Universidade Federal Fluminense
Instituto de Ciência e Tecnologia
Bacharelado em Ciência da Computação

Rafael Andrade da Rocha
Rômulo Eduardo Garcia Moraes

MyUni : Um Framework para o Desenvolvimento de Aplicativos para a Gerência de Informações Acadêmicas

Rio das Ostras - RJ
2017

RAFAEL ANDRADE DA ROCHA
RÔMULO EDUARDO GARCIA MORAES

MYUNI : UM FRAMEWORK PARA O DESENVOLVIMENTO DE APLICATIVOS PARA A GERÊNCIA DE INFORMAÇÕES ACADÊMICAS

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel. Área de Concentração: Engenharia de Software.

Orientador: Prof. DSc. SÉRGIO CRESPO COELHO DA SILVA PINTO

Rio das Ostras - RJ
2017

RAFAEL ANDRADE DA ROCHA
RÔMULO EDUARDO GARCIA MORAES

MYUNI : UM FRAMEWORK PARA O DESENVOLVIMENTO DE APLICATIVOS PARA A GERÊNCIA DE INFORMAÇÕES ACADÊMICAS

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel. Área de Concentração: Engenharia de Software.

Aprovada em Dezembro de 2017.

BANCA EXAMINADORA

PROF. DSC. SERGIO CRESPO COELHO DA SILVA PINTO - ORIENTADOR
UNIVERSIDADE FEDERAL FLUMINENSE

PROF. DSC. ADRIANA PEREIRA DE MEDEIROS
UNIVERSIDADE FEDERAL FLUMINENSE

PROF. DSC. CARLOS BAZILIO MARTINS
UNIVERSIDADE FEDERAL FLUMINENSE

Rio das Ostras - RJ
2017

Aos nossos pais, irmãos, e avós, que com muito carinho e amor, não mediram esforços para que pudessemos chegar até esta etapa das nossas vidas.

Agradecimentos

Aos nossos professores, e em especial, ao nosso orientador, Prof. DSc. Sérgio Crespo, que sempre nos apoio, motivou, e forneceu grande parte do conhecimento necessário para a realização deste trabalho.

Aos nossos amigos de curso, Fagner Alves, Felipe Lugão e Rodrigo Rodovalho, que sempre estiveram ao nosso lado, e com os quais pudemos adquirir e compartilhar conhecimentos diversos além dos adquiridos na universidade, e que foram de grande importância para a realização desse trabalho.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Sumário

Resumo

Um dos usos comuns dos smartphones é a gerência de arquivos, eventos e compromissos do dia-a-dia, sejam eles pessoais ou profissionais. Nesse contexto, é de se esperar que os universitários utilizem os smartphones para acessar serviços e aplicativos com o objetivo de gerenciar e otimizar suas informações acadêmicas. Baseando-se na utilização de tais serviços e no crescente número de aplicativos para fins de gerência de informações acadêmicas, este trabalho tem como objetivo propor a criação de um framework baseado em componentes para auxiliar na criação de aplicativos para a gerência de informações acadêmicas. Esses componentes poderão então ser combinados ou adaptados para se criar uma aplicação com as funcionalidades e características que melhor atendam os universitários.

Palavras-chave: Frameworks. Componentes de Software. Gerência de Informações Acadêmicas. Aplicativos para Dispositivos Móveis.

Abstract

One of the common uses of smartphones is the management of daily, personal or professional files, activities and appointments. In this context, university students are expected to use smartphones to access services and applications to manage and optimize their academic information. Based on the use of such services and the increasing number of applications for academic information management purposes, this work aims to propose the creation of a component based framework to assist the creation of academic information management applications for smartphones. These components could then be combined or adapted to provide an application with the features and characteristics chosen by the developer.

Keywords: Frameworks. Software Components. Mobile Applications. Academic Information Management.

Capítulo 1

Introdução

1.1 Contextualização

A vida na universidade é por muitas vezes, agitada e caótica. Além dos conteúdos aprendidos dentro e fora de sala, os alunos ainda têm que lembrar e se manterem atualizados sobre datas de entregas de exercícios, apresentações de trabalhos e provas, localização de laboratórios, horário das aulas, data de entrega de livros etc. Com tantas tarefas e informações para lembrar, é muito comum que os alunos utilizem aplicativos de dispositivos móveis que forneçam funcionalidades de calendário, listas de tarefas, notas e alarmes, para facilitar o gerenciamento de tais informações.

No entanto, apesar de permitirem a gerência das informações acadêmicas, esses aplicativos muitas vezes não são integrados, tornando o gerenciamento das informações uma tarefa demorada e muitas vezes caótica, pois é necessário acessar e alternar dentre vários aplicativos até que seja possível encontrar a informação que se estava procurando. Além disso, esses aplicativos não são integrados com as informações acadêmicas do aluno (*e.g* plano de estudos, histórico, calendário de provas etc.), fazendo com que a inserção desses dados nos aplicativos seja uma tarefa muitas vezes tediosa.

Tendo em vista esses problemas, muitas universidades acabaram desenvolvendo aplicativos que fornecessem aos seus alunos um meio prático para a gerência de informações acadêmicas, como planos de estudos, situação de inscrição em disciplinas, histórico etc., e também de consulta de informações sobre a universidade, como notícias, calendário de eventos, cardápios dos restaurantes universitários, itinerários de transporte universitário, serviços de biblioteca etc.

Alguns exemplos de universidades que oferecem aplicativos com algumas dessas funcionalidades são a UFF (Universidade Federal Fluminense), UFRJ (Universidade Federal do Rio de Janeiro), UFPE (Universidade Federal de Pernambuco), PUC-SP (Pontifícia Universidade Católica de São Paulo), e UFRGS (Universidade Federal do Rio Grande do Sul), que oferecem a seus alunos, respectivamente, os seguintes aplicativos para dispositivos Android [1], UFF Mobile [2], Portal do Aluno UFRJ [3], UFPE Mobile [4], PUC-SP Mobile [5] e UFRGS Mobile [6]. Nas figuras 1.1, 1.2 e 1.3 são mostradas as telas principais dos aplicativos UFF Mobile, UFPE Mobile e PUC-SP Mobile, respectivamente.

Figura 1.1: Tela principal do aplicativo UFF Mobile.

Figura 1.2: Tela principal do aplicativo UFPE Mobile.

Figura 1.3: Tela principal do aplicativo PUC-SP Mobile.

1.2 Problema

Mesmo havendo um esforço mútuo no desenvolvimento de aplicativos que facilitem a gerência de informações acadêmicas, produzir tais aplicativos não é uma tarefa fácil e rápida. No entanto, existem estratégias que visam minimizar esses problemas. Uma delas, é a utilização de técnicas de reuso de software, onde o processo de desenvolvimento é voltado à reutilização de softwares existentes, ao invés da criação de software inteiros "a partir do zero". Esse tipo de técnica surgiu das demandas por menores custos de produção e manutenção, além da necessidade de entrega mais rápida e com maior qualidade [7].

Uma das técnicas de reuso de software é a utilização de frameworks. Segundo (Schmidt; Gokhale; Natarajan, 2004)[8], um framework é "um conjunto integrado de artefatos de software (como classes, objetos e componentes) que colaboram para fornecer uma arquitetura reutilizável para uma família de aplicações relacionadas". Os frameworks podem então ser entendidos como softwares pré-fabricados e que fornecem um conjunto pré-determinado de elementos e funcionalidades - que podem ser modificadas ou estendidas - para a criação de diferentes softwares que pertençam a um mesmo domínio. Portanto, a utilização de frameworks se mostra uma ótima estratégia para acelerar o processo de desenvolvimento de softwares.

1.3 Proposta

Os aplicativos para a gerência de informações acadêmicas, mesmo sendo desenvolvidos e oferecidos por diferentes universidades, em sua maioria, possuem estrutura e funcionalidades similares. Logo, é de se esperar que, caso um novo aplicativo para a gerência de informações acadêmicas venha a ser desenvolvido, ele também possua estrutura e funcionalidades similares as existentes nos outros aplicativos desse domínio.

Baseando-se nisso e no fato de que os frameworks fornecem uma arquitetura reutilizável e um conjunto de elementos para a criação de uma família de aplicações de um mesmo domínio, esse trabalho tem o objetivo de propor a criação de um pequeno framework baseado em componentes para auxiliar os alunos de universidades a desenvolverem aplicativos para a gerência de informações acadêmicas.

Por ser um framework baseado em componentes, os alunos poderão combinar e adaptar os componentes fornecidos para criar aplicativos com funcionalidades e características escolhidas por eles, afinal, não há ninguém melhor do que os próprios alunos para determinarem quais as funcionalidades são realmente úteis em tais aplicativos. Para validar a proposta, foi criada uma pequena prova de conceito, que para ser realizada, seguiu os mesmos processos necessários para a criação de frameworks comerciais.

Capítulo 2

Embasamento Teórico

2.1 Reuso de Software

O reuso de software, é uma estratégia de engenharia de software onde o processo de desenvolvimento é conduzido de maneira a maximizar a reutilização de softwares existentes (*e.g.* bibliotecas, componentes, aplicações completas) para criar novos softwares com um menor custo de produção e manutenção, e permitir a entrega de softwares de maneira mais rápida e com maior qualidade [7].

Segundo (Sommerville, 2010)[7], o reuso de software pode ser classificado, de maneira geral, de acordo com o tamanho da unidade de software que se está reutilizando para criar novos softwares:

1. **Reutilização de aplicação**: Toda a aplicação pode ser reutilizada, incorporando-a a outras aplicações sem a necessidade de realizar mudanças, ou configurando-a para diferentes clientes.

2. **Reutilização de componentes**: Os componentes de um aplicativo, que podem variar de tamanho desde subsistemas completos a objetos únicos, podem ser reutilizados. Por exemplo, um subsistema de identificação de padrões que foi desenvolvido como parte de um sistema de processamento de texto, pode ser reutilizado em um sistema de gerenciamento de banco de dados.

3. **Reutilização de objetos e funções**: Elementos de software que implementam funções específicas, ou um objeto de uma classe, podem ser reutilizados diretamente ou agrupados na forma de bibliotecas. Essa forma de reutilização baseada em bibliotecas é muito utilizada, pois permite que classes e funções que resolvem problemas específicos, e muitas vezes complexos, possam ser ligadas ao código do software em desenvolvimento, reduzindo a quantidade de erros, o tempo, e custo de desenvolvimento. Em áreas como algoritmos matemáticos e gráficos, onde conhecimentos especializados são necessários para desenvolver objetos e funções eficientes, esta é uma abordagem particularmente eficaz.

Ainda segundo Sommerville, uma outra forma de reutilização é a "reutilização de conceitos", onde, ao invés de se reutilizar elementos de software, é reutilizada uma ideia, um conceito, ou um método de trabalho. Tal forma de reutilização, não inclui detalhes de implementação, e portanto, pode ser utilizada e adaptada para uma variedade de situações, como por exemplo, na criação de padrões de projeto. Quando

os conceitos são reutilizados, o processo de reutilização inclui uma atividade onde os conceitos abstratos são instanciados para criar componentes de software concretos e reutilizáveis.

É importante citar que, para se desenvolver softwares que sejam reutilizáveis, é necessário que os processos de desenvolvimento do software sejam adaptados para levar em consideração a reutilização. Isso significa que as etapas de análise e levantamento de requisitos, projeto, e implementação do software, devem incluir atividades explícitas para procurar e avaliar componentes que possam ser reutilizados.

2.1.1 Técnicas de Reuso

Existem diversas técnicas de reuso, e a maioria delas exploram o fato de que aplicações de um mesmo domínio são semelhantes e têm potencial para reutilização, e que a reutilização é possível em diferentes níveis, desde funções simples até aplicações completas. A tabela 2.1 mostra algumas das principais técnicas de reuso de software.

Técnica	Descrição
Padrões de arquitetura	Arquiteturas de software que suportam tipos comuns de aplicações, são utilizadas como base no desenvolvimento de outros aplicativos.
Padrões de projeto	Abstrações genéricas que ocorrem com frequência em diferentes aplicações, são representadas como padrões de objetos e interações concretas e abstratas.
Desenvolvimento baseado em componentes	Aplicações são desenvolvidas integrando diferentes componentes (coleções de objetos), que devem seguir um padrão de comunicação e comportamento.
Frameworks	Conjuntos de classes abstratas e concretas que podem ser adaptadas e estendidas para criar uma família de aplicações de um determinado domínio.
Bibliotecas	Classes e funções que implementam abstrações e resolvem um problema específico de um determinado domínio.

Tabela 2.1: Técnicas de reuso de software.[7]

2.1.2 Vantagens e Desvantagens

Como visto, uma das vantagens da reutilização de software é a redução do esforço e recursos envolvidos no seu desenvolvimento. Outra vantagem é que, através da reutilização, menos componentes do software em desenvolvimento precisam ser projetados, especificados, implementados e validados. Além dessas, outras vantagens são mostradas na tabela 2.2.

Vantagem	Descrição
Maior confiabilidade	O software reutilizado, que foi utilizado e testado em diversos sistemas, deve ser mais confiável do que o novo software, uma vez que suas falhas de projeto e implementação deveriam ter sido encontradas e corrigidas em utilizações anteriores.
Risco de processo reduzido	O custo do software existente já é conhecido, enquanto os custos de desenvolvimento são sempre uma questão de julgamento. Este é um fator importante para o gerenciamento de projetos porque reduz a margem de erro na estimativa de custo do projeto. Isto é particularmente verdadeiro quando componentes de software relativamente grandes como subsistemas são reutilizados.
Uso efetivo por especialistas	Ao invés de fazer o mesmo trabalho todas as vezes, os especialistas em aplicações de um determinado domínio podem desenvolver softwares reutilizáveis que encapsulam seus conhecimentos.
Cumprimento de padrões	Alguns padrões, como padrões de interface do usuário, podem ser implementados como um conjunto de componentes reutilizáveis. Por exemplo, se os menus em uma interface de usuário forem implementados usando componentes reutilizáveis, todos os aplicativos apresentam os mesmos formatos de menu aos usuários.
Desenvolvimento rápido	Levar um sistema para o mercado o mais cedo possível é muitas vezes mais importante do que os custos globais de desenvolvimento. Um software que utiliza técnicas de reutilização, pode ter sua produção acelerada porque o tempo de desenvolvimento e validação pode ser reduzido.

Tabela 2.2: Vantagens do reuso de software.[7]

Além de vantagens, a técnica de reuso de software também possui algumas problemas associados, como mostra a tabela 2.3.

Problema	Descrição
Custos elevados de manutenção	Se o código-fonte de uma aplicação ou componente reutilizados não está disponível, os custos de manutenção podem ser maiores porque os elementos reutilizados podem tornar-se cada vez mais incompatíveis conforme são realizadas modificações no sistema que as utilizam.
Falta de ferramentas de suporte	Algumas ferramentas de software não apoiam o desenvolvimento com reutilização. Isto é particularmente verdadeiro para ferramentas que dão suporte à engenharia de sistemas embarcados.
Síndrome do "não inventado aqui"	Alguns engenheiros de software preferem reescrever componentes porque acreditam que podem melhorar. Isto tem a ver, em parte, com a confiança, e em parte com o fato de que escrever software original é visto como um desafio maior do que reutilizar o software de outras pessoas.
Criar, manter e utilizar uma biblioteca de componentes	Criar uma biblioteca de componentes reutilizáveis e garantir que os desenvolvedores de software possam usar essa biblioteca pode ser custoso. Os processos de desenvolvimento devem ser adaptados para garantir que as bibliotecas de componentes sejam utilizáveis.
Encontrar, entender e adaptar componentes reutilizáveis	Os componentes de software devem ser encontrados em uma biblioteca, seu funcionamento deve ser entendido e, às vezes, ela deve ser adaptada para trabalhar em um novo ambiente.

Tabela 2.3: Problemas associados ao reuso de software.[7]

É importante notar que, mesmo levando em conta as vantagens e desvantagens, a real complexidade da utilização de técnicas de reuso em um projeto de software vai depender dos requisitos da aplicação que se deseja desenvolver, da tecnologia, dos recursos disponíveis, da experiência da equipe de desenvolvimento e de qual técnica de reuso será utilizada. Portanto, é necessário analisar cada caso cuidadosamente.

2.2 Desenvolvimento de Softwares Baseados em Componentes

Como visto anteriormente, uma das técnicas de reuso de software, é a técnica de desenvolvimento baseada em componentes, também conhecida como CBSE (*Component Based Software Engeneering*). Nessa técnica, um conjunto de componentes de softwares é definido, implementado, combinado e integrado para criar uma aplicação.

Os componentes de software, segundo (Sommerville, 2010)[7], "são abstrações com um nível maior que os objetos, e são definidos pelas suas interfaces. Eles geralmente são maiores do que objetos individuais e todos os detalhes de implementação estão ocultos de outros componentes". Complementar a essa definição, (Weinreich; Sametinger, 2001)[9] dizem que componentes são semelhantes às classes, pois, assim como elas, componentes podem definir o comportamento e a criação de objetos. Ainda segundo eles, ao contrário das classes, a implementação de um componente é geralmente completamente

escondida, e internamente, essas implementações podem ser feitas por uma única classe ou por múltiplas classes. Já (Szyperski; Gruntz; Murer, 2003)[10] define que um componente de software é uma unidade de composição com interfaces especificadas e dependências explícitas, e pode ser utilizado de forma independente ou através de composição com outros componentes.

De maneira geral, os componentes possuem as seguintes propriedades [7][9][11]:

- **Reutilização**: Os componentes devem ser projetados para serem reutilizados em diferentes situações e em diferentes aplicações. No entanto, alguns componentes podem ser projetados para uma tarefa específica.
- **Padronização**: Os componentes devem seguir um modelo padrão de interfaces, metadados, documentação, composição e implantação de componentes.
- **Encapsulamento**: Um componente deve fornecer interfaces que permitem a execução das funcionalidades fornecidas por ele, mas não devem expor detalhes dos processos internos ou de qualquer variável ou estado interno.
- **Independência**: Os componentes devem ser projetados para ter dependências mínimas em outros componentes.
- **Extensibilidade**: Um componente deve poder ser estendido a partir de componentes existentes para fornecer novos comportamentos.
- **Substituição**: Os componentes devem poder ser substituídos livremente por outros componentes similares.
- **Documentação**: Os componentes devem ser documentados para que os usuários possam decidir se os componentes atendem às suas necessidades ou não. Além disso, é importante que todas as interfaces do componente sejam especificadas.

2.2.1 Componentes como Serviços

Uma outra maneira de se definir um componente, segundo (Sommerville, 2010)[7], é visualizá-lo como um provedor de serviços. Nesse sentido, quando um sistema precisa de um determinado serviço, ele chama um componente que fornece esse serviço sem se preocupar com seus detalhes e funcionamento internos. Alguns exemplos providos por Sommerville são, um componente em um sistema bibliotecário, que pode fornecer um serviço de pesquisa que permite aos usuários procurar diferentes catálogos de bibliotecas; e um componente que fornece serviços para conversão de imagens de um formato gráfico para outro. Ainda segundo ele, ao visualizarmos um componente como um provedor de serviços, podemos defini-lo com as seguintes características:

1. Um componente é uma entidade executável independente que é definida por suas interações. Você não precisa de nenhum conhecimento sobre seu código-fonte para usá-lo e ele pode ser referenciado como um serviço externo ou incluído diretamente em uma aplicação.

2. Os serviços oferecidos por um componente são disponibilizados através de uma interface e todas as interações ocorrem através dessa interface. A interface do componente é expressa em termos de operações parametrizadas e seu estado interno nunca é exposto.

Para funcionar como um serviço, os componentes devem possuir duas interfaces, que representam, respectivamente, os serviços que o componente fornece e os serviços que o componente necessita para funcionar corretamente, como mostra a figura 2.1:

- As interfaces 'fornecidas' definem os serviços fornecidos pelo componente. Elas definem os métodos que podem ser chamados por um usuário do componente. Em um diagrama de componentes UML, a interface 'fornece' para um componente é indicada por um círculo no final de uma linha do ícone do componente.

- As interfaces 'requeridas' especificam quais serviços devem ser fornecidos por outros componentes para que um determinado componente funcione corretamente. Se estes componentes não estiverem disponíveis, o componente que se deseja utilizar não funcionará, porém, isso não compromete a independência ou implantação de um componente. Na UML, o símbolo para uma interface requerida é um semicírculo no final de uma linha do ícone do componente.

Figura 2.1: Representação UML de um componente.

É importante dizer que, embora não seja mostrado na figura, os métodos das interfaces providas e requeridas possuem parâmetros associados.

Um exemplo de componente pode ser visto na figura 2.2. Nela, é possível ver o modelo de um componente que foi projetado para coletar dados de sensores. As interfaces fornecidas incluem métodos para adicionar, remover, iniciar, parar e testar os sensores. Já as interfaces requeridas, incluem métodos que permitem conectar o componente a sensores de dados e a um gerenciador de sensores.

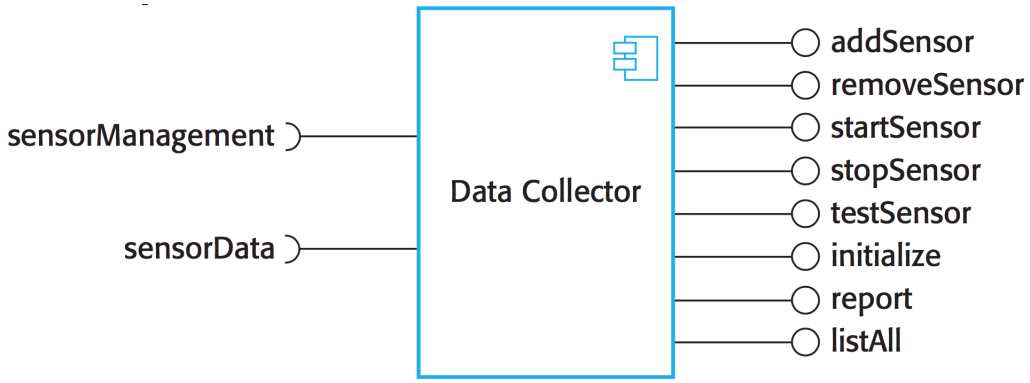

Figura 2.2: Modelo de um componente de coleta de dados.[7]

2.2.2 Modelo de Componentes

Como visto, para garantir que os softwares baseados em componentes sejam desenvolvidos de maneira correta, é necessário que sejam seguidas algumas regras e padrões de implementação, infraestrutura, implantação etc. Essas regras permitem que os componentes fornecidos possam funcionar corretamente sozinhos ou em conjunto nos ambientes nos quais se deseja introduzi-los. Esse conjunto de regras e padrões, é chamado de modelo de componentes [9]. Alguns exemplos de modelos de componentes são o Java EE (Java Platform, Enterprise Edition)[12] e o .NET[13]. Abaixo, são listados e descritos alguns dos elementos e regras básicas que constituem um modelo de componentes [7][9].

- **Interfaces**: Uma interface é uma especificação dos serviços que um cliente pode solicitar de um componente. Um componente fornece uma implementação desses serviços e os usuários de um componente devem se comunicar com os componentes através dessas interfaces. Os componentes também podem ser alterados internamente, mas devem continuar a satisfazer os contratos definidos pelas suas interfaces.

- **Interoperabilidade**: Deve haver um *middleware* que fornece suporte para a integração correta dos componentes, permitindo que componentes independentes e distribuídos possam funcionar corretamente juntos. O *middleware* deve garantir que o desenvolvedor tenha o mínimo de trabalho para configurar e manipular os componentes, permitindo que ele tenha mais tempo para focar em problemas específicos da aplicação. Além dessas tarefas, o *middleware* também pode oferecer suporte para alocação de recursos, gerenciamento de transações, segurança e concorrência.

- **Customização**: Um consumidor deve poder adaptar um componente antes da sua instalação ou uso. Como os componentes são geralmente tratados como uma caixa-preta - revelando o mínimo da sua implementação - os componentes só podem ser personalizados usando interfaces de personalização claramente definidas. Uma interface de personalização permite que ferramentas de personalização e implantação modifiquem desde propriedades simples a comportamentos complexos.

- **Composição**: A composição de componentes é a combinação de dois ou mais componentes de software que produzem um novo comportamento de componente. Um padrão de composição per-

mite a criação de uma estrutura maior, composta de vários componentes. Os componentes dessa infraestrutura geralmente interagem uns com os outros através de invocações de métodos. Além disso, é importante que o modelo de componentes defina como as interfaces devem ser modeladas para apoiar tal composição.

