

Desenvolvimento de Sistemas de Software

Mestrado Integrado em Engenharia Informática

André Diogo



A75505

António Silva



A73827

Gonçalo Pereira



A74413

Helder Gonçalves



A64286

30 de Dezembro de 2016

Conteúdo

1	Introdução	2
2	Análise	3
2.1	Análise de Requisitos	5
2.2	Diagrama de Use Cases	6
2.3	Interface com o Utilizador	6
3	Arquitetura Lógica	8
3.1	Arquitetura Lógica I	9
3.2	Arquitetura Lógica II	10
3.3	Arquitetura Lógica Final	12
4	Conclusão	14
	Anexos	15

Capítulo 1

Introdução

Este projeto foi proposto no âmbito da Unidade Curricular de **Desenvolvimento de Sistemas de Software** e tem como objetivo o desenvolvimento de um sistema de suporte à partilha de despesas num apartamento. Desta forma, **o cliente fictício** pretende uma aplicação que permita suportar o registo de despesas e a gestão do seu pagamento por parte dos moradores registados, mantendo uma conta corrente para cada um.

Para além das especificações funcionais, foi também sugerido o uso de tecnologias orientadas a objetos para a implementação da aplicação, assim como uma concepção e especificação suportada pela linguagem **UML**, de acordo com a metodologia seguida ao longo do semestre.

Nesta fase final do projeto, além da prévia exploração e elaboração do **Modelo de Domínio**, **Modelo de Use Case** e a **especificação dos Use Case**, implementamos também diferentes diagramas, entre os quais se destacam os **Diagramas de Sequência**, **Diagramas de Implementação** e **Máquinas de Estado**.

Assim, e aquando da realização do projeto, o grupo pretende criar uma aplicação robusta e concisa, eliminando quaisquer falhas e simplificando o quotidiano dos utilizadores da plataforma.

Capítulo 2

Análise

Decidimos modelar um **sistema de gestão de despesas** que simulasse um uso mais típico de arrendamento em que várias pessoas arrendam uma habitação a um senhorio e como tal este tem um certo poder de gestão de despesas que não está disponível aos moradores, mas estes também podem decidir entre si dividir as suas despesas.

Um pouco nesta veia, decidimos modelar um sistema **democrático** entre os moradores, evitando uma solução mais **fácil** e **óbvia** de nomear um destes como responsável (tal como acontece, por vezes, em residências universitárias) em que os moradores votam para uma despesa entrar em vigor e propõem despesas livremente e alterações a essas despesas, inclusive quanto e como acham que toda a gente devia pagar e estas apenas entram em vigor após haver unanimidade entre os envolvidos.

Para tal concebemos um sistema em que o senhorio **tem a capacidade de gerar despesas ditas gerais** que apenas ele controla e a que os moradores se sujeitam. Os moradores depois podem entre si gerar despesas ditas locais, que **têm de ser aprovadas por todos os envolvidos** para entrar em vigor ou para serem removidas, e, caso sejam propostas alterações, surgem como novas despesas que precisam também de ser aprovadas. Quando uma despesa local é aprovada devem-se eliminar as despesas alteradas recursivamente, porque é lógico só uma configuração poder entrar em vigor.

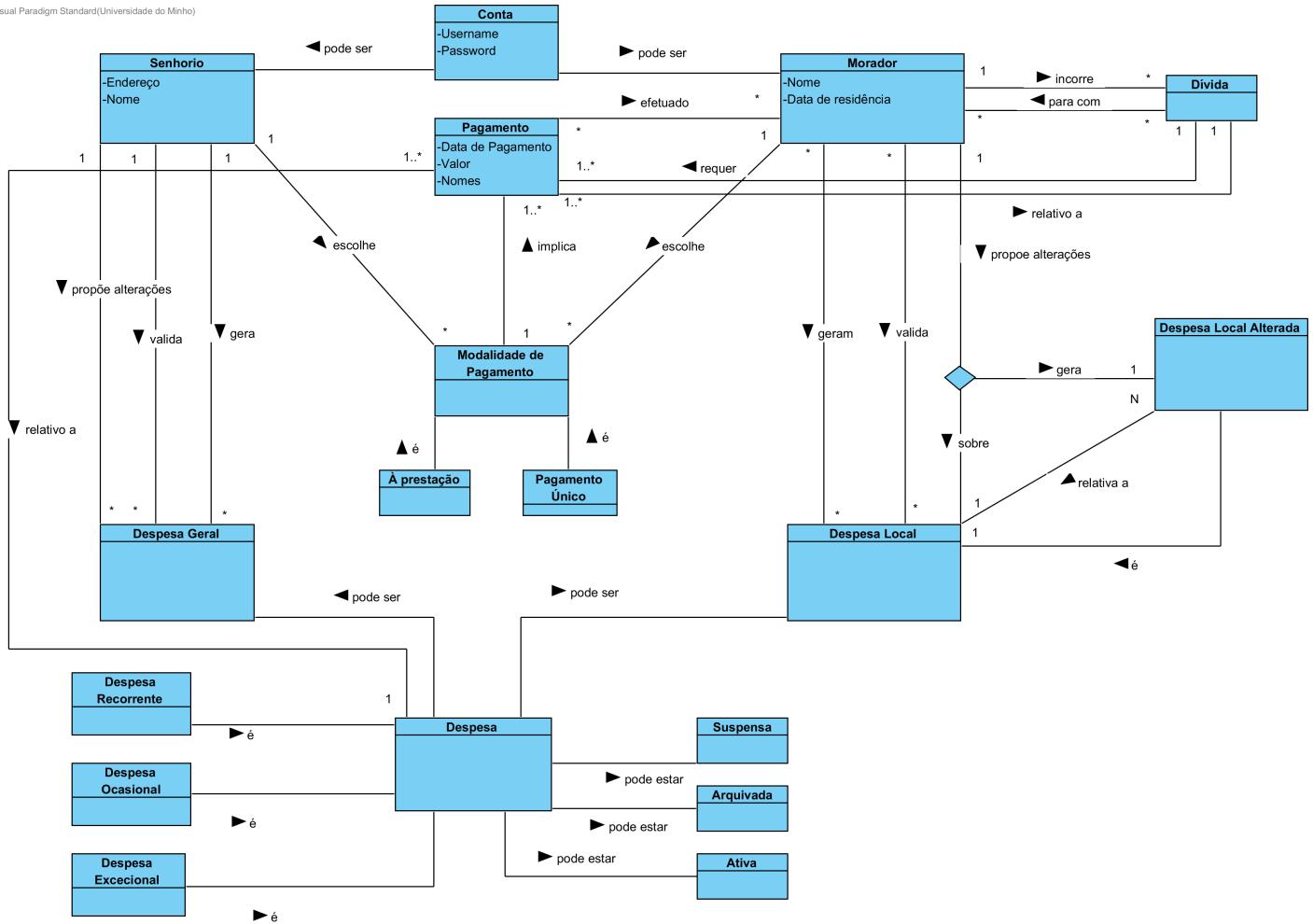


Figura 2.1: Modelo de Domínio.

Decidimos que deveria haver a **possibilidade de os moradores decidirem dividir as despesas por quota**, numerário ou equitativamente, sendo que a interface trataria de acomodar estas decisões.

Partimos então à definição de uma série de requisitos que o nosso programa deveria conseguir cumprir, dos quais extraímos os use cases dos dois principais tipos de utilizadores, senhorio e moradores.

2.1 Análise de Requisitos

- Os moradores e senhorio terão cada um uma conta corrente.
- Os moradores devem poder registar despesas.
- O senhorio deve poder registar despesas.
- As despesas registadas pelo senhorio não podem ser alteradas pelos moradores.
- As despesas registadas por moradores requerem que todos os moradores estejam de acordo para serem aprovadas, alteradas ou removidas.
- Os moradores devem poder pagar despesas a outros moradores podendo ficar em dívida uns para com os outros.
- Ao registar uma despesa, quer o senhorio ou o morador que a regista indica todos os moradores envolvidos e a quota ou numerário que cada morador deve pagar, assim como o valor da despesa e o método de pagamento de cada morador.
- A qualquer altura os moradores ou senhorio devem poder consultar informações uns dos outros.
- Alguma informação como passwords, username devem poder ser alteradas a qualquer altura para qualquer conta.
- Apenas o senhorio pode alterar na totalidade, remover, ou criar novos moradores.
- Uma despesa por aprovar pode ser alterada a qualquer momento por qualquer morador nela envolvida para todos os envolvidos.
- Uma despesa gerada por moradores, ao ser alterada, gera uma nova despesa que requer aprovação de todos, mantendo-se em existência também a original.
- Uma despesa ao ser aprovada, elimina as que dela foram alteradas ou as originais caso seja uma despesa alterada e passa a estar ativa.
- Uma despesa ativa é arquivada apenas quando todos os pagamentos a efetuar por todos os moradores sejam efetuados (quer sejam feitos por outros moradores e ocorram dívidas).
- Existem duas modalidades de pagamento possíveis, pagamento único, ou pagamento à prestação.
- Quando uma despesa estiver ativa não pode ser alterada.
- Uma despesa pode ser recorrente, de modo a que especificando o intervalo de tempo em que ocorre e durante quanto tempo recorrerá, seja criada uma nova despesa idêntica no fim de cada período de tempo até ao fim da sua duração. Caso a despesa seja dos moradores, a despesa criada no fim de cada intervalo de tempo começa por estar por aprovar, neste modo pode-se alterar os valores a pagar para algo como, por exemplo, um serviço de subscrição mas apenas no fim de um pagamento da subscrição. Caso a despesa seja do senhorio, o senhorio pode alterar os valores a pagar caso ainda não haja nenhum pagamento efetuado.
- Uma despesa pode ser ocasional, de modo a que ao criar uma nova despesa com um nome idêntico esta automaticamente seja preenchida com os valores da despesa de nome idêntico arquivada mais recentemente, sendo porém possível alterar os seus valores.
- Uma despesa pode ser extraordinária em que é preciso especificar todos os seus parâmetros na inteiridade.

2.2 Diagrama de Use Cases

Após o esboço do **modelo de domínio**, procedemos ao desenvolvimento do **diagrama de use cases**, com o intuito de estabelecer a relação entre a execução de qualquer uma das ações e os diferentes resultados.

Como tal, e consoante o tipo de utilizador (que requere, obrigatoriamente, autenticação), existem diversas tarefas, expostas no **diagrama de use cases**, que tal utilizador pode efectuar.

Tentámos também detalhar o possível "diálogo" entre os atores envolvidos e o sistema em geral com algum pormenor, resultando em especificações para cada um destes use cases.

Assim, estas diferentes possibilidades de execução das funcionalidades, são visíveis no **diagrama de use cases** abaixo enunciado.

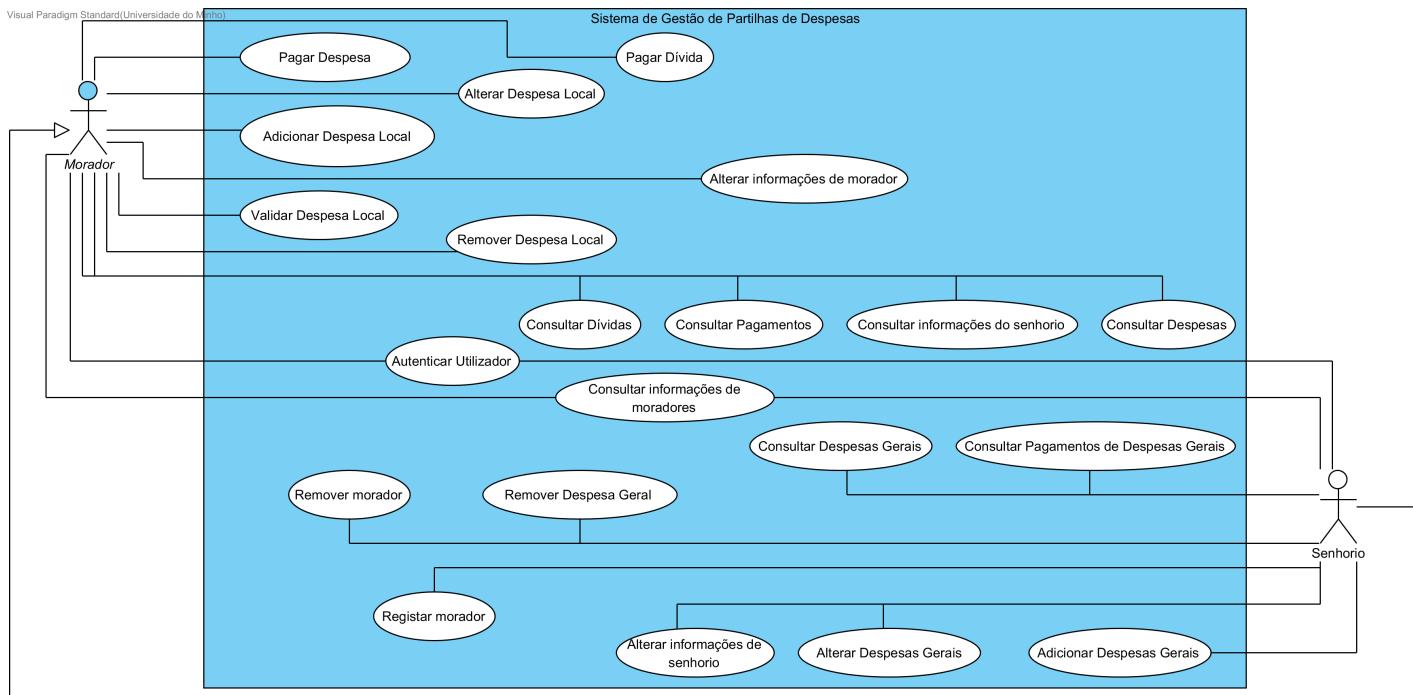


Figura 2.2: Diagrama de Use Cases.

2.3 Interface com o Utilizador

Implementada uma versão bastante básica da interface com o utilizador, conseguimos prototipar alguns dos métodos mais simples, dado que, devido à vasta dimensão do projecto, não nos foi possível a transformação de todos os métodos em código Java (interagindo, também, com a **Base de Dados local**). Como tal, foi-nos possível a implementação dos seguintes métodos:

1. atualizaSenhorio()
2. existeUtilizador()
3. buscaListaMoradores()
4. registrarDespesaLocal()
5. registrarDespesaGeral()
6. registrarMorador()
7. registrarSenhorio()

8. validaLogin()

Que, em si, interagem com a base de dados composta pelas seguintes entidades:

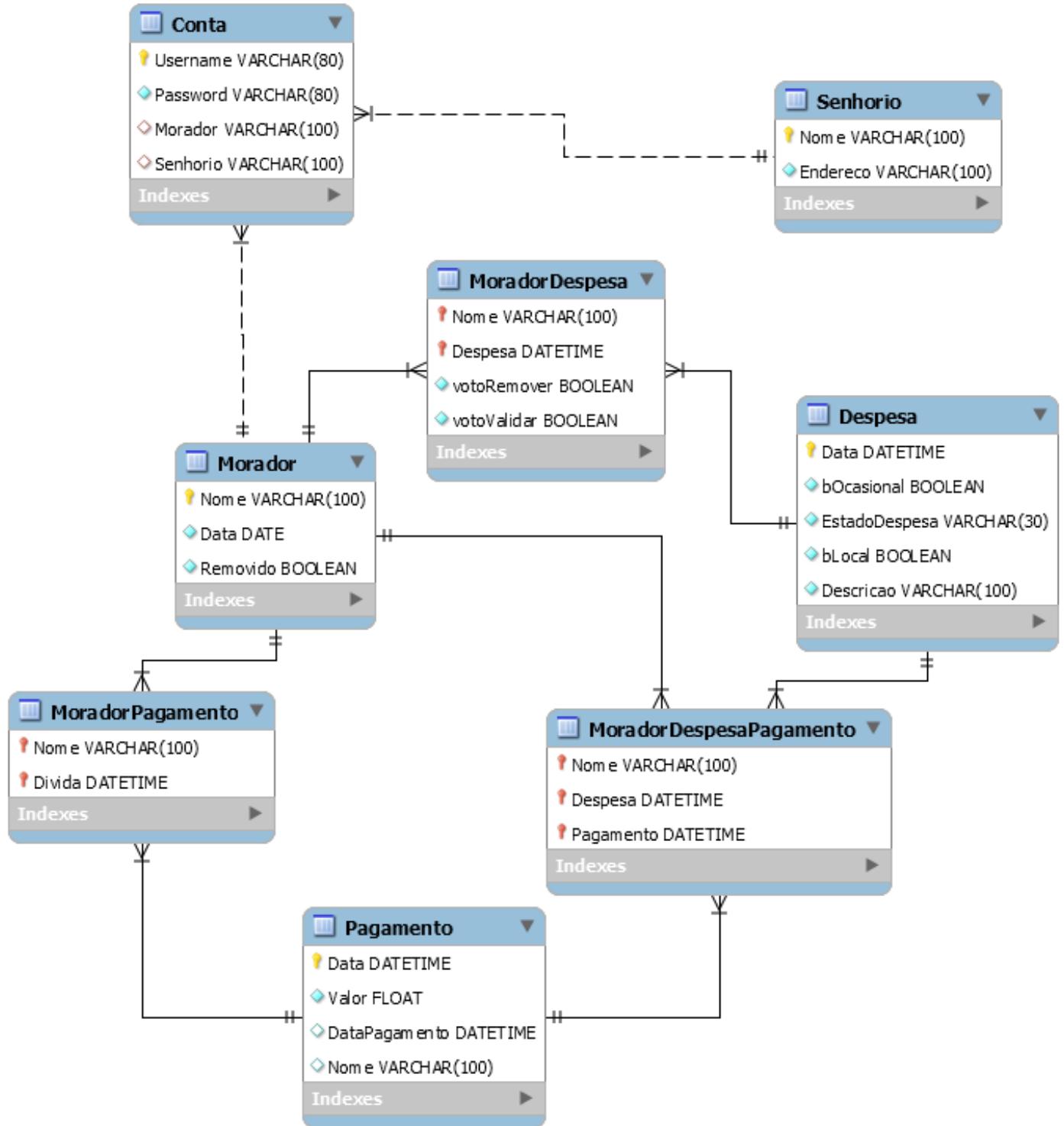


Figura 2.3: Base de Dados implementada.

Porém partir das necessidades destes use cases definimos em diagramas de estado o comportamento de uma possível interface com o utilizador, em anexo.

Capítulo 3

Arquitetura Lógica

O processo de desenvolvimento de uma arquitetura para a lógica de negócio do nosso **sistema de gestão de despesas** passou por 3 fases: uma **primeira arquitetura lógica**, com um diagrama de classes simples e um conjunto de diagramas de sequência de sistema, uma **segunda arquitetura lógica** com subsistemas identificados e refinamento destes diagramas de sequência para abrangerem os subsistemas e uma terceira e **final arquitetura lógica** com persistência de dados através da adição de DAOs e a substituição de alguns dos componentes nos diagramas de sequência por estes mesmos.

3.1 Arquitetura Lógica I

Para uma primeira arquitetura lógica tratámos de definir um ponto de acesso à lógica de negócio capaz de efetuar as operações necessárias à satisfação dos use cases, o Facade, operações que derivámos em diagramas de sequência de sistema da especificação dos use cases.

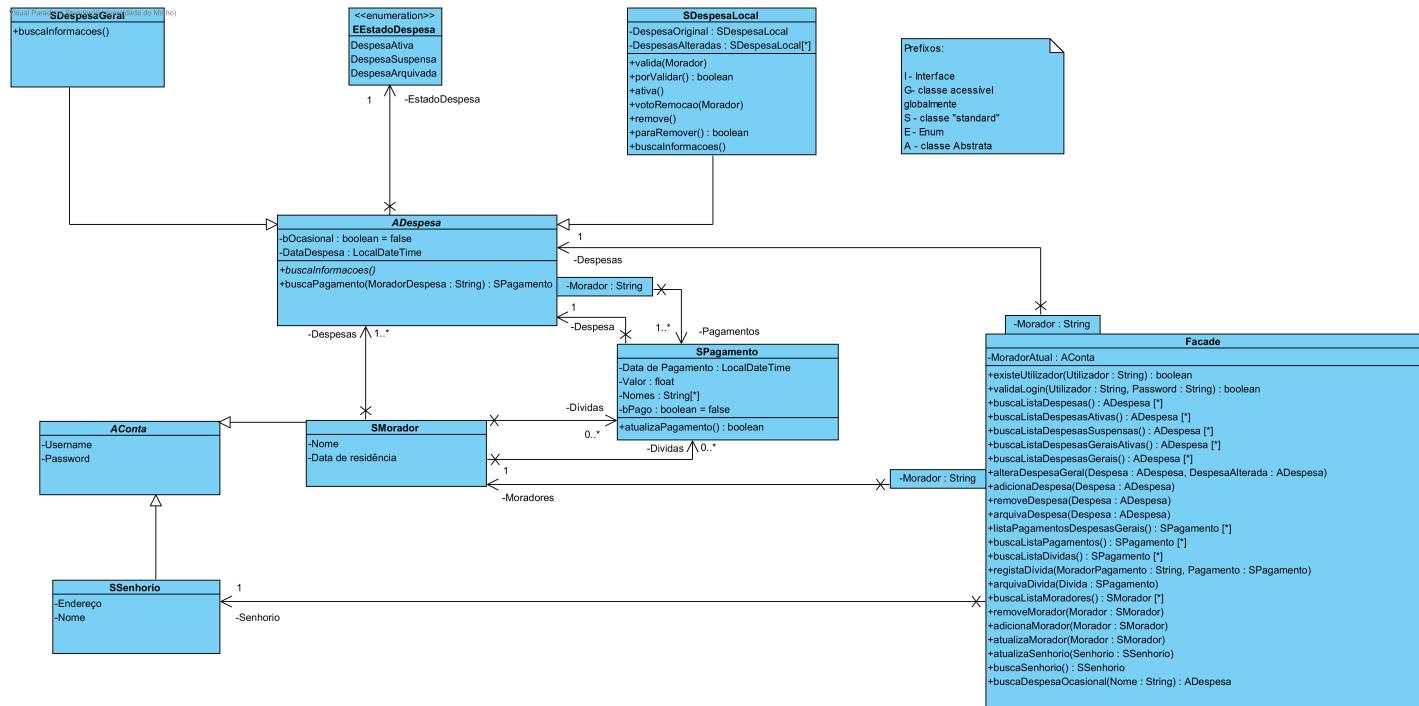


Figura 3.1: Diagrama de Classes.

3.2 Arquitetura Lógica II

Para esta segunda arquitetura lógica tratámos de definir os principais subsistemas do nosso programa que potencialmente poderiam ser componentizados, e de refinar os nossos diagramas de sequência de sistema para incluir os subsistemas.

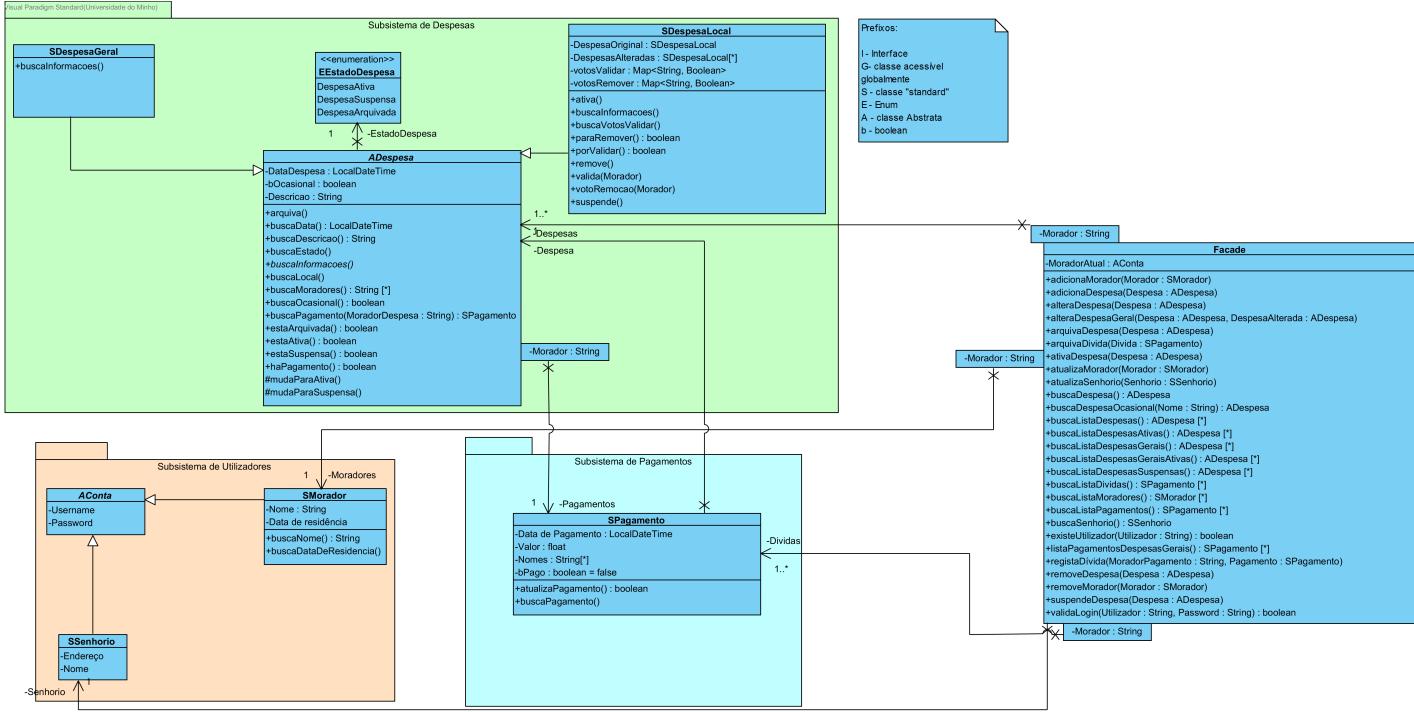


Figura 3.2: Diagrama de Classes com subsistemas.

Acabámos, **por falta de tempo**, de dar algum acesso "extra" à *interface* de utilizador às classes destes subsistemas, mas sem **nunca violar o encapsulamento**. Para tal, a *interface* realiza operações unica e somente em clones de instâncias de classes nestes subsistemas, neste caso ainda sem persistência em disco, e única e somente as pode obter através da única classe a que tem acesso direto, o **Facade**. Deste modo, o **Facade** pode extraír um clone de uma **Despesa** e fornecê-lo à *interface* e a *interface* chamar métodos de modificação deste clone e pedir ao **Facade** para atualizar a **Despesa** de facto. Detalhámos este comportamento num diagrama de pacotes considerando apenas estas duas camadas. Isto permitiu-nos evitar uma proliferação de novos métodos e muitos métodos verbosos pelo **Facade** e subsistemas da lógica de negócio.

De seguida tratámos de modelar também a implementação de alguns dos métodos mais relevantes da API exposta pelo **Facade**, em anexo.

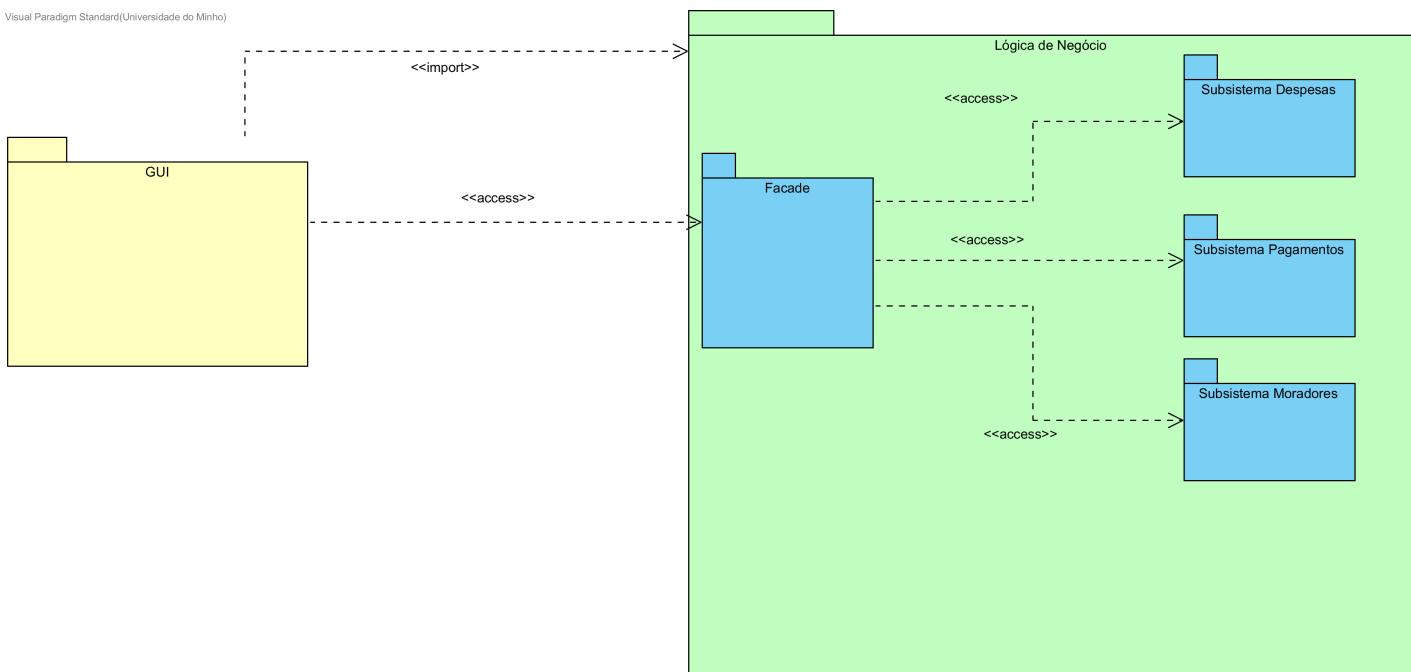


Figura 3.3: Diagrama de Pacotes em Duas Camadas.

3.3 Arquitetura Lógica Final

Para esta terceira e final arquitetura lógica tratámos de definir DAOs que nos permitissem efetuar a persistência em disco de toda a informação necessária ao funcionamento do sistema e definir a maneira como estes interagem com os subsistemas. Decidimos implementar a interface Map nos três DAOs que identificámos, para permitir a fácil incorporação nos diagramas de sequência de implementação já definidos na arquitetura anterior. Decidimos ter apenas estes três DAOs pois pareceu-nos a escolha mais razoável para não complicar demasiado a implementação dos métodos com grandes consultas a base de dados e simultaneamente evitando trazer um conjunto desmedido de informação da base de dados uma vez, nem em demasia, nem em falta.

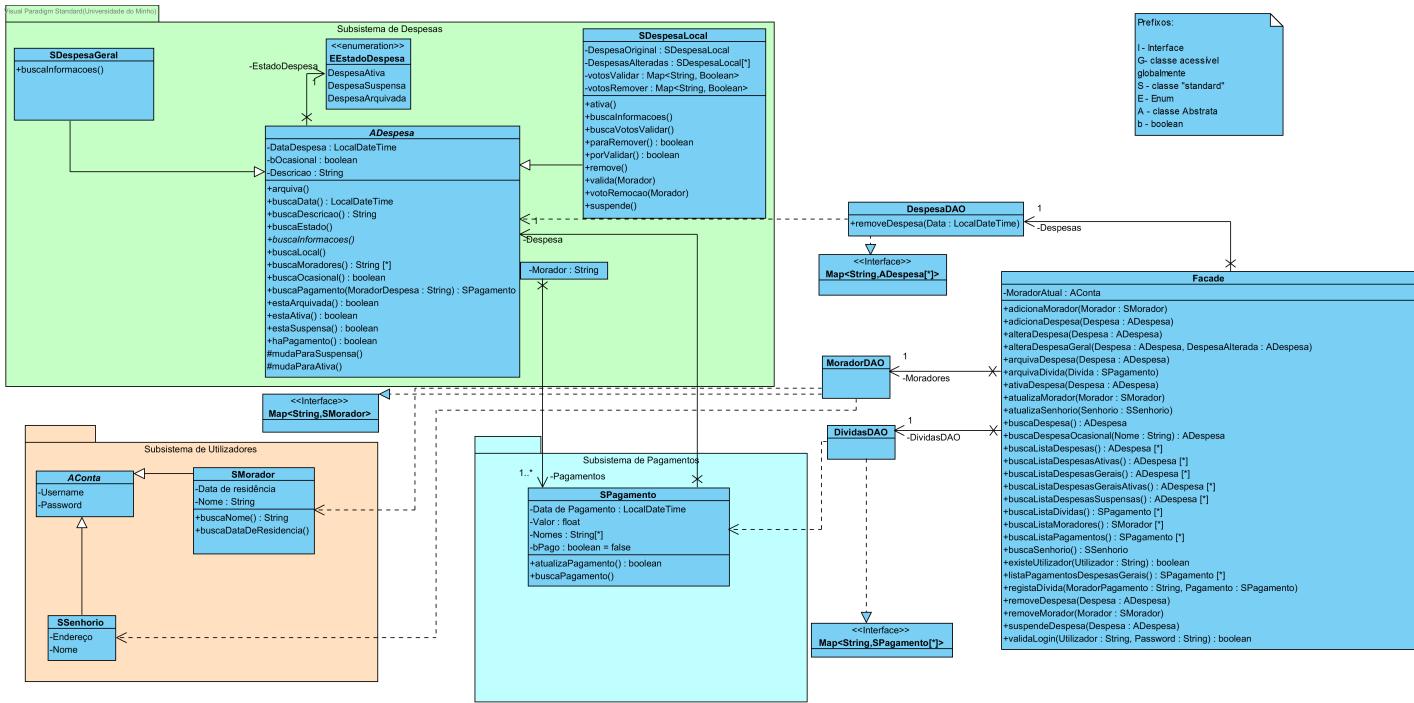


Figura 3.4: Diagrama de Classes com DAOs.

Nesta altura, e quase concluída a modelação, modelámos também uma proposta final de arquitetura em termos de pacotes e em termos de instalação.

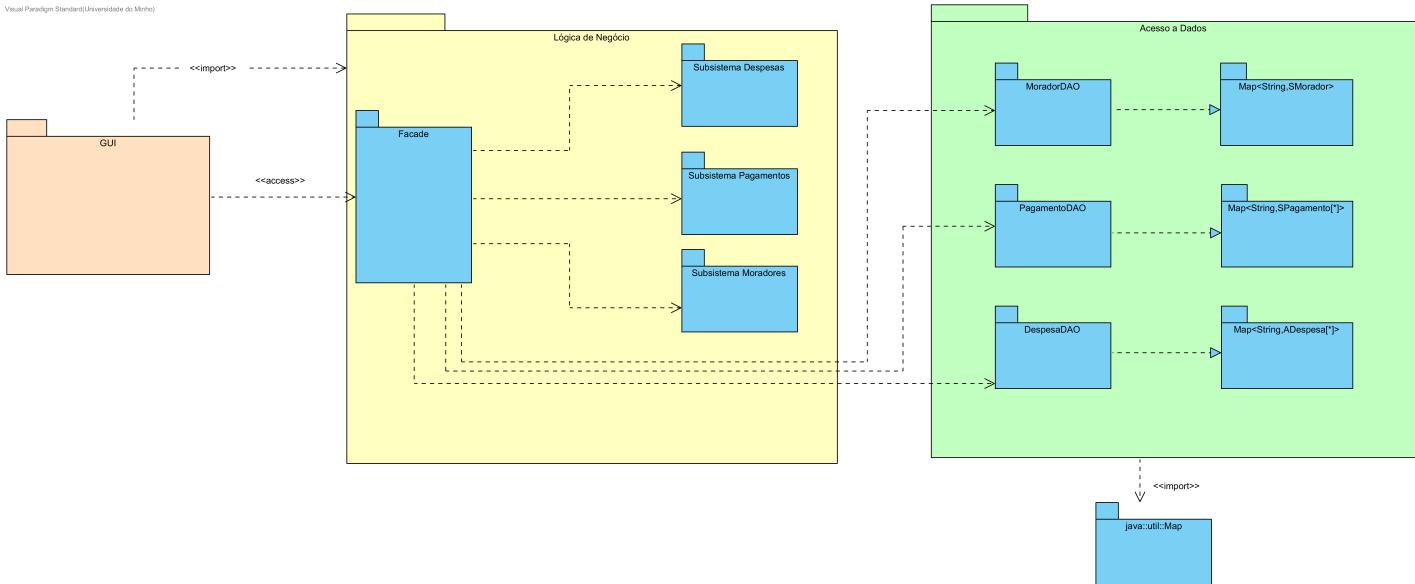


Figura 3.5: Diagrama de Pacotes em Três Camadas.

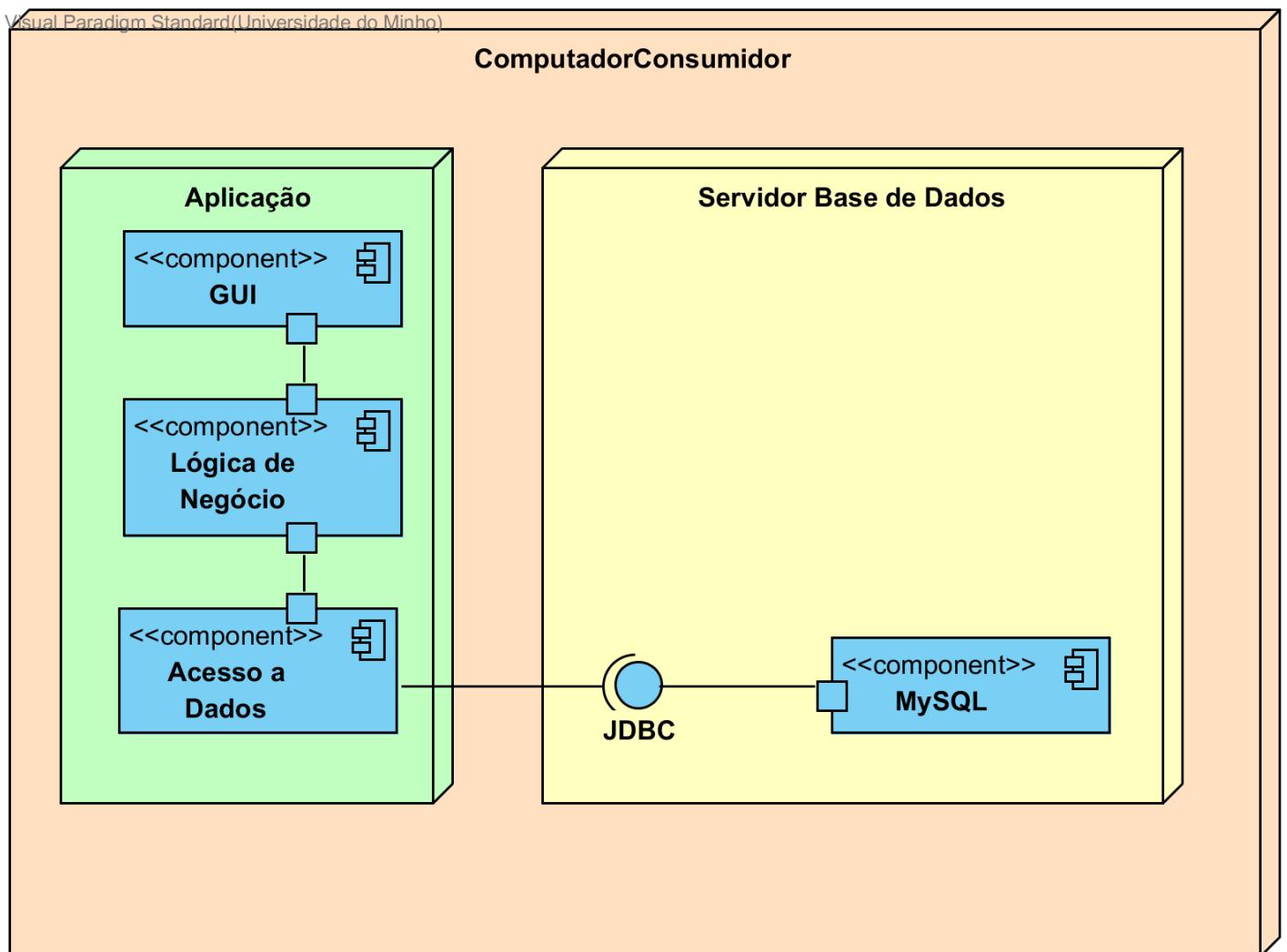


Figura 3.6: Diagrama de Instalação.

Capítulo 4

Conclusão

Atingidos os objetivos propostos para ambas as fases do projeto, é fácil perceber as vantagens de usar uma ferramenta como o **Visual Paradigm** na planificação das bases da aplicação. Deste modo, o desenvolvimento da aplicação é feito de um modo mais coerente, sendo detectados problemas aquando da modelação da aplicação em vez de serem apenas detectados na fase de implementação. Assim, é possível otimizar o processo de desenvolvimento de software devido ao tempo poupado e à melhor definição da vontade do cliente.

Ao ser feita uma planificação do desenvolvimento da aplicação, foi-nos permitida uma construção mais coesa e fundamentada, facilitando-nos, depois, a transformação em código. Foi também possível a constatação de que poderíamos utilizar qualquer linguagem para construir esta aplicação, visto que os diagramas expostos no **Visual Paradigm** assim o permitem, através de uma generalização do que haveria a ser feito.

