Explorando Estilos Arquiteturais para Servir Sistemas Inteligentes

Washington Luiz Meireles de Lima
Ygor Tavela Alves da Silva

Proposta de Trabalho de Formatura Supervisionado
apresentado à disciplina
MAC0499

Orientadores:
Prof. Dr. Alfredo Goldman
Me. Renato Cordeiro Ferreira

São Paulo, Abril de 2022

Conteúdo

Capítulo 1

Introdução

Um sistema de software que possui uma inteligência capaz de evoluir e melhorar com o tempo, particularmente analisando como os usuários interagem com o sistema, é referido como um sistema inteligente (Hulten , 2019). Sistemas inteligentes podem possuir variadas finalidades: uma simples tradução feita no Google Tradutor, recomendações de playlists com músicas adequadas ao perfil de qualquer pessoa com o Spotify, até mesmo revolucionando a pesquisa em biocomputação com o AlphaFold (Ewen Callaway , 2022). Tais sistemas são semelhantes à sistemas da informação tradicionais, principalmente no que diz respeito a ter um objetivo e entregar valor aos seus usuários. No entanto, a inteligência oriunda de modelos de aprendizado de máquina caracterizam unicamente tais sistemas.

Dentro do ciclo de vida de um sistema inteligente, podemos dividir uma aplicação de aprendizado de máquina em três eixos de mudança (Sato **et al.** , 2019): dados, modelo, e código. Os dados e modelos, concedem uma característica particular aos sistemas inteligentes. O sistema se encontra em um estado de constante evolução, o que o torna mais complexo, mais difícil de entender e, mais difícil de testar. Tal fato influencia a concepção da arquitetura de software para sistemas inteligentes, isto é, como o eixo de código deve se estruturar para comportar as mudanças promovidas ao longo do tempo pelos outros dois eixos.

Assim como sistemas de software tradicionais, o processo de design de sistemas inteligentes não deve garantir apenas que o sistema funcione com um propósito claro, mas também, que seja um sistema robusto, barato de manter, e com um longo tempo de vida. Essas características estão diretamente relacionados com a concepção de uma boa arquitetura de software. A arquitetura de software de um sistema é uma *forma* dada para o sistema por aqueles que o constroem. A estrutura dessa *forma* se traduz em como os componentes do sistema são divididos, como os componentes estão dispostos e, como os componentes se relacionam entre si. Tal *forma* tem como objetivo garantir uma base sólida para o sistema, proporcionando um sistema fácil de entender, desenvolver e servir para o cliente (Martin , 2017).

Dentro da arquitetura de um sistema inteligente, surge um importante questionamento relacionado à escalabilidade (Lakshmanan **et al.** , 2020): *Como servir um modelo de tal forma que ele suporte milhões de requisições de predições em um curto período de tempo?* Para abordar tal questionamento, é fundamental entender a comunicação entre quem serve e quem consome o modelo. Comumente, a principal característica de uma comunicação a se determinar é a sua natureza temporal – síncrona ou assíncrona, e a partir disso escolher o protocolo mais adequado para a comunicação.

Dadas as peculiaridades de um sistema inteligente e as necessidades de se construir uma boa arquitetura de software, o objetivo desta pesquisa é o de explorar padrões arquiteturais para servir um sistema inteligente, buscando avaliar cenários e *trade-offs* entre os padrões adotados. Consequentemente, este projeto visa encontrar os principais prós e contras de cada abordagem, além de melhor discutir a aplicabilidade de cada um dos padrões.

Este documento descreve a proposta de desenvolvimento de alternativa para servir um sistema inteligente. No Capítulo 2, serão discutidos conceitos relevantes para o que está sendo discutido. No

?? é detalhado a proposta deste trabalho. No **??**, é apresentado um plano de projeto para executar o trabalho.

