

Introdução

O aplicativo desenvolvido neste trabalho, conforme definido nas etapas anteriores, tem como objetivo permitir que o usuário aluno da UFRGS acesse de maneira centralizada e facilitada informações cotidianas como grade horária, próximas tarefas e entregas, além do cardápio e tickets do RU. O projeto foi dividido em quatro grandes partes, as quais foram implementadas de forma paralela e independente. A primeira parte está relacionada às telas do aplicativo e à interface de interação do usuário. A segunda e terceira parte estão relacionadas com as disciplinas e tarefas, respectivamente. Por fim, a quarta parte está relacionada com cardápio e tickets do RU.

Mudanças em relação à Etapa anterior

Foram implementadas todas as classes definidas no diagrama entregue na etapa anterior, com exceção da classe ValidadorHorario. Além disso, ao longo da implementação, o grupo identificou a necessidade do desenvolvimento de outras classes. A adição dessas novas classes teve como objetivo obter um melhor funcionamento do aplicativo e adequar-se às boas práticas vistas em aula, evitando a presença de *code smells*. O diagrama de classes atualizado está disponível a seguir, assim como o detalhamento dessas mudanças.

Na superclasse Task (tarefa), as alterações foram a retirada de disciplina dos atributos, visto que a classe tarefa está dentro de uma disciplina e não precisaria ter essa informação dentro de tarefa, além disso, foi criado método getStatus e foi removido o método visualizarTarefa, pois isso se dá diretamente na interface visual. Na subclasse Work foi ajustado o único atributo para todo, visto que é importante ao ter um

trabalho extenso, poder colocar o que precisa ser feito para concluí-lo. Implementamos todo o conteúdo previsto no diagrama de classes em código, ajustando os nomes para inglês.