Gostaríamos, porém, caso tivessemos tido a oportunidade, e numa perspectiva mais prática, de ter um sistema mais cómodo e que permitisse clientes a votar e propor alterações e despesas concorrentemente, mas por simplicidade e falta de tempo, decidimos em algo mais básico, num simples modelo sem concorrência, inteiramente contido numa máquina.

Por fim, concluímos que a utilidade do **Visual Paradigm** é extrema, de modo que a sua utilidade se reflecte na facilidade de manutenção e alteração de métodos, através de uma simples mudança de mensagens enviadas.

Anexos

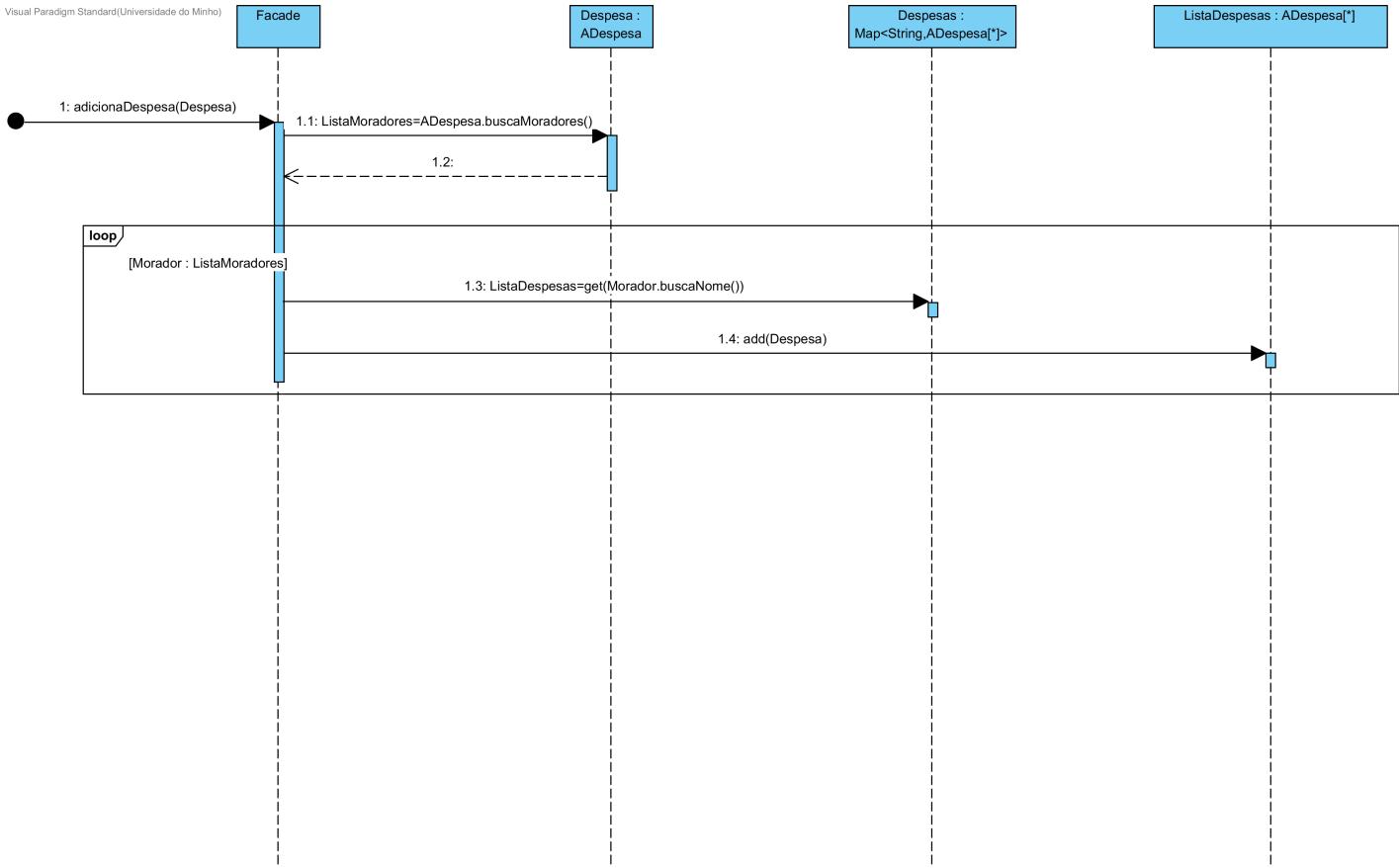


Figura 1: Diagrama de Implementação do método `adicionaDespesa(Despesa : ADespesa)`.

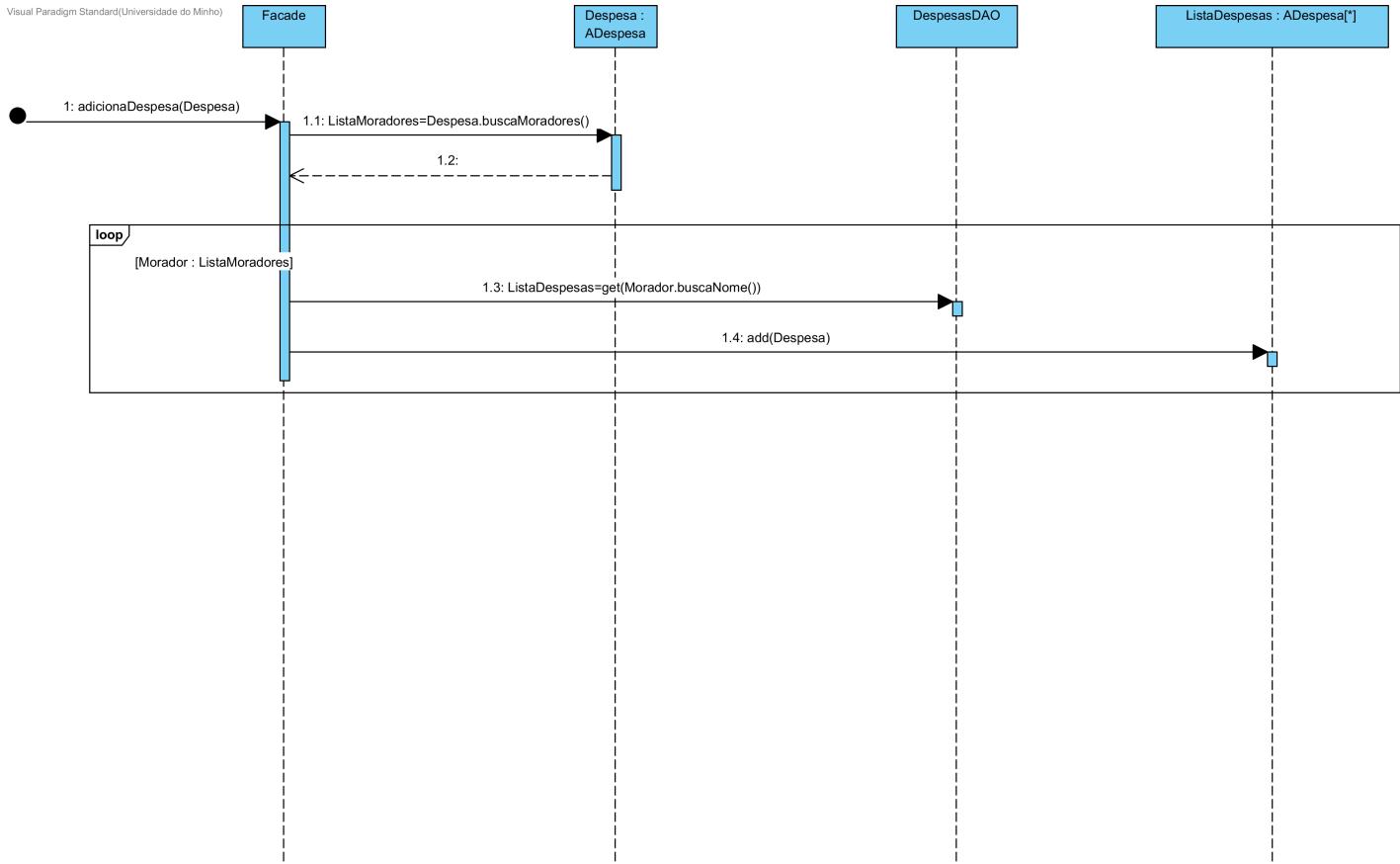


Figura 2: Diagrama de Implementação do método `adicionaDespesa(Despesa : ADespesa)` com DAOs.

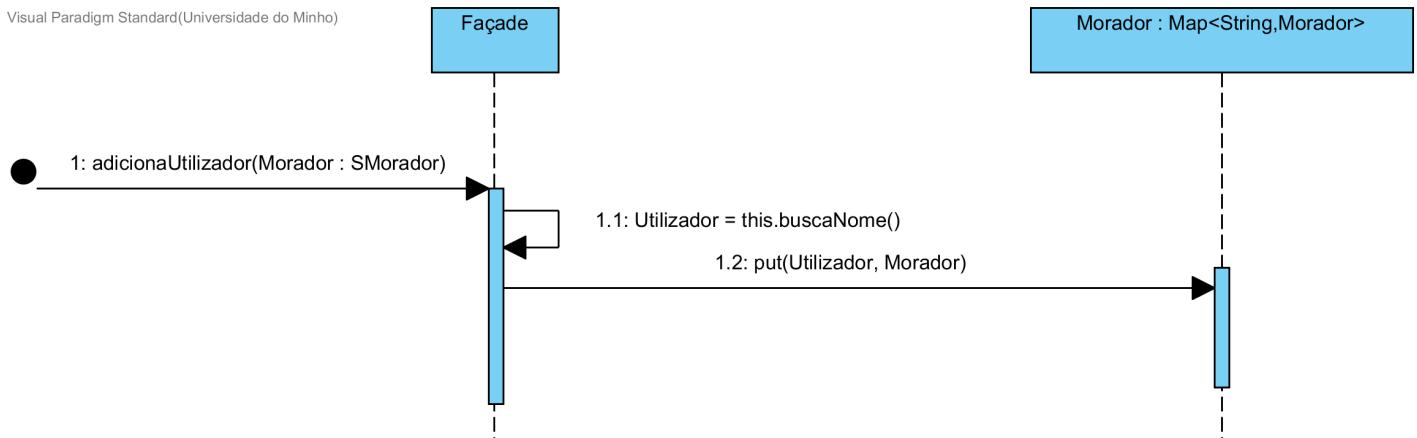


Figura 3: Diagrama de Implementação do método `adicionaMorador(Morador : SMorador)`.

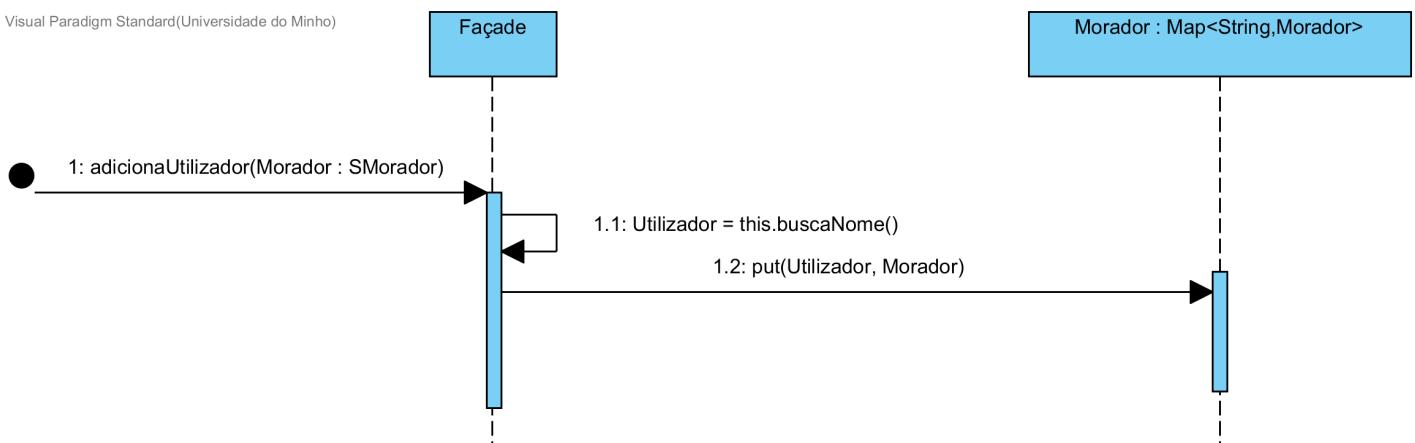


Figura 4: Diagrama de Implementação do método `adicionaMorador(Morador : SMorador)` com DAOs.

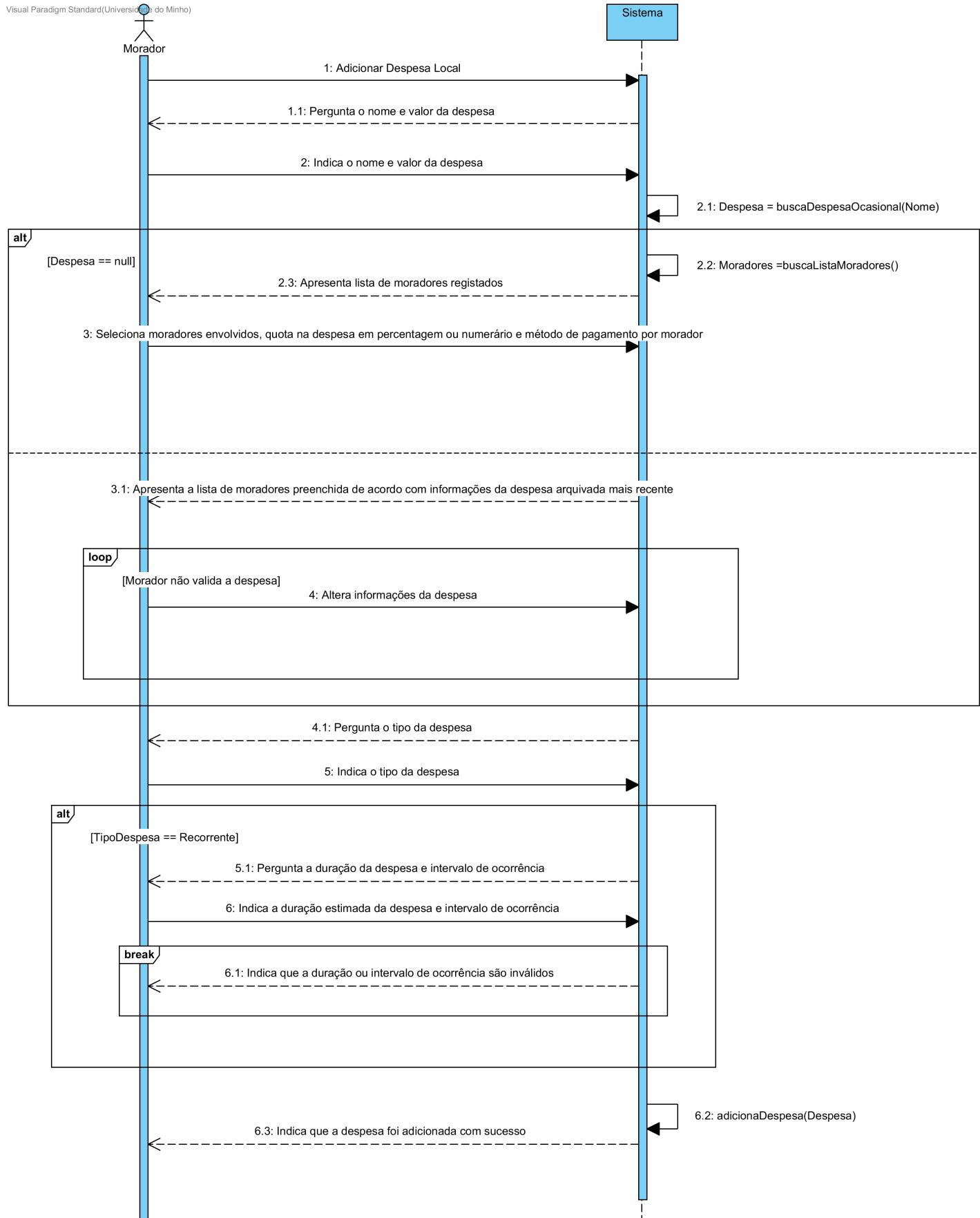


Figura 5: Diagrama de Sequência para Adicionar Despesa Local.

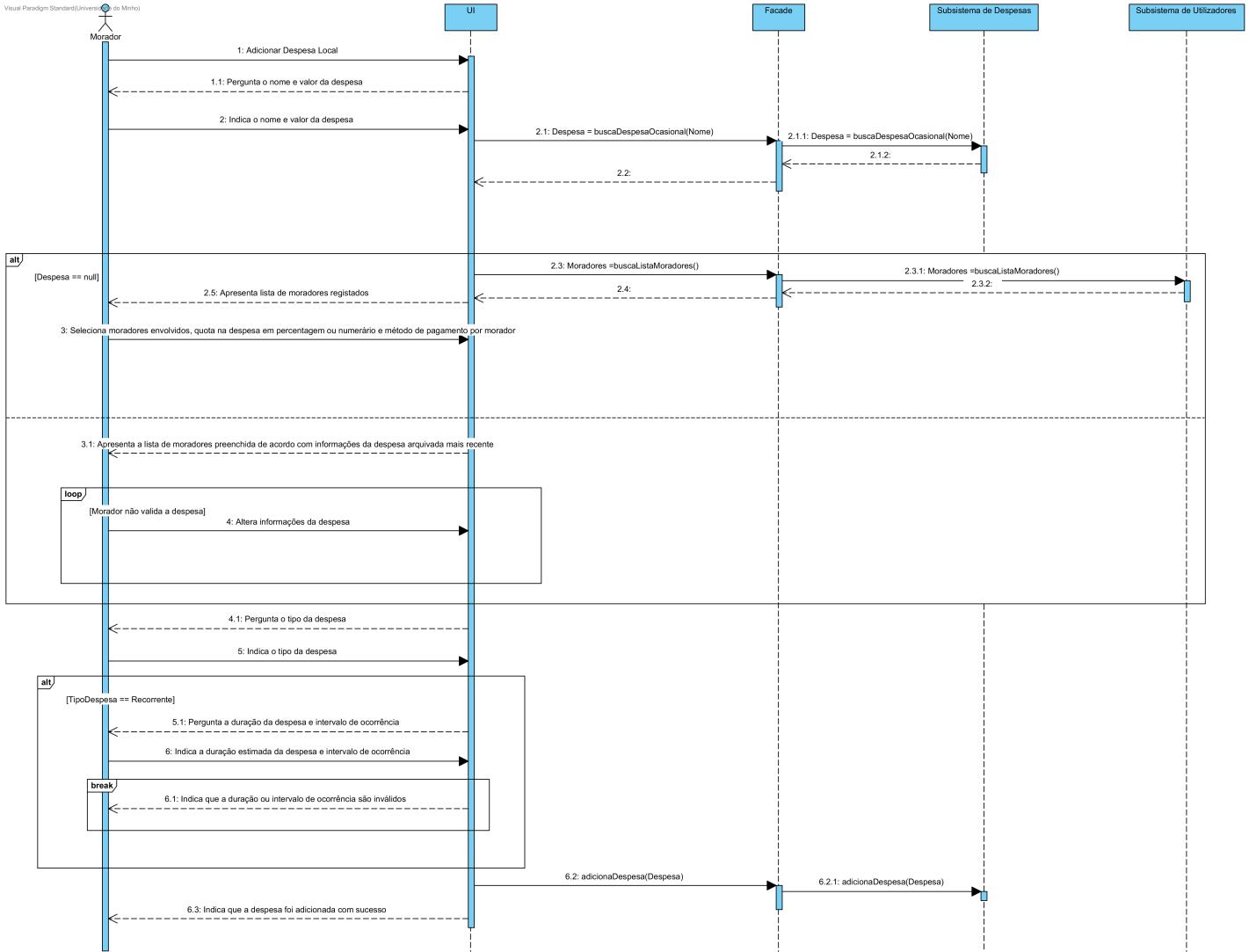


Figura 6: Diagrama de Sequência para Adicionar Despesa Local, com submenu.

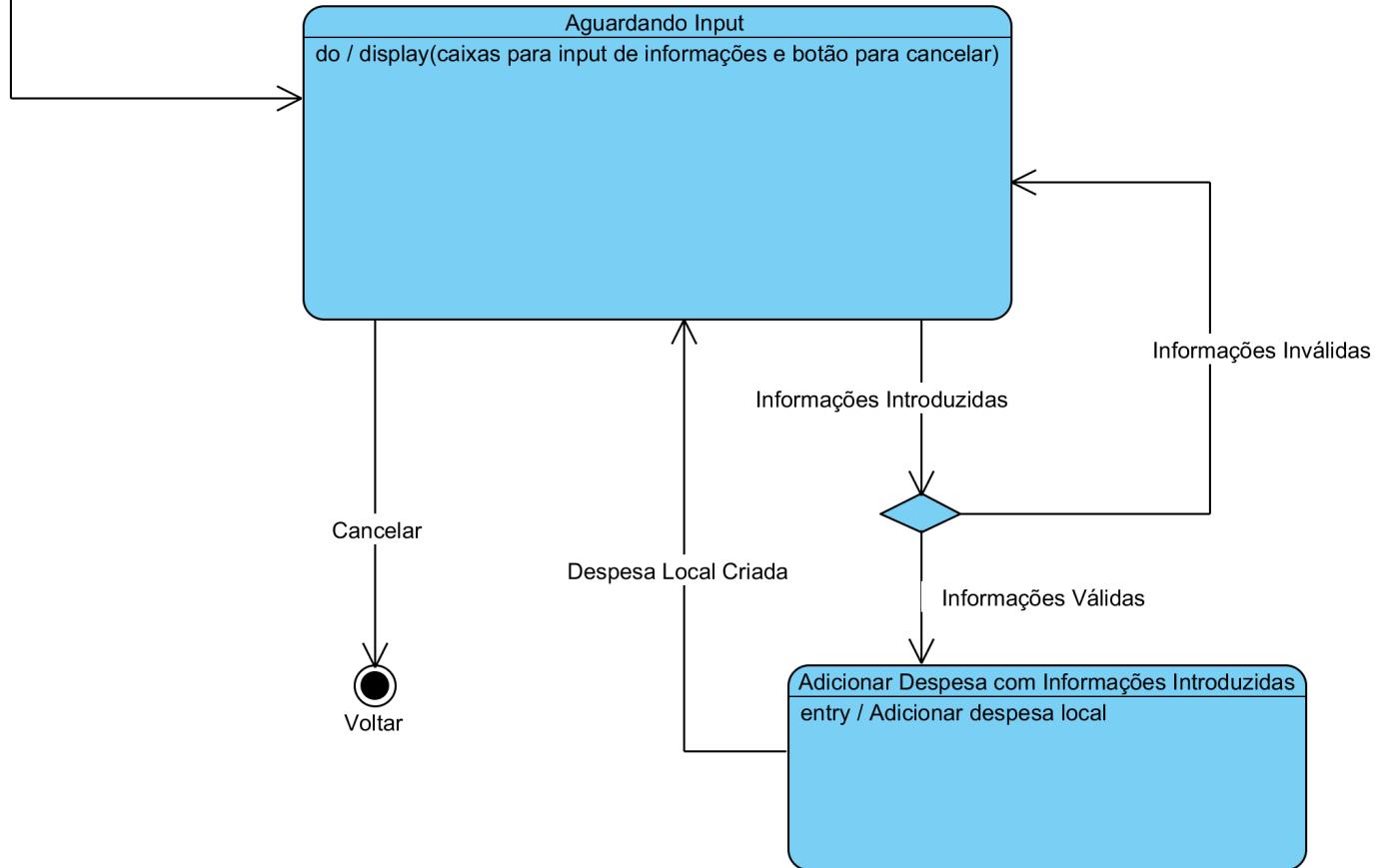


Figura 7: Submenu para Adicionar Despesa Local.

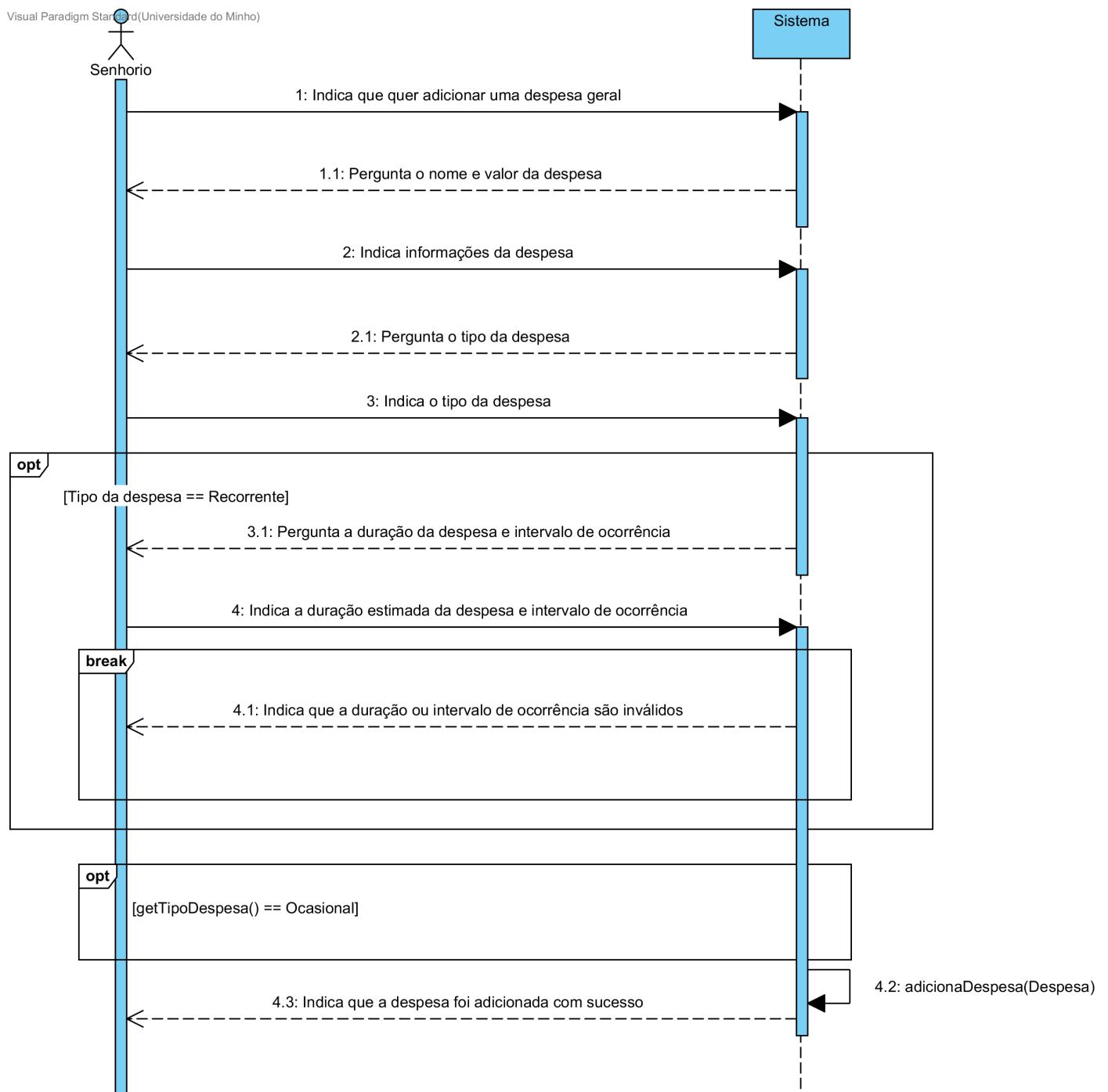


Figura 8: Diagrama de Sequência para Adicionar Despesas Gerais.

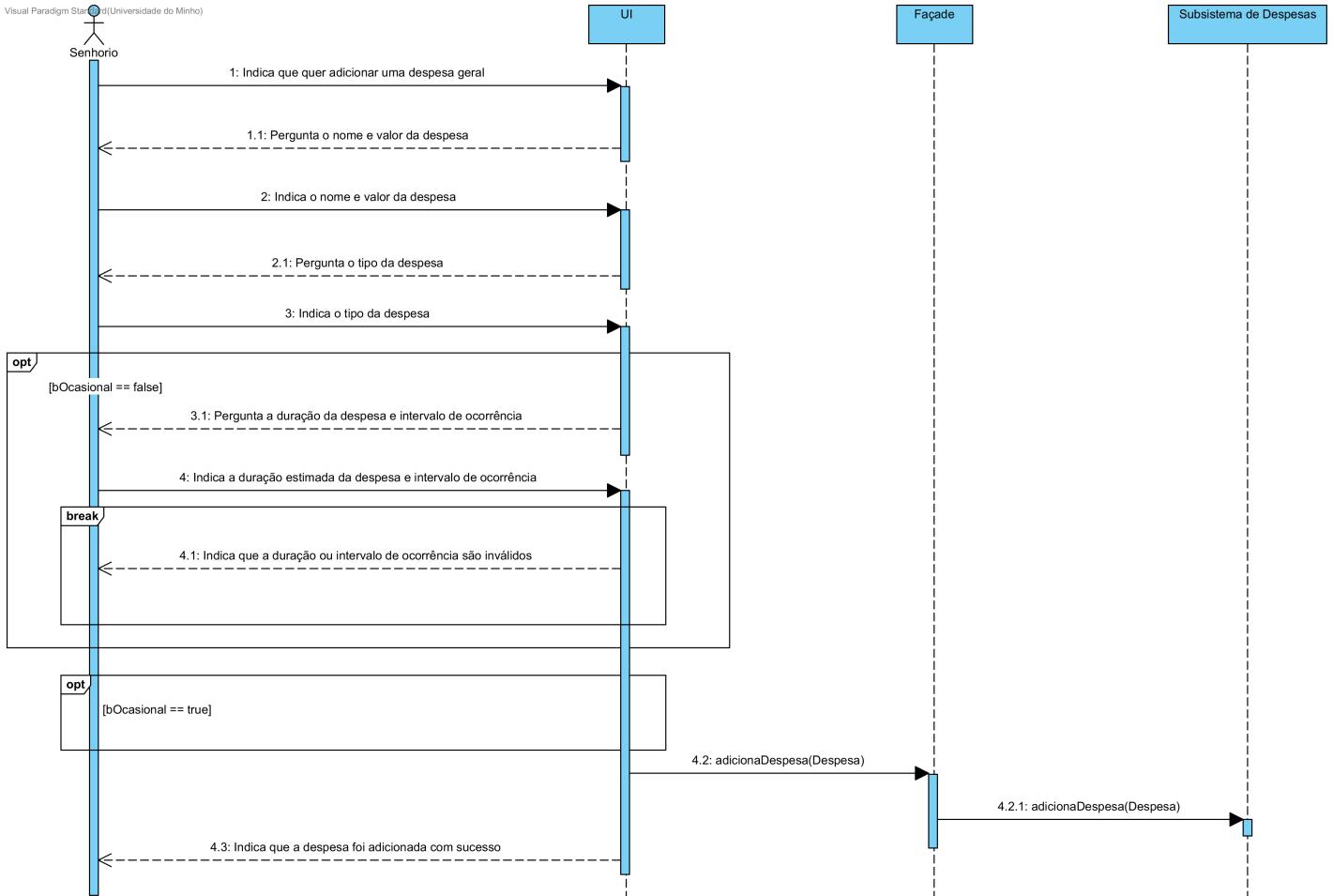


Figura 9: Diagrama de Sequência para Adicionar Despesas Gerais, com subsistemas.

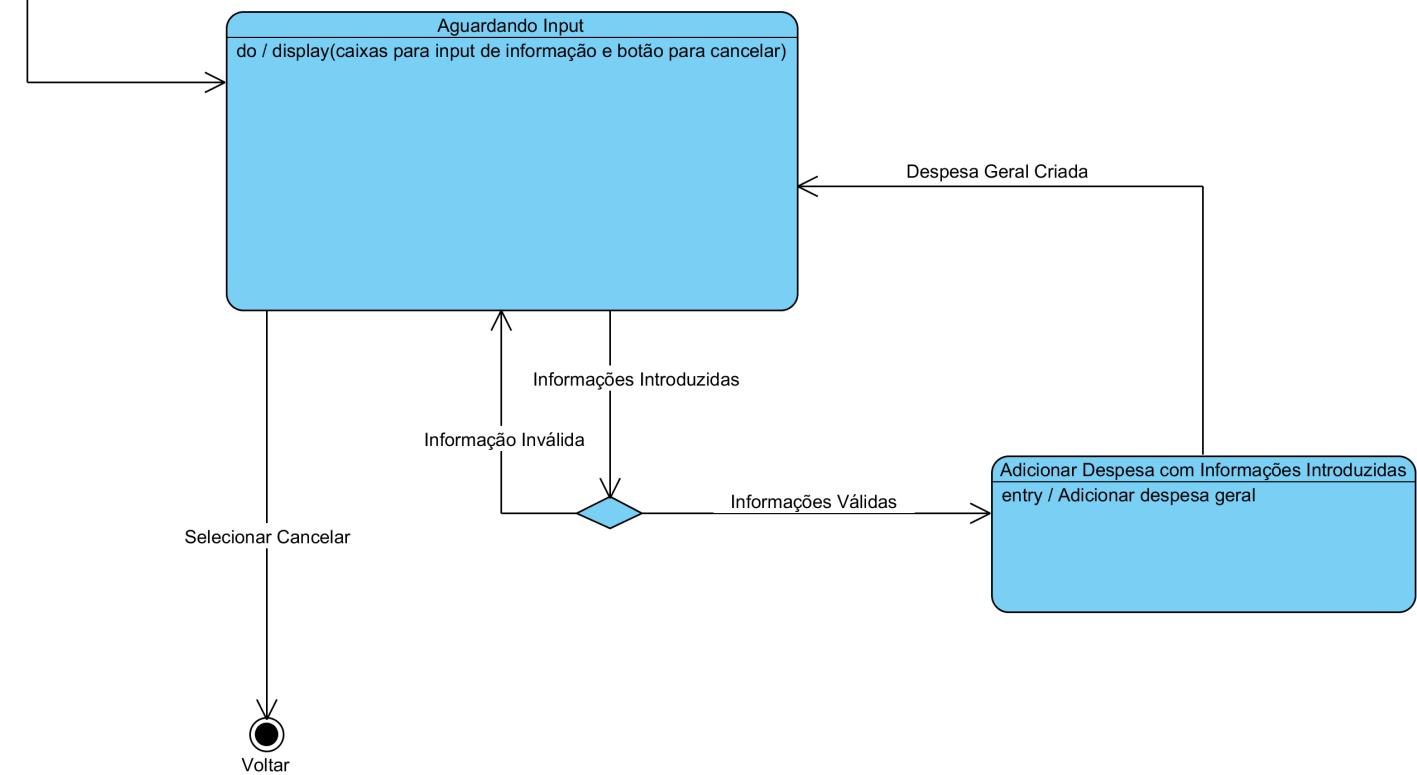
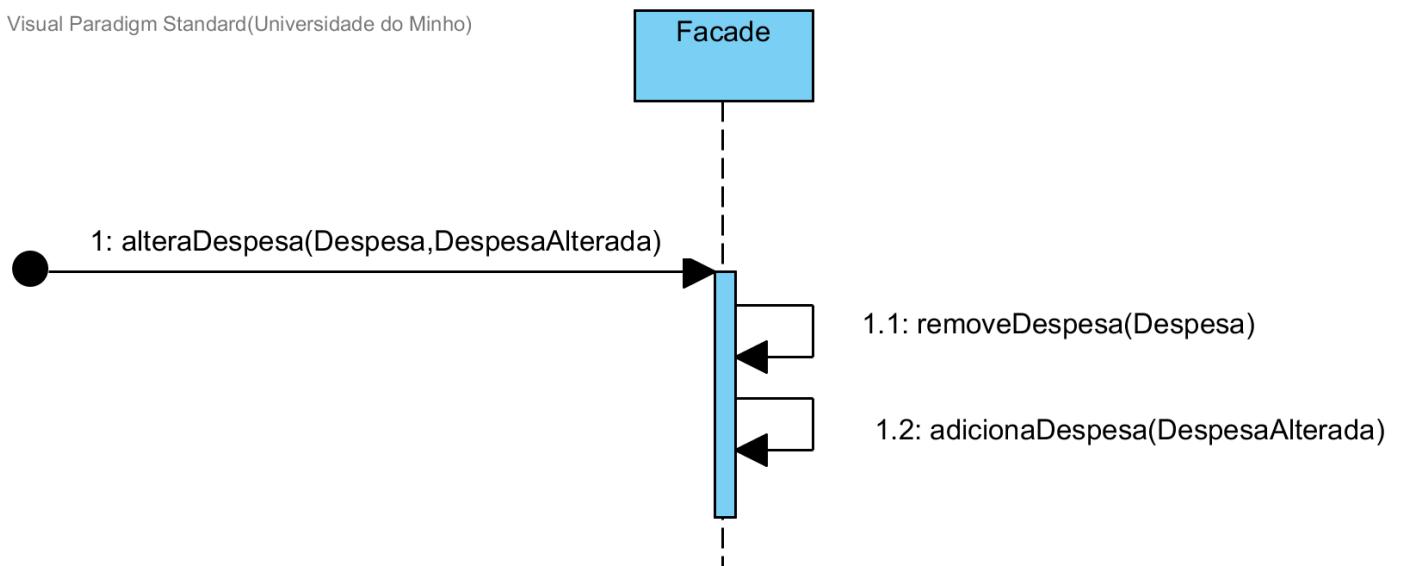


Figura 10: Submenu para Adicionar Despesas Gerais.

Figura 11: Diagrama de Implementação do método `alteraDespesa(Despesa : ADespesa, DespesaAlterada : ADespesa)`.

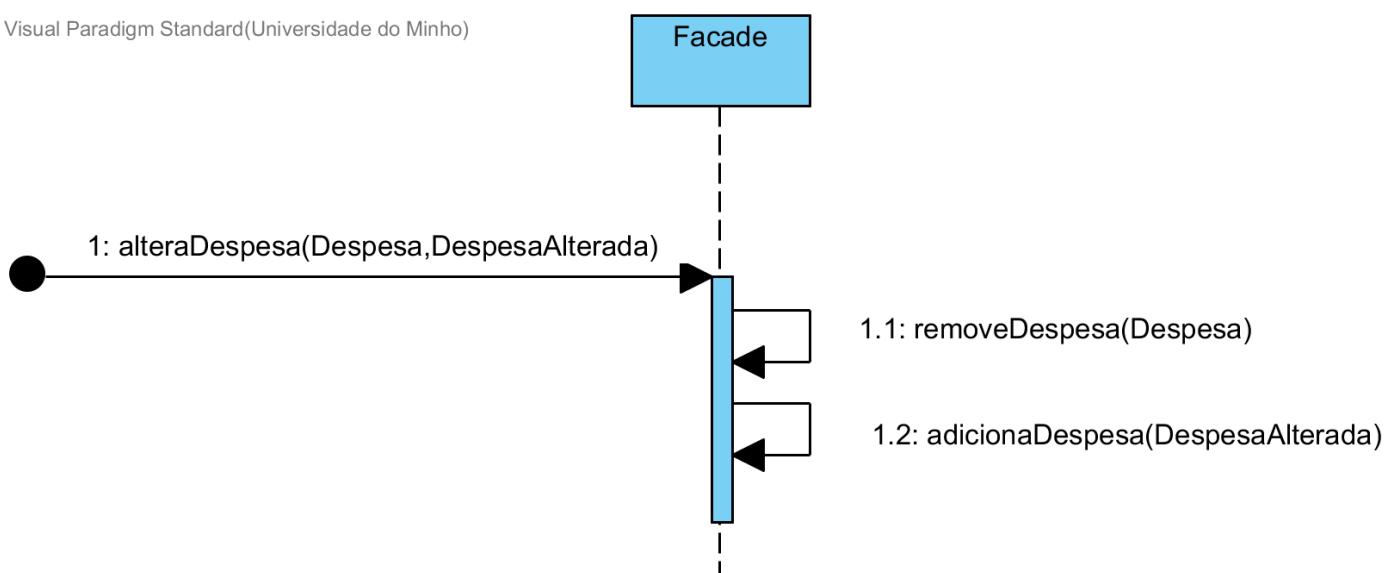


Figura 12: Diagrama de Implementação do método `alteraDespesa(Despesa : ADespesa, DespesaAlterada : ADespesa)` com DAOs.