Capítulo 2

Revisão de Literatura

2.1 Sistemas Inteligentes

Sistemas Inteligentes são aqueles em que existe alguma inteligência (utilizando técnicas de inteligência artificial ou aprendizado de máquina) aprendendo e evoluindo com dados (Hulten , 2019). Por este motivo, a implementação de um sistema inteligente impõem diferentes exigências que os diferenciam de sistemas não inteligentes. O ciclo de vida de tais sistemas incluem: como criar a inteligência, como mudar a inteligência, como organizar a inteligência, e como lidar com erros ao longo do tempo. Para isso, construir um sistema inteligente efetivo requer balancear cinco componentes principais:

- Como definir um objetivo que seja claro;
- Como apresentar a saída dos modelos de aprendizado aos usuários;
- Como executar a inteligência;
- Como criar uma inteligência que cumpra com o seu objetivo;
- Como orquestrar o ciclo de vida da inteligência.

2.2 Aprendizado de Máquina

A área da inteligência artificial, ou IA, assim como na filosofia e psicologia, busca entender o funcionamento de agentes inteligentes e como contruí-los (Russel e Norvig , 2012). Algumas definições de IA podem se agrupar em quatro categorias de sistemas:

1. que pensam como humanos,
2. que pensam racionalmente,
3. que agem como humanos, e
4. que agem racionalmente.

As definições 1 e 3 se baseiam na capacidade humana e em estudos empíricos, envolvendo hipóteses e experimentos. Por outro lado, as definições 2 e 4 baseiam-se no conceito ideal de inteligência, tido como racionalidade, no qual combinam-se matemática e engenharia. Dentre essas definições, a quarta é onde se enquadra o Aprendizado de Máquina.

O Aprendizado de Máquina é uma área de IA que se preocupa em construir algoritmos através de dados, os quais podem vir da natureza, criados pelos humanos ou gerados por outros algoritmos. Pode ser definido também pelo processo de coleta de um conjunto de dados e pelo treinamento de um modelo estatístico usando esse conjunto através de algum algorítimo de aprendizado.

2.3 Arquitetura de Software

A arquitetura de software define as partes que compõem o software, como são as sua estruturas e como elas se relacionam entre si. A estrutura de um software, é o que permite que ele seja flexível o suficiente para que rapidamente evolua e mude o seu comportamento para atender uma dada necessidade, ou seja, é o que o torna maleável o suficiente para deixar o maior número de opções disponíveis pelo maior tempo possível (Martin , 2017).

2.3.1 Por quê uma boa arquitetura é importante?

Além do próprio valor entregue pela solução do software em si, uma outra ótica a se avaliar o valor de um software se da pela sua estrutura (Martin , 2017). Muitas vezes por não ser algo aparente aos usuários finais ou, pelo custo de tempo e esforço, a arquitetura acaba sendo deixada de lado no processo de desenvolvimento do software. Desta forma, é importante notar que o design de arquitetura não se refere apenas ao processo inicial de desenvolvimento, mas sim à todo ciclo de vida de um sistema.

Via de regra, um maior tempo de vida de sistema sempre irá beneficiar o uso de boas práticas de desenvolvimento, independentemente de todo gasto com tempo e esforço para o desenvolvimento de um software de alta qualidade (Martin Fowler , 2019). Com isso, a principal finalidade da construção da arquitetura é reduzir custos de desenvolvimento, aumentando a compressão do código pelos programadores, melhorando a manutenibilidade do sistema, facilitando o entendimento da divisão entre requisitos importantes e requisitos que são detalhes ao sistema, etc.

2.4 Princípios de Design

Princípios de design nos dizem como devemos organizar as funções e as estruturas de dados em agrupamentos, e como esses agrupamentos devem estar interconectados (Martin , 2017). Basicamente, os princípios de projeto promovem diretrizes gerais que podem ou não serem seguidas durante o desenvolvimento de um software, com o objetivo de melhorar a estrutura de um sistema de software.

2.4.1 Princípio da Responsabilidade Única

O princípio pode ser condensado na seguinte frase (Martin , 2017):

"Um módulo deve ser responsável por um, e apenas, um ator"

Um "módulo" pode ser classificado como um arquivo fonte, ou então – em algumas linguagens, como um conjunto coeso de funções e estruturas de dados. O "responsável" diz respeito ao fato de que o módulo só deve ter uma razão para mudar, e pensando num sistema de software, tais razões para mudança sempre serão requerido por um "ator" (um usuário do sistema por exemplo).