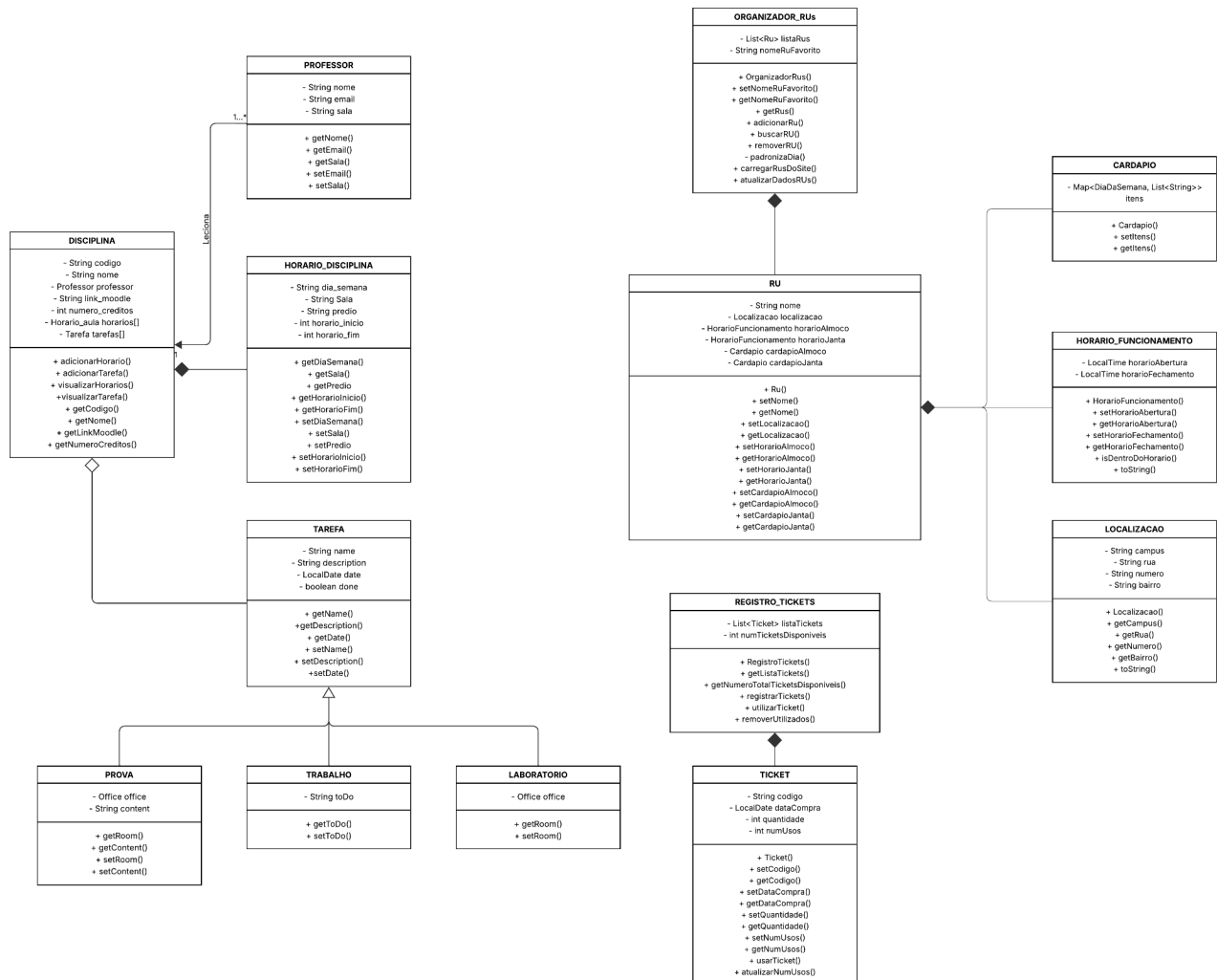


Diagrama de classes atualizado

No package de Courses (Disciplinas), foram inseridos as classes de Course (Disciplina), Teacher (Professor), Office (Sala) e Schedule (Horario_Disciplina). Dessa maneira foi criado uma classe extra para ajustar a sala de um professor e tornar o código mais legível e evitar o code smell de obsessão primitiva. Além disso foram feitas as seguintes alterações em relação ao diagrama da etapa anterior:

- Adicionado getters e setters para todos atributos da Classe Courses (Disciplina)
- Métodos de visualizar horários e tarefas podem ser resumidos ao seus getters, a visualização gráfica cabe a interface gráfica.

- O atributo Teacher (Professor) se tornou uma lista e consequentemente foi criado o método add Teacher.
- Adicionado polimorfismo aos construtores da classe Course, podendo criar um objeto com professores e horários instanciados ou não.
- Criada uma classe para a sala do professor.
- Foram escolhidas estruturas e bibliotecas mais apropriadas para alguns atributos como java.time.LocalDateTime

O diagrama anterior previa um total de quatro classes relacionadas a cardápio e tickets do RU: Ru, Ticket, OrganizadorRu, RegistroTicket. No entanto, visando ajustar-se às boas práticas vistas em aula, foram desenvolvidas três novas classes: Cardapio, Localizacao e HorarioFuncionamento.

No diagrama anterior, o cardápio (tanto do almoço como da janta) era representado por uma lista de itens indexados com o dia da semana na classe Ru, o que não contribuía para a legibilidade do código e encapsulamento. Por isso foi criada uma classe Cardapio. As classes Localizacao e HorarioFuncionamento tem como principal objetivo eliminar o uso de muitos dados primitivos para representar separadamente componentes que deveriam ser representados como uma unidade e também para padronização.

Por fim, foram adicionados getters e setters que não estavam previstos no diagrama para reforçar encapsulamento e métodos necessários para funcionamento adequado das funcionalidades previstas na definição do aplicativo. Por exemplo, ao extrair informações do cardápio do site oficial da universidade, os dias da semana estavam no formato “SEGUNDA-FEIRA” e a classe cardápio indexava os itens pelos dias da semana no formato “SEGUNDA”. Para resolver essa incompatibilidade foi criado um método auxiliar na classe OrganizadorRu, responsável pela atualização dos cardápios. Esse método auxiliar padronizava os dias da semana para indexação, removendo o sufixo “-FEIRA” e também substituindo o “ç” por “c”.

Implementação

Na etapa de implementação, o foco do grupo foi transpor as classes modeladas na etapa anterior para o código, priorizando a representação correta dos componentes da vida acadêmica dentro do aplicativo. Essa seção detalha qual componente da vida acadêmica cada classe modela, seus atributos e métodos e decisões de design relevantes.