- **Suporte a evolução**: Softwares baseados em componentes devem permitir a evolução do sistema no qual são aplicados. Componentes que atuam como um servidor para outros componentes podem ter que ser substituídos por versões mais recentes que oferecem funcionalidades novas ou melhoradas. Uma nova versão pode não só ter uma implementação diferente, mas pode fornecer interfaces modificadas ou novas. Os clientes existentes de tais componentes, idealmente, não devem ser afetados ou devem ser afetados minimamente possível por essas mudanças. Além disso, versões antigas e novas de um componente devem poder coexistir no mesmo sistema. Portanto, regras e padrões de evolução dos componentes e o controle de versão são extremamente importantes como parte de um modelo de componentes.

- **Configuração e implantação**: Um padrão de implantação especifica como um componente deve ser configurado e instalado em uma infraestrutura de componentes. Além disso, devem existir serviços para suportar a criação e gerenciamento das instâncias dos componentes.

Para que o desenvolvimento baseado em componentes seja realizado de maneira correta, é necessário, além de seguir um modelo de componentes, que todo o processo de desenvolvimento do software seja adaptado para incluir atividades que têm o objetivo de identificar, criar e integrar os componentes reutilizáveis. Em essência, as principais diferenças entre o processo padrão de desenvolvimento de softwares e o desenvolvimento baseado em componentes, são [7]:

1. Os requisitos dos usuários não devem ser muito detalhados, pois requisitos que são muito específicos limitam o número de componentes que poderiam atender a esses requisitos.

2. Os requisitos devem ser refinados e modificados dependendo dos componentes disponíveis. Se os requisitos do usuário não puderem ser supridos com os componentes disponíveis, é necessário avaliar os requisitos relacionados que podem ser apoiados.

3. Existe uma nova atividade de busca e refinamento de componentes após a arquitetura do sistema ter sido projetada. Essa atividade tem o objetivo de reavaliar os componentes de maneira a verificar se eles se adequam corretamente aos outros componentes.

4. O desenvolvimento passa a ser um processo de composição, onde os componentes são integrados uns aos outros com o objetivo de criar uma aplicação. Além disso, é possível adicionar novas funcionalidades além das fornecidas pelos componentes.

2.3 Frameworks

Outra técnica de reuso de software, e que por vezes também é combinada com a técnica de desenvolvimento de softwares baseados em componentes, é a utilização de frameworks para o desenvolvimento de

aplicações. Segundo (Bosch et al., 2000)[14], um framework é um sistema incompleto que fornece padrões de design e arquitetura, e um conjunto de implementações concretas e parciais, que podem ser utilizadas e adaptadas para criar aplicações completas para um determinado domínio. Uma outra definição de frameworks é:

> "Um conjunto de classes cooperantes que compõem um design reutilizável para uma classe específica de softwares. Um framework fornece um padrão de arquitetura, dividindo o design do software em classes abstratas e definindo suas responsabilidades e colaborações. Um desenvolvedor personaliza o framework para uma aplicação específica através da criação de subclasses e composição das instâncias das classes do framework".(Gamma et al., 1994)[15]

Portanto, a principal aplicação dos frameworks é permitir o desenvolvimento rápido de uma família de aplicações similares em um domínio, através da utilização, adaptação e extensão dos elementos fornecidos pelo framework. Esse processo de criação de aplicações a partir de um framework é chamado de *instanciação de framework*, e a aplicação resultante é chamada de *instância do framework* [14].

2.3.1 Características dos Frameworks

Embora também sejam considerados uma técnica de reuso, os frameworks possuem algumas características adicionais em relação às outras técnicas. Essas características são [16]:

- **Inversão de controle**: Para minimizar o esforço e a complexidade na implementação de algumas funcionalidades de uma aplicação, é muito comum que sejam utilizadas bibliotecas. Para utilizar as funcionalidades oferecidas por essas bibliotecas, a aplicação que a utiliza deve chamar tais funções. Portanto, é a aplicação quem tem o controle sobre a chamada e tratamento dos retornos das funcionalidades da biblioteca. No entanto, nos frameworks, esse controle é invertido, pois é o próprio framework quem controla o fluxo de execução dos códigos fornecidos pelas instâncias que o utiliza.

- **Arquitetura definida e flexível**: Os frameworks definem uma arquitetura base, que deve ser seguida pelas suas instâncias. Essa arquitetura fornecida tem o objetivo de fornecer um conjunto de configurações e elementos que são imutáveis e devem ser seguidos por todas as instâncias do framework, e portanto, não podem ser alterados. Tais elementos imutáveis, são chamados *frozen-spots*.

- **Adaptação através de pontos de variação**: Ao mesmo tempo que a arquitetura de um framework define um conjunto de *frozen-spots*, ela também define alguns pontos de extensão e modificação. Tais pontos de extensão e modificação, são chamados de *hot-spots*, e são necessários para que o framework possa ser utilizado por uma infinidade de aplicações em um domínio específico, de maneira que os usuários do framework possam realizar adaptações e extensões de acordo com os requisitos da instâncias que eles estejam criando.

2.3.2 Elementos de um Framework

Para que os frameworks possam oferecer as características mencionadas e cumprir seus requisitos de reusabilidade, extensão e flexibilidade, eles devem ter minimamente, os elementos listados abaixo [14]. A figura 2.3 mostra, de maneira geral, como esses elementos se relacionam em um framework:

1. **Documentos de Design**: Inicialmente, o design de um framework pode consistir de diagramas de classes, sequências (ou outros diagramas), texto escrito ou apenas uma ideia na cabeça dos desenvolvedores. De maneira geral, as documentações do framework tem o objetivo de auxiliar tanto os desenvolvedores a criar, manter e evoluir o framework, quanto auxiliar os usuários do framework a utilizá-lo da maneira correta.

2. **Componentes**: Um componente de um framework tem a mesma definição de componentes visto na técnica de desenvolvimento de softwares baseados em componentes. Assim, os componentes de um framework têm o objetivo de fornecer um conjunto de serviços através da utilização correta de suas interfaces providas e requeridas.

3. **Interfaces**: As interfaces de um framework descrevem um conjunto de serviços providos e requeridos por uma determinada classe ou componente do framework.

4. **Classes abstratas**: As classes abstratas são uma implementação incompleta. Elas podem ser utilizadas para definir comportamentos comuns para um grupo de classes e componentes do framework, e também para as implementações concretas delas, que são realizadas pelas instâncias do framework.

5. **Classes**: No nível mais baixo do framework, estão as classes. As classes só diferem dos componentes no fato de que seus métodos fornecidos e requeridos não são representados por interfaces do framework. Normalmente, as classes são usadas internamente pelos componentes para realizar ou delegar funcionalidades, ou seja, os usuários do framework não verão ou terão acesso a essas classes.

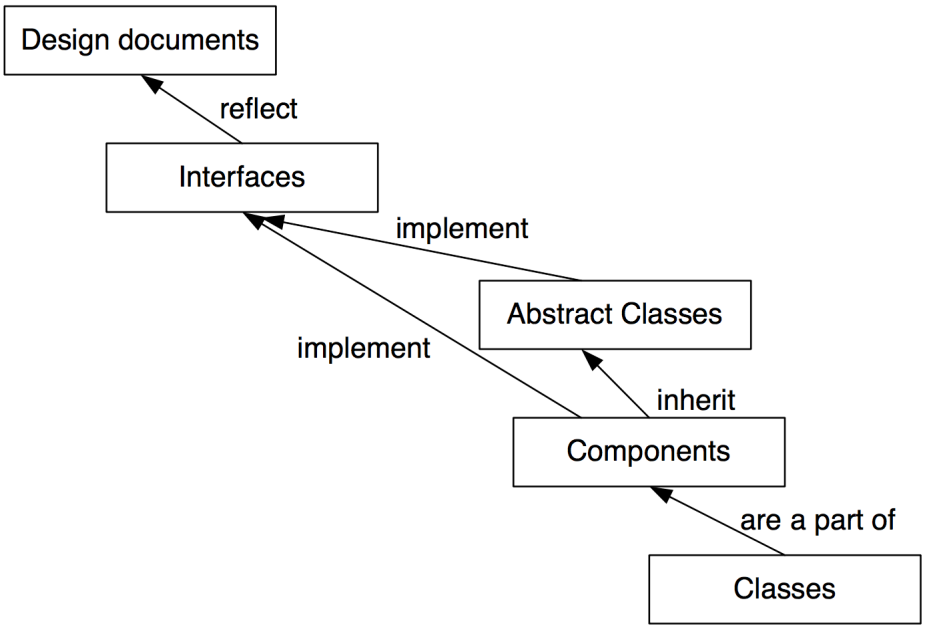

Figura 2.3: Relação entre os elementos de um framework.

Percebe-se, portanto, que para se desenvolver frameworks flexíveis e de qualidade, é importante que se faça bastante uso de técnicas e conceitos de desenvolvimento de softwares orientados a objetos.

2.3.3 Classificação dos Frameworks

Existem diversas maneiras de se classificar os frameworks, seja pelo seu propósito, técnica desenvolvimento, ou flexibilidade. A seguir serão mostradas algumas classificações de acordo com o propósito do framework [14].

- **Frameworks de aplicação**: Os frameworks de aplicação visam fornecer um conjunto completo de funcionalidades normalmente necessárias para se desenvolver aplicações completas. Essas funcionalidades geralmente envolvem, coisas como uma GUI, gerenciamento de documentos, bancos de dados, etc.

- **Frameworks de domínio**: Quando uma aplicação deve ser criada para um domínio específico (*e.g.* sistema bancário, sistemas de alarme, ERP etc.), geralmente ela deve ser desenvolvida a partir do zero, o que gera um alto grau de complexidade, e tempo para o desenvolvimento. Os frameworks de domínio são utilizados, portanto, para facilitar a criação de aplicações para esses domínios específicos, reduzindo a quantidade de trabalho que precisa ser feito para implementá-los, fornecendo para isso um conjunto de funcionalidades comumente encontradas em aplicações dos domínios para os quais foram desenvolvidos.

- **Frameworks de suporte**: Os frameworks de suporte têm o objetivo de fornecer ferramentas para domínios menores e mais específicos, como gerenciamento de memória, arquivos, mídia, etc. Por

conta disso, os frameworks de suporte geralmente são utilizados em conjunto com os frameworks de aplicação e os frameworks de domínio.

Além da classificação dos frameworks de acordo com o seu propósito, os frameworks também são classificados de acordo com seu nível de flexibilidade. Essas categorias são *whitebox* e *blackbox*:

- Nos frameworks *whitebox*, os componentes não são fornecidos diretamente para uso pelo usuário, sendo necessário a realização de implementações de interfaces e a criação de subclasses para que o componente possa ser utilizado. Por conta disso, é necessário que o usuário tenha entendimento sobre o funcionamento interno do framework, daí o nome *whitebox* (caixa branca). Por conta disso, os frameworks *whitebox* são mais flexíveis e extensíveis, pois o usuário pode alterar o comportamento dos componentes de acordo com suas necessidades, mas ao mesmo tempo, isso o torna mais difícil de usar, pois os usuários necessitam de um maior conhecimento sobre o funcionamento interno do framework.

- Nos frameworks *blackbox*, os detalhes do funcionamento interno do framework são (quase todos) escondidos do usuário, cabendo a ele apenas escolher e combinar os componentes que ele deseja usar. Para isso é necessário que ele tenha conhecimento apenas sobre quais componentes são fornecidos pelo framework, suas interfaces e especificações. Por conta disso, os frameworks *blackbox* são mais fáceis de usar, no entanto, eles são menos flexíveis, pois o usuário não pode alterar o comportamento interno dos componentes.

Quanto a técnica utilizada para desenvolver o framework, eles podem ser classificados em [16]:

- **Frameworks Orientados a Objetos**: Os frameworks orientados a objetos são os frameworks "padrão", que definem um design e um conjunto de classes concretas e abstratas, que têm o objetivo de fornecer uma implementação parcial para permitir a criação de uma família de aplicações em um domínio específico. Os frameworks orientados a objetos geralmente são utilizados através da criação de subclasses e implementações concretas das classes abstratas fornecidas por ele, sendo necessário que o usuário tenha conhecimento das relações entre essas classes fornecidas. Portanto, um framework orientado a objetos pode ser classificado como um framework *whitebox*.

- **Frameworks Baseados em Componentes**: Diferente dos frameworks orientadas a objetos, o foco do framework baseado em componentes está nos componentes oferecidos por ele e suas interfaces. Consequentemente, os frameworks baseados em componentes são definidos como uma coleção de diferentes componentes, cada um com um comportamento e propósito específico, que podem ser combinados de uma maneira pré-definida com objetivo de realizar um conjunto de tarefas em um domínio específico. Portanto, os frameworks baseados em componentes são utilizados através da escolha e composição correta dos componentes oferecidos por ele, cabendo ao usuário realizar a implementação de algumas interfaces que garantam o funcionamento e comunicação de tais componentes, não sendo necessário que eles tenham conhecimento sobre o funcionamento interno dos componentes. Por conta disso, os frameworks baseados em componentes podem ser classificados como frameworks *black-box*.

2.4 UML-F

Uma das principais atividades do desenvolvimento de softwares é a criação de diagramas e modelos, que têm como objetivo facilitar o planejamento e análise do software que se deseja criar permitindo, inclusive, que sejam realizadas melhorias, e identificação e correção de erros, antes mesmo que o software seja de fato implementado. Por conta disso, é importante que sejam criados modelos e diagramas durante o desenvolvimento de frameworks.

Uma das principais linguagens utilizadas para a criação de modelos e diagramas é a UML (Unified Modeling Language) [17]. No entanto, segundo (Fontoura; Pree; Rumpe, 2000)[18], a UML padrão não fornece elementos apropriadas para a criação de modelos para o desenvolvimento de frameworks, principalmente pelo fato de a UML não prover em seus diagramas, meios para indicar quais são os pontos de variação e quais são as restrições de instanciação. Por conta dessas e outras limitações, (Fontoura; Pree; Rumpe, 2000)[18], propõe a criação de várias extensões a UML padrão para permitir a modelagem explícita da estrutura e comportamentos permitidos dos pontos de variação de frameworks. Tais extensões foram definidas principalmente pela aplicação dos mecanismos de extensibilidade integrados a UML. Essas extensões formam uma base para um novo perfil UML que é próprio para o desenvolvimento de frameworks, o UML-F [18].

2.4.1 Extensões UML-F

Na UML-F, os diagramas UML são estendidos através da adição de novas tags. A tabela 2.4 mostra de maneira resumida essas novas extensões propostas pela UML-F e as descrições de suas aplicações.

Nome da extensão	Tipo de extensão	Aplicável as notações dos elementos UML	Descrição
{appl-class}	Boolean Tag	Classe	Classes que existem apenas em instâncias do framework. Novas classes específicas de aplicação podem ser definidas durante a instanciação do framework.
{variable}	Boolean Tag	Método	O método deve ser implementado durante a instanciação do framework.
{extensible}	Boolean Tag	Classe	A interface da classe depende da instanciação do framework: novos métodos podem ser definidos para estender a funcionalidade da classe.
{static}	Boolean Tag	Interfaces Extensíveis, Métodos Variáveis e Classes Extensíveis	O ponto de variação não requer instanciação em tempo de execução. As informações em falta devem ser fornecidas em tempo de compilação.
{dynamic}	Boolean Tag	Interfaces Extensíveis, Métodos Variáveis e Classes Extensíveis	O ponto de variação requer instanciação em tempo de execução. As informações em falta podem ser fornecidas somente durante o tempo de execução.
{incomplete}	Boolean Tag	Generalização e Realização	Novas subclasses podem ser adicionadas nesta relação de generalização ou realização.
{optional}	Boolean Tag	Eventos	Indica que um determinado evento é opcional. É útil para especificar um comportamento que deve ser seguido pelo ponto de variação instanciado.

Tabela 2.4: Resumo dos elementos UML-F e seus significados. [18]

2.4.2 Exemplo de Aplicação

Como exemplo de aplicação da UML-F na definição de pontos de variação de um framework, será utilizado o mesmo exemplo apresentado em [18]. Nesse exemplo, é utilizado um subsistema estudantil de um framework para o desenvolvimento de sistemas de educação baseados na web [19]. As figuras 2.4, 2.5 e 2.6, mostram, respectivamente, um diagrama de classes UML representando uma visão estática do sistema, um diagrama de sequências baseado na UML, representando a interação entre as instâncias das duas classes da figura 2.5, e uma representação UML da hierarquia da classe *Actor* do sistema. A partir

dessas figuras, (Fontoura; Pree; Rumpe, 2000)[18] afirmam que:

- O método *selectCourse()* é um método abstrato de uma classe abstrata *SelectCourse*. Isso significa que durante a instanciação do framework, os usuários devem criar subclasses de *SelectCourse* e fornecer uma implementação concreta do método *selectCourse()*. O problema com essa representação, é que embora *selectCourse()* seja um ponto de variação, nos diagramas do projeto não há nenhuma indicação de como ele deve ser instanciado.

- O método *tipOfTheDay()* também é um ponto de variação do framework. A razão é que alguns aplicativos criados a partir do framework podem querer mostrar dicas enquanto outros não o fazem. No entanto, embora não seja especificado ou sinalizado no diagrama, sua utilização não é obrigatória, o motivo, é que isso poderia levar a uma interface complexa para *ShowCourse*, com muitos métodos que não seriam necessários em várias instâncias do framework.

- A hierarquia da classe *Actor* também é um ponto de variação e é usada para permitir que novos tipos de atores sejam definidos dependendo dos requisitos de uma determinada instância do framework. Os tipos de ator padrão são estudantes, professores e administradores, no entanto, novos tipos podem ser necessários, como bibliotecários e secretários. Isso significa que os aplicativos criados a partir do framework sempre possuem pelo menos três tipos de atores, alunos, professores e administradores, mas vários outros tipos de atores podem ser definidos de acordo com os requisitos específicos do aplicativo. No entanto, assim como no caso anterior, esse ponto de variação também não é especificado claramente no diagrama UML.

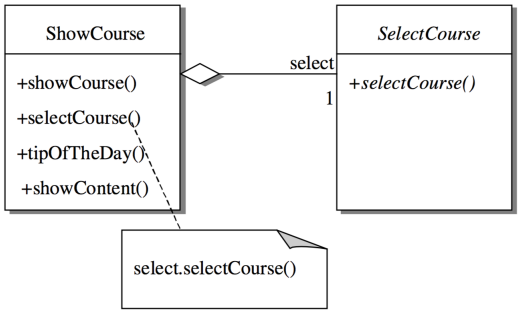

Figura 2.4: Diagrama de classes UML do subsistema estudantil.[18]

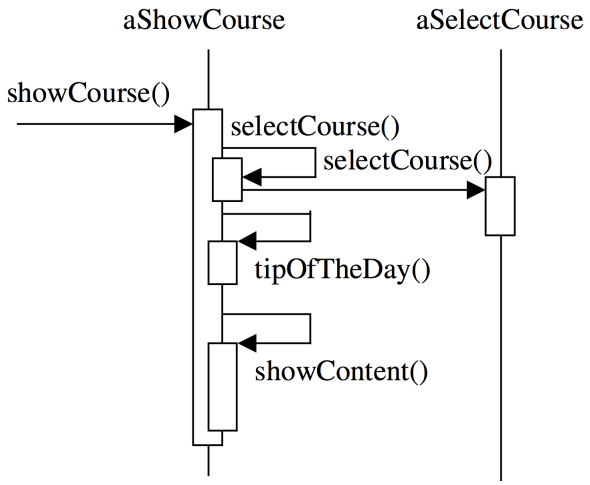

Figura 2.5: Diagrama de sequências do subsistema estudantil.[18]

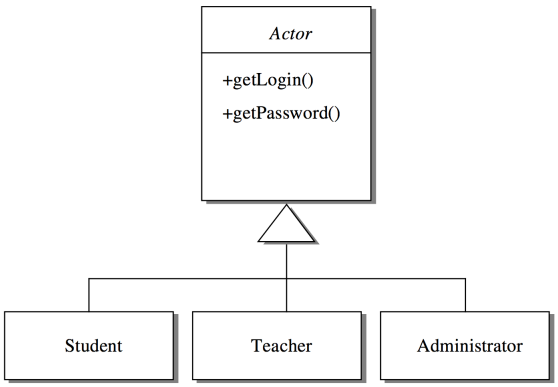

Figura 2.6: Hierarquia da classe *Actor*.[18]

Para solucionar esses problemas de representação, é utilizado UML-F para remodelar os diagramas e garantir a definição e representação explícita dos pontos de variação do framework. Na figura 2.7 é mostrada a remodelagem dos diagramas de classe e na figura 2.8 é dado um exemplo de como um diagrama de sequência da UML-F pode ser utilizado para limitar o comportamento dos pontos de variação de um framework.

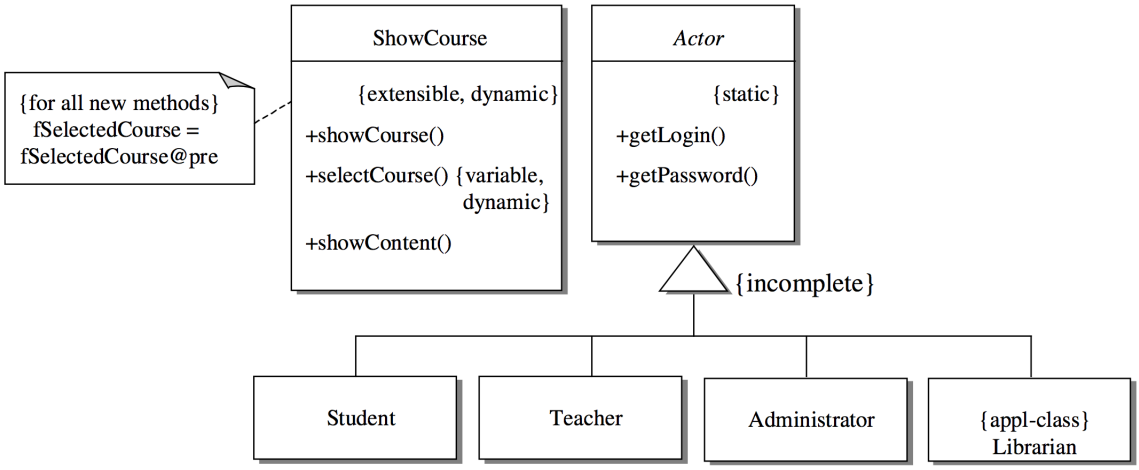

Figura 2.7: Diagrama de classes UML-F.[18]

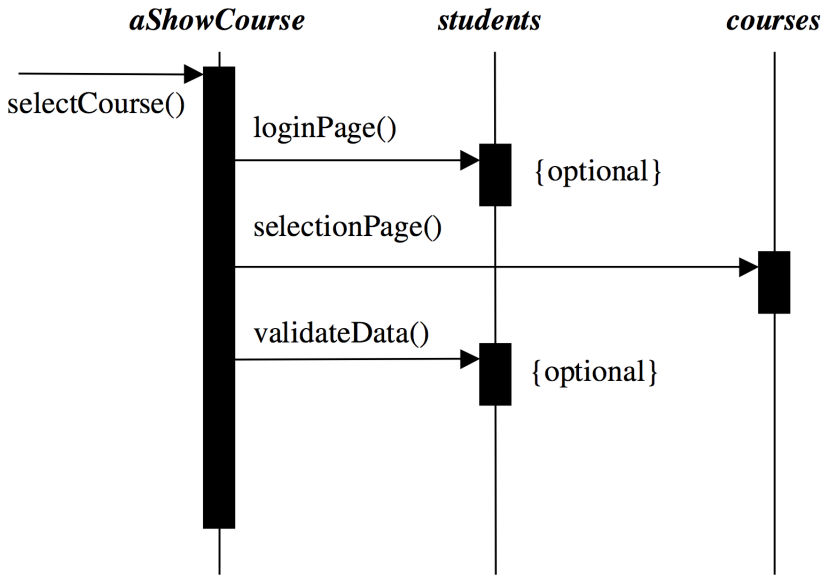

Figura 2.8: Diagrama de sequência UML-F.[18]

A partir das figuras 2.7 e 2.8, (Fontoura; Pree; Rumpe, 2000)[18] fazem as seguintes afirmações:

- O método *selectCourse()* é marcado com uma tag {*variable*} para indicar que sua implementação pode variar dependendo da instanciação do framework e deve ser implementado com um comportamento específico para cada instância dele.

