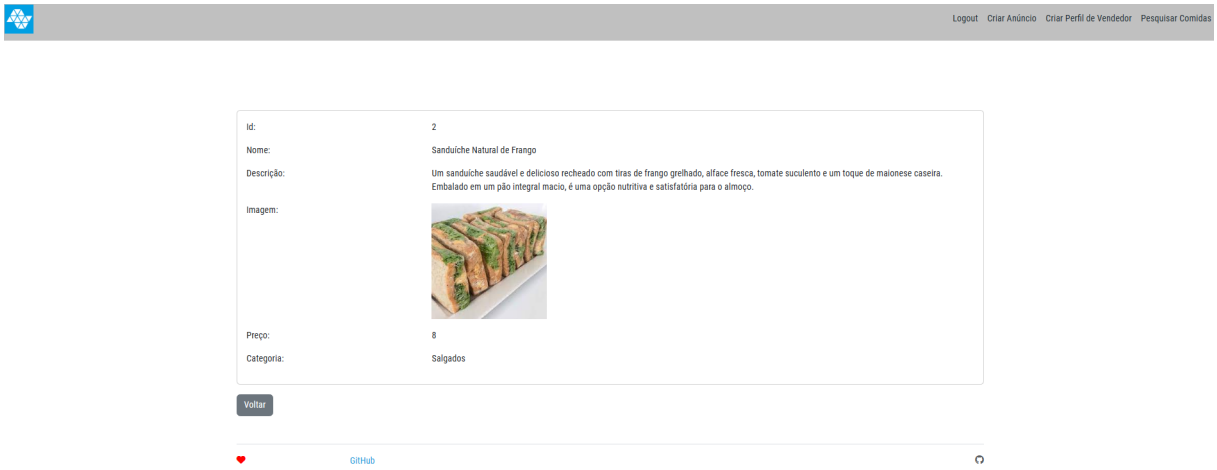
Figura 18 – Tela de busca de anúncios



Fonte: Elaborada pelo autor.

A partir da tela principal onde os anúncios são listados, o usuário pode abrir tanto a página de detalhes do item, mostrado na Figura 19, quanto para a página do vendedor (Figura 20), onde é detalhado seu perfil e também é possível contactá-lo diretamente pelo WhatsApp através de um botão de redirecionamento:

Figura 19 – Tela de detalhes do anúncio



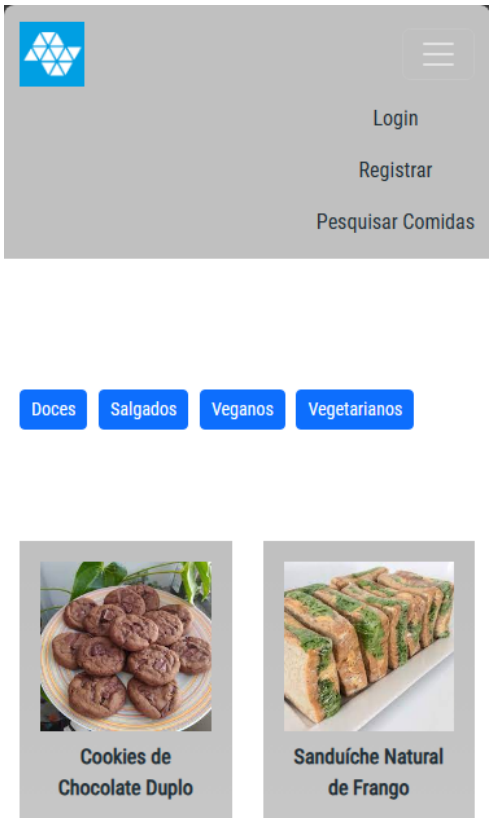
Fonte: Elaborada pelo autor.

Figura 20 – Tela de detalhes do vendedor

Fonte: Elaborada pelo autor.

