DESENVOLVIMENTO DE SISTEMAS DE SOFTWARE

LICENCIATURA EM ENGENHARIA INFORMÁTICA – 3º ANO

Submissão número 2 – Grupo 26




Dany Pinheiro da Silva – 51787
José Miguel da Costa Andrade – 51795
Artur Miguel Matos Mariano – 51813
Luís Carlos Serra Roque Oliveira – 52735

ÍNDICE

INTRODUÇÃO

Esta fase do trabalho propunha a realização do diagrama de classes e os diagramas de sequência.

A fase anterior propunha que a construção do modelo de domínio e diagrama de use-cases.

Sobre a anterior submissão, há a salientar o facto de que foram alterados pormenores no diagrama de use-cases (o modelo de domínio, permaneceu, para já, inalterado).

DIAGRAMA DE CLASSES

Considerações iniciais

Considere-se, para a leitura deste relatório, que apesar do UML ser uma linguagem de modelação polivalente, vulgo ser adaptável a várias linguagens de programação, como C, Java, etc…, foram tecidas considerações que tiveram um pressuposto de que o UML é uma linguagem de modelação especialmente concebida para modelar aplicações ou projectos desenvolvidos em Java. Isto acontece não por ser verdade (como acima referido é uma linguagem polivalente) mas sim porque se sabia de antemão que o projecto iria ser desenvolvido em Java e a própria unidade curricular assenta numa lógica de programação por objectos (no conteúdo relativo a programação), mais concretamente à linguagem Java.

Assim, o objectivo principal deste sub-capítulo é essencialmente registar que todos os elementos do grupo têm noção de que foram cometidos abusos de linguagem, por assim dizer, que não deixando de ser abuso de linguagem são, no entender dos elementos constituintes do grupo, uma espécie de mal necessário, para que o ambiente do relatório seja mais específico e para que não exista uma eventual generalização de conceitos que impediriam que fosse criado um ambiente tão próximo do trabalho, proporcionando assim um ambiente distante.

Uma pequena introdução ao problema

O problema da necessidade de presença de um diagrama de classes é um problema que, na verdade, não começa neste ponto: é necessário ter presentes todas as noções básicas da programação orientada a objectos para se perceber o quão importante é, de facto, o diagrama de classes.

Numa tipologia de linguagem que se rege por objectos a principal matriz identidade que possui é (ainda que possa parecer estranho fazer-se este tipo de abordagem tão lógica e óbvia) essa mesma: considerar todos os componentes da aplicação como objectos reais.

Para sustentar e exemplificar a noção base anterior pode-se recorrer a um pequeno exemplo: num software que se pretende implementar numa empresa de pequeno e comércio directo, existem um conjunto finito de objectos essenciais com que o sistema (tanto os utilizadores como os administradores) terá que possuir. É necessário, a título de exemplo, que a loja possua um preçário, independentemente do seu tamanho.

Com o avanço no tempo o preçário tornou-se obrigatório assumir-se como físico, ou seja, já não se está propriamente na era em que o dono da loja sabe os preços dos seus artigos sem recorrer a nada ou, na pior das hipóteses, a um papel que se encontre armazenado numa área de acesso restrito.

Assim, o preçário é um bom exemplo de um objecto, físico, que uma loja ou empresa de comércio directo tenha que possuir. No tipo de linguagem de programação que se aplica nesta unidade curricular, a linguagem orientada a objectos (POO), o preçário assume-se então como um objecto essencial no conjunto de objectos a implementar. Esse objecto terá um determinado conjunto de métodos associados que permitirão agir sobre esse objecto. Por exemplo, um preçário pode ser consultado (para listagem) por produto, pode ser consultado (para listagem) por categorias, pode retornar o preço de um dado produto de uma dada categoria…

Como o conjunto de classes, apesar de finito, acaba por ser complexo na medida que o nível e a qualidade da sua construção, determinam, em grande margem, a funcionalidade (ou ausência dela) da aplicação, considerou-se necessário fazer esta pequena abordagem para que se frisasse correctamente que o diagrama de classes tem uma importância extremamente acrescida numa aplicação.

Todas as considerações tecidas gozam de grande generalidade pois o diagrama de classes, não tem, ou pelo menos a definição que ganhou nas linhas anteriores não é a mais específica (embora correcta) pois existe um conceito (o diagrama de use-cases) que completa, acrescenta e clarifica toda a lógica que se apresentou. Esse conceito aparecerá mais à frente.

Ainda que o diagrama de use-cases seja considerado prioritário na construção da aplicação em relação ao diagrama de classes, por questão de conveniência de apresentação de dados e explicação dos conceitos dados da matriz que identifica esta tipologia de linguagem, apresentou-se o diagrama de classes primeiro. Com isto, quer-se clarificar que a ordem de apresentação não determinou a lógica de construção. Além disso, considere-se também que os diagramas são complementares e considerou-se que não seria a melhor solução torná-los independentes e construí-los separadamente.

O diagrama de classes como um manancial da construção da aplicação

O diagrama de classes, como anteriormente referido, determina um plano de classes a ser implemento na aplicação (GereComSaber, no caso) construída na linguagem de programação Java.

O diagrama de classes é constituído por um conjunto de dez classes que configuram a navegabilidade e lógica de encadeamento da aplicação. A navegabilidade da aplicação é, no seu cômputo geral, definida por um conjunto de métodos que definem, em traços gerais, os percursos que é preciso efectuar para atingir uma determinada instância de um determinado objecto, para, por sua vez, se poder efectuar ou um método da respectiva API ou atingir um atributo (que na classe é uma variável de instância).

O diagrama pode ser visto como uma espécie de um conjunto de referências (que contêm "semi"-referências, os atributos) que permitem fazer a ligação entre as várias classes. A tradução desta definição é, na prática, o percurso que faz, como foi anteriormente dito, para atingir uma instância ou atributo de instância.

Além disso, os objectos necessários para a implementação da aplicação têm um conjunto de atributos definido, que como foi dito podem ser "atingidos" através de certas classes. A palavra "atingido/(a)" implícita um conceito que o digrama de classes impõe e auxilia a construir: as estruturas de dados da aplicação.

O diagrama de classes indica as classes presentes na construção da aplicação, mas mais que isso, indica a sua disposição e armazenamento. Aplicando este conceito

ao exemplo anterior, ao exemplo do preçário, pode-se deduzir que esse mesmo preçário poderá ser um conjunto de tuplos <Preço de um Artigo, Artigo>, em que a disposição tem razão de ser mas será detalhadamente explicada à frente.

Este "conjunto" de tuplos (pares) poderá antes ser visto como uma "tabela" pois em POO até as próprias estruturas de dados podem ser vistas como objectos. Aliás, essa é por muitos uma teoria defendida, embora este assunto não seja alvo de estudo neste relatório. Essa "tabela" em POO corresponde a um Hash que é uma estrutura de dados que deriva da estrutura base "Tabela de Hash", mas que pode ser dividida em HashMap ou HashSet.

Esta ideia de que o diagrama de classes implícita os tipos de estruturas de dados que são necessários é importantíssima. Aliás, é precisamente essa noção que está presente quando se evocam termos como "navegabilidade". Este tipo de lógica "programatical" é uma das várias características de que a cadeira de Desenvolvimento de Sistemas de Software goza: ela indica-nos as classes presentes, a forma como estarão organizadas e a navegabilidade e rotas que são precisas ser assumidas para se atingir um determinado objecto ou atributo.

Para exemplificar claramente a noção de atingibilidade d'um objecto ou d'um atributo, pode-se evocar um exemplo concreto, já que deve ser importante demonstrar isso, dada a quantidade de vezes que este assunto foi referido. Imagine-se uma tabela (como estrutura de dados). Essa tabela relaciona uma dada Caneta (objecto) com um dado preço (integer). Imagine-se agora outra tabela distinta, que relaciona uma dada marca de canetas a uma "instância" da tabela anterior. Esta noção tem necessariamente um esquema associado. Mas, a nível de classes, a noção presente é a de que há uma classe MARCA que tem uma estrutura de dados que consegue relacionar um inteiro com um outro objecto CANETA.

No fim deste exemplo o que é de facto pretendido, é, hipoteticamente, saber qual a espessura de uma dada caneta de uma dada marca. Então é necessário "navegar" no sistema. Partindo da classe MARCA, terá que se entrar na estrutura de dados lá presente, (hipoteticamente com o nome "marcas") e terá que se encontrar a Caneta pretendida. Este encontro do pretendido terá que se dar com um método "Equals"

definido pelo programador, que se considerou não ser necessário ser alvo de estudo neste relatório já que já o foi no ano transacto na unidade curricular de POO.

Ao atingir a CANETA pretendida, pode-se então agora evocar o método que devolve a espessura real da caneta. Assim, dada a caneta X pode-se obter a espessura aplicando o método hipotético getEspessura().

Foi considerado que seria importante mostrar o quão o diagrama de classes facilita a visão sobre o programa final, pois é significativamente diferente ir programando nas classes à medida que se criam ou ter uma modelação concreta que transmita esta quantidade impressionante de informação.

O diagrama de classes

Este diagrama de classes representa as classes a ser implementadas na linguagem Java, quando se passar à fase de implementação da ferramenta. Assim sendo, existe uma classe que representa a própria entidade "Gere com saber" que possui uma base de dados (faz-se aqui uma referência às considerações sobre suposições de aplicação do UML "exclusivamente" à linguagem Java e à analogia entre classes e objectos na página 4).

A empresa, possui, assim, uma base de dados (que pode ser imaginada como um objecto real através, por exemplo, de um computador que sirva de base de alojamento para o software "GereComSaber"), base de dados essa que contém os dados dos fornecedores e dos clientes que possui. A base de dados, aloja, também, o histórico relativo às transacções e a informação relativa aos utilizadores do sistema (sistema de login).

A tabela de utilizadores tem outra classe associada "utilizadores" (a classe da tabela tem uma estrutura de dados com vários utilizadores). A classe de utilizadores possui os atributos relativos ao login. A lógica da existência desta classe reside no facto de o software prever e salvaguardar a situação de guardar os dados relativos à utilização

da aplicação num dado instante pelo funcionário respectivo, precavendo falta de informação relativa ao funcionário que esteve presente numa dada venda, recepção de pedido de serviço, etc… No fundo pode ser assumido como um controlador de presenças por instante.

À tabela de clientes está associada uma estrutura de dados que contém vários clientes. Apesar de merecer capítulo próprio pela importância que assume, urge falar da multiplicidade neste ponto. Isto acontece porque pode-se assumir como dogma o facto de, dadas duas quaisquer classes seguidas, a primeira possui uma estrutura de dados da segunda. No entanto essa estrutura de dados pode ser múltipla ou não. Um exemplo prático disso é a classe dos clientes que apesar de estar inserida na tabela de clientes (vários clientes) possui um e um só contracto associado.

A multiplicidade é uma noção muito importante neste diagrama pois indica uma quantidade de informação impressionante na altura de construção das classes. A multiplicidade indica o tamanho das estruturas de dados associadas ás várias classes.

Essa multiplicidade é visível no diagrama e é bastante intuitiva. Uma relação de 0..* indica uma classe com disponibilidade para possuir como atributo uma estrutura de dados com vários objectos de outra classe, a ela associada. A relação unitária estabelece apenas um atributo da classe de destino na classe de origem.

A interpretação do diagrama é uma "repetição" do que foi dito anteriormente, em todos os subconjuntos de classes presentes a par. Existem, no entanto, as excepções da interface "Comparator", em que todas as classes que têm ligação a essa interface a implementam. Na prática isso indica que podem ser implementados os métodos que a interface tem definidos. Foi utilizada esta interface porque diminuiu substancialmente o número de métodos iguais que estavam implementados (equals) para três classes distintas (funcionários, clientes e fornecedores).

A última referência ao diagrama de classes tem a ver com a agregação que pode ser partilhada ou composta que correspondem ao "diamante" vazio e cheio, respectivamente. Essa agregação indica, no caso de ser partilhada que eliminar uma classe não significa eliminar a sua "precedente" no diagrama. A agregação composta

indica o contrário, indica que eliminando a classe de origem no diagrama a sua classe de destino (ou seja “precedente”) é eliminada.

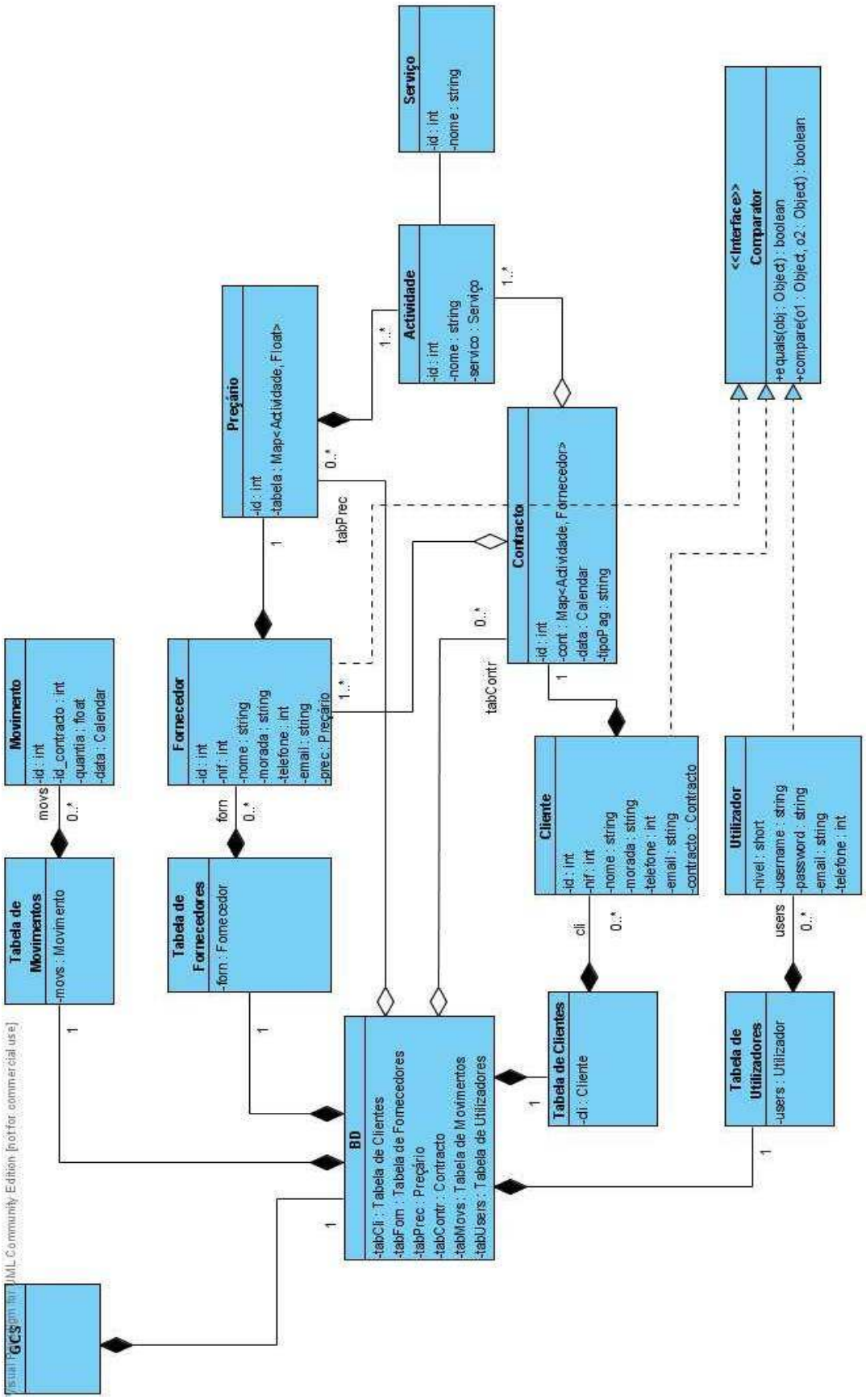


DIAGRAMA DE USE-CASES

O diagrama de use-cases: pequena abordagem introdutória

O diagrama de use-cases representa, acima de ainda mais informação, uma espécie de guião ou de um bloco de notas que contém funcionalidades gerais mas especialmente específicas que a aplicação contém ou, geralmente, virá a conter. Esse guião representa também a modelação que vai ser tomada, conseguindo transmitir um início, meio e fim.

Este diagrama é muitas vezes tomado como um panorama geral de uma aplicação futura (em construção inicial, por meio de modelação e projecção). Uma analogia facilmente compreensível pode ser tomada como a situação da construção de um edifício, que tal como esse mesmo edifício tem um "projecto" ou planta de construção, também hoje em dia uma aplicação informática e software geral possui ou possuem essa mesma planta.

Uma "planta" de um software passa por um conjunto racional de funcionalidades de que esse software terá de gozar. A construção através de código, em programação, dá-se (em grande parte) com base no diagrama de use-cases.

O diagrama de use-cases é uma perspectiva da aplicação de um nível muito alto que pode ser inclusive ser considerada a perspectiva do nível mais alto que se pode obter do programa ou aplicação. Essa aplicação, contudo, não oferece (e daí ser de um nível muito alto) uma visão sobre a implementação daquilo que pré dispõe a ser visto como o conjunto de funcionalidades.

Este diagrama, é, por todas as razões apresentadas, alterado muitas vezes ou, de modo mais correcto, em vários momentos da sua construção. Tem de existir uma refinação do diagrama já que muitas das funcionalidades que a aplicação tem de ter não são obtidas aquando do inicio da sua construção, já que o cliente da aplicação pode facilmente mudar de opinião ou não ser sempre claro, ou a implementação de uma certa funcionalidade pode eventualmente implicar a existência de um outra … e distinta funcionalidade e até porque existe uma tendência clara para o programador (ou outro dos construtores da aplicação) não associar imediatamente todas as funcionalidades à

realidade, que só é um dado adquirido à medida que se vai apercebendo das capacidades que a aplicação realmente possui ou vai fazendo pequenas analogias com a realidade para se aperceber de cenários que pura e simplesmente não fazem sentido num determinado contexto.