- Na classe *ShowCourse* é utilizada a tag {*extensible*} para indicar que sua interface pode ser estendida durante a instanciação do framework, permitindo a adição de novas funcionalidades, como o método *tipOfTheDay()* - representado no diagrama UML da figura 2.4. No entanto, isso não significa que novos métodos devem ser adicionados diretamente a classe.

- A tag {*incomplete*} é utilizada no relacionamento de generalização da classe *Actor* para indicar que novas subclasses de *Actor* podem ser fornecidas para atender aos requisitos de aplicações criadas a partir do framework. Essa tag permite que o usuário do framework crie tantas classes de uma determinada interface extensível quanto for necessário.

- A tag {*app-class*} é usada para indicar um espaço reservado no framework, onde as classes específicas da aplicação podem ser ou já foram adicionadas, um exemplo, é a classe *Librarian*.

- As tags {*dynamic*} e {*static*} indicam se a instanciação de tempo de execução é necessária. A tag {*dynamic*} é utilizada porque é um requisito do usuário ter uma reconfiguração dinâmica para os pontos de variação que lidam com a exibição do curso. Já a tag {*static*} é utilizada para a interface extensível *Actor*, já que os novos tipos de atores não precisam ser definidos durante o tempo de execução.

- O diagrama de sequência mostra o padrão de interação principal para um aluno selecionando um curso. Como pode ser decidido pela implementação real, é opcional se o aluno deve fazer o login antes de selecionar um curso ou se os dados são validados. Esse tipo de opção é definido usando a tag {*optional*}, que é utilizada para indicar interações que não são obrigatórias e até mesmo para definir fluxos alternativos.

A partir desses diagramas, é possível perceber que eles fornecem uma boa especificação dos pontos de variação e suas restrições de instanciação. Inclusive, (Fontoura; Pree; Rumpe, 2000)[18], sugerem que tais diagramas podem ser utilizados como uma alternativa para as extensas documentações de frameworks existentes, que simplesmente indicam tais extensões e restrições através de grandes documentos contendo as hierarquias das classes do framework.

Capítulo 3

Metodologia

Quando utilizamos um framework, esperamos que ele seja robusto (*i.e.* execute corretamente e não apresente problemas, mesmo em casos imprevistos), flexível (*i.e.* pode ser usado em muitas aplicações aparentemente diferentes), extensível (*i.e.* pode ser estendido com o mínimo de modificação do código existente) e fácil de manter (*i.e.* pode ser facilmente modificado para melhorar o desempenho ou corrigir "bugs") [8].

No entanto, para se desenvolver frameworks com tais características, é necessário um longo processo iterativo e incremental de desenvolvimento. Esse tipo de abordagem incremental é necessária, pois dificilmente se tem o *insight* necessário para identificar todas as abstrações e solucionar todos os problemas de maneira adequada de uma única vez. Inclusive, é muito comum que algumas abstrações se tornem evidentes depois que o framework já tenha sido criado e utilizado, fazendo com que o framework tenha que ser analisado e refinado novamente [14]. Portanto, deve-se ter em mente que durante o processo desenvolvimento de frameworks, mudanças de comportamento, arquitetura e implementação ocorrem com frequência.

Por conta desses problemas, é extremamente necessário utilizar um modelo de processo adequado para o desenvolvimento de frameworks. Tal modelo deve fornecer um processo iterativo e que auxilie todas as etapas de desenvolvimento, desde o design e planejamento, até a criação e evolução do framework.

Embora existam diversos modelos de processos para o desenvolvimento de softwares, nem todos se mostram adequados para o desenvolvimento de frameworks. Os métodos de desenvolvimento ágil [20], por exemplo, focam no desenvolvimento e evolução incrementais do software, de maneira que as funcionalidades mais simples e essenciais para o funcionamento do software sejam implementadas iterativamente sob demanda. No entanto, geralmente não são considerados outros requisitos de flexibilidade e reúso, pois assume-se que é mais fácil alterar o código fonte original para implementar novas funcionalidades do que adicionar flexibilidade com antecedência para fornecer suporte a reutilização e evolução do software. Portanto, os métodos ágeis não visam a criação de arquiteturas de software flexíveis e reutilizáveis, logo, métodos ágeis não se mostram adequados para serem usados no desenvolvimento de frameworks [16].

Já os métodos mais "pesados", como o RUP (Rational Unified Process) [21] , apesar de fornecerem processos robustos para o desenvolvimento de softwares com arquiteturas flexíveis e reutilizáveis para

um domínio específico e concreto, também não se mostram muito adequados para o desenvolvimento de frameworks, pois quando se gerencia um framework, é necessário levar em conta não só o desenvolvimento e evolução do próprio framework, mas também das diversas aplicações concretas que serão criadas a partir dele. Além disso, a quantidade de etapas e artefatos necessários para aplicar corretamente esses tipos de processo, geram, além de uma complexidade desnecessária, aumento no custo e no tempo de produção do software.

Por outro lado, modelos de processo leves costumam definir apenas alguns artefatos e exigem menos esforço para serem gerenciados. Como um framework sofre diversas mudanças durante seu desenvolvimento, sendo necessário que os artefatos que o descrevem sejam revisados e reescritos constantemente, é importante reduzir o esforço na criação e gerenciamento desses artefatos, de modo que só seja produzido o necessário para a execução da iteração. Consequentemente, o uso de modelos de processo leves proporcionam um gerenciamento mais eficiente do desenvolvimento de frameworks [16].

3.1 Modelo de Processo *ProMoCF*

Atualmente, existem alguns processos para o desenvolvimento de frameworks orientados a objetos, como o *hot-spot-driven* [22]. Nesse método, utilizam-se *hot-spot-cards*, que são adaptações dos cartões CRC (Class Responsibility Collaboration) [23], para identificar e gerenciar os requisitos e abstrações que serão aplicados no framework.

Embora o método hot-spot-driven seja amplamente utilizado, segundo (Scherp, 2007)[16], por se tratar de um método para o desenvolvimento de frameworks orientados a objetos, ele não é adequado para o desenvolvimento de frameworks baseados em componentes devido a dois fatores que se seguem:

1. Na abordagem hot-spot-driven, os cartões CRC são utilizados para identificar as relações e classes abstratas do modelo de objetos do framework. No entanto, apesar de serem úteis na identificação dos componentes do framework, depois de definí-los, eles não são reconsiderados durante o restante do processo. Portanto, para fornecer suporte para o desenvolvimento de frameworks baseados em componentes, é necessária uma atividade explícita para a identificação dos componentes.

2. As orientações fornecidas em relação ao hot-spots com granularidade de apenas um método não são suficientes. Além disso, não fica claro a quantidade de funcionalidades que cada hot-spot deve ter. Por esse motivo, eles definem que a granularidade dos hot-spots e hot-spot-cards devem ser de exatamente um método. Além disso, eles introduzem o conceito de *group-hot-spot-cards*, que são agrupamentos lógicos de hot-spot-cards de granularidade de um método. Esses hot-spot-card-groups são usados para identificar as funcionalidades e os componentes do framework e especificar os requisitos de flexibilidade desses componentes.

   A estrutura dos group-hot-spot-cards é semelhante à estrutura dos hot-spot-cards. Eles têm um nome, uma descrição, uma breve descrição do comportamento do group-hot-spot-card em pelo menos duas situações concretas e duas *checkboxes* para determinar se é necessário realizar uma adaptação em tempo de execução, e se é necessário fornecer uma ferramenta de configuração para

usuários finais. Opcionalmente, pode ser adicionada uma lista dos hot-spot-cards correspondentes a esse group-hot-spot-card.

Devido a tais fatores, eles propõem a criação de um novo modelo de processo baseado no método *hot-spot-driven*, mas que é adaptado para o desenvolvimento de frameworks baseados em componentes, o ProMoCF (*Process Model for Component Frameworks*). A figura 3.1 mostra uma visão geral das etapas desse processo, e a seguir, são apresentadas as descrições dessas etapas:

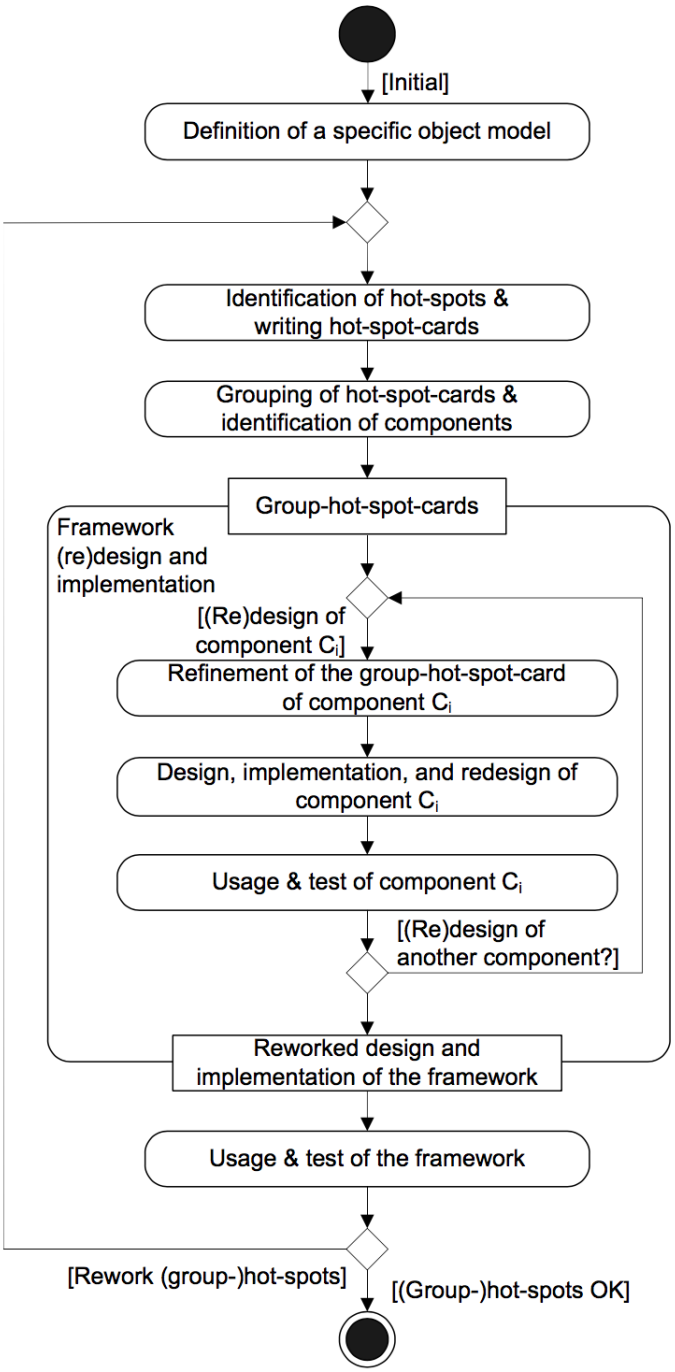

Figura 3.1: Visão geral das etapas do ProMoCF.[16]

- **Definição de um modelo de objeto específico**

Normalmente, os processos de desenvolvimento de softwares baseados em componentes começam por definir os componentes do aplicativo. Em contraste, no modelo ProMoCF, a tarefa inicial corresponde na criação de um modelo de objetos inicial. Este modelo de objeto é criado a partir da análise de aplicativos existentes no domínio considerado. É necessário começar o processo criando um modelo de objeto inicial, pois as funcionalidades e os componentes do framework surgem a partir de um processo gradual de abstração do modelo de objetos.

Caso as aplicações de modelo consideradas inicialmente já sejam baseadas em componentes, também pode ser criado um modelo de componentes inicial. Nesse caso, o modelo de objeto específico é acumulado pelos modelos de objetos dos componentes no modelo de componente.

- **Identificação de hot-spots e criação de hot-spot-cards**

  Um dos principais desafios ao se desenvolver frameworks, é identificar os pontos onde ele deve ser flexível. O problema é que é muito difícil identificar os aspectos do domínio que podem ser generalizados ou abstraídos apenas olhando para aplicações concretas. Porém há meios de tornar essa tarefa mais fácil.

  Um dos métodos que auxiliam na identificação dos hot-spots de um framework são os *hot-spot-cards*. Esses *hot-spot-cards* nada mais são do que variantes dos cartões CRC. Eles fornecem um meio simples, mas eficaz, para documentar e comunicar os requisitos de flexibilidade do framework. Além disso, a utilização dos *hot-spot-cards* permite não só facilitar a identificação dos hot-spots, mas também apoiar todo o projeto de design do framework.

  Os *hot-spot-cards* podem ser divididos em duas categorias, os *hot-spot-cards de dados* e os *hot-spot-cards de função*. Os *hot-spot-cards de dados* são utilizados para identificar e sinalizar os elementos do domínio que podem ser generalizados ou abstraídos durante o desenvolvimento do framework. Já os *hot-spot-cards de funções* são utilizados para identificar e sinalizar as funcionalidades que devem ser mantidas flexíveis durante o desenvolvimento do framework.

  Por ser uma adaptação do método *hot-spot-driven*, o fluxo principal de iteração do ProMoCF começa com a identificação de hot-spots e a criação de hot-spot-cards.

- **Agrupamento de hot-spot-cards e identificação de componentes**

  Nessa etapa, os hot-spot-cards são organizados em grupos, de maneira que os hot-spot-cards que descrevem requisitos de flexibilidade para tarefas similares ou relacionadas, pertençam a um mesmo grupo lógico. Para cada grupo lógico de hot-spot-cards criado, tem-se um group-hot-spot-card correspondente.

  É completamente razoável considerar que cada um desses grupos lógicos seja considerado um componente do framework. Além disso, é possível criar group-hot-spot-cards que sejam constituídos de outros group-hot-spot-cards. Como conseqüência, agrupar group-hot-spot-cards significa construir uma hierarquia de componentes. Dessa maneira, o ProMoCF promove o desenvolvimento de instâncias flexíveis dos componentes do framework.

É importante ressaltar que nesta etapa é possível adicionar novos group-hot-spot-cards caso eles não tenham sido definidos nas iterações anteriores. Isso pode acontecer quando não fica muito claro quais hot-spot-card constituem um group-hot-spot-cards.

Como exemplo de criação de group-hot-spot-cards, consideremos um framework para gerenciamento de arquivos e pastas. Nesse framework, seria razoável agrupar um hot-spot-card *Adicionar um documento a uma pasta* juntamente com um hot-spot-card *Remover um documento de uma pasta*, formando-se assim um group-hot-spot-card, que nesse caso, poderia se chamar *Gerenciar documentos em uma pasta*, e corresponderia a um componente de gerencia de documentos no framework.

- **Design e Implementação**

  Nessa etapa, são realizadas as atividades de design e implementação do framework. A principal diferença dessa etapa em relação ao processo hot-spot-design original, é que ele possui subatividades para o desenvolvimento dos componentes do framework. Para cada componente que deve ser projetado e/ou melhorado na iteração corrente dessa etapa, as seguintes subatividades são realizadas:

  1. Novos hot-spots do group-hot-spot-cards são identificados. Isso resulta em hot-spot-cards e group-hot-spot-cards refinados.
  2. O componente é projetado e implementado. Para isso, as interfaces do componente são especificadas e pelo menos uma instância concreta do componente é implementada. Essa atividade também inclui a escrita da documentação necessária para que os usuários do framework saibam como utilizar o componente.
  3. A implementação concreta é testada, aplicando-a em um ambiente de teste. O design e a implementação do componente podem ser revisados e aprimorados nas seguintes iterações da etapa de design e implementação.

  É importante saber que, uma vez que nem todos os componentes são refinados a cada iteração, eles podem ter maturidade diferente. Por exemplo, enquanto alguns componentes já estão na fase de implementação, outros ainda podem estar na fase de requisitos ou de design. Além disso, o desenvolvimento dos componentes do framework também pode ser conduzido concorrentemente com essa etapa, embora isso não seja mostrado no diagrama. O resultado final dessa etapa, é um design revisado e uma implementação melhorada dos componentes do framework.

- **Utilização e Teste**

  Nessa etapa, o framework é testado, avaliando-se a qualidade e reutilização de seu design. A única maneira de identificar erros e pontos fracos do design, é desenvolver aplicações concretas que usam o framework criado. O desenvolvimento de uma aplicação concreta significa compor componentes do fraemwork de acordo com as regras definidas por ele. Os componentes utilizados na criação da aplicação concreta podem ser os fornecidos pelo framework, mas também podem ter sido criados através da modificação dos mesmos.

  Os erros identificados, devem ser imediatamente corrigidos e os pontos fracos no design, devem ser melhorados na próxima iteração da etapa de design e implementação.

Portanto, o ProMoCF, por se tratar de um processo simples, iterativo e flexível para o desenvolvimento de frameworks baseados em componentes, se torna ideal para a criação rápida de frameworks não muito complexos. Consequentemente, na maioria dos casos, os recursos necessários para conduzir o desenvolvimento de um framework utilizando o ProMoCF são bem menores se comparados a outros processos, principalmente pelo fato de que somente os documentos relacionados aos hot-spot-cards, group-hot-spot-cards e a documentação do código fonte são escritas durante o processo de desenvolvimento do fraemwork. Muitas vezes, uma documentação completa só é escrita se uma versão do framework deve ser entregue para uma organização externa.

Capítulo 4

MyUni

Como visto anteriormente, várias universidades procuram fornecer a seus alunos aplicativos que auxiliam na gerência de informações acadêmicas. No entanto, desenvolver tais aplicativos requer muito tempo e esforço. Porém, é possível minimizar esses fatores fazendo-se uso de técnicas de reúso de software, como a utilização de frameworks. Levando-se em conta esses fatores, propõe-se a criação de um framework baseado em componentes chamado MyUni - que é contração das palavras My (do inglês, "minha") e University (do inglês, "universidade").

O principal objetivo do MyUni é facilitar e acelerar o desenvolvimento de aplicativos para a gerência de informações acadêmicas, fornecendo para isso uma arquitetura reutilizável e um conjunto de componentes reutilizáveis que permitam a implementação de diversas funcionalidades comumente encontradas em aplicativos desse domínio. Ao fornecer tais componentes, muitos detalhes de implementação acabam sendo abstraídos, ficando a cargo do desenvolvedor realizar pequenas configurações e implementações específicas. Por conta disso, a complexidade no desenvolvimento de aplicativos para a gerência de informações acadêmicas acaba sendo reduzida, sendo possível, inclusive, que não só as universidades, mas também os próprios alunos, mesmo com pouca experiência de desenvolvimento, possam criar aplicativos para a gerência de informações acadêmicas para suas próprias universidades de maneira mais rápida.

Antes de se realizar todo o processo de desenvolvimento do MyUni, foi necessário escolher um modelo de processo que fosse ideal para o desenvolvimento de frameworks baseados em componentes. Por conta disso, o ProMoCF foi escolhido. Para realizar o desenvolvimento do framework, foi escolhida a plataforma Android como base para o design e implementação do MyUni. O Android foi escolhido, pois além de ser a plataforma mais utilizada em dispositivos móveis [24], ela é a mais acessível quando se trata de desenvolvimento de aplicativos, pois além de ter seu código-fonte aberto, é possível desenvolver aplicativos Android em ambientes Linux ou Windows, diferente da plataforma iOS, que só permite o desenvolvimento de seus aplicativos em sistemas operacionais da Apple, dificultando a aquisição de hardware e software para o desenvolvimento.

Assim, o desenvolvimento do MyUni foi realizado seguindo o modelo de processo ProMoCF, e de maneira que o framework, após ter sido implementado, pudesse ser utilizado para desenvolver aplicativos de gerência de informações acadêmicas para a plataforma Android. Nos capítulos a seguir, serão mostrados

mais detalhes sobre o processo de desenvolvimento do MyUni.

4.1 Definição de um Modelo de Objetos

Como visto, nessa etapa é necessário analisar algumas aplicações concretas do domínio com o objetivo de servirem como ponto de referência para a identificação e definição do conjunto de classes e elementos do domínio, assim como suas relações e interações, com o objetivo de facilitar a identificação dos componentes e funcionalidades que poderiam vir a constituir o MyUni. Para isso, três aplicativos foram selecionados, UFF Mobile, UFPE Mobile e PUC-SP Mobile. Esses aplicativos foram selecionados por dois principais motivos: primeiro, por ser possível utilizar as funcionalidades oferecidas por eles sem a necessidade de credenciais de acesso, pois na grande maioria dos aplicativos para a gerência de informações acadêmicas, é necessário possuir um login e senha, que só é fornecido para os alunos daquela universidade; segundo, porque eles oferecem um conjunto de funcionalidades que foram julgadas essenciais para os alunos que utilizam tais aplicativos.

Antes de se realizar a definição do modelo de objetos, foi realizada uma análise inicial para identificar quais funcionalidades são comumente encontradas nos três aplicativos analisados, e também quais funcionalidades são exclusivas entre eles. Durante essa análise, verificou-se que, nos três aplicativos, as funcionalidades oferecidas por eles eram agrupadas por seções de acordo com o contexto da funcionalidade. Por conta disso, foi utilizado o conceito de serviços, de maneira que cada seção fosse representada como um serviço e o conjunto de funcionalidades da seção fosse representado pelas funcionalidades oferecidas pelo serviço. Assim, após analisar os três aplicativos escolhidos, os seguintes serviços foram identificados:

- **Agenda:** Permite, de maneira geral, gerenciar informações de eventos e compromissos relacionados a universidade, como aulas, provas, trabalhos, palestras, etc.
- **Histórico:** Permite, de maneira geral, a visualização do histórico acadêmico do aluno.
- **Linhas de Ônibus:** Permite, de maneira geral, a visualização das linhas de ônibus que passam pelas unidades da universidade.
- **Mapas:** Permite, de maneira geral, a visualização de mapas com a localização das unidades (prédios, blocos, laboratórios, restaurantes etc.) da universidade.
- **Notícias:** Permite, de maneira geral, a visualização de notícias relacionadas a universidade.
- **Perfil:** Permite, de maneira geral, a visualização do perfil acadêmico do aluno, que contêm matrícula, nome, etc.
- **Plano de Estudos:** Permite, de maneira geral, a visualização do plano de estudos do período corrente.
- **Restaurante Universitário:** Permite, de maneira geral, a visualização dos cardápios oferecidos nos restaurantes universitários.

A tabela 4.1 mostra a relação entre os aplicativos e a presença ou não dos serviços identificados.

	Aplicativo		
Serviço	UFF Mobile	UFPE Mobile	PUC-SP
Agenda	✓	✓	X
Histórico	✓	X	X
Linhas de Ônibus	✓	✓	X
Login	✓	✓	X
Mapas	X	✓	✓
Notícias	✓	✓	✓
Perfil	✓	✓	X
Plano de Estudos	✓	X	X
Restaurante Universitário	✓	✓	✓

Tabela 4.1: Relação entre os aplicativos analisados e a presença ou não dos serviços identificados.

Após a identificação de tais serviços e funcionalidades, foi necessário criar diagramas e modelos que pudessem auxiliar o processo de desenvolvimento do MyUni. Dessa maneira seria mais fácil visualizar e analisar de maneira mais clara as classes e elementos do domínio e suas relações. O modelo utilizado como base para a definição do modelo de objetos foi o modelo conceitual de negócios. Segundo (Cheesman; Daniels, 2001)[25], "um modelo de conceito de negócios é um modelo conceitual. Não é um modelo de software, mas um modelo da informação que existe no domínio do problema. O principal objetivo do diagrama do modelo conceitual de negócios é capturar conceitos e identificar relacionamentos". Ainda segundo Cheesman; Daniels

> "Os modelos conceituais de negócios geralmente capturam classes conceituais e suas associações. Essas associações podem ou não ter suas multiplicidades especificadas. O modelo pode conter atributos, se eles são significativos, mas eles não precisam ser especificados, e as operações não serão usadas. Uma vez que a ênfase do modelo é capturar o conhecimento do domínio, e não sintetizá-lo ou normalizá-lo, raramente será utilizado generalização neste modelo. Simultaneamente, as relações de dependência geralmente não são usadas."(Cheesman; Daniels, 2001)[25]

Assim, após a identificação dos serviços, e das funcionalidades oferecidas por eles, foram criados os modelos conceituais de negócio para cada serviço oferecido por cada aplicativo. Essa análise individual faz-se necessária, pois embora alguns aplicativos ofereçam as mesmas funcionalidades, os dados e informações que eles manipulam possuem algumas diferenças, como será mostrado.

