UNIVERSIDADE POSITIVO
Análise e Desenvolvimento de Sistemas

38263297 Kalil Maciel Pock
38988879 Matheus Gustavo Saldanha Folle
38693500 Matheus Müller dos Santos

Documento de Arquitetura de Software: Sistema de Academia

Curitiba
2025

Kalil Maciel Pock
Matheus Gustavo Saldanha Folle
Matheus Müller dos Santos

Documento de Arquitetura de Software: Sistema de Academia

Documento de Arquitetura de Software (DAS) apresentado à disciplina de Arquitetura de Software do Curso de Análise e Desenvolvimento de Sistemas da Universidade Positivo como requisito parcial para obtenção da nota da A2.
Orientador: Prof. Marlon Andre Peron Generoso

Curitiba
2025

RESUMO

O objetivo deste trabalho é projetar e documentar a arquitetura de software para um Sistema Gerenciador de Academia , como parte da avaliação da disciplina de Arquitetura de Software. O foco principal é criar uma solução com alta manutenibilidade (RNF02) e baixo acoplamento entre os componentes , permitindo futuras evoluções, como a troca da interface de usuário. A metodologia adotada incluiu o levantamento de Requisitos Funcionais (RF) e Não Funcionais (RNF) , que foram modelados através de Estórias de Usuário e Casos de Uso detalhados. Para a modelagem visual, foram utilizados diagramas UML, especificamente o Diagrama de Classes, focado exclusivamente no Modelo (Model) para evitar poluição, e o Diagrama de Sequência, detalhando o fluxo principal de "Cadastrar Cliente". A arquitetura escolhida foi o padrão Model-View-Controller (MVC) , e foram analisados Padrões de Projeto GoF aplicáveis ao contexto, como Factory, Singleton, Adapter e Composite . Como resultado, foi gerado este Documento de Arquitetura de Software (DAS), que detalha a estrutura do sistema. Adicionalmente, um protótipo de código em Java foi desenvolvido , apresentando recortes (screenshots) que comprovam a correta transição de responsabilidades da camada View (captura de dados) , para o Controller (orquestração) e deste para o Model (regras de negócio e persistência) . Conclui-se que a arquitetura MVC é a solução mais robusta e adequada para o escopo do projeto, atendendo plenamente ao requisito de manutenibilidade e provando ser uma alternativa mais simples e coesa do que Microsserviços para este cenário.

Palavras-chave: Arquitetura de Software; MVC; Padrões de Projeto; UML; Manutenibilidade.

ABSTRACT

The objective of this assessment is to design and document the software architecture for a Gym Management System, as part of the evaluation for the Software Architecture discipline. The main focus is to create a solution with high maintainability (RNF02) and low coupling between components, allowing for future evolutions, such as changing the user interface. The methodology adopted included surveying Functional Requirements (RF) and Non-Functional Requirements (RNF), which were modeled through User Stories and detailed Use Cases. For visual modeling, UML diagrams were used, specifically the Class Diagram, focused exclusively on the Model to avoid clutter, and the Sequence Diagram, detailing the main flow of "Register Customer". The chosen architecture was the Model-View-Controller (MVC) pattern, and applicable GoF Design Patterns were analyzed within the context, such as Factory, Singleton, Adapter, and Composite . As a result, this Software Architecture Document (DAS) was generated, detailing the system's structure. Additionally, a Java code prototype was developed, presenting snippets (screenshots) that prove the correct transition of responsibilities from the View layer (data capture) , to the Controller (orchestration) , and from this to the Model (business rules and persistence). It is concluded that the MVC architecture is the most robust and suitable solution for the project's scope, fully meeting the maintainability requirement and proving to be a simpler and more cohesive alternative than Microservices for this scenario.

Keywords: Software Architecture; MVC; Design Patterns; UML; Maintainability.