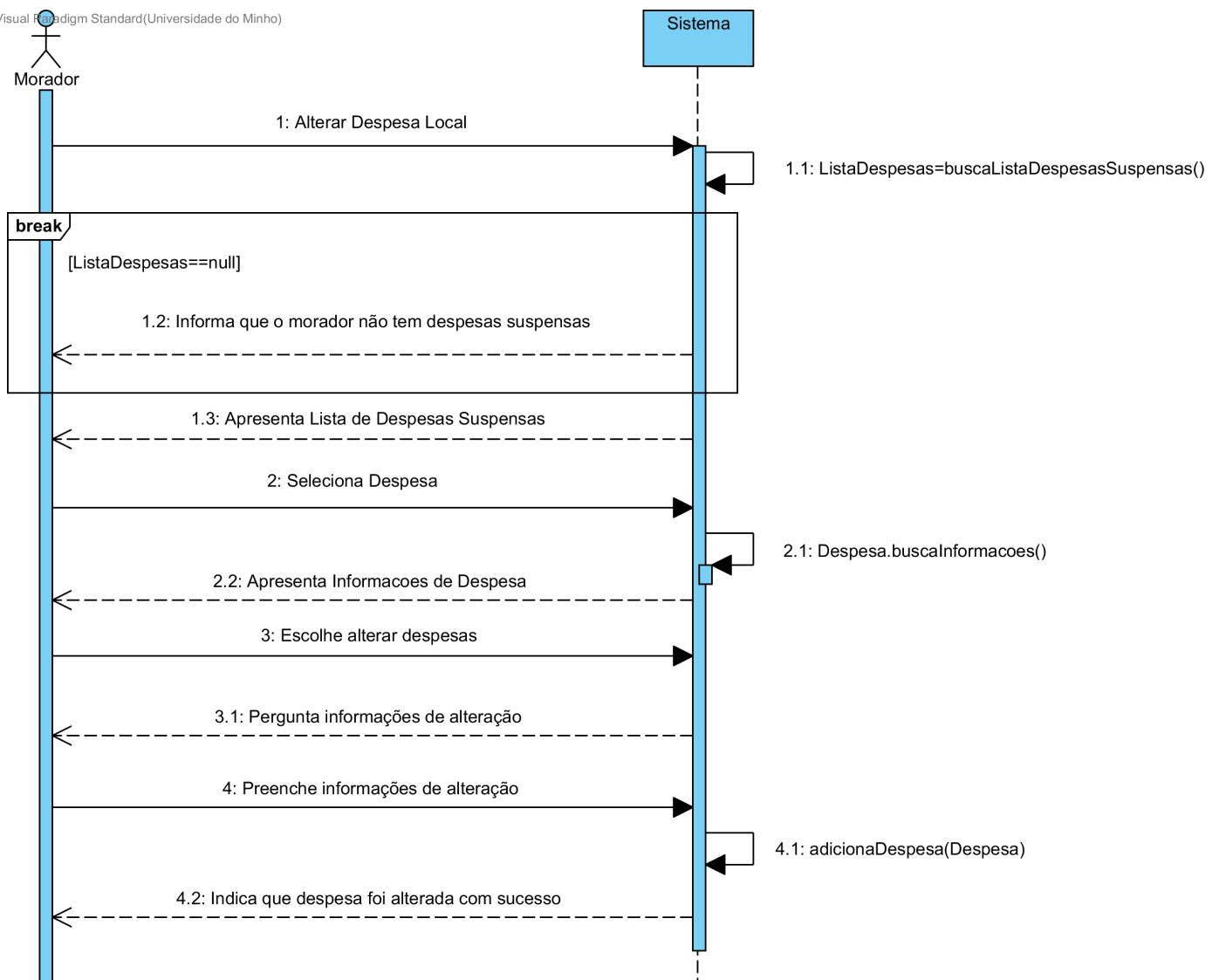


Figura 13: Diagrama de Sequência para Alterar Despesa Local.

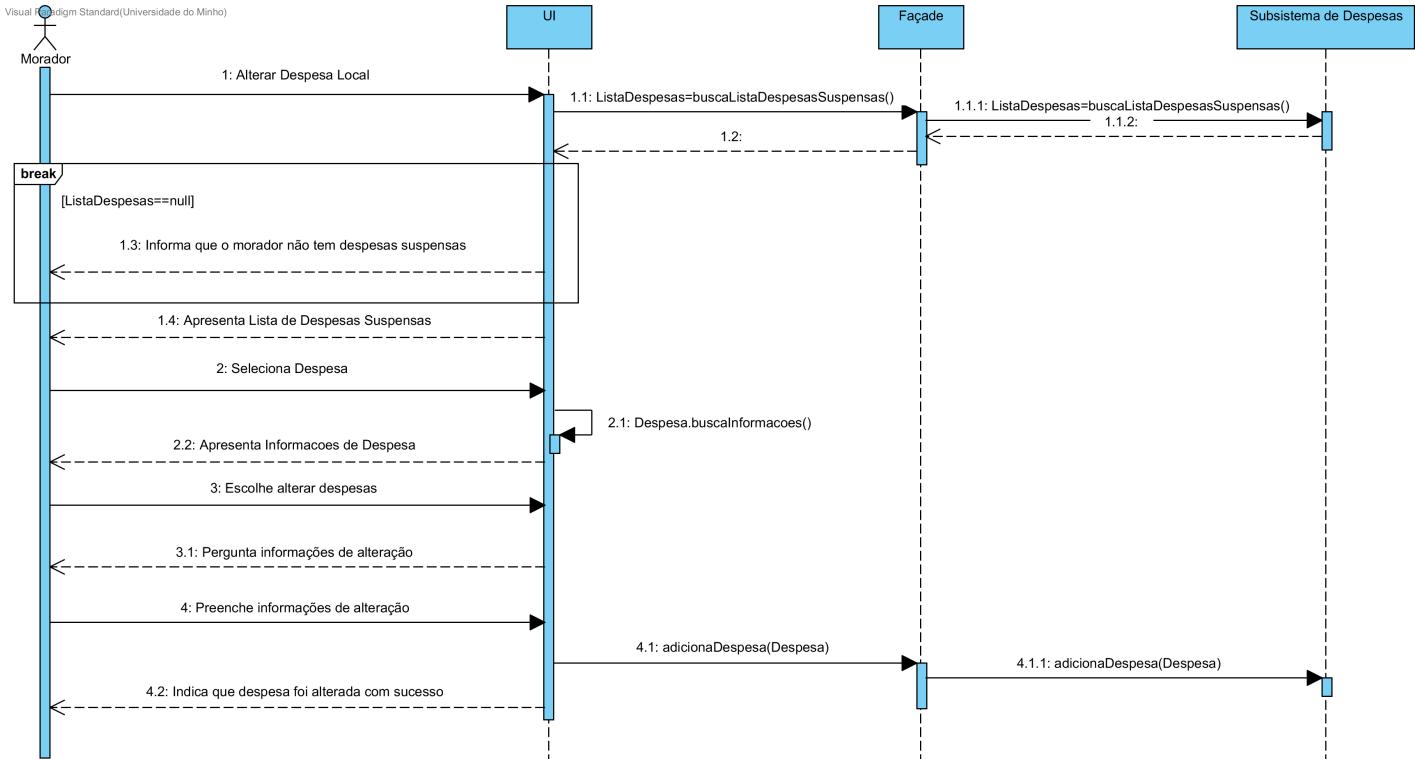


Figura 14: Diagrama de Sequência para Alterar Despesa Local, com subsistemas.

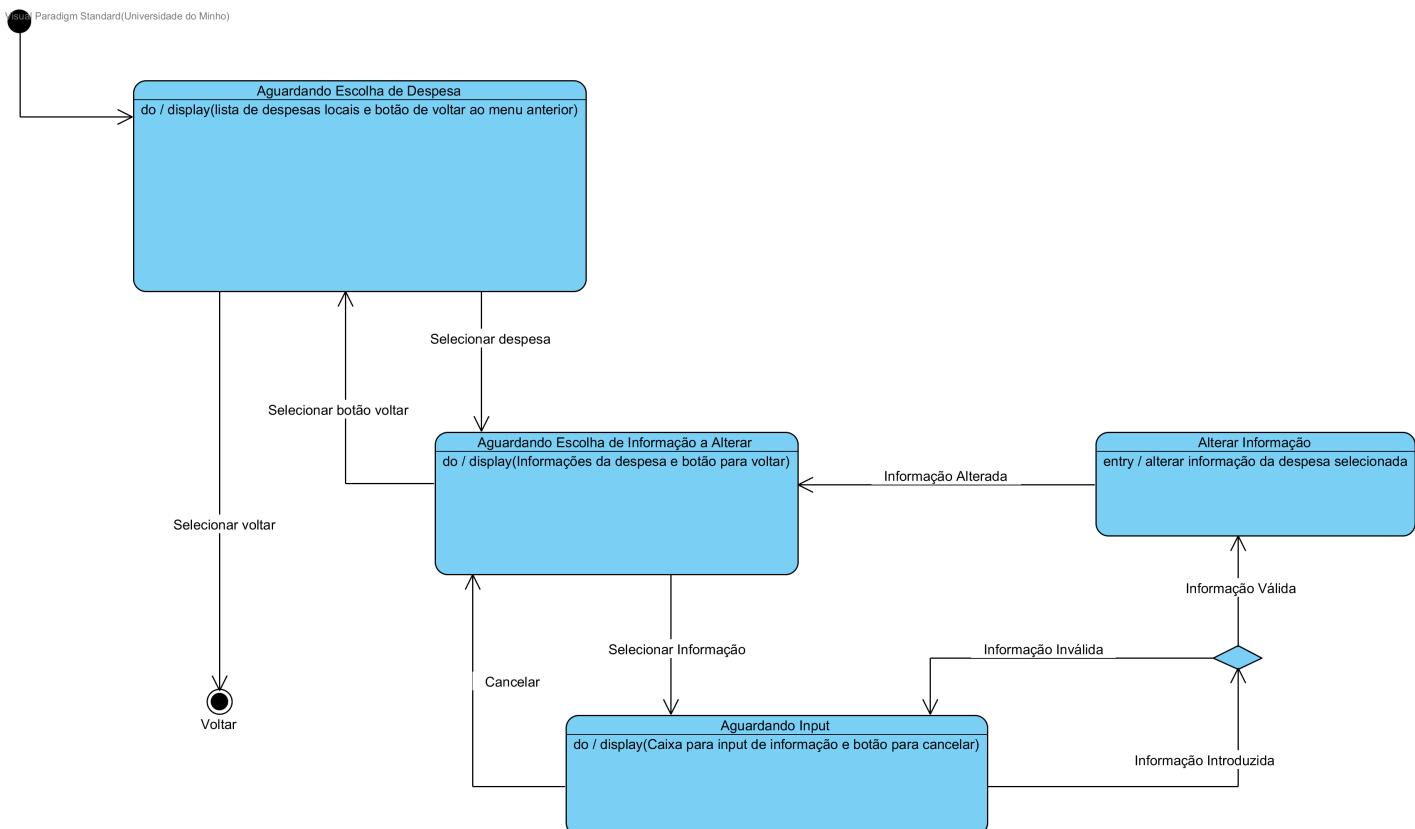


Figura 15: Submenu para Alterar Despesa Local.

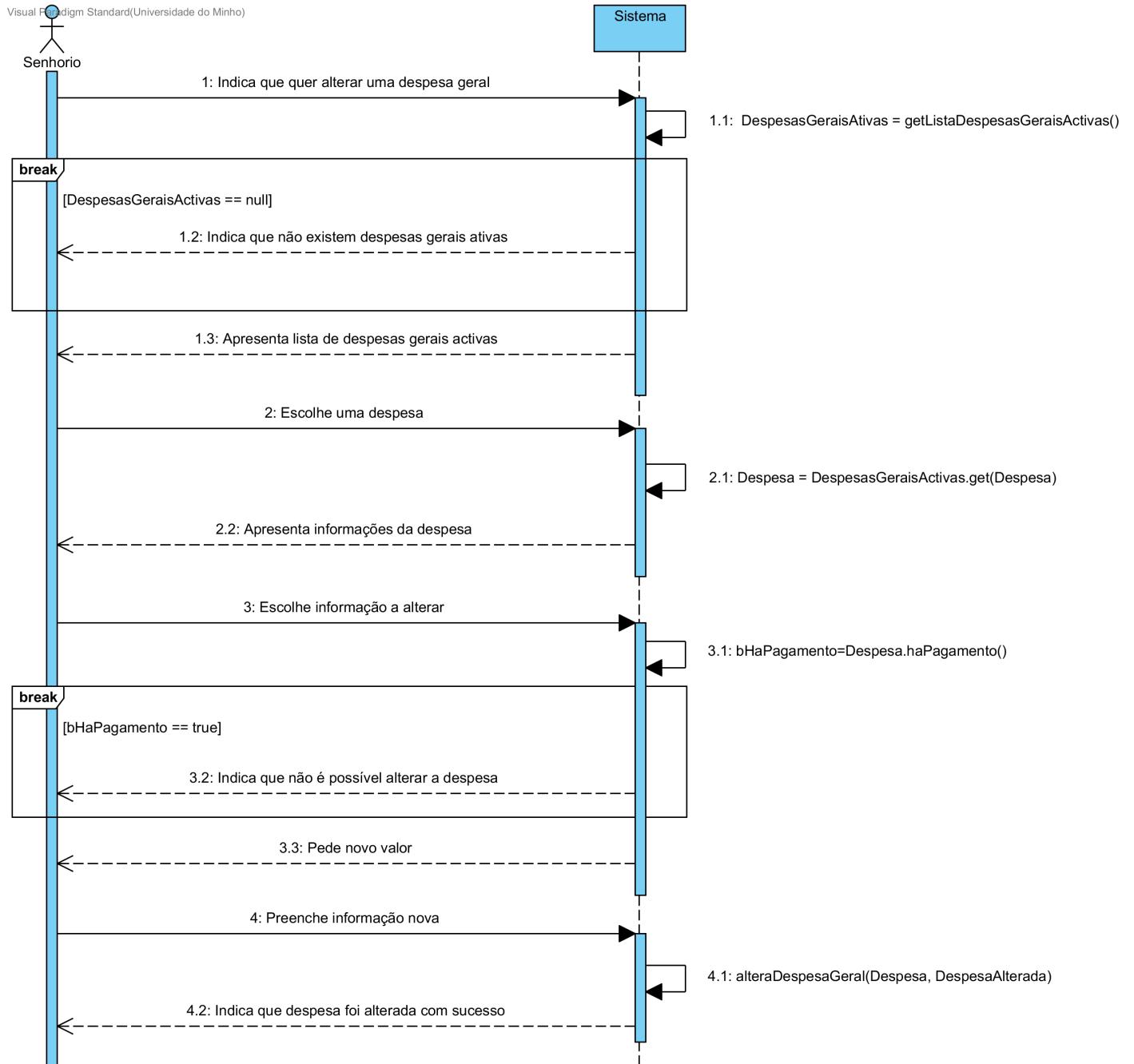


Figura 16: Diagrama de Sequência para Alterar Despesas Gerais.

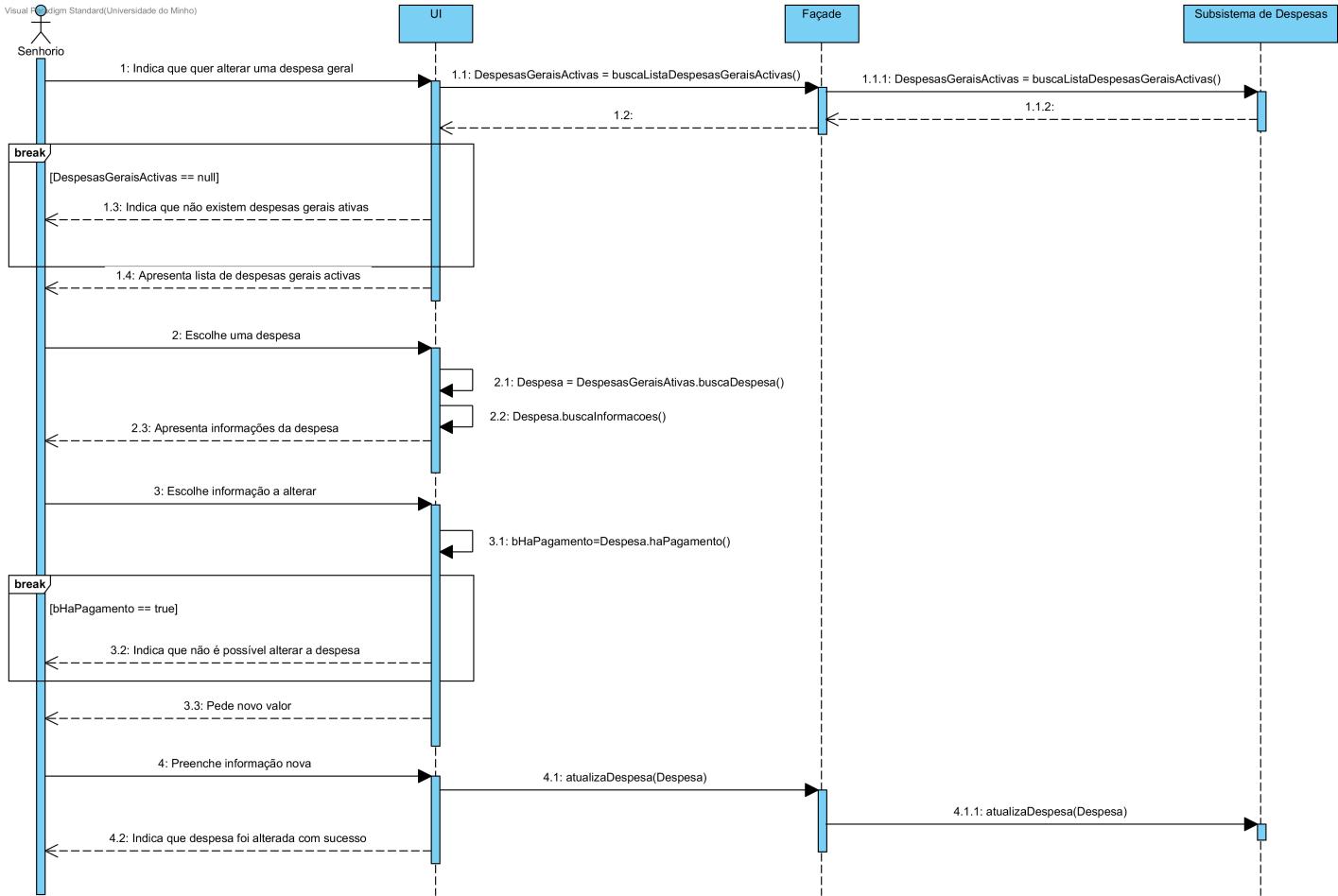


Figura 17: Diagrama de Sequência para Alterar Despesas Gerais, com subsistemas.

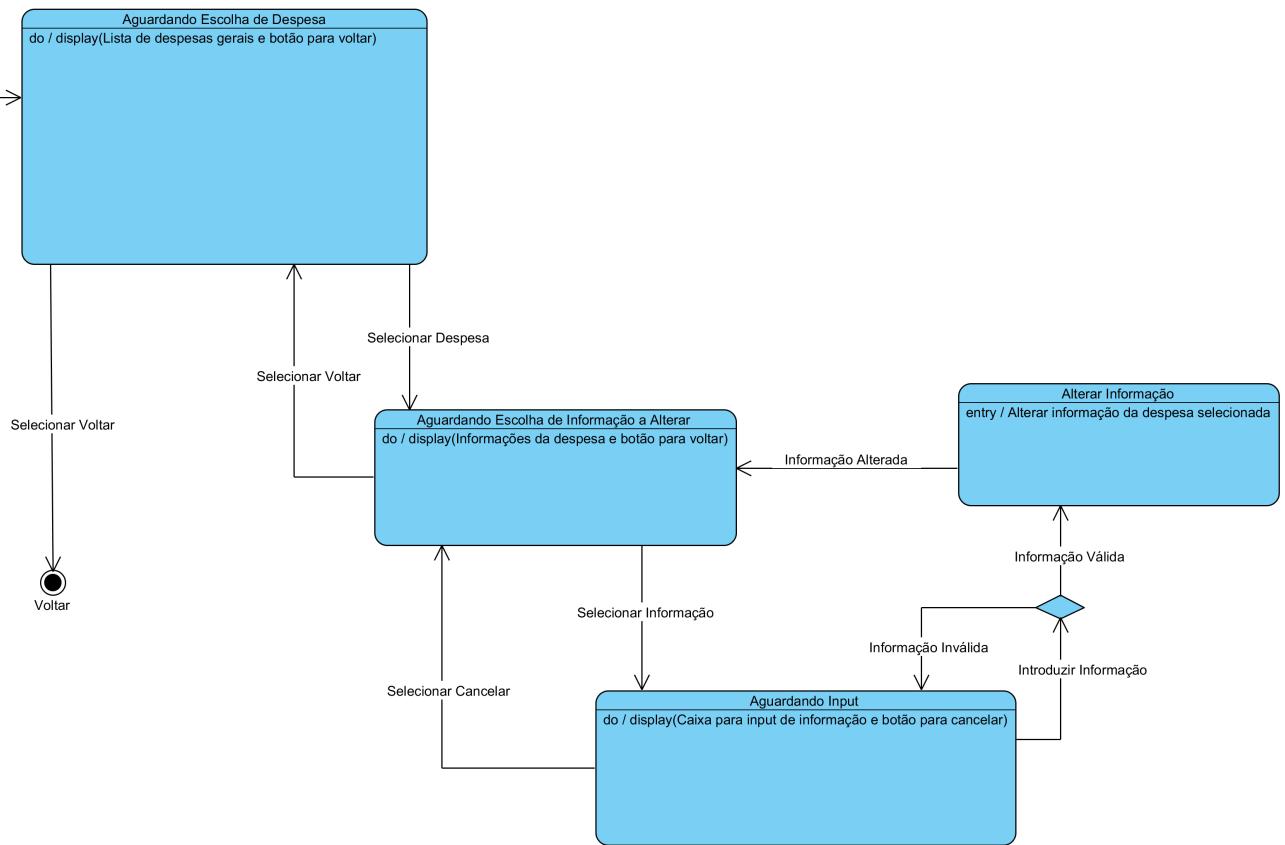


Figura 18: Submenu para Alterar Despesas Gerais.

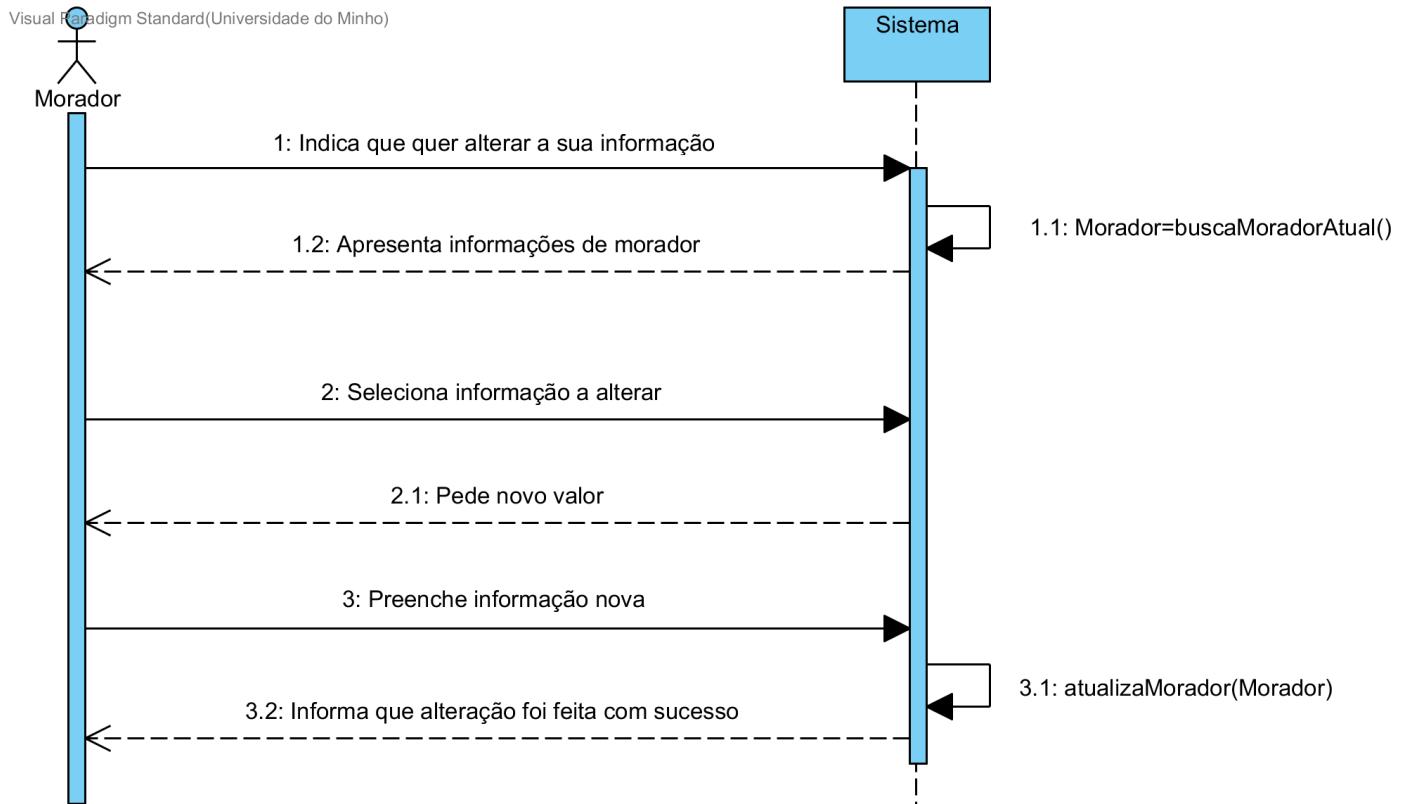


Figura 19: Diagrama de Sequência para Alterar Informação de Morador.

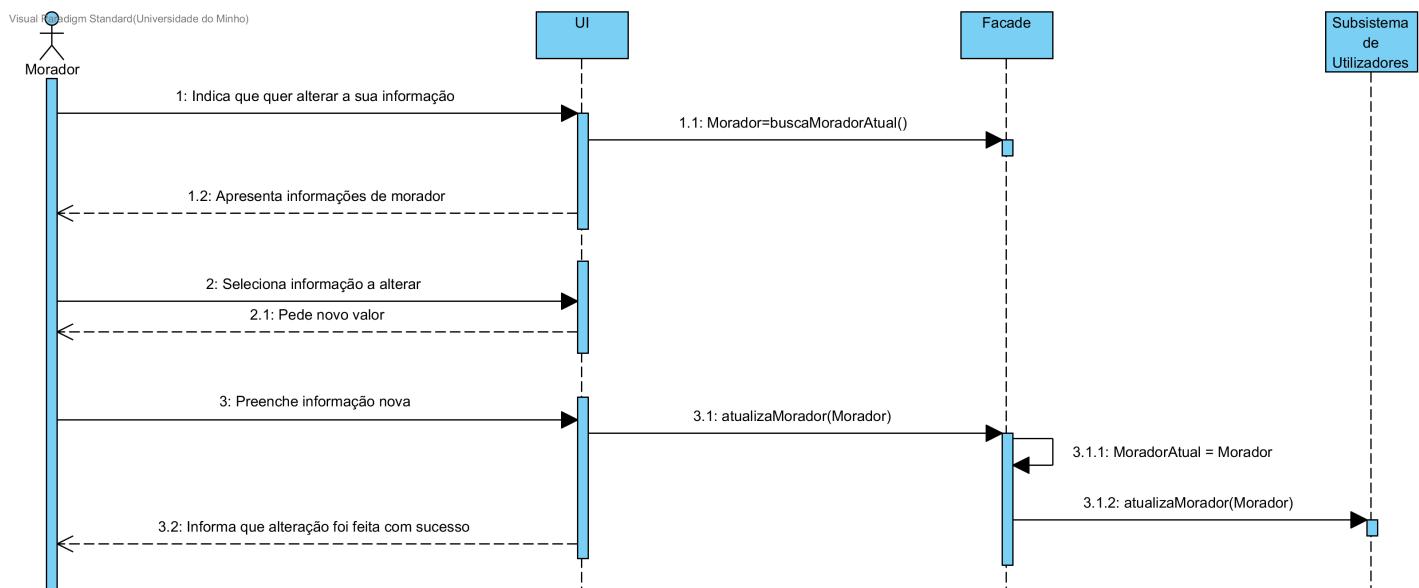


Figura 20: Diagrama de Sequência para Alterar Informação de Morador, com subsistemas.

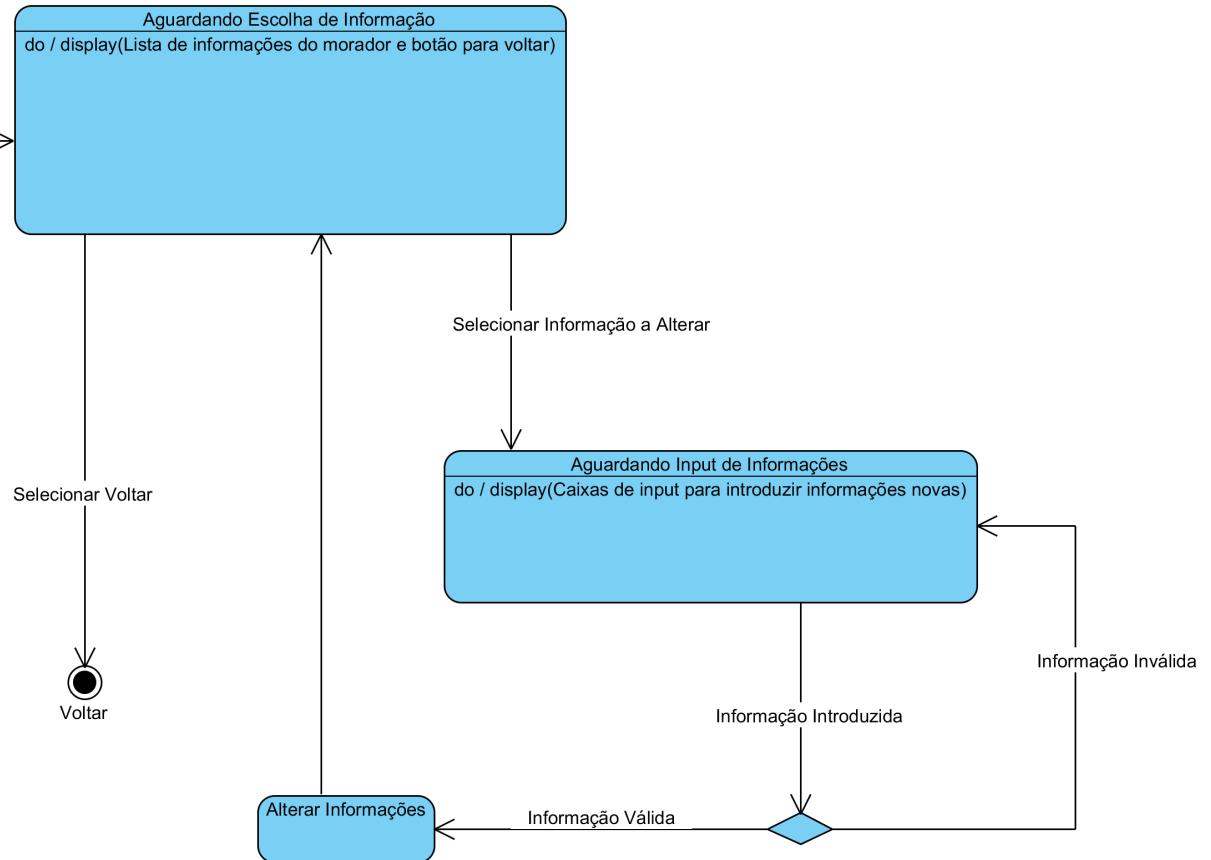


Figura 21: Submenu para Alterar Informação de Morador.

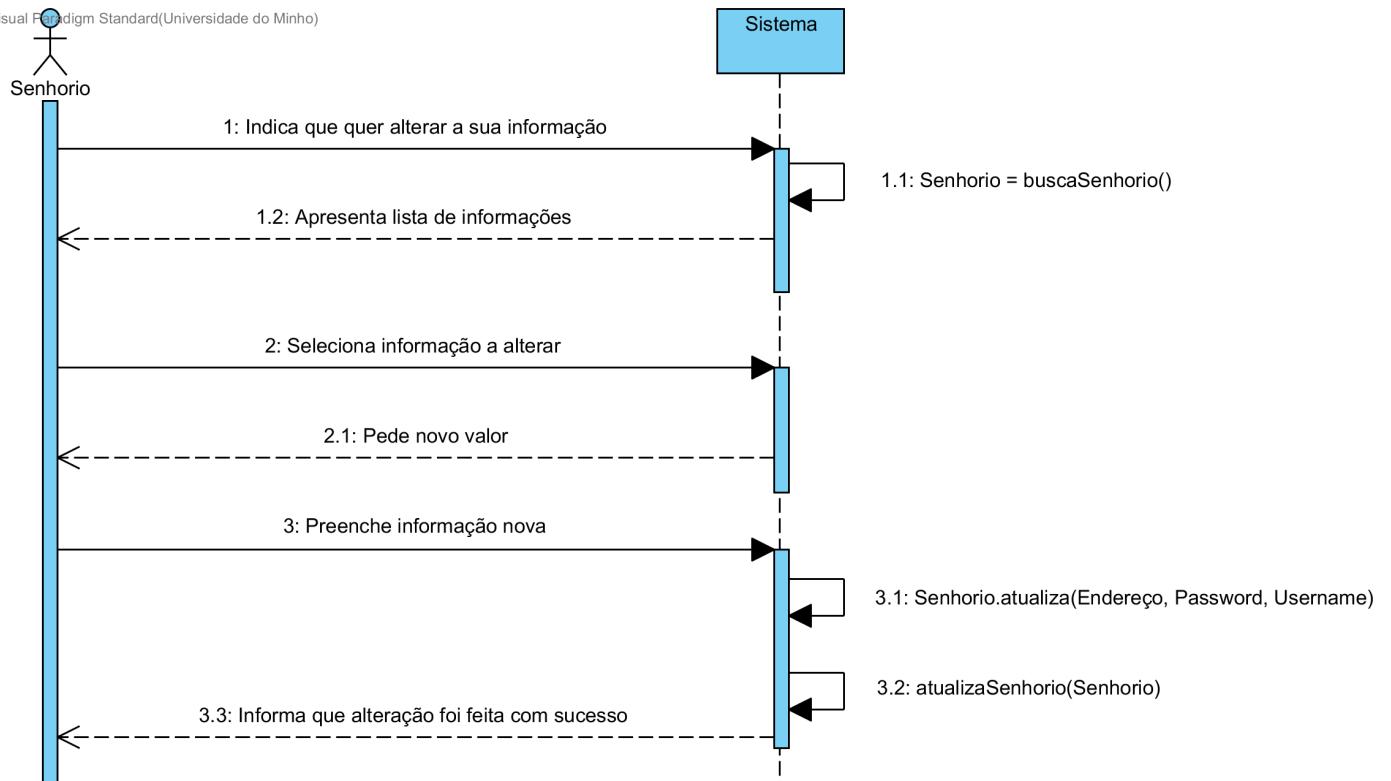


Figura 22: Diagrama de Sequência para Alterar Informação de Senhorio, com subsistemas.

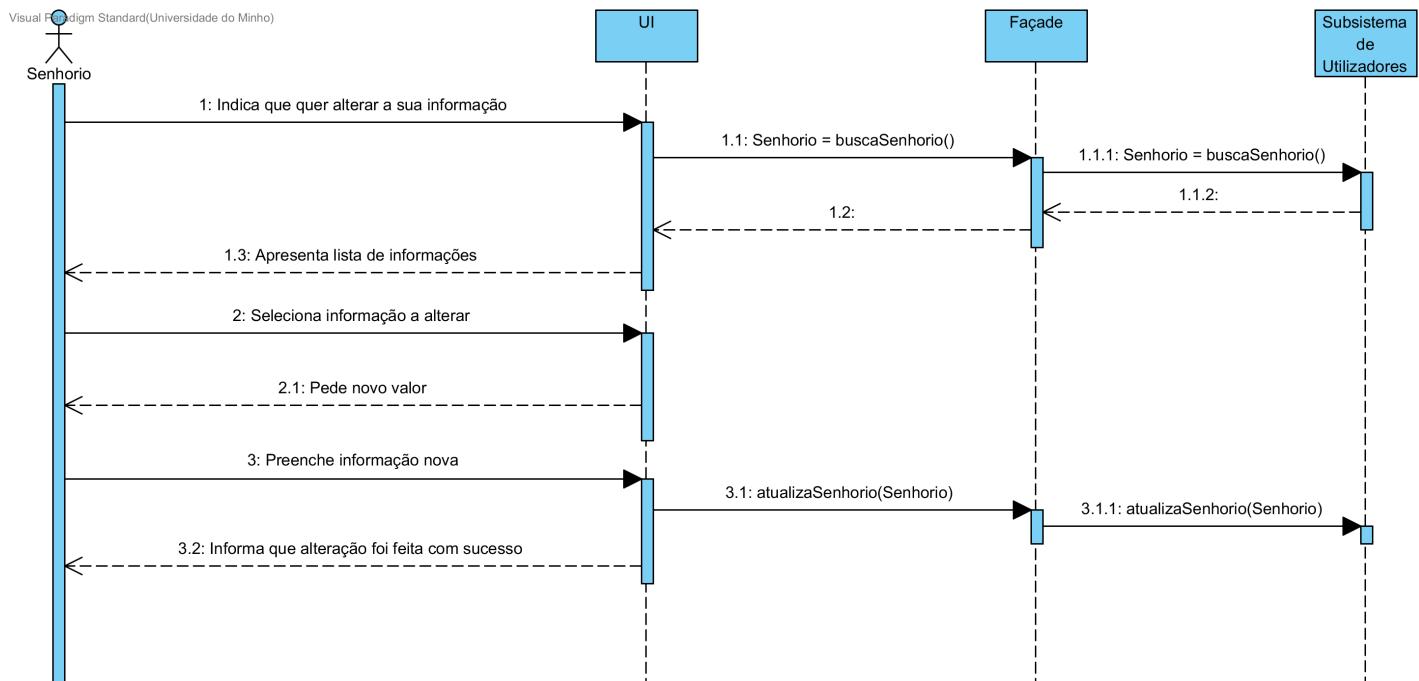


Figura 23: Diagrama de Sequência para Alterar Informação de Senhorio, com subsistemas.

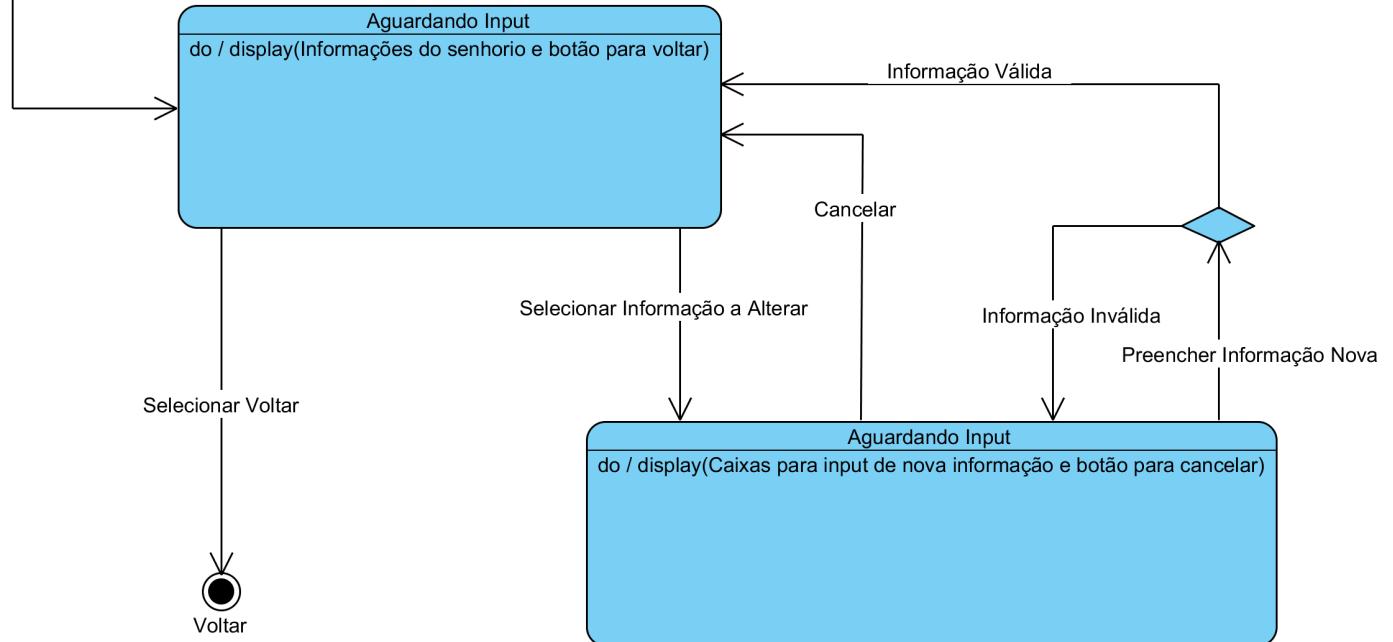


Figura 24: Submenu para Alterar Informação de Senhorio.

