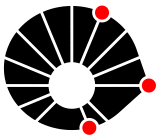

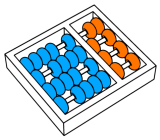

Grupo Caramelo: Sistema de gerenciamento de academia

Ana Clara Zoppi Serpa - RA 165880
Bruno de Marco Apolonio - RA 195036
Gabriel Oliveira dos Santos - RA 197460
Lucas Costa de Oliveira - RA 182410
Vítor Mosso Dario - RA 207024

Technical Report - IC-18-01 - Relatório Técnico
July - 2018 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Caramelo: Sistema de gerenciamento de academia

Ana Clara Zoppi Serpa Bruno de Marco Apolonio Gabriel Oliveira dos Santos
Lucas Costa de Oliveira Vítor Mosso Dario

Resumo

Neste relatório abordamos o processo de desenvolvimento do projeto de MC302 ao longo deste semestre. Utilizamos a ferramenta GitHub para controle de versões do projeto e um grupo no WhatsApp para nos comunicar e discutir o andamento do mesmo, devido às diferenças de disponibilidade entre os membros. Dividimos o relatório em seções de acordo com as etapas de desenvolvimento designadas pelo professor (Etapa 0, Etapa 1, Etapa 2, Etapa 3, Etapa 4 e *Release* Final) e descrevemos o que foi implementado/decidido em cada uma delas e por quê.

Pudemos aprender muito com esse projeto, vimos uma ideia simples se expandir para algo mais complexo e aplicamos os conceitos de Orientação a Objetos aprendidos em sala. Isso foi fundamental para melhor compreensão desses conceitos e fixação dos conhecimentos adquiridos.

Sumário

1 Etapa 0: Proposta de Software

Nessa etapa, decidimos qual seria o tema do nosso projeto e quais suas funcionalidades. Optamos por implementar um sistema de gerenciamento de academias com cadastro de clientes, atividades e instrutores e a associação destes, por exemplo: o cliente Gabriel faz pilates de segunda e quarta, das 16h às 17h, e o instrutor responsável pelas aulas de pilates desse horário é o Lucas.

Escrevemos um documento descrevendo as funcionalidades que previámos para o sistema e entregamos para o professor. Como elas já foram descritas no documento, aqui são mencionadas mais brevemente:

- Cadastro dos clientes que frequentam a academia
- Alteração dos dados de um cliente já cadastrado
- Desativar um cliente
- Cadastro de atividades que a academia oferece
- Alteração dos dados da atividade, por exemplo, preço
- Remoção de uma atividade
- Associar/desassociar horários às atividades oferecidas
- Associar/desassociar clientes aos horários e atividades
- Cadastrar instrutores que trabalham na academia
- Alterar dados de um instrutor cadastrado
- Desativar um instrutor
- Consultar qual atividade possui mais clientes
- Consultar qual atividade possui menos clientes
- Consultar qual atividade tem o maior preço
- Consultar qual atividade tem o menor preço

Em vez de remover clientes e instrutores, optamos por desativá-los. Tivemos essa ideia pensando no seguinte cenário: um cliente decide deixar de frequentar a academia, mas, após algum tempo, retorna. Seria prático se já tivéssemos os dados dele e bastasse reativá-lo. O mesmo ocorre com instrutores que parem de trabalhar na academia, mas depois retornem. Ou, por exemplo, a academia talvez tenha interesse em tentar contactar um instrutor para contratá-lo novamente, ou um cliente antigo para perguntar se ele deseja retornar.

2 Etapa 1: Classes do sistema

Nessa etapa, pensamos sobre as classes e relacionamentos entre elas e construímos um diagrama UML. O professor indicou correções, as quais realizamos em outras etapas. O diagrama a seguir está parcialmente correto e o apresentamos apenas para facilitar a compreensão das ideias.