A classe Task é a superclasse de Exam, Lab e Work, usando a ideia de herança, criamos atributos e métodos que são comuns a todas as tasks. Além disso, Task se relaciona com Course através de uma agregação. Os atributos comuns são name, description, date e status. Status informa se foi concluído ou não. Além do construtor com todos os atributos, temos os métodos getters e setters de todos os atributos.

A classe Work é subclasse de Task, como atributo exclusivamente dela temos toDo, que vai uma explicação do que precisa ser feito para o trabalho e, junto disso, terá o getter e setter desse atributo, o construtor usando o “super” para os atributos da classe-mãe. A classe Lab é também subclasse de Task, como atributo criado nela temos room e nos métodos o getter e setter de room, além do construtor. Na subclasse Exam temos dois atributos room e content, e seus getters e setters e o construtor.

A classe Course foi implementada para reunir as principais informações de uma disciplina, como código, nome, créditos, professores, horários, tarefas e o link do Moodle. Os atributos são tipados de acordo com sua função, utilizando listas para professores, horários e tarefas, e tipos simples para os demais dados. O construtor principal permite criar cursos já com listas preenchidas, enquanto construtores alternativos facilitam a criação de objetos vazios ou parciais. O encapsulamento é garantido por atributos privados e acesso via getters e setters, que também validam os dados. Os métodos de adição (addTeacher, addSchedule, addTask) asseguram que apenas objetos válidos sejam inseridos nas listas, lançando exceções para valores nulos, e o método setLinkMoodle permite atualizar o link do Moodle com validação.

A classe Teacher representa um professor, com atributos para nome, e-mail e um objeto Office (sala). Todos os atributos são privados e acessados por getters e setters, que também validam os dados. A relação entre Course e Teacher é de associação simples: um curso pode ter vários professores, e cada professor pode estar associado a um curso e também a um escritório (Office). A classe Office encapsula informações sobre sala e prédio, com métodos para acessar e modificar esses valores, além de um método para retornar uma descrição textual do escritório. Assim, a relação entre Teacher e Office é de composição simples também, pois o escritório faz parte da identidade do professor.

A classe Schedule representa o horário de uma atividade, contendo os atributos privados: dia da semana (weekday), sala (office), horário de início (begin_time) e horário de término (end_time). Todos os atributos possuem métodos getters e setters, que validam os

dados ao serem modificados. O método `getScheduleDetails()` retorna uma descrição textual completa do horário, incluindo o dia, o intervalo de tempo e os detalhes da sala. Assim foi adicionado uma relação entre `Schedule` e `Office` de associação simples, pois cada horário está vinculado a um objeto `Office` e uma relação de composição com `Course`, pois existe somente associado a uma instância de `Course`, horários só existem para cadeiras cursadas.

A classe `Ru` representa os restaurantes universitários da universidade. Cada restaurante tem uma localização associada. A classe `Localizacao` contém atributos como campus, rua, número e bairro. A escolha desses atributos para representar a localização do `Ru` foram baseadas no site oficial da universidade. A classe contém um método que retorna a localização completa no tipo *string* de forma padronizada. Cada instância da classe `Ru` também tem associados horários de almoço e de jantar, assim como cardápios de almoço e de jantar. Assim como no caso de `Localizacao`, foram implementadas classes específicas para representar horários e cardápios. A classe `HorarioFuncionamento`, além dos getters e setters de atributos, possui um método que verifica se determinado horário está dentro do período de funcionamento. A classe `Cardapio` tem como principal objetivo garantir que os itens não sejam afetados por referências externas, para isso os getters e setters realizam cópias seguras da lista.

A classe `OrganizadorRus` tem como principal objetivo administrar a lista de `Rus` existentes. Para isso, possui métodos de adição e remoção de `Rus` da lista. Além disso, é papel dessa classe atualizar os cardápios, por meio do método `carregarRusDoSite`, que utiliza a biblioteca `JSoup` para acessar o url da página oficial da universidade na qual se encontram os cardápios (<https://www.ufrgs.br/prae/cardapio-ru/>) e extrair as informações. Os itens do cardápio são inseridos em uma lista indexada pelo dia da semana.