4.1.1 UFF Mobile

Agenda

Uma das funcionalidades do UFF Mobile, permite a visualização de uma lista de provas e trabalhos que devem ser feitos pelo aluno. Todas as informações sobre as provas e trabalhos são criadas e gerenciadas por ele. Para criar uma prova, é necessário inserir um nome, uma descrição e uma data de início. Já para criar um trabalho, é necessário inserir as mesmas informações que devem ser inseridas para uma prova, além de uma data de fim. Por essa funcionalidade ser similar a uma agenda, onde cada prova ou trabalho corresponde a um evento dessa agenda, concluiu-se que as provas e trabalhos seriam associados a uma classe *Agenda*.

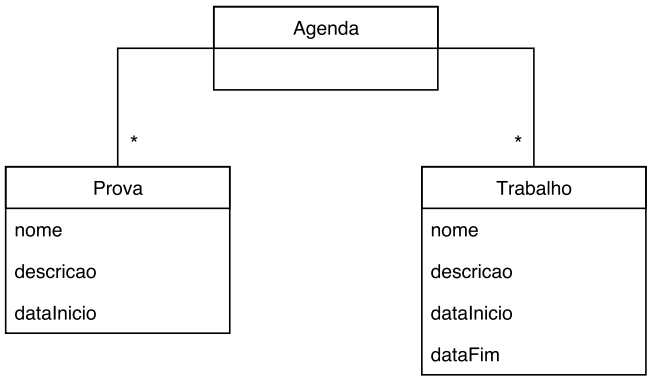

Figura 4.1: Modelo conceitual de negócios para a funcionalidade de agenda do UFF Mobile

Notícias

No UFF Mobile, a funcionalidade de notícias exibe uma lista de itens, onde cada item possui um título, uma data de publicação e um texto com o conteúdo da notícia. Portanto, essa funcionalidade possui apenas uma classe *Noticia*.

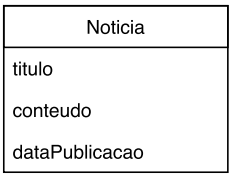

Figura 4.2: Modelo conceitual de negócios para a funcionalidade de notícias no UFF Mobile

Histórico

Na funcionalidade de histórico é possível visualizar informações sobre a carga horária total do curso, a carga horária cursada até o momento, o coeficiente de rendimento do aluno e a situação atual da matrícula dele. Além disso, é mostrada uma lista de itens que contém o nome das disciplinas cursadas pelo aluno, juntamente com a código da turma e o período no qual a disciplina foi cursada. Também são mostrados o status (aprovado, reprovado ou trancado) do aluno na disciplina e a nota final obtida. Analisando

a funcionalidade, chegou-se a conclusão de que o histórico pode ser representado como uma classe que possui um conjunto de itens de histórico associados. Esses itens de histórico são classes que possuem os dados da nota final e uma referencia a turma na qual ele cursou a disciplina.

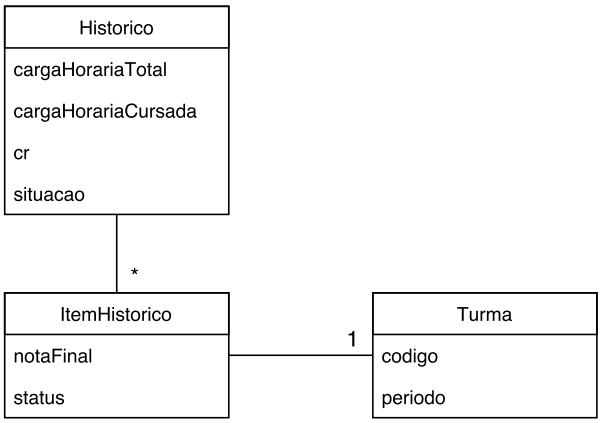

Figura 4.3: Modelo conceitual de negócios para a funcionalidade de histórico do UFF Mobile

Linhas de Ônibus

Nessa funcionalidade, é possível visualizar as linhas de ônibus que circulam pelas unidades da universidade, assim como os ônibus que operam em cada linha e as rotas realizadas por essas linhas. Cada rota de uma linha possui um conjunto de paradas, que representam os locais no qual o ônibus realiza embarque e desembarque. Assim, a funcionalidade foi modelada da seguinte maneira.

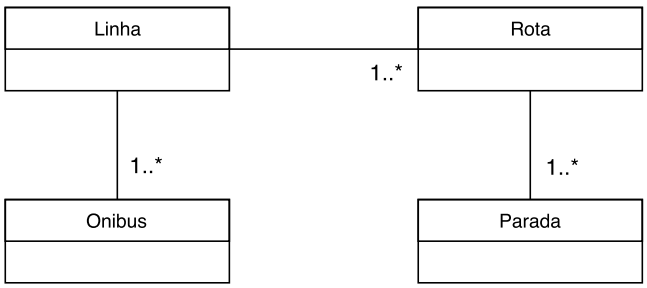

Figura 4.4: Modelo conceitual de negócios para a funcionalidade de linhas de ônibus do UFF Mobile

Login

Para poder utilizar o UFF Mobile e ter acesso as funcionalidades oferecidas por ele, é necessário que o usuário forneça seu CPF e uma senha previamente cadastrada. Juntos, esses dados constituem a credencial de acesso ao sistema. Assim a funcionalidade de login do UFF Mobile pode ser modelada como uma credencial de acesso, que pertence ao aluno usuário do sistema.

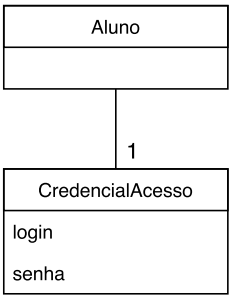

Figura 4.5: Modelo conceitual de negócios para a funcionalidade de login do UFF Mobile

Perfil

Na funcionalidade de perfil, é possível visualizar informações sobre o perfil do aluno no sistema acadêmico, como o seu nome, matricula e curso no qual ele está inscrito. Analisando esses dados, essa funcionalidade foi modelada de maneira que o perfil do aluno fosse representado como uma classe *Aluno*, que é a classe base do perfil e contém os dados pessoais do aluno, como o nome. Associado ao aluno, temos uma classe Matricula, que contém os dados de número de matrícula e o curso no qual ele está inscrito.

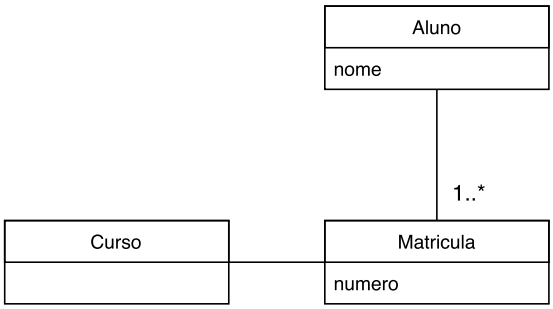

Figura 4.6: Modelo conceitual de negócios para a funcionalidade de perfil do UFF Mobile

Plano de Estudos

No plano de estudos, é possível visualizar todas as aulas que o aluno tem no período corrente. Cada aula do plano de estudos possui informações sobre a turma e informações sobre horário de início e término da aula, assim como informações sobre a sala na qual ela ocorre. A partir dessas informações, chegou-se ao seguinte modelo conceitual.

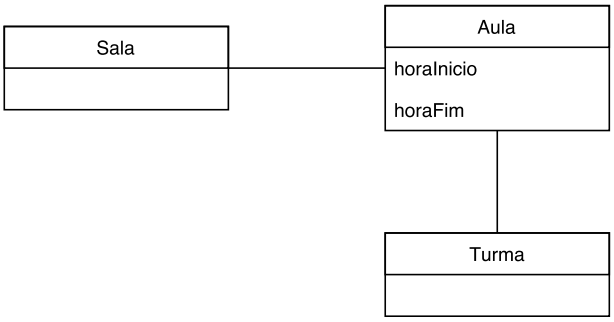

Figura 4.7: Modelo conceitual de negócios para a funcionalidade plano de estudos do UFF Mobile

Restaurante Universitário

Essa funcionalidade permite a visualização dos cardápios oferecidos nos restaurantes universitários. Cada cardápio possui um conjunto de pratos associados a ele, portanto, o modelo conceitual de negócios fica da seguinte maneira.

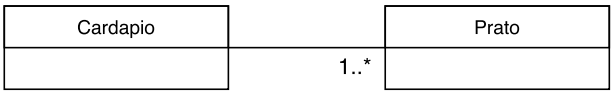

Figura 4.8: Modelo conceitual de negócios para a funcionalidade de restaurante universitário do UFF Mobile

4.1.2 UFPE Mobile

Agenda

No UFPE Mobile, a funcionalidade de agenda permite que o aluno visualize o calendário acadêmico para o ano letivo corrente. O calendário é composto por eventos, que possuem um nome, uma data de início, e uma data de fim. Se abstrairmos a funcionalidades de calendário, podemos dizer que um calendário nada mais é do que uma agenda com diversos eventos. Dessa forma, o calendário poder ser representado como uma classe *Agenda* e os eventos do calendário seriam representados por uma classe *Evento*.

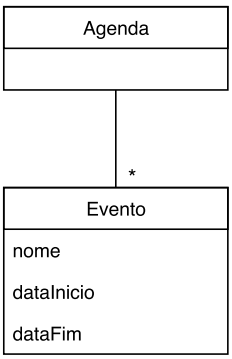

Figura 4.9: Modelo conceitual de negócios para a funcionalidade de notícias do UFPE Mobile

Notícias

Ao analisar a funcionalidade de notícias no UFPE Mobile, viu-se que ela era modelada de maneira similar a funcionalidade de notícias do UFF Mobile. A única diferença, é que no UFPE Mobile, as notícias são agrupadas por campus. Portanto, nesse caso, a classe *Noticia* possui uma associação com o campus a qual a notícia se refere.

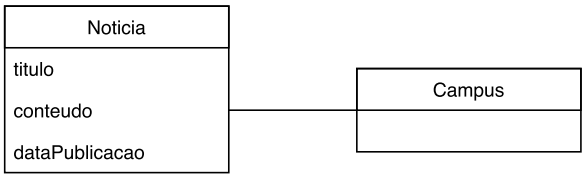

Figura 4.10: Modelo conceitual de negócios da funcionalidade de notícias do UFPE Mobile

Linhas de Ônibus

Essa funcionalidade também possui bastantes similaridades com a funcionalidade encontrada no UFF Mobile. A diferença é que no UFPE Mobile, não são mostrados os dados sobre os ônibus que operam nas linhas. Por esse motivo, o diagrama de classes para essa funcionalidade no UFPE Mobile foi definido como o mesmo diagrama do UFF Mobile, mas sem as associações da classe *Linha* com a classe *Ônibus*.

Login

Assim como no UFF Mobile, para acessar algumas informações restritas aos alunos da universidade, é necessário fornecer um usuário e uma senha, que juntos, constituem uma credencial de acesso ao sistema. Portanto, a modelagem da funcionalidade de login no UFPE é similar a modelagem da funcionalidade de login do UFF Mobile.

Mapas

Na funcionalidade de mapas, é possível que o aluno visualize um mapa com a localização das unidades (*i.e.* centros, departamentos, bibliotecas, restaurantes universitários, etc.) de um determinado campus. Essas unidades podem ser acessadas a partir da interação com o mapa, ou através da pesquisa em uma lista com os nomes de localizações pré-definidas.

Analisando a funcionalidade, chegou-se a conclusão de que cada localização do mapa, é representado como uma classe *Localizacao*, que possui um nome e dados de latitude e longitude. Cada conjunto de localizações pode ser agrupado para constituir um *Mapa* específico para um *Campus*. Assim, o modelo conceitual para essa funcionalidade fica da seguinte forma.

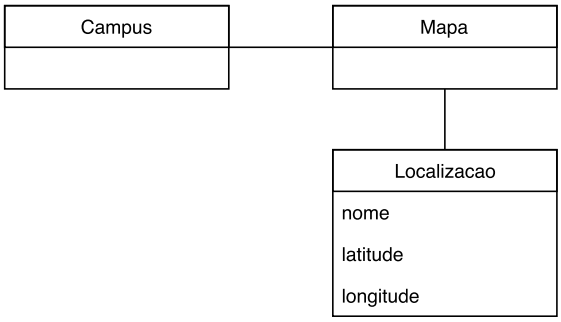

Figura 4.11: Modelo conceitual de negócios para a funcionalidade de mapas do UFPE Mobile

Perfil

Assim como no UFF Mobile, também é possível visualizar informações sobre o perfil do aluno no sistema acadêmico através da utilização de uma credencial de acesso, constituída de um login e uma senha. No entanto, como não foi possível obter credenciais válidas para acessar o sistema e realizar uma análise mais detalhada da funcionalidade, ela foi modelada da mesma maneira que no UFF Mobile.

Restaurante Universitário

Nessa funcionalidade, são mostrados os cardápios oferecidos no restaurante universitário. Cada cardápio é separado por dia da semana e possui um conjunto de três refeições associadas (café da manha, almoço e janta). As refeições possuem um horário no qual elas são servidas e o preço cobrado. Por fim, cada refeição possui um conjunto de pratos que a constitui. Assim, conclui-se que o modelo conceitual para essa funcionalidade é composto de uma classe *Cardapio*, que associado a um conjunto de refeições, que por sua vez possui um conjunto de pratos associados a elas.

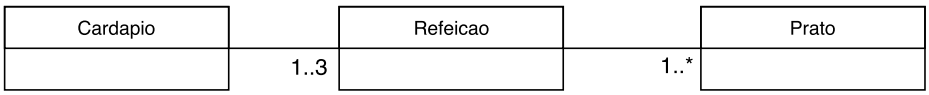

Figura 4.12: Modelo conceitual de negócios para a funcionalidade restaurante universitário do UFPE Mobile

4.1.3 PUC-SP Mobile

Notícias

A funcionalidade de notícias do PUC-SP possui as mesmas definições da funcionalidade de notícias do UFF-Mobile, portanto, suas modelagens são idênticas.

Mapas

A funcionalidade de mapas do PUC-SP possuí as mesmas definições da funcionalidade de mapas do UFPE Mobile, portanto, suas modelagens são idênticas.

Restaurante Universitário

Assim como na funcionalidade de restaurante universitário do UFF Mobile, no PUC-SP, cada cardápio possui um conjunto de pratos associados. Assim, o modelo conceitual para essa funcionalidade é idêntico ao do UFF Mobile.

4.1.4 Modelo Conceitual de Negócio Unificado

Após a criação dos modelos conceituais para cada funcionalidade de cada aplicativo, foi realizada uma unificação dos modelos com o objetivo de indentificar relações e caractarísticas similares entre os modelos. Dessa maneira, seria possível transformar tais similaridades em heranças e abstrações, por exemplo, que poderiam vir a facilitar ainda mais o processo de identificação e criação dos hot-spots e frozen-spots do MyUni.

Para realizar essa unificação foi utilizado o conceito de *viewpoints*. Segundo (Fontoura et al., 2000)[26], "viewpoints nada mais são do que perspectivas dos sistemas que estão sendo analisados e descritos. Essas perspectivas podem ser representadas por artefatos de design orientados a objetos". No caso do MyUni, os viewpoints são representados pelos modelos conceituais de negócio criados a partir da análise das funcionalidades de cada aplicativo. Ainda segundo Fontoura et al., para realizar a unificação dos viewpoints de maneira eficiente, algumas regras devem ser seguidas, como:

- Os mesmos conceitos devem ser representados pelos mesmos nomes. As regras de unificação usam nomes para definir a semântica dos conceitos envolvidos. Assim, se dois *viewpoints* diferentes tiveram diferentes nomes para representar o mesmo conceito, um procedimento de refatoração deve ser usado para renomear um deles. As refatorações podem ser aplicadas a classes, atributos e métodos.
- Os tipos de atributos devem ser consistentes. Os atributos que representam o mesmo conceito (e por esse motivo com o mesmo nome em todos os viewpoints) também devem ter o mesmo tipo.
- Hierarquias cíclicas devem ser evitadas. A unificação de viewpoints não pode gerar um ciclo de herança.

Para facilitar a unificação e aplicação das regras citadas acima, os modelos conceituais foram primeiramente agrupados por funcionalidade. Dessa maneira, seria mais fácil identificar as possíveis heranças e abstrações, uma vez que funcionalidades similares geralmente trabalham sobre o mesmo conjunto de dados. Em alguns casos, não foi possível unificar um modelo com outros, pois as funcionalidades modeladas eram exclusivas do aplicativo, tornando a unificação impossível. Um exemplo, é a funcionalidade de histórico, que é exclusiva do UFF Mobile. Já em outros casos, as funcionalidades eram idênticas e possuíam o mesmo modelo, portanto, sua unificação era desnecessária, como por exemplo, a funcionalidade

de mapa do UFPE Mobile e do PUC-SP, que possuem o mesmo modelo. Assim, para as funcionalidades cujos modelos eram passíveis de unificação, as seguintes unificações foram realizadas.

Agenda

Como visto, a funcionalidade de agenda do UFF Mobile permite que o aluno possa adicionar provas e trabalhos a uma agenda. Já no UFPE Mobile, essa funcionalidade permite que o aluno visualize o calendário acadêmico do ano letivo corrente, que também é composto por eventos. Analisando os modelos conceituais, percebe-se que tanto provas e trabalhos, como os eventos do calendário acadêmico, são associados a uma agenda e possuem minimamente as informações sobre a data de início e a data de fim do evento. Portanto, todas essas classes podem ser tratadas como uma subclasse de uma classe *Evento*. Embora não seja definido pelos aplicativos, foi criado na classe evento, uma relação com uma classe de localização, pois em diversas outras aplicações que oferecem funcionalidade de agenda, é possível adicionar a localização na qual um evento ocorre. Além disso, a adição dessa relação, permitiu que novas abstrações pudessem ser feitas, como será visto mais adiante.

Linhas de Ônibus

Ao analisar os modelos para a funcionalidade de linhas de ônibus do UFF Mobile e do UFPE Mobile, percebemos que ambos definem relações entre as classes *Linha*, *Rota* e *Parada*. Além disso, se analisarmos o conceito da classe *Parada* nos dois aplicativos, podemos chegar a conclusão de que uma *Parada*, nada mais é do que uma subclasse de uma classe *Localizacao*.

Notícias

Para a funcionalidade de notícias, a unificação dos modelos resultou em um modelo idêntico ao definido pela funcionalidade de notícias do UFPE Mobile.

Plano de Estudos

Embora a funcionalidade de planos de estudos seja exclusiva do UFF Mobile, ela é passível de abstração, e como consequência, de uma possível unificação. Olhando o modelo para essa funcionalidade, podemos ver que o plano de estudos é constituído por aulas, que possuem, dentre outros atributos, um horário de início e de fim, e uma sala de aula na qual ela é realizada. Se analisarmos melhor esse modelo e realizarmos algumas abstrações, podemos concluir que um plano de estudos nada mais é do que uma *Agenda* e cada aula desse plano de estudos representa um *Evento*. Além disso, as salas nas quais ocorrem as aulas, nada mais são do que a *Localizacao* na qual elas ocorrem.

Restaurante Universitário

Para a funcionalidade de restaurante universitário, a unificação dos modelos resultou em um modelo idêntico ao definido para o UFPE Mobile. Embora os outros aplicativos não tenham uma classe *Refeicao* associada a cardápios e pratos, foi decidido refatorar os modelos de modo que ela fosse adicionada, pois

julgou-se importante permitir a criação de cardápios separados por refeição.

Após realizar as unificações e refatorações necessárias para cada modelo conceitual de cada funcionalidade, um novo modelo conceitual de negócios unificado foi criado, como pode ser visto na figura 4.13. Percebe-se pelo modelo conceitual de negócios unificado, que diversas abstrações e generalizações acabaram surgindo. Isso acabou facilitando o processo de identificação e criação dos hot-spots, uma vez que foi necessário apenas criar hot-spots que manipulassem tais generalizações e abstrações, o que acabou maximizando o reúso de hot-spots e componentes, e minimizando o número de hot-spots fornecidos, consequentemente, minimizando a complexidade e maximizando a flexibilidade do MyUni.

4.2 Identificação de Hot-spots e Criação de Hot-spot-Cards

A partir da análise das funcionalidades identificadas e do modelo conceitual de negócios unificado, foram definidos e criados os hot-spots do MyUni. Esses hot-spots foram definidos de maneira que eles pudessem ser utilizados para implementar funcionalidades similares as encontradas nos três aplicativos analisados. Além disso, como mencionado anteriormente, os hot-spots também foram definidos de maneira que eles manipulassem principalmente abstrações e generalizações, e não classes concretas e especializações, pois dessa maneira, um mesmo hot-spot poderia ser reutilizado na implementação de diferentes funcionalidades que manipulassem tais abstrações e generalizações.

Uma decisão importante tomada durante a identificação e criação dos hot-spots foi que o MyUni deveria fornecer métodos que permitissem que o usuário do framework pudesse definir a maneira como os dados seriam acessados, manipulados, e como eles seriam apresentados para o usuário final do aplicativo. Portanto, a grande maioria dos hot-spots foram projetados para garantir que, para cada funcionalidade ou conjunto de funcionalidades identificadas, o desenvolvedor pudesse definir sua própria maneira de obter, manipular e apresentar os dados.

A seguir são listados alguns dos hot-spots identificados, juntamente com suas descrições simplificadas.