LISTA DE FIGURAS

Figura 1 - Diagrama de Casos de Uso **11**
Figura 2 - Diagrama de Classes **13**
Figura 3 - Diagrama de Sequência **14**
Figura 4 - Interação com o Usuário **17**
Figura 5 - A transição entre View e Controller **17**
Figura 6 - Relacionamento entre Controller e Model **17**

LISTA DE TABELAS

Tabela 1– Detalhamento do Caso de Uso: Descrição **11**
Tabela 2 – Detalhamento do Caso de Uso: Fluxo Principal **12**
Tabela 3 – Detalhamento do Caso de Uso: Fluxo Alternativo **12**

SUMÁRIO

1. INTRODUÇÃO **8**

2. DESENVOLVIMENTO **9**

2.1 Requisitos **9**

2.1.1 Requisitos Funcionais (RF) - (Item 1) 9

2.1.2 Estórias do Usuário (US) - (Item 2) 9

2.1.3 Requisitos Não Funcionais (RNF) 10

2.2 Modelagem de Casos de Uso **10**

2.2.1 Diagrama dos Casos de Uso 10

2.2.2 Detalhamento do Caso de Uso 11

2.3. Modelagem de Classes e Sequência **13**

2.3.1 Diagrama de Classes (Modelo) 13

2.3.2 Diagrama de Sequência (UC001 - Cadastrar Cliente) 14

2.4 Arquitetura e Padrões de Projeto **15**

2.4.1 Justificativa da Arquitetura (MVC) 15

2.4.2 Padrões de Projeto (GOF) Aplicados 15

2.5 Protótipo da Arquitetura (Item 7) **16**

2.5.1 Recorte 1: A View (Interação com Usuário) 16

2.5.2 Recorte 2: A Transição (View -> Controller) 17

2.5.3 Recorte 3: O Controller (Controller -> Model) 17

3. CONCLUSÃO **18**

REFERÊNCIAS **19**

1. INTRODUÇÃO

O presente Documento de Arquitetura de Software (DAS) detalha o projeto de um Sistema Gerenciador de Academia, desenvolvido como requisito avaliativo da disciplina de Arquitetura de Software no curso de Análise e Desenvolvimento de Sistemas da Universidade Positivo. O principal objetivo deste trabalho é aplicar os conceitos de arquitetura e padrões de software estudados em sala de aula na modelagem e documentação de um sistema coeso e manutenível.

O sistema proposto visa gerenciar as operações básicas de uma academia, permitindo o cadastro e gerenciamento de clientes, exercícios e planos de treino. Para atender aos requisitos funcionais e não funcionais levantados, optou-se pela utilização do padrão arquitetural Model-View-Controller (MVC) , visando principalmente a separação de responsabilidades e o baixo acoplamento entre os componentes, facilitando futuras manutenções e evoluções.

A metodologia empregada envolveu a especificação de requisitos através de Requisitos Funcionais (RF) e Estórias de Usuário (US) , seguida pela modelagem visual utilizando a Linguagem de Modelagem Unificada (UML), com a elaboração de Diagramas de Casos de Uso, Diagrama de Classes (focado no Modelo) e Diagrama de Sequência para o fluxo principal. Adicionalmente, foram analisados e justificados Padrões de Projeto (GoF) aplicáveis ao contexto, e um protótipo de código em Java foi apresentado para demonstrar a transição entre as camadas da arquitetura MVC.

Este documento está estruturado da seguinte forma: a Seção 1 apresenta esta Introdução. A Seção 2 detalha o Desenvolvimento do projeto, subdividida em Requisitos (2.1), Modelagem de Casos de Uso (2.2), Modelagem de Classes e Sequência (2.3), Arquitetura e Padrões de Projeto (2.4) e Protótipo da Arquitetura (2.5). Por fim, a Seção 3 apresenta a Conclusão do trabalho.

2. DESENVOLVIMENTO

2.1 Requisitos

Nesta seção, detalhamos os Requisitos Funcionais (Item 1), as Estórias de Usuário (Item 2) e os Requisitos Não Funcionais do sistema.

2.1.1 Requisitos Funcionais (RF) - (Item 1)

A seguir, listamos os principais requisitos funcionais do sistema (Item 1), utilizando o formato Verbo + Substantivo, conforme exemplificado nos materiais da disciplina:

- **Manter Clientes:** Permitir o cadastro, consulta, atualização e remoção de clientes.
- **Manter Exercícios:** Permitir o cadastro, consulta e remoção de exercícios no catálogo.
- **Manter Treino:** Permitir a montagem, consulta, modificação e remoção de treinos.
- **Consultar Exercícios:** Permitir a visualização do catálogo de exercícios disponíveis.
- **Avaliar Treino:** Permitir (opcionalmente) que o cliente avalie um treino realizado.