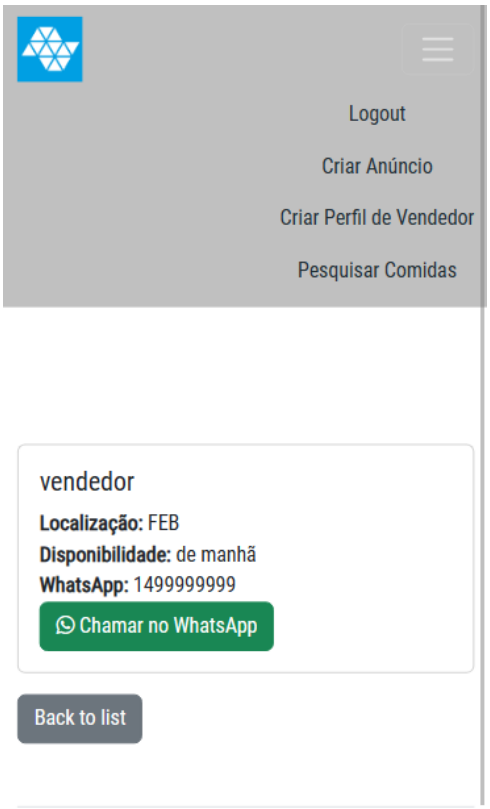
Levando em conta que a maioria dos usuários preferem acessar esse tipo sistema pelo celular, a responsividade era um requisito importante. Apesar de estar passível de melhorias, as figuras 21 e 22 mostram as principais telas (de navegação e detalhe do vendedor) com a visualização *mobile*.

Figura 21 – Mobile - tela de busca de anúncios



Fonte: elaborada pelo autor

Figura 22 – Mobile - tela de detalhe do vendedor



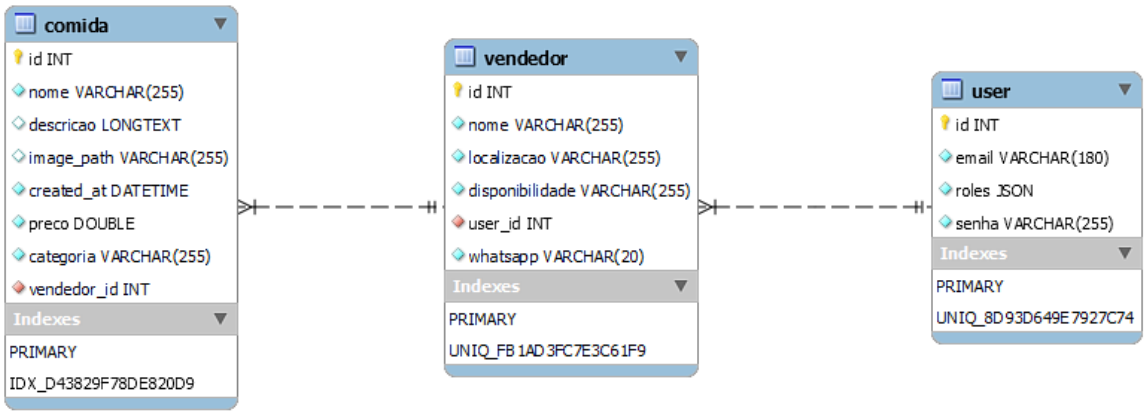
Fonte: elaborada pelo autor

Embora diversas funcionalidades planejadas para serem desenvolvidas nos requisitos iniciais tenham sido implementadas, o projeto passou por adaptações para que o sistema fosse finalizado e houvesse tempo de fazer uma análise do código de acordo com os princípios SOLID dentro do cronograma. Por isso, a implementação do sistema de avaliação dos anúncios não foi feita, apesar de ser um requisito que agregaria valor ao sistema, principalmente em relação à forma em que as pessoas anunciam alimentos na universidade atualmente.