- **Obter localizações:** Permite a implementação de métodos para a obtenção e manipulação de localizações próximas a um determinado ponto do mapa;
- **Exibir localizações:** Permite a implementação de métodos para a exibição de localizações em um mapa;
- **Obter linhas de ônibus:** Permite a implementação de métodos para a obtenção e manipulação de linhas de ônibus;
- **Exibir linhas de ônibus:** Permite a implementação de métodos para a exibição de informações sobre linhas de ônibus;
- **Obter eventos:** Permite a implementação de métodos para a obtenção e manipulação de eventos e uma agenda;

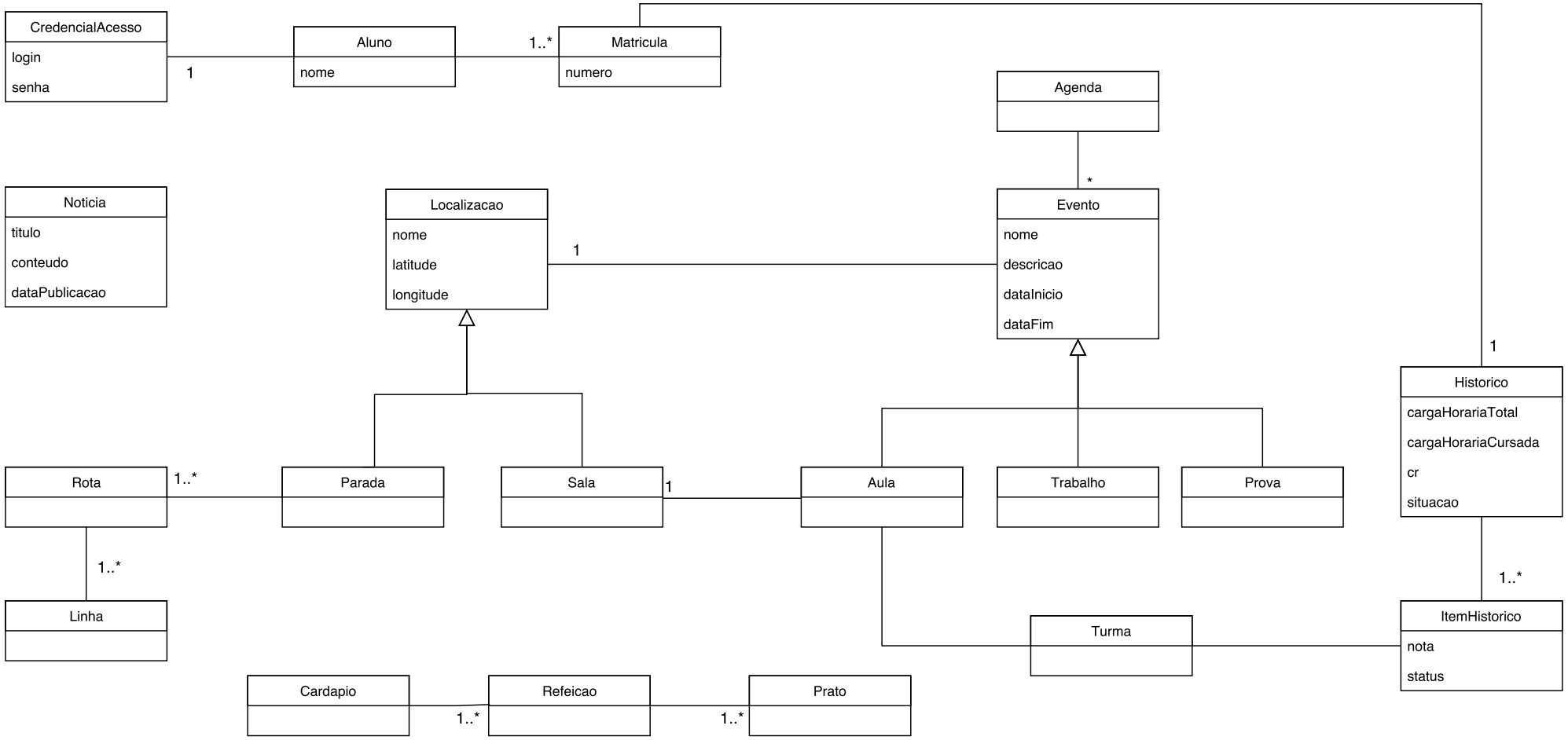

Figura 4.13: Modelo conceitual de negócios unificado

- **Exibir eventos:** Permite a implementação de métodos para a exibição de eventos de uma agenda;
- **Obter histórico acadêmico:** Permite a implementação de métodos para a obtenção e manipulação do histórico acadêmico do aluno;
- **Exibir histórico acadêmico:** Permite a implementação de métodos para a obtenção e manipulação do histórico acadêmico do aluno;
- **Obter perfil:** Permite a implementação de métodos para a obtenção e manipulação do perfil do usuário;
- **Exibir perfil:** Permite a implementação de métodos para a exibição de localizações em um mapa;
- **Obter notícias:** Permite a implementação de métodos para a obtenção e manipulação de notícias;
- **Exibir notícias:** Permite a implementação de métodos para a exibição de notícias;
- **Realizar login:** Permite a implementação de métodos para a realização de login no sistema a partir da validação de credenciais de acesso;
- **Liberar acesso:** Permite a implementação de métodos a serem executados após a validação de credenciais de acesso;

Embora outros hot-spots tenham sido identificados, eles não foram de fato criados e utilizados no MyUni, pois um dos objetivos era manter o framework simples e objetivo, além do fato de que é possível que o programador crie seus próprios componentes e funcionalidades caso eles não sejam fornecidos por padrão. Além da identificação dos hot-spots e criação dos hot-spot-cards, também foram identificados alguns frozen-spots de dados e de funções. Os frozen-spots de funções tinham como objetivo definir as implementações das interfaces fornecidas pelos componentes, garantindo um fluxo pré-definido de interação entre os componentes independente das instâncias criadas com o MyUni, alguns exemplos de frozen-spots de funções são:

- **Obter localizações:** Executa um fluxo pré-definido de operações que obtém dados de localização de um repositório e em seguida os exibe em uma tela;
- **Obter linhas de ônibus:** Executa um fluxo pré-definido de operações que obtém dados de linhas de ônibus de um repositório e em seguida os exibe em uma tela;
- **Obter eventos:** Executa um fluxo pré-definido de operações que obtém dados de eventos de um repositório e em seguida os exibe em uma tela;
- **Obter histórico acadêmico:** Executa um fluxo pré-definido de operações que obtém dados de histórico acadêmico de um repositório e em seguida os exibe em uma tela;
- **Obter perfil:** Executa um fluxo pré-definido de operações que obtém dados de perfil de um repositório e em seguida os exibe em uma tela;

- **Obter notícias:** Executa um fluxo pré-definido de operações que obtém dados de notícias de um repositório e em seguida os exibe em uma tela;
- **Realizar login:** Permite a implementação de métodos para a realização de login no sistema a partir da validação de credenciais de acesso;

Já os frozen-spots de dados, tinham como objetivo definir os dados que não poderiam ser modificados pelas instâncias do framework, de maneira a garantir uma associação padronizada entre as classes, e consequentemente, o correto funcionamento do MyUni. Alguns exemplos de frozen-spots de dados são as classes *Noticia*, *Parada*, *Rota*, *Linha*, *Historico*, *ItemHistorico*, *Sala*, *Aluno* e *Matricula*.

4.3 Agrupamento de Hot-Spot-Cards e Identificação de Componentes

Após a identificação e criação dos hot-spot-cards, foi feita uma análise e agrupamento dos mesmos com o objetivo de criar os group-hot-spot-cards e definir os componentes do framework. Para isso os hot-spot-cards foram separados em grupos lógicos, de maneira que os hot-spots neles contidos, manipulassem as mesmas entidades e procurassem resolver um problema em comum.

É importante lembrar que nas etapas iniciais de criação do MyUni foi realizada uma análise das funcionalidades presentes nos três aplicativos utilizados como base, o que acabou facilitando não só a criação dos hot-spot-cards, mas também dos group-hot-spot-cards, pois as funcionalidades identificadas, de maneira geral, também procuravam solucionar um problema comum, e manipulavam um conjunto similar de dados.

Abaixo, são listados de maneira simplificada alguns dos principais group-hot-spots-cards identificados, juntamente com uma breve descrição dos hot-spot-cards que os constituem:

- **Gerenciar Agenda**: Contém hot-spot-cards de obtenção e exibição de eventos em uma agenda;
- **Gerenciar Localizações**: Contém hot-spot-cards de obtenção e exibição de localizações próximas de um determinado ponto em um mapa;
- **Gerenciar Restaurante Universitário**: Contém hot-spot-cards de obtenção e exibição de cardápios de restaurante universitário;
- **Gerenciar Linhas de Ônibus**: Contém hot-spot-cards de obtenção e exibição de linhas de ônibus;
- **Gerenciar Histórico**: Contém hot-spot-cards de obtenção e exibição do histórico acadêmico;
- **Gerencias Notícias**: Contém hot-spot-cards de obtenção e exibição de notícias;
- **Gerenciar Perfil**: Contém hot-spot-cards de obtenção e exibição de informações de perfil do usuário;
- **Gerenciar Sessão**: Contém hot-spot-cards para a realização de login, logout e validação de acesso;

Dados os group-hot-spot-cards acima, percebe-se que eles, de fato, contêm hot-spot-cards relacionados a solução de problemas de contextos específicos e manipulam conjuntos similares de dados. Portanto, definiu-se que cada group-hot-spot-card corresponderia a um componente, que deveria fornecer métodos para a obtenção, manipulação e apresentação de um conjunto específico de dados. Assim, levando-se em conta a lista de group-spot-cards identificados, foram definidos os seguintes componentes: agenda, localização, restaurante, linhas de ônibus, histórico, notícias, perfil e sessão. A partir da definição de tais componentes, ainda foi necessário especificar o conjunto de interfaces requeridas e providas por cada um deles, seu funcionamento interno, e a maneira como eles deveriam ser criados e gerenciados pelo MyUni e suas instâncias. Esses detalhes serão apresentados com a seguir, na etapa de design e implementação do framework.

4.4 Design e Implementação

Sabe-se que códigos de qualidade são códigos fáceis de ler, evoluir e realizar manutenção. Portanto, o MyUni foi planejado de maneira a seguir as melhores práticas de desenvolvimento, como por exemplo, a utilização dos princípios SOLID [27] a definição de um padrão de comportamento e relacionamento entre os componentes, de modo a facilitar o entendimento e utilização do framework. Para fornecer tais características, foi necessário definir uma arquitetura para o MyUni.

4.4.1 Definição da Arquitetura

Analisando os componentes identificados, percebe-se que a maioria deles possuem hot-spots relacionados a obtenção, manipulação e apresentação de dados. Assim, seguindo o princípio de responsabilidade única, cada componente poderia ser composto por interfaces específicas para cada responsabilidade (obtenção, manipulação, e apresentação). Levando-se em conta esse fato, chegou-se a conclusão que cada componente do MyUni poderia ser criado seguindo os conceitos do padrão de design MVP (Model-View-Presenter) [28], cujo objetivo é exatamente definir elementos com responsabilidades específicas e um fluxo de comunicação e relacionamento entre eles. Desse modo, seguindo os conceitos do MVP, cada componente poderia ser composto pelos seguintes elementos:

- Uma interface Model, que possui métodos referentes aos hot-spots de obtenção e manipulação de dados. O Model foi definido como uma interface, pois para o MyUni, não importa a maneira como os dados são obtidos ou manipulados. Na verdade, o que importa para o MyUni, é que esses dados sigam um determinado padrão de atributos e relacionamentos, de maneira que eles possam ser processados internamente pelo framework. Essa interface é definida como uma interface requerida, pois o componente necessita de um meio de obtenção dos dados que serão utilizados por uma determinada funcionalidade.

- Uma interface View, que possui métodos referentes aos hot-spots de apresentação de dados. A View foi definida como uma interface, pois o MyUni não define um padrão de apresentação dos dados, apenas espera que os aplicativos possam executar um conjunto específico de métodos para a

apresentação de dados. Essa interface é definida como uma interface requerida, pois o componente necessita de uma referência a uma View, para a qual os dados serão enviados para a apresentação, após terem sido obtidos pela implementação da interface Model.

- Uma interface Presenter, que tem o principal objetivo de expor as funcionalidades fornecidas pelo componente, sendo portanto, o meio pelo qual as funcionalidades providas pelo componente são chamadas. O componente que implementar essa interface deve controlar a comunicação e o fluxo de dados entre as implementações da View e do Model que forem fornecidas ao componente.

Ainda como parte do modelo de componentes, também foi identificado a necessidade da existência de um meio de configuração e gerenciamento dos componentes do MyUni. Dessa forma, a criação e comunicação dos componentes poderiam ser abstraídas do usuário, tornando a utilização do MyUni mais fácil. Portanto, foi definido que o MyUni seria composto de um camada de *middleware*, que seria responsável por oferecer serviços para a configuração e gerenciamento dos componentes do framework. A figura 4.14 mostra, de maneira geral, a arquitetura do MyUni.

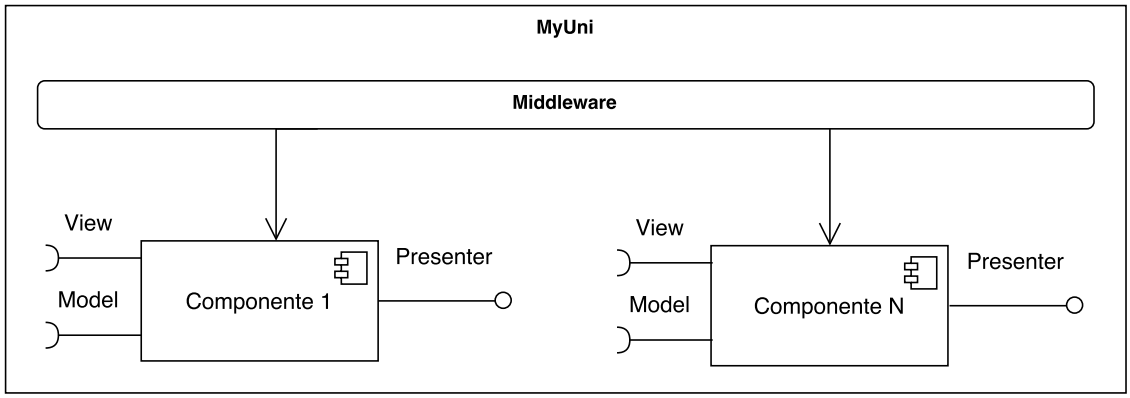

Figura 4.14: Visão geral da arquitetura do MyUni

Por ter sido desenvolvido para a plataforma Android, o MyUni precisou que os elementos que compunham sua arquitetura fossem criados de maneira que eles pudessem ser executados corretamente junto com os elementos do framework de desenvolvimento do Android. Tal adequação implicou na adição de novas classes, relacionamentos e configurações fornecidas pelo framework do Android, e que são necessárias para o correto funcionamento de qualquer código que venha a ser executado na plataforma.

Uma das principais adaptações foi fazer com que o MyUni tivesse que estender a classe *Application*, que de acordo com a documentação oficial do framework Android [29] é a classe responsável por manter o estado global de qualquer aplicação que seja executada na plataforma Android. Por conta disso, ela é a base para qualquer aplicativo, sendo inclusive instanciada antes de qualquer outra classe quando um processo de um aplicativo é inciado no Android. Assim, como uma das principais caraterísticas dos frameworks é a inversão de dependência, onde o framework é o principal responsável por controlar os códigos implementados pelo desenvolvedor, foi necessário que o MyUni estendesse a classe *Application*, para que fosse possível controlar o estado de suas instâncias, e fornecer um ponto de acesso para os

serviços básicos oferecidos pelo MyUni, como a configuração e gerenciamento dos componentes.

A existência da classe *Activity*, que também é uma classe muito importante do framework Android, acabou refletindo na maneira como o MyUni foi projetado. Segundo a documentação oficial [30], uma *Activity* é "uma coisa única e focada que o usuário pode fazer. Quase todas as Activities interagem com o usuário, portanto, a classe de Activity cuida da criação de uma tela na qual você pode colocar sua UI". Além disso, toda classe *Activity* possui uma referência para a classe *Application*. Assim, caso um usuário precise interagir diretamente com um aplicativo para executar uma determinada funcionalidade, é necessário que essa interação e execução seja realizada via uma classe *Activity*. Como o MyUni fornece componentes que possuem interfaces para a apresentação de dados (View) e interfaces que permitem a execução das funcionalidades oferecidas (Presenter) pelos componentes, isso significa que a *Activity* deve implementar a interface *View* e deve ser responsável por requisitar e acessar as funcionalidades dos componentes via sua interface *Presenter*.

A figura 4.15 mostra de maneira simplificada, através de um diagrama de classes baseado na UML, como as classes do framework Android se integram com os elementos do MyUni.

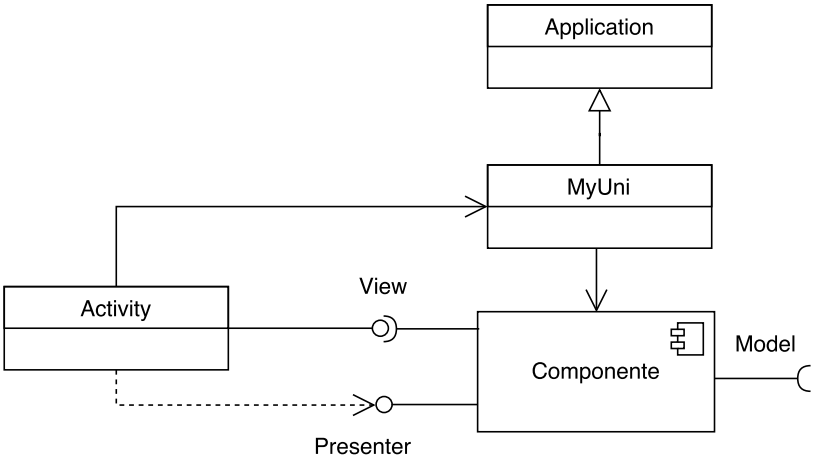

Figura 4.15: Relação entre as classes do framework Android e o MyUni.

4.4.2 Especificação de Componentes

Com base na arquitetura do MyUni e suas definições, foram realizadas as especificações dos componentes do MyUni. Para facilitar o processo de especificação dos componentes, foi utilizada a UML-F, que é própria para a modelagem de frameworks. É importante dizer que, durante a especificação, foi definido que os métodos correspondentes aos hot-spots identificados seriam escritos em inglês. No entanto, as entidades tiveram seus nomes mantidos. Por exemplo, o hot-spot *obter eventos* do componente de eventos, foi definido como um método *loadEventos()*.

A seguir, serão mostradas as especificações simplificadas de dois do principais componentes do MyUni, o componente de agenda e o componente de localização. Também será mostrado brevemente como esses componentes podem ser utilizados para criar instâncias do MyUni. Por possuir uma arquitetura bem definida, os outros componentes do MyUni seguem as mesmas lógicas de especificação que serão apresentadas.

Componente de Agenda

O componente de agenda, como definido nas etapas anteriores, deveria possuir interfaces que permitissem a implementação de funcionalidades de agenda, através do fornecimento de métodos de obtenção, manipulação, e visualização de eventos. Portanto, a principal entidade do componente é a classe *Evento*. Como visto na etapa de unificação dos modelos conceituais de negócio, qualquer entidade que possa ser representada como um evento pode ser definida como uma subclasse de *Evento*, como as classes *Aula*, *Prova* e *Trabalho*, que também possuem os mesmos atributos de evento, mas podem ter atributos adicionais. A figura 4.16 mostra a hierarquia da classe *Evento*.

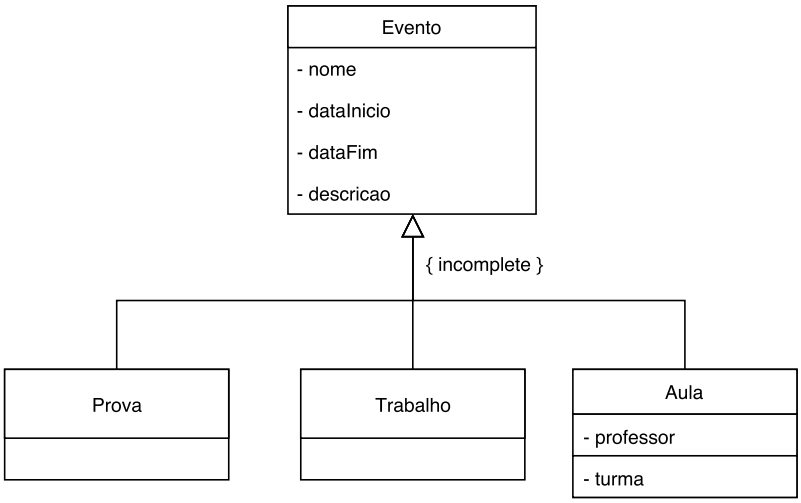

Figura 4.16: Hierarquia da classe *Evento*.

Para definir as interfaces providas e requeridas pelo componente de agenda, os métodos identificados foram agrupados de acordo com sua responsabilidade, seguindo a arquitetura definida pelo MyUni. Durante a especificação do componente, foi definido que suas interfaces poderiam ser estendidas para que os usuários do MyUni pudessem criar novos componentes com funcionalidades extras, de acordo com suas necessidades. Assim, a interface *View* do componente, ficou com os métodos relacionados a exibição dos eventos (*showEventos()*) e a interface *Model*, ficou com os métodos relacionados a obtenção dos dados de eventos (*loadEventos()*).

Por fim, como definido, a interface *Presenter* ficou com os métodos que permitem a execução das operações oferecidas pelo componente. É importante dizer que, embora o nome dos métodos do *Presenter* sejam iguais aos do *Model*, suas implementações são completamente diferentes. Também é importante relembrar que a interface *Presenter* deve ser implementada pelo componente de agenda fornecido pelo MyUni, e seu comportamento não pode ser modificado diretamente pelo usuário do framework, ou seja, a implementação do componente constitui um frozen-spot. No entanto, isso não impede que o desenvolvedor crie uma nova implementação de um componente de agenda. A figura 4.17, mostra a especificação do componente de Agenda do MyUni.

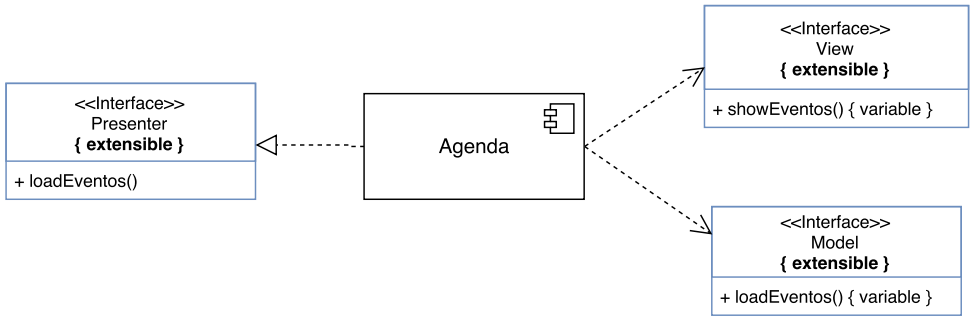

Figura 4.17: Especificação UML-F para o componente de agenda.

Na figura 4.18 é mostrado um diagrama de sequência referente ao funcionamento do componente de agenda. Nele, é mostrado como o componente atua para fornecer a funcionalidade de exibição de eventos em uma determinada instância do MyUni. Na imagem, é mostrado que o fluxo de exibição de eventos inicia quando um usuário interage com uma implementação da interface *View*, que normalmente é uma classe *Activity*. Ao interagir com a *Activity*, a aplicação deve requisitar ao componente de agenda, que implementa a interface *Presenter*, a execução do método *loadEventos()*. Ao receber a chamada, o componente automaticamente inicia o processo de execução da funcionalidade de exibição de eventos, requisitando os eventos para o *Model*. Ao obter a lista de eventos do Model, o componente envia a lista obtida para a *View*, chamando o método *showEventos()*.

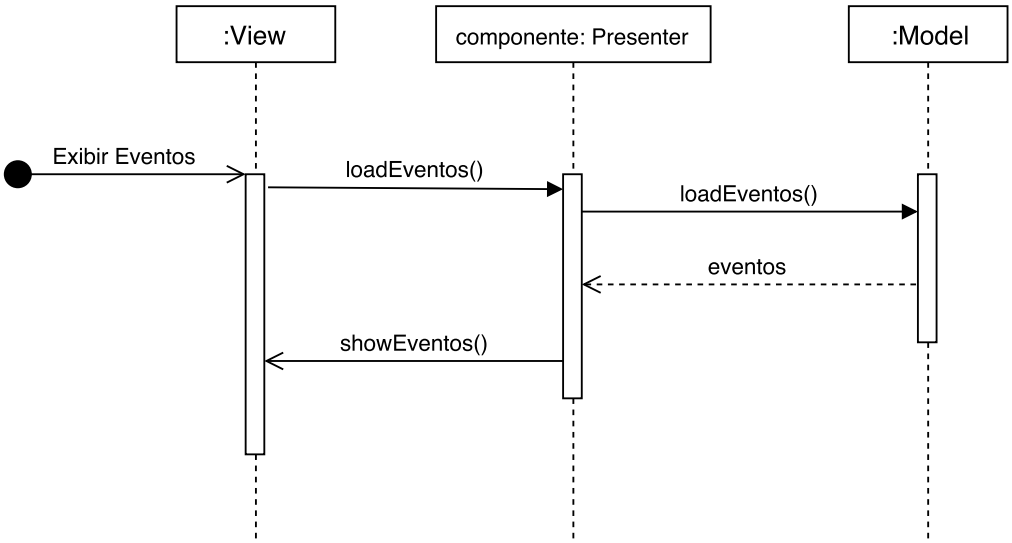

Figura 4.18: Diagrama de sequência para a exibição de eventos pelo componente de agenda.

Assim, dada a especificação do componente de agenda, percebe-se que é possível implementar, por exemplo, a funcionalidade de plano de estudos. Isso é possível, pois foi definido que a classe *Aula* é uma subclasse da classe *Evento*. Assim, para implementar a funcionalidade de plano de estudos numa instância do MyUni, basta que o desenvolvedor utilize o componente de agenda e implemente e forneça as interfaces *View*, para exibir o plano de estudos, e *Model*, para obter as aulas do plano de estudos.

Componente de Localização

O componente de localização, como definido nas etapas anteriores, deveria possuir interfaces que permitissem a implementação de funcionalidades de localização, através do fornecimento de métodos de exibição de mapas de localizações e rotas entre as localizações. Portanto, a principal entidade do componente, é a classe *Localizacao*. Como visto na etapa de unificação dos modelos conceituais de negócio, qualquer entidade que possa ser representada como uma localização, pode ser definida como uma subclasse de *Localizacao*, como as classes *Parada* e *Sala*, que são fornecidas por padrão pelo MyUni, e possuem os mesmos atributos da classe *Localizacao*, embora possam ter atributos adicionais. Na figura 4.19, é mostrada a especificação da classe localização e suas subclasses fornecidas.