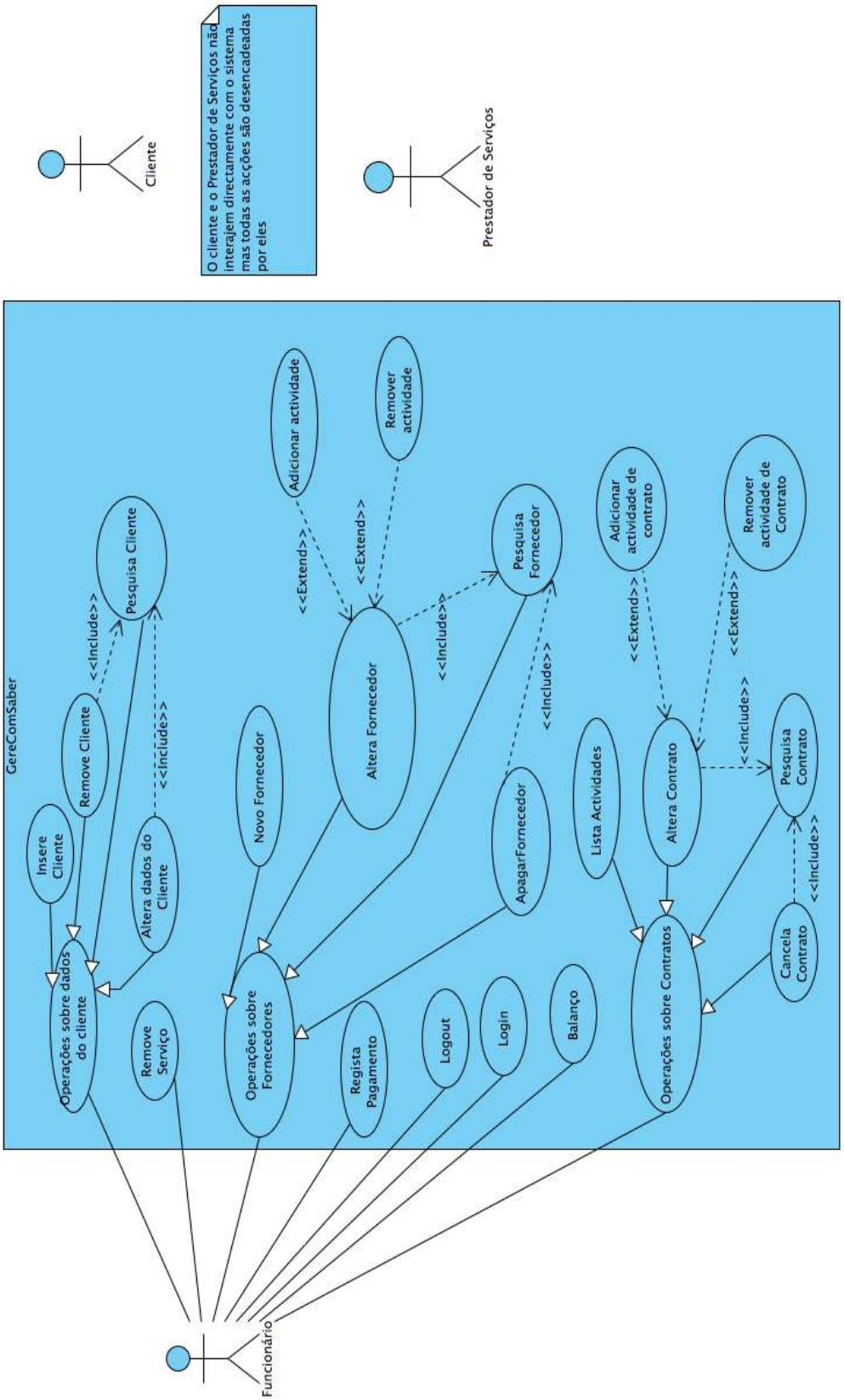
Por todas as razões presentes no último parágrafo, salvaguarda-se aqui a hipótese de alteração de elementos constituintes da aplicação (quer seja o diagrama de use-cases ou outro qualquer), porém este assunto será alvo de estudo em local mais apropriado, em capítulo seguinte.

O *Flow of Events* de cada use-case é igualmente importante ao diagrama de use-cases. Considere-se, até, como uma parte integrante da "inteligibilidade" do diagrama de use-cases. Nem tudo pode ser quantificado em importância, porque tal como um automóvel tem que ter um motor ou um quadro de chassis para andar ou como um homem tem de ter coração ou cérebro para viver, uma aplicação que goze de modelação com base na sua construção não possui apenas um diagrama de use-cases: possui também o "respectivo" *flow of events*.

Esta "propriedade", por assim dizer, descreve a interacção entre os utilizadores e o sistema. Faz, em modo grosseiro, como que uma ponte ou passagem para o nível seguinte da modelação da aplicação, onde é necessário efectuar os diagramas de sequência. Esses, por questões de coerência, seguem padrões definidos no *flow of events*. Nesse ponto da modelação o elemento da construção da aplicação ou software pode-se aperceber facilmente que o *flow of events* necessita de ser alterado ou "refinado".

Todas as considerações anteriores foram uma espécie de introdução de conceitos mas salvaguarde-se a posição de que têm, como é óbvio, uma relação directa com os problemas e actos do grupo que construiu essa aplicação e espelha não só um manual para um utilizador comum deste relatório perceber o que são os conceitos mas também para demonstrar, a pessoas instruídas nas várias questões tudo o que foi feito na aplicação concreta que foi desenvolvida e está a ser alvo de estudo neste relatório.

O novo diagrama de use-cases



Alterações no use-case em relação à primeira submissão

Depois dos comentários feitos ao trabalho concluído na primeira submissão, repensou-se o trabalho por forma a verificar, se de facto faria sentido pensar de novo o diagrama de use-cases. De facto, decidiu-se alterar o diagrama em questão pois existiam alguns use-cases não muito "racionais". Para título de exemplo surge nomeadamente o diagrama do fornecedor de serviços, que estava a efectuar um *include* de um novo serviço no sistema que por sua vez fazia também um novo *include* de uma nova actividade.

Foi retirado e refinado, esse erro. Actualmente o diagrama prevê uma alteração no fornecedor. No actual diagrama existe então mais coerência e eficiência no que toca a esse ponto.

O use-case de alteração de contracto foi também alterado, na medida em que estava demasiadamente genérico e foi alterado no sentido de ser mais específico, tendo sido "extendido" em dois use-cases: listar actividades e o novo use-case de alteração.

Também devido aos comentários repensou-se a nomenclatura do principal actor do sistema, o administrador. Este passou a ser denominado de "funcionário" pois o principal utilizador do software, na realidade, não é o administrador do sistema mas um dado funcionário que a empresa nomeie para que, num dado instante, esteja a fazer o controle necessário dos pedidos dos clientes e a registá-los no sistema para que possam ser satisfeitos.

DIAGRAMAS DE SEQUÊNCIA

O diagrama de sequência: pequena abordagem introdutória

A criação de modelos de comportamento dos sistemas é efectuada recorrendo aos diagramas de sequência. Os diagramas de interacção modelam o comportamento dos componentes dos sistemas, e em particular os diagramas de sequência representam interacções entre objectos, mostrando as mensagens trocadas mutuamente enquadrando tudo numa sequência temporal.

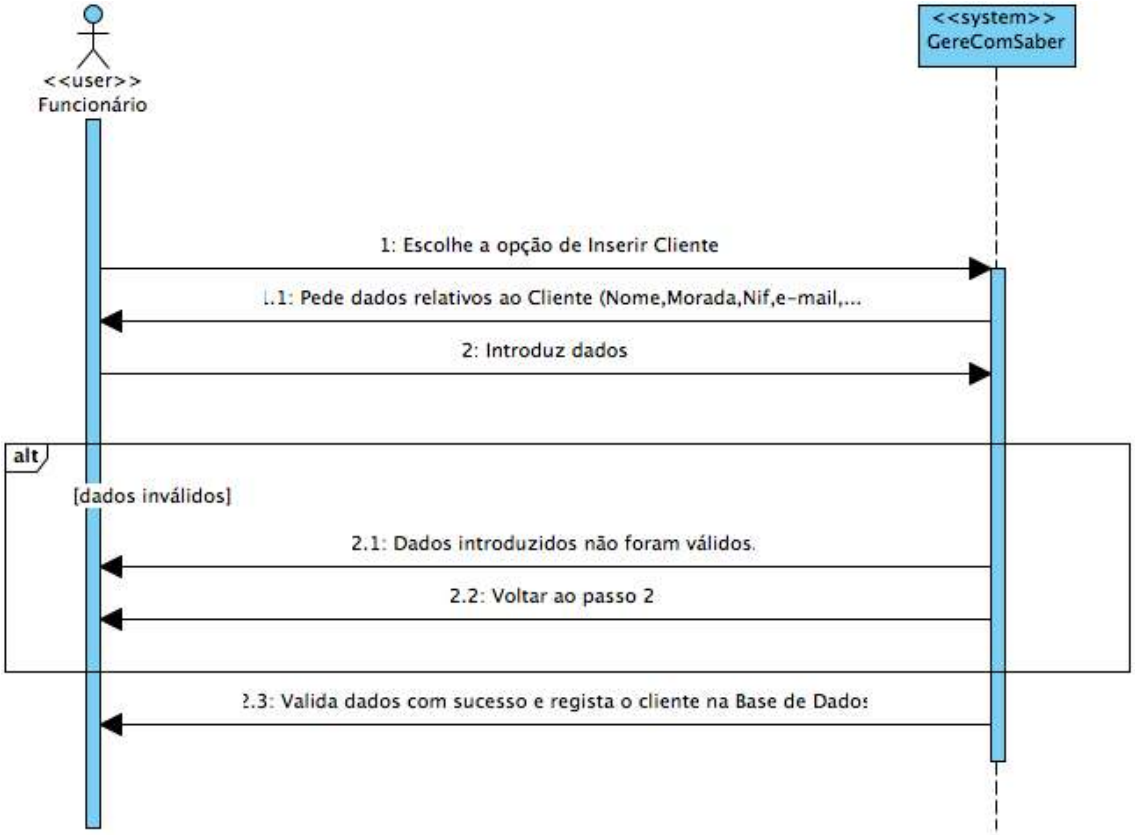
Este tipo de diagramas apresenta o sistema como uma caixa negra que não indica o sub-sistema, representando os eventos gerados pelos actores e as respostas a nível de comportamento (operações do sistema).

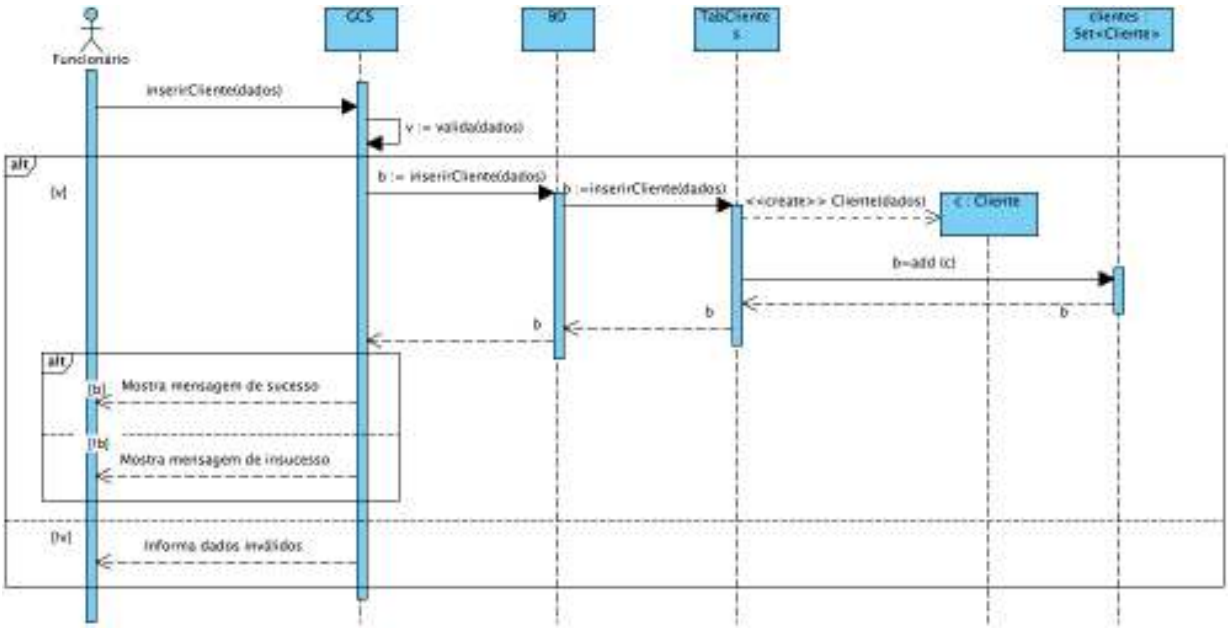
Estes diagramas representam o mesmo que os diagramas de comunicação, mas a ênfase é colocada na ordenação temporal das mensagens. Pode eventualmente dar-se a análise do sítio onde se faz o processamento pelas diferentes classes.

Como não se justificaria por motivos obviáveis, os elementos do grupo consideraram que não seria estritamente necessário fazer a apresentação detalhada de todos os diagramas, ou, por outras palavras, colocar em texto aquilo que é perceptível nos próprios diagramas. Seguem, de seguida, os diagramas (são apresentados dois tipos: os simples e os de implementação). Os simples são a forma gráfica representativa dos *flow-events* use-cases. Os diagramas de implementação relacionam-se com o diagrama de classes e configuram a implementação real da futura implementação.

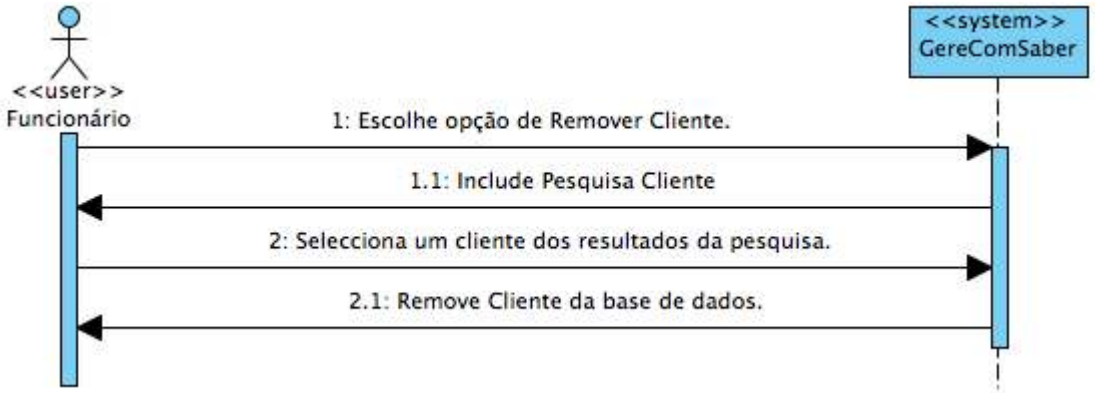
São apresentados, por ordem de conveniência, primeiramente os diagramas simples e depois os de implementação a pares.

