Padrões de Projeto no Desenvolvimento de Solução de Software *Full Stack*: Um Relato de Experiência

Reinaldo Wendt[1], **Eduardo Tiadoro**[1], **Miguel Muniz**[1], **Maicon Bernardino**[1]

[1]Laboratory of Empirical Studies in Software Engineering (LESSE)
Universidade Federal do Pampa (UNIPAMPA) – Alegrete – RS – Brasil

{reinaldowendt,eduardotiadoro,miguelmuniz}.aluno@unipampa.edu.br,
bernardino@acm.org

Abstract. *This article presents an experience report on the use of design patterns in the development of a full stack application aimed at the agribusiness sector. The solution, conceived as a digital platform for renting agricultural machinery, was built using a client-server architecture with Express.js, SvelteKit, and Firebase. Patterns such as Repository, State, Builder, Command, and Observer were applied to ensure modularity, architectural clarity, and easier maintenance. The analysis highlighted benefits in code organization, reuse, and readability, but also revealed challenges in adapting patterns to specific contexts. The study provides practical insights into the effectiveness of these patterns in distributed solutions.*

Resumo. *Este artigo apresenta um relato de experiência sobre o uso de padrões de projeto no desenvolvimento de uma aplicação full stack voltada ao setor do agronegócio. A solução, concebida como uma plataforma digital para aluguel de máquinas agrícolas, foi construída a partir de uma arquitetura cliente-servidor, utilizando Express.js, SvelteKit e Firebase. Foram aplicados padrões como Repository, State, Builder, Command e Observer, visando modularidade, clareza arquitetural e manutenção facilitada. A análise evidenciou benefícios em organização, reutilização e legibilidade do código, mas também desafios na adaptação de padrões a contextos específicos. O estudo contribui com reflexões práticas sobre a efetividade desses padrões em soluções distribuídas.*

1. Introdução

A adoção de padrões de projeto (*design patterns*) tem se consolidado como uma prática fundamental no desenvolvimento de software, oferecendo soluções reutilizáveis para problemas recorrentes de estrutura e comportamento em sistemas computacionais [Gamma et al. 1994]. Esses padrões não apenas promovem uma melhor organização interna do código, mas também facilitam a comunicação entre desenvolvedores ao estabelecer uma linguagem comum para o design de software. Quando aplicados em conjunto com uma arquitetura bem definida, eles contribuem significativamente para a modularidade, escalabilidade e manutenção dos sistemas.

No contexto atual do desenvolvimento de sistemas, aplicações *full stack* têm ganhado relevância por integrarem tanto a camada de apresentação quanto a de persistência e lógica de negócios em uma solução única e coesa. Nesse cenário, a utilização de padrões de projeto torna-se especialmente pertinente, uma vez que a complexidade inerente à

integração entre cliente e servidor demanda mecanismos que favoreçam a reutilização, a clareza arquitetural e a redução de acoplamentos indevidos.

Assim, o objetivo principal deste artigo é relatar a experiência de desenvolvimento de uma aplicação com arquitetura cliente-servidor, destacando o papel dos padrões de projeto no processo de construção e manutenção do sistema. Ao final, espera-se oferecer uma reflexão crítica sobre os benefícios, limitações e aprendizados advindos da aplicação desses padrões em um contexto prático, contribuindo para o avanço do conhecimento sobre sua efetividade em arquiteturas distribuídas.

Dessa forma, este artigo está organizado como segue. Na Seção 2 apresenta-se a fundamentação teórica, discutindo conceitos e classificações dos padrões de projeto. A Seção 3 revisa iniciativas semelhantes encontradas na literatura. Em seguida, a Seção 4 descreve os procedimentos adotados para o desenvolvimento da solução. A caracterização da aplicação proposta é detalhada na Seção 5, enquanto a Seção 6 disserta a discussão e análise sobre os padrões adotados no projeto. Por fim, a Seção 7 apresenta as considerações finais, ressaltando os principais resultados e perspectivas futuras.