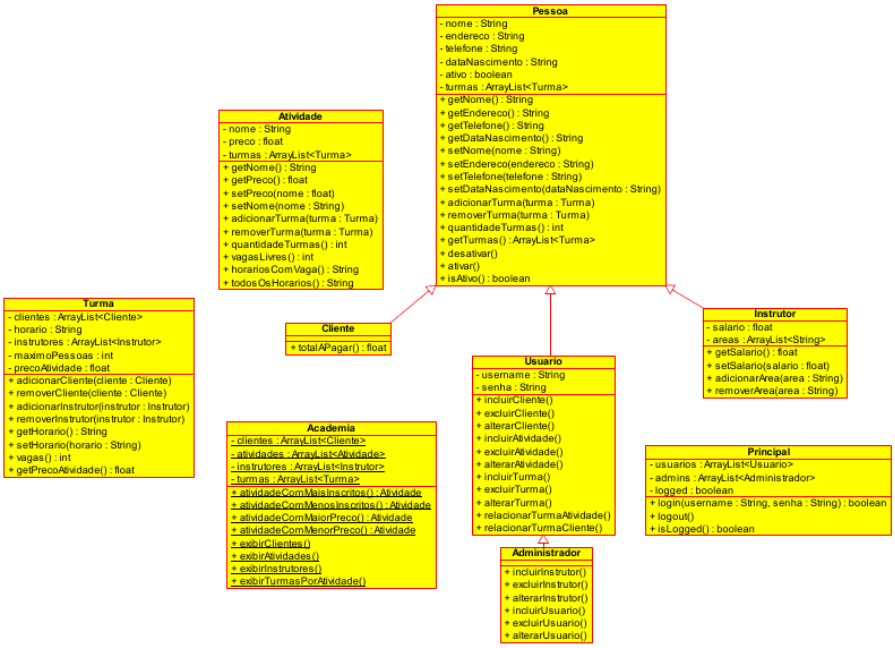

Figura 1: Diagrama UML para a *release* 0

Decidimos que o relacionamento entre cliente, horário, atividade e instrutor seria feito por meio de uma nova classe, a classe Turma, da seguinte forma:

- Uma atividade possui n turmas.
- Uma turma possui n clientes, n instrutores e um horário que representaremos com o tipo String.
- Um cliente possui n turmas, afinal, ele pode realizar várias atividades diferentes.
- Um instrutor possui n turmas porque ele pode dar aulas em várias turmas diferentes.

Percebemos que clientes e instrutores possuiriam nome, endereço, telefone, data de nascimento, lista de turmas e a indicação de que estão ou não ativos no sistema como atributos em comum. Portanto, decidimos criar a classe Pessoa com esses atributos, *getters* e *setters* e fazer com que Cliente e Instrutor fossem suas subclasses.

O cliente possui, além dos métodos e atributos da classe Pessoa, o método que calcula quanto ele precisa pagar pelas atividades realizadas.

Como uma atividade possui n turmas, mas as turmas possuem apenas clientes, instrutores e horário, ou seja, a turma não sabe qual é a sua atividade, decidimos, por questões de facilidade, guardar o preço da atividade na classe Turma também. Dessa forma, para calcular quanto o cliente precisa pagar, basta iterar por sua lista de turmas e acumular os preços.

O instrutor possui, além dos métodos e atributos da classe Pessoa, um atributo salário e uma lista de áreas (que representamos com uma lista de Strings) nas quais atua.

A classe Academia seria nossa base de dados, guardando em listas todos os clientes, instrutores, atividades e turmas do sistema e possuindo métodos para exibí-los.

Decidimos também implementar uma etapa de *login* no sistema e separar os usuários entre usuários comuns e usuários administradores. Um usuário comum poderia incluir/alterar/remover/desativar clientes, atividades e turmas, enquanto um usuário administrador poderia fazê-lo também para instrutores e usuários. Portanto, Administrador é uma subclasse de Usuário.

Usuários possuem *username* e senha. Nessa etapa do projeto, pensamos em fazer com que Usuário fosse subclasse de Pessoa, para guardarmos também seus dados (nome, endereço, telefone, data de nascimento).

3 Etapa 2: *Release* 0

Para essa *release*, iniciamos a implementação seguindo o diagrama UML construído na Etapa 1. Criamos as classes em Java, seus métodos e implementamos alguns deles, deixando outros vazios para serem implementados posteriormente.

4 Etapa 3: *Release* 1

Para essa *release*, implementamos os métodos que, na anterior, estavam vazios. Realizamos algumas mudanças ao perceber alternativas e necessidades não previstas.

Percebemos que não precisaríamos guardar nome, telefone, endereço e data de nascimento dos usuários do sistema, apenas *username* e senha para controlar o acesso. Portanto, Usuário e Administrador deixaram de ser subclasses de Pessoa.

Decidimos, em vez de ter as classes Usuário e Administrador, ter apenas a classe Gerenciador e um *enum* (Permissões) para controlar o que cada gerenciador pode fazer. O enum contém dois valores possíveis: COMUM e ADMIN. A classe Gerenciador possui métodos que refletem as operações que o gerenciador realiza ao usar o sistema, sendo elas:

- Incluir/alterar/desativar cliente.
- Incluir/alterar/desativar instrutor - apenas se a permissão for ADMIN.
- Incluir/alterar/remover um gerenciador - apenas se a permissão for ADMIN.
- Ver lista de gerenciadores do sistema - apenas se a permissão for ADMIN.