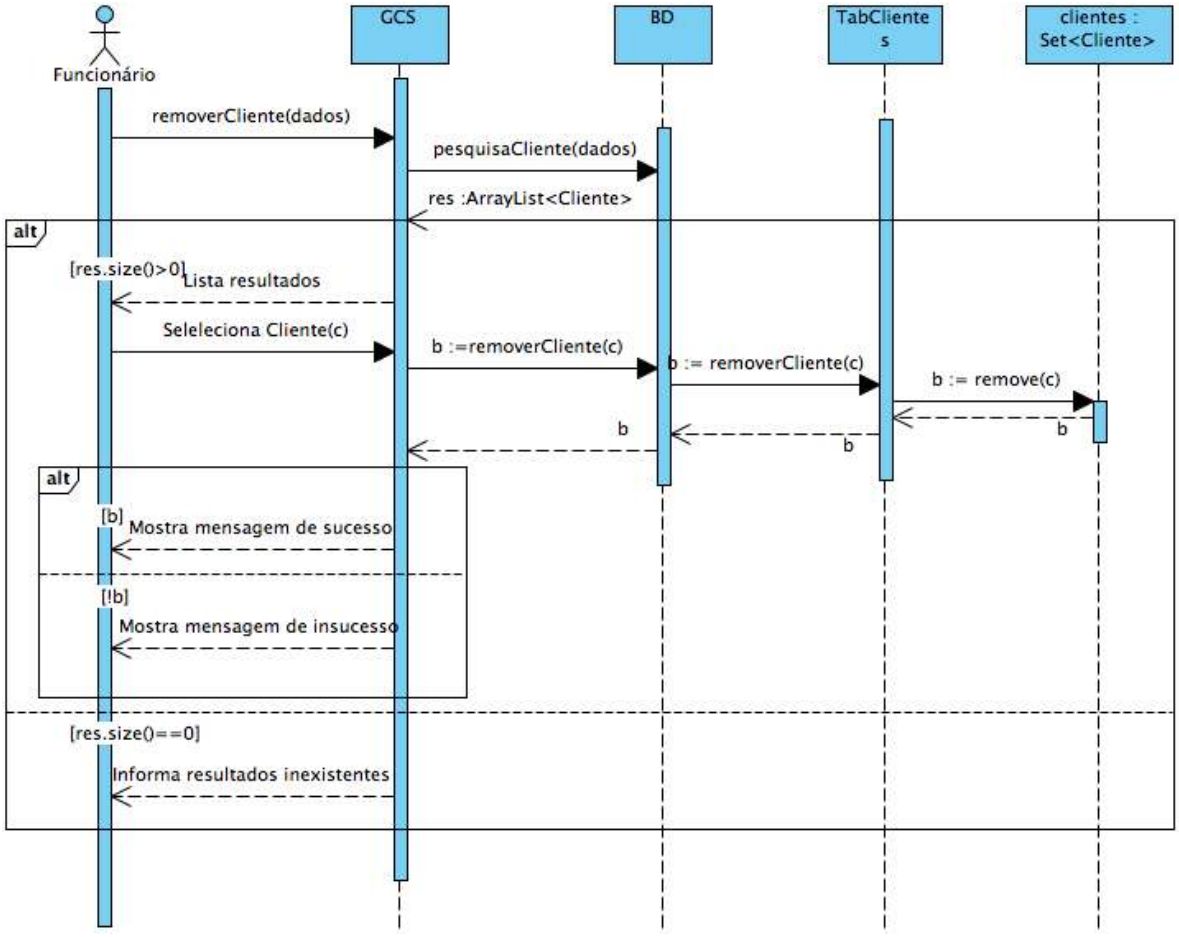
Inserção de Cliente



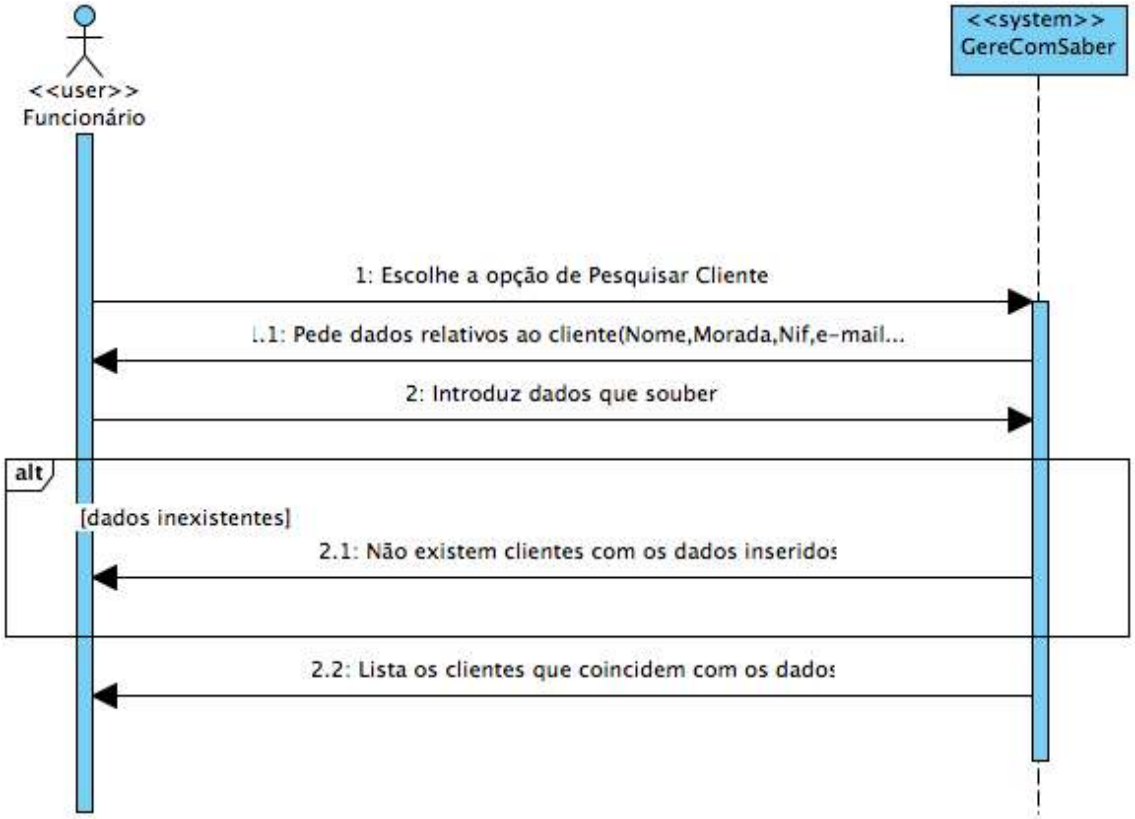


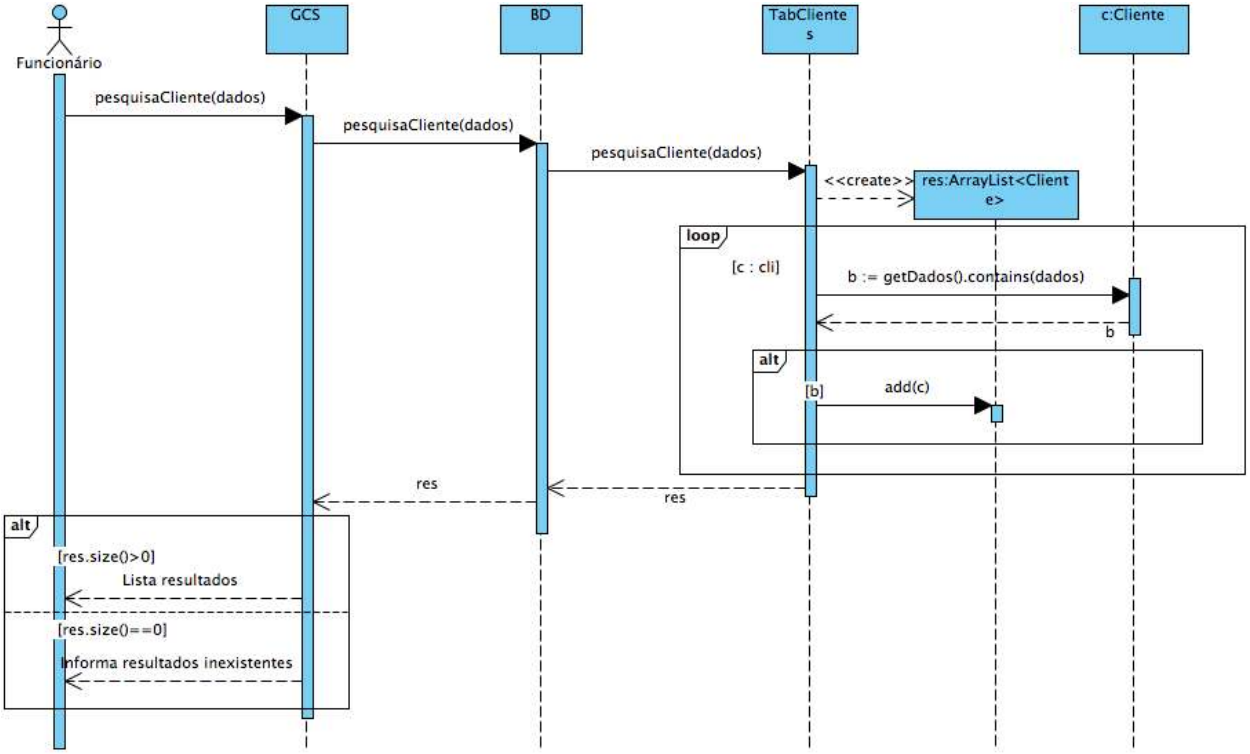
Remoção de Cliente



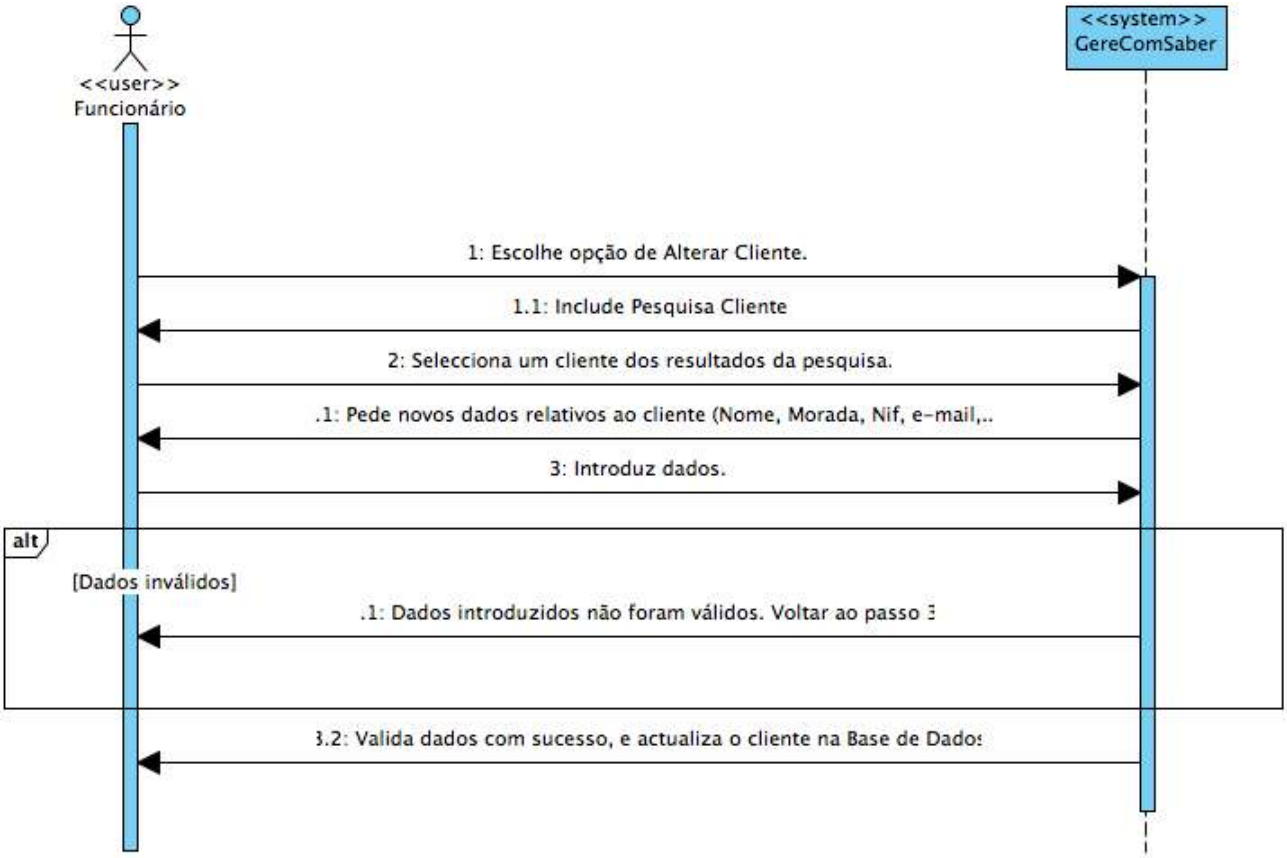


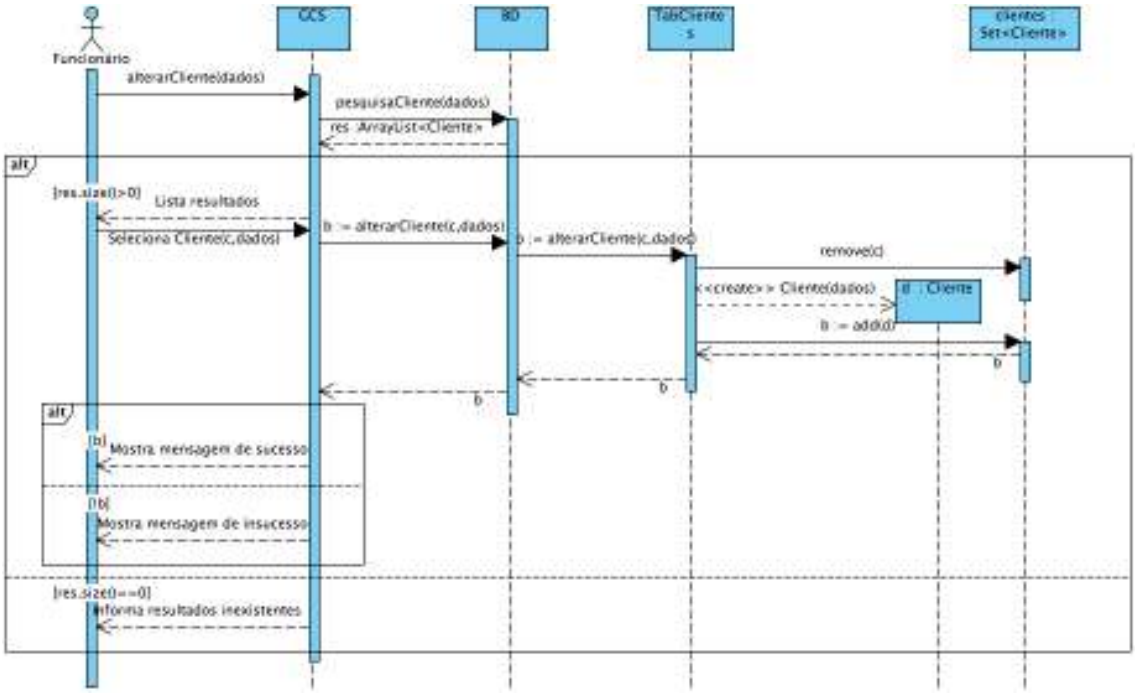
Pesquisa de Cliente



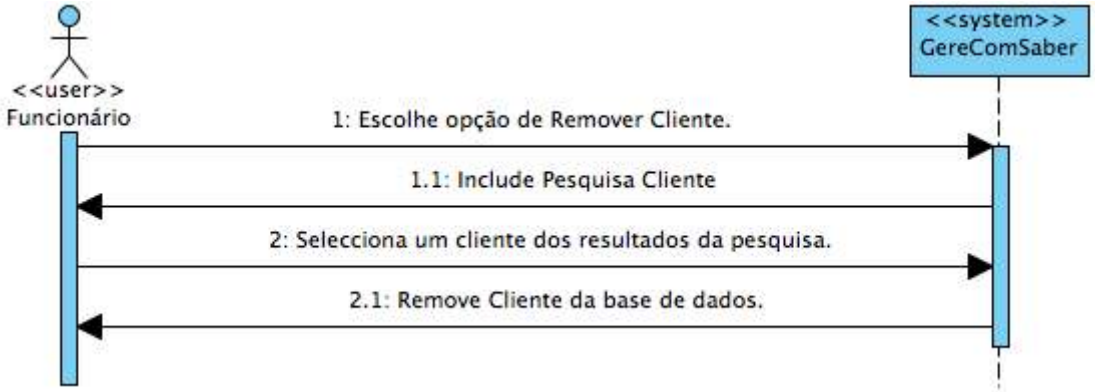


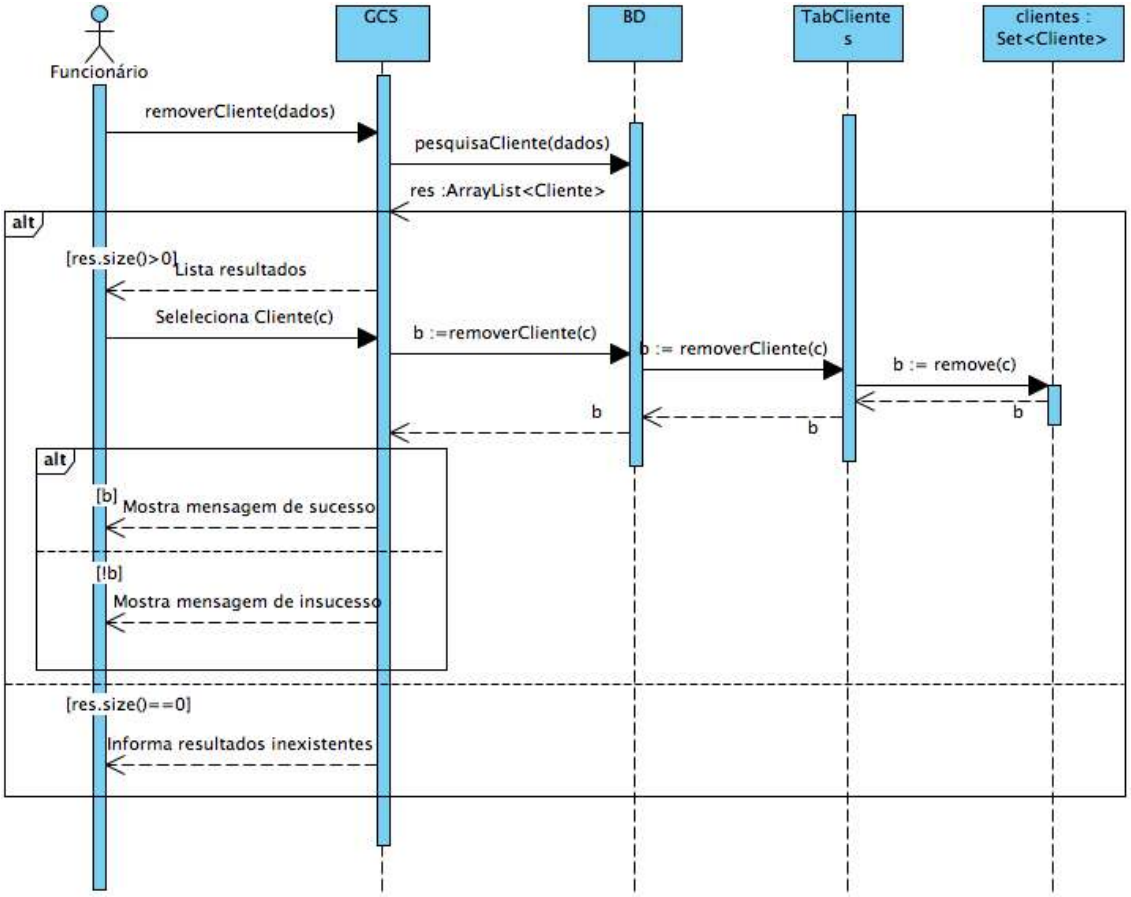
Alteração de Cliente



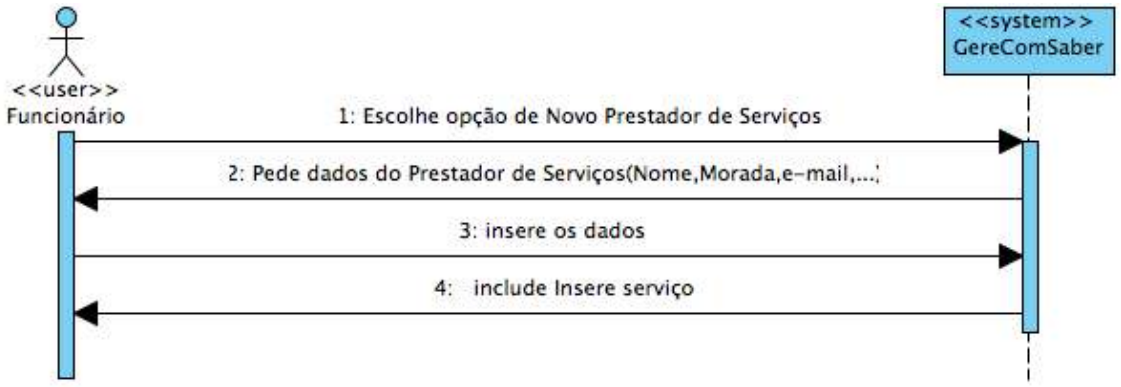


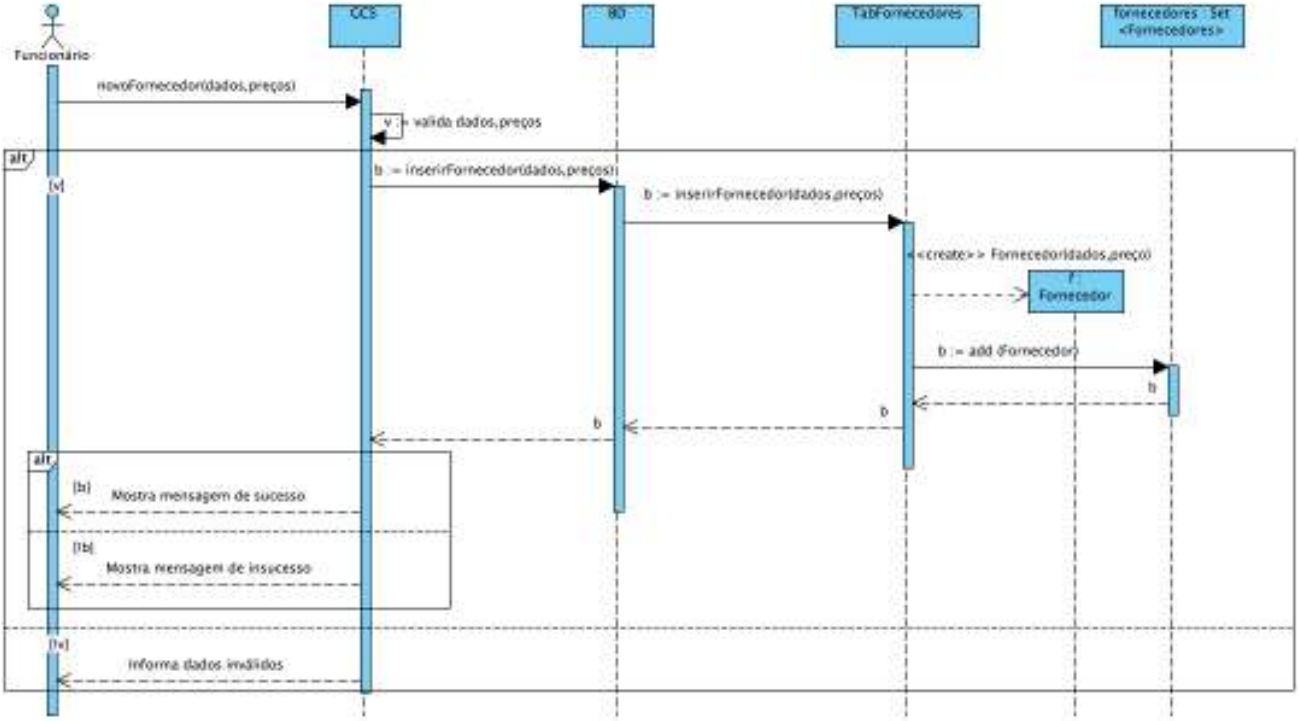
Remoção de serviço