As classes `Ticket` e `RegistroTickets` representam a parte do aplicativo focada em gerenciar o uso de tickets no restaurante da universidade. Os métodos principais da classe `Ticket` permitem definir e obter os valores de código e quantidade e também verificar se o ticket pode ser usado e incrementar a contagem de usos. A classe `RegistroTickets` atua como um gerenciador central, mantendo uma lista de tickets e rastreando o número de tickets total. Dessa forma há uma separação clara de responsabilidades: `Ticket` lida com o estado e comportamento de um único ticket (código), enquanto `RegistroTickets` gerencia a coleção e as operações em um nível superior.

Teste

Para a superclasse Task, casos de teste: Construtor e getters é testado os valores atribuídos ao criar o objeto pelo construtor e o de Setters, onde busca modificar os atributos após a criação.

Para a subclasse Work, casos de teste: Construtor é testado a herança dos atributos e o `todo` e o de getters e setters para ver se os valores estão sendo atualizados corretamente. Os testes da subclasse Exam são semelhantes ao anterior, primeiro faz-se um teste para verificar a herança e os novos atributos da subclasse e depois testa se getters e setters modificam valores corretamente e o mesmo ocorre nos testes da subclasse Lab.

Para a classe Course, os testes garantem que a inicialização dos objetos e o funcionamento dos métodos de acesso e modificação estejam corretos. São verificados os valores dos atributos após a criação do objeto, bem como a atualização das listas internas e demais campos via setters, assegurando integridade e encapsulamento.

Para a classe Teacher, os testes verificam se os atributos nome, e-mail e escritório são corretamente atribuídos e recuperados, além de testar a modificação desses dados. Isso garante que a associação com o objeto Office funciona e que os dados do professor podem ser alterados com segurança.

Para a classe Office, os testes cobrem a criação do objeto, a alteração dos atributos e a geração da descrição textual do escritório. Assim, assegura-se que os métodos de acesso e modificação funcionam corretamente e que a representação textual reflete os dados do objeto.

Os testes da classe Schedule verificam se os atributos são corretamente atribuídos e recuperados, testando os métodos getters e setters para todos os campos. Também é testado o método `getScheduleDetails()`, garantindo que a descrição textual do horário seja gerada corretamente. Os testes asseguram que a associação com o objeto Office funciona e que os horários podem ser modificados e consultados de forma segura.

Os testes unitários da classe Cardapio visaram assegurar o encapsulamento da classe. O primeiro teste, `testGetItensRetornaCopia`, verifica se o método `getItens()` retorna uma cópia defensiva dos dados internos, esperando que modificações externas na cópia retornada não afetem o estado original do Cardapio. Já o segundo teste,

`testConstrutorCriaCopiaDefensiva`, assegura que o construtor do `Cardápio` também cria uma cópia defensiva dos dados fornecidos, garantindo que alterações no mapa original não modifiquem o estado interno do objeto.

Os testes para as classes `HorarioFuncionamento` e `Localizacao` validam a integridade e funcionalidade de seus métodos e a apresentação dos dados. Para `HorarioFuncionamento`, o `testGettersAndSetters` verifica a correta leitura e modificação dos horários de abertura e fechamento. O `testIsDentroDoHorario` assegura que o método `isDentroDoHorario` avalia precisamente se um horário está dentro do intervalo definido, considerando as bordas. Além disso, o `testToStringFormat` confirma a formatação adequada da representação textual do horário. Já para a classe `Localizacao`, o `testGetters` valida que os métodos de acesso retornam corretamente os dados do campus, rua, número e bairro. Por fim, o `testToStringFormat` verifica se o método `toString` da `Localizacao` retorna uma string formatada corretamente.

Os testes da classe `OrganizadorRus` verificam a funcionalidade de gerenciamento de `Rus`. O `testAdicionarRuOrdenado` garante que o método `insere` `RUs` na ordem alfabética de seus nomes. O `testBuscarRUExistente` assegura que `buscarRU` encontra corretamente um `RU` presente na lista. Em contraste, o `testBuscarRUNaoExistente` valida que `buscarRU` lança uma `RuNaoEncontradoException` quando o `RU` buscado não existe na lista. Para remoção, o `testRemoverRUExistente` confirma que `removerRU` remove um `RU` existente e retorna `true`, enquanto o `testRemoverRUNaoExistente` verifica que o método retorna `false` ao tentar remover um `RU` inexistente, sem lançar exceções. Os testes da classe `Ru` abrangem a verificação de seus métodos de acesso e modificação para os atributos principais, se modificam e retornam corretamente os dados do objeto.