*(**Nota:** Estes requisitos são modelados em detalhe na Seção 2: Modelagem de Casos de Uso)*

2.1.2 Estórias do Usuário (US) - (Item 2)

As estórias a seguir modelam os requisitos funcionais (Item 2) sob a perspectiva dos atores, conforme solicitado:

Ator: Administrador (Gerente/Instrutor)

- **US01**: Como Administrador, eu quero cadastrar um novo cliente no sistema, informando seus dados pessoais e objetivo (ex: Hipertrofia), para que ele possa ter um registro e montar seus treinos.
- **US02**: Como Administrador, eu quero listar todos os clientes cadastrados, para ter uma visão geral de quem está ativo na academia.
- **US03**: Como Administrador, eu quero buscar um cliente específico por ID, para consultar ou atualizar suas informações rapidamente.
- **US04**: Como Administrador, eu quero remover o cadastro de um cliente, para manter a base de dados atualizada com alunos ativos.
- **US05**: Como Administrador, eu quero cadastrar um novo exercício na base do sistema, para que ele fique disponível na montagem de treinos.
- **US06**: Como Administrador, eu quero listar todos os exercícios disponíveis, para gerenciar o catálogo de atividades da academia.
- **US07**: Como Administrador, eu quero remover um exercício obsoleto, para manter o catálogo de exercícios relevante.

Ator: Cliente (Aluno)

- **US08**: Como Cliente, eu quero montar um novo treino, selecionando exercícios do catálogo, para organizar minha rotina de atividades.

- **US09**: Como Cliente, eu quero consultar meus treinos salvos, para poder executá-los na academia.
- **US10**: Como Cliente, eu quero remover um treino antigo, para manter minha lista de treinos atualizada com meus objetivos.
- **US11**: Como Cliente, eu quero adicionar exercícios a um treino existente, para ajustar ou evoluir minha rotina.
- **US12**: Como Cliente, eu quero remover exercícios de um treino existente, para modificar minha rotina.

2.1.3 Requisitos Não Funcionais (RNF)

Os seguintes requisitos descrevem os atributos de qualidade e restrições técnicas do sistema, que influenciam diretamente a escolha da arquitetura.

- **RNF01** (**Desempenho**): O sistema deve responder a consultas (como listar clientes ou exercícios) em menos de 2 segundos, mesmo com um aumento no volume de dados.
- **RNF02 (Manutenibilidade):** A arquitetura do sistema deve permitir fácil manutenção e evolução. Deve ser possível alterar a interface do usuário (ex: de console para web) com o mínimo impacto possível nas regras de negócio. (Justificativa para o Baixo Acoplamento do MVC).
- **RNF03 (Segurança):** O acesso às funções de gerenciamento (Manter Cliente, Manter Exercício) deve ser restrito a usuários autenticados com o perfil de **Administrador**.
- **RNF04 (Disponibilidade):** O sistema deve ter uma disponibilidade de 99%, garantindo que possa ser acessado por clientes e administradores durante o horário de funcionamento da academia.
- **RNF05 (Usabilidade):** O sistema deve ter menus claros e um fluxo de navegação intuitivo, minimizando a curva de aprendizado para novos administradores.