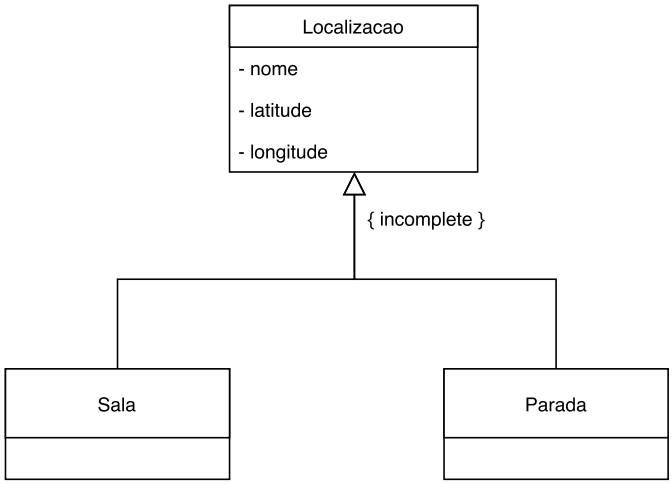

Figura 4.19: Hierarquia da classe *Localizacao*.

Assim como no componente de agenda, para especificar as interfaces providas e requeridas pelo componente de localização, os métodos identificados foram agrupados de acordo com sua responsabilidade, seguindo o padrão de arquitetura do MyUni. Além disso, eles também foram definidos como extensíveis para permitir que os desenvolvedores pudessem criar novos componentes a partir de tais interfaces. Assim, a interface *View* ficou com os métodos relacionados a exibição de localizações (*showLocalizacoes()*), e a interface *Model* ficou com os métodos relacionados a obtenção das localizacões (*loadLocalizacoes()*)).

Por fim, como definido, o *Presenter* ficou com os métodos que permitem a execução das operações oferecidas pelo componente. Assim como no componente de agenda, embora os nomes dos métodos do *Presenter* e *Model* sejam iguais, eles possuem implementações distintas. Além disso, a implementação concreta do componente de localizações também implementa a interface *Presenter* de localização e seu comportamento também não pode ser modificado, sendo um frozen-spot. A figura 4.20 mostra a especificação do componente de localização.

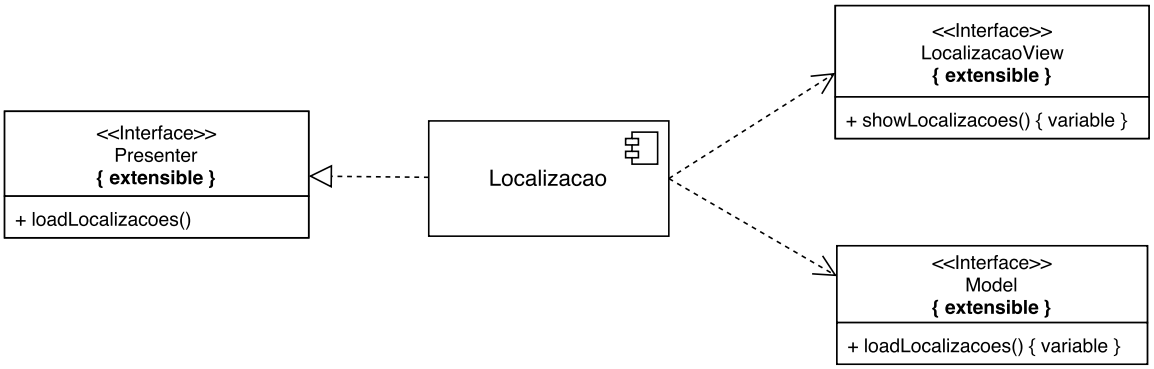

Figura 4.20: Especificação UML-F do componente de localização.

Assim, dada a especificação do componente de localização, percebe-se que é possível implementar, por exemplo, uma funcionalidade de exibição de mapa de salas. Isso é possível, pois foi definido que a classe *Sala* é uma subclasse da classe *Localizacao*. Portanto, para implementar uma funcionalidade de mapa de salas numa instância do MyUni, basta que o desenvolvedor utilize o componente de localizações e forneça uma implementação da interface *Model*, para obter a lista de salas, e uma implementação da interface *View*, para exibir um mapa com as salas obtidas.

Na figura 4.21 é mostrado um diagrama de sequência referente ao funcionamento do componente de localização quando utilizado para fornecer uma funcionalidade de exibição de localizações em uma instância do MyUni. No caso, é mostrado um exemplo para uma funcionalidade de exibição de mapa de salas. Na figura, é mostrado que o fluxo de exibição do mapa de salas inicia quando um usuário interage com uma implementação da interface *View* com o objetivo de exibir um mapa de salas. Ao interagir com a *View*, a aplicação deve requisitar ao componente de localizações, através da interface *Presenter*, a execução do método *loadLocalizacoes()*. Ao receber a chamada, o componente inicia o processo de execução da funcionalidade de exibição de localizacoes, requisitando para uma implementação do *Model* um conjunto de localizações. Ao obter uma resposta do *Model* contendo as salas, o componente de localizações envia a lista de salas obtidas para a *View*, para que elas possam ser exibidas através da chamada do método *showLocalizacoes()*.

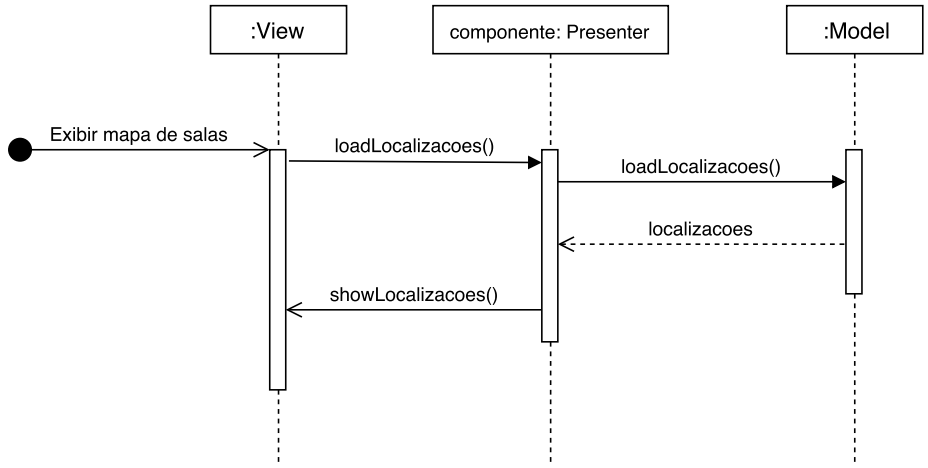

Figura 4.21: Diagrama de sequência para a exibição de salas pelo componente de localização.

A partir das especificações desses e de outros componentes, foi realizada a etapa de implementação, que será mostrada a seguir.

4.4.3 Implementação

Como definido, o MyUni foi desenvolvido para que pudesse ser utilizado na plataforma Android, e por conta disso, ele foi implementado utilizando a linguagem de programação Kotlin [31], que é uma das linguagens oficiais, juntamente com Java, para o desenvolvimento de aplicativos Android. Essa linguagem foi escolhida pois além de ser interoperável com código Java e ter uma sintaxe mais simples, ela possui diversos recursos não suportados pelo Java [32] e que facilitam o desenvolvimento de aplicações, como *null safety*, *smart cast*, *extension functions*, *propriedadades* etc.

Para realizar a implementação foi utilizada a IDE Android Studio 3.0 [33], que é a IDE oficial e recomendada pela Google para o desenvolvimento de aplicativos Android. O Android Studio possui diversas ferramentas que auxiliam o desenvolvimento de aplicativos, como templates de códigos, ferramentas de análise de performance, suporte nativo ao Kotlin, suporte ao Gradle, editor de temas, editor de layouts, integração com serviços da Google (*e.g.* AdMob[34], Firebase[35] e Google Cloud[36]), etc.

Com o objetivo de facilitar a implementação do MyUni, também foram utilizadas algumas bibliotecas. Para auxiliar o processo de injeção de dependências no framework, foi utilizada a bilioteca Dagger 2 [37]. Já para facilitar a execução de métodos de maneira assíncrona, foi utilizada a biblioteca RxJava [38].

Por se tratar de um framework, o MyUni foi implementado como um *módulo de biblioteca* [39]. Foi necessário implementá-lo dessa maneira pois o MyUni não é uma aplicação completa, mas sim um conjunto de classes e interfaces que devem ser integradas a outros aplicativos.

A estrutura do código foi feita de maneira que cada componente oferecido pelo MyUni tivesse seu próprio pacote contendo todas as classes e interfaces referentes a ele. É importante dizer que, independente do pacote, suas classes e interfaces foram definidas de maneira a terem responsabilidades específicas, além de uma nomenclatura similar. Isso permitiu que os pacotes, independente do componente ao qual se

referiam, possuíssem elementos similares, tornando o código consistente e mais fácil de entender. A figura 4.22 mostra a estrutura de pacotes do MyUni.

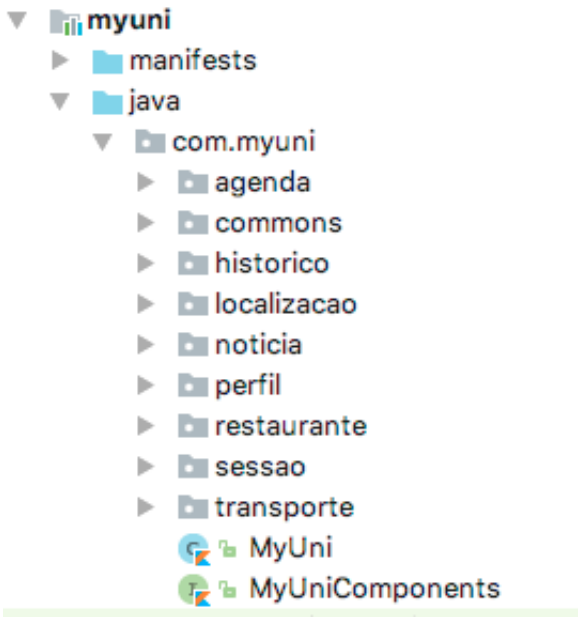

Figura 4.22: Estrutura de pacotes do MyUni.

Para mostrar alguns detalhes mais específicos de implementação, será utilizado como exemplo o componente de agenda do MyUni. A figura 4.23 mostra os detalhes do pacote referente ao componente de agenda.

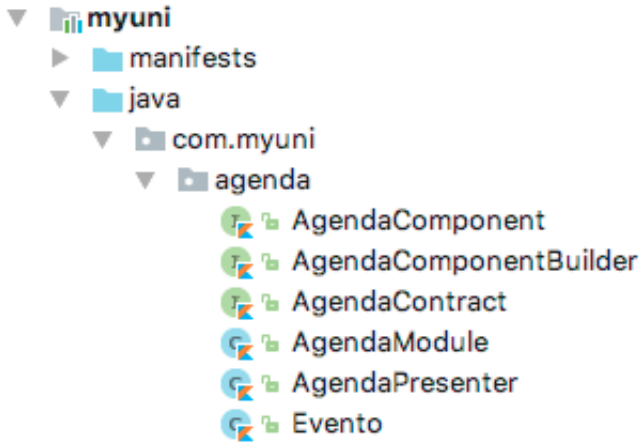

Figura 4.23: Conteúdo do pacote de agenda do MyUni.

Um dos principais elementos de um pacote é a interface que possui o sufixo *Contract*. Essa classe é responsável por reunir todas as interfaces requeridas e fornecidas por um componente. No caso do componente de agenda, essa interface é representada pela interface *AgendaContract*. É possível ver o código da interface em 4.1.

Código 4.1: Interface AgendaContract do componente de agenda.

```
interface AgendaContract {

    interface View: BaseView {

        fun showEventos(eventos: List<Evento>)
    }

    interface Presenter: BasePresenter<View> {

        fun loadEventos(dataInicio: LocalDateTime, dataFim: LocalDateTime)
    }

    interface Model {

        fun loadEventos(dataInicio: LocalDateTime, dataFim: LocalDateTime): Observable<List<Evento>>

    }
}
```

A classe *AgendaPresenter* representa a implementação da interface *Presenter* do componente de agenda. Portanto, é essa classe quem de fato executa os métodos fornecidos pelo componente de agenda. Abaixo, é possível ver o código da classe.

Código 4.2: Implementação da interface Presenter do componente de agenda.

```

// Ao instanciar o Presenter é necessário que o usuário forneça
// uma implementação da interface Model
class AgendaPresenter(private val model: AgendaContract.Model): AgendaContract.Presenter {

    // Referência a implementação da interface View criada pelo desenvolvedor
    lateinit var view: AgendaContract.View

    // Associa a implementação da View a esse Presenter
    override fun attachView(view: AgendaContract.View) {

        this.view = view
    }

    // Implementação do método de obter eventos da interface Presenter
    override fun loadEventos(dataInicio: LocalDateTime, dataFim: LocalDateTime) {

        // Carrega os eventos a partir do Model fornecido pelo desenvolvedor.
        // O código é executado de maneira assíncrona e em seguida os eventos
        // retornados são enviados para serem exibidos pela implementação da View fornecida
        model.loadEventos(dataInicio, dataFim)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribeOn(Schedulers.io())
                .subscribe({ eventos -> view.showEventos(eventos) })
    }
}
```

A classe *AgendaModule* representa um módulo do Dagger 2. O código da classe pode ser visto em 4.3. Esse módulo é responsável por instanciar e fornecer as dependências necessárias para o correto funcionamento do componente de agenda, ou seja, ele fornece métodos para acessar e criar implemen-

tações das interfaces *Presenter* e *Model*. Note que, apesar do componente também necessitar de uma implementação da interface *View*, ela não é fornecida diretamente pelo módulo, pois essa dependência é definida diretamente pelo desenvolvedor durante a instanciação do MyUni, como será visto na seção 4.5.

Código 4.3: Implementação da classe AgendaModule do componente de agenda.

```
1
2  // Módulo contendo métodos que fornecem as implementações necessárias
3  // para o correto funcionamento do componente de agenda.
4  @Module
5  class AgendaModule(private val model: AgendaContract.Model) {
6
7      // Fornece uma implementação do Model.
8      // Essa implementação deve ser fornecida pelo desenvolvedor durante a criação desse módulo.
9      @Provides
10     fun providesModel(): AgendaContract.Model {
11
12         return model
13     }
14
15     // Fornece uma implementação do Presenter.
16     // Essa implementação é fornecida pelo próprio framework.
17     @Provides
18     fun providesPresenter(model: AgendaContract.Model): AgendaContract.Presenter {
19
20         return AgendaPresenter(model)
21     }
22 }
```

A interface *AgendaComponent* define um componente do Dagger 2. Esse componente é responsável por criar e retornar uma implementação do *Presenter*. Para isso, esse componente necessita de um módulo que forneça todas as dependências necessárias para criar um *Presenter*. No caso, esse módulo é o *AgendaModule*. Vale lembrar que esse componente é utilizado apenas para configurar as dependências internas do MyUni utilizando Dagger 2, e portanto, não é utilizado diretamente pelos desenvolvedores para criar um componente de agenda. O código da interface *AgendaComponent* pode ser visto em 4.4.

Código 4.4: Implementação do AgendaComponent.

```
1
2  // Interface que define um componente Dagger que fornece um Presenter.
3  // Esse componente precisa de um módulo que forneça as dependências
4  // necessárias para instanciar um Presenter.
5  @Subcomponent(modules = [(AgendaModule::class)])
6  interface AgendaComponent {
7
8      // Método que fornece a implementação do Presenter do componente de agenda
9      fun presenter(): AgendaContract.Presenter
10 }
```

Como visto, a interface *AgendaComponent* não deve ser utilizada pelos desenvolvedores para criar e acessar o componente de agenda. Para isso, é necessário utilizar o método *buildAgendaComponent()* da interface *AgendaComponentBuilder*, que é implementada pelo *middleware* (classe *MyUni*). Portanto, para obter uma instância do componente de agenda , que é representado pela interface *Presenter*, é

necessário acessar a classe *MyUni* e executar o método *buildAgendaComponent()*. O código da interface *AgendaComponentBuilder* pode ser visto em 4.5.

Código 4.5: Implementação da interface AgendaComponentBuilder do componente de agenda.

```
1
2 // Interface contendo o método que cria e retorna o componente de agenda.
3 interface AgendaComponentBuilder {
4
5     fun buildAgendaComponent(model: AgendaContract.Model): AgendaContract.Presenter
6 }
```

Por fim, a implementação do *middleware*, que é representado pela classe *MyUni* e é responsável por fornecer os métodos para a utilização dos componentes do MyUni, pode ser vista em 4.6.

Código 4.6: Implementação da classe MyUni (middleware).

```
1
2 // Middleware responsável por fornecer os métodos para a utilização dos componentes do MyUni.
3 // Para cada componente fornecido, o middleware deve implementar
4 // a interface Builder de cada componente.
5 open class MyUni : Application(), HistoricoComponentBuilder, SessaoComponentBuilder,
6 AgendaComponentBuilder {
7
8     lateinit var componentsBuilder: MyUniComponents
9
10    // Cria um componente Dagger raiz, responsável por auxiliar a criação
11    // de todos os outros componentes do MyUni.
12    override fun onCreate() {
13        super.onCreate()
14
15        componentsBuilder = DaggerMyUniComponents.builder().build()
16    }
17
18    // Cria um componente de agenda e retorna sua interface Presenter.
19    // Ao executar esse método é necessário que o desenvolvedor forneça
20    // uma implementação da interface Model para o componente
21    override fun buildAgendaComponent(model: AgendaContract.Model): AgendaContract.Presenter {
22
23        return componentsBuilder.buildAgendaComponent(AgendaModule(model)).presenter()
24    }
25
26    /*... */
27 }
```

O código fonte completo do MyUni pode ser acessado em [40]. Na seção a seguir, serão apresentados exemplos de como utilizar o MyUni na prática para criar diferentes aplicativos para a gerência de informações acadêmicas.

4.5 Utilização e Teste

Para verificar se um framework realmente cumpre seu propósito de permitir a criação de diferentes aplicações de um mesmo domínio, é recomendado que sejam criadas aplicações concretas - também conhecidas

como instâncias - que o utilizem. Ao se criar instâncias do framework, é possível verificar se ele realmente atende os requisitos de reúso e flexibilidade que ele propõe.

Portanto, para validar o MyUni, foram criados três aplicativos a partir dele. Para validar os requisitos de flexibilidade e reúso, cada aplicativo foi desenvolvido de maneira a fornecer um conjunto de funcionalidades similares e um conjunto de funcionalidades exclusivas. Para tornar o código das instâncias mais simples, também foi utilizada a biblioteca Dagger 2 para auxiliar o processo de injeção de dependências nas classes.

A seguir serão mostrados os detalhes de como cada aplicativo foi criado utilizando o MyUni.

4.5.1 Instância Um

A primeira instância do MyUni foi criada de maneira a fornecer um aplicativo muito simples para acesso de informações universitárias pessoais de um aluno, como plano de estudos e histórico. Além disso, para garantir a segurança das informações do aluno, também foi utilizada uma funcionalidade de login via credenciais de acesso. A figura 4.24 mostra as funcionalidades propostas para esse aplicativo.

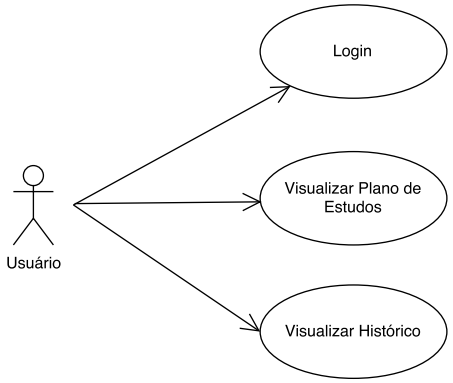

Figura 4.24: Diagrama de casos de uso para a instância um.

A funcionalidade de histórico foi planejada de maneira que o usuário pudesse acessar o histórico acadêmico, que deveria exibir uma lista de todas disciplinas que ele cursou, juntamente com as notas finais e a situação final (aprovado ou reprovado). Já a funcionalidade de plano de estudos foi planejada de maneira que o usuário pudesse visualizar o plano de estudos do período atual que ele está cursando, mostrando para isso, uma grade com as aulas que ele tem em cada dia da semana. Por fim, a funcionalidade de login foi planejada de maneira que, para acessar o plano de estudos e o histórico, o aluno tivesse que fornecer um login e uma senha, que deveriam ser validados para liberar o acesso. Para implementar essas funcionalidades, foram utilizados os seguintes componentes fornecidos pelo MyUni: histórico, eventos e sessão.

O componente de histórico foi utilizado pois ele fornece todo o fluxo de obtenção e exibição do histórico, ficando a cargo do desenvolvedor implementar apenas as interfaces necessárias. Assim, para fornecer tal funcionalidade, foi necessário implementar as interfaces *View* e *Model* do componente de histórico. Em seguida, foi necessário criar uma instância do componente de histórico e fornecer as

implementações da *View* e do *Model* para o mesmo, para que depois o desenvolvedor pudesse chamar os métodos fornecidos pelo componente através da interface *Presenter*.

É importante dizer que esse pequeno fluxo de criação das implementações, fornecimento das dependências, e criação e chamada do componente de histórico, também é seguido para os outros componentes, e por conta disso em todas as instâncias do MyUni, esses fluxos se repetem, independente dos componentes que ela esteja usando.

A interface *View* foi implementada pela classe *HistoricoActivity*, como mostra o código 4.7.

Código 4.7: Implementação da interface View pela classe HistoricoActivity.

```
1
2  class HistoricoActivity : AppCompatActivity(), HistoricoContract.View {
3
4      //Inicialização do layout
5      private val historicoList by lazy { findViewById<RecyclerView>(R.id.historicoList) }
6      private val historicoListAdapter by lazy { HistoricoListAdapter(this, null) }
7      private val cr by lazy { findViewById<TextView>(R.id.cr) }
8
9      //Variável que receberá a implementação da interface Presenter do componente de histórico.
10     private lateinit var historicoPresenter: HistoricoContract.Presenter
11
12     //Injeta a implementação da interface Model específica para essa instância.
13     //A injeção é feita via Dagger 2.
14     @Inject lateinit var historicoModel: HistoricoContract.Model
15
16     override fun onCreate(savedInstanceState: Bundle?) {
17
18         /*
19         Injeta a implementação da interface Model na variável historicoModel.
20         Esse método é específico da aplicação, que é representada pela classe CustomApp,
21         que é uma subclasse da classe MyUni.
22         */
23         (application as CustomApp).inject(this)
24
25         /*
26         Inicialização do componente de histórico via classe CustomApp.
27         Note que para criar o componente, é necessário fornecer uma implementação
28         da interface Model, que nesse caso, é o objeto historicoModel.
29         */
30         historicoPresenter = (application as CustomApp).buildHistoricoComponent(historicoModel)
31
32         super.onCreate(savedInstanceState)
33         setContentView(R.layout.activity_historico)
34
35         historicoList.adapter = historicoListAdapter
36
37         // Associa a View ao componente de histórico e solicita o carregamento do histórico e do CR.
38         presenter.attachView(this)
39         presenter.loadHistorico()
40         presenter.loadCr()
41     }
42
43     override fun showHistorico(historico: Historico) {
44
```

```
45          /*
46          Atribui os itens de histórico ao ListAdapter que exibirá o histórico
47          e notifica a Activity para realizar um refresh na lista.
48          */
49          historicoListAdapter.put(historico.itens)
50          historicoListAdapter.notifyDataSetChange()
51      }
52
53      override fun showCr(cr: Float) {
54
55          cr.text = cr.toString()
56      }
57  }
```

Já a interface *Model*, como visto, foi implementada pela classe *HistoricoRepository*, que representa um repositório remoto, de onde serão obtidos os dados de histórico e o coeficiente de rendimento do aluno. No código 4.8 é possível ver a implementação da interface Model.