2. Padrões de Projeto

Os padrões de projeto são soluções reutilizáveis para problemas recorrentes que surgem durante o desenvolvimento de software [Shalloway and Trott 2004]. Eles não são algoritmos prontos, mas sim descrições ou modelos que orientam como estruturar e organizar o código para resolver um determinado problema de forma eficiente. O conceito foi amplamente difundido na área de engenharia de software a partir da obra de [Gamma et al. 1994], que define como "descrições de soluções recorrentes para problemas que ocorrem com frequência no design de software orientado a objetos".

Entre os principais benefícios do uso de padrões de projeto, destacam-se a melhoria da comunicação entre equipes, a facilidade de manutenção e evolução do código e o aumento da reutilização de soluções testadas [Martin 2008]. Além disso, eles contribuem para reduzir a complexidade de sistemas ao fornecer abstrações de alto nível que favorecem a legibilidade e a escalabilidade das aplicações [Ali and Elish 2013]. Assim, o emprego de padrões assegura um desenvolvimento mais robusto e manutenível, consolidando princípios essenciais de qualidade de software.

2.1. Classificação dos Padrões

A literatura organiza os padrões de projeto em três categorias principais: criacionais, estruturais e comportamentais. Essa classificação tem como objetivo agrupar soluções conforme o tipo de problema que resolvem. Seguindo a definição de [Gamma et al. 1994]:

(i) Padrões Criacionais: São empregados para encapsular a lógica de criação de objetos, evitando o acoplamento direto entre classes e promovendo maior flexibilidade. Em sua obra, Gamma *et al.* apresentam 5 padrões criacionais, sendo eles: **(1)** *Abstract Factory*, **(2)** *Builder*, **(3)** *Factory*, **(4)** *Prototype*, **(5)** *Singleton*. Eles são especialmente úteis em cenários onde o processo de criação envolve múltiplas etapas ou quando diferentes representações de um mesmo objeto precisam coexistir. Dessa forma, esses padrões tornam o código mais adaptável a mudanças futuras e reduzem dependências rígidas.

(ii) Padrões Estruturais: Tange a forma como os objetos são organizados e combinados para formar estruturas de software. O objetivo é facilitar a modularidade, rea-

proveitamento de código e a separação clara de responsabilidades. São ao total 7, sendo eles: **(1)** *Adapter*, **(2)** *Bridge*, **(3)** *Composite*, **(4)** *Decorator*, **(5)** *Facade*, **(6)** *Flyweight*, **(7)** *Proxy*. Esses padrões são essenciais para criar arquiteturas mais reutilizáveis e expansíveis, além de facilitarem a integração entre componentes heterogêneos no sistema.

(iii) Padrões Comportamentais: Tratam da comunicação e coordenação entre objetos. Esses padrões buscam tornar os sistemas mais dinâmicos ao separar responsabilidades. Ao todo são 11, sendo eles: **(1)** *Chain of Responsibility*, **(2)** *Command*, **(3)** *Interpreter*, **(4)** *Iterator*, **(5)** *Mediator*, **(6)** *Memento*, **(7)** *Observer*, **(8)** *State*, **(9)** *Strategy*, **(10)** *Template Method*, **(11)** *Visitor*. Ao adotar esses padrões, os desenvolvedores conseguem projetar sistemas mais claros e adaptáveis a novas regras de negócio.

3. Trabalhos Relacionados

Esta seção investiga estudos e iniciativas semelhantes já existentes na literatura. A análise dessas obras tem como objetivo identificar abordagens e oportunidades de contribuição, funcionando como um referencial para a proposta apresentada neste trabalho.