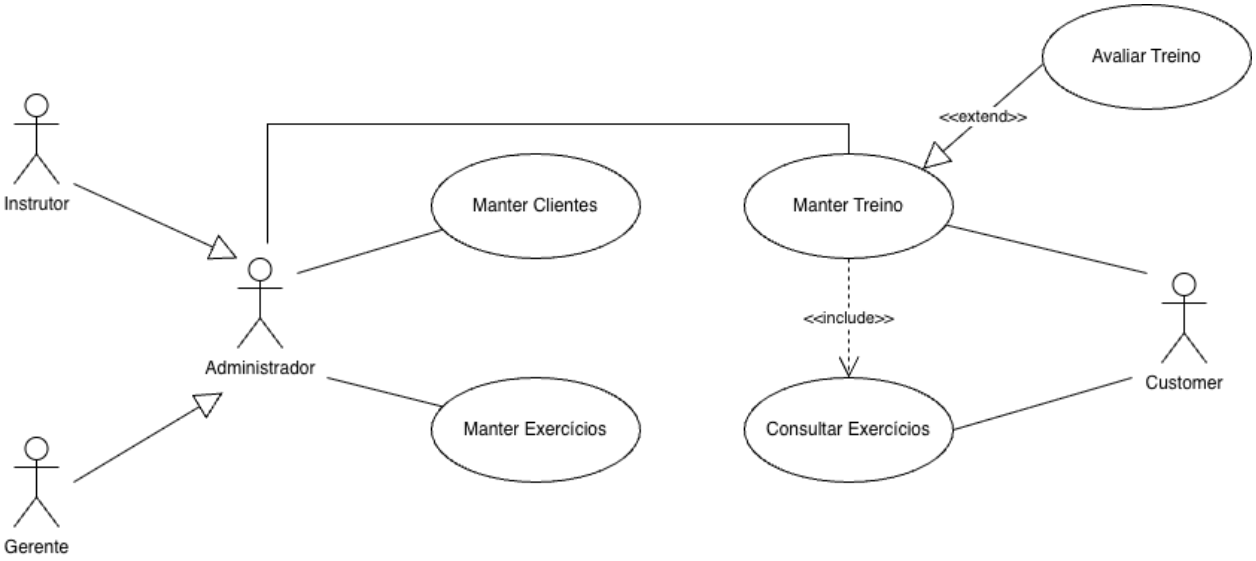
2.2 Modelagem de Casos de Uso

Esta seção apresenta o diagrama de Casos de Uso geral do sistema (Item 2), seguido pelo detalhamento tabular do principal Caso de Uso de fluxo do sistema: Cadastrar Cliente (Item 4).

2.2.1 Diagrama dos Casos de Uso

O diagrama a seguir ilustra as principais interações dos atores Administrador e Cliente com o sistema, incluindo os relacionamentos de Herança, Inclusão (<<include>>) e Extensão (<<extend>>), conforme os requisitos da disciplina.

Figura 1 - Diagrama de Casos de Uso



2.2.2 Detalhamento do Caso de Uso

A seguir, detalhamos o Caso de Uso Cadastrar Cliente utilizando o formato tabular.

Caso de Uso: UC001 - Cadastrar Cliente

Tabela 1– Detalhamento do Caso de Uso: Descrição

Campo	**Descrição**
Requisitos Relacionados	US01
Descrição Detalhada	Permite ao Administrador registrar um novo cliente na base de dados do sistema, coletando suas informações pessoais e objetivo.
Pré-Condição	O Administrador deve estar autenticado no sistema (RNF03).
Pós-Condição de Sucesso	O novo Cliente é criado e persistido na lista de clientes do sistema.
Pós-Condição de Falha	O sistema informa um erro (ex: formato de data inválido) e o cliente não é registrado.
Atores Principais	Administrador
Atores Secundários	N/A
Gatilho (Trigger)	Administrador seleciona a opção "1 - Cadastrar cliente" no Menu de Clientes.

Fonte: Elaborado pelos autores (2025).

Fluxo Principal

Tabela 2 – Detalhamento do Caso de Uso: Fluxo Principal

Passo	Ação
01	O Administrador seleciona a opção "1 - Cadastrar cliente" no menu (CustomerView).
02	O Sistema (CustomerView) solicita: "Nome do cliente: ".
03	O Administrador insere o nome.
04	O Sistema (CustomerView) solicita: "Data de nascimento (dd/MM/yyyy): ".
05	O Administrador insere a data de nascimento (ex: "10/05/1990").
06	O Sistema (CustomerView) solicita: "Objetivo: 1 - Hipertrofia...".
07	O Administrador seleciona o número correspondente ao objetivo (ex: "1").
08	O Sistema (CustomerView) invoca o CustomerController (método registerCustomer), passando os dados coletados.
09	O Sistema (CustomerController) instancia um novo objeto Customer, passando os dados e a lista de clientes (para geração do ID).
10	O Sistema (CustomerController) adiciona o novo objeto Customer à lista customers.
11	O Sistema (CustomerView) exibe a mensagem: "Cliente cadastrado com sucesso."
12	O caso de uso é finalizado.

Fonte: Elaborado pelos autores (2025).

Fluxo Alternativo (A2.1 - Data em Formato Inválido)

Tabela 3 – Detalhamento do Caso de Uso: Fluxo Alternativo

Passo	Ação
05.1	O Administrador insere uma data em formato inválido (ex: "10-20-1990").
05.2	O Sistema (CustomerView) captura a exceção DateTimeParseException no bloco try-catch.
05.3	O Sistema (CustomerView) exibe a mensagem de erro: "Formato inválido. Use o formato dd/MM/yyyy."
05.4	O sistema encerra a operação de cadastro e retorna o Administrador ao menu anterior (o Menu de Clientes), permitindo que ele inicie a operação novamente.

Fonte: Elaborado pelos autores (2025).