2.4.2 Princípio Aberto-Fechado

O princípio estabelece que o comportamento de um artefato de software deve buscar ser extensível, de tal forma que não haja modificações neste artefato (Martin , 2017). Apesar da simples declaração, tal princípio tem um grande valor na arquitetura de sistemas, já que a prática de tal princípio faz com que os componentes sejam separados em como, porque, e quando eles mudam. Desta forma, os componentes ficam organizados numa hierarquia de dependências que protege componentes de alto-nível de mudanças provocadas por componentes de baixo-nível.

2.4.3 Princípio da Segregação de Interface

O princípio estabelece que nenhum código deve ser forçado a depender em métodos que ele não usa (Martin , 2017). Em linguagens estaticamente tipadas, como o Java, essa preposição faz bastante sentido se pensarmos na implementação de interfaces. Tal fato, pode conduzir para o pensamento equivocado que o princípio resolve um problema exclusivo a um paradigma de linguagem específica. No entanto, o principio é intimamente ligado a um problema de arquitetura, já que a dependência em coisas que não precisamos pode causar problemas não esperados. Um exemplo de um problema arquitetural relacionado a isso, é o caso de dependências transitivas em que indiretamente um sistema pode depender coisas que não deveria por depender de algo que depende dessas coisas.

2.4.4 Princípio da Inversão de Dependência

O princípio estabelece que os sistemas mais flexíveis são aqueles cujas dependências do código fonte se referem apenas a abstrações, e não concretizações (Martin , 2017). Apesar de parecer uma declaração radical ao indicar que deve-se apenas depender de abstrações, os elementos concretos se referem apenas aos elementos *voláteis* do sistema, ou seja, aqueles em que há constante desenvolvimento e passam por mudanças frequentes. Tal princípio tem o seu valor, principalmente pelo fato de que abstrações são estáveis em relações a mudanças, o que possibilita dividir o sistema em dois componentes: um abstrato e outro concreto.

2.5 Padrões de Design

Padrões de design é uma maneira barata, rápida, e eficiente de resolver problemas comuns em programação (Beck , 2007), características as quais diferenciam os padrões aos princípios de design (2.4). Cada padrão engloba um problema em específico, com a discussão de fatores que afetam o problema e um conselho de como resolver o problema de forma rápida para criar uma solução satisfatória. No design de arquitetura de software (2.3), os padrões são uma importante ferramenta para os desenvolvedores, contribuindo para reduzir o gasto de tempo e energia para definir uma forma de resolver um problema. No paradigma de programação orientada a objetos, um padrão de design pode ser classificado a partir do seu propósito (Gamma **et al.** , 1996):

1. Padrão de criação: Abstraí o processo de instanciação de objetos, tornando o sistema independente de como os objetos são criados, compostos, e representados.

2. Padrão estrutural: Explica como classes e objetos são compostos para formar estruturas maiores, de tal forma que sejam mantidos a flexibilidade e a eficiência da mesma.