Em Do Amaral Santos *et al.* (2016), os autores investigaram o conhecimento, os incentivos e as dificuldades de desenvolvedores de software sobre a utilização de padrões de projeto na indústria. O objetivo do trabalho foi identificar os fatores que encorajam ou desencorajam o uso de padrões, bem como a influência de aspectos organizacionais nesse processo. A metodologia consistiu em uma pesquisa quantitativa, por meio da aplicação de um questionário a 34 desenvolvedores de software atuantes no mercado. Os resultados apontaram que a cultura da empresa e seus processos de desenvolvimento são os principais influenciadores na adoção de padrões. Os maiores fatores que dificultam seu uso, segundo os participantes, são os prazos curtos dos projetos e a falta de conhecimento sobre os próprios padrões. O estudo concluiu que, apesar de os desenvolvedores reconhecerem claramente os benefícios dos padrões, a sua aplicação prática é frequentemente impedida por barreiras organizacionais e pressão por entregas rápidas.

Em Manik (2019), o autor avaliou o impacto do uso de padrões de projeto no desenvolvimento de um wrapper para uma API RESTful. O objetivo do estudo foi analisar as mudanças nas métricas de software ao aplicar os padrões *Builder*, *Observer* e *Factory*. A metodologia consistiu em gerar uma versão do código-fonte diretamente do modelo e, em seguida, uma segunda versão após a aplicação manual dos padrões. O autor usou análise estática para medir métricas de coesão, complexidade, acoplamento, herança e tamanho em ambas as versões. Os resultados mostraram que a implementação dos padrões de projeto aumentou todas as métricas analisadas. Entretanto, não se pode afirmar se isso é positivo ou negativo, pois tais aumentos podem representar um *trade-off* para melhorar outros atributos de qualidade de software a longo prazo, como flexibilidade, manutenibilidade e reusabilidade, que não foram medidos no estudo.

O trabalho de Tran *et al.* (2021) visou formalizar soluções para boas e más práticas comuns de APIs REST no formato de padrões de projeto e antipadrões. O estudo usou como metodologia o *Design Science Research*, e iniciou com uma revisão da literatura que identificou 19 práticas, que foram divididas em 8 "técnicas" (solucionáveis via arquitetura) e 11 "não técnicas". Para as práticas técnicas, os autores propuseram soluções adaptando padrões de projeto (como *Factory*, *Visitor* e *Proxy*) e fornecendo implementações de exemplo em Java Spring e ASP.NET Core. A validação dessas soluções foi realizada

por meio de uma pesquisa e entrevistas com 55 desenvolvedores profissionais. Os resultados mostraram alta aceitação para a maioria das soluções. O estudo concluiu que as soluções propostas são relevantes e aplicáveis em ambientes de produção, oferecendo um catálogo que pode servir como base para futuras implementações e pesquisas.

4. Metodologia

Nesta seção, será apresentado o arcabouço metodológico dos procedimentos adotado para o desenvolvimento e avaliação dos artefatos produzidos durante o trabalho.

4.1. *Design Science Research*

A metodologia adotada neste trabalho é a *Design Science Research* (DSR), uma abordagem para a condução de pesquisas que visam à construção e avaliação de artefatos voltados à solução de problemas em contextos reais. Segundo [Hevner et al. 2004], a DSR busca equilibrar rigor científico e relevância prática, permitindo que o desenvolvimento de soluções seja fundamentado tanto em bases teóricas quanto nas necessidades do ambiente em que serão aplicadas. Esse arcabouço se mostra adequado para investigações em Engenharia de Software, uma vez que favorece a criação de artefatos que não apenas solucionam problemas, mas também contribuem para o avanço do conhecimento científico.

A DSR estrutura o processo de pesquisa por meio da interação de três ciclos: ciclo da relevância, responsável por assegurar a conexão entre o problema investigado e o contexto prático; ciclo do rigor, que integra conhecimentos da literatura e fundamentos teóricos; e ciclo de design, que contempla a construção, avaliação e refinamento do artefato desenvolvido. Essa organização sistemática garante que os artefatos resultantes sejam simultaneamente úteis e cientificamente fundamentados [Horita et al. 2018].

4.2. Aplicação da *Design Science Research* no Estudo

A aplicação da metodologia neste estudo ocorreu de forma iterativa, abrangendo desde o projeto até a consolidação da solução. Essa abordagem favoreceu a evolução contínua, resultando em um artefato mais robusto e alinhado às necessidades do domínio.