Além disso, o projeto foi planejado para ser *open-source* desde o início. Isso significa que qualquer pessoa pode contribuir para o seu desenvolvimento, adicionando novas funcionalidades, como o sistema de avaliação, e melhorando as existentes. Isso está alinhado com a visão do projeto de evoluir com a ajuda da comunidade de desenvolvedores.

O Modelo Entidade-Relacionamento (MER) da Figura 23 mostra como o sistema ficou organizado. Ele ilustra as estruturas de dados e as relações que suportam as funcionalidades atuais, sem a diferenciação de usuário para vendedor e consumidor e também sem a existência da entidade Avaliação prevista inicialmente.

Figura 23 – MER do sistema



Fonte : Elaborado pelo autor.

Com as principais funções do sistema em funcionamento, o tempo restante foi dedicado à análise e refatoração do código seguindo os princípios SOLID.

4.6 Refatoração e análise do código conforme os princípios SOLID

A aplicação dos princípios SOLID é evidente em várias partes do sistema. A separação entre a lógica de negócios, lógica de apresentação e acesso a dados é uma prática recomendada que é naturalmente incentivada pelo uso do Framework Symfony. Alguns exemplos serão demonstrados a seguir.

No repositório FoodRepository, todas as operações de acesso a dados relacionadas a alimentos foram encapsuladas, garantindo que cada classe tenha uma única responsabilidade se adequando ao SRP. O Código 6 mostra como este princípio é aplicado:

Código 6 – Aplicação do SRP no Repository

```
// src/Repository/FoodRepository.php

class FoodRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Food::class);
    }

    public function findFoodByCategory(string $category): array
    {
        return $this->createQueryBuilder('f')
```

```
            ->andWhere('f.category = :category')
            ->setParameter('category', $category)
            ->orderBy('f.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult()
        ;
    }

    public function findAllFoods(): array
    {
        return $this->createQueryBuilder('f')
            ->orderBy('f.id', 'ASC')
            ->getQuery()
            ->getResult()
        ;
    }

    public function findByCategoryName($name): array
    {
        return $this->createQueryBuilder('f')
            ->join('f.category', 'c')
            ->where('c.name = :name')
            ->setParameter('name', $name)
            ->getQuery()
            ->getResult();
    }
}
```

Fonte: elaborado pelo autor

Na camada das interfaces, os templates Twig apresentam os dados que são passados do Controller, sem necessidade de realizar qualquer lógica de negócios ou acesso direto aos dados. Isso é consistente com a separação clara de responsabilidades e pode ser observado pela variável "food" que é enviada pelo Controller com os dados de todos os anúncios que devem ser exibidos:

Código 7 – Aplicação do SRP na View

```
{# templates/food_item/index.html.twig #}

{% extends 'base.html.twig' %}
```

```
{% block body %}
    <!-- Loop para exibir os itens de comida -->
    {% for foodItem in food %}
        <!-- Exibicao do item de comida -->
    {% else %}
        <p>No records found</p>
    {% endfor %}
{% endblock %}
```

Fonte: elaborado pelo autor

Dessa forma, o fluxo de dados entre os arquivos a partir de uma chamada da listagem de anúncios fica definido como demonstrado no fluxograma da Figura 24:

Figura 24 – Fluxo de dados

Fonte : Elaborado pelo autor.

Outro exemplo de aplicação do SRP aparece no Controller. Usando de exemplo FoodItemController, sua responsabilidade é gerenciar o ciclo de vida das entidades Food em termos de ações do usuário, como criar, editar e excluir. É possível observar como o Controller é simplificado, delegando a persistência dos dados para o EntityManager e o acesso aos dados para o FoodRepository.