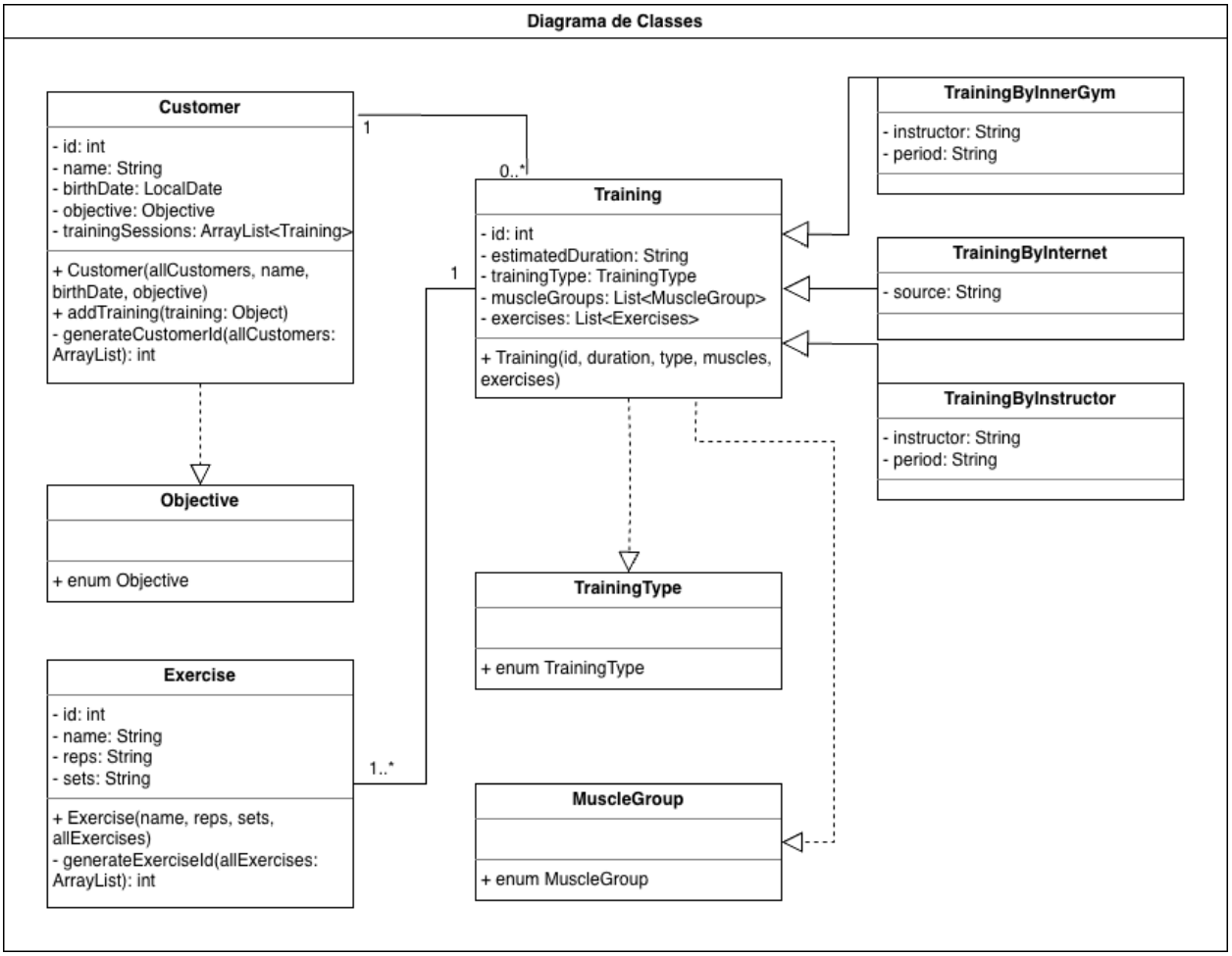
2.3. Modelagem de Classes e Sequência

Esta seção apresenta os artefatos visuais que detalham a estrutura de dados (Diagrama de Classes) e o fluxo de interação (Diagrama de Sequência) da arquitetura MVC.

2.3.1 Diagrama de Classes (Modelo)

Conforme as boas práticas de MVC e a solicitação do professor, o diagrama de classes foca exclusivamente nas classes que compõem o **Modelo (Model)** do sistema. As classes de View e Controller foram omitidas intencionalmente para não poluir o diagrama e focar na estrutura de dados e regras de negócio.

Figura 2 - Diagrama de Classes



Fonte: Elaborado pelos autores (2025).