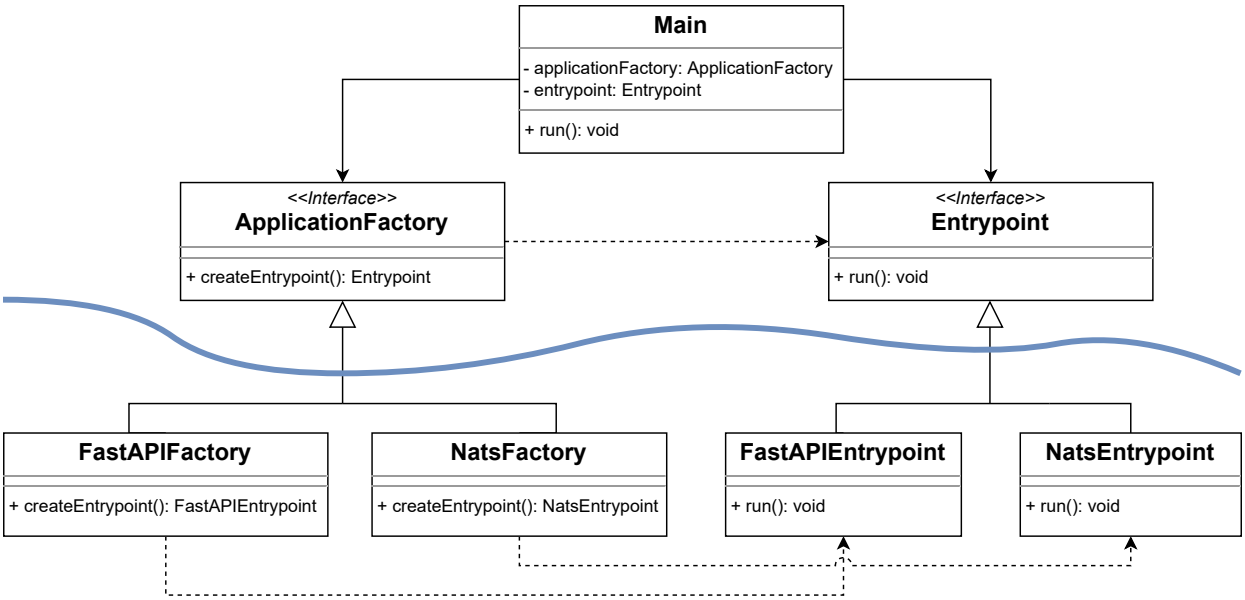
3. Padrão comportamental: Explica como organizar as responsabilidades e a comunicação entre objetos.

2.5.1 Padrão Método Fábrica

O padrão *Método Fábrica* é um padrão de criação, comumente utilizado no paradigma de orientação a objetos. Uma interface é definida para criar um objeto, no entanto, as subclasses decidem qual classe instanciar (Gamma **et al.** , 1996). O padrão, via de regra, segue os princípios de segregação de interface (2.4.3) e o de inversão de dependência (2.4.4), o que torna o padrão útil para definir uma fronteira entre classes abstratas e concretas. Desta forma, naturalmente as dependências do código fonte são invertidas contra o fluxo de controle do código.

Na figura (2.1) temos um exemplo de aplicação do padrão, é possível notar a linha azul que define uma fronteira entre o que é abstrato e o que é concreto. Um outro ponto importante a se notar, é que a classe *Main* depende apenas de abstrações, o que facilita a extensão da aplicação para qualquer outro tipo de *Entrypoint*, desde que haja uma *Factory* adequada e a injeção da dependência dessa nova *Factory* dentro da classe *Main*.

Figura 2.1: *Aplicação Padrão Método Fábrica*



2.5.2 Padrão *Publisher-Subscriber*

O padrão *Publisher-Subscriber* ajuda a manter o estado de componentes concorrentes sincronizados (Bushmann **et al.** , 1996). Para permitir isso, o padrão habilita a propagação de mudanças unidirecionalmente: um *publisher* notifica um número qualquer de *subscribers* sobre a mudança do seu estado. Esse padrão é baseado no padrão comportamental *Observador* descrito para aplicações em programação orientada a objetos (Gamma **et al.** , 1996). No entanto, a sua definição pode ser estendida para diferentes tipos de sistemas distribuídos, pois permite uma alternativa desacoplada para notificar qualquer mudança de estado – do *publisher* – para uma quantidade qualquer de *subscribers*.

2.5.3 Padrão Injeção de Dependência

O padrão *Injeção de Dependência* é um padrão aplicado principalmente em linguagens orientada a objetos. A sua ideia se baseia na existência de um objeto separado – um *assembler*, cuja função é a de popular as dependências de uma classe apropriadamente (Martin Fowler , 2004). Desta forma, é possível evitar o acoplamento de classes com depedências em classes concretas, como a do exemplo abaixo em que é instanciado uma classe concreta *ModelGateway* é instanciada diretamente no construtor da classe *PredictUseCase*:

```
class PredictUseCase():
    def __init__(self):
        self.model_gateway = ModelGatewayImpl()
```