Ciclo de Relevância: Este estudo foi guiado pelas demandas do agronegócio, que motivaram a criação de uma plataforma digital voltada inicialmente à divulgação de aluguéis de maquinários agrícolas. A partir da análise dessas necessidades, apoiada em uma revisão do estado da prática em aplicações semelhantes [Wendt et al. 2025], propôs-se o desenvolvimento de uma solução para tal. Esse alinhamento entre problema e contexto prático assegurou que o artefato desenvolvido tivesse utilidade direta.

Ciclo de Rigor: Este estudo foi sustentado por uma base teórica consolidada em literatura, garantindo que o desenvolvimento da solução não se limitasse a atender a uma demanda prática imediata, mas também estivesse fundamentado em princípios sólidos de Engenharia de Software. Para isso, foram considerados referenciais sobre o uso de padrões de projeto como instrumentos para melhorar a manutenibilidade, extensibilidade e qualidade de sistemas. Essa fundamentação proporcionou critérios objetivos para a seleção dos padrões empregados evitando escolhas *ad hoc*.

Ciclo de *Design*: Este ciclo será explorado de forma detalhada na Seção 5, com ênfase especial na Seção 5.3, em que são apresentados os padrões de projeto efetivamente

implementados na solução proposta. Nessa etapa são descritas as decisões de design adotadas, sua justificativa e a forma como cada padrão contribuiu para a construção do artefato, alinhando teoria e prática no desenvolvimento da aplicação.

5. Caracterização da Solução

A presente Seção descreve a solução desenvolvida no âmbito deste estudo. O objetivo é apresentar de como a aplicação foi concebida e construída, evidenciando as decisões arquiteturais e tecnológicas que nortearam o processo de desenvolvimento.

5.1. Contexto e Motivação

O artefato desenvolvido consiste em uma plataforma digital voltada à divulgação de serviços de aluguel de maquinários agrícolas. A escolha desse domínio justifica-se pela relevância do setor do agronegócio no cenário econômico brasileiro e pela crescente demanda por soluções tecnológicas que promovam maior eficiência e acessibilidade. As Figuras 1 e 2 representam, respectivamente, um exemplo da tela de listagem de anúncios e outro da tela de negociação do sistema.

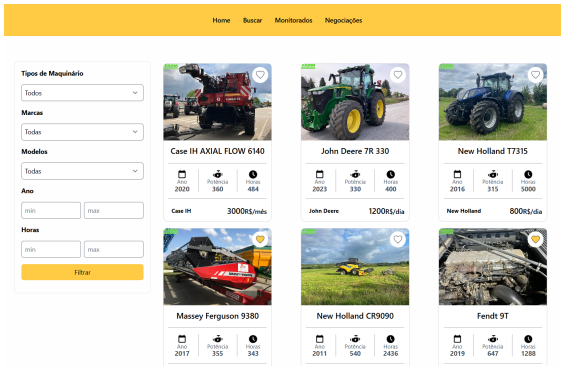

Figura 1. Tela de Anúncios

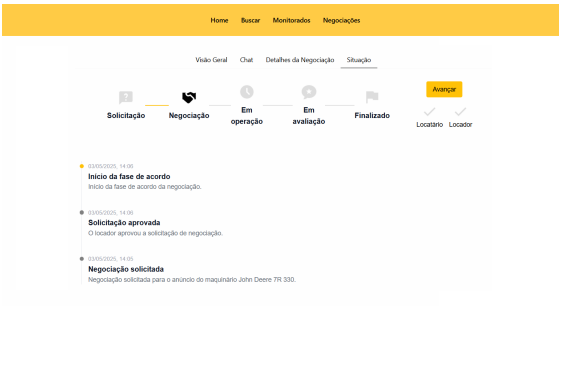

Figura 2. Tela de Negociação

A plataforma busca aproximar fornecedores de maquinário de potenciais clientes, oferecendo um ambiente centralizado para negociação de equipamentos, contribuindo assim para a modernização de práticas de economia compartilhada no âmbito agrícola.