Código 8 – Aplicação do SRP no Controller

```
// src/Controller/FoodItemController.php
#[Route('/food/item')]
class FoodItemController extends AbstractController
{
    #[Route('/category/{categoryName}', name: '
        app_food_item_by_category', methods: ['GET'])]
    public function indexByCategory(FoodRepository $foodRepository,
        CategoriesProvider $categoriesProvider, string $categoryName):
         Response
    {
        $categoryName = strtolower($categoryName);
        $foods = $foodRepository->findBy(['category' => $categoryName
            ]);
        $categories = $categoriesProvider->getCategories();
```

```
        return $this->render('food_item/index.html.twig', [
            'food' => $foods,
            'categories' => $categories,
        ]);
    }
}
```

Fonte: elaborado pelo autor

Embora o FoodItemController esteja seguindo o SRP, existe a possibilidade de refatorá-lo para que se adeque ao ISP. A injeção de dependências nos métodos da classe pode ser revisada para que a dependência do EntityManagerInterface seja estabelecida de forma mais alinhada ao ISP.

Código 9 – Refatoração do controller para se adequar ao ISP

```
// src/Controller/FoodItemController.php
#[Route('/food/item')]
class FoodItemController extends AbstractController
{
    private FoodRepository $foodRepository;
    private CategoriesProvider $categoriesProvider;
    private EntityManagerInterface $entityManager;

    public function __construct(
        FoodRepository $foodRepository,
        CategoriesProvider $categoriesProvider,
        EntityManagerInterface $entityManager
    ) {
        $this->foodRepository = $foodRepository;
        $this->categoriesProvider = $categoriesProvider;
        $this->entityManager = $entityManager;
    }

    #[Route('/category/{categoryName}', name: '
       app_food_item_by_category', methods: ['GET'])]
    public function indexByCategory(string $categoryName): Response
    {
        $categoryName = strtolower($categoryName);
        $foods = $this->foodRepository->findBy(['category' =>
           $categoryName]);
        $categories = $this->categoriesProvider->getCategories();
```

```
        return $this->render('food_item/index.html.twig', [
            'food' => $foods,
            'categories' => $categories,
        ]);
    }
}
```

Fonte: elaborado pelo autor

Nesta refatoração, todas as dependências são injetadas através do construtor, o que simplifica o método indexByCategory e melhora a coesão da classe. Além disso, isso facilita a manutenção e os testes, pois as dependências são centralizadas, tornando claro que FoodItemController depende de FoodRepository, CategoriesProvider, e EntityManagerInterface para suas operações. Essa mudança também facilita possíveis melhorias futuras, como a definição de interfaces mais específicas para as dependências, caso o FoodRepository ou o CategoriesProvider comecem a oferecer múltiplos métodos não utilizados pelo Controller, alinhando ainda mais com o ISP.

Agora levando em conta a relação de Food com as categorias. Essa entidade possui um campo category do tipo *string*. Isso implica que as categorias são gerenciadas diretamente pela entidade Food, o que vai contra o SRP, pois Food acaba tendo mais de uma razão para mudar (alterações em informações da comida ou nas categorias). Além disso, o ISP não é diretamente aplicável aqui, pois não estamos lidando com interfaces que possam estar sobrecarregadas por funcionalidades não utilizadas. O código 10 mostra o estado dessa relação antes da refatoração:

Código 10 – Categorias como atributo(string) de Food

```
#[ORM\Entity(repositoryClass: FoodRepository::class)]
class Food
{
    //..
    //outros atributos
    //..
    #[ORM\Column(length: 255)]
    private ?string $category = null;

        public function getCategory(): ?string
    {
        return $this->category;
    }

    public function setCategory(string $category): static
```

```
    {
        $this->category = $category;

        return $this;
    }
    //..
    //outros metodos
    //..
}
```

Fonte: elaborado pelo autor

Para melhorar a relação entre esses dados e alinhar-se com os princípios SOLID, pode-se refatorar essa estrutura, extraindo a responsabilidade de categorias para uma nova entidade chamada Category.