  As operações que podem ser realizadas apenas por ADMIN foram implementadas com um *if*. Se a permissão é ADMIN, os comandos seguintes correspondentes à operação são executados. Se a permissão não é ADMIN, é exibido um aviso.
- Incluir/alterar/remover atividade.
- Incluir/alterar/remover turma.
- Relacionar um cliente a uma turma, ou seja, incluí-lo naquela turma.
- Relacionar um instrutor a uma turma, ou seja, colocá-lo como responsável por aquela turma.
- Desrelacionar clientes/instrutores de uma turma.

- Verificar quais clientes estão em uma turma específica.
- Verificar quais clientes realizam uma atividade específica.
- Verificar quais instrutores cuidam de turmas de uma dada atividade.
- Consultar quantidade de clientes que realizam uma atividade.
- Consultar turmas de um cliente/instrutor.

A classe Principal contém uma lista (*ArrayList*) de gerenciadores e uma instância de Gerenciador para guardar qual gerenciador está logado. Na tentativa de *login*, procuramos um gerenciador com o *login* e a senha informados pelo usuário. Se existe, o gerenciador logado passa a ser esse que encontramos e exibimos uma mensagem (menu) com as operações possíveis. Conforme o gerenciador digita o número correspondente à operação, chamam-se os métodos da classe Gerenciador correspondentes. O código possui um *switch case* que chama os métodos correspondentes aos números das operações. Por exemplo, seja 1 a operação de incluir um cliente: o código chama *gerenciadorLogado.incluirCliente()*.

Os métodos da classe Gerenciador imprimem mensagens na tela solicitando os dados necessários, criando objetos a partir deles e então chamando métodos da classe BaseDados que alteram suas listas de clientes, instrutores e atividades conforme adequado. A classe que guardaria as listas de entidades do sistema antes se chamava Academia, mas mudamos seu nome para BaseDados.

Por exemplo, para uma inclusão de cliente, o fluxo seria o seguinte:

- O método da classe Gerenciador é chamado.
- Um objeto Cliente é criado com base nos dados fornecidos.
- O método da classe BaseDados é chamado, incluindo o cliente no *ArrayList* de clientes que a instância de BaseDados possui.

Durante a implementação deles, percebemos que precisávamos de um identificador único para as turmas criadas, para que pudéssemos associar os clientes e instrutores a elas e excluí-las. Por conta disso, acrescentamos à classe Turma um atributo ID, a ser informado pelo gerenciador durante a criação da turma, junto com a atividade e o horário. Também decidimos acrescentar à classe Pessoa o atributo RG, para que fosse possível identificar unicamente cada cliente e instrutor para encontrá-los nas operações de associação a turmas, exclusão e alteração de dados.

Para desativar um cliente, o fluxo seria o seguinte:

- O método da classe Gerenciador é chamado.
- O RG do cliente é solicitado.
- O método da classe BaseDados é chamado, procurando o cliente e removendo-o caso exista.

A classe BaseDados possui, portanto, listas de clientes, atividades e instrutores, métodos para incluir/alterar/desativar/removê-los, verificar se existem dado seu identificador único (ID de turma ou RG), relacioná-los/desrelacioná-los e verificar atividades com mais/menos clientes e com maior/menor preço. Esses métodos são chamados pela classe Gerenciador, que realiza a interação com o usuário para conseguir os dados necessários.

As respectivas validações também foram implementadas: por exemplo, não é possível incluir clientes de RG repetido e o sistema exibe uma mensagem avisando sobre isso. Tentativas de

desativar clientes/instrutores inexistentes também são tratadas, assim como exclusão de turmas e atividades, entre outros casos.

Nessa etapa, as principais funcionalidades estavam prontas com suas respectivas validações, como desejado para essa *release*.

Atualizamos o diagrama UML para que refletisse essas mudanças, acrescentando algumas correções mencionadas pelo professor (cardinalidades, associações, agregações etc). Não apresentamos esse diagrama neste relatório por conta de problemas com legibilidade do texto na imagem, mas submetemos o diagrama no Google Classroom.

Também fizemos alguns protótipos de interface gráfica (telas) para o software, já que a interação com o usuário se dava apenas via console e gostaríamos de mudar isso.

Durante a apresentação do código dessa *release*, o professor sugeriu que implementássemos métodos de busca, por exemplo, buscar clientes por nome/endereço/telefone e aumentássemos a complexidade do sistema acrescentando mais conceitos de Programação Orientada a Objetos, como herança e polimorfismo. Isso foi feito nas *releases* seguintes.