5.2. Visão Geral da Plataforma Desenvolvida

A plataforma desenvolvida adota uma arquitetura de software baseada no modelo cliente-servidor, estruturada como uma aplicação *web*. O servidor foi implementado como uma *API REST*, construído em `Express.js` com a linguagem `JavaScript`, responsável pelo processamento das requisições, pela aplicação das regras de negócio e pela integração com a camada de persistência de dados. O armazenamento das informações é realizado no *Firebase Firestore Database*, um banco de dados NoSQL orientado a documentos, que oferece escalabilidade e integração com aplicações baseadas em nuvem.

O cliente, por sua vez, corresponde à interface web acessada pelos usuários, desenvolvida em `Svelte` com o `SvelteKit` como *framework*. Complementado pelas bibliotecas `SkeletonUI` e `Tailwind CSS` para construção de interfaces responsivas. Essa combinação de tecnologias no *front-end* possibilitou a criação de uma experiência de usuário fluida, alinhada a boas práticas de design e usabilidade.

A Figura 3 apresenta a visão arquitetural do software a partir de uma representação baseada no *C4 model*, evidenciando os principais componentes da solução e suas interações [Brown 2021]. Essa visão fornece uma abstração de alto nível que facilita o entendimento da plataforma, destacando a divisão entre as camadas cliente e servidor e a forma como elas se conectam à base de dados.

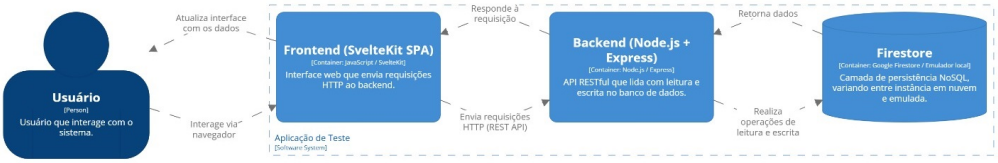

Figura 3. Arquitetura do Software

5.3. Implementação de Padrões de Projeto

Esta seção apresenta os principais padrões de projeto aplicados, detalhando seu papel e a forma como contribuíram para a organização e evolução do sistema.

5.3.1. Repository

O padrão *Repository* foi utilizado para organizar a camada de persistência de dados, promovendo a separação entre a lógica de negócio e o acesso ao banco de dados, além de centralizar a manipulação das entidades do sistema. Foi implementada uma classe genérica `Repository`, responsável por prover métodos básicos de manipulação de dados (como `create`, `read`, `update`, `delete`). A partir dela, foram derivadas classes específicas que reutilizam os métodos principais e, quando necessário, definem métodos auxiliares próprios para atender particularidades de cada entidade. Essa abordagem contribuiu para reduzir redundâncias e facilitar futuras alterações na camada de persistência. A Figura 4 apresenta um diagrama de classes simplificado da implementação do padrão no sistema.

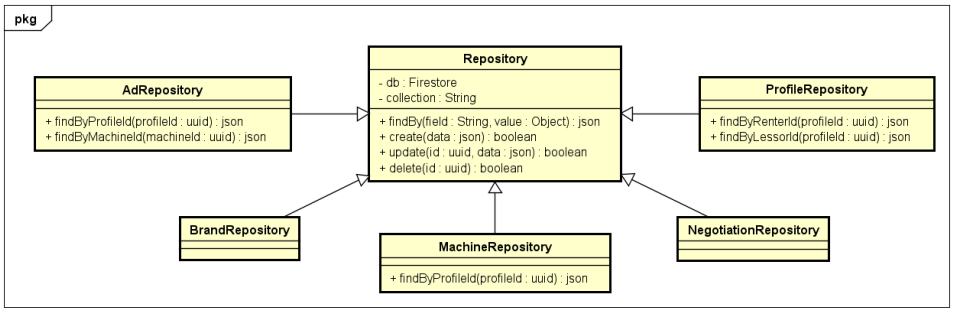

Figura 4. Diagrama de Classes da Implementação do Padrão `Repository`

5.3.2. State

O padrão *State* foi aplicado para modelar os diferentes estados do ciclo de vida de uma negociação. A negociação pode evoluir por diferentes etapas: `Solicitada` → `Negociação` → `Operação` → `Avaliação` → `Finalizada`.