O padrão fornece uma forma de inversão de controle do fluxo do código que reduz o acoplamento das classes com as suas dependências, já que separa a instanciação dos objetos de dependência com o seu uso em si. São definidos três tipos de *Injeção de Dependência*: Injeção de construtor, de interface, e *setter*. Ajustando o exemplo acima para aplicar a injeção de construtor, passamos a desacoplar a instanciação da dependência *ModelGateway* do construtor da classe *PredictUseCase*:

```
class PredictUseCase():
    def __init__(self, model_gateway: ModelGateway):
        self.model_gateway = model_gateway
```

2.6 Estilos Arquiteturais

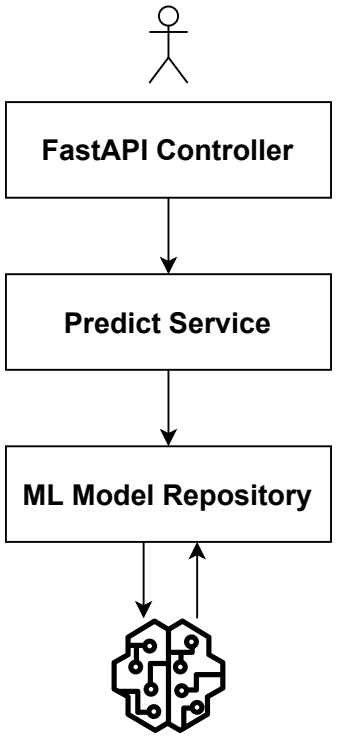
Um estilo de arquitetura é a inspiração que está por trás da ideia da arquitetura de software (2.3). Tal inspiração é definida como um conjunto de princípios (2.4) e padrões (2.5) a serem adotados para construir uma estrutura que atenda os requisitos funcionais e não funcionais do sistema. Mais especificamente, um estilo arquitetural determina o *vocabulário* de componentes e as formas de interação entre eles que podem ser usadas nesse estilo, em um conjunto de restrições (Garlan e Shaw , 1993).

2.6.1 Arquitetura em Camadas

A arquitetura em camadas é um dos estilos mais simples e de menor custo a ser adotado. Neste estilo arquitetural, são criados camadas horizontais com diferentes responsabilidades cujas dependências devem ser sempre no sentido das camadas adjacentes (Martin , 2017). Normalmente são definidas três camadas principais (Martin Fowler , 2015): a camada de apresentação (UI), a camada da lógica de negócios, e a camada de acesso aos dados.

Um exemplo de aplicação de um sistema inteligente que emprega o estilo arquitetural em camadas pode ser visualizado na figura (2.2). Na figura o *FastAPI Controller* é responsável por apresentar o sistema para os clientes via (2.9.1), o *Predict Service* é responsável por realizar todo tipo de tratamento necessário a entrada de dados e, por fim, buscar o modelo adequado do *ML Model Repository* para realizar a predição adequada.

Figura 2.2: *Exemplo de Arquitetura em Camadas aplicada num sistema inteligente*



2.6.2 Arquitetura Hexagonal

A arquitetura hexagonal é um estilo arquitetural que busca separar as regras de negócio da aplicação dos detalhes de implementação, tais como o *framework*, a interface de usuário, banco de dados, etc (Cockburn , 2005). A ideia fundamental se baseia no uso de *Portas* e *Adaptadores*, que são conceitos que se baseiam fortemente nos princípios de responsabilidade única (2.4.1), aberto-fechado (2.4.2), segregação de interface (2.4.3), e inversão de dependência (2.4.4). Além destes princípios, um padrão que é muito importante para o estilo, é o de injeção de dependências (2.5.3).

Tal padrão possibilita o cumprimento do princípio da *Regra da Dependência* (Martin , 2017), na qual as dependências sempre devem apontar para camadas mais internas, que por sua vez devem ser completamente ignorantes em relação às camadas externas.