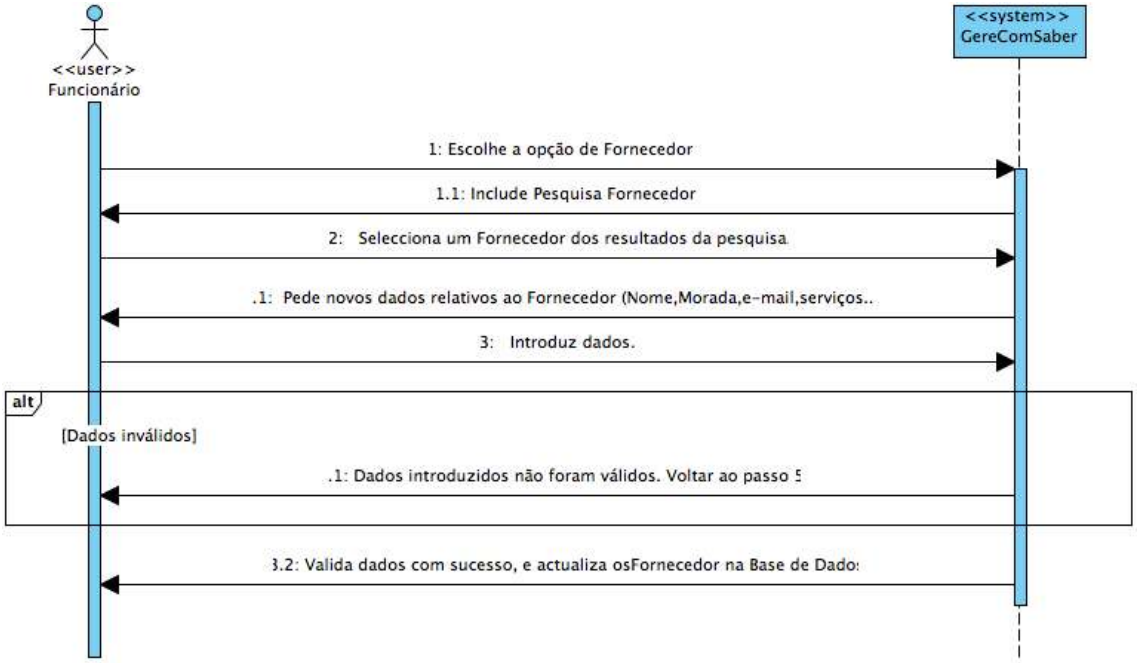


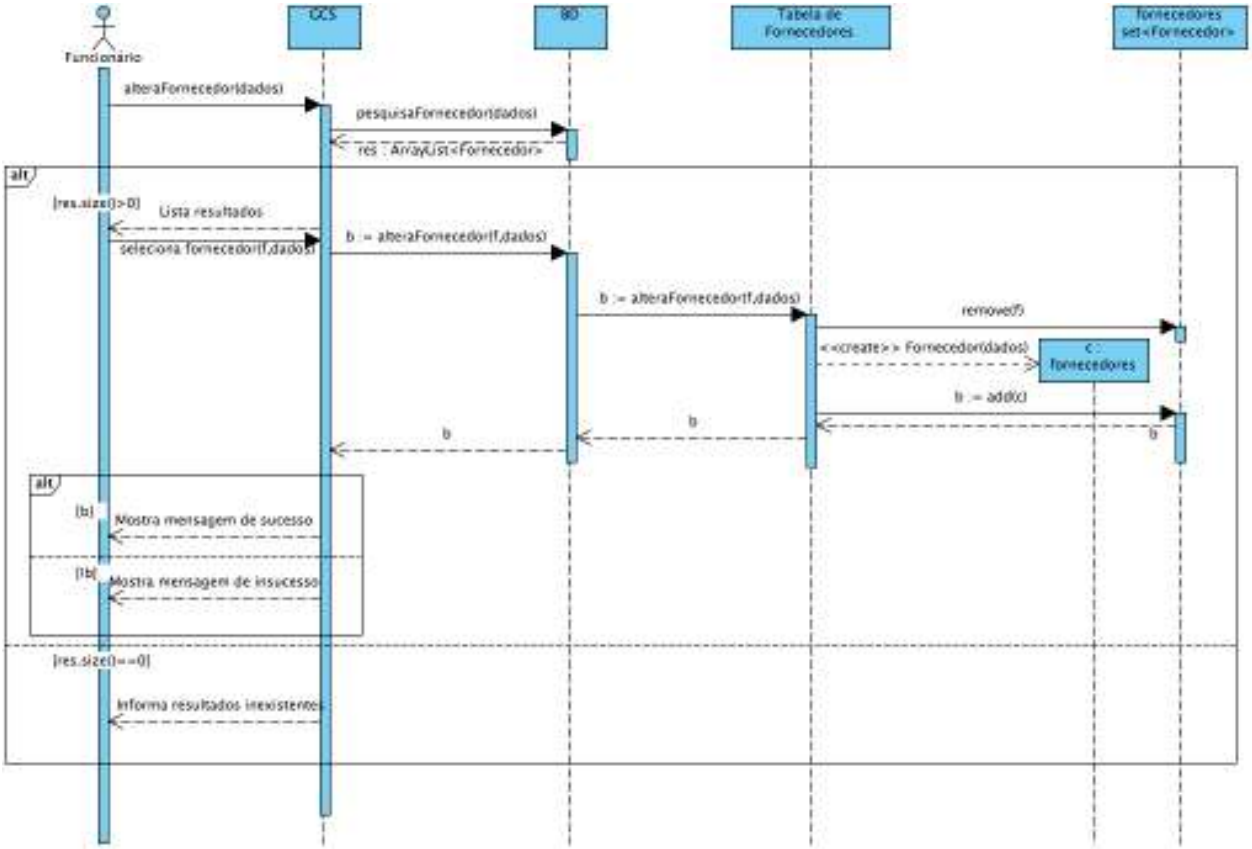
Novo Fornecedor



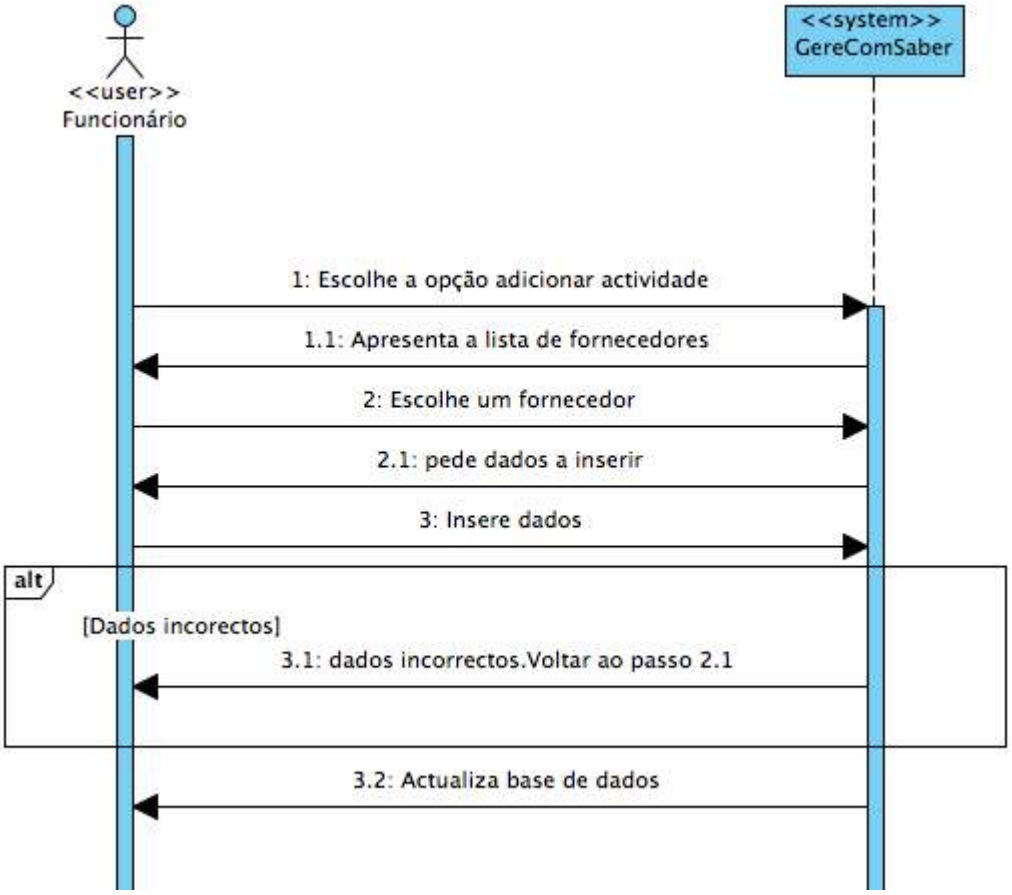

Funcionário
GCS
BD
TabFornecedores
fornecedores : Set «Fornecedores»
novoFornecedor(dados,preços)
v := valida dados,preços
alt
[v]
b := inserirFornecedor(dados,preços)
b := inserirFornecedor(dados,preços)
«create» Fornecedor(dados,preço)
f : Fornecedor
b := add (Fornecedor)
b
b
b
alt
[b]
Mostra mensagem de sucesso
[!b]
Mostra mensagem de insucesso
[!v]
informa dados inválidos