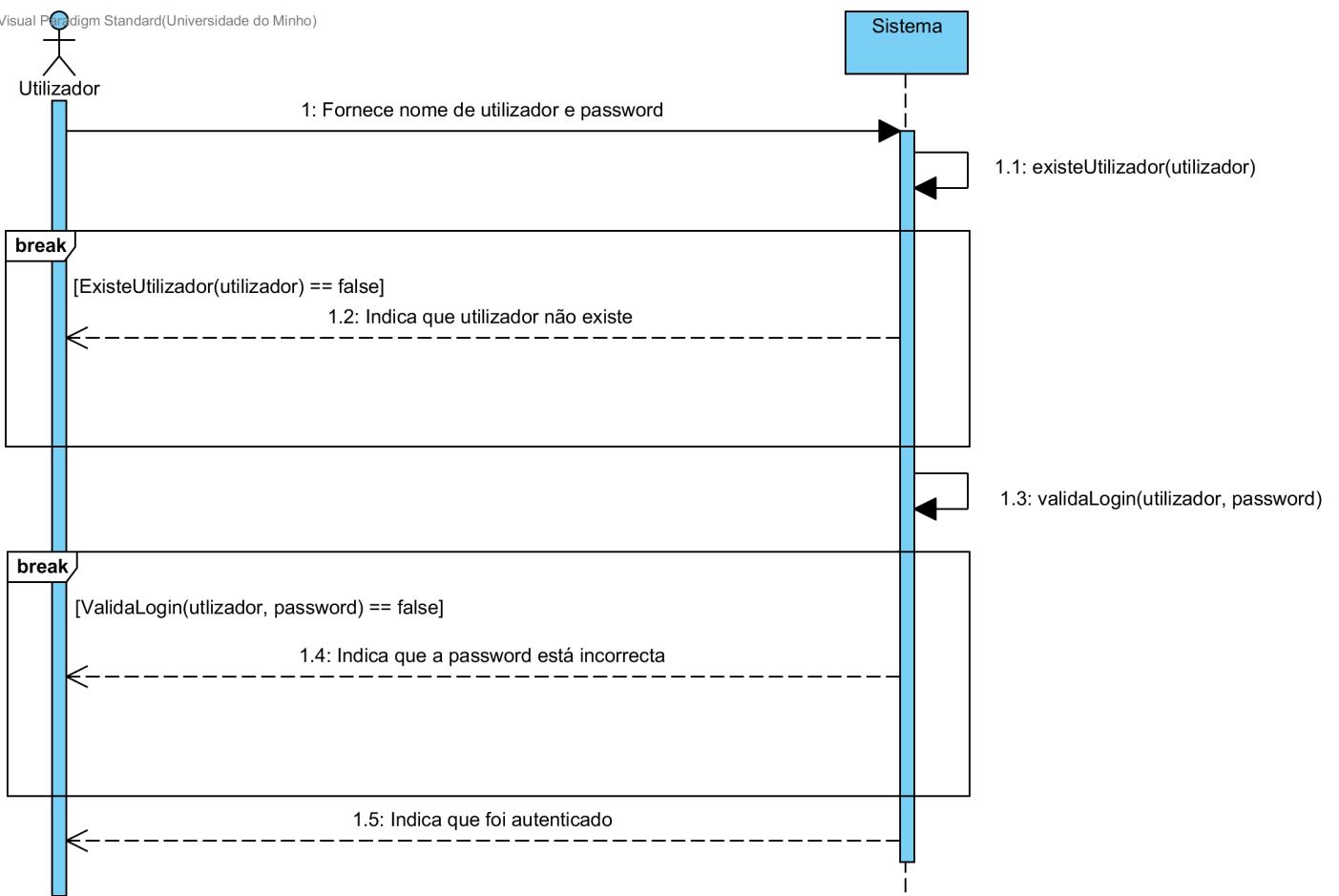


Figura 25: Diagrama de Sequência para Autenticar Utilizador.

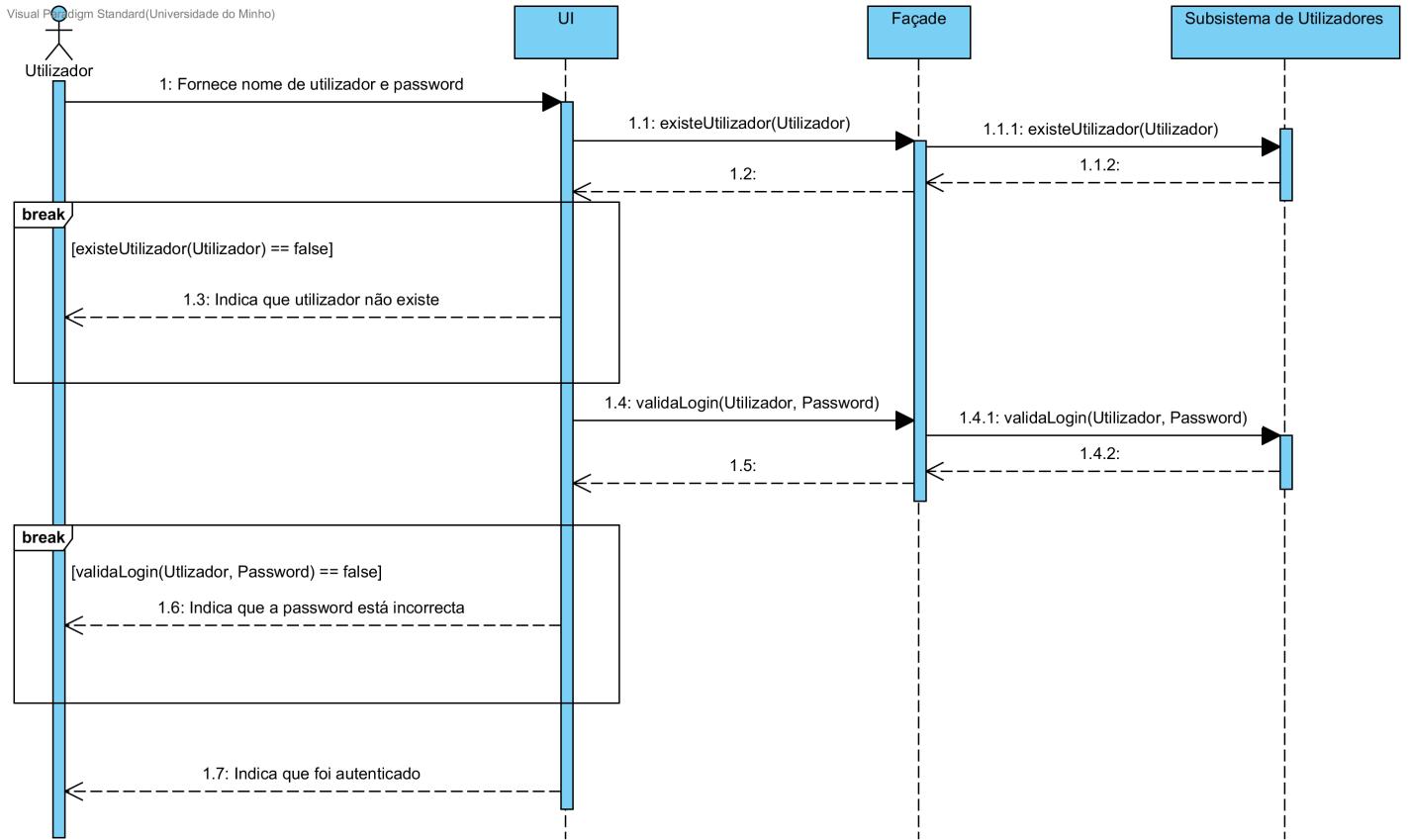


Figura 26: Diagrama de Sequência para Autenticar Utilizador, com subsistemas.

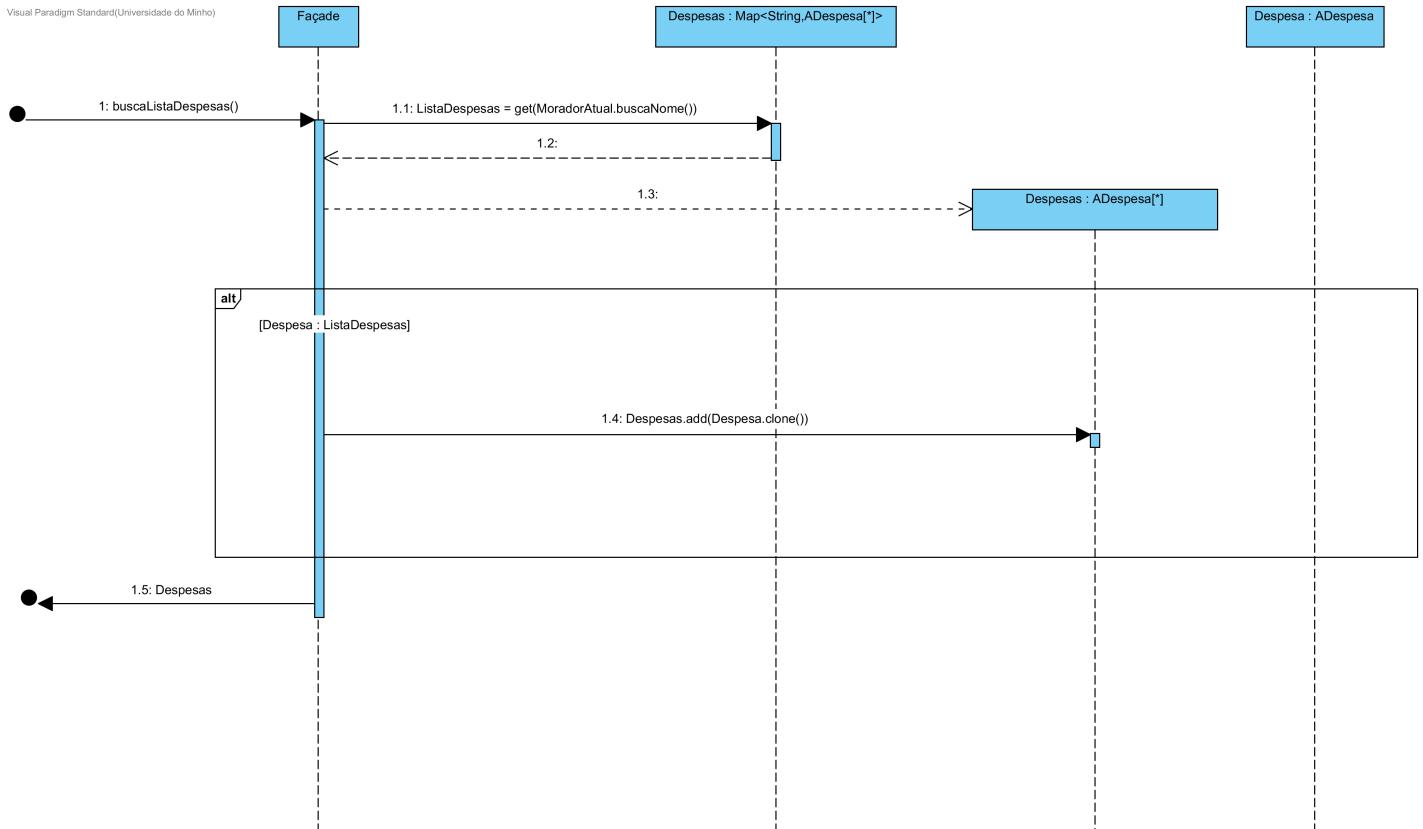


Figura 27: Diagrama de Implementação do método `buscaListaDespesas() : ADespesa[*]`.

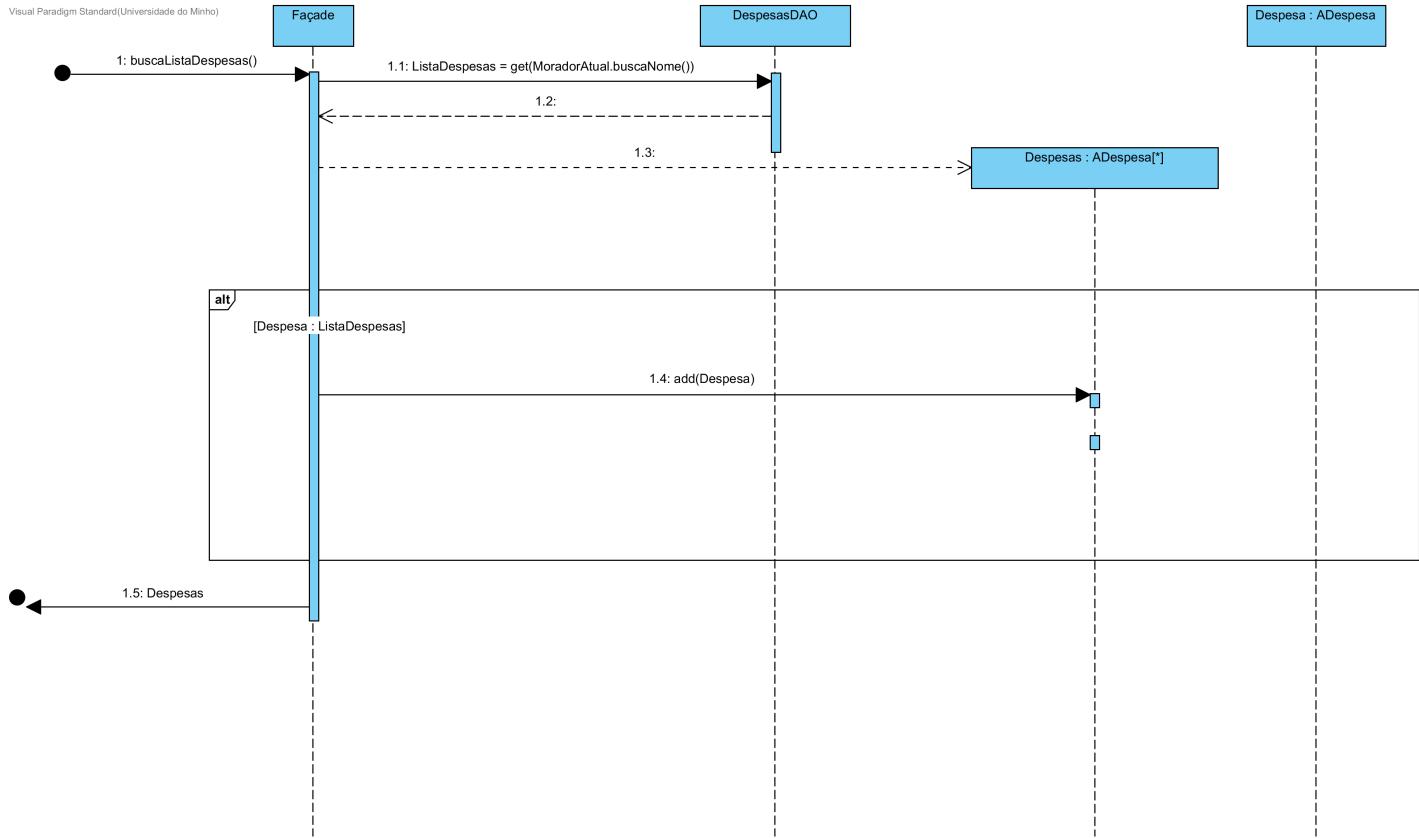


Figura 28: Diagrama de Implementação do método `buscaListaDespesas()` : `ADespesa[]` com DAOs.

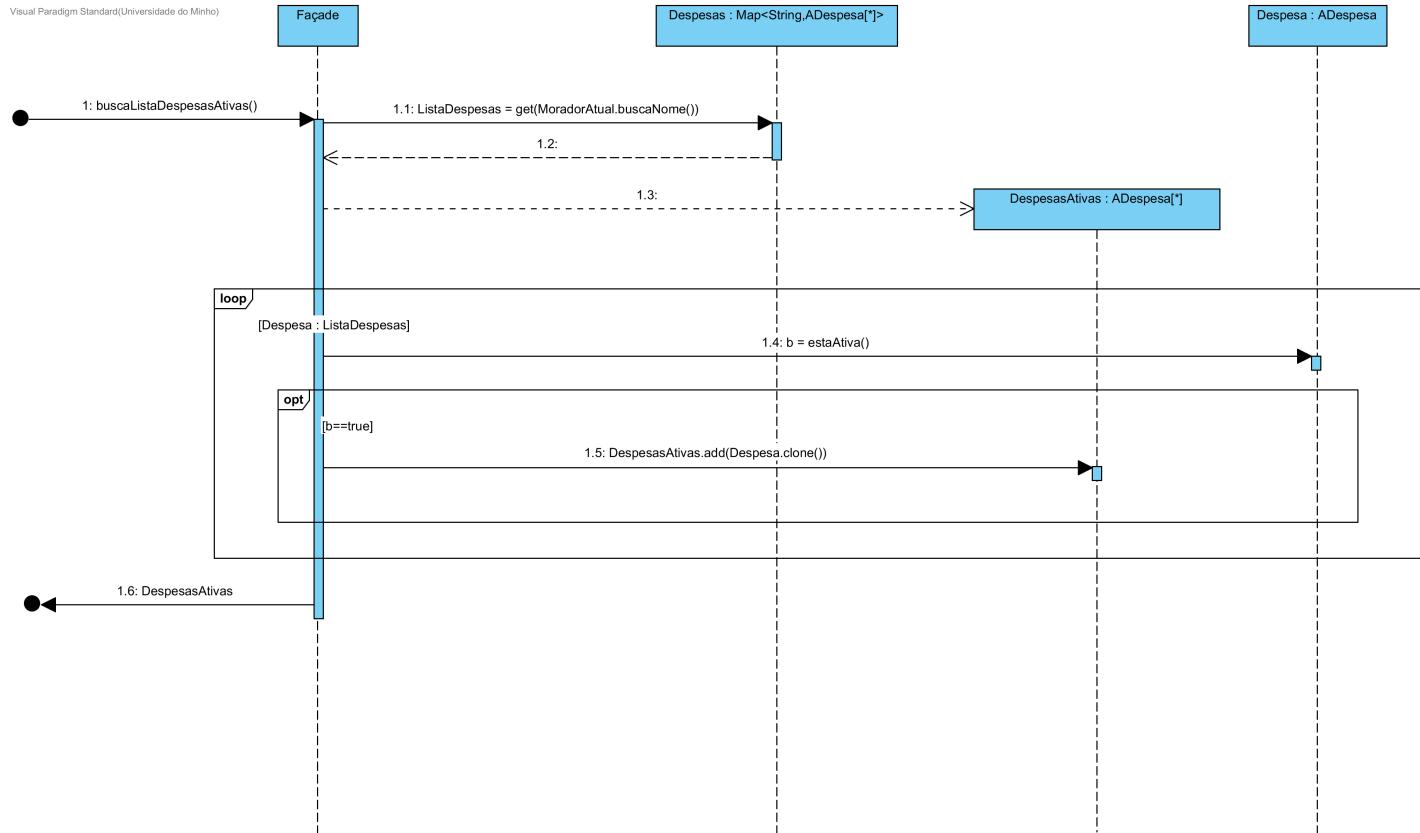


Figura 29: Diagrama de Implementação do método `buscaListaDespesasAtivas()` : `ADespesa[*]`.

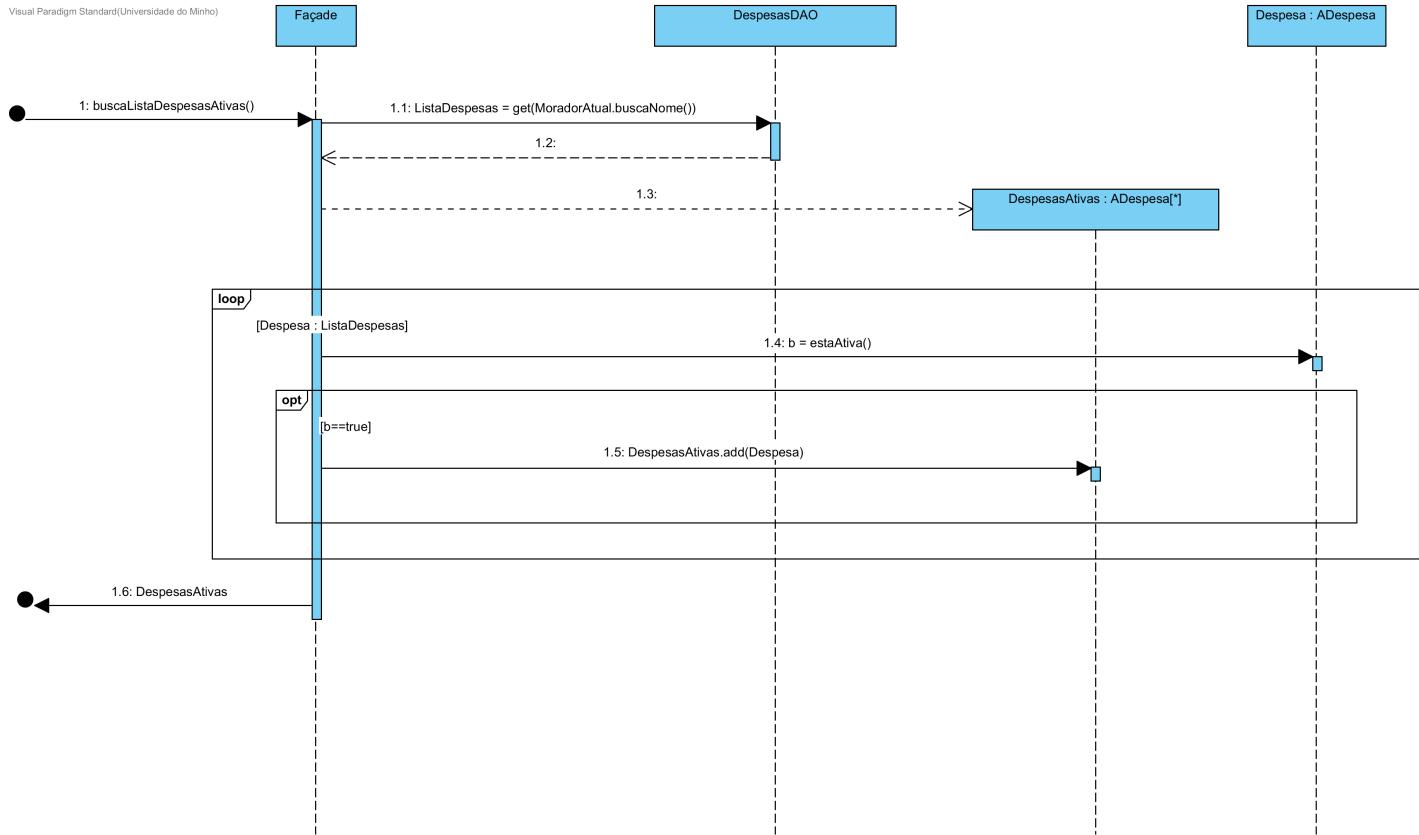


Figura 30: Diagrama de Implementação do método `buscaListaDespesasAtivas()` : `ADespesa[*]` com DAOs.

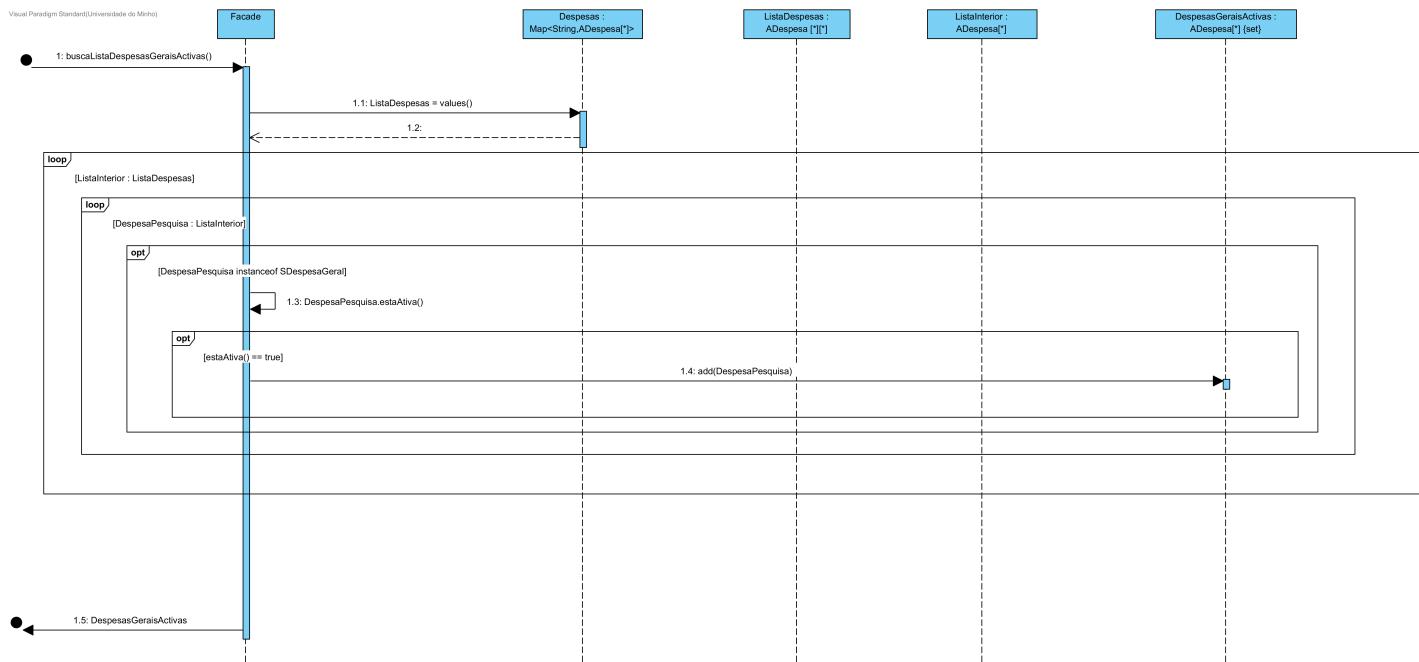


Figura 31: Diagrama de Implementação do método `buscaListaDespesasGeraisActivas()` : `ADespesa[*]`.

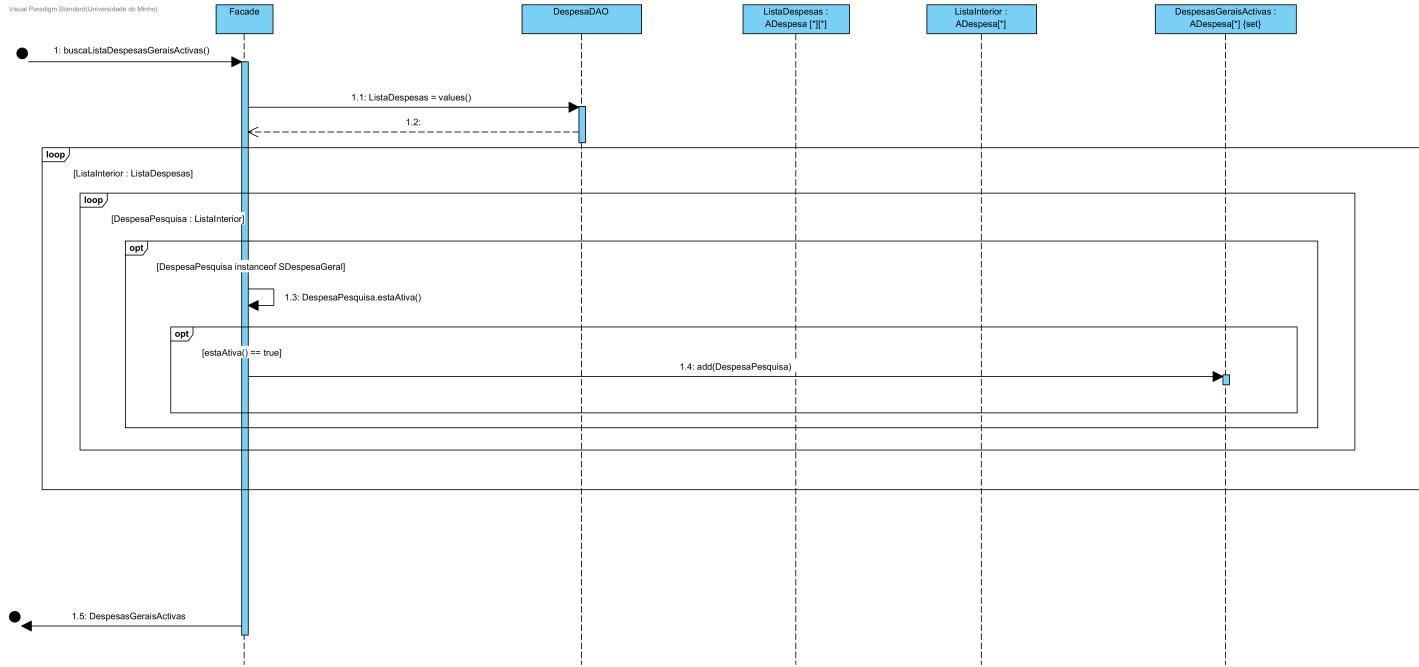


Figura 32: Diagrama de Implementação do método `buscaListaDespesasGeraisActivas()` : `ADespesa[*]` com DAOs.

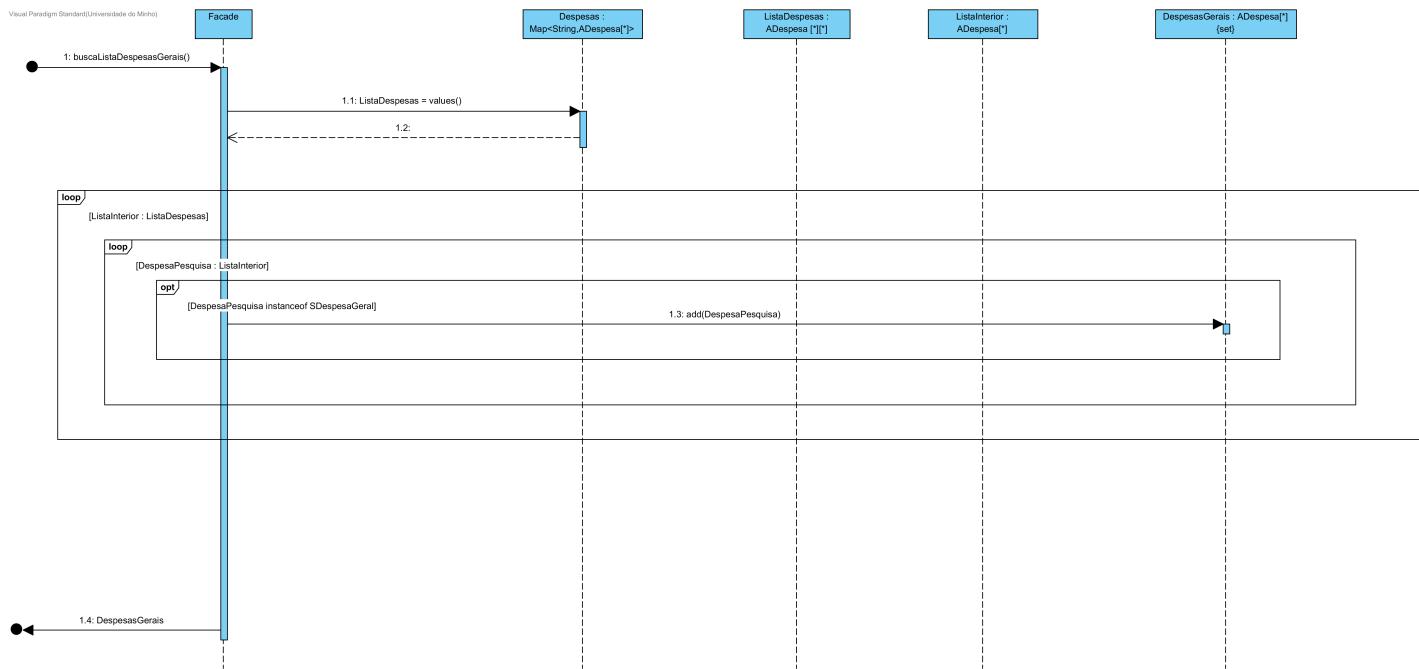


Figura 33: Diagrama de Implementação do método `buscaListaDespesasGerais()` : `ADespesa[*]`.

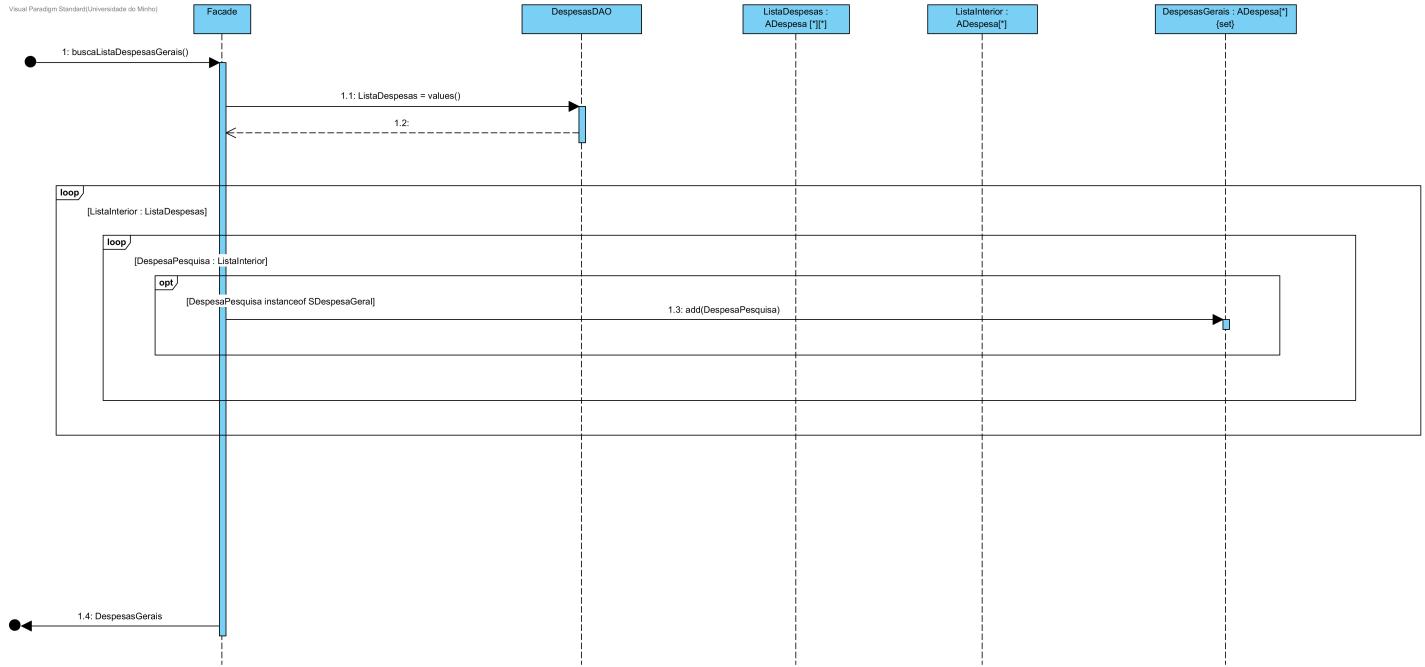


Figura 34: Diagrama de Implementação do método `buscaListaDespesasGerais() : ADespesa[]` com DAOs.

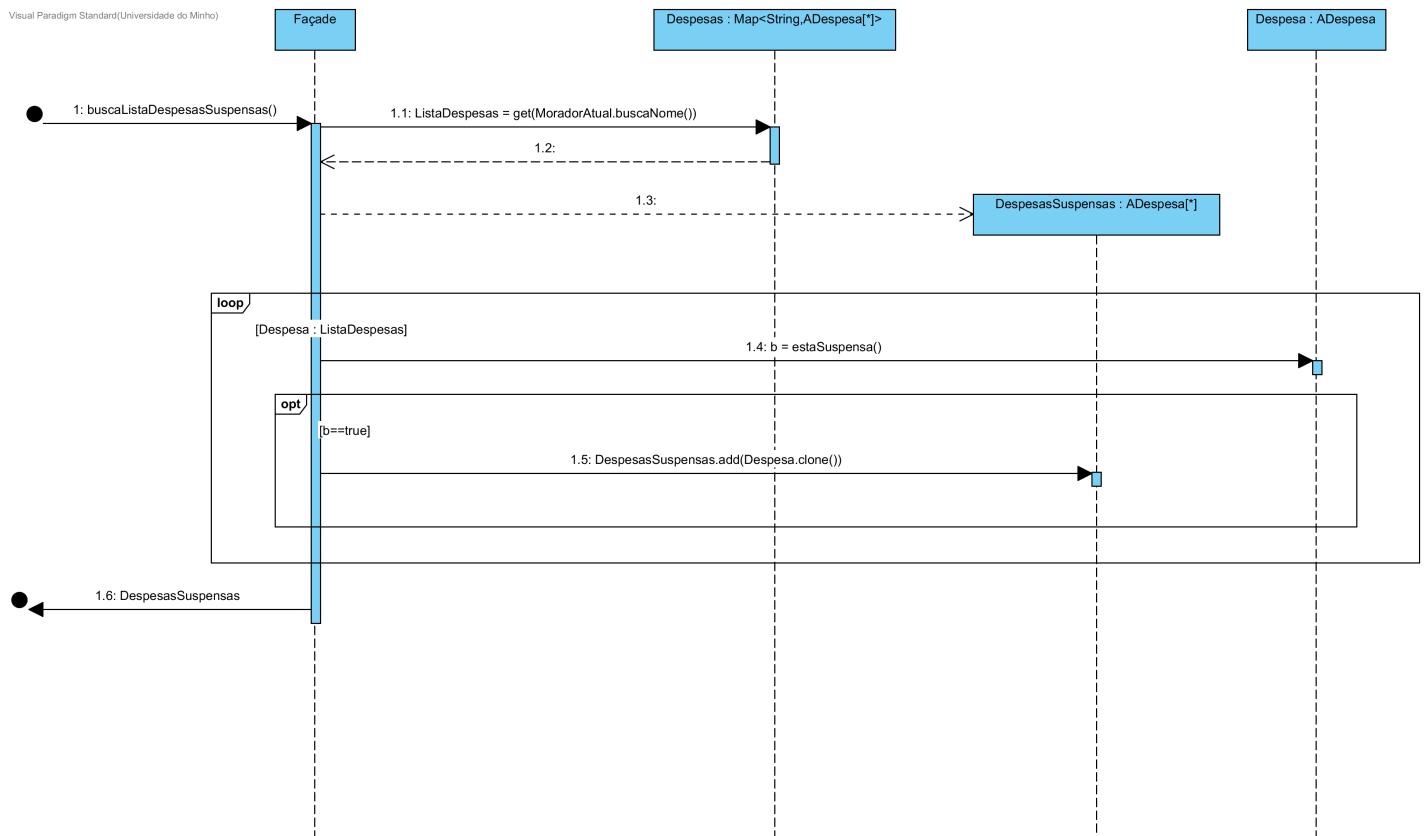


Figura 35: Diagrama de Implementação do método `buscaListaDespesasSuspensas() : ADespesa[]`.

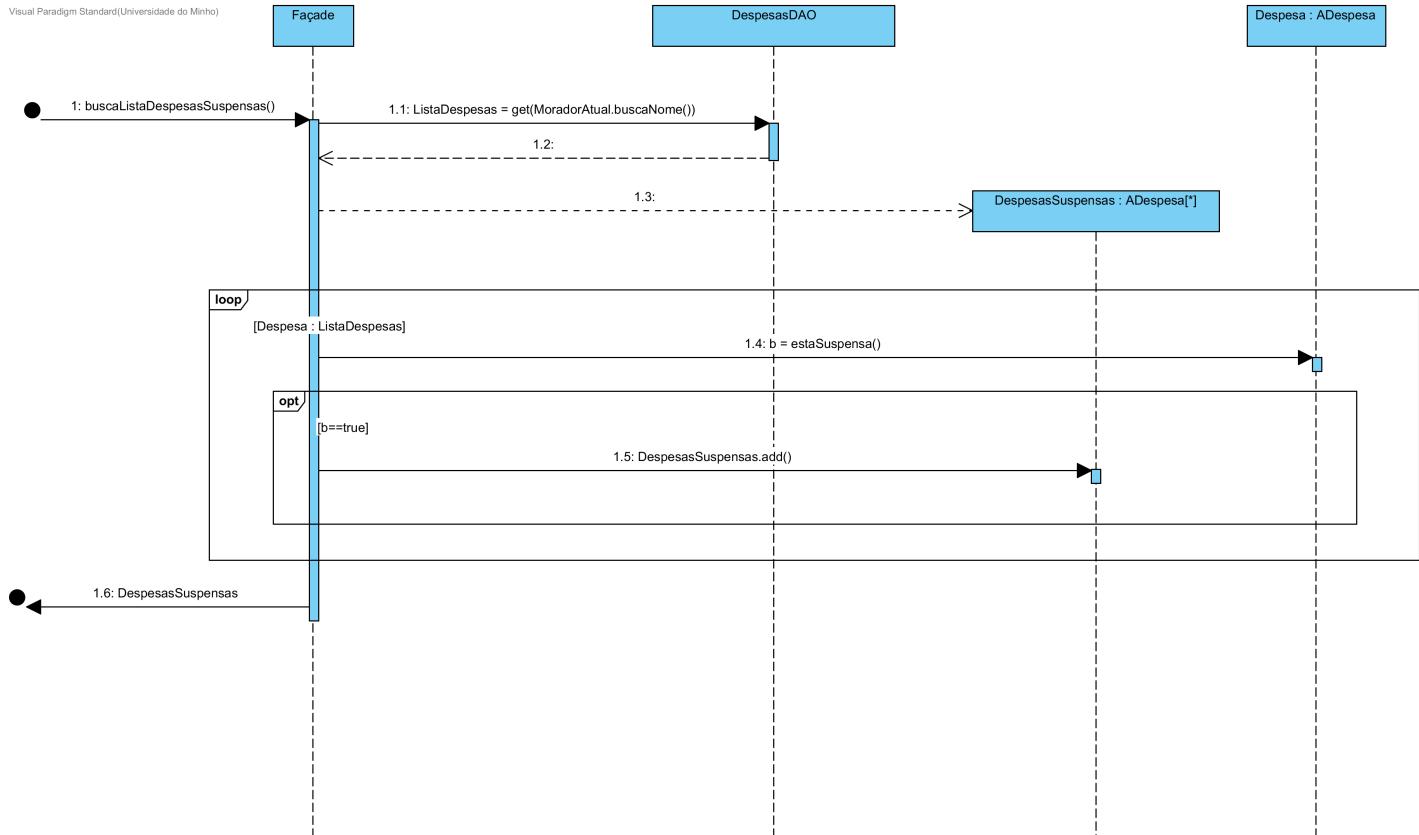


Figura 36: Diagrama de Implementação do método `buscaListaDespesasSuspensas()` : `ADespesa[*]` com DAOs.