Código 11 – Refatoração de categorias em uma classe Category

```
/**
 * @ORM\Entity(repositoryClass=CategoryRepository::class)
 */
class Category {
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private string $name;

    // getters and setters ...
}

/**
 * @ORM\Entity(repositoryClass=FoodRepository::class)
 */
class Food {
    // ...

    #[ORM\ManyToOne(targetEntity=Category::class)]
    #[ORM\JoinColumn(nullable: false)]
    private ?Category $category = null;
```

```
    // O metodo setCategory agora aceita uma instencia de Category ao
        inves de uma string.
    public function setCategory(Category $category): self {
        $this->category = $category;
        return $this;
    }

    // O metodo getCategory agora retorna uma instancia de Category
       ao inves de uma string.
    public function getCategory(): ?Category {
        return $this->category;
    }

    // ...
}
```

Fonte: elaborado pelo autor

A separação da entidade Category segue o SRP, pois agora tem-se uma classe que é responsável exclusivamente pelas informações de categoria. A entidade Food se torna mais simples e com uma única responsabilidade: gerenciar informações da comida.

Quanto ao ISP, apesar de ainda não existir uma aplicação direta (pois não há uso de interfaces neste contexto específico), a separação em entidades distintas segue o espírito do ISP ao não forçar a entidade Food a depender de métodos que não usa diretamente. Se futuramente surgirem interfaces para interagir com Food ou Category, cada uma poderia implementar apenas os métodos que são relevantes para suas funcionalidades, respeitando o ISP.

A refatoração sugerida simplifica a manutenção da aplicação e facilita expansões futuras, tais como a adição de categorias com atributos e comportamentos específicos, sem afetar a lógica inerente à entidade Food. Essa abordagem promove uma aderência mais rigorosa aos princípios SOLID, contribuindo para a robustez e escalabilidade do *marketplace*.

5 Conclusão

No âmbito da permanência estudantil, a comercialização de alimentos emerge como uma solução viável para os estudantes que enfrentam desafios para se manter na universidade, especialmente na ausência de auxílios consistentes. O sistema desenvolvido neste trabalho apresenta-se como um meio de centralizar e organizar anúncios de vendas de comida, em alternativa aos métodos descentralizados e menos eficientes, como os grupos de WhatsApp. Esta centralização visa facilitar a inclusão e a gestão eficaz dos anúncios, respondendo a uma necessidade da comunidade estudantil.

Reconhecendo a natureza dinâmica do software, que demanda adaptações constantes e a inclusão de novas funcionalidades, o sistema foi desenvolvido com um foco na clareza e na possibilidade de expansões futuras. A base teórica solidificada permitiu uma compreensão dos princípios que orientam o desenvolvimento de software eficiente e sustentável.

As funcionalidades do sistema foram implementadas com o suporte das ferramentas de geração de código providas pelo Symfony e pelo Bootstrap, visando simplicidade, responsividade e uma organização que respeita os princípios SOLID e o padrão MVC. As refatorações demonstradas ao longo do trabalho evidenciaram a facilidade com que o sistema pode ser mantido e expandido, refletindo um *design* de software que preza pela adaptabilidade e extensibilidade.

Como uma contribuição para a comunidade da UNESP, o sistema foi disponibilizado sob uma licença de código aberto no GitHub, incentivando a continuidade e o desenvolvimento colaborativo do projeto. Essa iniciativa destaca a importância da colaboração, visando caracterizar o sistema como uma criação da comunidade e para a comunidade, esperando impactar positivamente na experiência estudantil e na gestão de atividades comerciais dentro do contexto universitário.

6 Trabalhos futuros

Para futuros avanços no projeto, baseando-se nas opiniões colhidas durante a etapa de levantamento de requisitos e na flexibilidade do sistema para aceitar novas extensões, identificam-se diversas funcionalidades que poderiam agregar valor ao sistema. A implementação de um sistema de avaliações permitiria aos usuários expressar a satisfação com os produtos ofertados, criando um atrativo que não existe no principal meio de compra e venda de comida utilizado atualmente. Otimizações no *layout* visando uma experiência de usuário aprimorada em dispositivos móveis ampliariam o acesso e a usabilidade do sistema. Além disso, a introdução de um sistema de autenticação restrito aos membros da UNESP fortaleceria a segurança e a integridade da plataforma.