Em relação aos conceitos fundamentais do estilo, as *Portas* são definidas como um ponto de entrada e saída independente do consumidor para dentro/fora da aplicação, em muitas linguagens um porta será nada mais que uma simples interface que não possui nenhum conhecimento da implementação concreta até ser injetada em tempo de execução. Enquanto que, os *Adaptadores* são classes que adaptam uma interface em outra, e podem ser classificados de duas formas:

- Adaptador Condutor (ou Primário): São adaptadores que dependem de uma porta e injetam uma implementação concreta da porta, que no caso é um caso de uso do domínio da aplicação a ser utilizado.
- Adaptador Conduzido (ou Secundário): São adaptadores que são uma implementação concreta de uma porta e são injetados no domínio da aplicação. Vale destacar que o domínio apenas conhece a porta que foi implementada pelo adaptador, ou seja, o domínio conhece apenas a interface da porta.

Um dos principais problemas que esse estilo arquitetural se propõe a resolver é a de criar uma fronteira clara entre o que é regra de negócio e o que é detalhe para o sistema. Diferentemente do que ocorre na Arquitetura em Camadas (2.6.1), o domínio não possui conhecimento do que é externo a ele, ou seja, o que é interno abstraí totalmente a infraestrutura externa como banco de dados, controladores de API REST, sistemas de mensageria, etc.

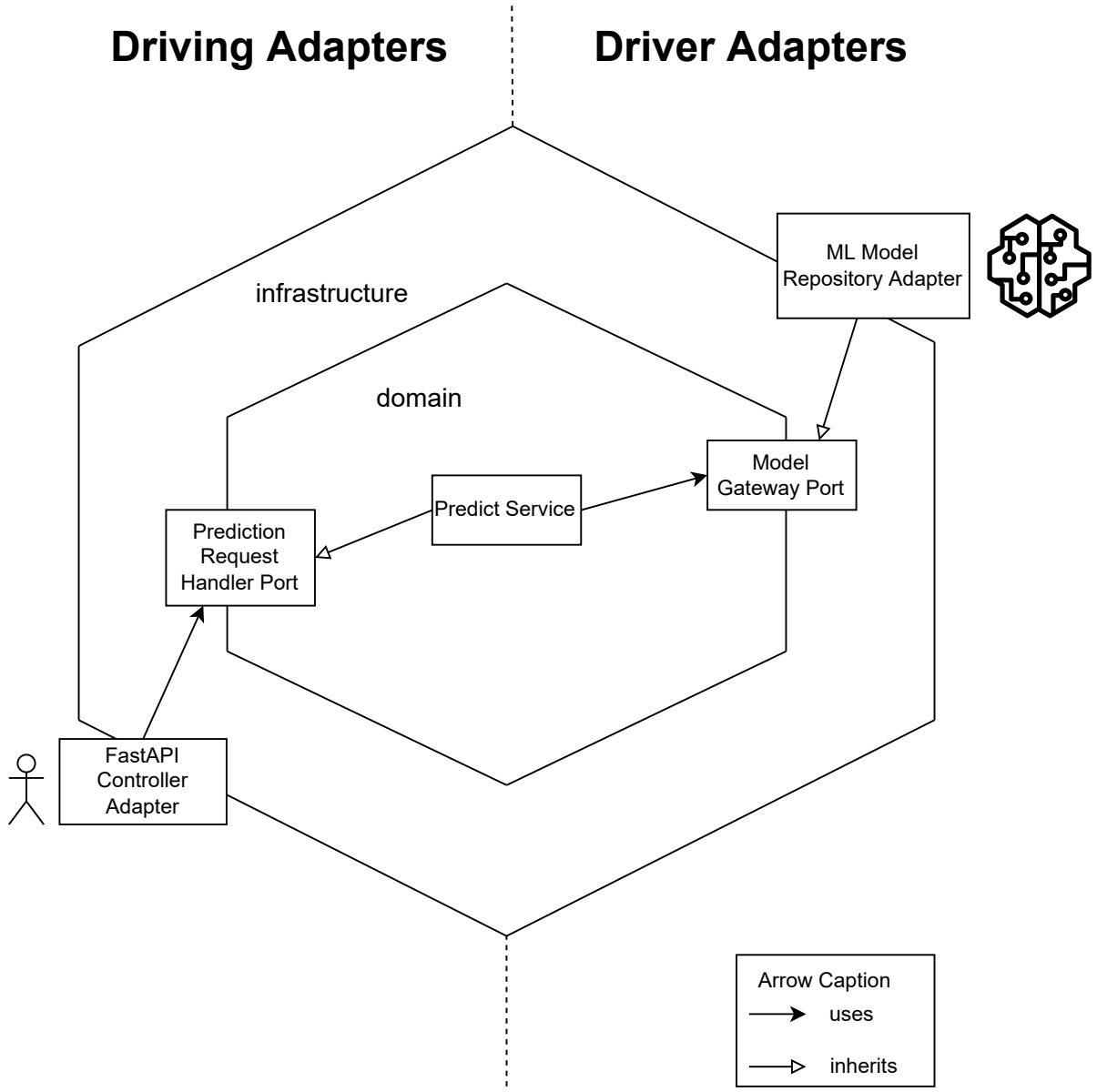
Na figura (2.3) é demonstrado uma aplicação do estilo num sistema inteligente análogo ao exemplo da Arquitetura em Camadas (2.2). O aumento da complexidade da aplicação fica bastante claro, o que é um fator considerável ao considerar tal estilo. No entanto, a distinção bem definida entre o que é infraestrutura e o que é domínio é o que torna a arquitetura hexagonal um estilo de grande valor.