Legenda dos Relacionamentos:

- **Generalização:** As classes TrainingBy... herdam da classe Training.
- **Associação 1..*:** Um Customer pode ter vários Training (0..*), e um Training é composto por vários Exercise (1..*).
- **Dependência (uso):** As classes principais utilizam os Enums Objective, TrainingType e MuscleGroup.

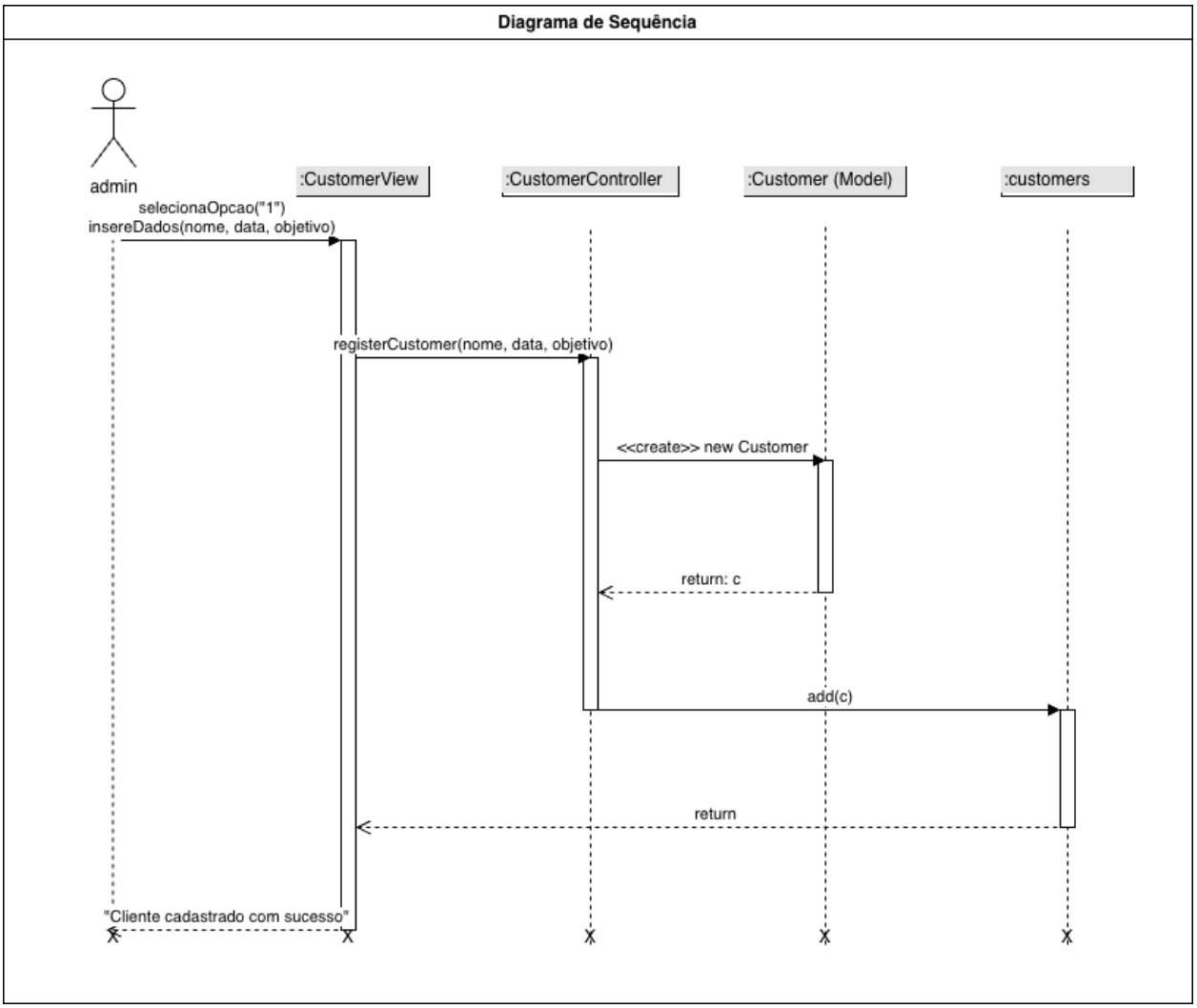
2.3.2 Diagrama de Sequência (UC001 - Cadastrar Cliente)

O diagrama a seguir detalha o fluxo de interação do Caso de Uso Cadastrar Cliente (Item 4). Ele demonstra a separação de responsabilidades da arquitetura MVC:

1. A View (CustomerView) é responsável por capturar os dados do ator.
2. A View aciona o Controller (CustomerController), passando os dados.
3. O Controller orquestra a lógica: ele instancia o Model (Customer) e o persiste (na lista customers).
4. O Controller não devolve dados complexos para a View, apenas confirma a operação.