Os testes da classe `RegistroTickets` garantem o funcionamento adequado do gerenciamento de tickets. O `testRegistrarTickets` verifica se o método `registrarTickets` adiciona corretamente novos tickets à lista interna e atualiza a contagem total de tickets disponíveis. O `testUtilizarTicketComSucesso` valida que o método `utilizarTicket` permite o uso de tickets dentro do limite de quantidade, decrementando os usos e a disponibilidade, e que o uso é negado após o limite ser atingido. Por fim, o `testUtilizarTicketCodigoIncorreto` assegura que a implementação lida corretamente com tentativas de uso de tickets com códigos inexistentes, retornando `false` e mantendo o número de tickets utilizado sem alterações.

Por fim, os testes da classe Ticket garantem que os métodos get e set para todos atributos funcionam corretamente, incluindo a verificação do valor inicial de numUsos ser zero e que o método atualizarNumUsos() incrementa a contagem de usos de forma adequada. Além disso, os testes validam o método usarTicket() em diferentes cenários, confirmando que ele retorna true quando há tickets disponíveis para uso e false quando eles terminam.

A utilização de testes unitários durante a implementação foi fundamental para garantir a qualidade e a robustez do código. Os testes permitiram identificar rapidamente erros de lógica, problemas de encapsulamento e inconsistências nos métodos de acesso e modificação dos objetos. Além disso, a escrita dos testes antes ou durante a implementação ajudou a definir claramente o comportamento esperado das classes, servindo como documentação viva do sistema.

O grupo percebeu que, ao adotar testes unitários, o processo de refatoração e evolução do código tornou-se mais seguro e eficiente, pois qualquer alteração que introduzisse um erro era imediatamente detectada pelos testes. Isso proporcionou maior confiança no funcionamento correto das classes do domínio e facilitou a integração entre diferentes partes do sistema. Em resumo, a experiência com testes unitários foi extremamente positiva e reforçou a importância dessa prática no desenvolvimento de software orientado a objetos.

Executável (aplicação, interface)

Execução da Aplicação no Android Studio

A aplicação desenvolvida pela equipe pode ser executada em dispositivos físicos com sistema Android ou por meio de um ambiente de simulação, utilizando o emulador integrado ao Android Studio. Considerando a finalidade deste relatório, o procedimento de execução será descrito com foco no uso do Android Studio e seu respectivo simulador, permitindo assim a análise conjunta da interface da aplicação e de seu código-fonte.

Para iniciar o processo de execução, é necessário que o usuário interessado tenha o Android Studio devidamente instalado em sua máquina de desenvolvimento. O guia oficial de instalação da ferramenta encontra-se disponível no endereço:

<https://developer.android.com/studio/install>. O Android Studio é compatível com os principais sistemas operacionais, incluindo Windows, Linux e macOS.

Além disso, o usuário precisa obter uma cópia local do repositório do projeto. Caso possua o Git instalado, o processo pode ser realizado por meio do terminal, utilizando o seguinte comando:

```
git clone git@github.com:SW-Engineering-Courses-Karina-Kohl/tcp-final-20251-grupo01.git
```

Essa etapa garante a clonagem completa do código-fonte e dos arquivos de configuração necessários para a correta execução da aplicação.

Após a instalação do Android Studio e a clonagem do repositório, o próximo passo consiste na abertura do projeto dentro da ferramenta. Para isso, deve-se seguir o procedimento abaixo:

1. Abrir o Android Studio.
2. Na tela inicial, selecione a opção “Open”.
3. Navegar até o local onde o repositório foi clonado.
4. Acessar a subpasta *src/androidApplication*, que contém o código-fonte da aplicação.
5. Confirmar a abertura do projeto.

Ao realizar esse processo, o Android Studio iniciará automaticamente a sincronização do projeto com o Gradle, seu sistema de automação de build. É importante aguardar a conclusão dessa etapa, uma vez que o ambiente pode realizar o download de bibliotecas e dependências adicionais necessárias para a correta compilação da aplicação.