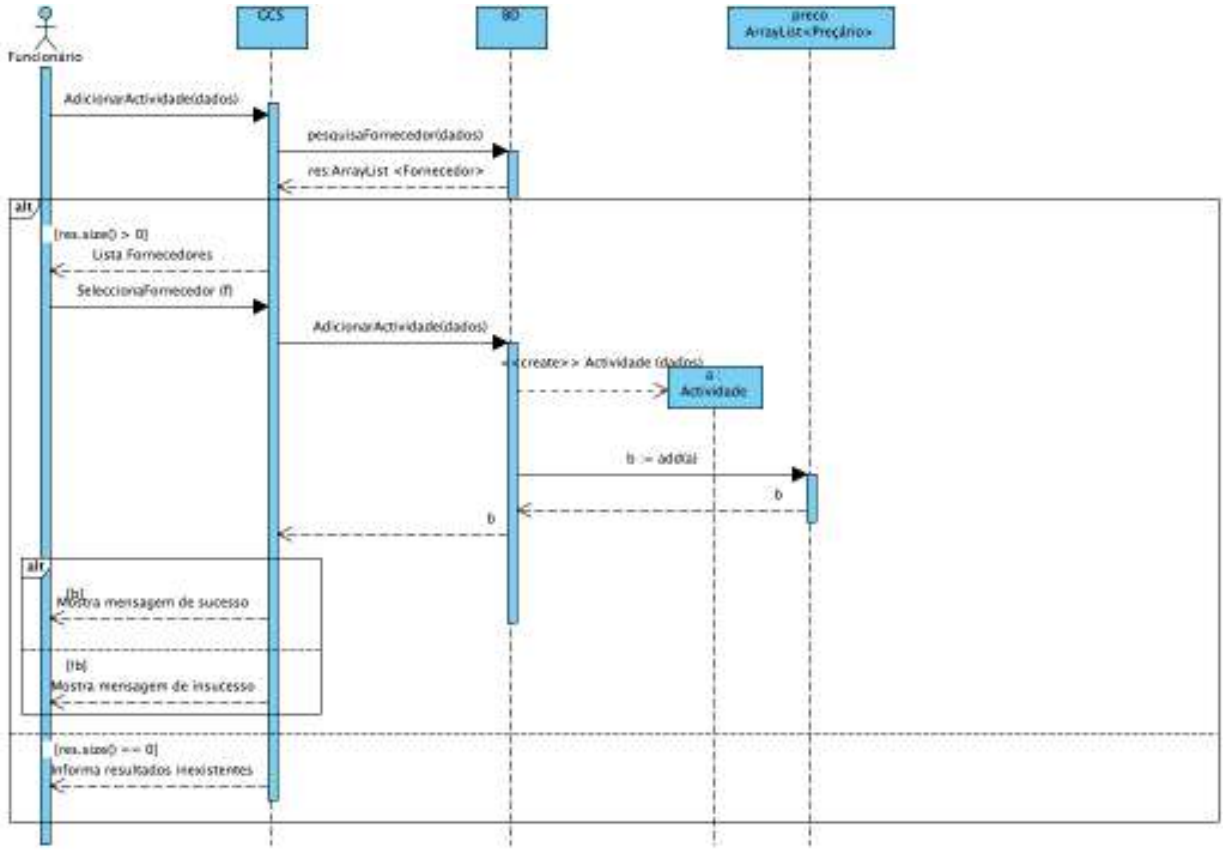
Alteração de Fornecedor



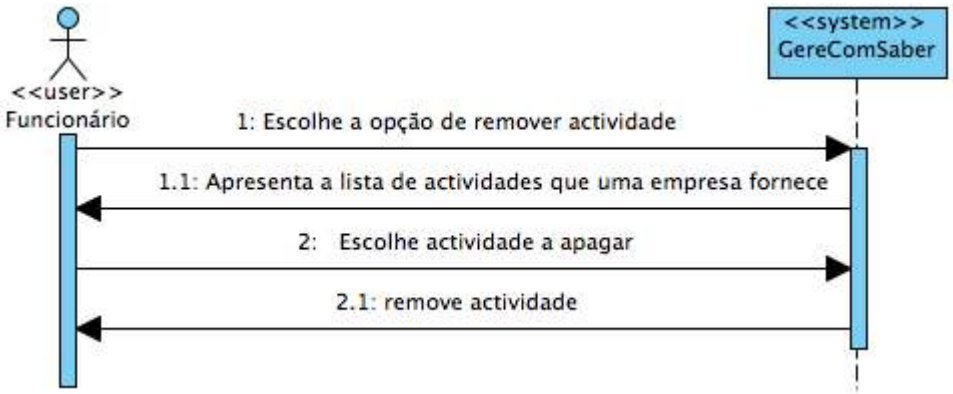


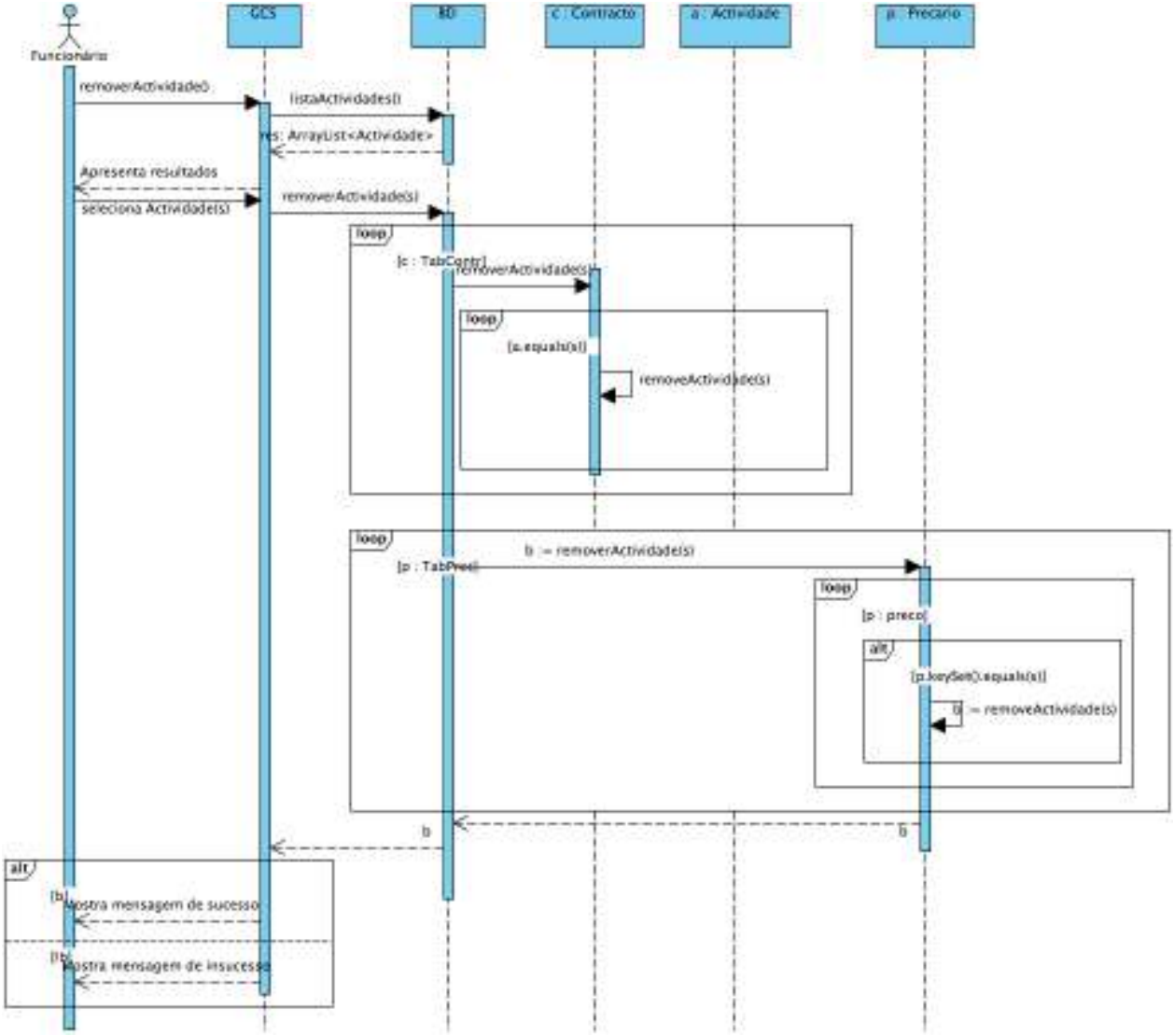
Adicionar actividade



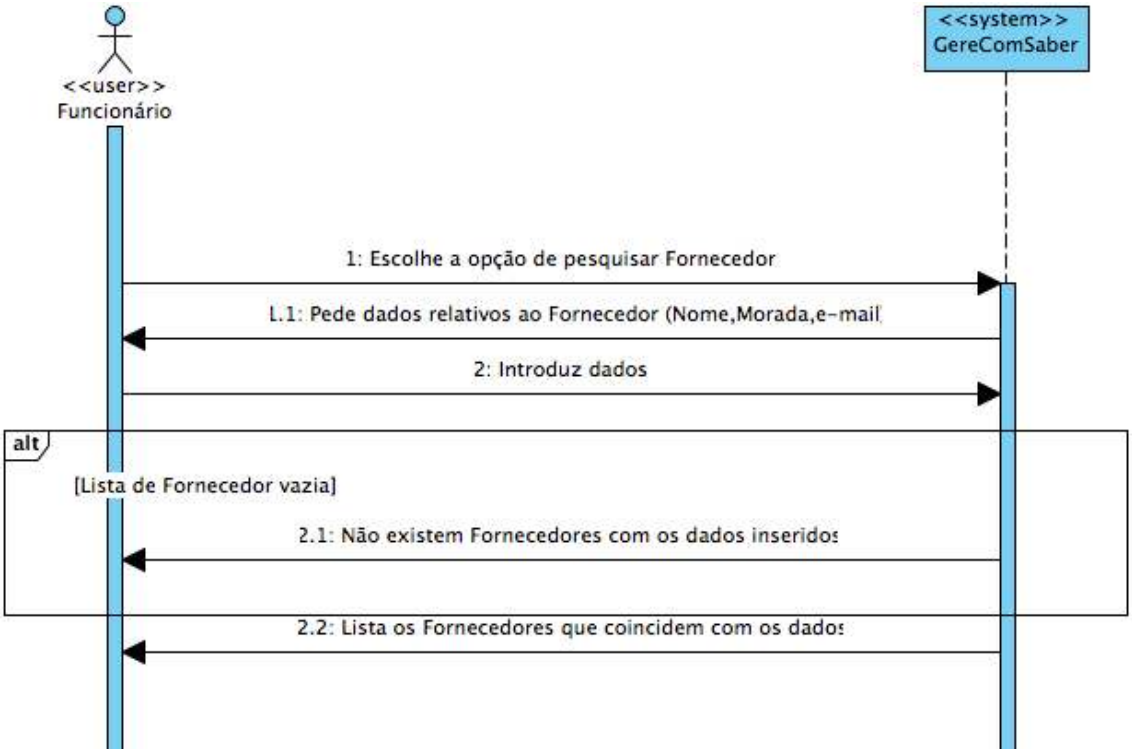


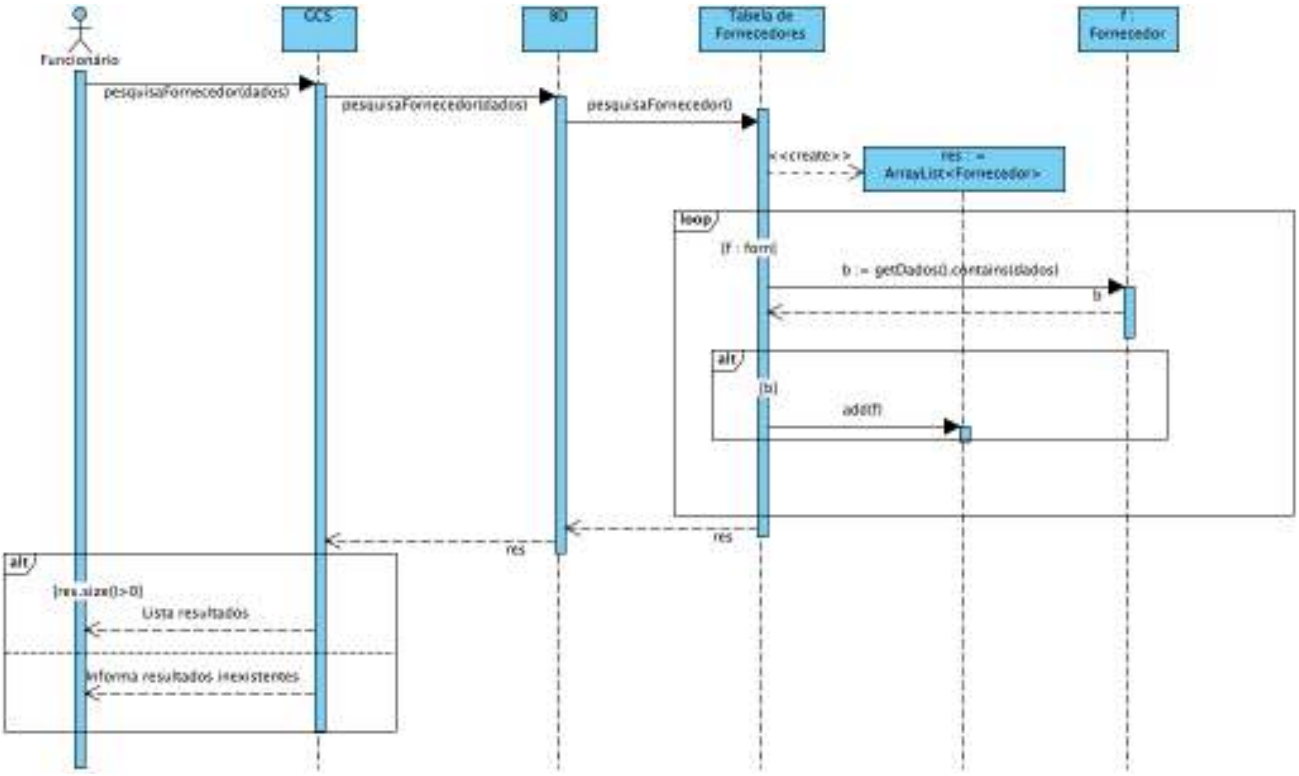
Remoção de actividade



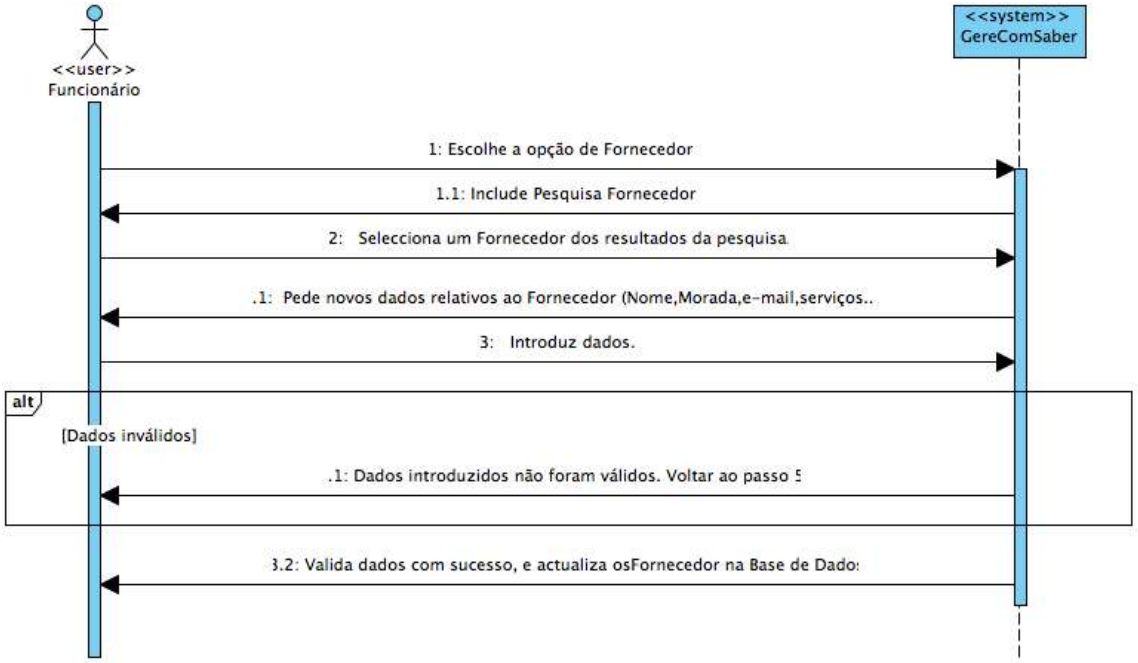

Funcionário
GCS
BD
c : Contracto
a : Actividade
p : Precario
removerActividade()
listaActividades()
res: ArrayList<Actividade>
Apresenta resultados
removerActividade(s)
selecciona Actividade(s)
loop
[c : TabContr]
removerActividade(s)
loop
[a.equals(s)]
removeActividade(s)
loop
b := removerActividade(s)
[p : TabPrec]
loop
[p : preco]
alt
[p.keySet().equals(s)]
b := removeActividade(s)
b
b
alt
[b]
Mostra mensagem de sucesso
[!b]
Mostra mensagem de insucesso