Note a seta de retorno pontilhada (passo 4 no diagrama abaixo, correspondente ao passo 9 do Fluxo Principal do UC), que representa o **retorno de dados** (a instância c do novo cliente) do Model para o Controller, conforme solicitado nos requisitos da disciplina.

Figura 3 - Diagrama de Sequência



Fonte: Elaborado pelos autores (2025).

2.4 Arquitetura e Padrões de Projeto

Esta seção detalha as decisões de design do sistema, justificando a escolha da arquitetura (Item 5) e os Padrões de Projeto (Item 6) aplicados.

2.4.1 Justificativa da Arquitetura (MVC)

O sistema foi projetado utilizando o padrão arquitetural **Model-View-Controller (MVC)**.

A escolha pelo MVC se baseia principalmente no Requisito Não Funcional de **Manutenibilidade (RNF02)**. O objetivo era construir um sistema com **baixo acoplamento** entre os componentes, facilitando futuras evoluções.

No padrão MVC:

- O **Model** (ex: classes Customer, Training) encapsula os dados e as regras de negócio, sem ter conhecimento de como será exibido.
- A **View** (ex: classes CustomerView, ExerciseView) é responsável por toda a interação com o usuário (no nosso caso, o console), sem conter regras de negócio.
- O **Controller** (ex: CustomerController) atua como o intermediário, recebendo requisições da View e manipulando o Model para responder.

A principal vantagem desta separação é a flexibilidade. Conforme o RNF02, se no futuro decidirmos substituir a View atual (console System.out.println) por uma interface Web (HTML/CSS), **nenhuma alteração seria necessária no Model**. Apenas o Controller seria ajustado para se comunicar com a nova View, provando o baixo acoplamento e a alta manutenibilidade da arquitetura.

Análise Comparativa: MVC vs. Microsserviços

Embora Microsserviços ofereçam alta escalabilidade e tolerância a falhas, sua adoção traria uma complexidade de infraestrutura (gerenciamento de rede, deploy de múltiplos serviços, consistência de dados) desnecessária para o escopo deste projeto.

Para um sistema de gerenciador de academia (provavelmente de pequeno ou médio porte), a arquitetura MVC já atende plenamente aos requisitos de manutenibilidade e desempenho (RNF01), representando uma solução mais simples, coesa e robusta para o problema proposto.

2.4.2 Padrões de Projeto (GOF) Aplicados

Conforme solicitado (Item 6), selecionamos um conjunto de padrões de projeto (GOF) que se aplicam ou poderiam se aplicar ao sistema para resolver problemas comuns de design.

1. Factory Method (Método de Fábrica)

- **Onde se aplica:** Na classe TrainingFactory.
- **Justificativa:** O sistema possui diferentes tipos de treinos (TrainingByInnerGym, TrainingByInternet, TrainingByPersonal), que são "filhos" da classe Training. O padrão **Factory Method** é usado aqui para encapsular a lógica de criação desses objetos. O Controller não precisa saber *como* criar um treino específico; ele apenas pede à TrainingFactory: "me dê um treino do tipo 'Internet'", e a fábrica cuida da instanciação correta.

2. Singleton (Instância Única)

- **Onde se aplica:** (Fictício, conforme permissão) Em uma classe DatabaseConnection.java.
- **Justificativa:** Para gerenciar o acesso ao banco de dados, o padrão **Singleton** garantiria que *apenas uma instância* da conexão com o banco fosse criada e compartilhada por todo o sistema. Isso economiza recursos (evitando abrir e fechar múltiplas conexões) e previne inconsistências de dados.

3. Adapter (Adaptador)

- **Onde se aplica:** (Fictício, conforme permissão) Em uma classe CatracaLegadaAdapter.java.
- **Justificativa:** Se o nosso sistema precisasse se integrar a um hardware legado (como uma catraca de academia com uma API antiga e incompatível), usaríamos um **Adapter**. Ele atuaria como um "tradutor", convertendo as chamadas do nosso Controller moderno para o formato que a API antiga da catraca entende, permitindo que sistemas incompatíveis trabalhem juntos.