2.7 Modelo C4

O modelo C4, é um modelo que se baseia num conjunto hierárquico de diagramas para representar visualmente uma arquitetura de software (Brown , 2018). Tais diagramas podem ser utilizados para descrever a arquitetura em diferentes camadas, onde cada um dos diagramas têm uma utilidade para cada audiência. O modelo C4 considera as estruturas estáticas de um sistema de software em termos de *contêineres*, *componentes* e *código*. Além disso, o modelo considera as pessoas que utilizam tais sistemas que construímos, ou seja, o *contexto* de uso do sistema. Mais especificamente, são utilizados 4 diagramas que seguem a seguinte hierarquia:

1. Contexto: Representa como o sistema se encaixa no mundo real em termos de como as pessoas e como outros sistemas de software interagem com o sistema.
2. Contêiner: Representa as aplicações, base de dados, microsserviços, etc, que compõem o sistema de software.
3. Componente: Representa um contêiner individualmente, destacando cada um dos seus componentes que o compõem.
4. Código: Representa um componente individualmente a nível de código (por exemplo, um diagrama UML de classes numa arquitetura que se baseia no paradigma de orientação a objetos).

Figura 2.3: *Exemplo de Arquitetura Hexagonal aplicada num sistema inteligente*



2.8 API

API é o acrônimo de *Application Programming Interface*, sendo um conjunto de definições e protocolos usados no desenvolvimento e na integração de aplicações. Pode ser entendida como um contrato entre o provedor e o consumidor, em que o primeiro estabelece padrão de comunicação e o segundo usa esse padrão para executar ações, obter ou transferir informações.

2.9 Padrões de comunicação

A integração entre diferentes aplicações via rede lida com diversos desafios: conexão não confiável e lenta, diferenças nas aplicações, e mudanças inevitáveis. Algumas das formas de integração são: arquivo de transferência, banco de dados compartilhado, invocação de procedimento remoto, e troca de mensagens. Todas essas abordagens têm vantagens e desvantagens, por isso, não é incomum que uma aplicação use múltiplas formas de integração afim de minimizar essas desvantagens (Hohpe e Woolf , 2012).

2.9.1 *API REST*

O protocolo *REST*, *Representational State Transfer*, é definido por um conjunto de restrições de arquitetura (Fielding , 2000): interface uniforme, cliente-servidor, *stateless*, *cache*, sistema em camadas e código por demanda. Dentre essas restrições, a interface uniforme é que faz a distinção da arquitetura *REST* das demais. Nela, são aplicados os princípios de generalização da engenharia de software para os componentes de interface, promovendo uma simplificação na arquitetura do sistema e maior visibilidade das interações. Para garantir a uniformidade da interface, quatro restrições são impostas: identificação dos recursos, manipulação dos recursos por representações, mensagens auto descritivas, e hipermídia como motor do estado da aplicação.