Com o projeto devidamente carregado e sincronizado, o ambiente estará pronto para a execução da aplicação. O procedimento padrão é o seguinte:

1. Selecionar um dispositivo virtual na barra de dispositivos, localizada na parte superior da interface do Android Studio. Caso nenhum emulador esteja configurado, o Android Studio solicitará a criação de um novo dispositivo virtual (AVD - Android Virtual Device).
2. Após a seleção do emulador, clicar no botão “Run”, identificado por um ícone de triângulo verde, também localizado na parte superior da interface. Alternativamente, é possível utilizar os seguintes atalhos de teclado, para Windows/Linux use Shift + F10 e para macOS use Control + R

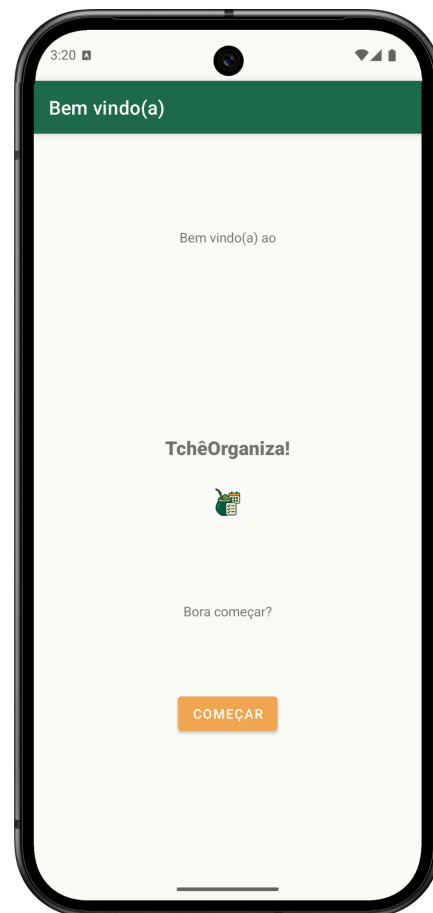
O Android Studio iniciará o processo de compilação e, em seguida, abrirá o emulador selecionado. O processo de inicialização do emulador pode levar alguns minutos

na primeira execução. Assim que o dispositivo virtual estiver pronto, a aplicação TchêOrganiza será instalada e executada automaticamente.

Uma vez em execução, o usuário poderá navegar pelas diferentes funcionalidades implementadas, realizando a interação direta com o aplicativo como se estivesse utilizando um dispositivo físico. Essa abordagem permite não apenas validar o funcionamento da aplicação, mas também facilitar a depuração de eventuais problemas no código-fonte, tendo em vista que o Android Studio oferece ferramentas integradas de monitoramento de logs e pontos de interrupção (breakpoints).

Guia de uso do TchêOrganiza

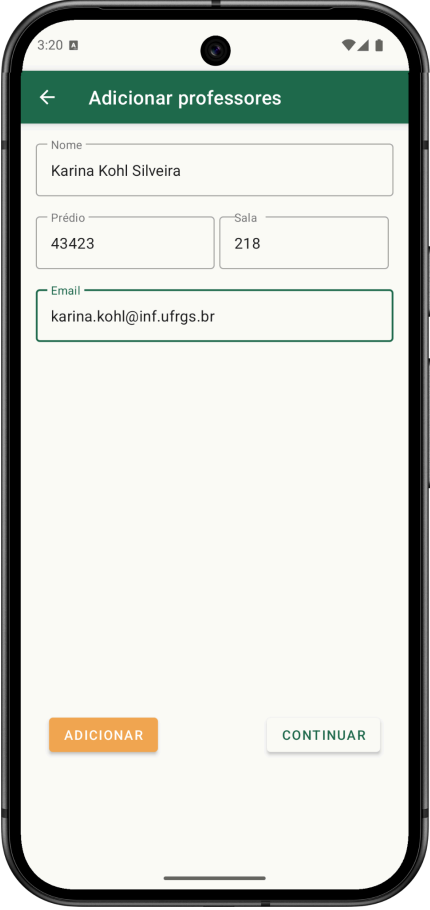
Ao iniciar o aplicativo, o usuário é apresentado a uma tela de boas-vindas, cujo objetivo é introduzir a proposta do TchêOrganiza e orientar o fluxo inicial de configuração. Nesta tela, é exibido o nome e logotipo do aplicativo, acompanhada de um botão de ação com o rótulo “Começar”.