4. Composite (Composto)

- **Onde se aplica:** Na relação entre Training e Exercise.
- **Justificativa:** O padrão **Composite** permite tratar objetos individuais (um Exercise) e composições de objetos (uma List<Exercise>) da mesma maneira. No nosso sistema, uma classe Training é um "composto" que possui uma List<Exercise> (as "partes"). Esse padrão facilita a manipulação de estruturas em árvore (ex: um treino é feito de exercícios), permitindo que um Training calcule sua duração total simplesmente somando a duração de suas partes.

2.5 Protótipo da Arquitetura (Item 7)

Esta seção apresenta o protótipo de código que demonstra a transição entre as camadas da arquitetura **Model-View-Controller (MVC)**.

O fluxo demonstrado é o do Caso de Uso UC001 - Cadastrar Cliente. Conforme a solicitação do professor por "prints" ou "recortes", usaremos **três screenshots (imagens)** do código-fonte para ilustrar a sequência exata da transição entre as camadas.

2.5.1 Recorte 1: A View (Interação com Usuário)

O primeiro print, da classe CustomerView.java, mostra a camada **View** interagindo com o usuário. Ela é responsável por exibir o menu, capturar a entrada (Scanner) e coletar os dados, sem possuir qualquer regra de negócio.

Figura 4 - Interação com o Usuário

```
14  public abstract class CustomerView {
15      public static void menu(CustomerController controller, Scanner entry) {
16          boolean itsExecuting = true;
17
18          while (itsExecuting) {
19              System.out.println(x:"\nMenu de Clientes");
20              System.out.println(x:"1 - Cadastrar cliente");
21              System.out.println(x:"2 - Listar clientes");
22              System.out.println(x:"3 - Buscar cliente por ID");
23              System.out.println(x:"4 - Remover cliente");
24              System.out.println(x:"0 - Voltar");
25              System.out.print(s:"Escolha uma opção: ");
26
27              String opcao = entry.nextLine();
28
29              switch (opcao) {
```

2.5.2 Recorte 2: A Transição (View -> Controller)

O segundo print mostra a linha-chave onde a **View** (ainda na CustomerView.java) finaliza seu trabalho e aciona o **Controller**, passando os dados que acabou de coletar. Esta é a fronteira exata entre a primeira e a segunda camada da arquitetura.

Figura 5 - A transição entre View e Controller

```
67
68                  controller.registerCustomer(nome, nascimento, objetivo);
69                  System.out.println(x:"Cliente cadastrado com sucesso.");
70                  break;
71
```

2.5.3 Recorte 3: O Controller (Controller -> Model)

O terceiro print é do método registerCustomer dentro da classe CustomerController.java. Ele mostra o **Controller** recebendo os dados da View, orquestrando a lógica de negócio e manipulando o **Model**: ele instancia um novo objeto Customer e o persiste na lista customers.

Figura 6 - Relacionamento entre Controller e Model

```
7   public class CustomerController {
8       private ArrayList customers;
9
10      public CustomerController(ArrayList listOfCustomers) {
11          this.customers = (listOfCustomers != null) ? listOfCustomers : new ArrayList();
12      }
13
14      public void registerCustomer(String name, LocalDate birthDate, Objective objective) {
15          Customer c = new Customer(customers, name, birthDate, objective);
16          customers.add(c);
17      }
18
```

3. CONCLUSÃO

O desenvolvimento deste Documento de Arquitetura de Software permitiu aplicar de forma prática os conceitos da disciplina ao projetar um Sistema Gerenciador de Academia. A metodologia incluiu a definição detalhada de requisitos funcionais e não funcionais, modelados através de Estórias de Usuário e Casos de Uso, e a utilização de diagramas UML para representar as visões lógica (Diagrama de Classes) e de processo (Diagrama de Sequência).

A escolha pela arquitetura MVC mostrou-se acertada para o escopo do projeto, priorizando a manutenibilidade e o baixo acoplamento , conforme justificado na análise comparativa com Microsserviços . A análise de Padrões de Projeto GoF (Factory, Singleton, Adapter, Composite) enriqueceu a solução proposta, demonstrando como problemas comuns de design podem ser abordados. O protótipo de código apresentado comprovou a viabilidade da arquitetura escolhida, ilustrando a correta transição de responsabilidades entre as camadas View, Controller e Model .

Conclui-se que o trabalho atingiu seus objetivos, resultando em uma arquitetura de software bem documentada, coesa e alinhada às boas práticas, pronta para guiar uma futura implementação ou servir como base para evoluções do sistema.

REFERÊNCIAS