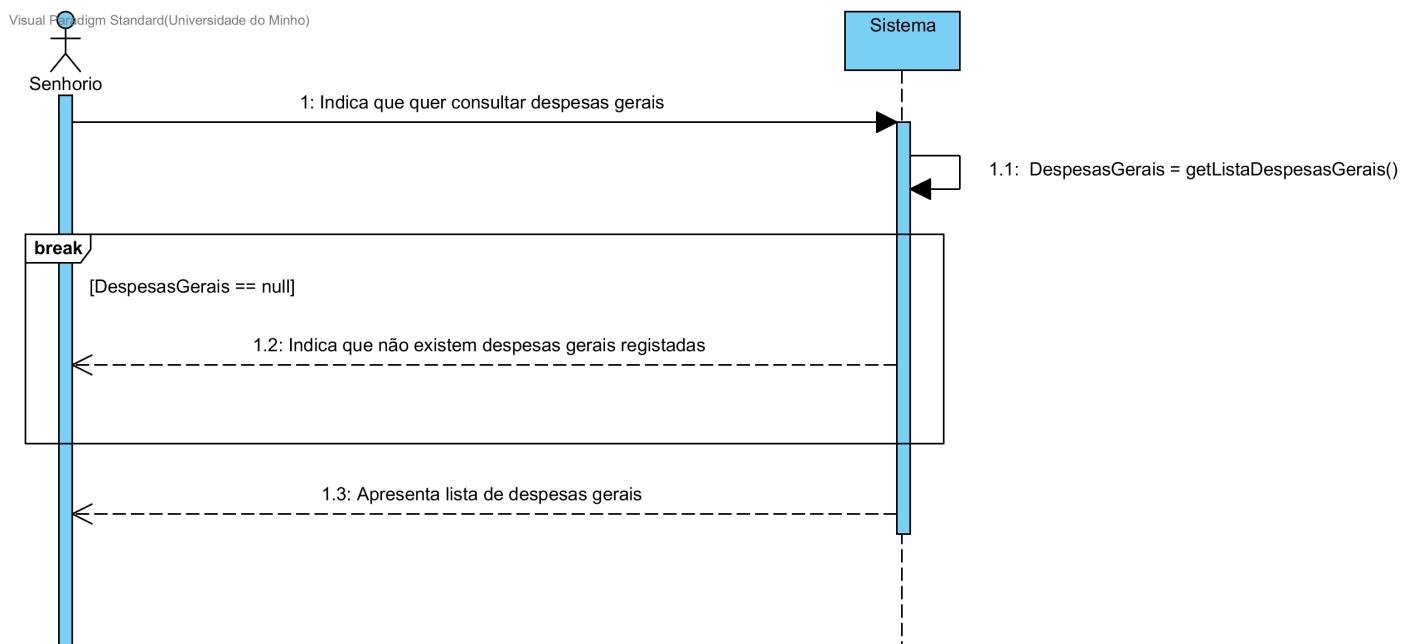


Figura 37: Diagrama de Sequência para Consultar de Despesas Gerais.

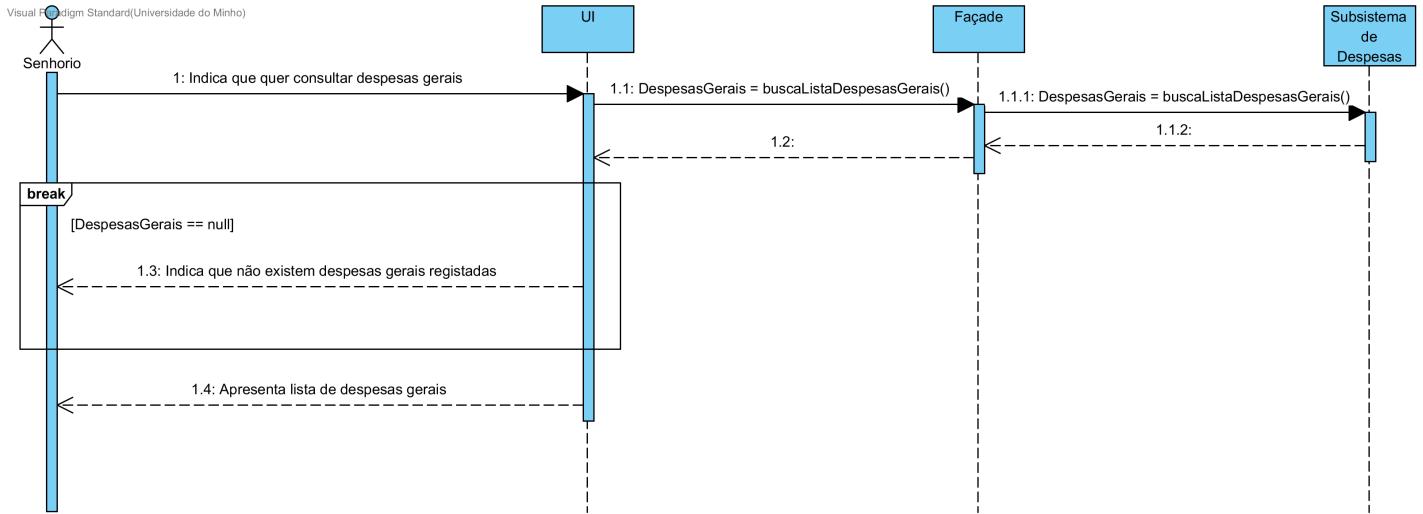


Figura 38: Diagrama de Sequência para Consultar de Despesas Gerais, com subsistemas.

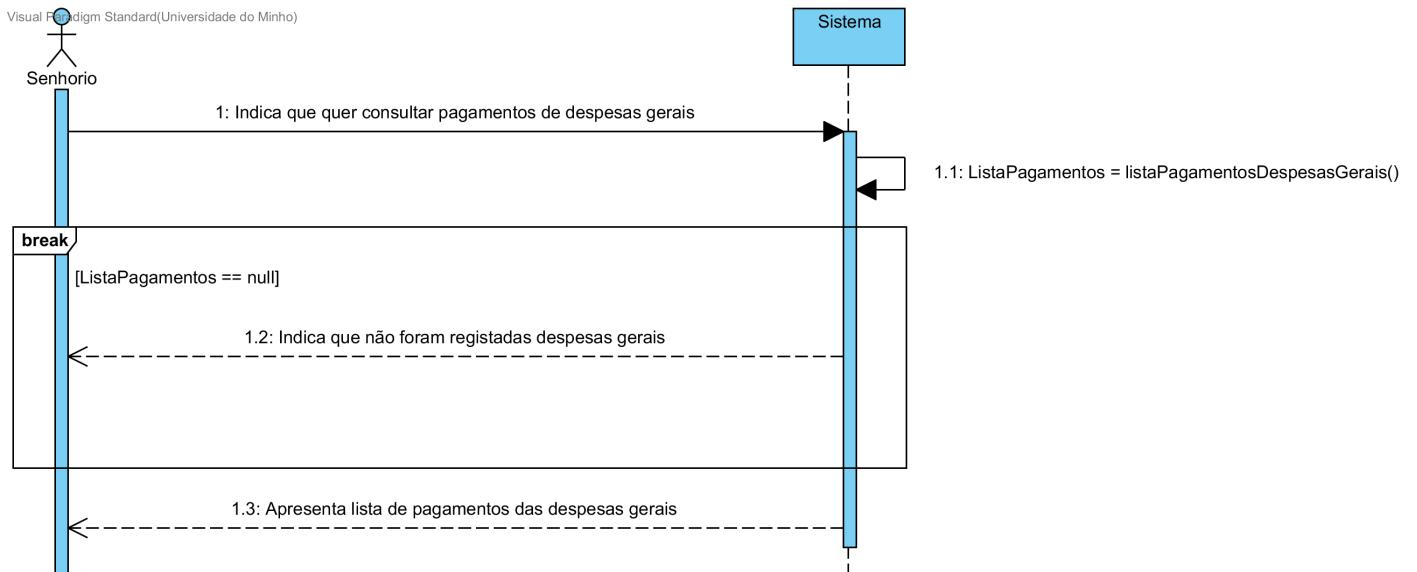


Figura 39: Diagrama de Sequência para Consultar de Pagamentos de Despesas Gerais.

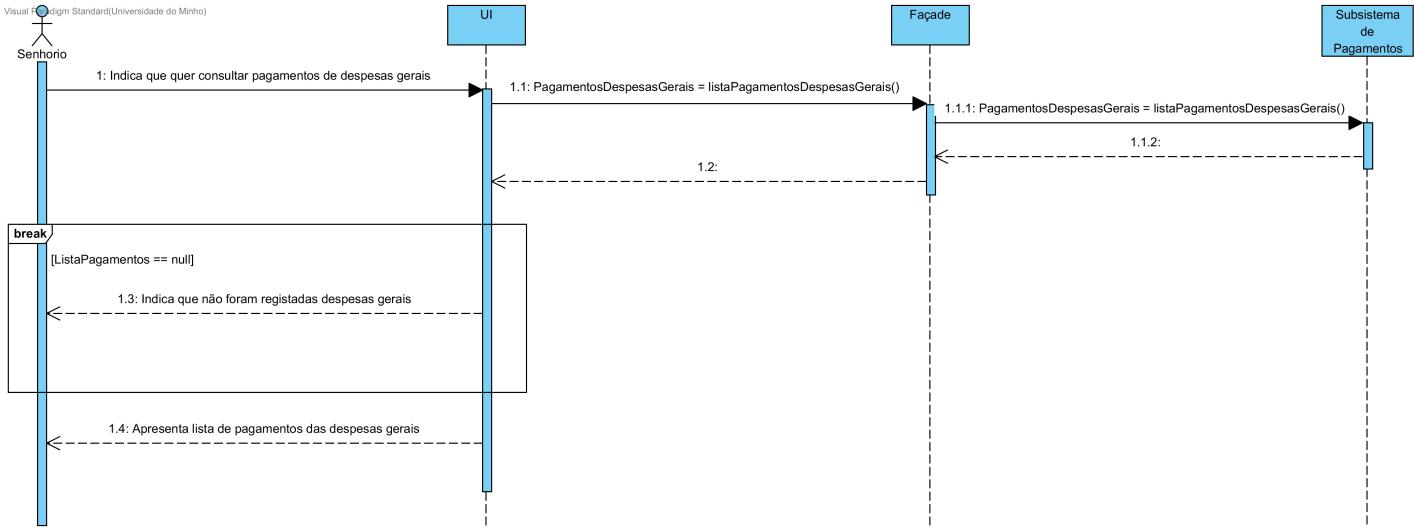


Figura 40: Diagrama de Sequência para Consultar de Pagamentos de Despesas Gerais, com subsistemas.

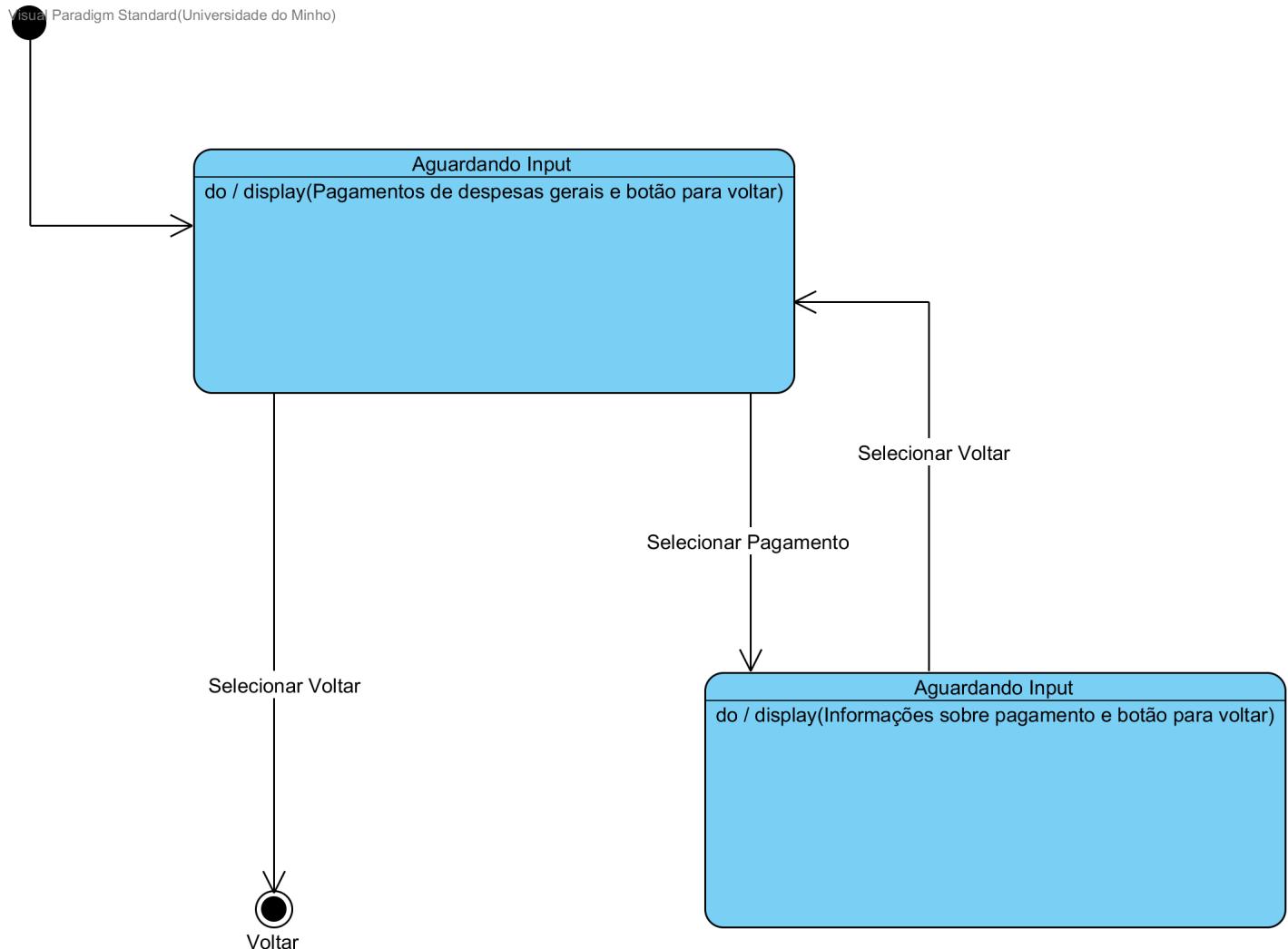


Figura 41: Submenu para Consulta de Pagamentos de Despesas Gerais.

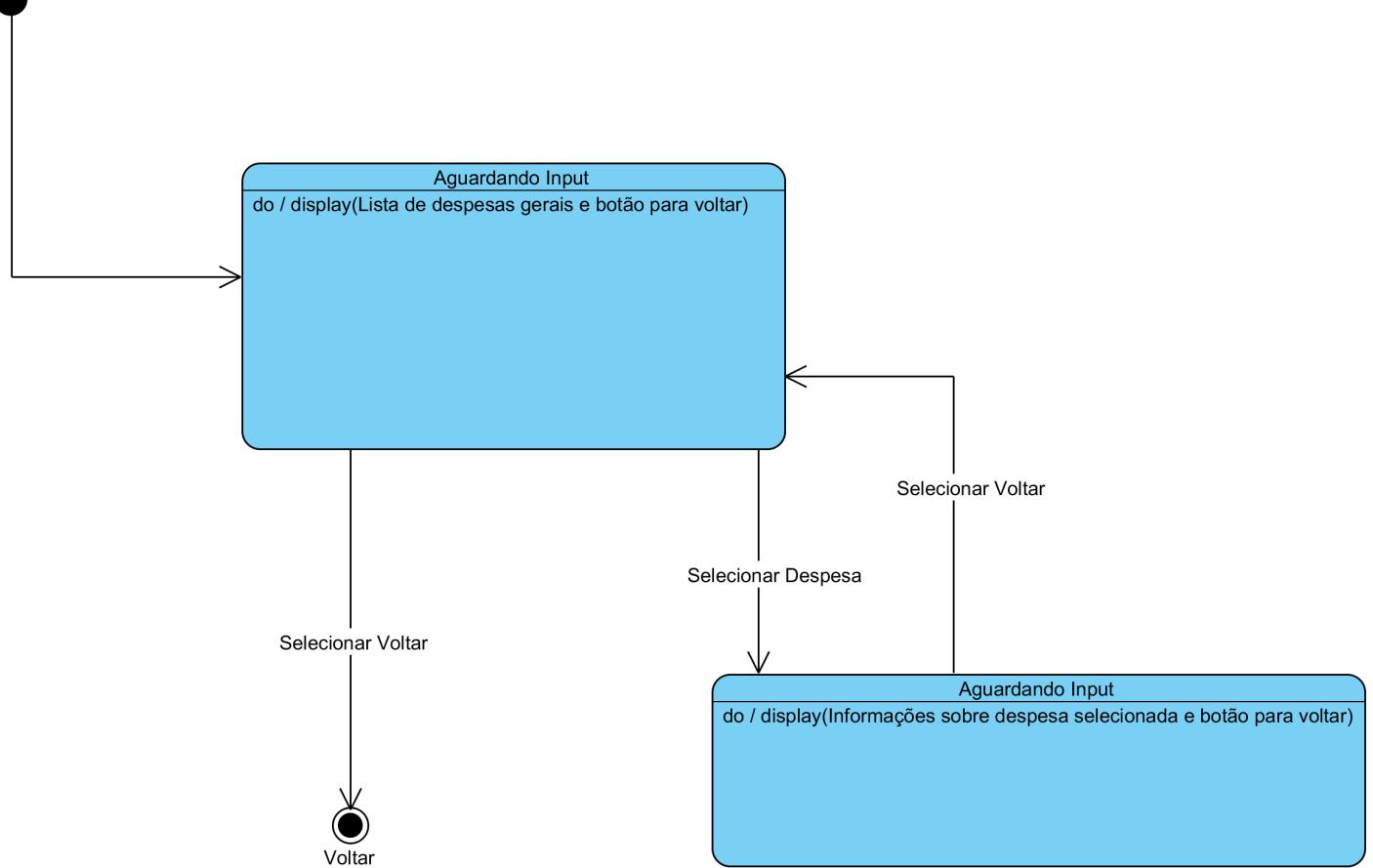


Figura 42: Submenu para Consulta de Despesas Gerais.

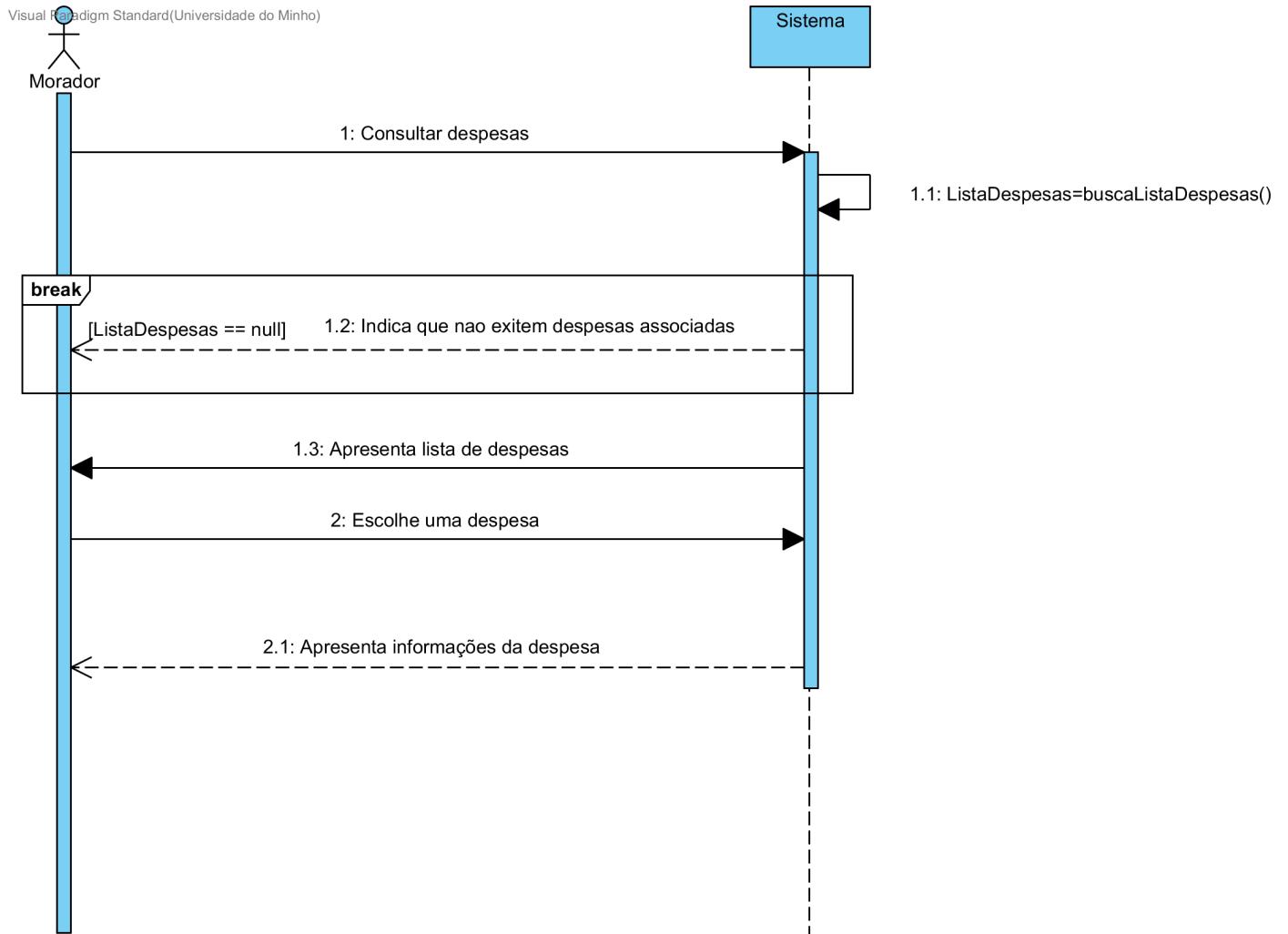


Figura 43: Diagrama de Sequência para Consultar Despesas.

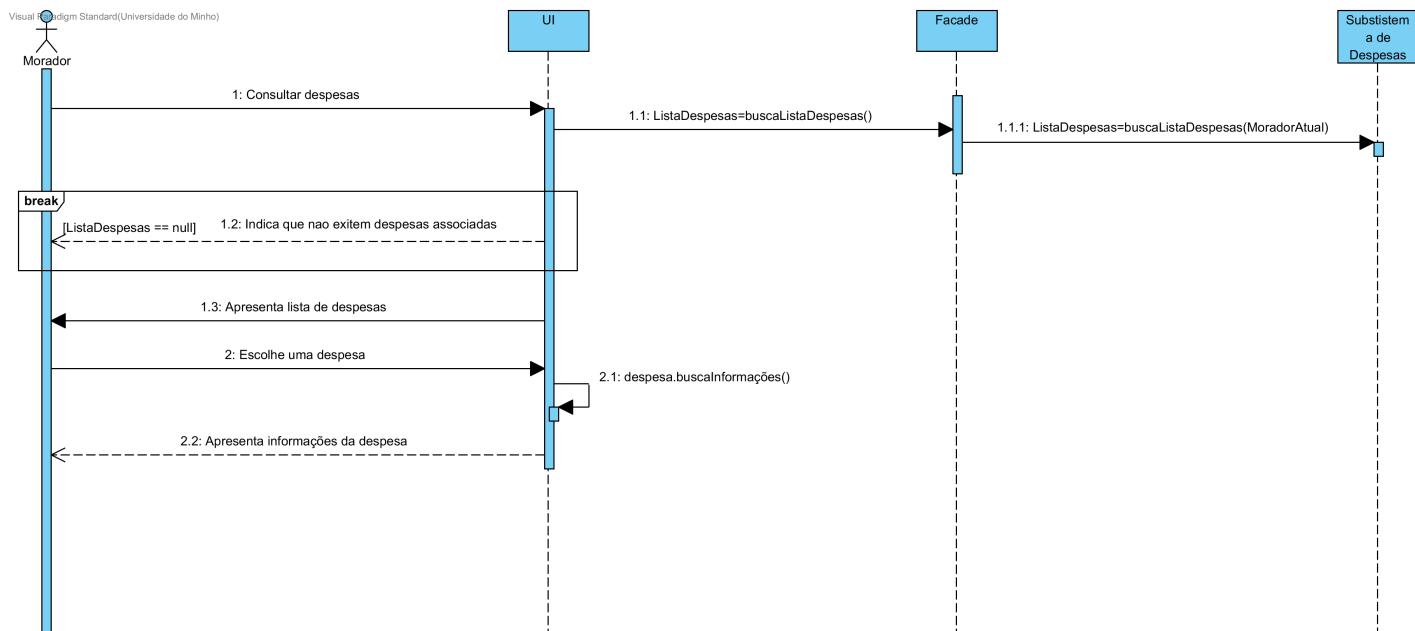


Figura 44: Diagrama de Sequência para Consultar Despesas, com subsistemas.

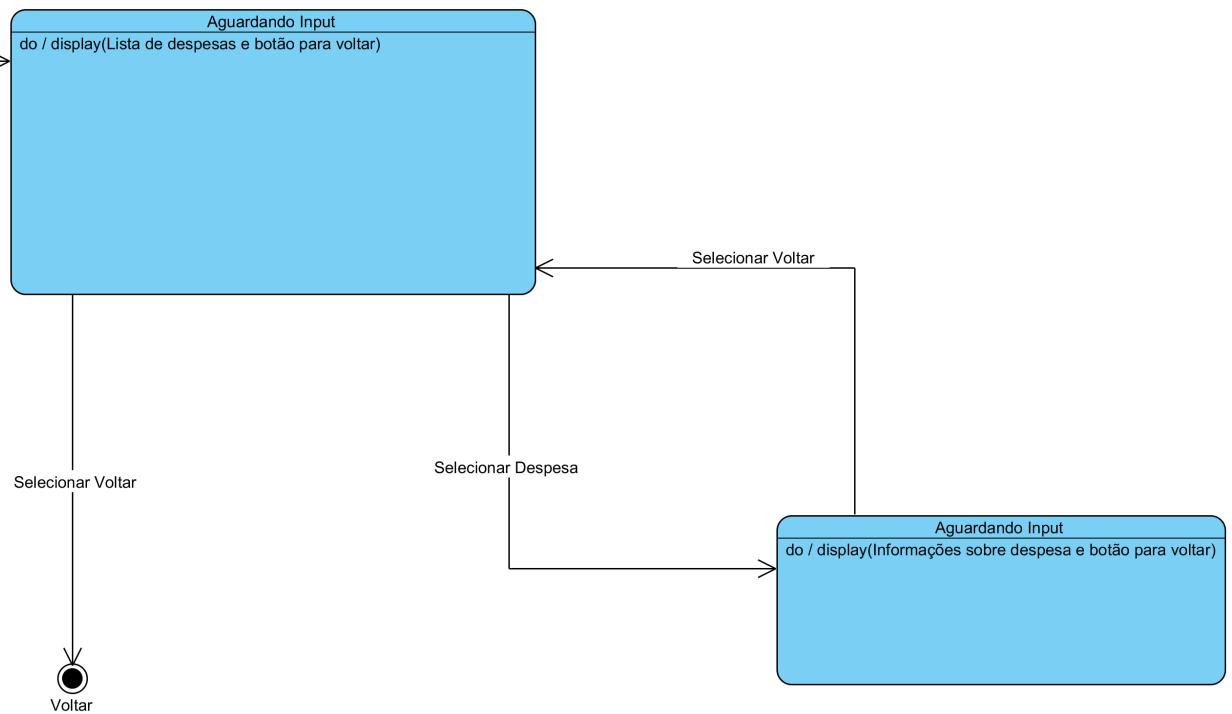


Figura 45: Submenu para Consultar Despesas.

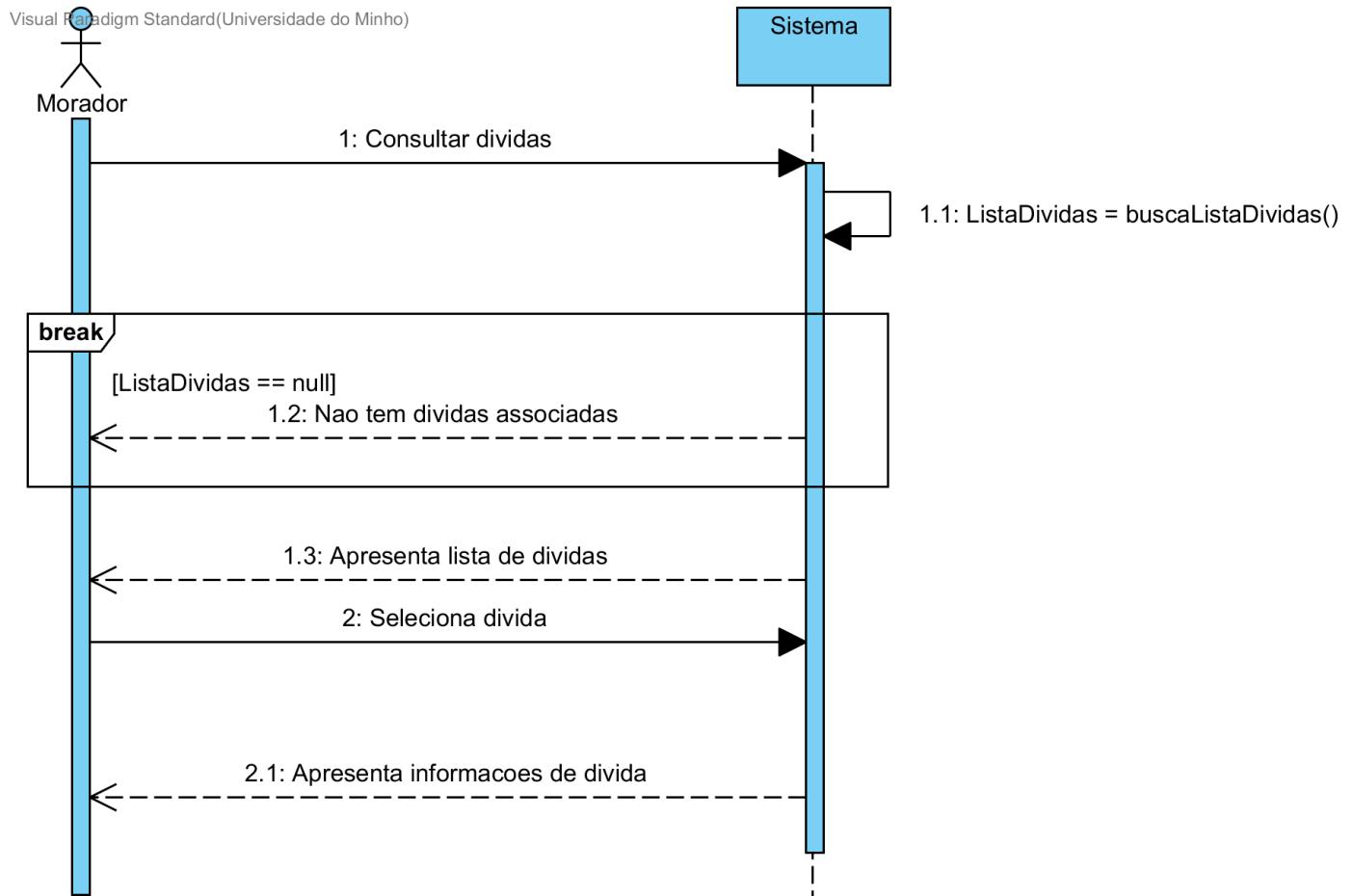


Figura 46: Diagrama de Sequência para Consultar Dívidas.

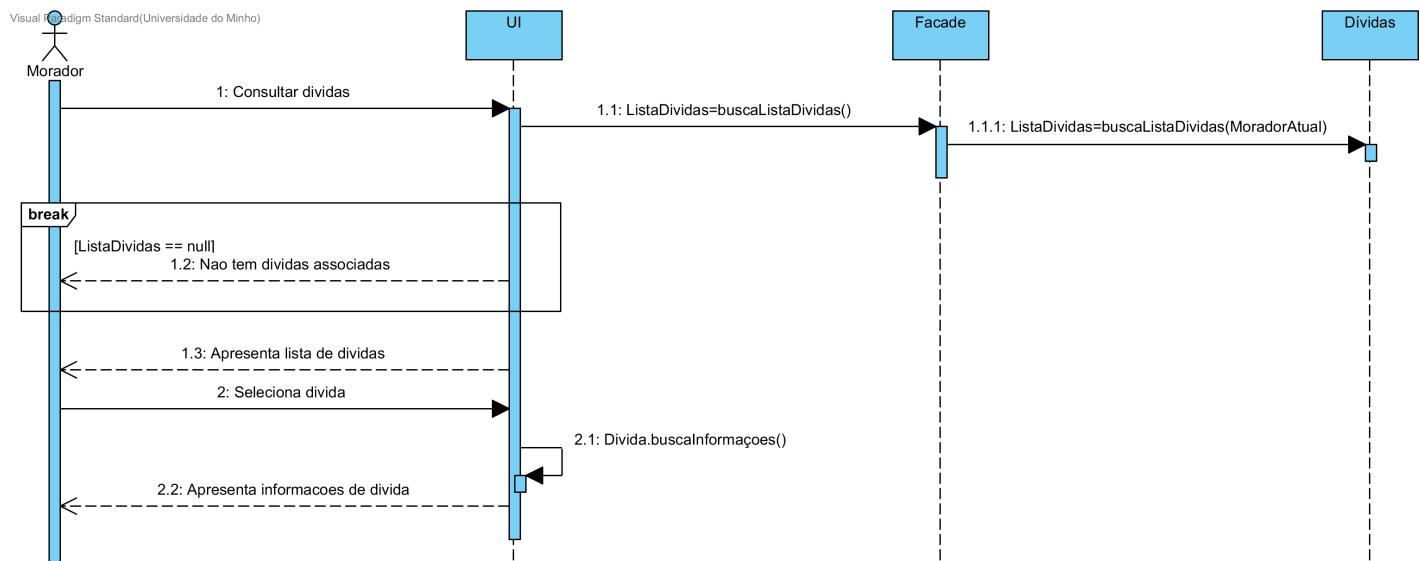


Figura 47: Diagrama de Sequência para Consultar Dívidas, com subsistemas.

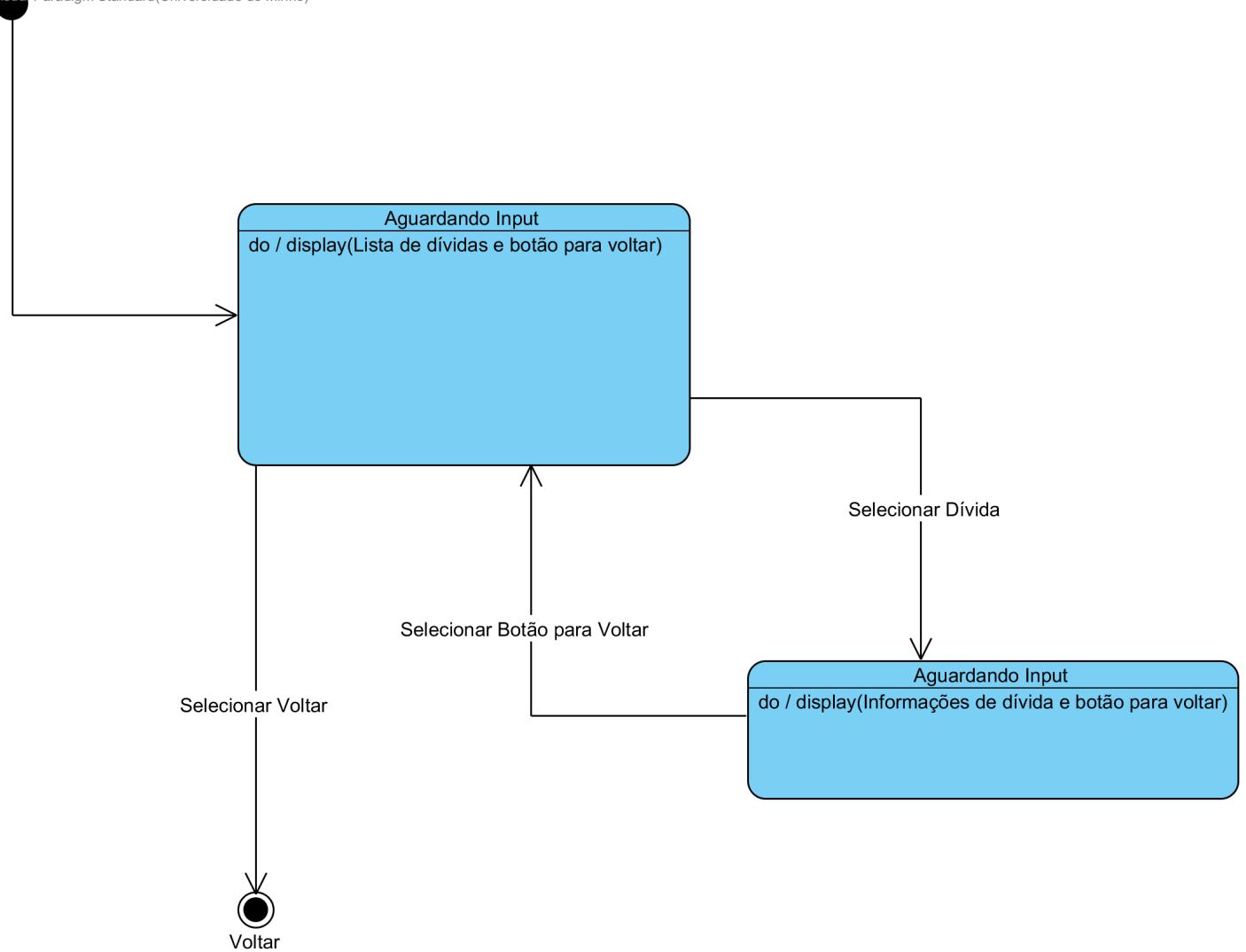


Figura 48: Submenu para Consultar Dívidas.

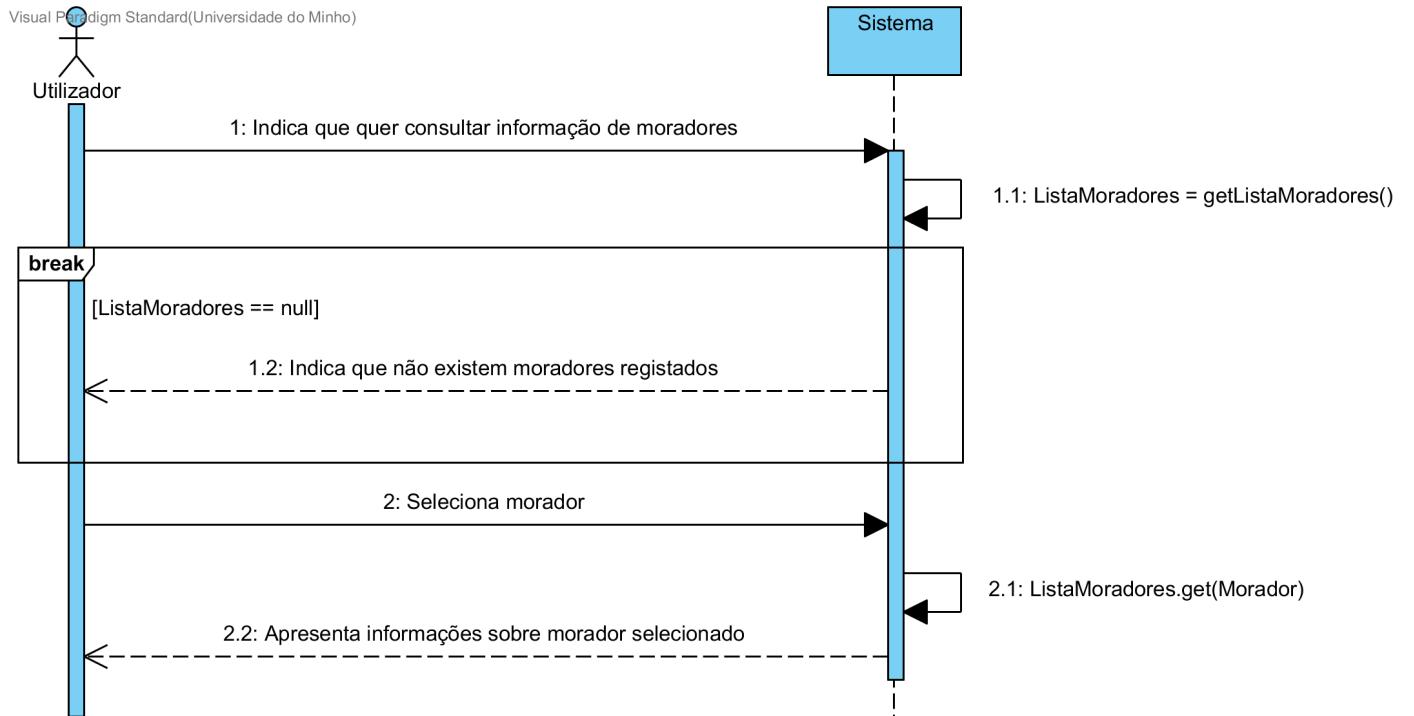


Figura 49: Diagrama de Sequência para Consultar Informação de Moradores.