Ao clicar no botão “Começar”, o usuário é direcionado para a primeira etapa de configuração: o cadastro de professores. Esta tela reflete diretamente a estrutura da classe Teacher (Professor), conforme definida no diagrama de classes. Os campos apresentados ao usuário são:

- Nome do professor
- Prédio
- Sala

Para realizar o cadastro, o preenchimento de todos os campos é obrigatório. Após inserir os dados, o usuário deve clicar no botão “Adicionar”, o que resultará na persistência do professor localmente no dispositivo. Caso necessário, o usuário poderá adicionar múltiplos professores, repetindo o processo.



The image shows a smartphone screen with a green header bar containing a back arrow and the text "Adicionar professores". The form has three input fields: "Nome" with the value "Karina Kohl Silveira", "Prédio" with the value "43423", and "Sala" with the value "218". Below these is an "Email" field with the value "karina.kohl@inf.ufrgs.br". At the bottom, there are two buttons: "ADICIONAR" in orange and "CONTINUAR" in green.

Nome	Prédio	Sala	Email
Karina Kohl Silveira	43423	218	karina.kohl@inf.ufrgs.br

Finalizado o cadastro de professores, o aplicativo direciona o usuário para a tela de cadastro de disciplinas. Nesta etapa, o usuário deve inserir as seguintes informações:

- Nome da disciplina
- Professor responsável (selecionado a partir da lista de professores cadastrados na etapa anterior)
- Horários da disciplina
- Data final da disciplina (normalmente correspondente ao término do semestre)

A adição dos horários é realizada por meio de um botão de ação identificado por um ícone “+” de cor verde. Ao clicar, o usuário pode inserir um novo horário com os seguintes campos:

- Dia da semana
- Horário de início
- Horário de término
- Prédio
- Sala

Se desejar remover um horário previamente inserido, o usuário pode utilizar o ícone de lixeira exibido ao lado de cada horário. Essa abordagem oferece flexibilidade ao permitir que o usuário defina com precisão a estrutura da disciplina conforme sua realidade acadêmica.

The image shows a smartphone screen with the 'Adicionar disciplinas' (Add disciplines) form. The form is titled 'Adicionar disciplinas' in a green header bar. Below the header, there are two input fields: 'Nome da disciplina' (Discipline name) with the value 'Técnicas de Construção de Programas' and 'Professor' (Professor) with the value 'Karina Kohl Silveira'. Below these fields is a section titled 'Horários da disciplina' (Discipline hours) with a green '+' icon to add more. This section contains two entries. The first entry is for 'Terça-feira' (Wednesday) and includes fields for 'Horário de início' (Start time) with '10:30', 'Horário de término' (End time) with '12:10', 'Prédio' (Building) with '43425', and 'Sala' (Room) with '112'. A green trash icon is next to the day name. The second entry is for 'Quinta-feira' (Thursday) and also has a green trash icon. At the bottom of the form are two buttons: 'ADICIONAR' (Add) in orange and 'CONTINUAR' (Continue) in white.

Após a definição das disciplinas, o usuário avança para a tela de cadastro de atividades (Tasks). Nessa etapa, é possível adicionar quantas atividades desejar, preenchendo os seguintes campos:

- Nome da atividade
- Data de realização ou entrega
- Tipo de atividade (Exame, Laboratório ou Trabalho), refletindo a estrutura de herança da classe Task
- Disciplina associada (selecionada a partir das disciplinas cadastradas anteriormente)
- Prédio (se aplicável)
- Sala (se aplicável)

Os campos exibidos no formulário de cadastro de atividade são dinâmicos e variam conforme o tipo de atividade selecionado, respeitando os atributos específicos de cada subclasse (Exam, Lab ou Work), conforme definido no modelo de domínio.