5 Etapa 4: *Release* 2

Seguindo as orientações do professor, implementamos métodos de busca e exploramos mais conceitos de POO no projeto. Também fizemos mais protótipos de interface gráfica.

Além disso, removemos a interação via console com o usuário, fazendo com que os métodos da classe Gerenciador recebessem por parâmetro os dados necessários. Fizemos isso pensando na ligação com a interface gráfica (da qual os parâmetros seriam recuperados futuramente).

5.1 Heranças, classes abstratas e polimorfismo

Percebemos que não haveria instanciação de objetos da classe Pessoa no sistema, apenas de Clientes e Instrutores. Portanto, a classe Pessoa se tornou abstrata.

Decidimos especializar mais os instrutores, tendo três tipos de instrutores: instrutores que recebem por dia trabalhado, instrutores que recebem por hora trabalhada e instrutores que são *personal trainers* de clientes. Com essa mudança, clientes passaram a ter o atributo Personal e a classe Instrutor se tornou, também, abstrata, porque todos os instrutores do sistema seriam de um desses tipos.

Decidimos que, além dos clientes comuns, teríamos clientes VIP que, além de frequentar turmas, teriam aulas a mais por semana. Mas o sistema possui instâncias tanto de clientes como de clientes VIP, então a classe Cliente continuou concreta.

5.2 Métodos de busca

Implementamos, na classe Gerenciador, métodos de busca de clientes por cada um de seus atributos. O mesmo para instrutores e atividades. Então é possível buscar clientes/instrutores por RG, nome, endereço, telefone, data de nascimento e/ou indicação de estar ativo no sistema. No caso dos instrutores, por área, lista de áreas e por salário também. Esses métodos retornam uma *ArrayList* de objetos da classe Cliente e uma *ArrayList* de objetos da classe Instrutor.

As atividades podem ser buscadas por nome, preço, lista de turmas e uma só turma. Os métodos retornam *ArrayList* de Atividade.

Essas buscas podem ser realizadas tanto por gerenciadores comuns como por administradores.

5.3 Interface gráfica

Nesta *release*, começamos a implementar a interface gráfica. Optamos pelo pacote Swing, que já vinha como padrão pela IDE do IntelliJ. Para começar, fizemos algumas telas de cadastro e adicionamos os campos. Houve uma pequena integração com o código já existente, porém a maioria das outras integrações foi realizada na *release* subsequente.

Começamos a modelar telas considerando as funções que haviam sido criadas e retornavam objetos, por exemplo, listas. Tivemos dificuldade com as telas pois não sabíamos qual pacote usar, a princípio. Depois de pesquisar um pouco, optamos pelo pacote Swing. Durante o desenvolvimento das telas, um integrante do grupo estava usando uma IDE diferente dos demais que tinha problemas para abrir os arquivos .form. Não conseguimos achar uma extensão correta, então optamos por dividir as tarefas de outra forma: este colega alteraria apenas as classes (.java), enquanto os outros integrantes alteravam os arquivos .form.

Apesar desse imprevisto, o atraso no desenvolvimento foi pequeno. Criar as telas e integrá-las com o código feito anteriormente foi um tanto trabalhoso. Tivemos erros em tipos de retorno e exibição de dados nas telas criadas, mas conseguimos resolver esses problemas. As telas criadas não abrangeram 100% do código criado, mas serviram de boa demonstração extra console do que o nosso projeto poderia fazer.

6 *Release* final

Com o intuito de utilizar o conhecimento sobre interfaces adquirido em sala de aula decidimos reimplementar a funcionalidade de *login*.

Implementamos uma interface (Logavel) que possui um método para realizar *login*, um para recuperar *username* e um para recuperar suas permissões de acesso ao sistema. Desse modo, todas as classes que podem logar no sistema devem implementar essa interface.

Em seguida implementamos as classes Admin e Comum, que representam *login*s de usuários comuns e de usuários administradores no sistema. A classe Gerenciador possui um atributo cujo tipo é Logavel e pode, portanto, ser um Admin ou um Comum (polimorfismo).

Como não implementamos persistência de dados, há um usuário padrão com *login* e senha conhecidos para que possamos logar e testar o sistema.

Nessa *release*, pudemos implementar todas as telas que seriam utilizadas no sistema e fazer boa parte das integrações com o código, como cadastros e consultas de todas as entidades mapeadas. Além disso, pudemos implementar telas de *login*. Não foi possível implementar um esquema de permissões na interface gráfica (como desabilitar botões e telas), mas a validação referente às permissões é efetuada nas classes e operações para as quais não há permissão não são realizadas.