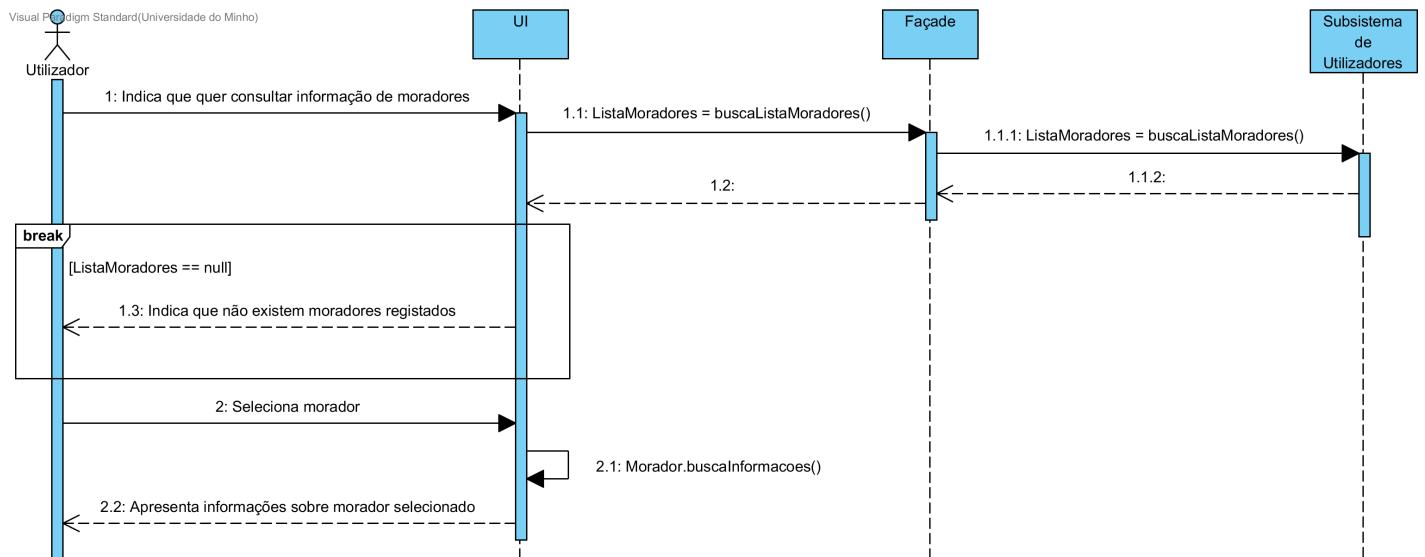


Figura 50: Diagrama de Sequência para Consultar Informação de Moradores, com subsistemas.

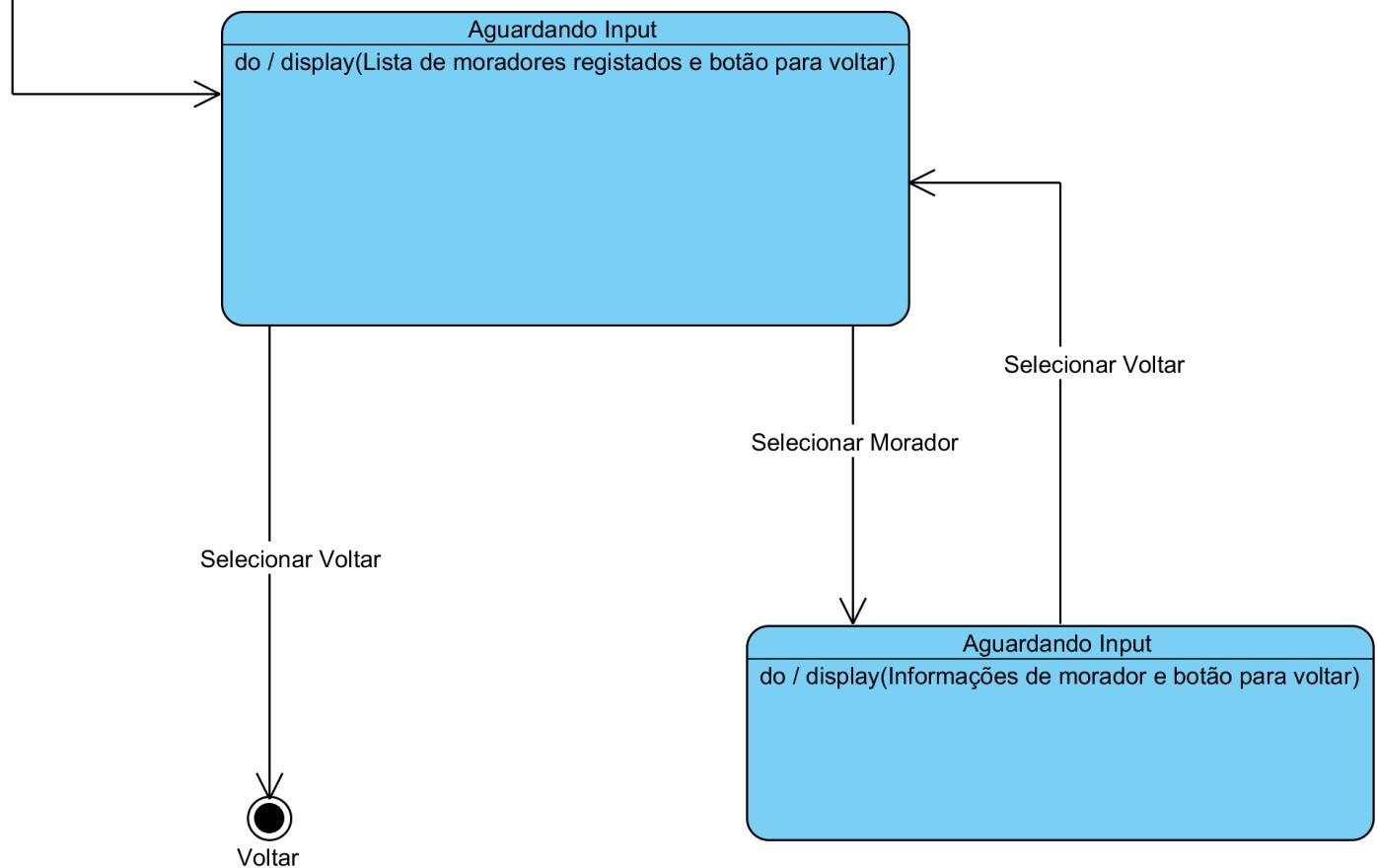


Figura 51: Submenu para Consultar Informação de Moradores.

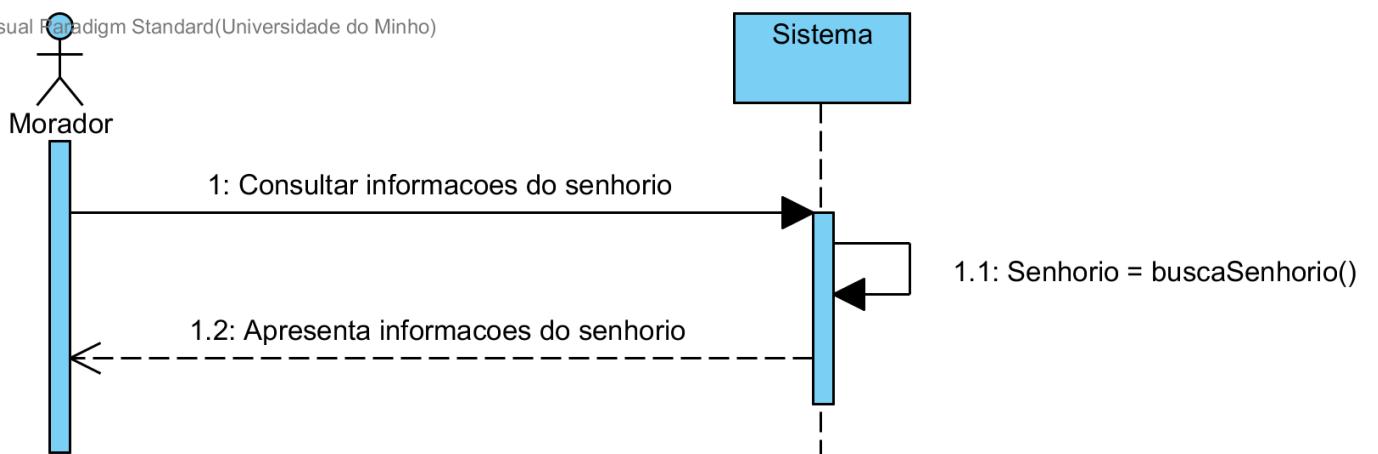


Figura 52: Diagrama de Sequência para Consultar Informação de Senhorio.

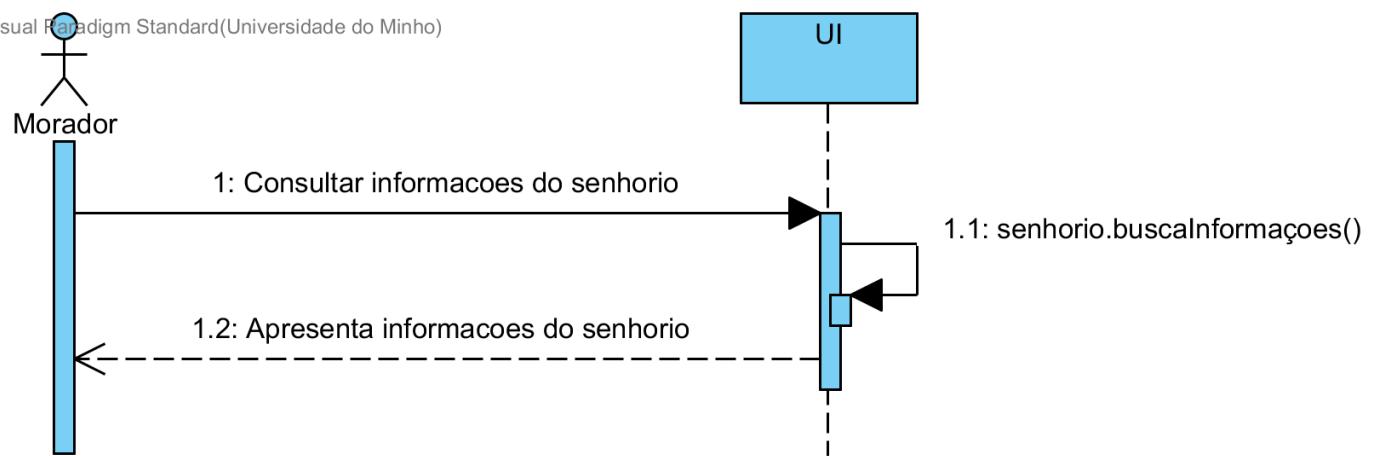


Figura 53: Diagrama de Sequência para Consultar Informação de Senhorio, com subsistemas.

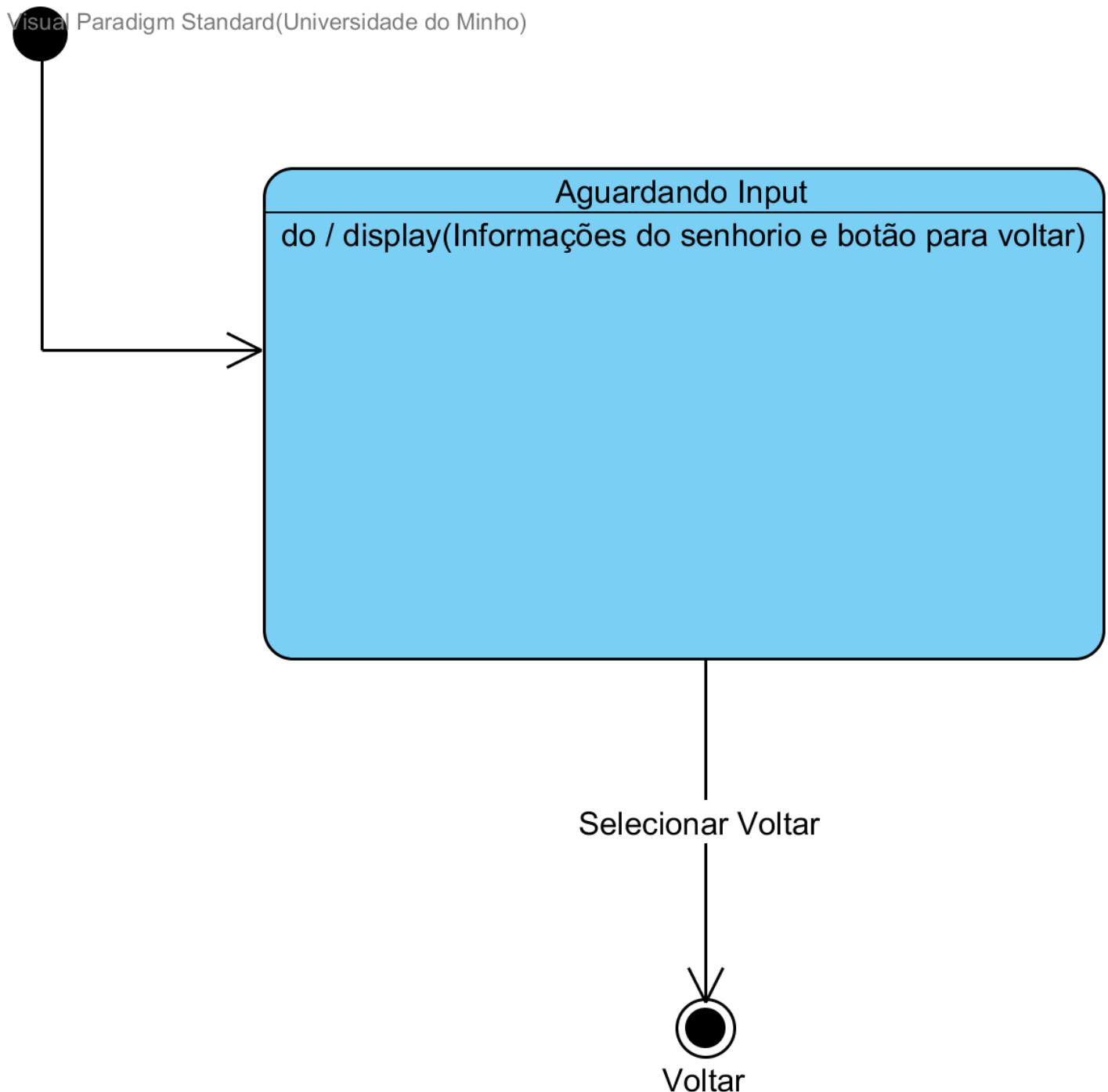


Figura 54: Submenu para Consultar Informação de Senhorio.

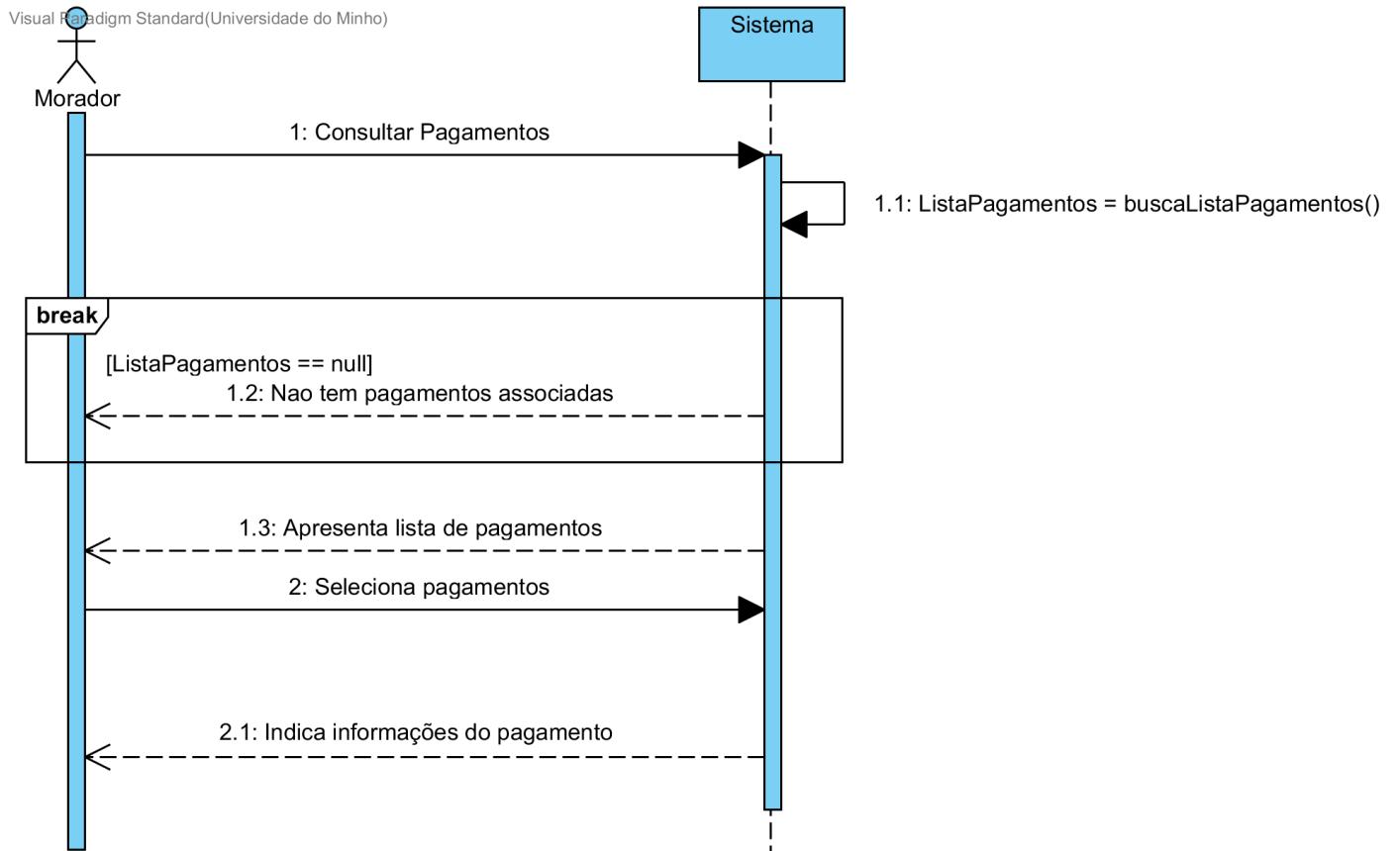


Figura 55: Diagrama de Sequência para Consultar Pagamentos.

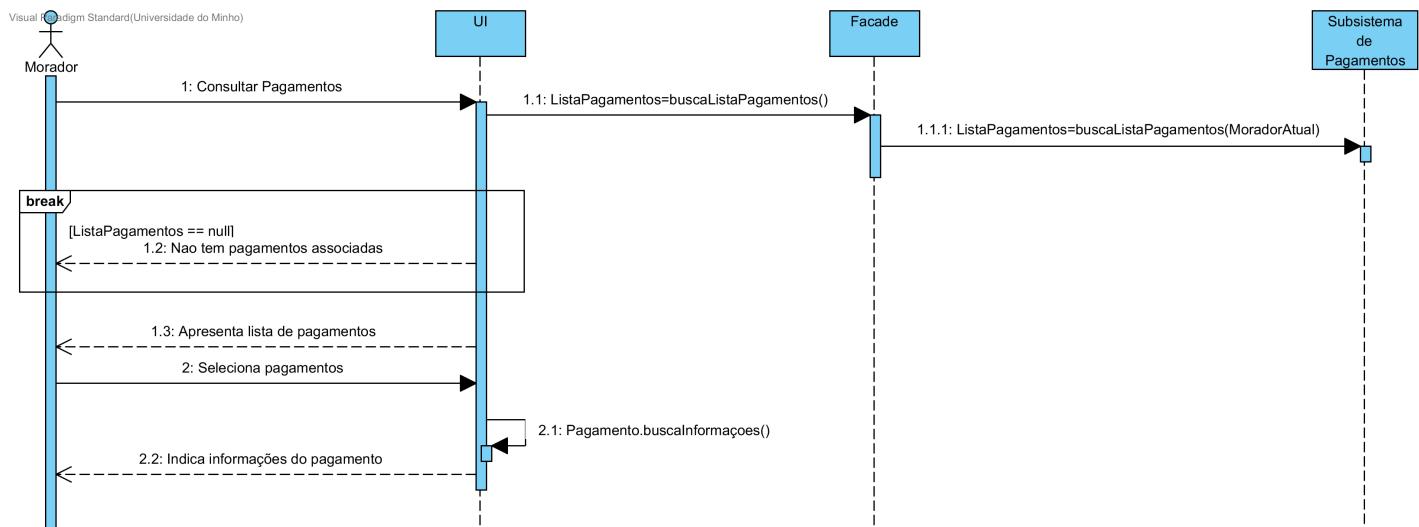


Figura 56: Diagrama de Sequência para Consultar Pagamentos, com subsistemas.

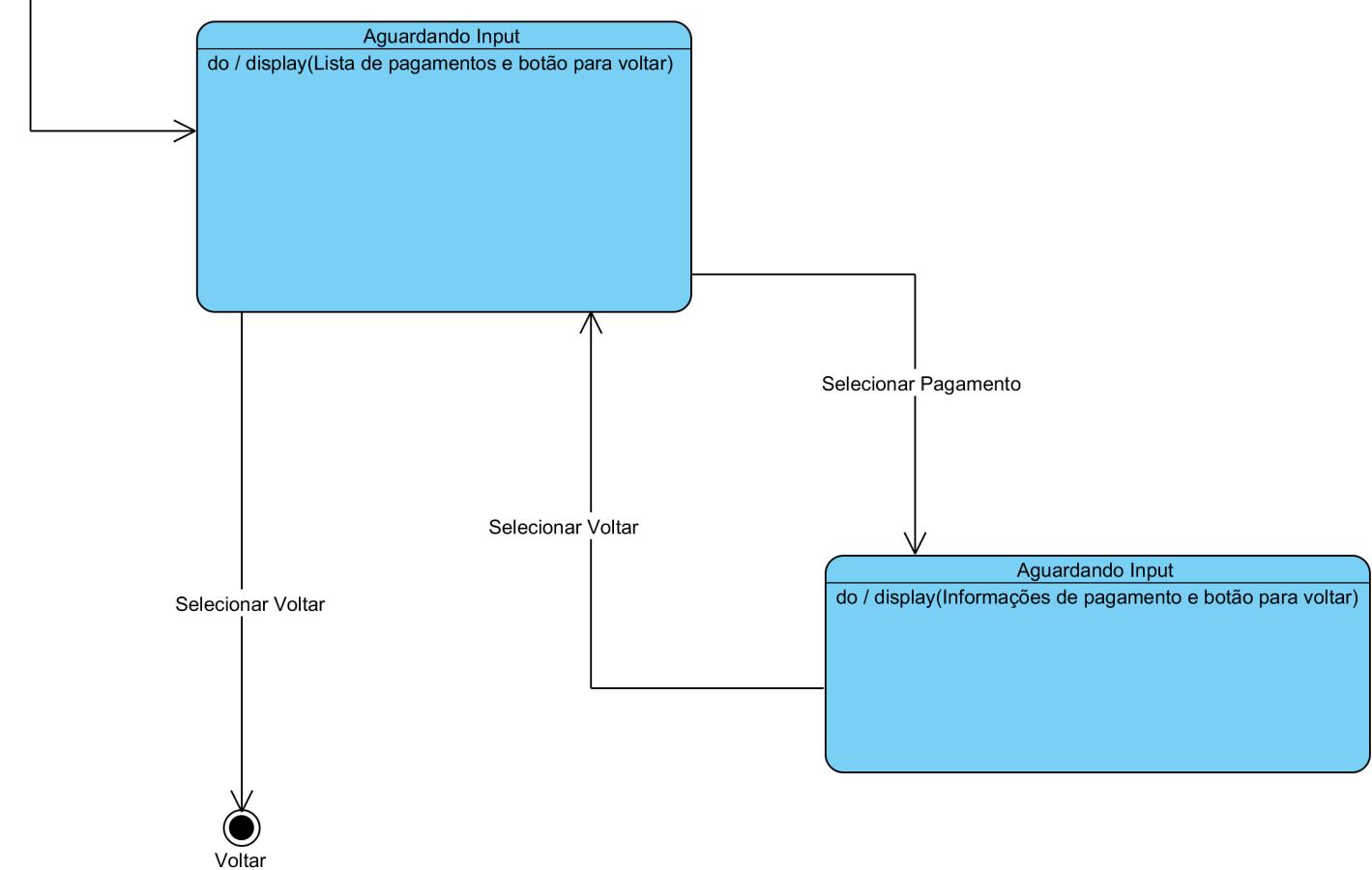
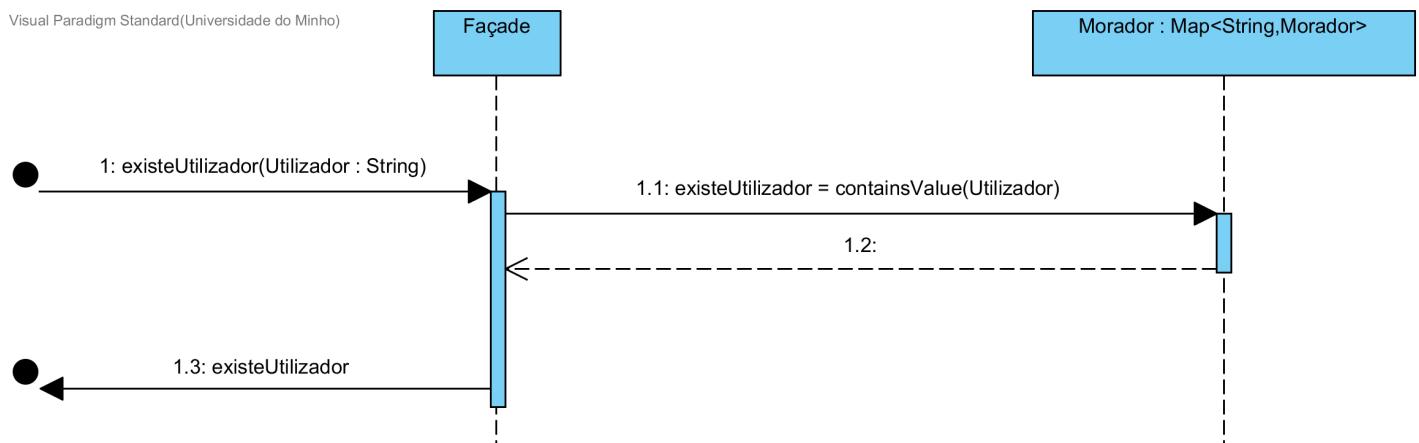


Figura 57: Submenu para Consultar Pagamentos.

Figura 58: Diagrama de Implementação do método `existeUtilizador(Utilizador : String) : boolean`.

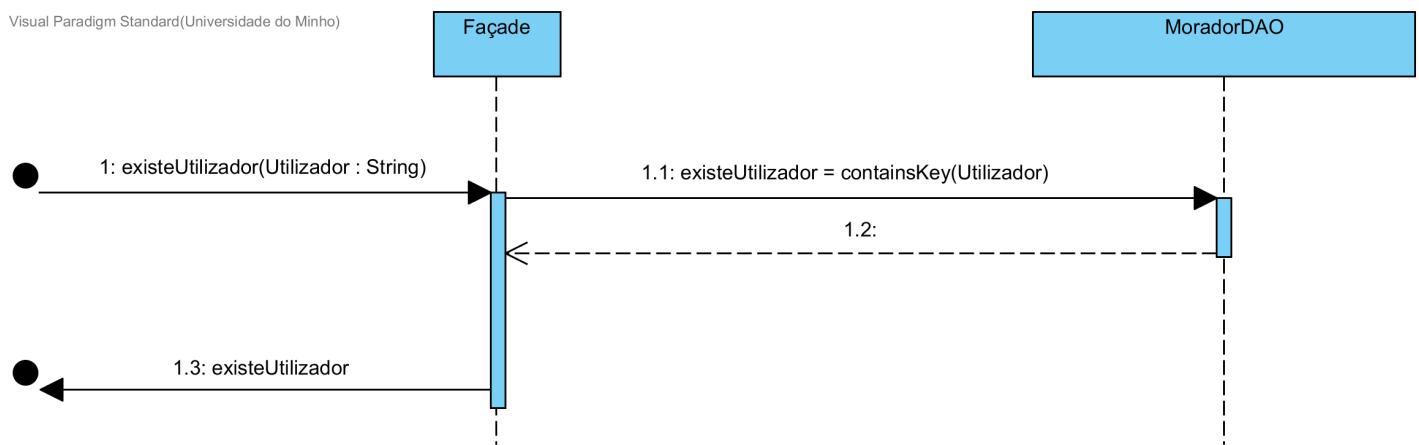


Figura 59: Diagrama de Implementação do método `existeUtilizador(Utilizador : String) : boolean` com DAOs.

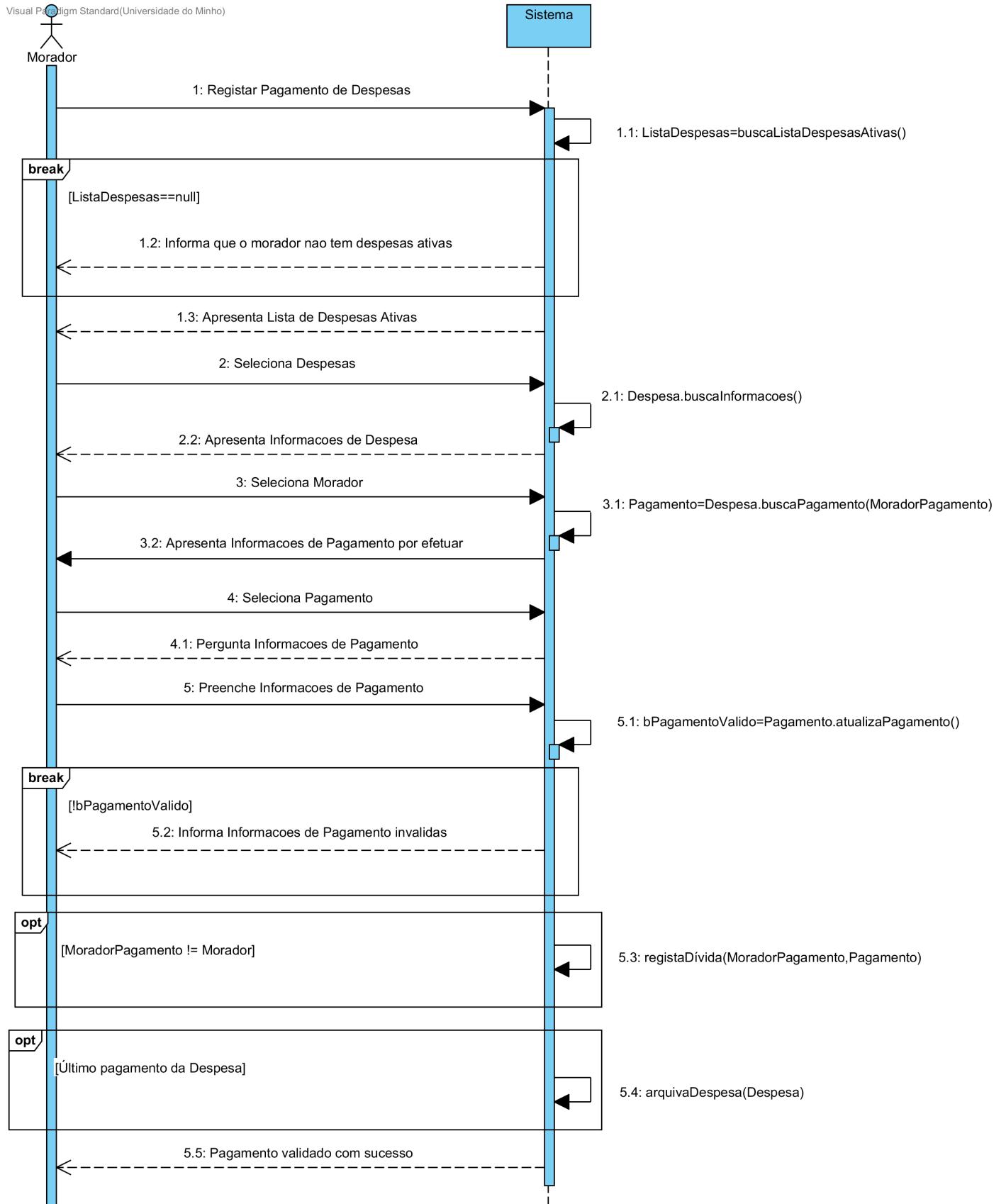


Figura 60: Diagrama de Sequência para Pagar Despesa.

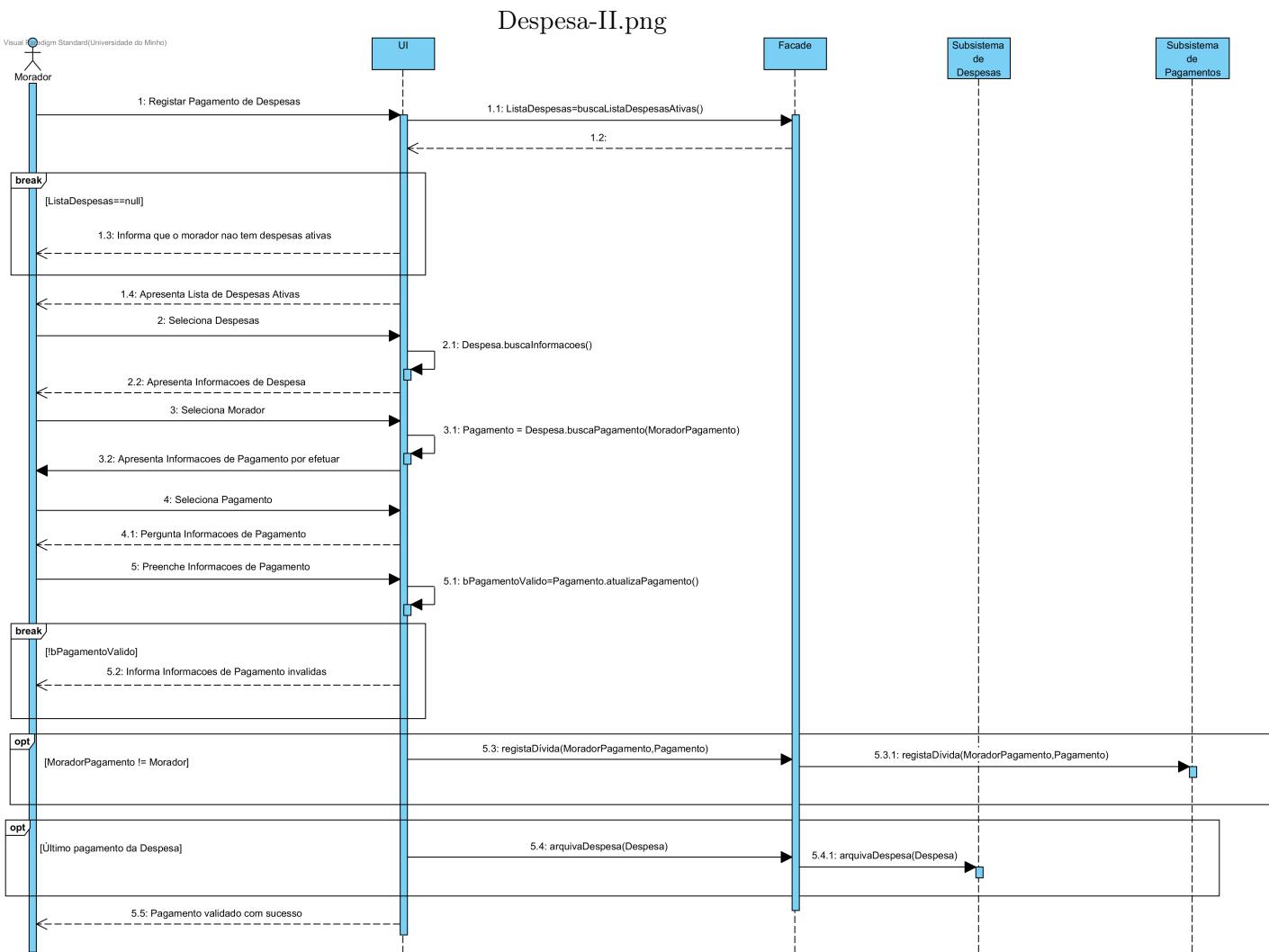


Figura 61: Diagrama de Sequência para Pagar Despesa, com subsistema.

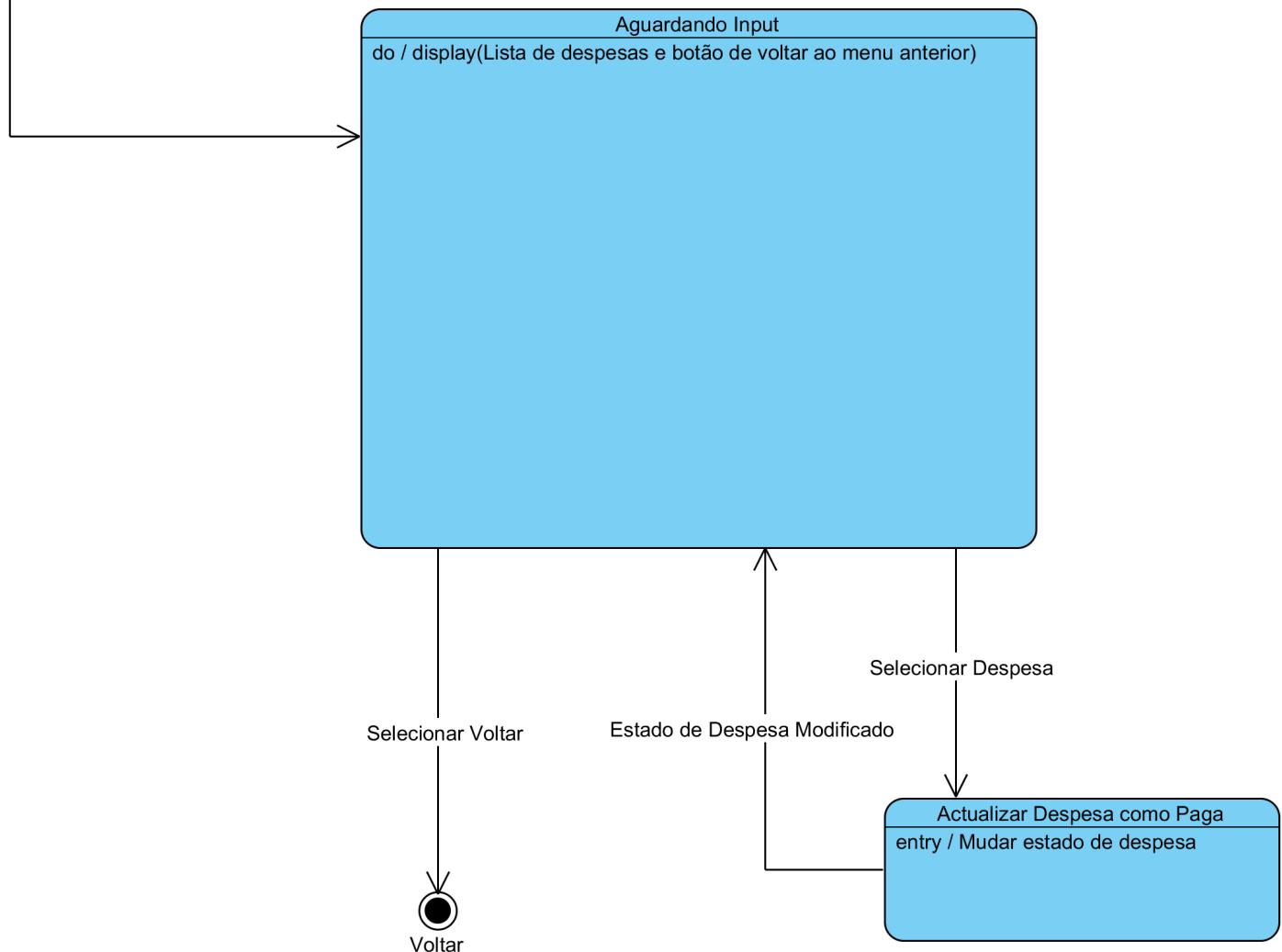


Figura 62: Submenu para Pagar Despesa.