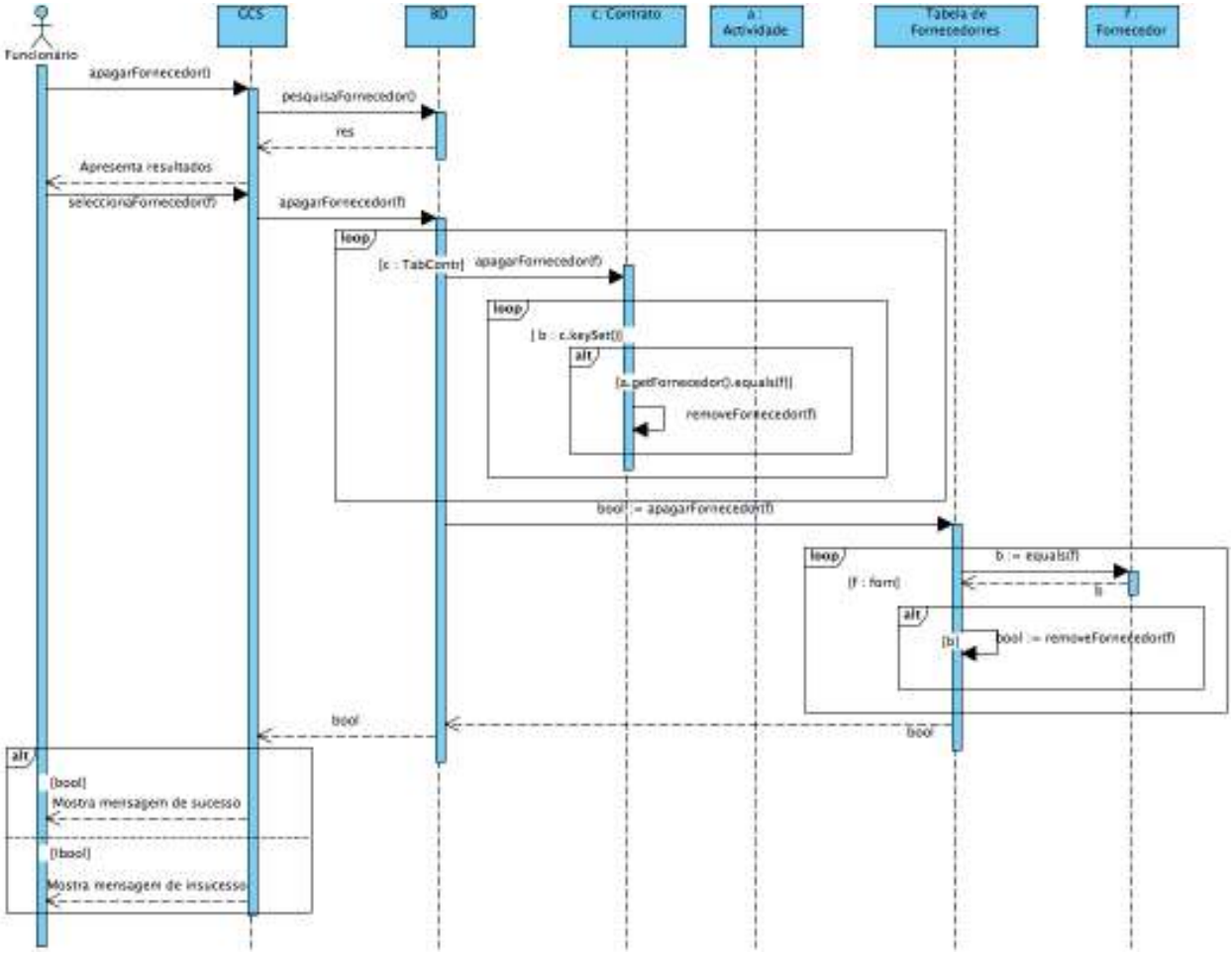
Pesquisa de Fornecedor



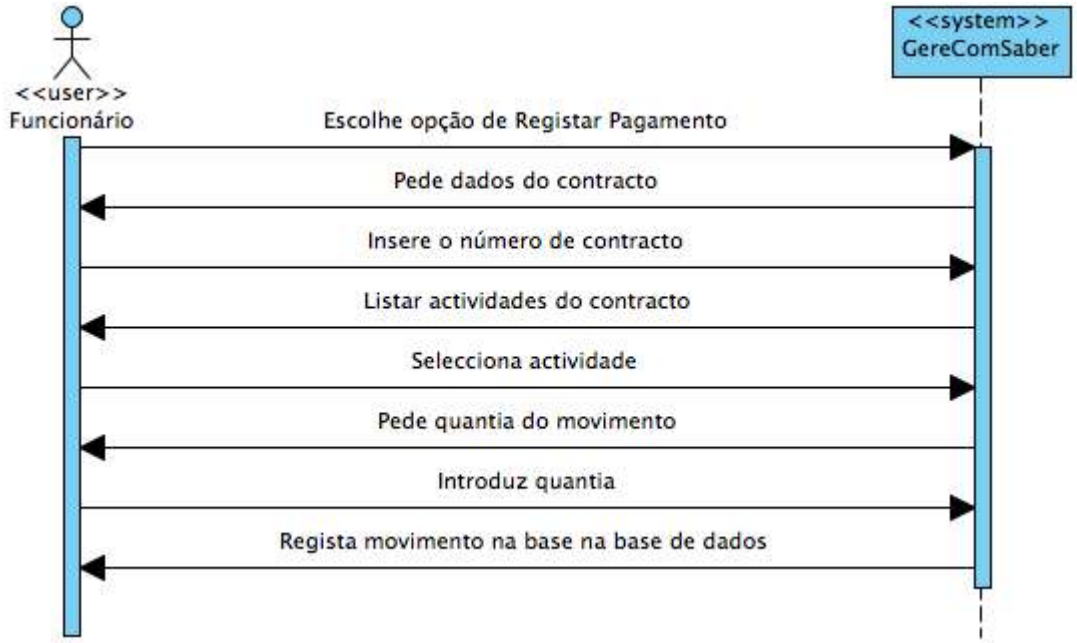


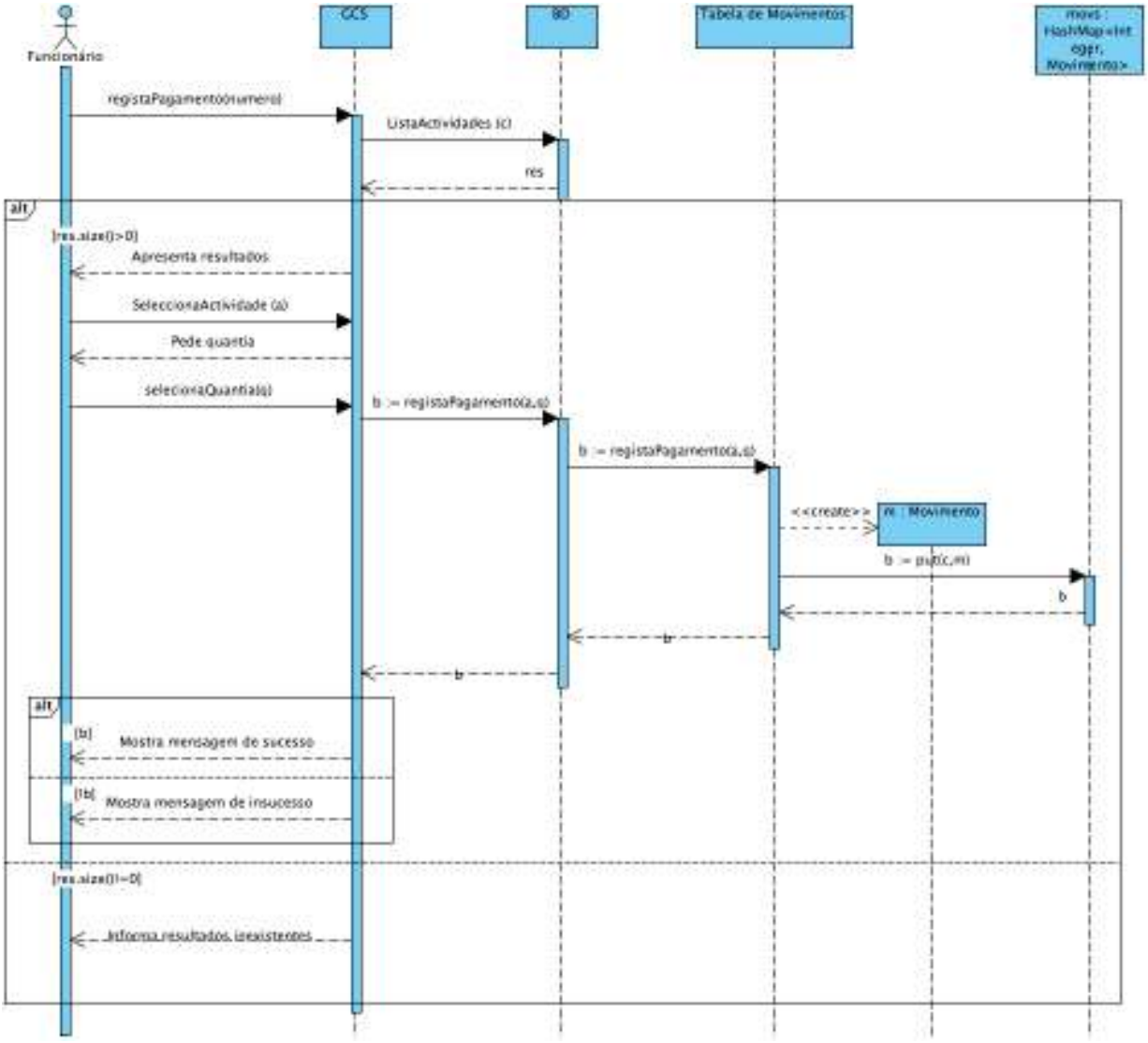
Apagar Fornecedor



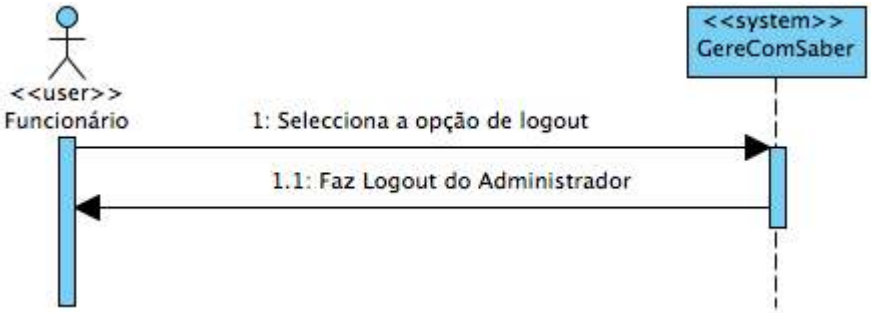


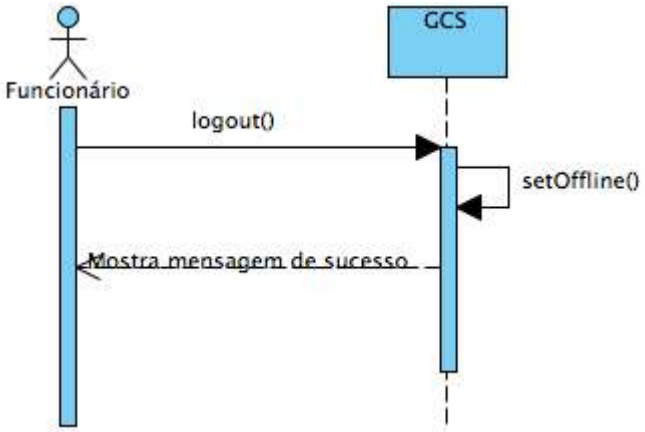
Registar Pagamentos



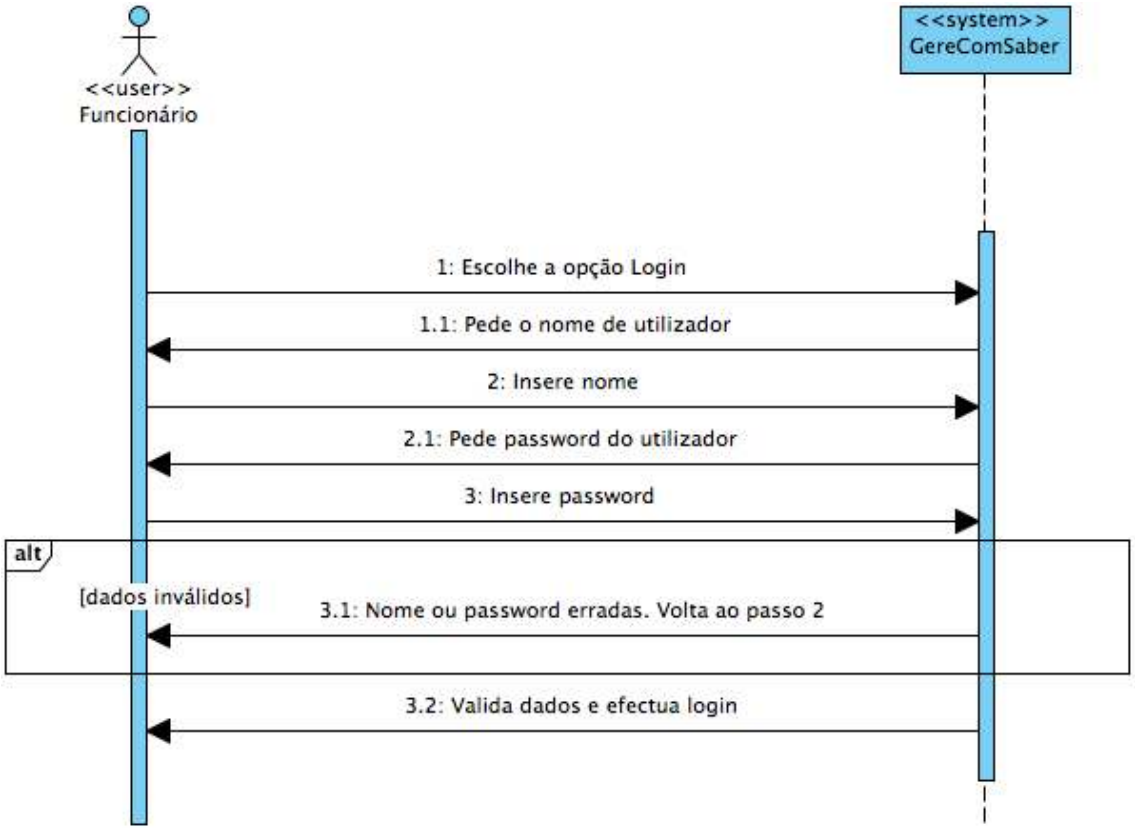


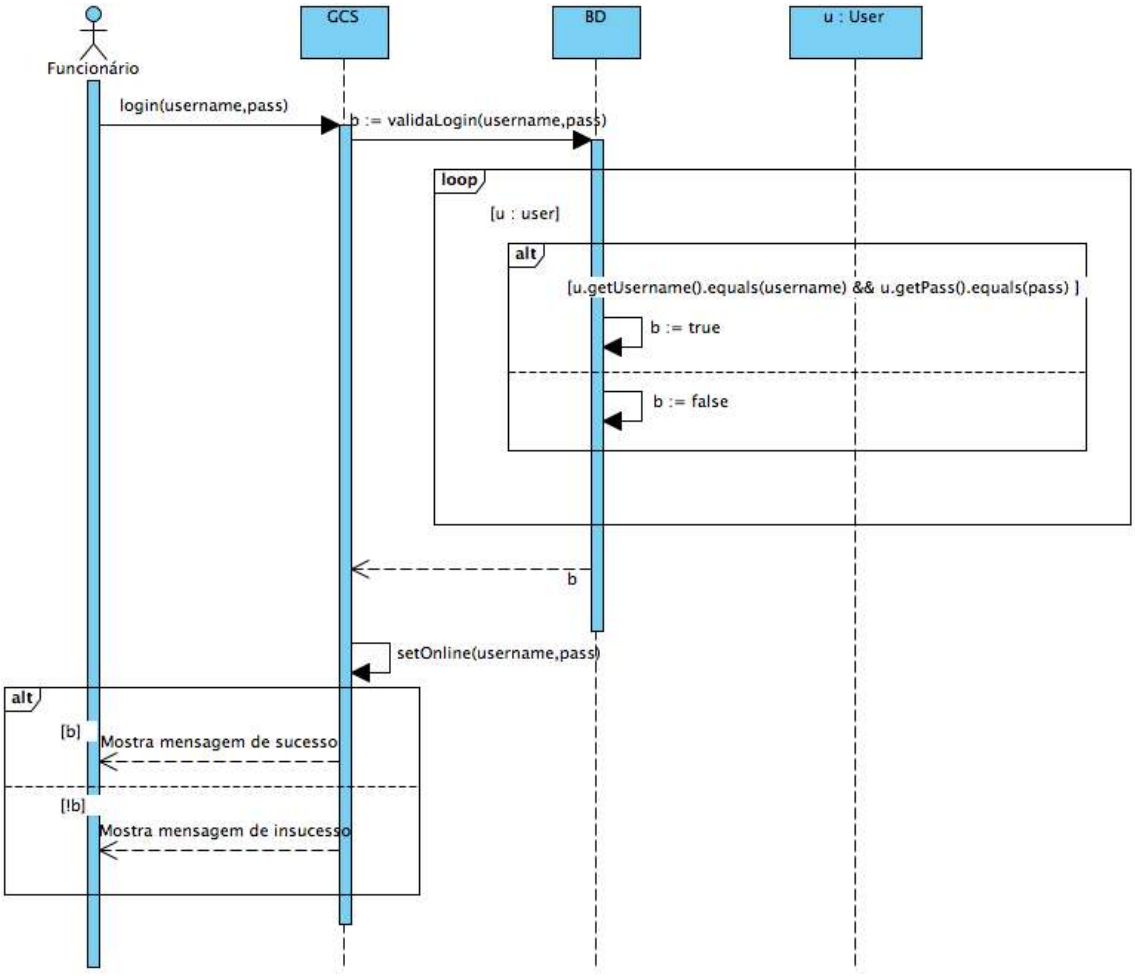
Logout


<<user>>
Funcionário
<<system>>
GereComSaber
1: Selecciona a opção de logout
1.1: Faz Logout do Administrador

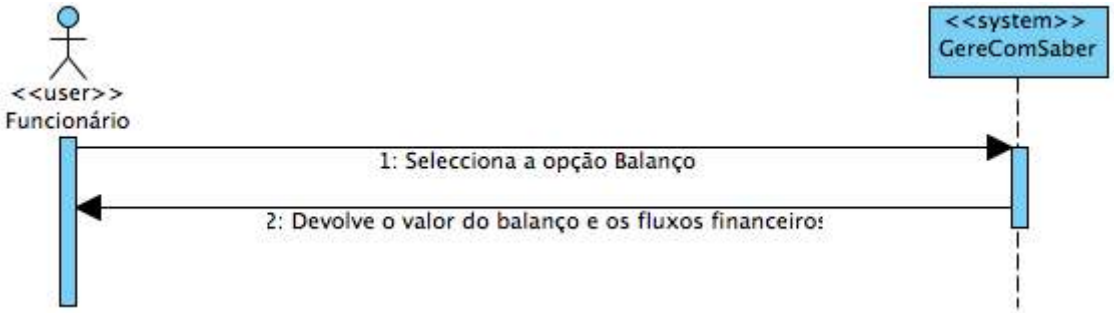

Funcionário
GCS
logout()
setOffline()
Mostra mensagem de sucesso

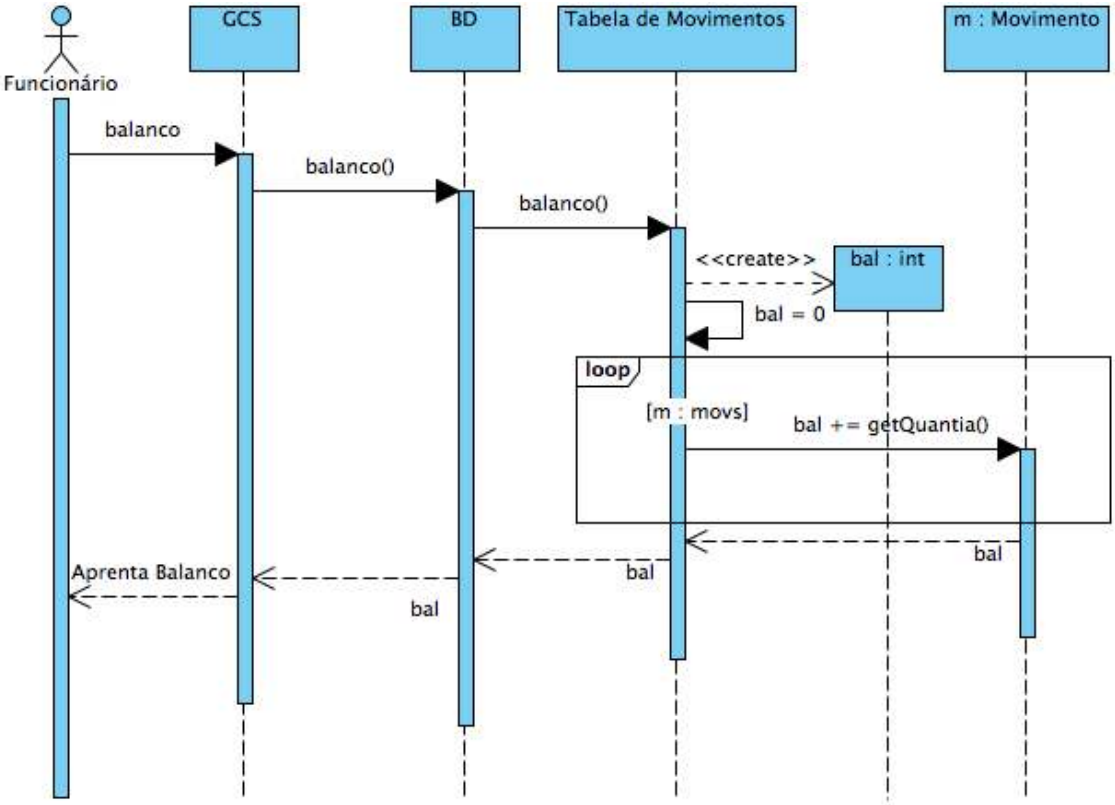
Login


<<user>>
Funcionário
<<system>>
GereComSaber
1: Escolhe a opção Login
1.1: Pede o nome de utilizador
2: Insere nome
2.1: Pede password do utilizador
3: Insere password
alt
[dados inválidos]
3.1: Nome ou password erradas. Volta ao passo 2
3.2: Valida dados e efectua login

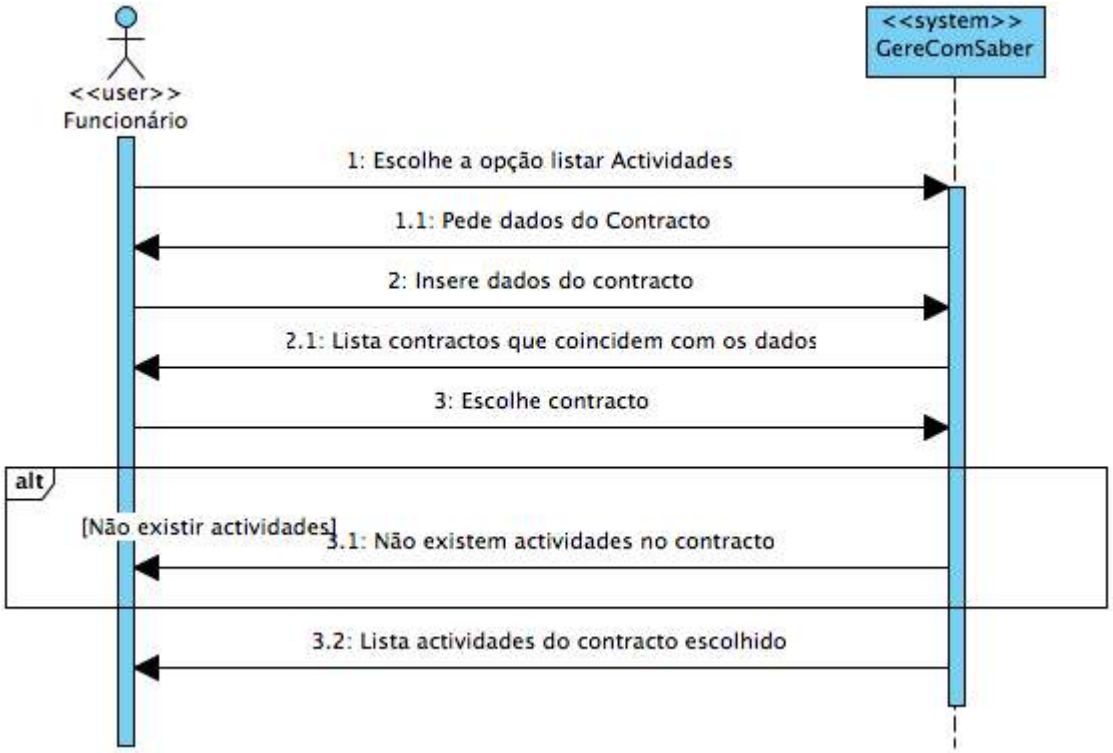

Funcionário
GCS
BD
u : User
login(username,pass)
b := validaLogin(username,pass)
loop
[u : user]
alt
[u.getUsername().equals(username) && u.getPass().equals(pass)]
b := true
b := false
b
setOnline(username,pass)
alt
[b]
Mostra mensagem de sucesso
[!b]
Mostra mensagem de insucesso