Código 4.8: Implementação da interface Model pela classe HistoricoRepository.

```
1
2  class HistoricoRepository @Inject
3  constructor(private val restService: HistoricoRestService): HistoricoContract.Model {
4
5   override fun loadHistorico(): Observable<Historico> {
6
7          /*
8          Retorna um Observable, que realiza uma chamada em background e retorna o histórico
9          assim que ele for assinado por um Subscriber, que nesse caso, é a implementação
10         do componente de histórico.
11         */
12         return restService.loadHistorico()
13     }
14
15     override fun loadCr(): Observable<Float> {
16
17    /*
18         Retorna um Observable, que realiza uma chamada em background e retorna o cr, assim que for
19         assinado por um Subscriber, que nesse caso, é a implementação do componente de histórico.
20         */
21         return restService.loadCr()
22     }
23 }
```

Por fim, em 4.9, é mostrada a implementação da interface *Presenter*, que é feita pela classe *HistoricoPresenter*. Como visto, essa classe constitui um frozen-spot do framework, portanto, não é possível alterar seu comportamento, apenas criar uma instância dela a partir da chamada do método *buildHistoricoComponent()* da classe *MyUni*.

Código 4.9: Implementação da interface Presenter pela classe HistoricoPresenter.

```
1
2  class HistoricoPresenter(private val model: HistoricoContract.Model): HistoricoContract.Presenter {
3
4      lateinit var view: View
```

```
5
6       override fun attachView(view: HistoricoContract.View) {
7
8           this.view = view
9       }
10
11      override fun loadHistorico() {
12
13          model.loadHistorico()
14                  .observeOn(AndroidSchedulers.mainThread())
15                  .subscribeOn(Schedulers.io())
16                  .subscribe { historico -> view.showHistorico(historico) }
17      }
18
19      override fun loadCr() {
20
21          model.loadCr()
22                  .observeOn(AndroidSchedulers.mainThread())
23                  .subscribeOn(Schedulers.io())
24                  .subscribe { cr -> view.showCr(cr) }
25      }
26  }
```

Na figura 4.25 é possível ver, de maneira geral, como as implementações se relacionam nessa instância.

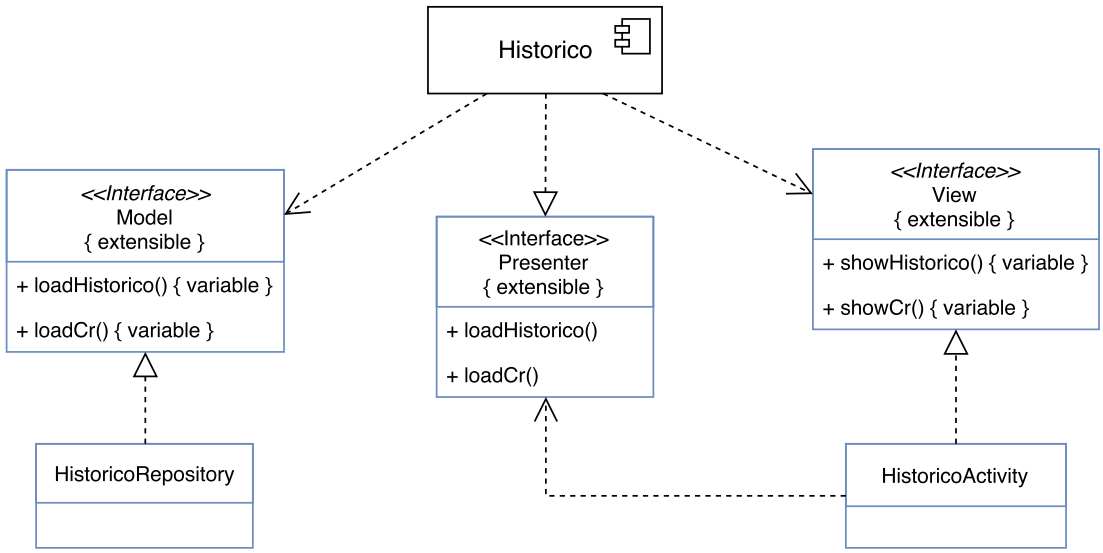

Figura 4.25: Utilização do componente de histórico na instância um.

O componente de eventos foi utilizado para implementar a funcionalidade de plano de estudos. Assim, para fornecer tal funcionalidade, foi necessário implementar as interfaces *View* e *Model* exigidas pelo componente de eventos. Como mostra a figura 4.26, a interface *View* foi implementada pela classe *PlanoEstudosActivity*. Já a interface *Model* foi implementada pela classe *PlanoEstudosRepository*, que representa um repositório remoto de onde serão obtidas e manipuladas as aulas do plano de estudos do aluno.

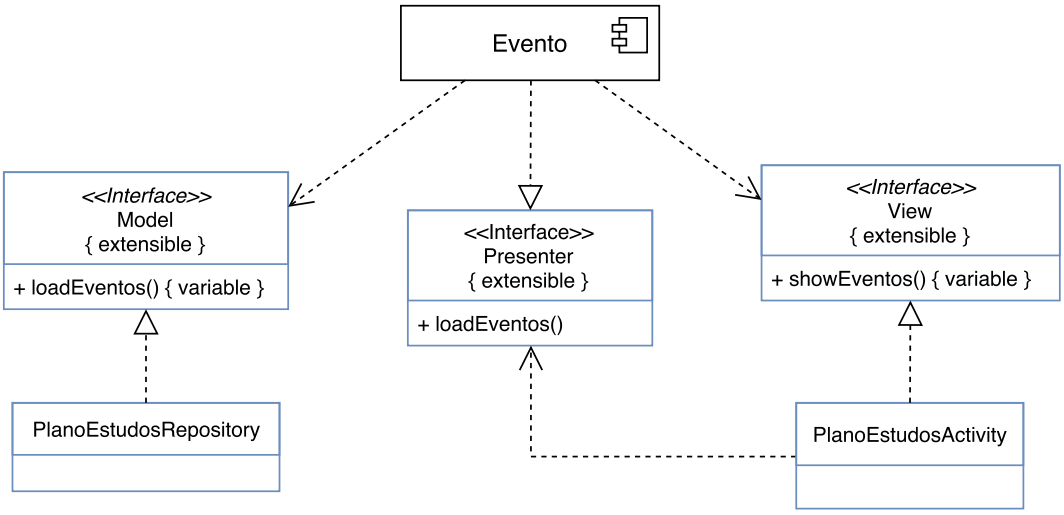

Figura 4.26: Utilização do componente de eventos na instância um.

Por fim, o componente de sessão foi utilizado para implementar a funcionalidade de login. Assim, para fornecer tal funcionalidade, foi necessário implementar as interfaces *View* e *Model* exigidas pelo componente de sessão. Na implementação da interface *Model*, que é feita pela classe *LoginValidator*, o desenvolvedor define como será o processo de validação das credenciais do aluno, através da implementação do método *validateCredenciais()*. A implementação pode ser vista em 4.10.

Código 4.10: Implementação da interface Model pela classe LoginValidator.

```
1
2  class LoginValidator @Inject
3  constructor(private val loginRestService: LoginRestService): Model {
4
5      override fun validateCredenciais(credencial: CredencialAcesso): Observable<Boolean> {
6
7       /*
8          Envia as credenciais de acesso para um servidor que irá validar os dados
9          e retornará true ou false dependendo do estado da validação.
10         */
11        return loginRestService.login(credenciais).map { response -> response == HTTP_STATUS.OK }
12     }
13 }
```

Já na implementação da *View*, que é feita pela classe *LoginActivity*, o programador define as ações que serão executadas caso as credenciais do usuário sejam validadas corretamente e o acesso seja concedido (*acessoGranted()*), e também as ações que serão executadas caso as credenciais do usuário não sejam validadas corretamente e o acesso seja bloqueado (*acessoBlocked()*). A implementação pode ser vista em 4.11.

Código 4.11: Implementação da interface View pela classe LoginActivity.

```
1
2  class LoginActivity : AppCompatActivity(), View {
3
4      //Inicialização do layout
5      private val cpf by lazy { findViewById<EditText>(R.id.cpf) }
```

```
6       private val senha by lazy { findViewById<EditText>(R.id.senha) }
7       private val entrar by lazy { findViewById<Button>(R.id.entrar) }
8
9       //Variável que receberá a implementação do Presenter do componente de sessão.
10      private lateinit var sessaoPresenter: Presenter
11
12      //Variável que receberá a implementação da interface Model.
13      @Inject lateinit var sessaoModel: Model
14
15      override fun onCreate(savedInstanceState: Bundle?) {
16
17          //Injeta a implementação da interface Model na variável sessaoModel.
18          (application as CustomApp).inject(this)
19
20          //Inicialização do componente de sessão via classe CustomApp.
21          sessaoPresenter = (application as CustomApp).buildSessaoComponent(sessaoModel)
22
23          super.onCreate(savedInstanceState)
24          setContentView(R.layout.activity_sessao)
25
26          // Associa a View ao componente de sessão
27          sessaoPresenter.attachView(this)
28      }
29
30      override fun onResume(){
31       super.onResume()
32
33       // Atribui um listener que irá chamar o método de login do componente de sessão.
34       entrar.setOnClickListener { _,_ ->
35
36            val credencialAcesso = CredencialAcesso( cpf.text.toString(), senha.text.toString())
37
38            sessaoPresenter.login(credencialAcesso)
39           }
40      }
41
42      override fun acessoGranted() {
43
44          //Redireciona o usuário para a tela de histórico.
45         startActivity(Intent(this, HistoricoActivity::class.java))
46      }
47
48      override fun acessoBlocked() {
49
50          //Exibe um alerta informando que houve erro ao realizar login.
51         AlertDialog.Builder(this)
52                     .setTitle("Atenção")
53                     .setMessage("Usuário ou senha incorretos.")
54                     .setPositiveButton("Ok", null)
55                     .show()
56      }
57  }
```

Na figura 4.27 é possível ver, de maneira geral, como as implementações das interfaces do componente de sessão se relacionam nessa instância.

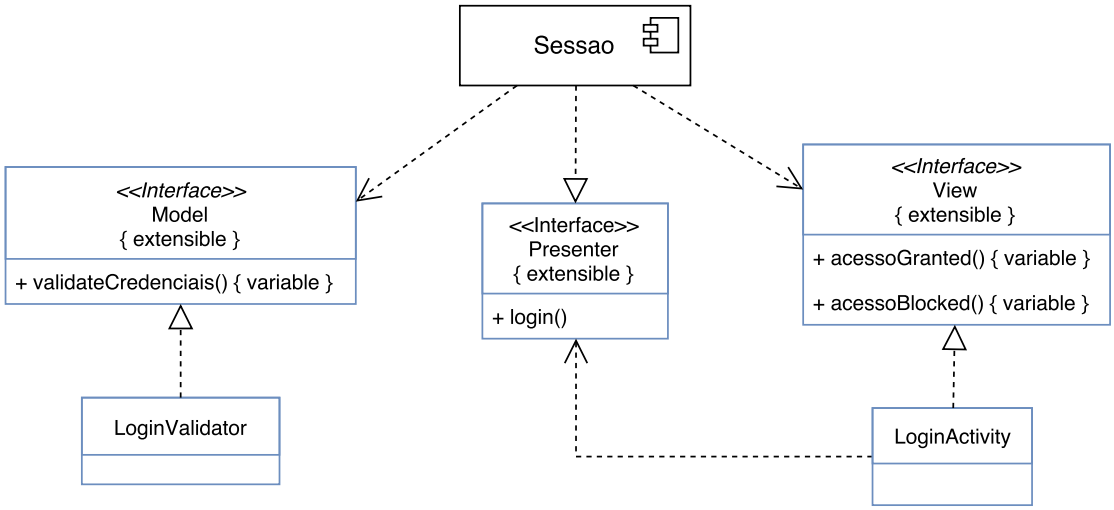

Figura 4.27: Utilização do componente de sessão na instância um.

Nas figuras 4.28, 4.29 e 4.30, são mostradas algumas imagens das funcionalidades implementadas sendo executadas nessa instância do MyUni.

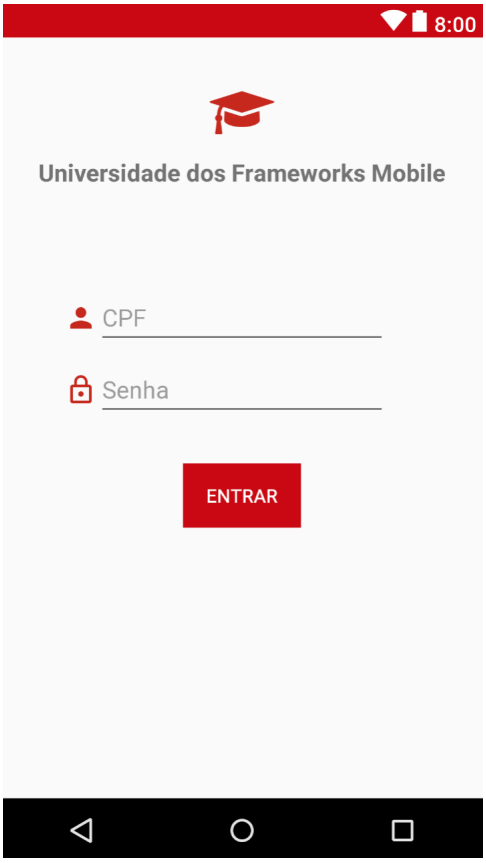

Figura 4.28: Funcionalidade de login na instância um.

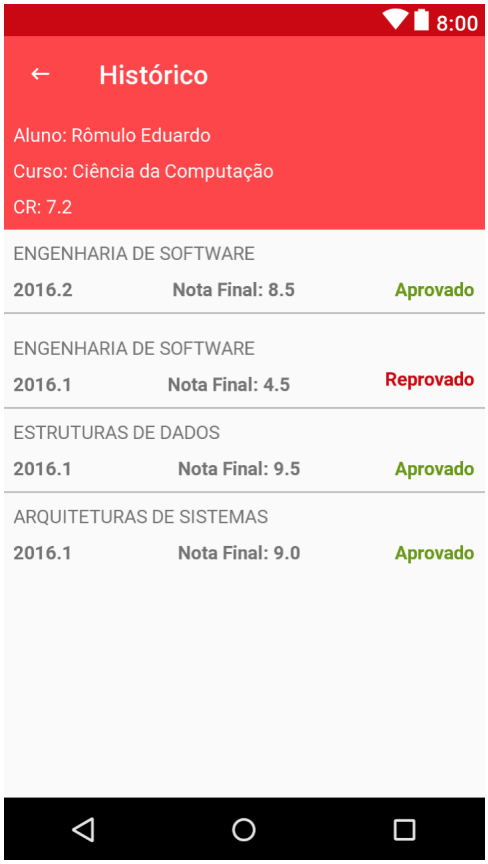

Figura 4.29: Funcionalidade de histórico na instância um.

Figura 4.30: Funcionalidade de plano de estudos na instância um.

4.5.2 Instância Dois

A segunda instância do MyUni foi criada de maneira a fornecer um aplicativo que pudesse ser utilizado por calouros da universidade, de modo que eles tivessem acesso a funcionalidades que permitissem visualizar um mapa com localizações de unidades e salas da universidade, uma lista de notícias sobre a universidade, e uma lista com as linhas de ônibus que circulam pela universidade juntamente com as rotas que essas linhas fazem. A figura 4.31 mostra um diagrama de casos de uso com as funcionalidades propostas para esse aplicativo.

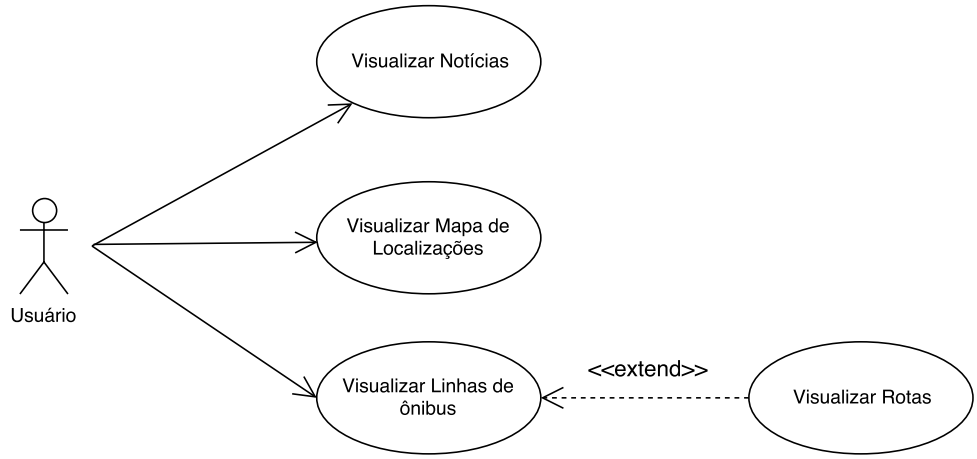

Figura 4.31: Diagrama de casos de uso para a instância dois.

A funcionalidade de mapa de localização de salas e unidades foi planejada de maneira que o usuário pudesse visualizar, a partir de um mapa, as unidades e salas da universidade. Para fornecer tal funcionalidade, foi necessário utilizar o componente de localização. Para isso, como mostra a figura 4.32, foi necessário implementar a interfaces *View* e *Model*, necessárias para o correto funcionamento do componente.

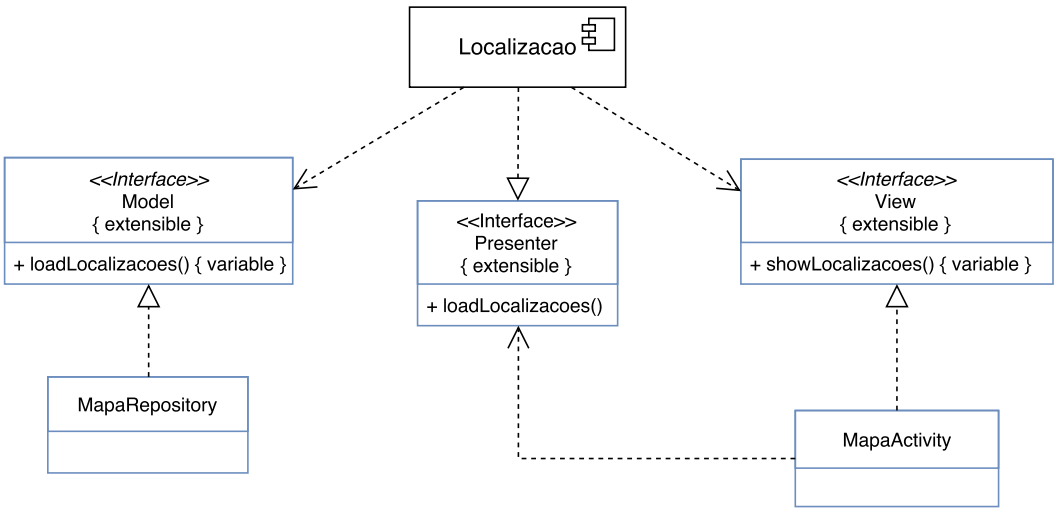

Figura 4.32: Utilização do componente de localização na instância dois.

A funcionalidade de linhas de ônibus foi planejada de maneira que o usuário pudesse visualizar uma lista com as linhas de ônibus disponíveis juntamente com as rotas que elas fazem. Para fornecer tais funcionalidades, foi utilizado o componente de linhas de ônibus. Para isso, como pode ser visto em 4.12, foi necessário realizar a implementação da interface *View* para realizar a exibição da lista de linhas de ônibus obtida, e também foi necessário implementar a interface *Model* para realizar a obtenção da lista de linhas de ônibus, como pode ser visto em 4.13.

Código 4.12: Implementação da interface View pela classe LinhasOnibusActivity.

```
1
2  class LinhasOnibusActivity : AppCompatActivity(), LinhasOnibusContract.View {
3
4      //Inicialização do layout
5      private val linhasOnibusList by lazy { findViewById<RecyclerView>(R.id.linhasOnibusList) }
6      private val linhasOnibusListAdapter by lazy { LinhasOnibusListAdapter(this, null) }
7
8      //Variável que receberá a implementação do Presenter do componente de localização
9      private lateinit var historicoPresenter: LinhasOnibusContract.Presenter
10
11     //Variável que receberá a implementação da interface Model
12     @Inject lateinit var linhasOnibusModel: LinhasOnibusContract.Model
13
14     override fun onCreate(savedInstanceState: Bundle?) {
15
16         /*
17         Injeta a implementação da interface Model na variável linhasOnibusModel.
18         Esse método é específica da aplicação, que é representada pela classe MeuApp,
19         que é uma subclasse da classe MyUni.
20         */
21         (application as MeuApp).inject(this)
22
23         /*
24         Inicialização do componente de histórico via classe CustomApp.
25         Note que para criar o componente, é necessário fornecer uma implementação
26         da interface Model, que nesse caso, é o objeto historicoModel.
27         */
28         linhasOnibusPresenter = (application as MeuApp).buildLinhasComponent(linhasOnibusModel)
29
30         super.onCreate(savedInstanceState)
31         setContentView(R.layout.activity_linhas_onibus)
32
33         linhasOnibusList.adapter = linhasOnibusListAdapter
34
35         //Associa a View ao componente de linhas de ônibus
36         linhasOnibusPresenter.attachView(this)
37
38         //Solicita o carregamento da lista de linhas de ônibus
39         linhasOnibusPresenter.loadLinhas()
40     }
41
42     override fun showLinhas(linhasOnibus: List<Linha>) {
43
44         /*
45         Atribui as linhas de ônibus ao ListAdapter
46         e notifica a Activity para realizar um refresh na lista.
47         */
48         linhasOnibusListAdapter.add(linhasOnibus)
49         linhasOnibusListAdapter.notifyDataSetChange()
50     }
51 }
```

Código 4.13: Implementação da interface Model pela classe LinhasOnibusRepository.

```
1
2  class LinhasOnibusRepository @Inject
3  constructor(private val linhasRestService: LinhasRestService): LinhasOnibusContract.Model {
```

```
4
5       override fun loadLinhas(): Observable<List<Linha>> {
6           return linhasRestService.loadLinhas()
7       }
8   }
```

Na figura 4.33 é possível ver, de maneira geral, como as implementações das interfaces do componente de linhas de ônibus se relacionam nessa instância.

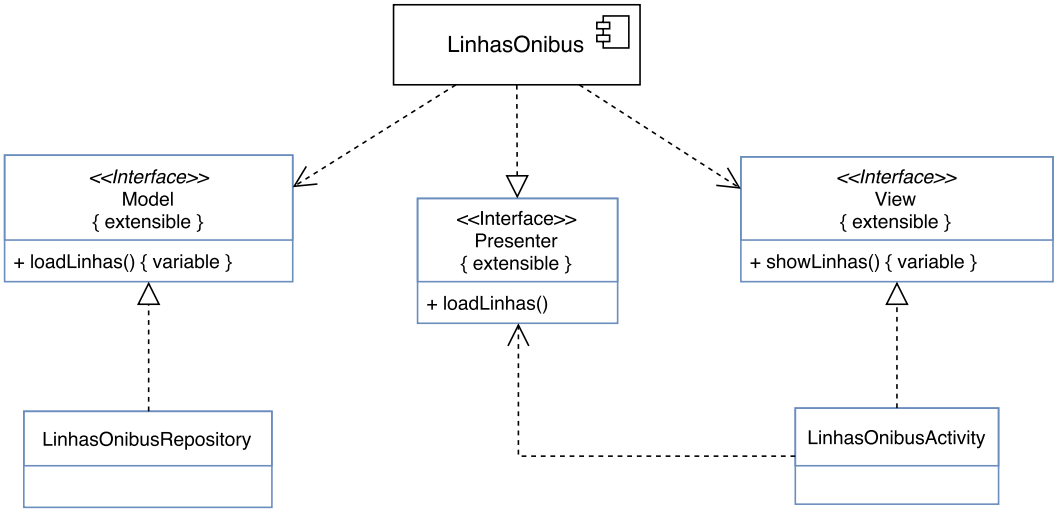

Figura 4.33: Utilização do componente de linhas de ônibus na instância dois.

Para implementar a funcionalidade de notícias, que basicamente tem o objetivo de obter e exibir uma lista das últimas notícias da universidade, foi necessário utilizar o componente de notícias e implementar a interface *Model*, para permitir a obtenção das notícias, e a interface *View*, para permitir a exibição das notícias obtidas, como é mostrado na figura 4.35.