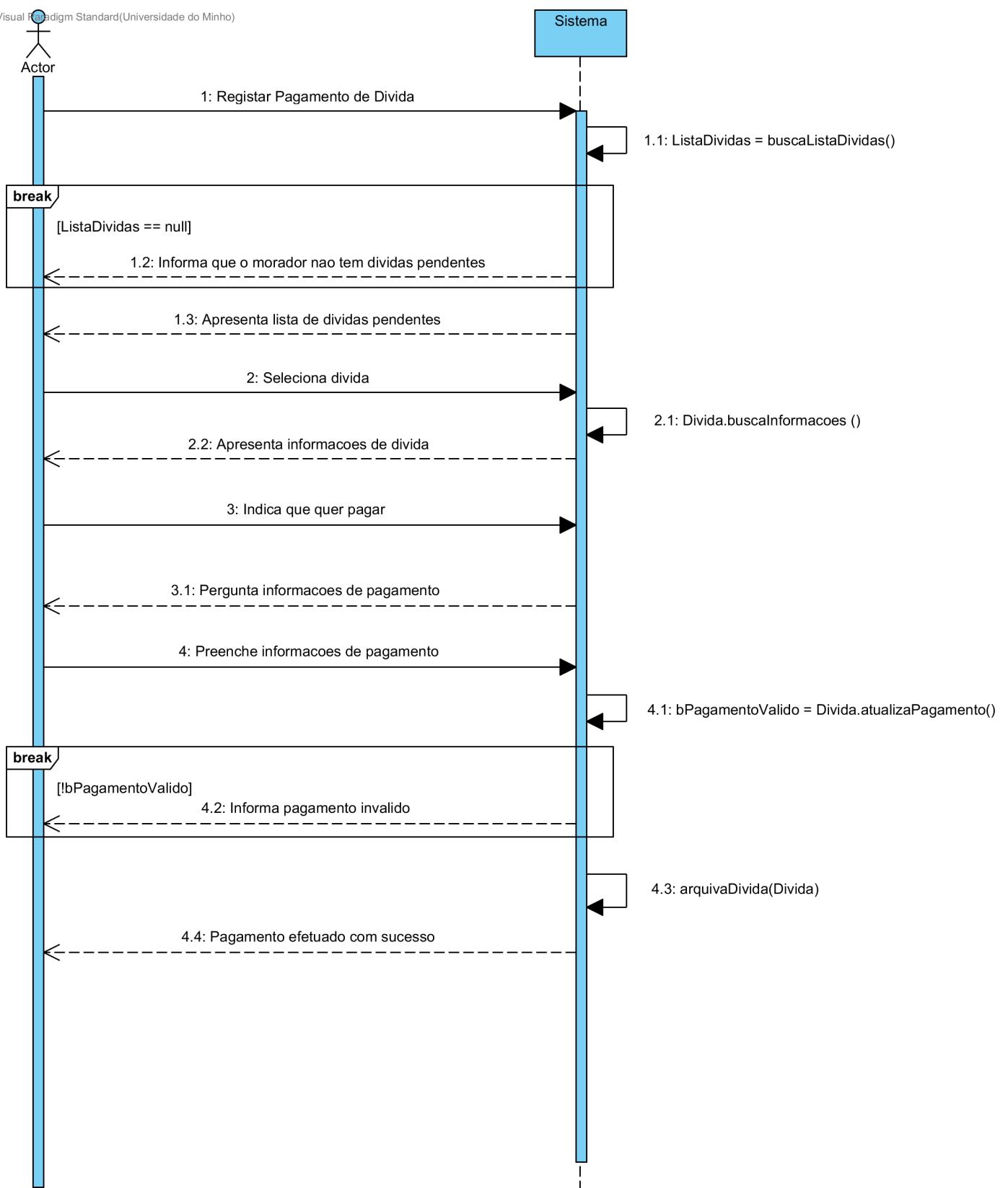


Figura 63: Diagrama de Sequência para Pagar Dívida.

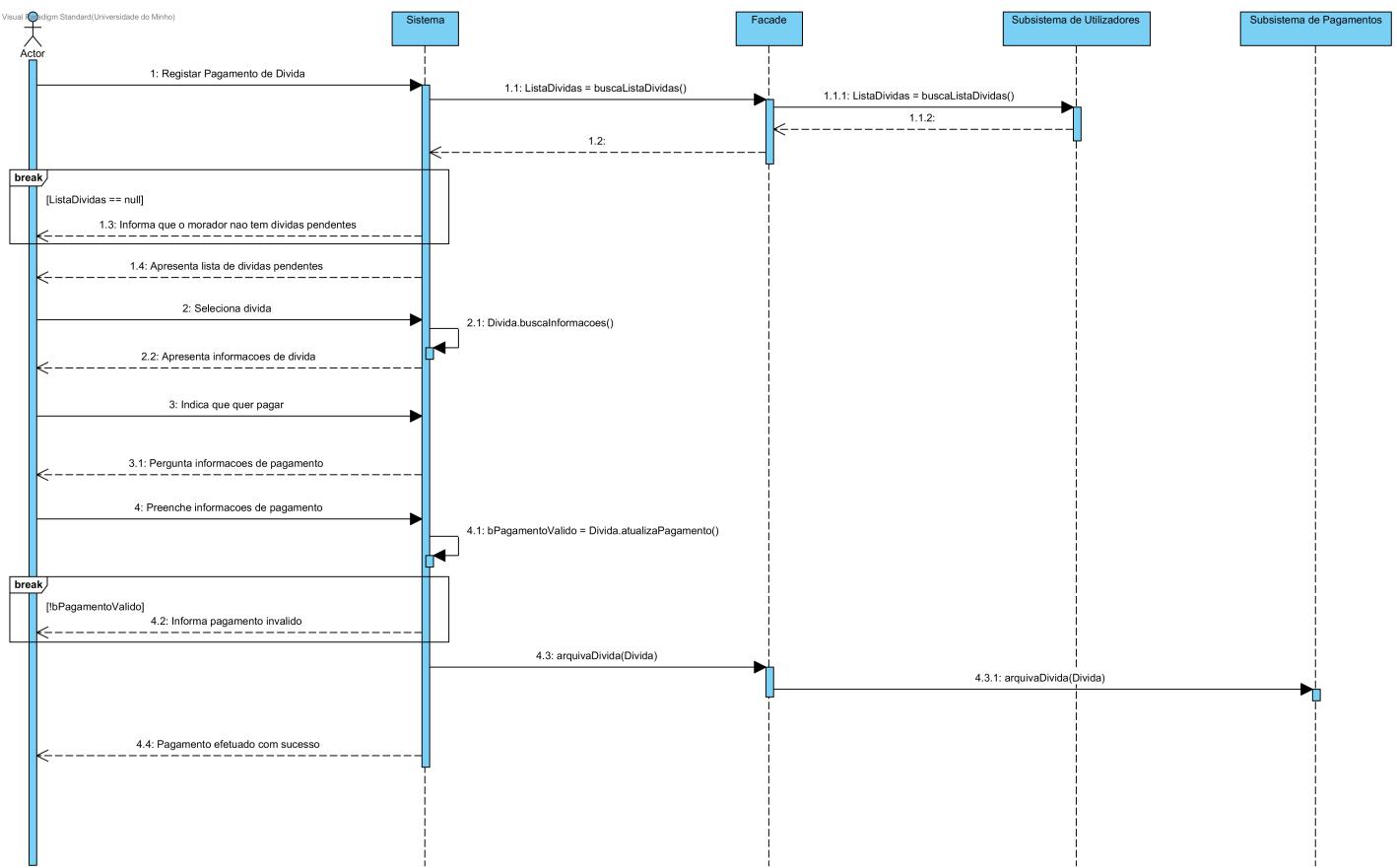


Figura 64: Diagrama de Sequência para Pagar Dívida, com subsistemas.

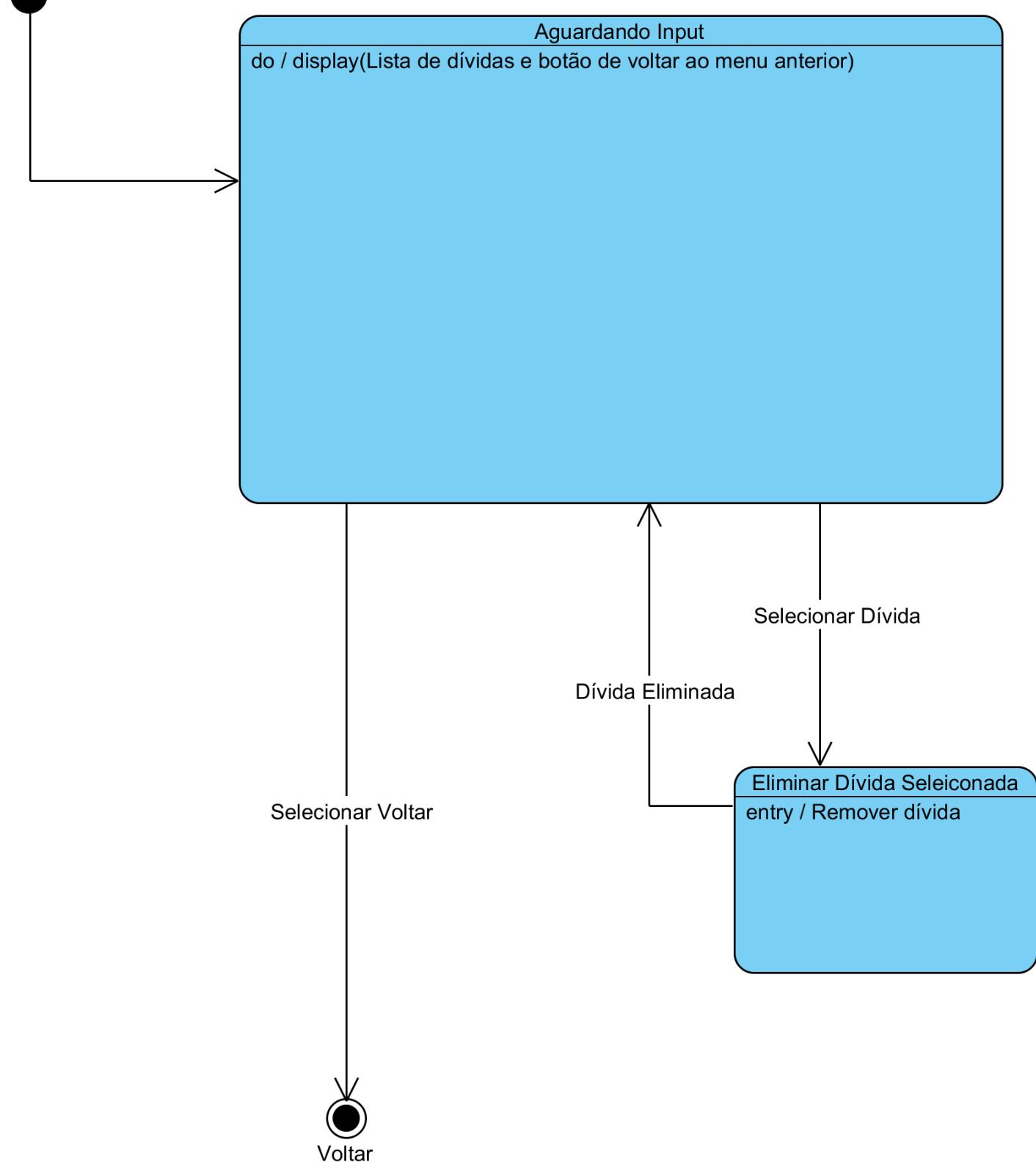


Figura 65: Submenu para Pagar Dívida.

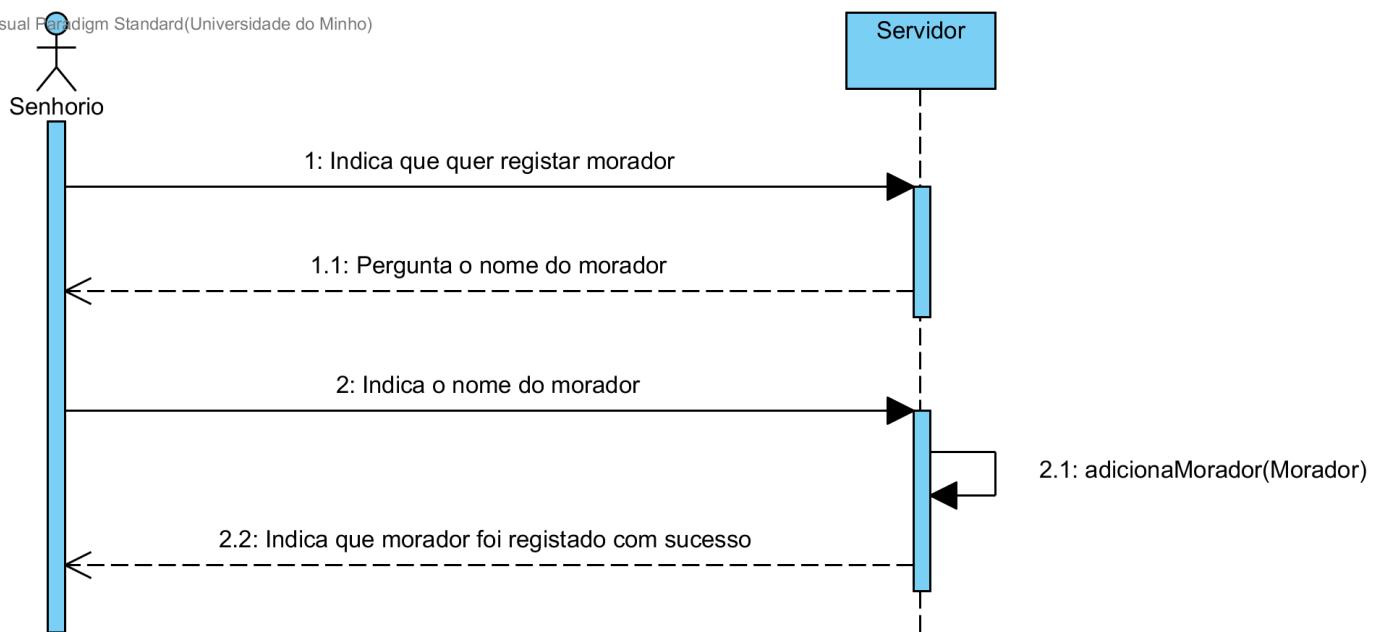


Figura 66: Diagrama de Sequência para Registar Morador.

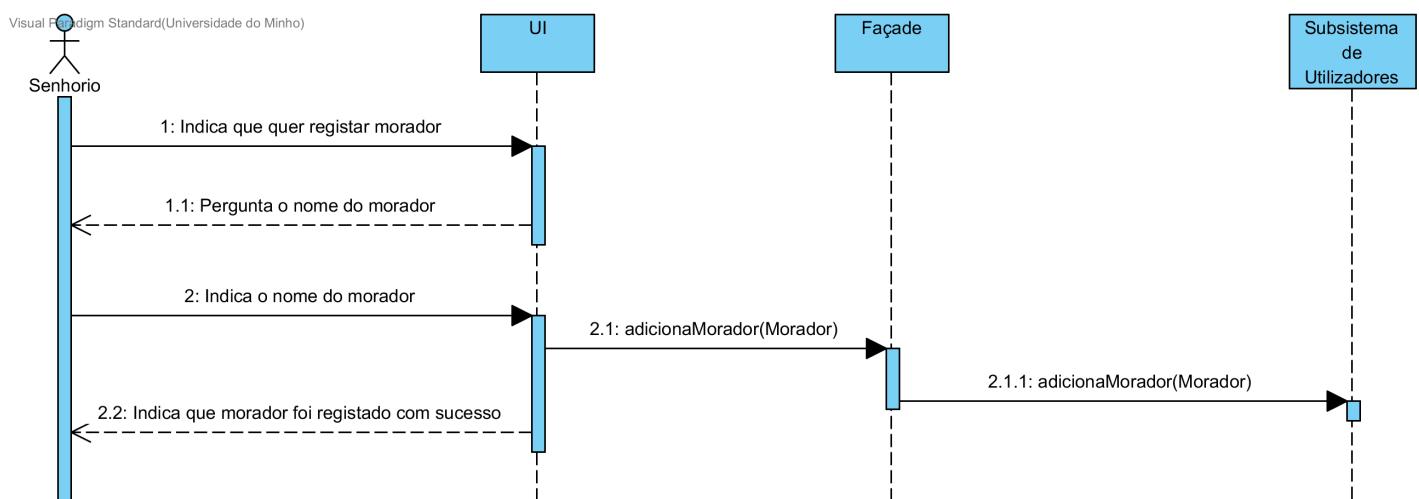


Figura 67: Diagrama de Sequência para Registar Morador, com subsistemas.

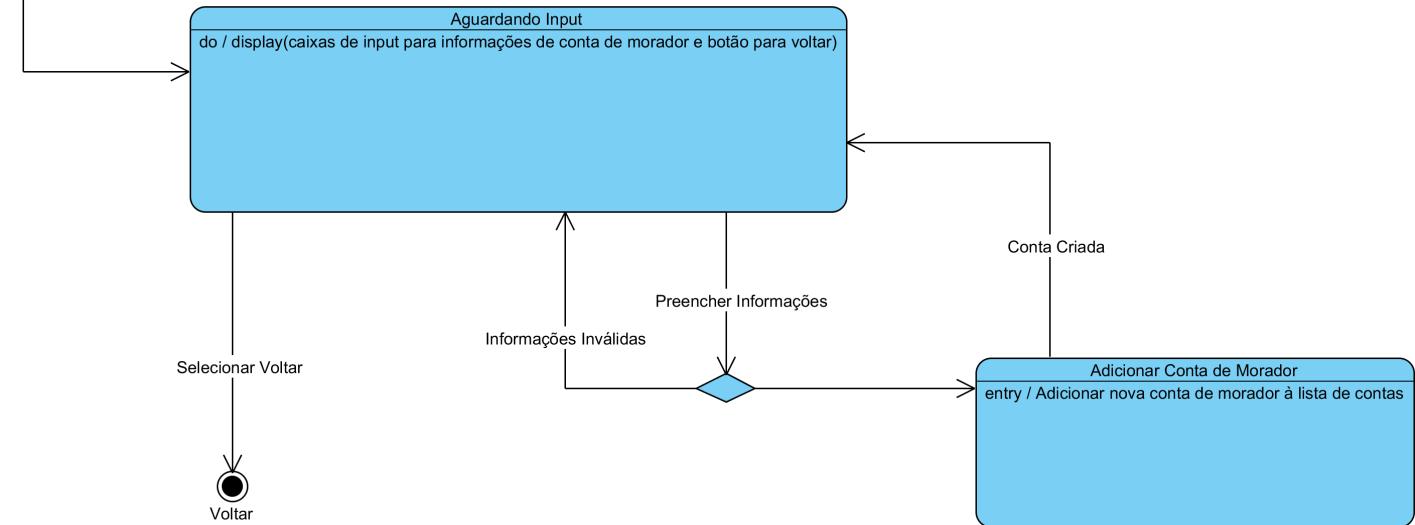
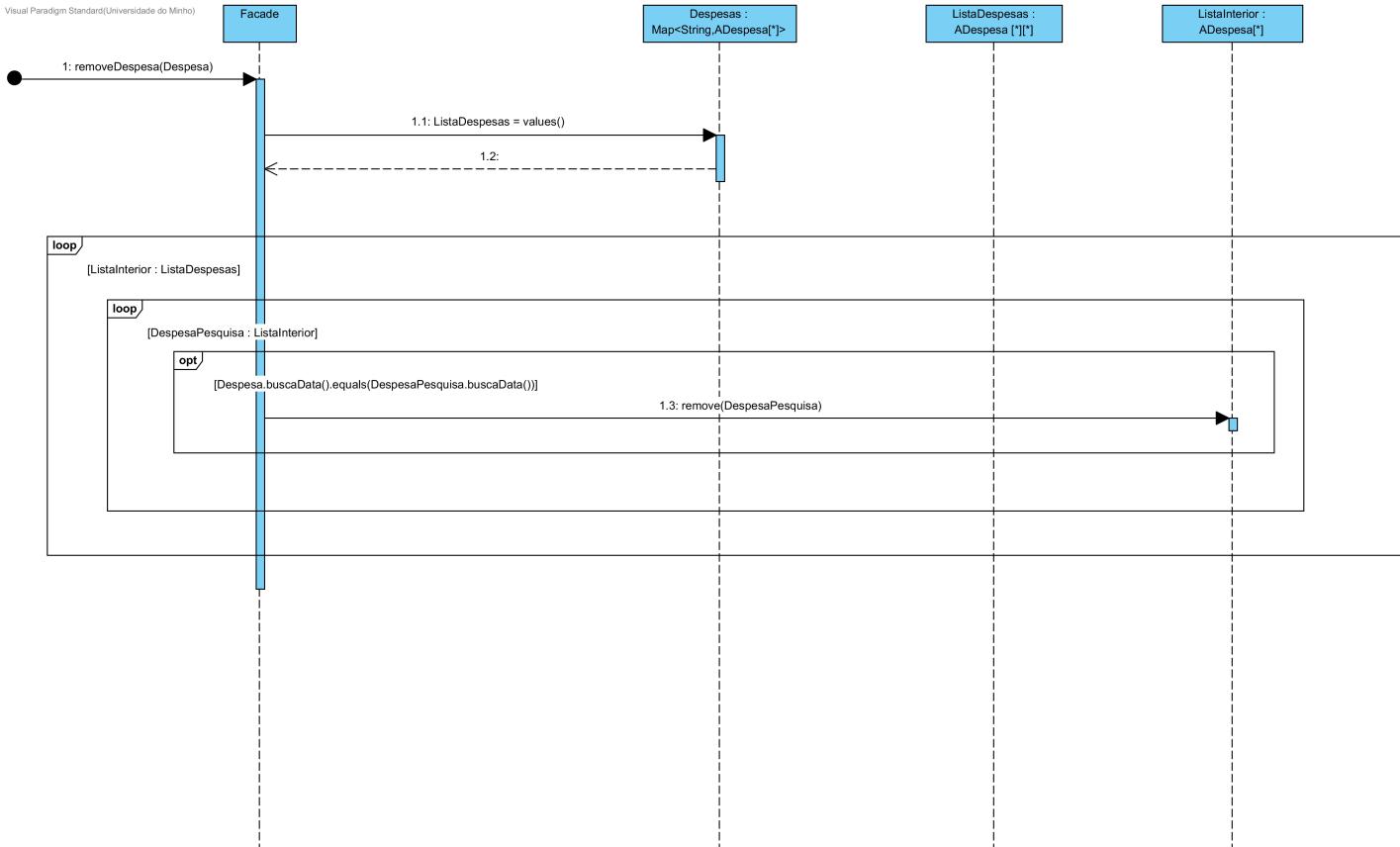


Figura 68: Submenu para Registar Morador.

Figura 69: Diagrama de Implementação do método `removeDespesa(Despesa : ADespesa)`.

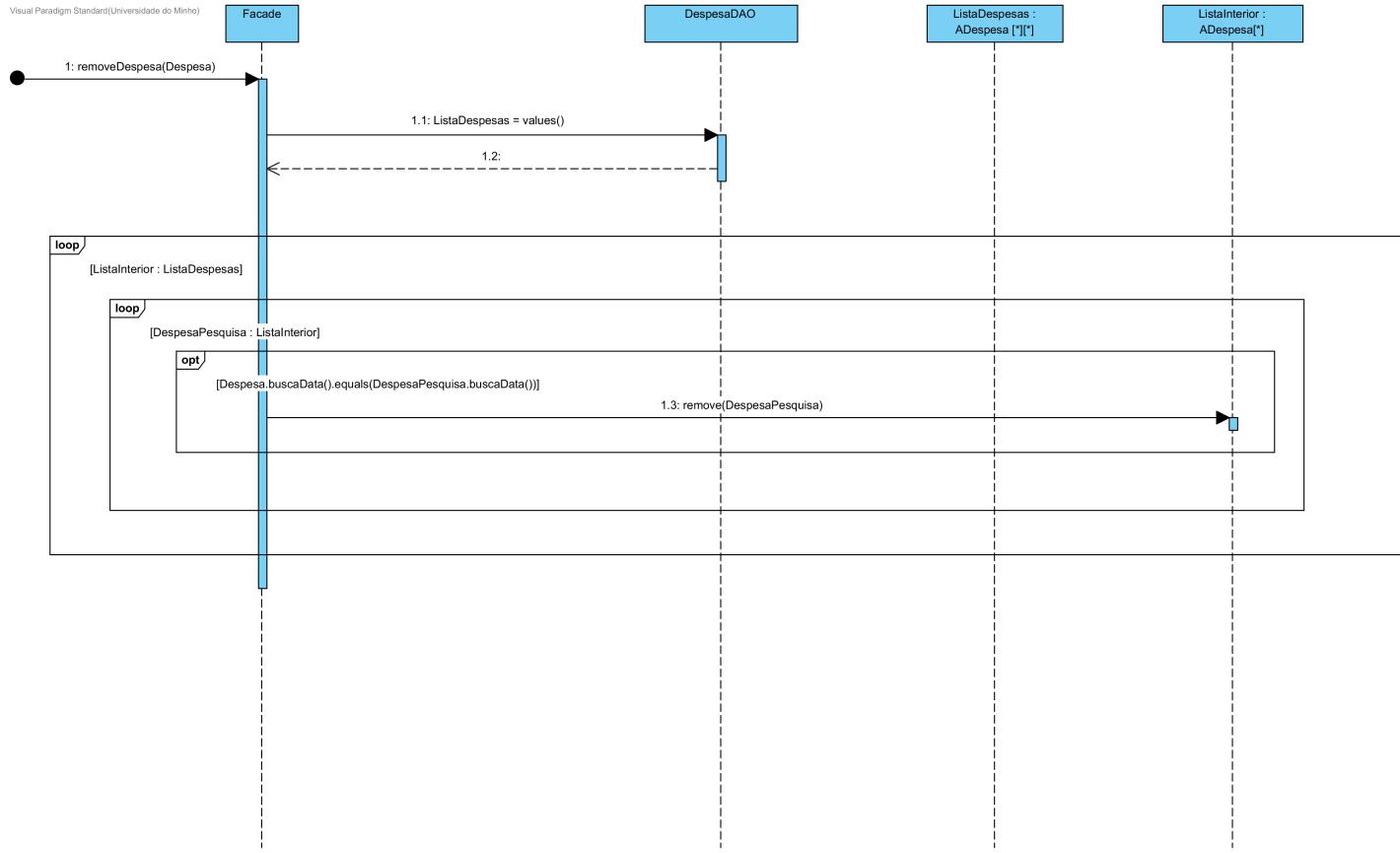


Figura 70: Diagrama de Implementação do método `removeDespesa(Despesa : ADespesa)` com DAOs.

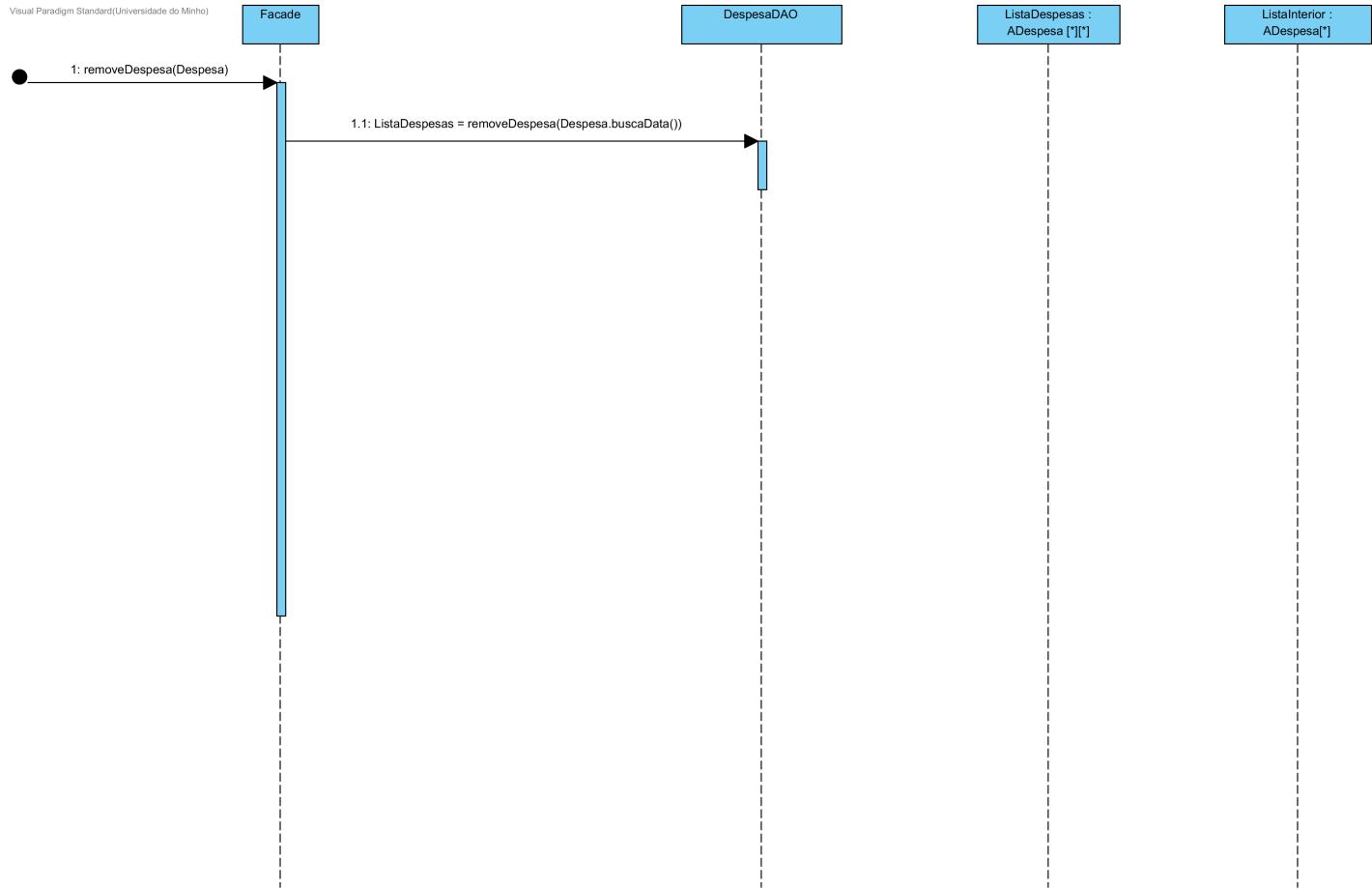


Figura 71: Diagrama de alternativa de Implementação do método `removeDespesa(Despesa : ADespesa)` com DAOs.

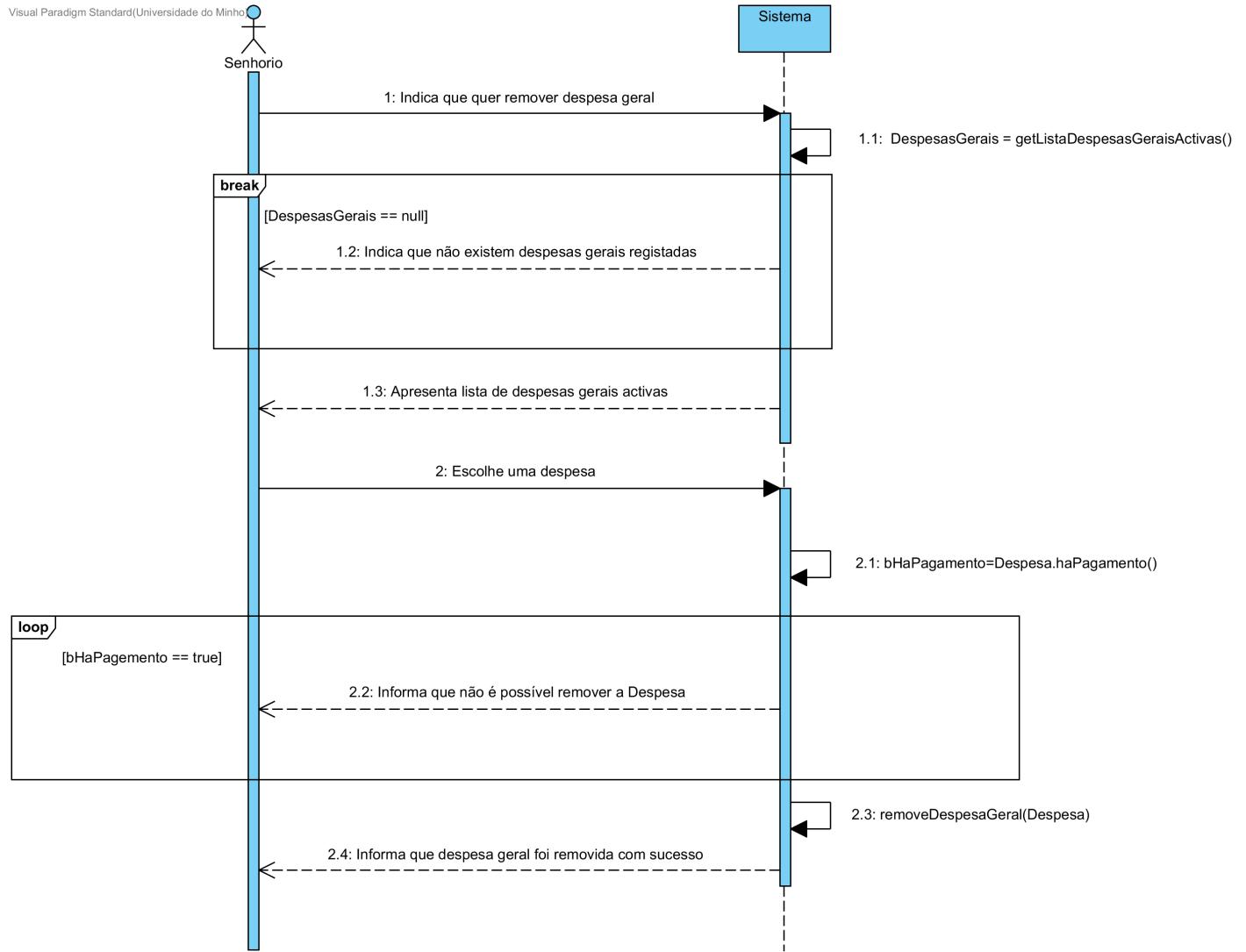


Figura 72: Diagrama de Sequência para Remover Despesa Geral.

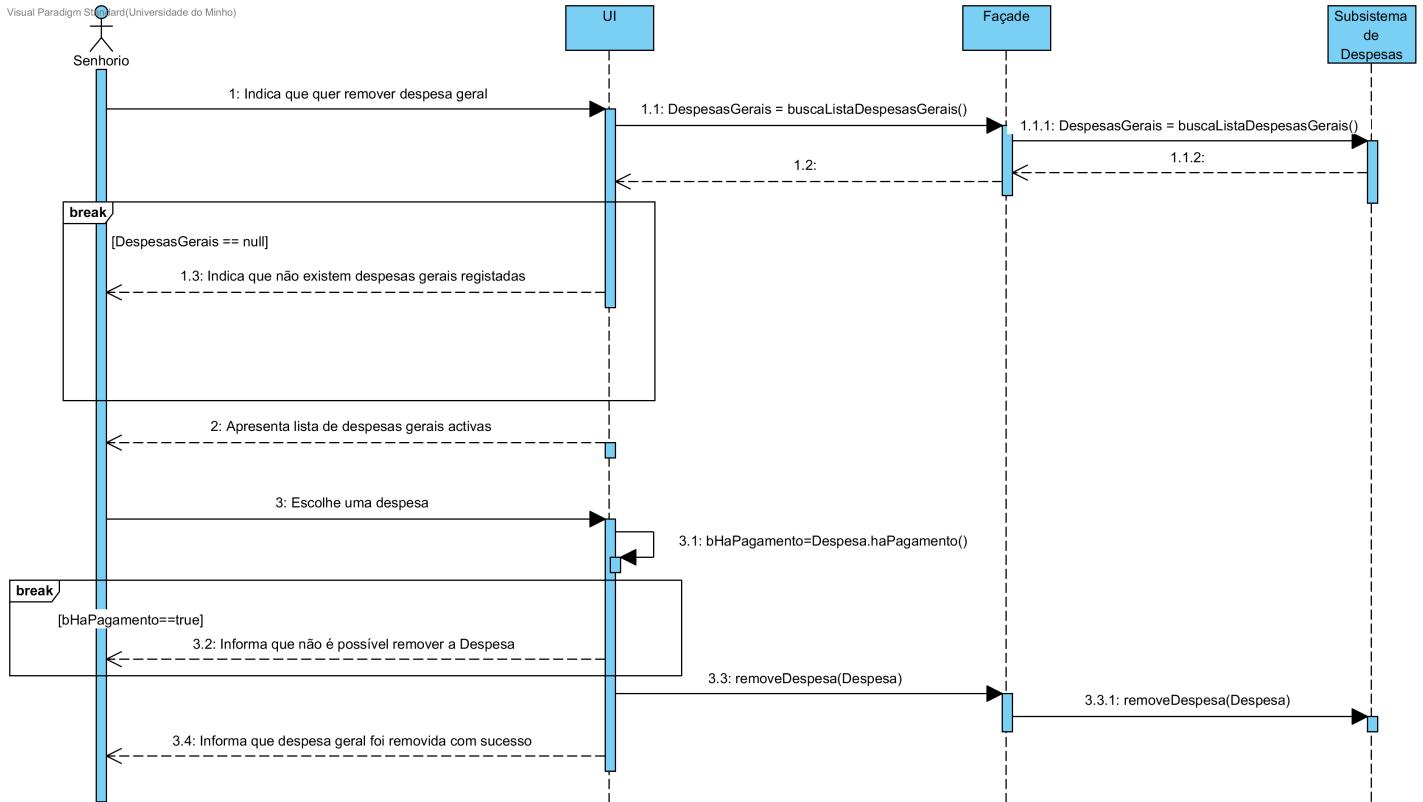


Figura 73: Diagrama de Sequência para Remover Despesa Geral, com subsistemas.

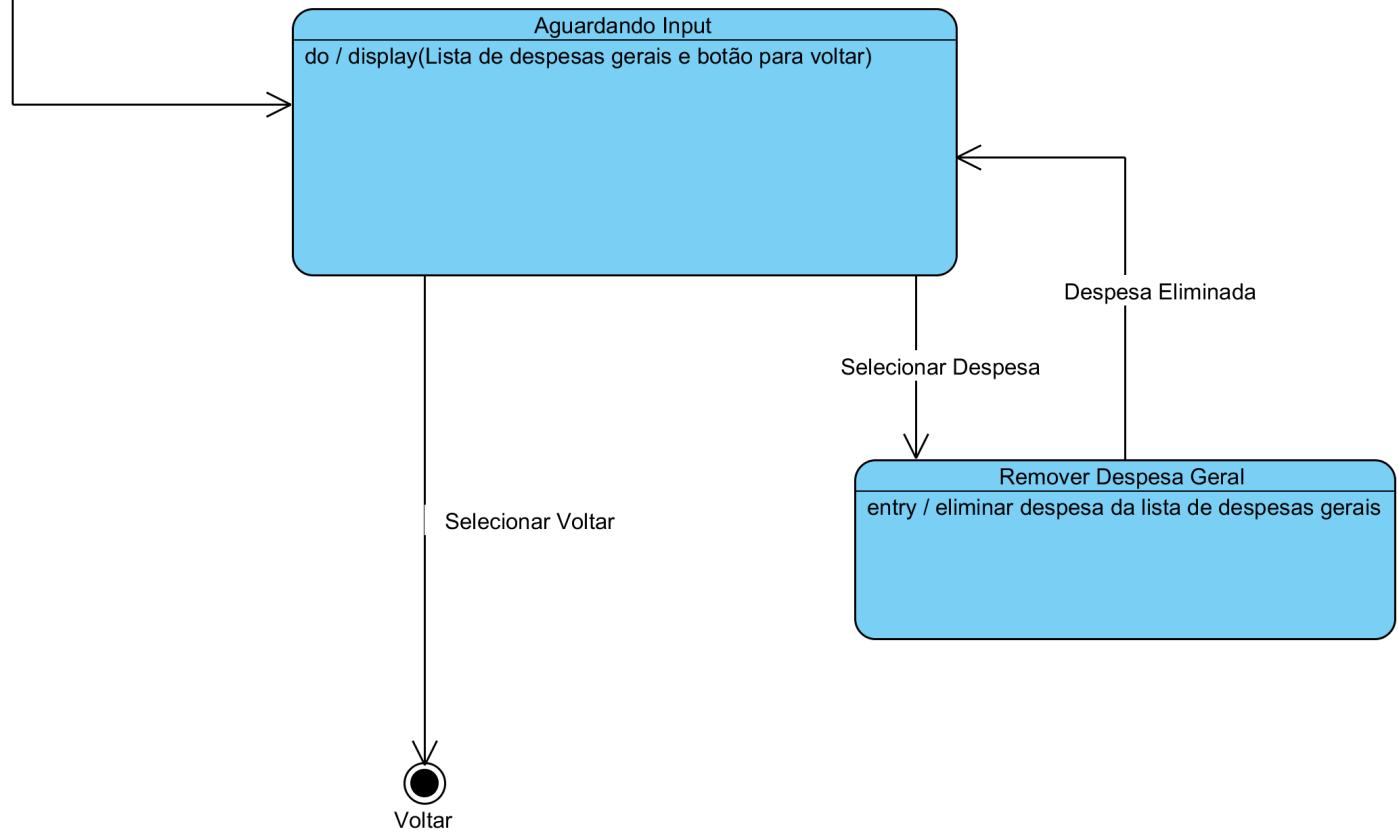


Figura 74: Submenu para Remover Despesa Geral.