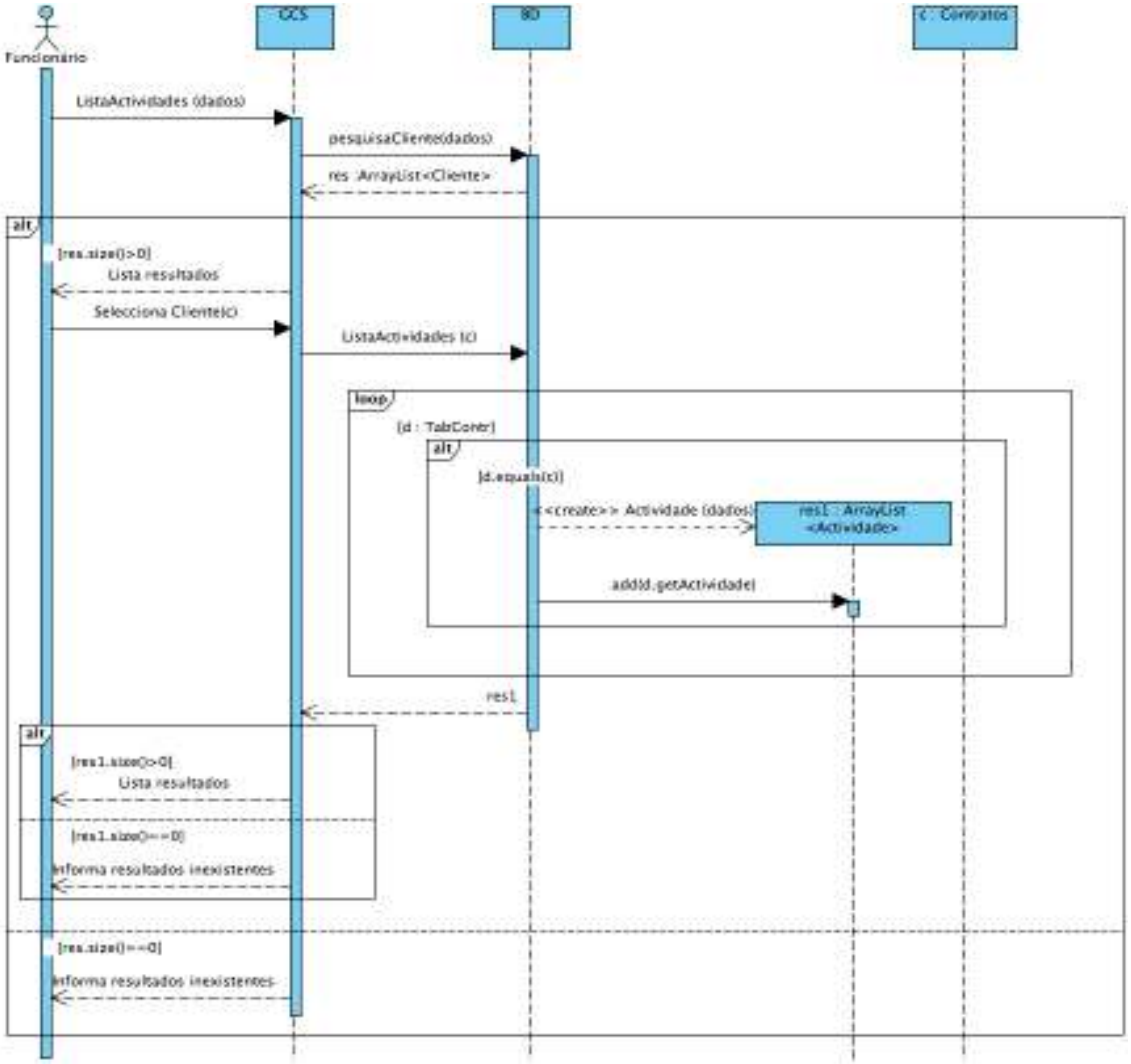
Balanço


<<user>>
Funcionário
<<system>>
GereComSaber
1: Selecciona a opção Balanço
2: Devolve o valor do balanço e os fluxos financeiros

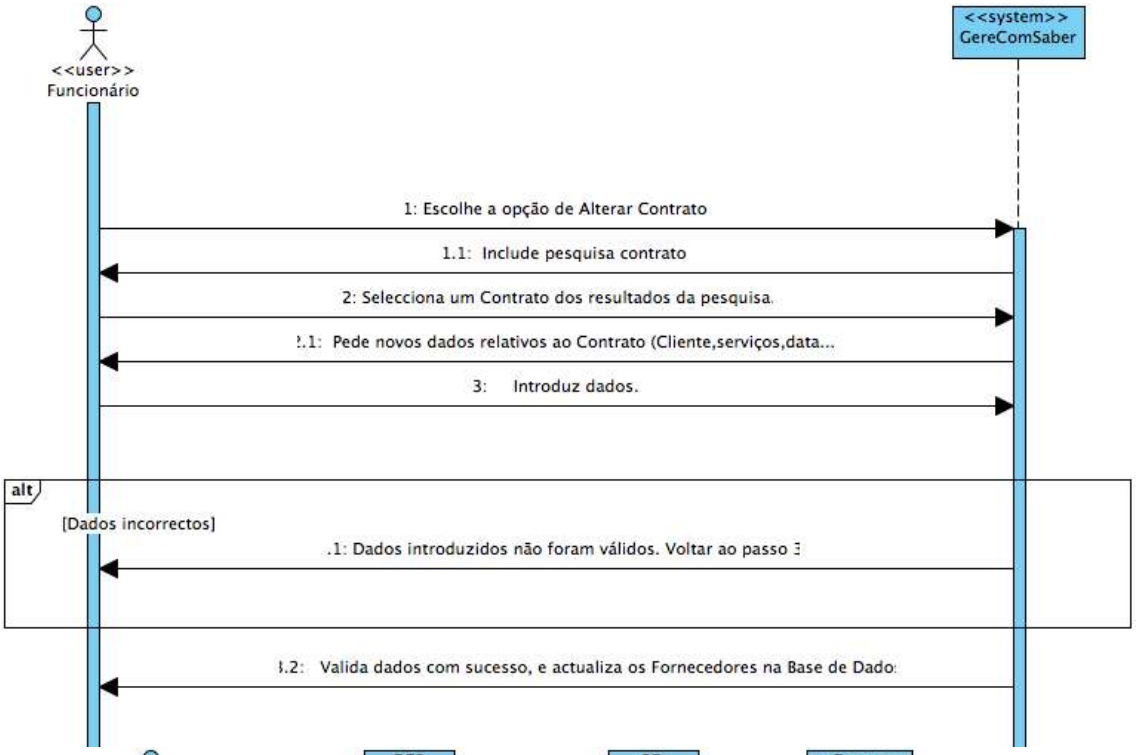

Funcionário
GCS
BD
Tabela de Movimentos
m : Movimento
balanco
balanco()
balanco()
<<create>>
bal : int
bal = 0
loop
[m : movs]
bal += getQuantia()
bal
bal
Aprenta Balanco
bal