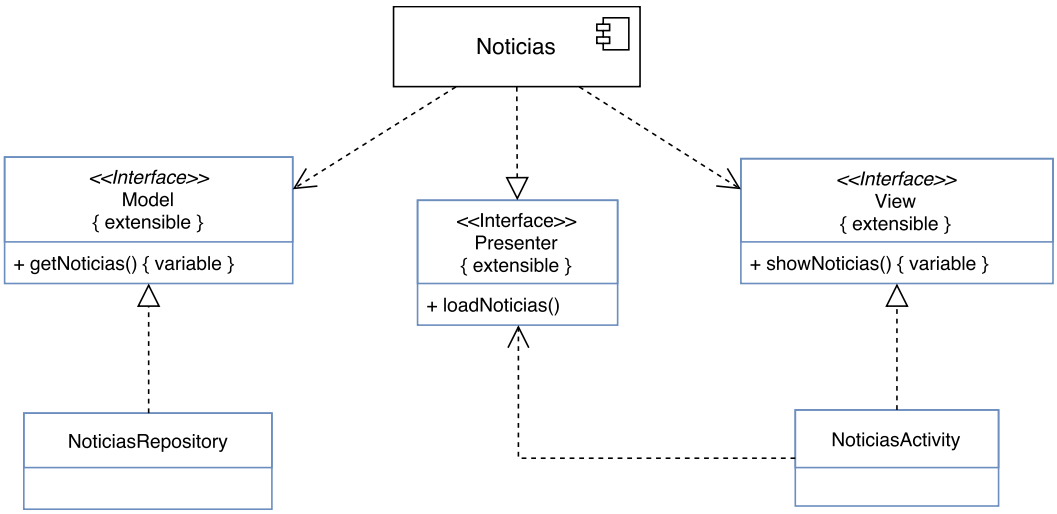

Figura 4.34: Utilização do componente de notícias na instância dois.

Após as implementações das interfaces requeridas para cada funcionalidade, foi necessário ape-

nas realizar a inicialização dos componentes e utilizá-los na aplicação. Para isso, basta que o aplicativo requisite os componentes desejados através da classe *MyUni*. Nas figura, são mostradas algumas imagens das funcionalidades implementadas sendo executadas nessa instância do MyUni.

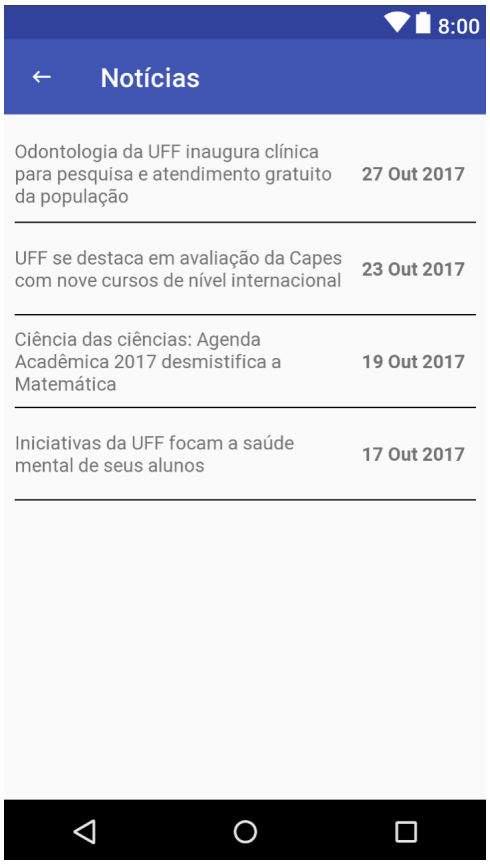

Figura 4.35: Funcionalidade de notícias na instância dois.

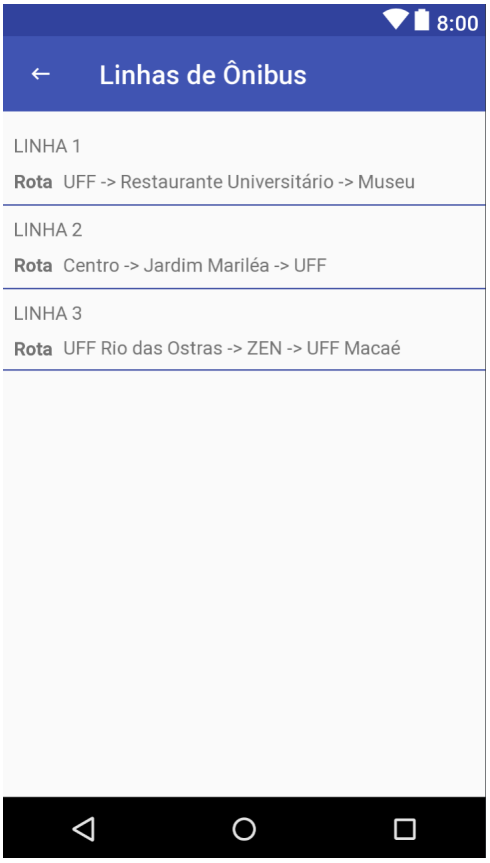

Figura 4.36: Funcionalidade de linhas de ônibus na instância dois.

4.5.3 Instância Três

Essa instância foi criada de maneira a fornecer um aplicativo com as mesmas funcionalidades encontradas nas duas primeiras instâncias, porém implementadas de maneira diferente, para mostrar como o framework permite flexibilidade de implementação.

As principais diferenças dessa instância em relação as outras são em relação as implementações das interfaces *View* dos componentes utilizados, ou seja, a maneira como os dados são apresentados para o usuário final nessa instância, ficou diferente da maneira como eles são apresentados nas outras duas. Nas figuras 4.37 e 4.38 é possível ver algumas diferenças visuais entre as funcionalidades das duas primeiras instâncias e da terceira instância.

Figura 4.37: Funcionalidade de login na instância três.

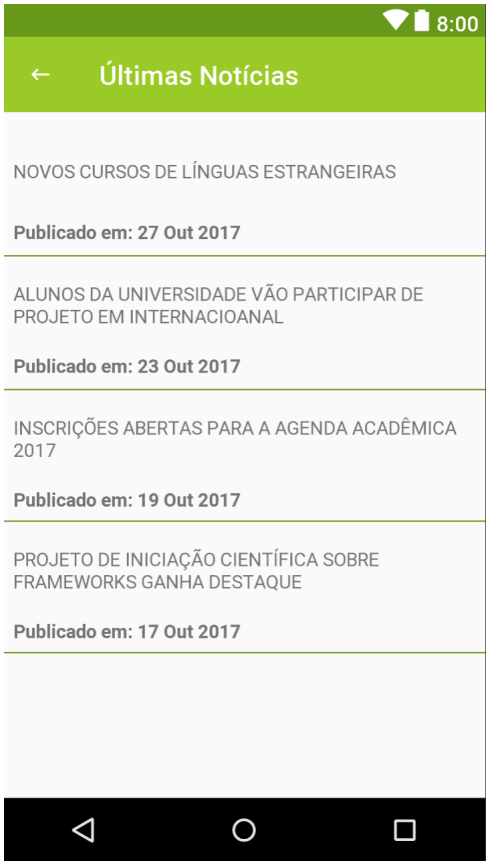

Figura 4.38: Funcionalidade de notícias na instância três.

Além das diferenças visuais, também houve algumas diferenças na implementação de algumas interfaces *Model* de alguns componentes, ou seja, a maneira como alguns dados são obtidos e tratados também foi modificada. Um pequeno exemplo, é a implementação da interface Model do componente de sessão. Como pode ser visto nessa implementação, para garantir a segurança das informações trocadas entre o servidor e a aplicação, as credenciais de acesso são criptografadas por uma classe *Encrypter* antes de serem enviadas. Essa classe foi criada pelo desenvolvedor da aplicação e portanto, é específica da instância.

Código 4.14: Implementação da interface Model pela classe CredencialValidator.

```
1
2  class CredencialValidator @Inject
3  constructor(private val sessaoRestService: SessaoRestService,
4              private val encrypter: Encrypter): SessaoContract.Model {
5
6              override fun validateCredenciais(credencial: CredencialAcesso): Observable<Boolean> {
7
8                  val loginCripto = encrypter.encryptLogin(credencial.login)
9                  val senhaCripto = encrypter.encryptSenha(credencial.senha)
10
11                 val credencialCripto = CredencialAcesso(loginCripto, senhaCripto)
12
13                 return sessaoRestService
14                             .login(credencialCripto)
15                             .map { response -> response == HTTP_STATUS.OK }
```

```
16              }
17  }
```

Além disso, também há pequenas diferenças na implementação da interface *View*. Uma delas, é que caso o acesso seja concedido ao usuário, ele é redirecionado para a tela de plano de estudos do aplicativo, que é representada pela classe *PlanoEstudosActivity*. Nessa instância, a implementação da *View* é realizada pela classe *MainActivity*, como pode ser visto em 4.15.

Código 4.15: Implementação da interface View pela classe MainActivity.

```
1
2   class MainActivity : AppCompatActivity(), SessaoContract.View {
3
4       //Inicialização do layout
5       private val matricula by lazy { findViewById<EditText>(R.id.cpf) }
6       private val senha by lazy { findViewById<EditText>(R.id.senha) }
7       private val entrar by lazy { findViewById<Button>(R.id.entrar) }
8
9       //Variável que receberá a implementação do Presenter do componente de sessão.
10      private lateinit var sessaoPresenter: SessaoContract.Presenter
11
12      //Variável que receberá a implementação da interface Model.
13      @Inject lateinit var sessaoModel: SessaoContract.Model
14
15      /* ... */
16
17      override fun acessoGranted() {
18
19          //Redireciona o usuário para a tela de plano de estudos.
20          startActivity(Intent(this, PlanoEstudosActivity::class.java))
21      }
22
23      /*...*/
24  }
```

Nas tabelas 4.2 e 4.3, é possível ver um resumo das diferenças nas implementações dos hot-spots de cada componente utilizado pelas instâncias. As tabelas mostram, respectivamente, as implementações dos hot-spots referentes as interfaces *Model* e dos hot-spots referentes as interfaces *View* dos componentes.

Hot-spot	Instância Um	Instância Dois	Instância Três
getHistorico()	Obtém o histórico através do endpoint /aluno/{id-aluno}/historico/cr de uma API REST fictícia	Não implementado	Obtém o histórico através do endpoint enpoint /estudante/{matricula}/historico/ de uma API REST fictícia
getCr()	Obtém o CR através do endpoint /aluno/{id-aluno}/historico/cr da API REST	Não implementado	Obtém o CR através do acesso do atributo CR do objeto retornado pelo endpoint de consulta de histórico
loadEventos()	Obtém o plano de estudos do aluno através do endpoint /aluno/{id-aluno}/plano da API REST	Não implementado	Obtém o plano de estudos do aluno através do endpoint /estudante/{matricula}/plano-estudos da API REST
validateCrendenciais()	Realiza o login do usuário no sistema através da chamada do endpoint /login da API REST	Não implementado	Criptografa as credenciais de acesso e em seguida realiza o login do usuário no sistema através da chamada do endpoint /acesso da API REST
loadLocalizacoes()	Não implementado	Obtém as localizações das unidades através do endpoint /universidade/unidades de uma API REST	Obtém uma lista de unidades da universidade através do endpoint /locais/ da API REST
loadLinhas()	Não implementado	Obtém uma lista de linhas de ônibus da universidade através do endpoint /universidade/linhas/ da API REST	Obtém uma lista de linhas de ônibus da universidade através do endpoint /transporte/ da API REST
getNoticias()	Não implementado	Obtém uma lista de noticias através do endpoint /universidade/notiticas/ da API REST	Obtém uma lista de noticias através de um feed RSS fictício

Tabela 4.2: Descrição das implementações das interfaces *Model* nas três instâncias.

Hot-spot	Instância Um	Instância Dois	Instância Três
showHistorico()	Exibe o histórico como uma lista, que exibe dados da disciplina, nota final e situação	Não implementado	Exibe os mesmos dados que a instância um, porém os dados são organizados e exibidos de maneira distinta.
showCr()	Exibe o CR no topo da tela.	Não implementado.	Mesmo comportamento que a instância um.
showEventos()	Exibe os dias da semana em abas. Para cada aba, é exibida uma lista com o plano de estudos. Os dados exibidos são o nome da disciplina, o horário de início e fim, e a sala.	Não implementado.	Exibe uma lista com os dias da semana. Ao clicar em um dia da semana, são exibidas as disciplinas daquele dia. Os dados exibidos são os mesmos que o da instância um.
acessoGranted()	Redireciona o usuário para a tela que exibe o histórico.	Não implementado.	Redireciona o usuário para a tela que exibe o plano de estudos.
acessoBlocked()	Exibe um alerta informando que as credenciais fornecidas estão incorretas.	Não implementado.	Exibe um alerta informando que as credenciais fornecidas estão incorretas e oferece uma opção para contatar o suporte técnico.
showLocalizacoes()	Não implementado.	Exibe as unidades em um mapa através de *pins*.	Exibe as unidades em um mapa através de *pins*. Ao clicar em um *pin*, o nome da unidade é exibido.
showLinhas()	Não implementado.	Exibe as linhas de ônibus em uma lista. Os dados exibidos são o nome da linhas, e o nome das paradas da rota que ela faz.	Mesmo comportamento que a instância um. Porém, ao clicar em uma linha, um mapa é aberto e nele são marcados os pontos referentes as paradas.
showNoticias()	Não implementado.	Exibe as notícias em uma lista. Os dados exibidos são o título e a data de publicação.	Exibe os mesmos dados que a instância dois, porém os dados são organizados e exibidos de maneira distinta

Tabela 4.3: Descrição das implementações das interfaces *View* nas três instâncias.

Capítulo 5

Conclusão e Trabalhos Futuros

Através da criação de três instâncias do MyUni, foi possível verificar que ele, de fato, permite a criação de diferentes aplicações para a gerência de informações acadêmicas através da utilização de seus componentes. Mesmo nos casos em que as funcionalidades eram similares e utilizavam os mesmos componentes, suas implementações poderiam ser realizadas de maneira diferente, fazendo com que cada instância tivessem comportamento e aparência distintos entre si.

A utilização do ProMoCF permitiu que o processo de criação do MyUni fosse realizado de maneira mais simples, principalmente pelo fato dele ser um processo leve, iterativo, e próprio para o desenvolvimento de frameworks baseados em componentes. Portanto, o ProMoCF mostrou-se um excelente modelo de processo para desenvolvimento do MyUni. Além disso, a utilização da linguagem UML-F, por ser própria para a modelagem e especificação de frameworks, também auxiliou o processo de definição dos componentes, interfaces, hot-spots e frozen-spots do MyUni.

O fato de o MyUni ser simples, também permite que outros alunos possam modificar e estender o próprio framework Assim, conclui-se que o MyUni cumpriu o seu propósito de fornecer um meio simples, rápido e flexível para o desenvolvimento de aplicações para o gerenciamento de informações acadêmicas, permitindo que os próprios alunos das universidades possam criar aplicativos com funcionalidades específicas para as suas necessidades a partir da utilização dos componentes fornecidos pelo MyUni, sem a necessidade de criar aplicativos a partir do zero.

É importante notar que, mesmo que os objetivos tenham sido alcançados, dificuldades foram encontradas durante o projeto. Uma das principais foi analisar uma grande quantidade de aplicações do domínio estudado para identificar suas similaridades e particularidades, a fim de se obter uma ideia inicial de quais componentes e funcionalidades poderiam ser fornecidos pelo MyUni. O problema é que ao se analisar uma grande quantidade de aplicações, um grande volume de modelos de objetos acaba sendo gerado, tornando o processo de análise mais difícil. Outra dificuldade encontrada foi garantir os requisitos de reúso e flexibilidade do framework, pois para garanti-los, foi necessário fornecer um grau moderado de abstração, de modo que o usuário precisasse realizar poucas implementações e ao mesmo tempo pudesse reutilizar os mesmos componentes de diferentes maneiras.

Como trabalhos futuros, propõe-se a implementação completa do MyUni para o Android, além da

portabilidade do framework para a plataforma iOS. Além disso estuda-se a possibilidade de se criar novos componentes para o MyUni, de maneira a permitir que suas instâncias possam fornecer funcionalidades de cunho colaborativo e social (*e.g.* chats, compartilhamento de arquivos, sistema de avaliações etc.).

Bibliografia

1 ANDROID Developers. Disponível em: <https://developer.android.com/index.html>. Acesso em: 27 out. 2017.

2 UFF, STI. *UFF Mobile*. Set. 2017. Disponível em: <https://play.google.com/store/apps/details?id=br.uff.uffmobile>. Acesso em: 27 out. 2017.

3 UFRJ, Equipe SIGA. *Portal Aluno UFRJ*. Jan. 2017. Disponível em: <https://play.google.com/store/apps/details?id=br.ufrj.gnosys.mobile>. Acesso em: 27 out. 2017.

4 PROCIT, UFPE-. *UFPE Mobile*. Jul. 2016. Disponível em: <https://play.google.com/store/apps/details?id=br.ufpe.mobile>. Acesso em: 27 out. 2017.

5 PUC-SP, Fundação São Paulo /. *PUC-SP Mobile*. Abr. 2014. Disponível em: <https://play.google.com/store/apps/details?id=com.pucsp.app>. Acesso em: 27 out. 2017.

6 UFRGS. *UFRGS Mobile*. Set. 2017. Disponível em: <https://play.google.com/store/apps/details?id=com.cpd.ufrgsmobile>. Acesso em: 27 out. 2017.

7 SOMMERVILLE, Ian. *Software Engineering*. 9 edition. Boston: Pearson, mar. 2010. ISBN: 978-0-13-703515-1.

8 SCHMIDT, Douglas C; GOKHALE, Aniruddha; NATARAJAN, Balachandran. Leveraging Application Frameworks. *Queue*, v. 2, n. 5, p. 66–75, jul. 2004. ISSN: 1542-7730. DOI: 10.1145/1016998.1017005. Disponível em: <http://doi.acm.org/10.1145/1016998.1017005>. Acesso em: 27 out. 2017.

9 WEINREICH, Rainer; SAMETINGER, Johannes. Component-based Software Engineering. In: HEINEMAN, George T.; COUNCILL, William T. (Eds.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. p. 33–48. ISBN: 978-0-201-70485-3. Disponível em: <http://dl.acm.org/citation.cfm?id=379381.379482>. Acesso em: 1 nov. 2017.

10 SZYPERSKI, Clemens; GRUNTZ, Dominik; MURER, Stephan. *Component software: beyond object-oriented programming*. 2nd ed. London: Addison-Wesley, 2003. (Addison-Wesley component software series). OCLC: 249177345. ISBN: 978-0-201-74572-6.

11 TUTORIALSPOINT.COM. *Software Architecture and Design Component-Based Architecture*. Disponível em: <http://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm>. Acesso em: 31 out. 2017.

12 JAVA Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle. Disponível em: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>. Acesso em: 2 jan. 2018.

13 .NET. Disponível em: <https://www.microsoft.com/net/>. Acesso em: 1 nov. 2017.

14 BOSCH, Jan; MOLIN, Peter; MATTSSON, Michael; BENGTSSON, PerOlof. Object-oriented Framework-based Software Development: Problems and Experiences. *ACM Comput. Surv.*, v. 32, 1es, mar. 2000. ISSN: 0360-0300. DOI: 10.1145/351936.351939. Disponível em: <http://doi.acm.org/10.1145/351936.351939>. Acesso em: 5 nov. 2017.

15 GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John; BOOCH, Grady. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition. Reading, Mass: Addison-Wesley Professional, nov. 1994. ISBN: 978-0-201-63361-0.

16 SCHERP, Ansgar. *A component framework for personalized multimedia applications*. Edewecht: OlWIR, Oldenburger Verl. für Wirtschaft, Informatik und Recht, 2007. (Oldenburg computer science series, 1). ISBN: 978-3-939704-11-9.

17 RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. ISBN: 978-0-321-24562-5.

18 FONTOURA, Marcus; PREE, Wolfgang; RUMPE, Bernhard. UML-F: A Modeling Language for Object-Oriented Frameworks. *arXiv:1409.6915 [cs]*, v. 1850, p. 63–82, 2000. arXiv: 1409.6915. DOI: 10.1007/3-540-45102-1_4. Disponível em: <http://arxiv.org/abs/1409.6915>. Acesso em: 27 out. 2017.

19 FONTOURA, Marcus; PINTO, Sérgio Crespo; CARLOS, José Pereira; LUCENA, Carlos Felipe. *ALADIN: An Architecture for Learningware Applications Design and Instantiation*. 1998. Tese (Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil. Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/98_34_fontoura.ps.gz>. Acesso em: 28 out. 2017.

20 WHAT is Agile Software Development? Jun. 2015. Disponível em: <https://www.agilealliance.org/agile101/>. Acesso em: 5 nov. 2017.

21 KRUCHTEN, Philippe. *The Rational Unified Process: An Introduction, Second Edition*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 978-0-201-70710-6.

22 PREE, Wolfgang. Hot-spot-driven framework development. In: *Summer School on Reusable Architectures in Object-Oriented software Development*. ACM, 1995. p. 123–127.

23 PREE, Wolfgang; POMBERGER, Gustav; KAPSNER, Franz. Framework component systems: Concepts, design heuristics, and perspectives. en. In: *Perspectives of System Informatics*. Springer, Berlin, Heidelberg, jun. 1996. (Lecture Notes in Computer Science), p. 330–340. ISBN: 978-3-540-62064-8 978-3-540-49637-3. DOI: 10.1007/3-540-62064-8_27. Disponível em: <https://link.springer.com/chapter/10.1007/3-540-62064-8_27>. Acesso em: 5 nov. 2017.

24 IDC: Smartphone OS Market Share. Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>. Acesso em: 6 nov. 2017.

25 CHEESMAN, John; DANIELS, John. *UML components: a simple process for specifying component-based software*. Boston, MA: Addison-Wesley, 2001. (Component software series). OCLC: ocm45080169. ISBN: 978-0-201-70851-6.

26 FONTOURA, Marcus; CRESPO, Sérgio; LUCENA, Carlos José; ALENCAR, Paulo S. C; COWAN, Donald D. Using viewpoints to derive object-oriented frameworks: a case study in the web-based education domain. *Journal of Systems and Software*, v. 54, n. 3, p. 239–257, nov. 2000. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(00)00054-6. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121200000546>. Acesso em: 27 out. 2017.

27 MARTIN, Robert C. *Design Principles and Design Patterns*. English. 2000. Disponível em: <https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>. Acesso em: 6 nov. 2017.

28 POTEL, Mike. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", Taligent Inc, maio 2011.

29 APPLICATION | Android Developers. Disponível em: <https://developer.android.com/reference/android/app/Application.html>. Acesso em: 6 nov. 2017.

30 ACTIVITY | Android Developers. Disponível em: <https://developer.android.com/reference/android/app/Activity.html>. Acesso em: 6 nov. 2017.

31 KOTLIN Programming Language. Disponível em: <https://kotlinlang.org/>. Acesso em: 5 nov. 2017.

32 COMPARISON to Java - Kotlin Programming Language. Disponível em: <https://kotlinlang.org/docs/reference/comparison-to-java.html>. Acesso em: 5 nov. 2017.

33 MEET Android Studio | Android Studio. Disponível em: <https://developer.android.com/studio/intro/index.html>. Acesso em: 5 nov. 2017.

34 GOOGLE AdMob - Mobile App Monetization & In App Advertising. Disponível em: <https://www.google.com/admob/>. Acesso em: 3 jan. 2018.

35 FIREBASE. Disponível em: <https://firebase.google.com/>. Acesso em: 3 jan. 2018.

36 GOOGLE Play Store: number of apps 2009-2017 | Statistic. Disponível em: <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. Acesso em: 27 out. 2017.

37 DAGGER A fast dependency injector for Android and Java. Disponível em: <https://google.github.io/dagger/>. Acesso em: 3 jan. 2018.

38 RXJAVA Reactive Extensions for the JVM a library for composing asynchronous and event-based programs using observable sequences for the Java VM. original-date: 2013-01-08T20:11:48Z. Jan. 2018. Disponível em: <https://github.com/ReactiveX/RxJava>. Acesso em: 4 jan. 2018.

39 CREATE an Android Library | Android Studio. Disponível em: <https://developer.android.com/studio/projects/android-library.html>. Acesso em: 4 jan. 2018.

40 MORAES, Rômulo Eduardo Garcia. *myuni: Trabalho de conclusão do curso de Ciência da Computação pela Universidade Federal Fluminense*. original-date: 2018-01-04T18:19:53Z. Jan. 2018. Disponível em: <https://github.com/regmoraes/myuni>. Acesso em: 4 jan. 2018.