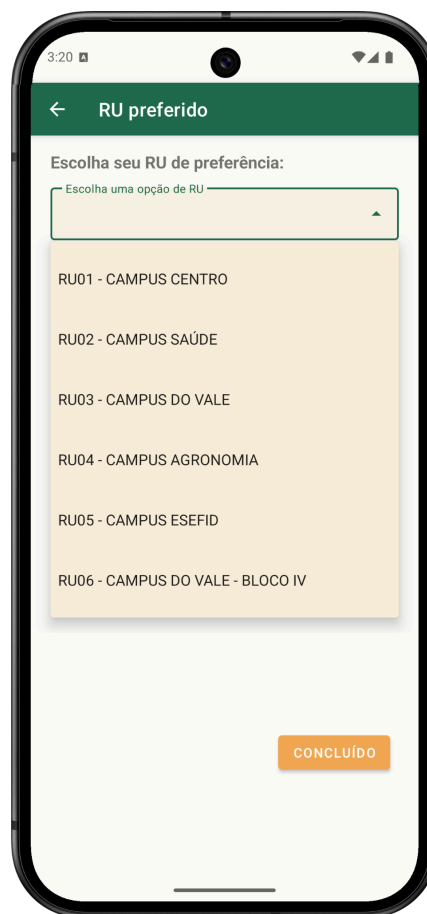


The image shows a smartphone screen with a green header bar containing a back arrow and the text "Adicionar atividades". The form below has the following fields:

- Nome da atividade:** A text input field containing "Apresentacao do Trabalho - Etapa 2".
- Data:** A date input field containing "22/06/2025".
- Tipo:** A dropdown menu with "Trabalho" selected.
- Disciplina:** A dropdown menu with "Tecnicas de Construcão de Programas" selected.
- Prédio:** A text input field containing "43424".
- Sala:** A text input field containing "100".
- Etapa 2:** A text area containing "Implementação e Teste".

At the bottom of the screen are two buttons: "ADICIONAR" (orange) and "CONTINUAR" (white with green border).

Na última etapa da configuração inicial, o usuário deve selecionar o Restaurante Universitário (RU) de sua preferência. Esta escolha será utilizada posteriormente para exibir o cardápio do RU selecionado na tela principal do aplicativo.



Interface Principal do Aplicativo

Após a configuração inicial, o aplicativo apresenta ao usuário sua interface principal, estruturada em quatro abas, acessíveis via uma barra de navegação inferior. As abas são:

Hoje

Nesta aba, o usuário visualiza de forma consolidada as seguintes informações relevantes para o dia atual:

- Disciplinas do dia, com horários e salas correspondentes
- Próximas atividades cadastradas, ordenadas por data
- Cardápio (almoço e jantar) do RU selecionado na configuração inicial



RU

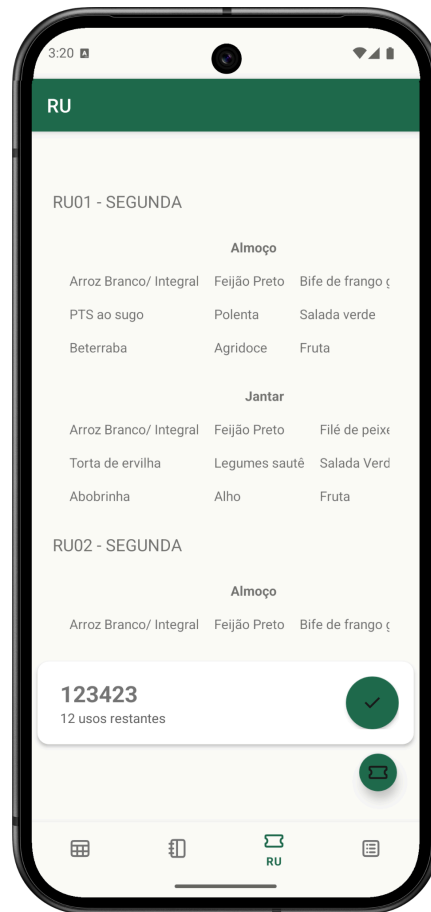
Apresenta o cardápio de todos os RUs da UFRGS para o dia atual, incluindo as refeições de almoço e jantar. Além disso, a aba permite a gestão de tickets, com as seguintes funcionalidades:

- Visualizar a lista de tickets existentes
- Adicionar novos tickets

Ao clicar para adicionar um ticket, o usuário deverá informar:

- Número do ticket
- Quantidade adquirida
- Data da compra

Uma vez adicionado, o novo ticket será exibido automaticamente na aba RU.



Todas as telas de adição de professores, disciplinas e atividades utilizam os mesmos *fragments* implementados para a etapa de apresentação inicial, garantindo consistência de experiência e interface ao longo de toda a aplicação.