Cada estado foi representado por uma classe distinta, implementando uma interface comum denominada `State`. Essa abordagem reduziu a complexidade do código ao evitar condicionais extensos para tratar mudanças de estado, além de tornar o fluxo da negociação mais legível e manutenível. A Figura 5 apresenta um diagrama de classes simplificado da implementação do padrão *State* no sistema.

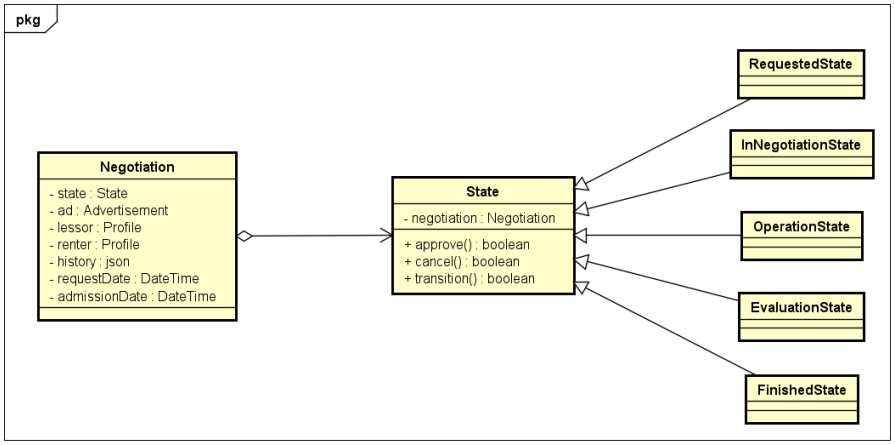

Figura 5. Diagrama de Classes da Implementação do Padrão `State`

5.3.3. Builder

O padrão *Builder* foi empregado para facilitar a criação de objetos complexos que exigem o preenchimento de múltiplos atributos, como anúncios, maquinários e negociações. Sua utilização evitou construtores excessivamente longos e proporcionou maior legibilidade na instanciação desses objetos, além de tornar o processo de criação mais flexível, reutilizável e menos propenso a erros em cenários de evolução do sistema. A Figura 6 apresenta um trecho simplificado em `JavaScript` que ilustra a aplicação do padrão.

```
1   async create() {
2       // ...
3       const negotiationDetailsUpdated = new NegotiationDetailsBuilder(
            negotiationDetails)
4         .setApproved(false)
5         .setReplied(false)
6         .setReplyDate(null)
7         .setRequestDate(null)
8         .setRequested(false)
9         .setPrice(price)
10        .setUnity(unity)
11        .setStartDate(startDate)
12        .setEndDate(endDate)
13        .setObservations(observations)
14        .build();
15      // ...
16       return await NegotiationDetailsRepository.update(negotiationDetailsData.id,
             negotiationDetailsUpdated);
17  }
```

Figura 6. Exemplo de Implementação do Padrão `Builder`

5.3.4. Command

O padrão *Command* foi adotado para organizar o tratamento das requisições entre o *front-end* e o *back-end*, em especial no processo de execução de `fetchs` e funções de `handle`. Cada requisição foi encapsulada em uma classe que implementa uma interface comum denominada `Command`, responsável por definir a operação a ser executada. Dessa forma, requisições puderam ser representadas como comandos independentes, desacoplando a lógica de invocação da lógica de execução. A Figura 7 apresenta um diagrama de classes simplificado da implementação do padrão no sistema.