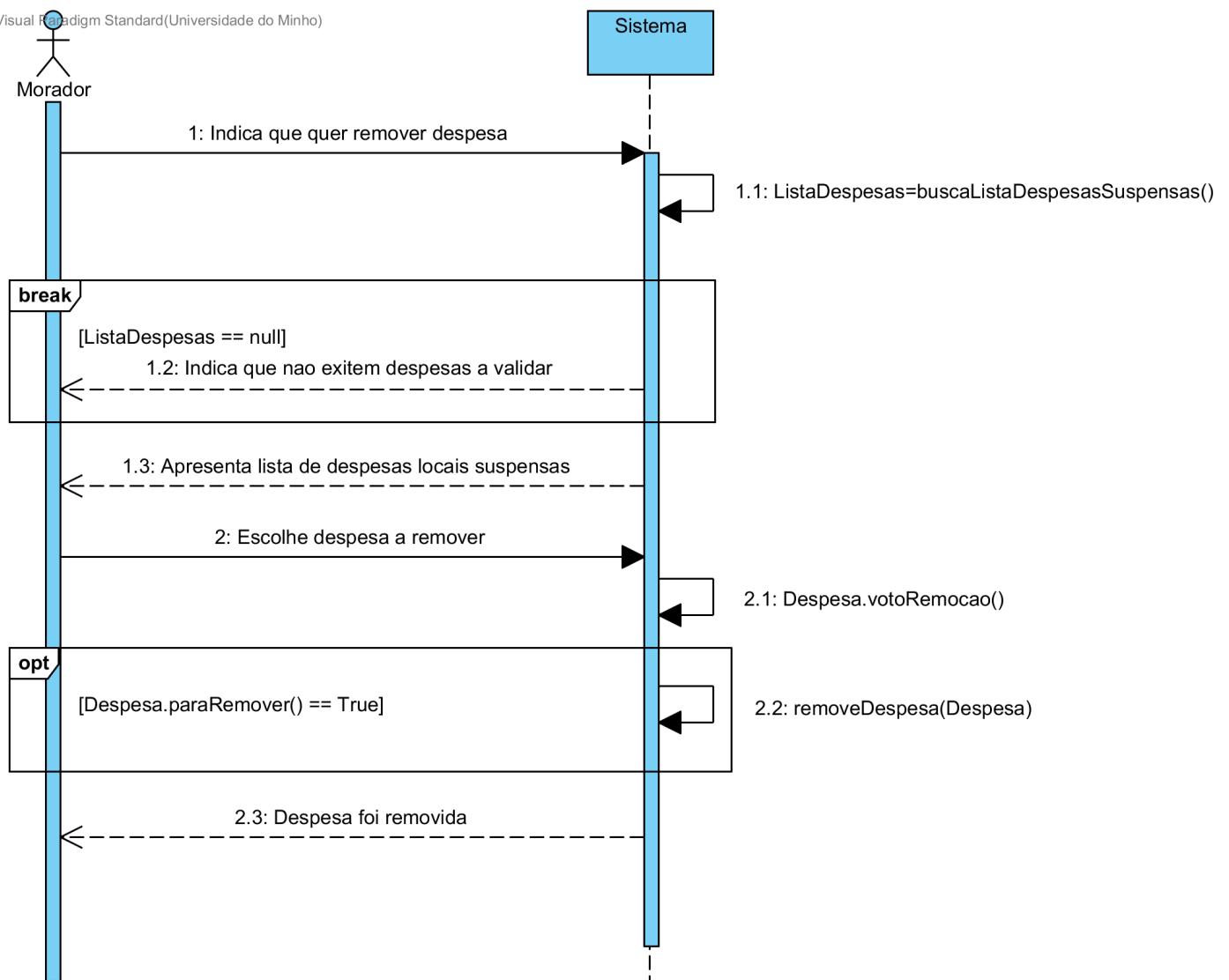


Figura 75: Diagrama de Sequência para Remover Despesa Local.

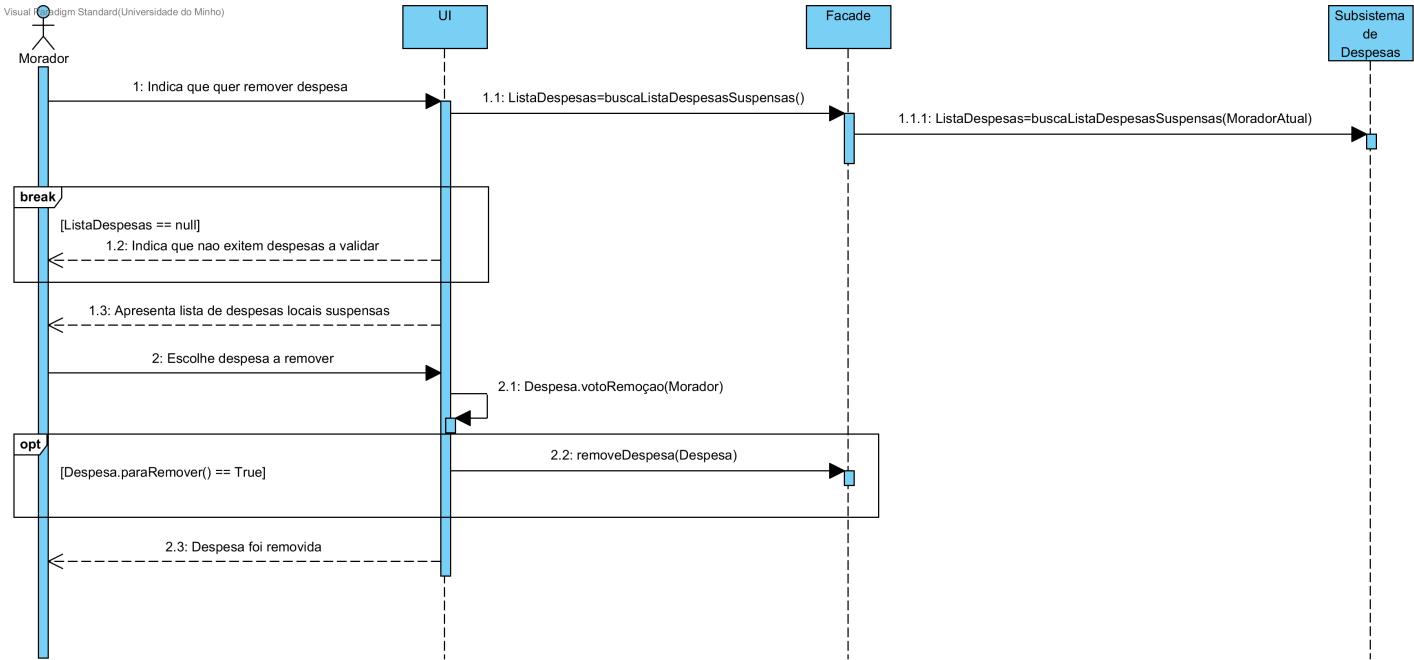


Figura 76: Diagrama de Sequência para Remover Despesa Local, com subsistemas.

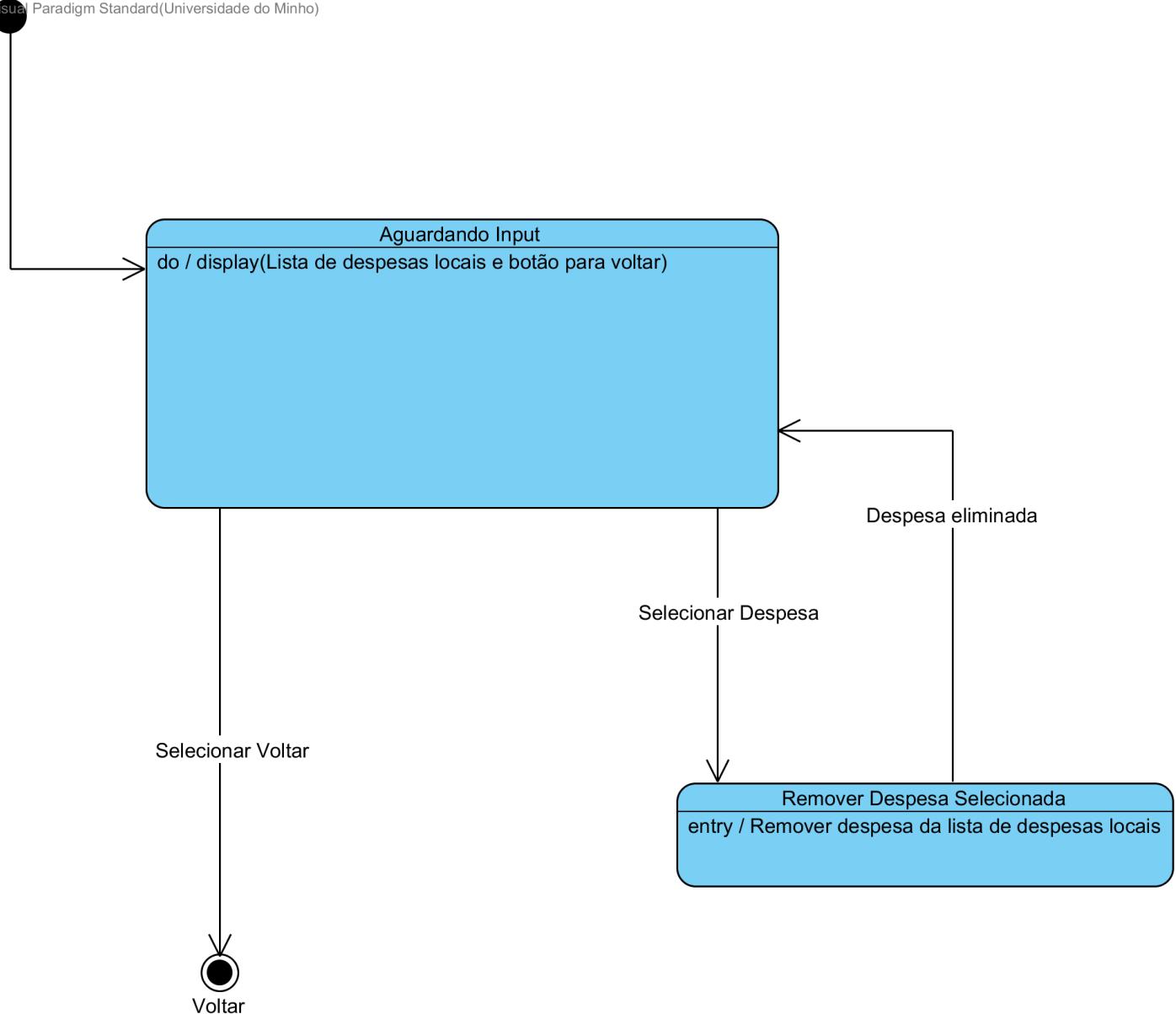


Figura 77: Submenu para Remover Despesa Local.

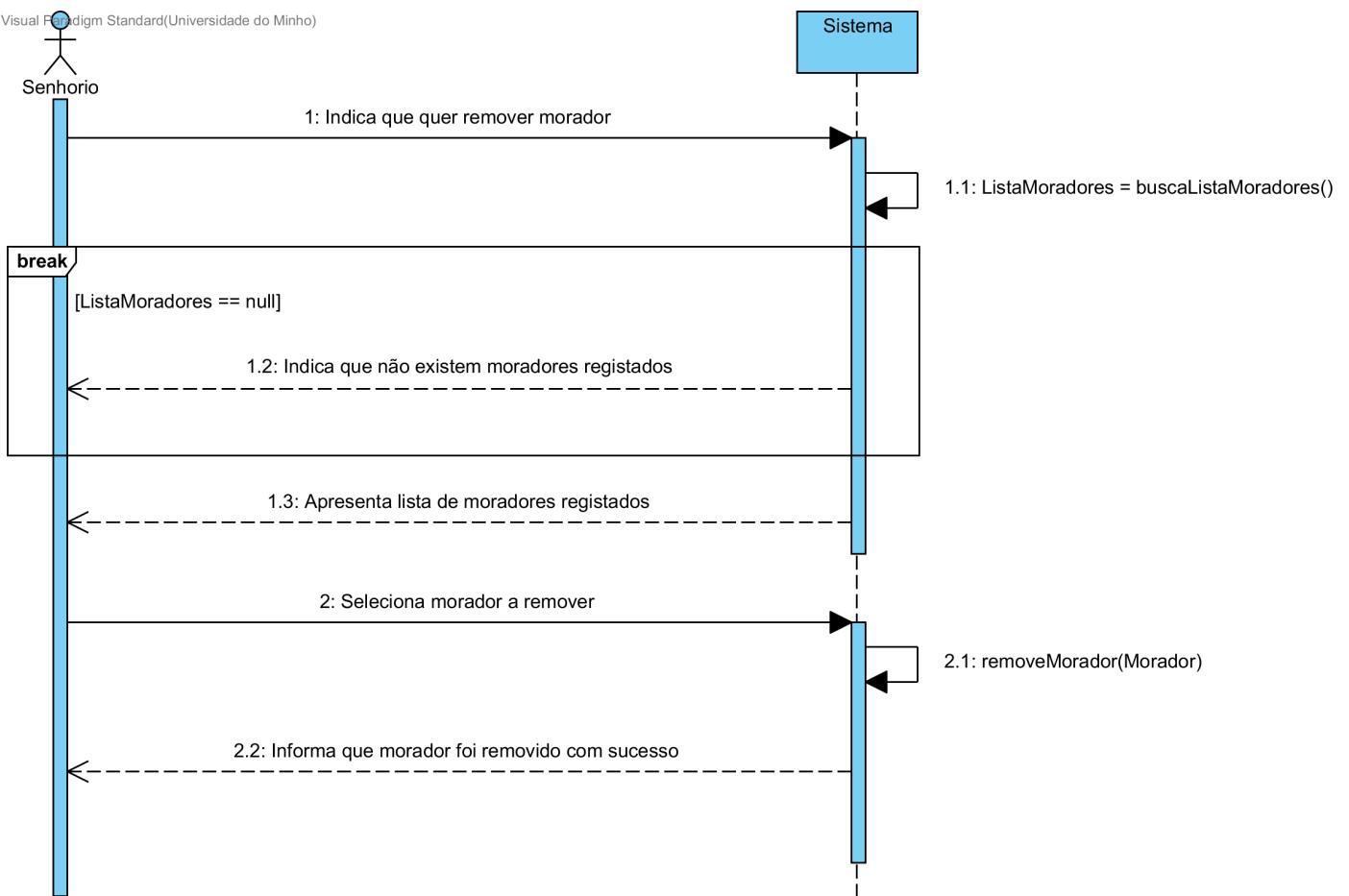


Figura 78: Diagrama de Sequência para Remover Morador.

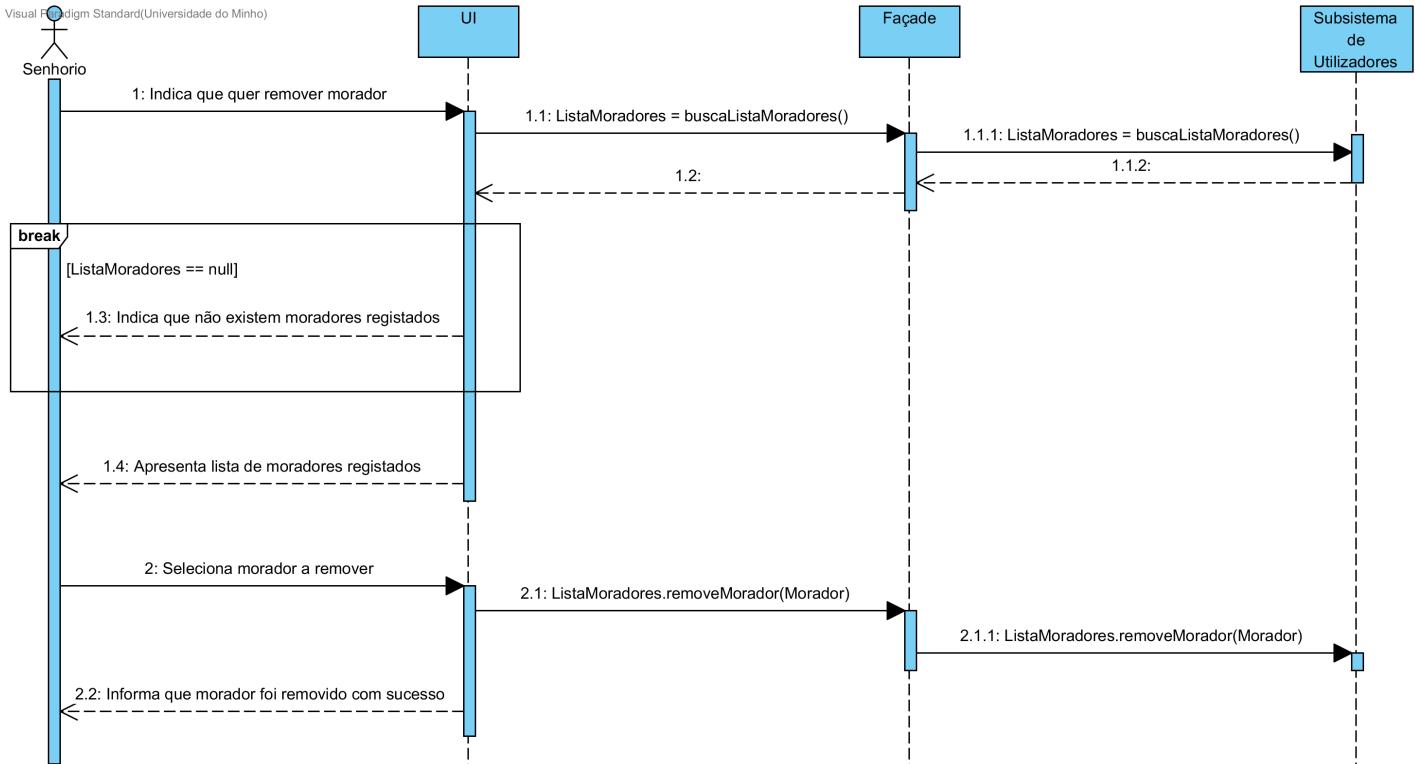


Figura 79: Diagrama de Sequência para Remover Morador, com subsistemas.

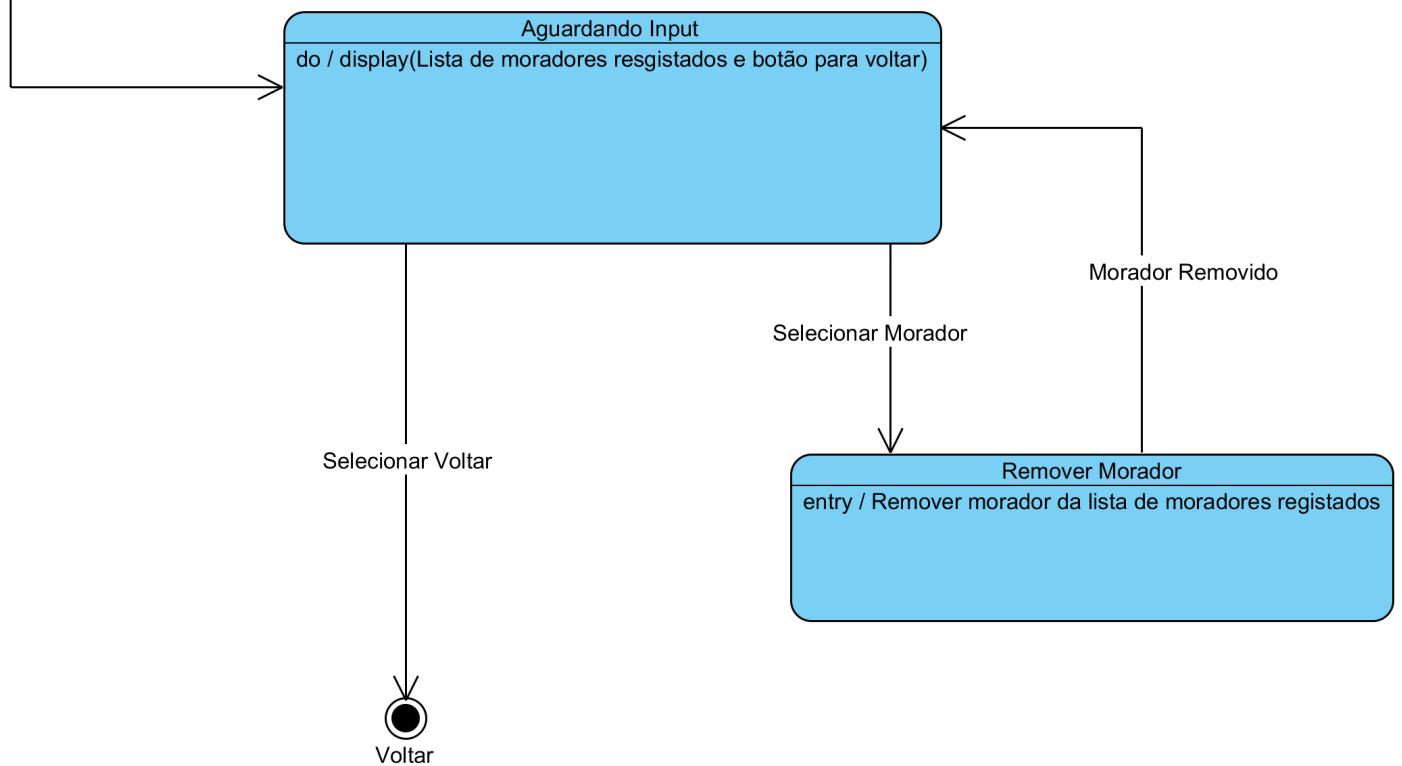


Figura 80: Submenu para Remover Morador.

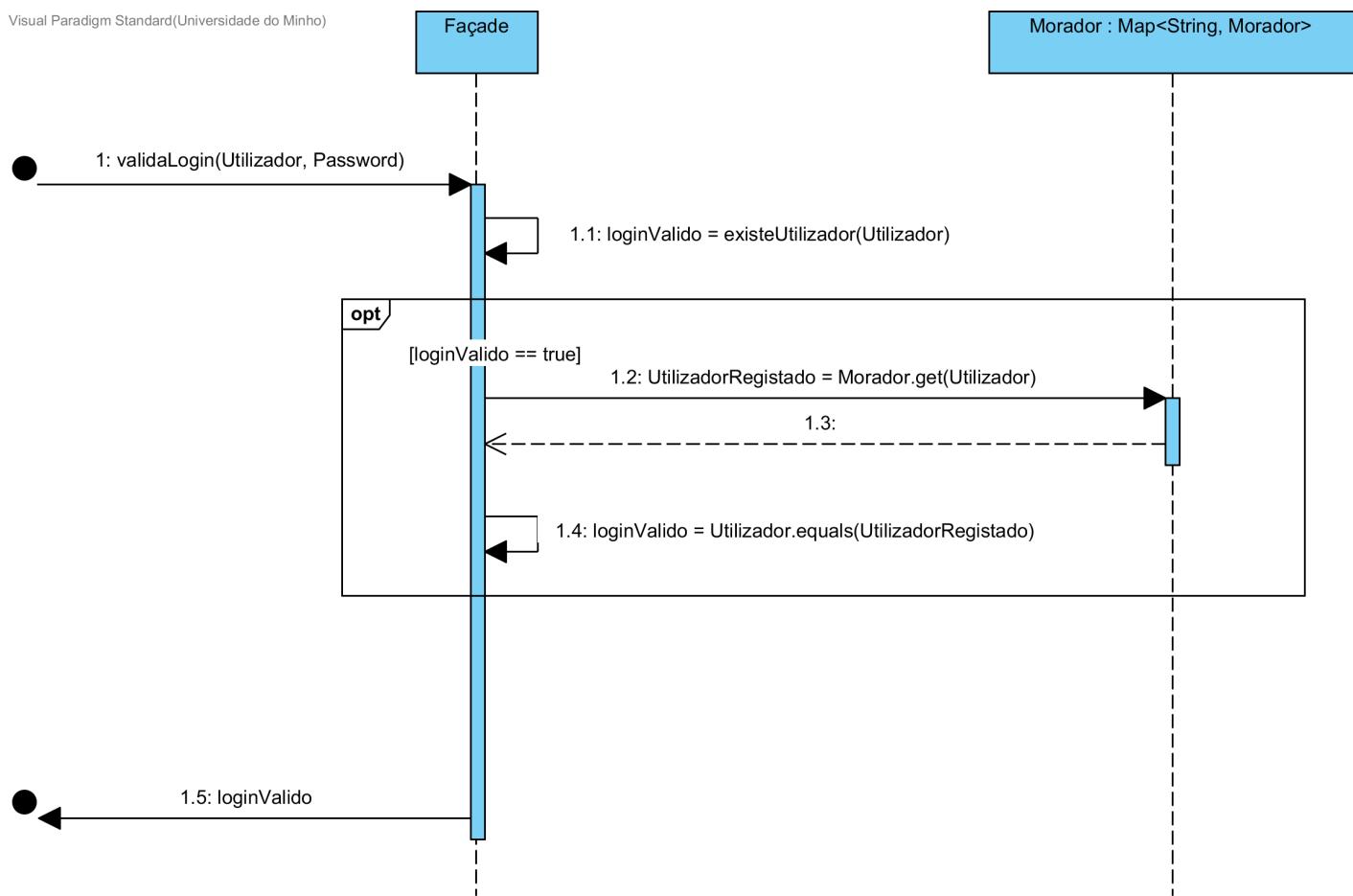


Figura 81: Diagrama de Implementação do método `validaLogin(Utilizador : String, Password : String) : boolean`.

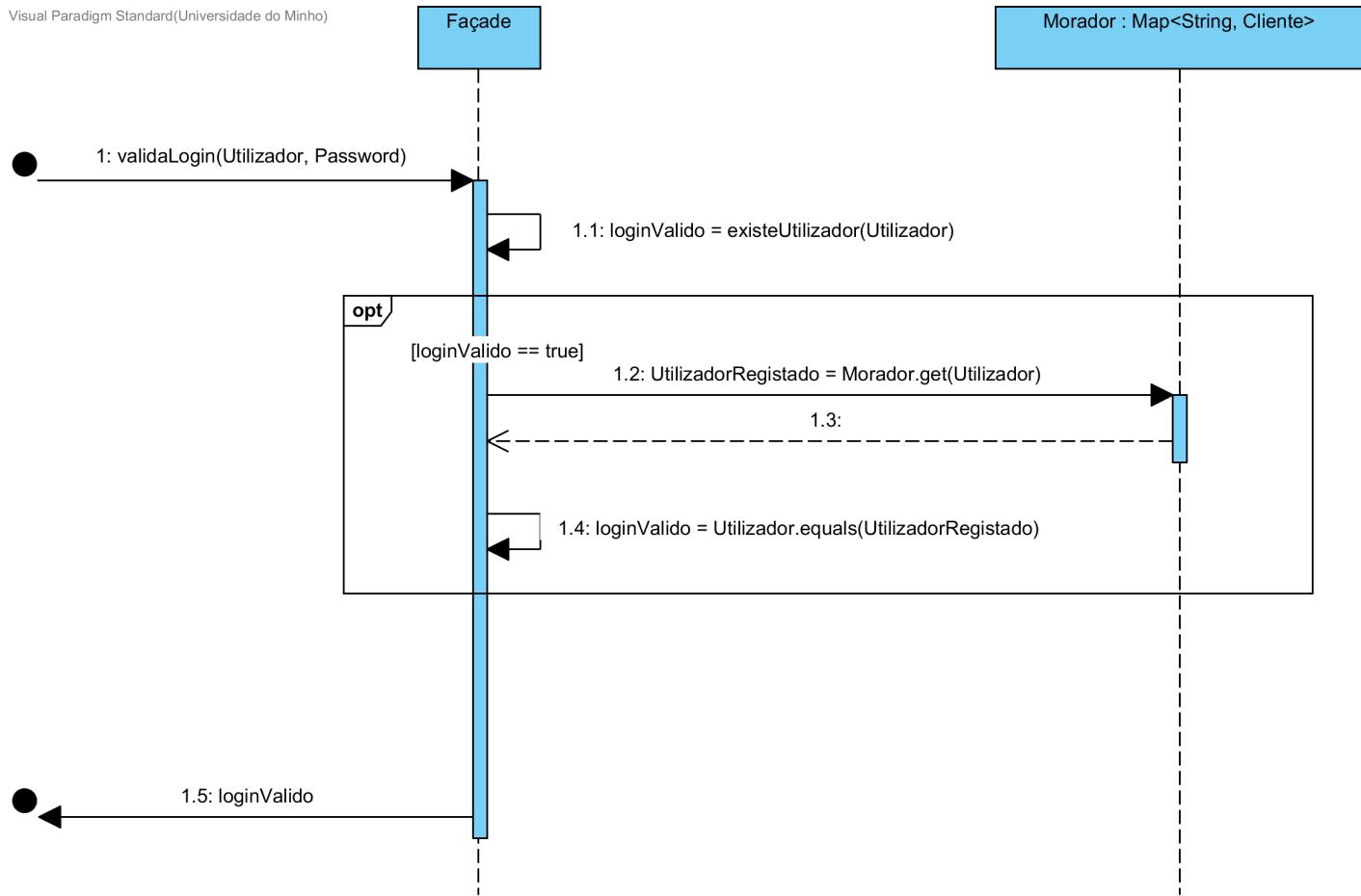


Figura 82: Diagrama alternativo de Implementação do método `validaLogin(Utilizador : String, Password : String) : boolean` com DAOs.

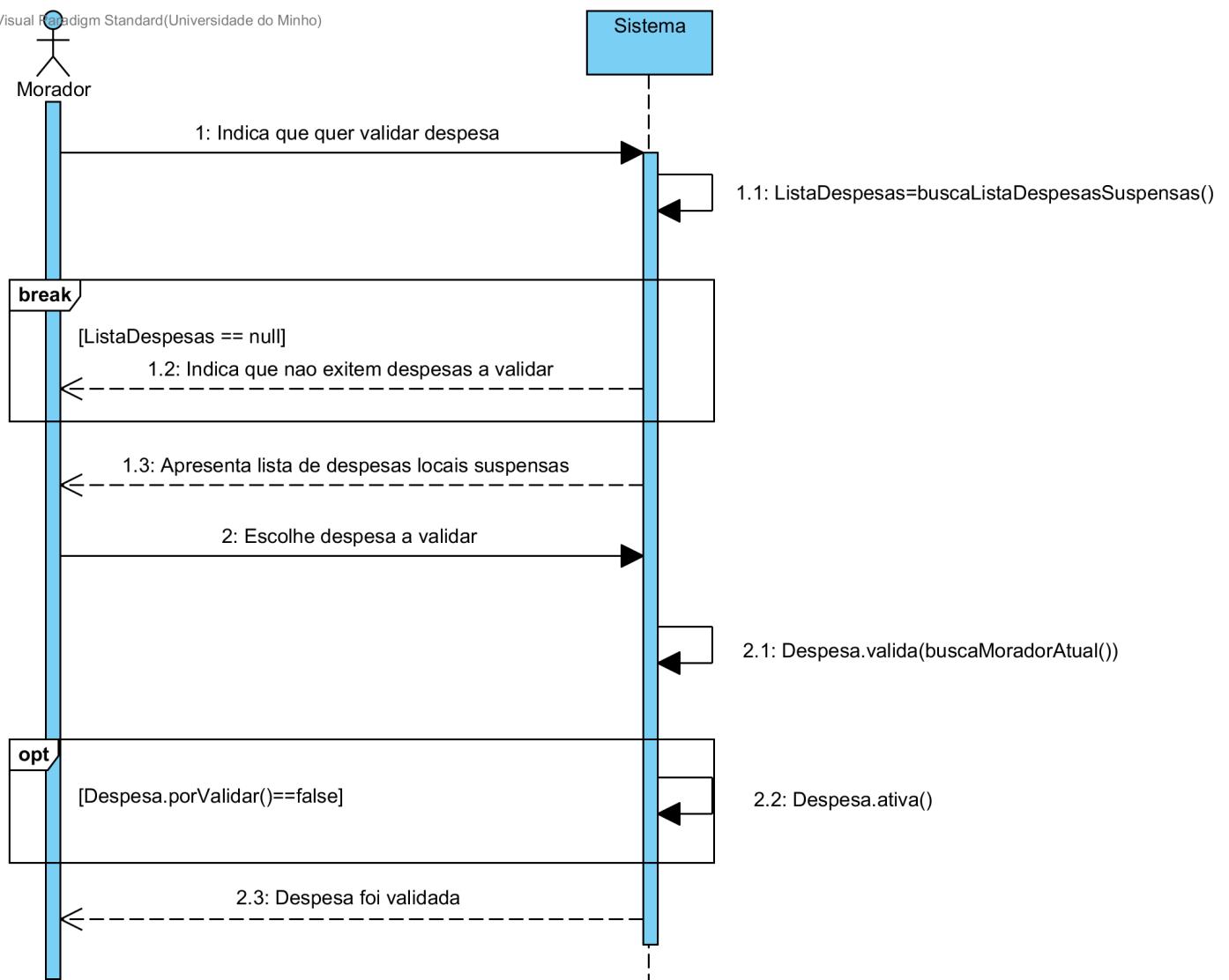


Figura 83: Diagrama de Sequência para Validar Despesa Local.

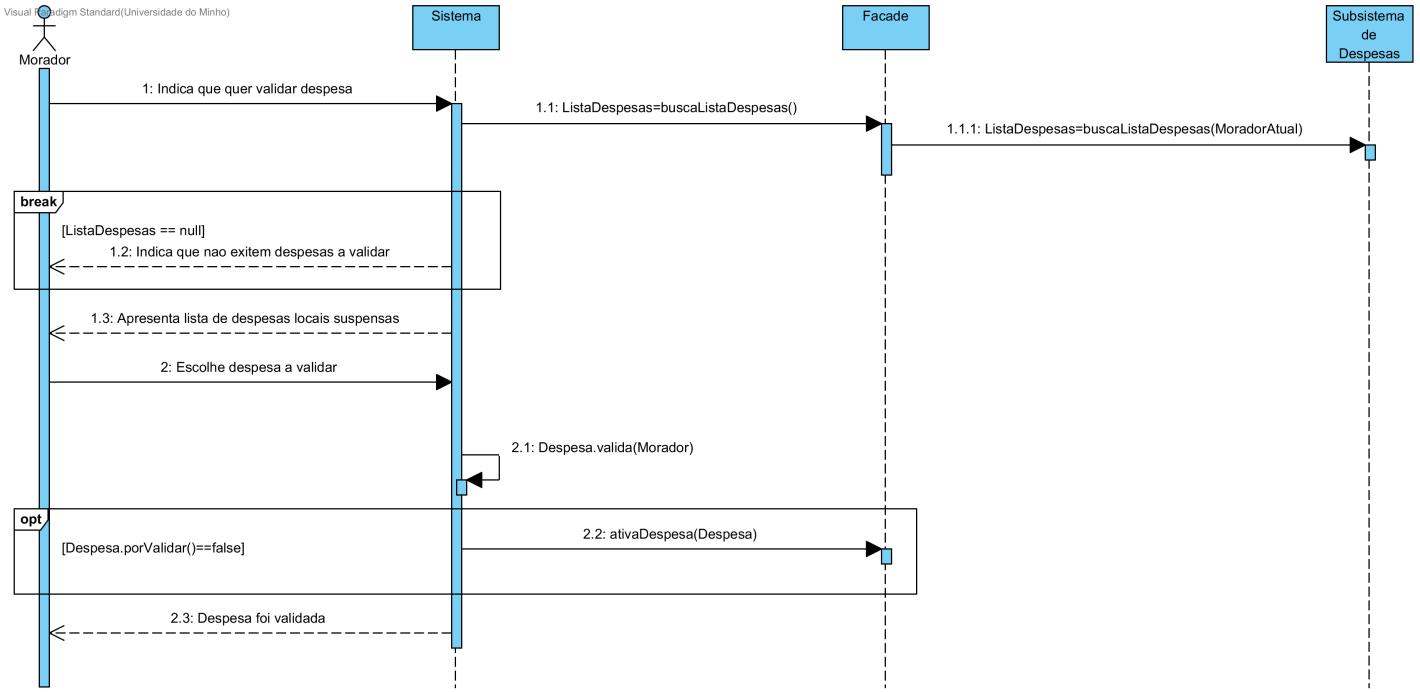


Figura 84: Diagrama de Sequência para Validar Despesa Local, com subsistemas.

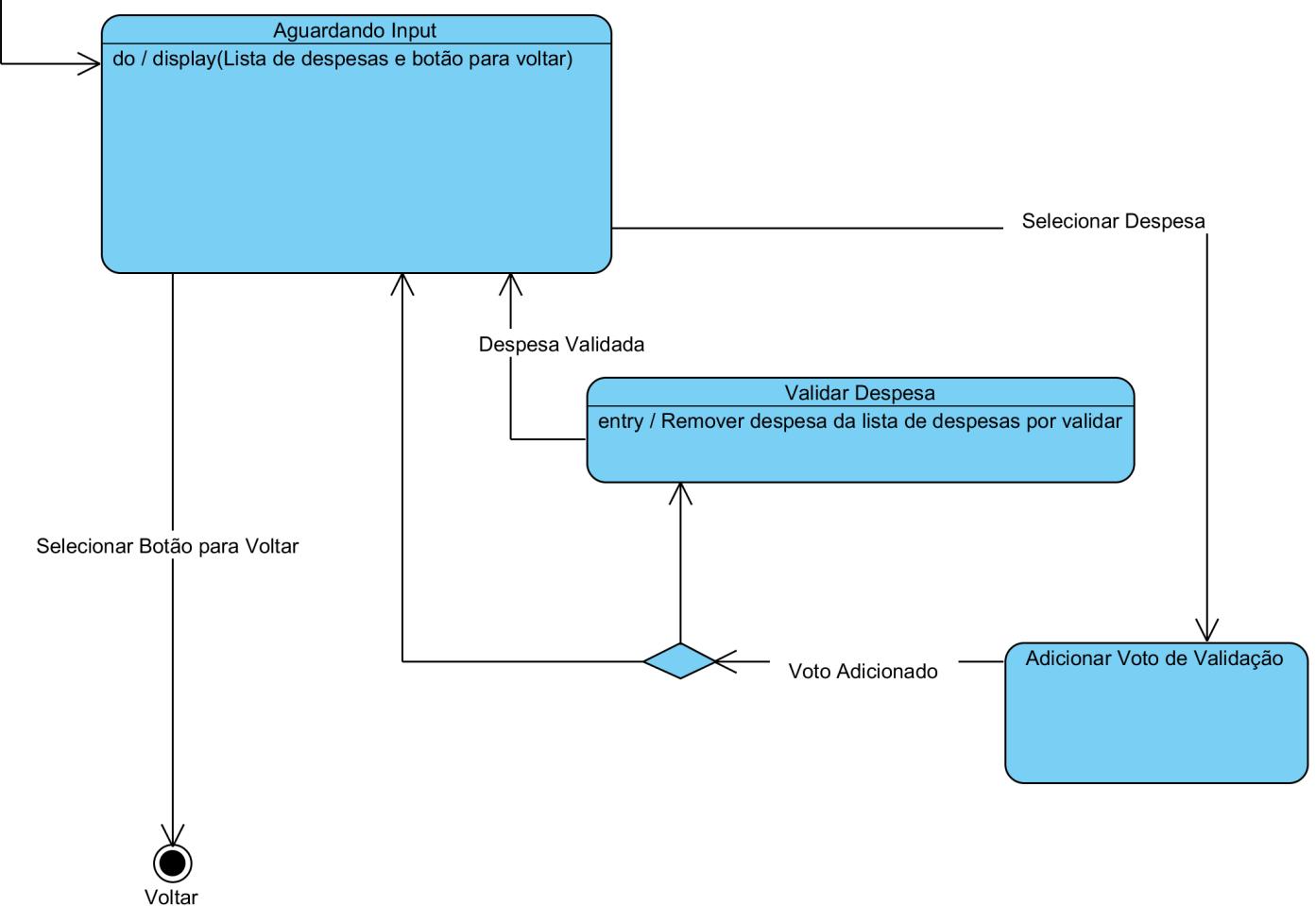


Figura 85: Submenu para Validar Despesa Local.