Uma importante característica de uma *API REST* é a sua natureza síncrona de comunicação, o que cria um grande acoplamento entre o cliente e o servidor. Um grande benefício, no entanto, é a sua baixa complexidade para implementação.

2.9.2 Mensageria

Mensageria é uma forma de comunicação que se baseia no uso de um sistema responsável por coordenar e gerir o envio e recebimento de mensagens, assim como a leitura e persistência das mesmas (Hohpe e Woolf , 2012), uma característica destes sistemas é a aplicação do padrão de design *Publisher-Subscriber* (2.5.2).

O padrão de comunicação é naturalmente assíncrono, tendo em vista que o *publisher* e o *subscriber* não estabelecem uma comunicação direta. Tal fato contribui para a redução do acoplamento, o que pode melhorar, por exemplo, a questão de disponibilidade do sistema. Todavia, uma desvantagem é que para estabelecer uma comunicação via mensagens é necessário o uso de um sistema de mensageria, o que aumenta o grau de dependência do sistema que o utiliza.

Bibliografia

Beck (2007) Kent Beck. **Implementation Patterns**. Citado na pág. 5

Brown (2018) Simon Brown. The C4 Model for Software Architecture, 6 2018. URL https://www.infoq.com/articles/C4-architecture-model/. Citado na pág. 8

Bushmann et al. (1996) F Bushmann, R Meunier, H Rohnert e Soft Ware Architecture. **Pattern-Oriented Software Architecture**, volume 1. Citado na pág. 6

Cockburn (2005) Alistair Cockburn. Hexagonal Architecture, 2005. URL https://alistair.cockburn.us/hexagonal-architecture/. Citado na pág. 7

Ewen Callaway (2022) Ewen Callaway. What's next for AlphaFold and the AI protein-folding revolution. URL https://www.nature.com/articles/d41586-022-00997-5. Citado na pág. 1

Fielding (2000) Roy Thomas Fielding. **Architectural Styles and the Design of Network-based Software Architectures**. Tese de Doutorado. Citado na pág. 10

Gamma et al. (1996) E Gamma, R Helm, R Johnson e J Vlissides. Design Patterns: Elements of Reusable Software. **Addison-Wesley Professional Computing Series**. ISSN ISBN: 0-201-63361-2. Citado na pág. 5, 6

Garlan e Shaw (1993) David Garlan e Mary Shaw. An Introduction to Software Architecture. doi: 10.1142/9789812798039{_}0001. Citado na pág. 7

Hohpe e Woolf (2012) Gregor Hohpe e Bobby Woolf. **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (Google eBook)**. Citado na pág. 9, 10

Hulten (2019) Geoff Hulten. **Building Intelligent Systems**. Apress, Berkeley, CA. ISBN 978-1-4842-3933-9. doi: 10.1007/978-1-4842-3933-9. Citado na pág. 1, 3

Lakshmanan et al. (2020) Valliappa Lakshmanan, Sara Robinson e Michael Munn. **Machine Learning Design Patterns: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps** . Citado na pág. 1

Martin (2017) Robert C Martin. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Citado na pág. 1, 4, 5, 7, 8

Martin Fowler (2004) Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 1 2004. URL https://martinfowler.com/articles/injection.html. Citado na pág. 6

Martin Fowler (2015) Martin Fowler. Presentation Domain Data Layering, 8 2015. URL https://martinfowler.com/bliki/PresentationDomainDataLayering.html. Citado na pág. 7

Martin Fowler (2019) Martin Fowler. Is High Quality Software Worth the Cost?, 2019. URL https://martinfowler.com/articles/is-quality-worth-cost.html. Citado na pág. 4

Russel e Norvig (2012) Stuart Russel e Peter Norvig. **Artificial intelligence—a modern approach 3rd Edition**. doi: 10.1017/S0269888900007724. Citado na pág. 3

Sato et al. (2019) Danilo Sato, Arif Wider e Christoph Windheuser. Continuous Delivery for Machine Learning. **Martin Fowler**. Citado na pág. 1