A adoção destas funcionalidades poderia ser facilitada pela natureza de código aberto do sistema, permitindo que outros membros da comunidade acadêmica pudessem contribuir com melhorias e novas características. É fundamental que tais implementações continuem a seguir os padrões estabelecidos no desenvolvimento original do sistema, mantendo a separação de responsabilidades e aderindo aos princípios SOLID para garantir que novas funcionalidades sejam incorporadas sem afetar as operações já existentes.

Por fim, a implementação de uma infraestrutura de hospedagem adequada possibilitaria a entrada do sistema em operação, concretizando o seu potencial de impacto e contribuição para a permanência estudantil na UNESP.

Referências

ARAÚJO, M. A. M. de; ALMEIDA, L. L. de; LOURO, D. W.; DEL-MASSO, M. C. S. O impacto da política de permanência estudantil na unesp – a percepção do aluno bolsista. *Revista Ciência em Extensão*, v. 7, n. 2, p. 17, 2011. ISSN 1679-4605.

BAURU, D. F. *Fwd: Esclarecimento - Restaurante Universitário*. 2022. Mensagem recebida por: <julio.varella@unesp.br> em 25 abr. 2022.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. [S.l.]: Addison-Wesley Professional, 1994. ISBN 0201633612.

GIT. *About*. 2023. Disponível em: https://www.git-scm.com/about. Acesso em 01 de novembro de 2023.

JOSHI, B. *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*. 1st. ed. USA: Apress, 2016. ISBN 1484218477.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003. ISBN 0135974445.

MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. ed. USA: Prentice Hall PTR, 2008. ISBN 0132350882.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.: s.n.], 2017. ISSN 13563890.

MARTIN, R. C. *Solid Relevance*. 2020. Disponível em: https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html. Acceso em 24 de outubro de 2023.

MORAES, F. *COM CORTE NOS AUXÍLIOS, UNIVERSIDADE VAI SENDO SILENCIOSAMENTE DEVOLVIDA ÀS ELITES*. 2022. Disponível em: https://www.intercept.com.br/2022/03/22/universidade-corte-auxilio-educacao-baixa-renda/. Acesso em: 12 abril 2023.

MYSQL. *What is MySQL?* 2023. Disponível em: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. Acesso em 01 de novembro de 2023.

PHP. *O que é o PHP?* 2023. Disponível em: https://www.php.net/manual/pt_BR/intro-whatis.php. Acesso em 01 de novembro de 2023.

POP, D.-P.; ALTAR, A. Designing an mvc model for rapid web application development. *Procedia Engineering*, v. 69, p. 1172–1179, 2014. ISSN 1877-7058. 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013. Disponível em: https://www.sciencedirect.com/science/article/pii/S187770581400352X. Acesso em 23 de Outubro de 2023.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software - 8ª Edição*. [S.l.: s.n.], 2016. ISBN 9788580555349. Disponível em: https://books.google.com.br/books?id=wexzCwAAQBAJ. Acesso em 28 de Outubro de 2023.

SMITH, D. d. S.; REIS, A. A. dos. Percepções analíticas sobre o "marketplace": uma revisão bibliográfica. *International Journal of Scientific Management and Tourism*, v. 9, n. 4, p. 2040–2060, 2023. Disponível em: https://doi.org/10.55905/ijsmtv9n4-008. Acesso em 01 de Novembro de 2023.

SYMFONY. *Six good reasons to use Symfony*. 2023. Disponível em: https://symfony.com/six-good-reasons. Acesso em 01 de novembro de 2023.

SYMFONY. *Why Use a Framework?* 2023. Disponível em: https://symfony.com/why-use-a-framework. Acesso em 31 de outubro de 2023.

TWIG. *What makes Twig better than PHP as a template engine?* 2023. Disponível em: https://twig.symfony.com/. Acesso em 01 de novembro de 2023.