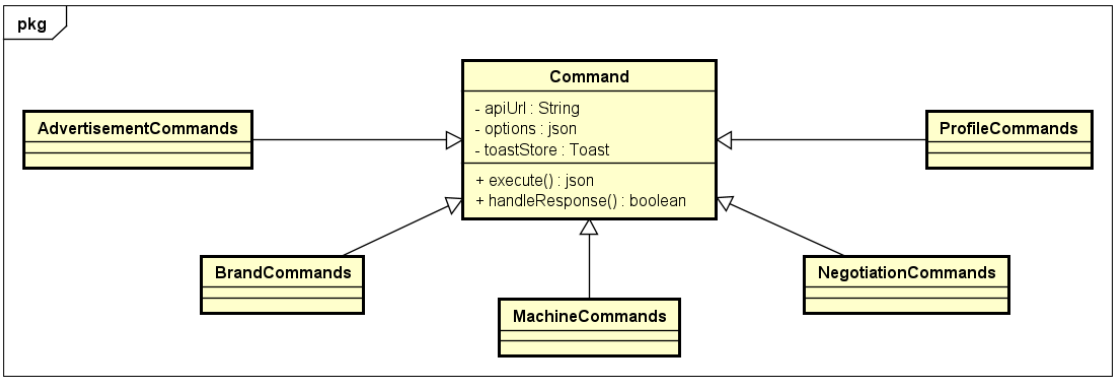

Figura 7. Diagrama de Classes da Implementação do Padrão `Command`

5.3.5. Observer

O padrão *Observer* encontra aplicação natural no *front-end* da plataforma, em virtude dos mecanismos reativos providos pelo `Svelte`. Diferentemente de outros padrões implementados explicitamente no código, o comportamento de observação é nativo do *framework* e se manifesta principalmente através das declarações reativas (utilizando o operador `$:`). O trecho de código a na Figura 8 ilustra o uso do padrão.

No exemplo apresentado, a expressão reativa monitora continuamente as variáveis `inputPopupBrand` e `inputPopupTypes`, funcionando como um mecanismo de observação automática. Sempre que uma delas sofre alteração, o bloco associado é reexecutado, resultando na busca assíncrona dos modelos correspondentes no repositório de dados. Em seguida, o array `models` é atualizado com as novas informações, e, em consequência, todos os componentes da interface que dependem dessa coleção são notificados e renderizados novamente. Esse comportamento garante maior responsividade do sistema, reduz a necessidade de chamadas manuais de atualização e reforça a aderência ao paradigma reativo adotado pelo `Svelte`.

6. Discussão e Análise

A adoção dos padrões de projeto no desenvolvimento da plataforma mostrou-se um exercício relevante para estruturar e organizar o código de maneira mais robusta. O padrão *Repository* permitiu centralizar o acesso aos dados. O *State* contribuiu para modelar o ciclo de vida das negociações. O *Builder* demonstrou ser adequado para a criação de objetos

```
    // ...
    $: if (inputPopupBrand && inputPopupTypes) {
        (async () => {
            let brandId = brands.find(
                (brand) => brand.label === inputPopupBrand,
            )?.value;
            let typeId = types.find(
                (type) => type.label === inputPopupTypes,
            )?.value;

            let modelsData = await fModel.getByBrandIdAndTypeId(
                brandId,
                typeId,
            );
            models = modelsData.map((type) => ({
                label: type.name,
                value: type.id,
            }));
        })();
    }
    // ...
```

Figura 8. Exemplo de implementação do padrão `Observer`

complexos. O *Command* possibilitou encapsular requisições ao servidor de forma modular, simplificando a manutenção e extensibilidade. Já o *Observer*, embora não tenha sido implementado de forma manual, foi aproveitado através dos recursos do `Svelte`.

Os padrões empregados trouxeram diversos benefícios para o projeto. Em primeiro lugar, contribuíram para uma maior organização estrutural do código, com clara separação de responsabilidades entre classes e módulos, o que favoreceu a reutilização de trechos de lógica em diferentes partes do sistema, reduzindo redundâncias. Outro ponto relevante foi a maior facilidade em realizar manutenções ou ajustes incrementais, já que o uso dos padrões forneceu uma base mais estável e previsível. Em termos pedagógicos, a prática consolidou o entendimento teórico dos padrões ao vinculá-los a um artefato real, o que reforçou a importância de seu uso em sistemas de software.