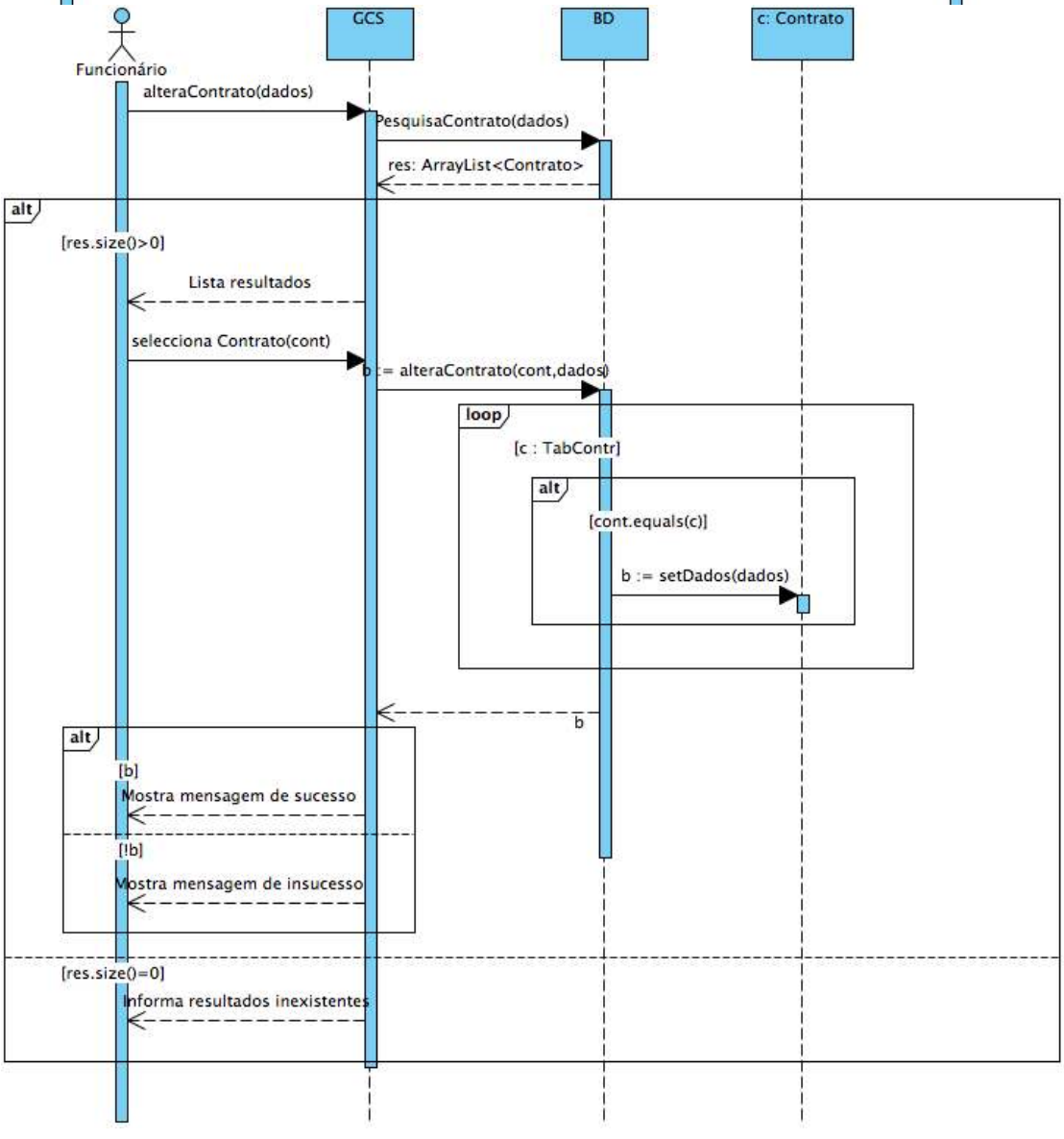
Lista de Actividades



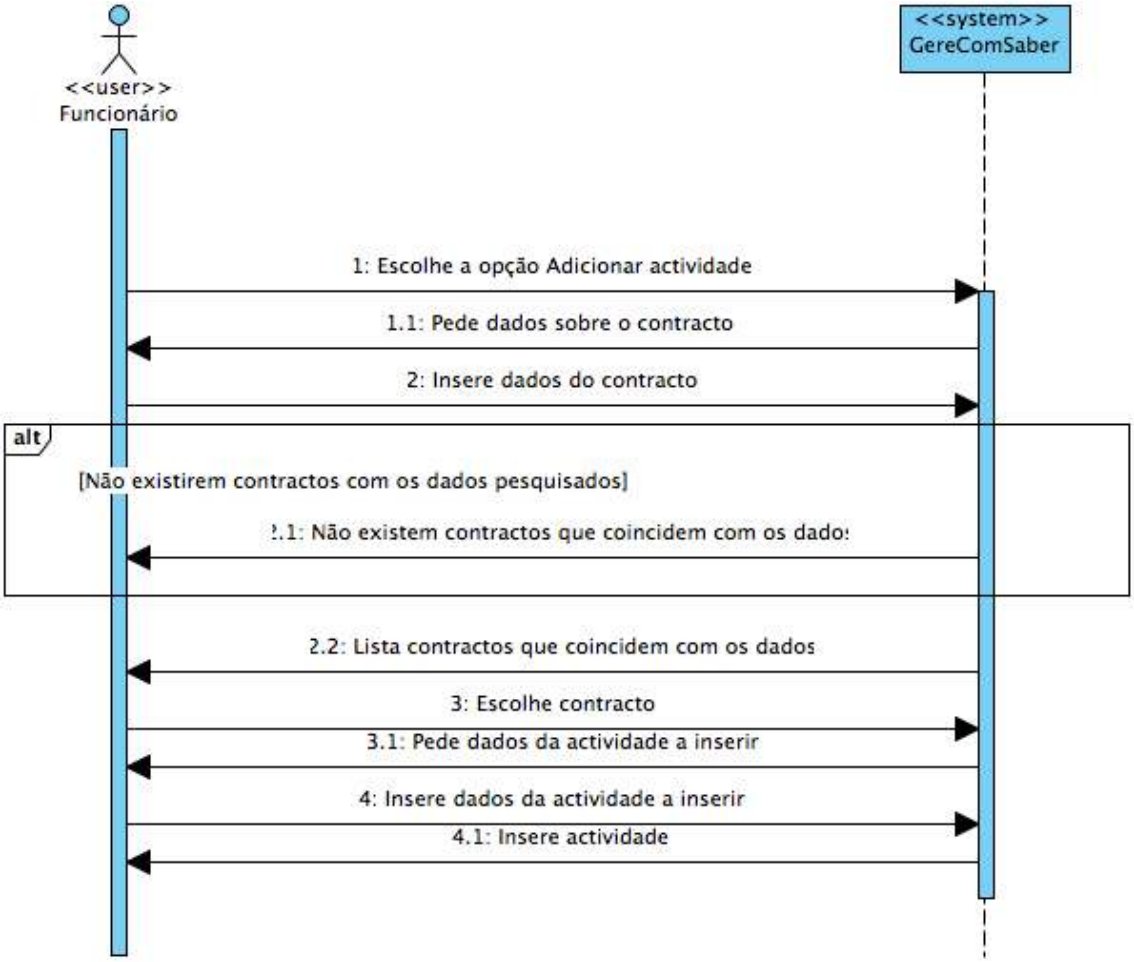


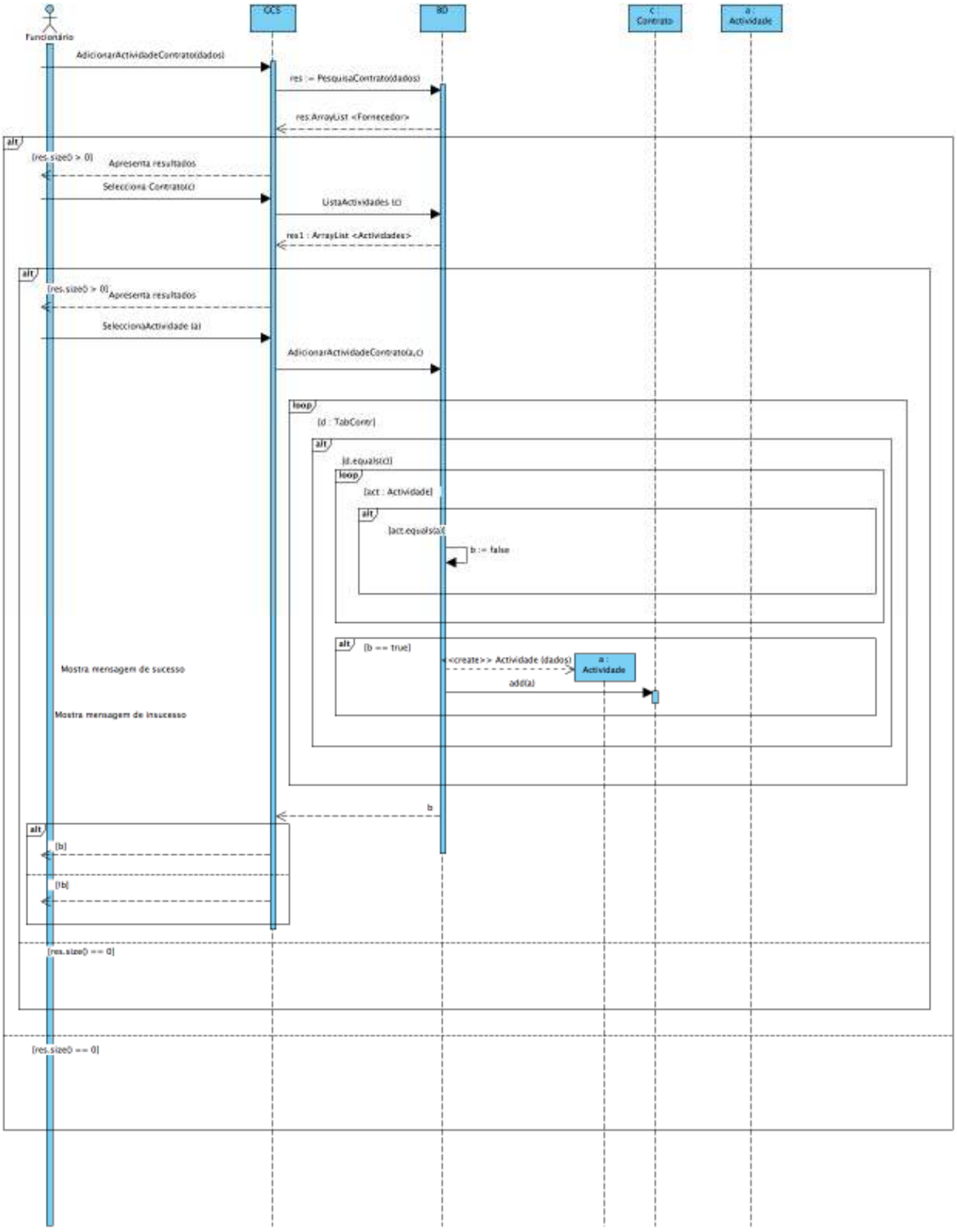
Altera contracto



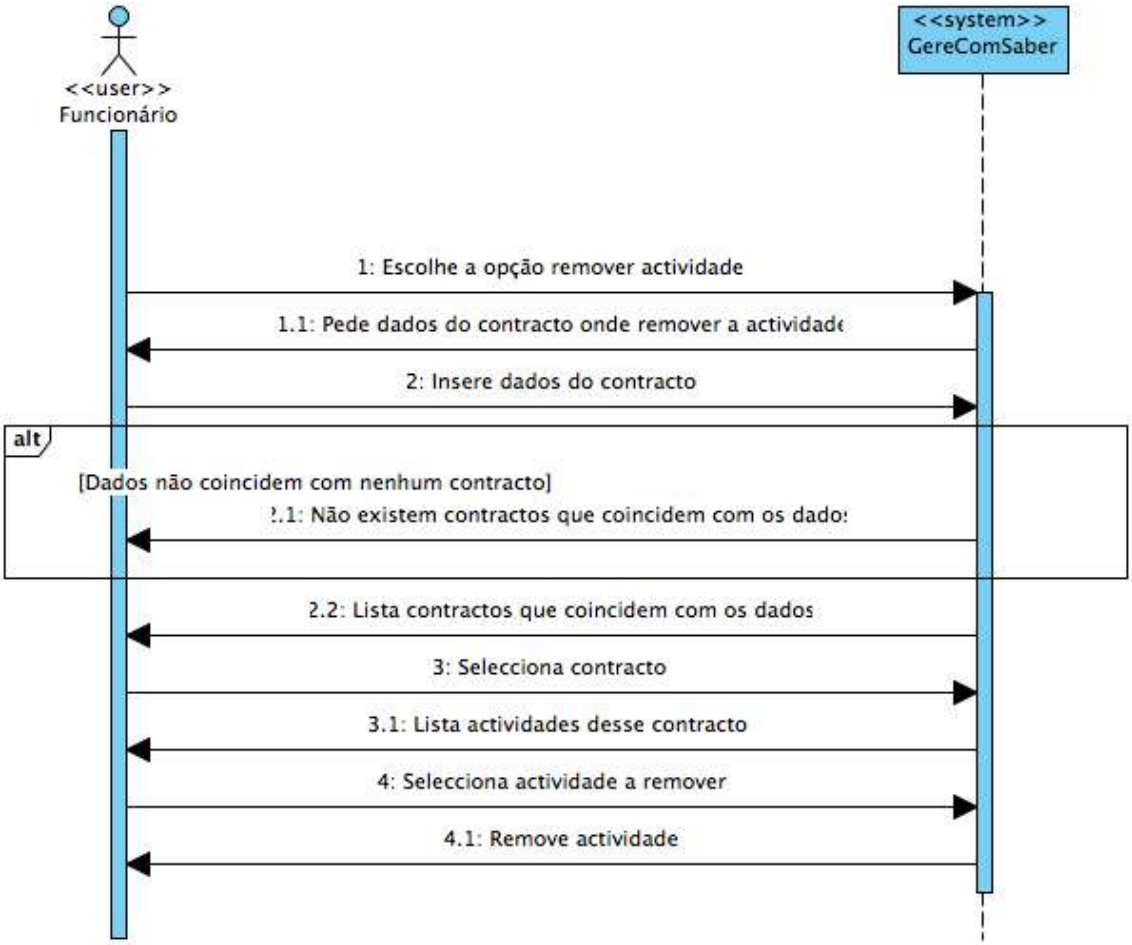


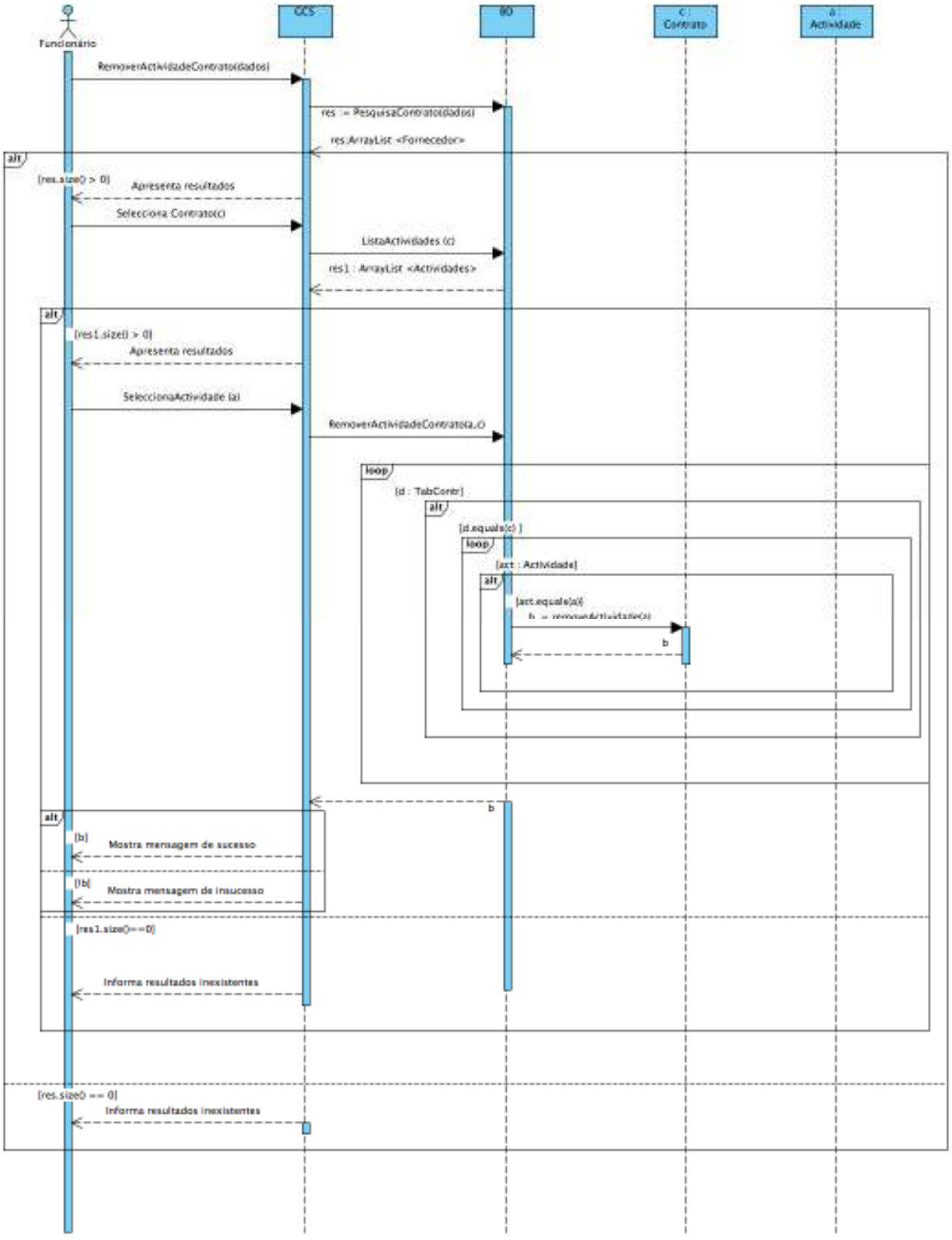
Adicionar actividade num contracto



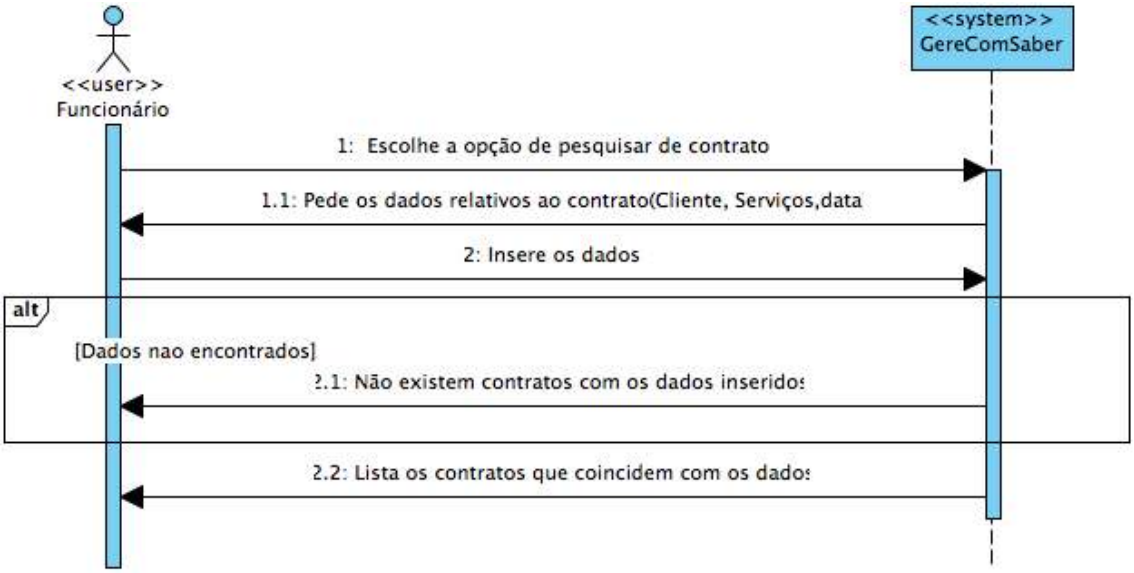

Funcionário
GCS
BD
c : Contrato
a : Actividade
AdicionarActividadeContrato(dados)
res := PesquisaContrato(dados)
res:ArrayList <Fornecedor>
alt
[res.size() > 0]
Apresenta resultados
Selecciona Contrato(c)
ListaActividades (c)
res1 : ArrayList <Actividades>
alt
[res.size() > 0]
Apresenta resultados
SeleccionaActividade (a)
AdicionarActividadeContrato(a,c)
loop
[d : TabConv]
alt
[d.equals(c)]
loop
[act : Actividade]
alt
[act.equals(a)]
b := false
alt
[b == true]
<<create>> Actividade (dados)
a : Actividade
add(a)
Mostra mensagem de sucesso
Mostra mensagem de insucesso
b
alt
[b]
[!b]
[res.size() == 0]
[res.size() == 0]

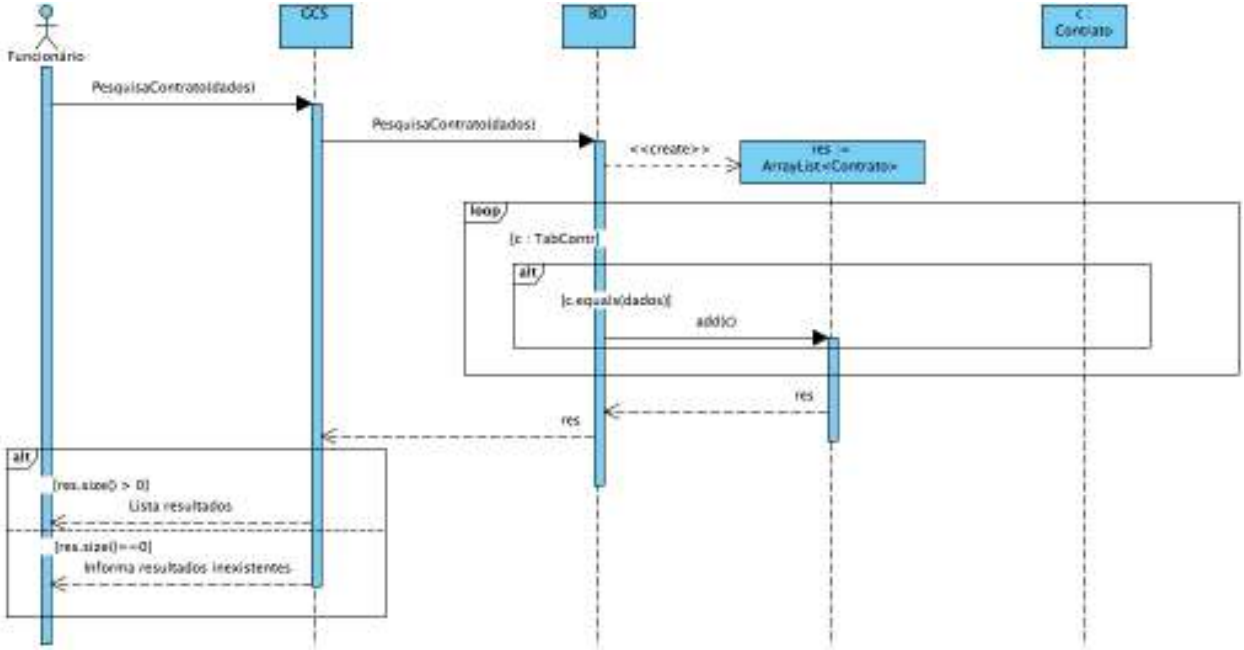
Remover actividade num contracto



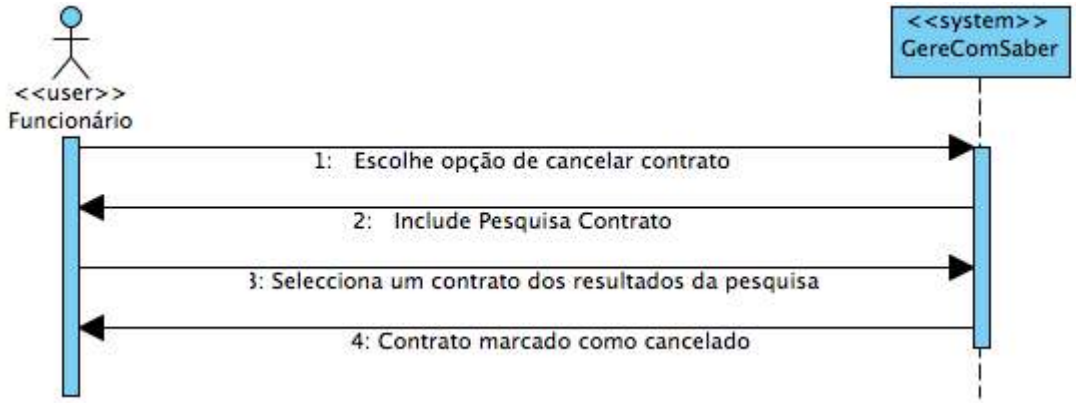

Funcionário
CCS
BD
c : Contrato
a : Actividade
RemoverActividadeContrato(dados)
res = PesquisaContrato(dados)
res:ArrayList <Fornecedor>
alt
[res.size() > 0]
Apresenta resultados
Selecciona Contrato(c)
ListaActividades (c)
res1 : ArrayList <Actividades>
alt
[res1.size() > 0]
Apresenta resultados
SeleccionaActividade (a)
RemoverActividadeContrato(a,c)
loop
[d : TabContr]
alt
[d.equals(c)]
loop
[act : Actividade]
alt
[act.equals(a)]
b
b
alt
[b]
Mostra mensagem de sucesso
[!b]
Mostra mensagem de insucesso
[res1.size()==0]
Informa resultados inexistentes
[res.size() == 0]
Informa resultados inexistentes

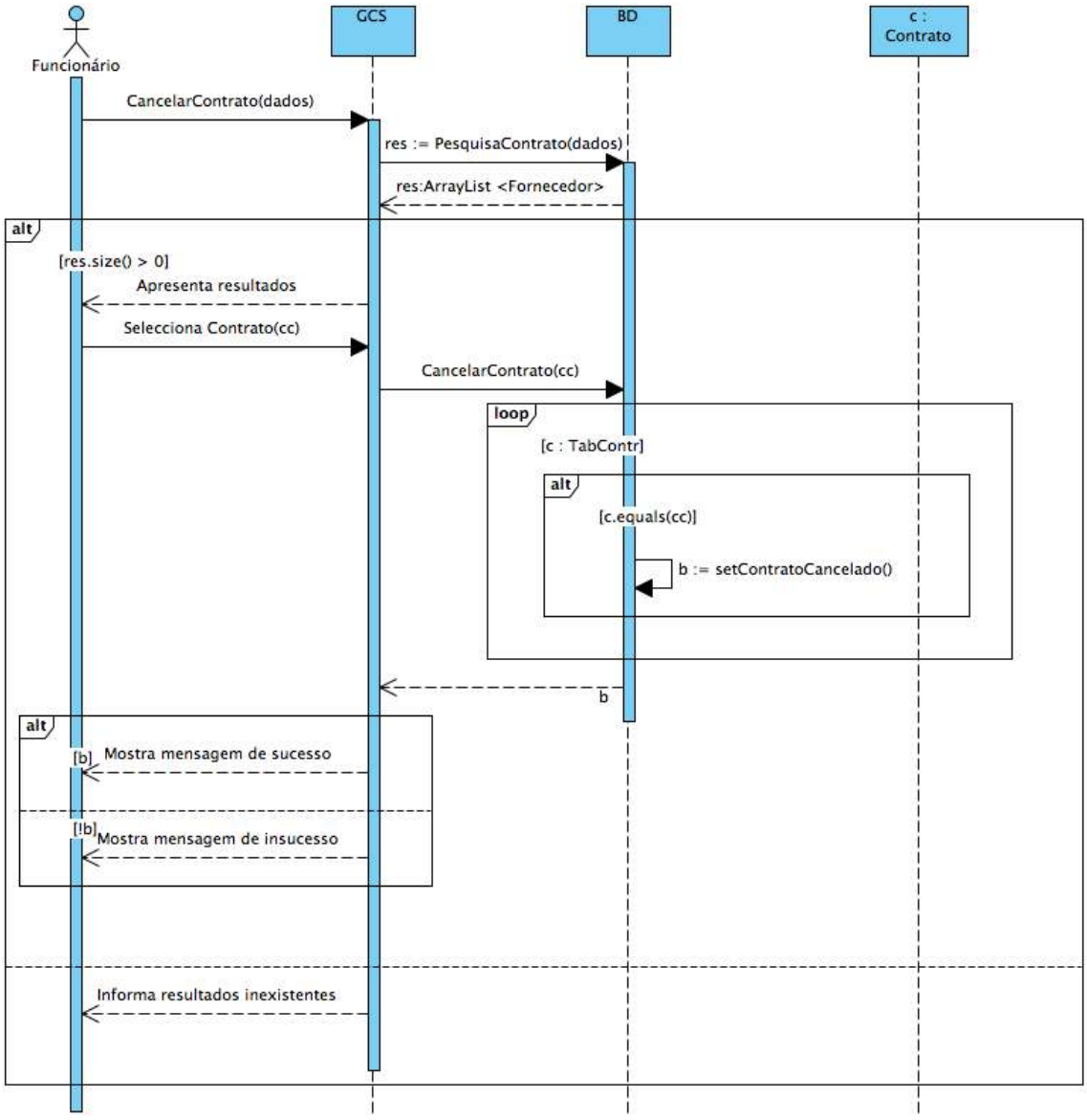
Pesquisa contracto





Cancela contracto





ALTERAÇÕES À FASE ANTERIOR

Todas as considerações anteriores especialmente de apresentação de conceitos que se consideram essenciais para a compreensão do trabalho desenvolvido nesta área, deixaram claro que os processos que definiram poderiam e eram, em primeira instância reconhecidos como tendenciosamente aptos a serem “refinados”.

Por não ser necessário este capítulo não apresentará mais conceitos mas evocará sim os anteriormente apresentados para sustentar a ideia justificativa do diagrama de use cases anterior ter sido alterado. Após mais uma submissão do trabalho (i.e. mais uma fase de desenvolvimento da aplicação) foi notório para os elementos do grupo que o diagrama de use-cases estaria muito genérico em certos pontos e por isso foi alterado.

CONCLUSÃO

Fica salvaguardada a posição de que se pode novamente refinar algum ponto do projecto na próxima submissão do trabalho, como já foi adiantado.

Nesta fase do trabalho foi notório que o UML não tem o estatuto de "estaticidade", visto que quanto mais se aplica mais se observa que há passos dados anteriormente que poderiam estar mais completos. Esse facto é só observado à medida que se vai construindo mais o modelo.

Ainda que possa parecer um pouco irrelevante e extremamente trabalhoso, o UML facilita o processo de programação. Note-se é a diferença de facilidade a nível de perda de tempo, objectividade e eficiência que se pode ter ao refinar um diagrama em UML, ao invés de o fazer na altura da aplicação da linguagem de programação.