Apesar dos avanços obtidos, alguns desafios e limitações foram identificados durante o processo. Em certos casos, a aplicação dos padrões ocorreu de forma simplificada, o que reduziu o pleno aproveitamento de suas potencialidades em cenários mais complexos. Observou-se também que a escolha do padrão nem sempre se mostrou totalmente aderente ao problema, exigindo adaptações que alteraram parcialmente a fidelidade às implementações clássicas. Além disso, o estudo restringiu-se a um conjunto específico de padrões e situações de uso, o que limita a generalização dos resultados e deixa em aberto a análise de sua efetividade em sistemas de maior escala ou em domínios distintos.

7. Considerações Finais

Este trabalho apresentou um relato de experiência sobre a aplicação de padrões de projeto no desenvolvimento de uma plataforma digital voltada ao setor do agronegócio. A adoção de soluções como *Repository*, *State*, *Builder*, *Command* e *Observer* contribuiu para a organização estrutural do código, a separação de responsabilidades e a maior clareza arquitetural, além de reforçar o aprendizado prático dos conceitos teóricos envolvidos. Os resultados obtidos evidenciam que o uso de padrões pode oferecer ganhos significativos em termos de manutenibilidade, reutilização e legibilidade, ao mesmo tempo em que se

mostraram uma oportunidade de integração entre teoria e prática em um artefato real.

Apesar dos benefícios observados, o estudo apresenta algumas ameaças à sua validade. Entre elas, destacam-se a aplicação dos padrões em um contexto restrito, tanto em termos de domínio quanto de escala, o que limita a generalização dos resultados. Além disso, a adaptação de certos padrões ao problema específico pode ter reduzido sua fidelidade às implementações clássicas. Trabalhos futuros podem ampliar a análise incluindo outros padrões, avaliando sua adoção em sistemas de maior porte e explorando métricas quantitativas para complementar as reflexões qualitativas apresentadas. Dessa forma, o estudo contribui para o entendimento dos benefícios, potencialidades e limitações do uso de padrões de projeto em soluções *full stack*.

Agradecimentos

À FAPERGS, ao CNPq e à Pró-Reitoria de Pesquisa e Pós-Graduação (PROPPI) da Unipampa pelo suporte financeiro.

Referências

Ali, M. and Elish, M. O. (2013). A comparative literature survey of design patterns impact on software quality. In *Proceedings of the ICISA*.

Brown, S. (2021). *Software Architecture for Developers*. Leanpub.

do Amaral Santos, M. G., de A Souza, M. R., and Figueiredo, E. (2016). Padrões de projeto em java: Um estudo prático sobre a utilização e benefícios. In *Anais do I Workshop sobre Aspectos Sociais, Humanos e Econômicos de Software*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston, MA.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*.

Horita, F., Graciano Neto, V., and dos Santos, R. (2018). *Design Science Research em Sistemas de Informação e Engenharia de Software: Conceitos, Aplicações e Trabalhos Futuros*.

Manik, L. P. (2019). Design pattern evaluation on a restful api wrapper: A case study of software integration with an internet payment gateway using model-driven architecture. *Journal of Information Technology and Computer Science*.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Shalloway, A. and Trott, J. R. (2004). *Design patterns explained*. Addison-Wesley, 2 edition.

Tran, V. T., Abdellatif, M., and Guéhéneuc, Y.-G. (2021). Formalising Solutions to REST API Practices as Design (Anti)Patterns. In *Proceedings of the Service-Oriented Computing*. Springer International Publishing.

Wendt, R., Tiadoro, E., Basso, F., and Bernardino, M. (2025). Bridging the gap in agricultural sharing economy: A systematic review for evaluating information systems for machinery efficiency. In *Proceedings of the 27th International Conference on Enterprise Information Systems - Volume 2: ICEIS*. SciTePress.