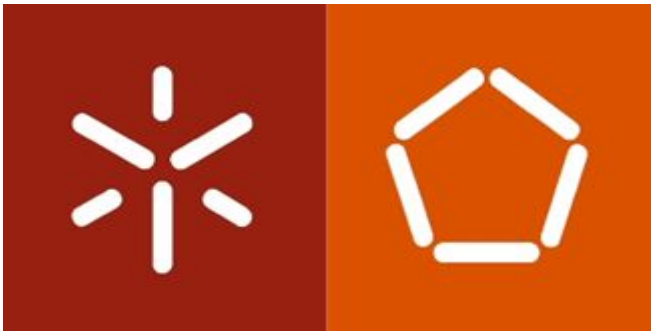
Desenvolvimento de Sistemas de Software

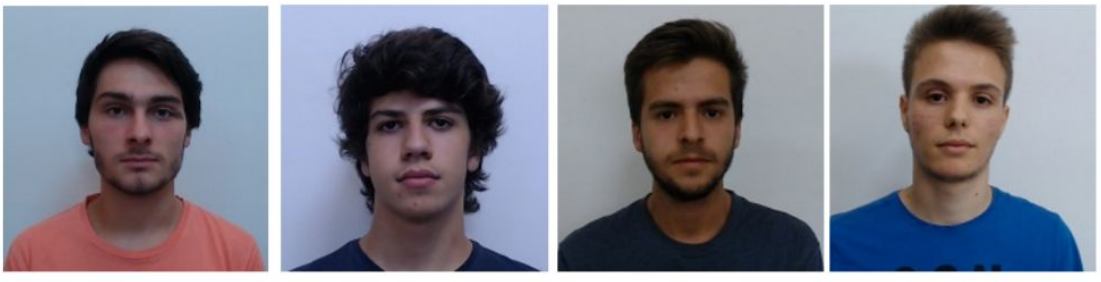
Relatório escrito da terceira fase do projeto prático

Ano letivo 2019/2020



Universidade do Minho - Mestrado Integrado Engenharia Informática

Grupo nº 9



Hugo Matias - a85370 João Abreu - a84802 Duarte Vilar - a85517 Tiago Magalhães - a84485

1 - Introdução

No âmbito da unidade curricular de *Desenvolvimento de Sistemas de Software* foi nos proposto o desenvolvimento de um *media center*, não logo começando a programar, mas sim a modelar. O propósito deste tipo de abordagem é agilizar o processo de compreensão do problema que se encontrava à nossa frente, de modo a arranjar soluções de forma mais eficiente e produtiva, de maneira a termos um software bem estruturado e de fácil manutenção.

O faseamento deste projeto permitiu-nos entrar nesse mecanismo de desenvolvimento de programas de forma mais pedagógica e simples, remetendo a elaboração prática para esta fase final, onde tudo o que fora elaborado nas tarefas anteriores, seria posto em óptica e em uso. É nesta fase que os elementos do grupo vêm o fundamento de toda a esquematização de diagramas anteriormente criados, e se apercebem das razões que levam a que este tipo de planeamento estrutural seja utilizado. A percepção teórica apresentada posteriormente à primeira e segunda fase deste projeto, nas aulas teóricas e práticas, incutiu no grupo ideias mais bem formadas para a construção final deste programa, permitindo que se tomassem decisões basilares para o sucesso deste projeto.

É de notar que todas as alterações feitas entre a segunda e a presente fase são neste relatório apontadas, de modo a justificar todo o raciocínio apresentado. Para além disso reservamo-nos ao uso de imagens para uma melhor apresentação da modelação, mockups, diagramas de use case, entre outros, sempre o melhor justificadas possível, de modo a manter uma apresentação mais limpa e de melhor compreensão para quem lê este manuscrito.

2 - Modelação

1.Modelo de domínio

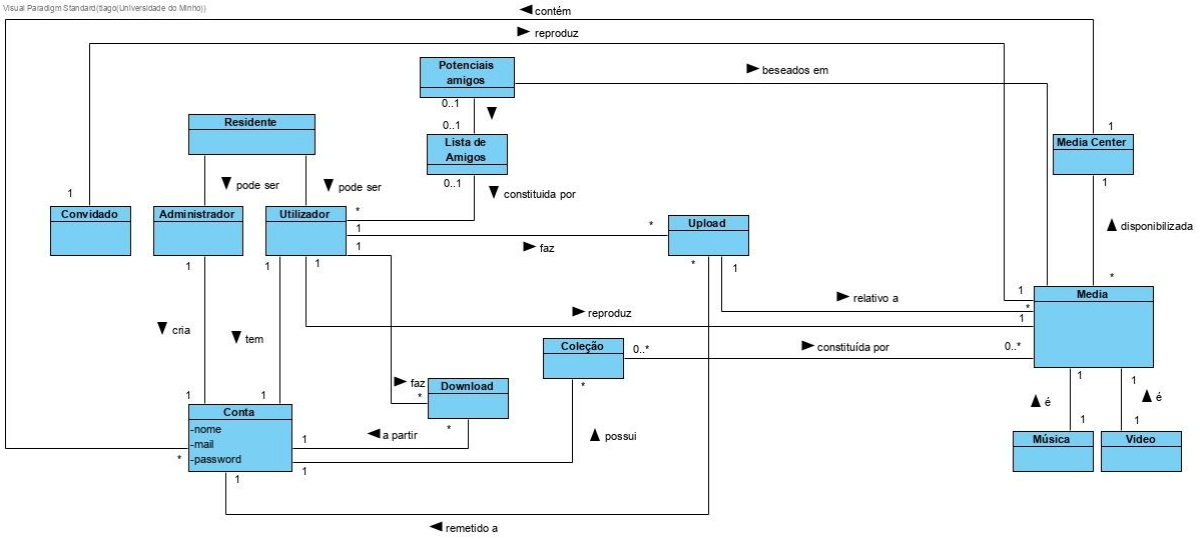


Fig 1-Modelo domínio 1ª fase.

Na 1ª fase do projeto construímos o modelo de domínio presente na imagem acima, que nos ajudou a desconstruir, analisar e a compreender o enunciado e a perceber o que o este, que num caso prático seria um cliente, pedia.

No entanto com avançar do projeto achamos que devia ser atualizado o diagrama, devido à existência de alguma entidades que não fazem sentido neste, bem como o nome de algumas entidades. Sendo assim,o diagrama atualizado encontra-se abaixo apresentado:

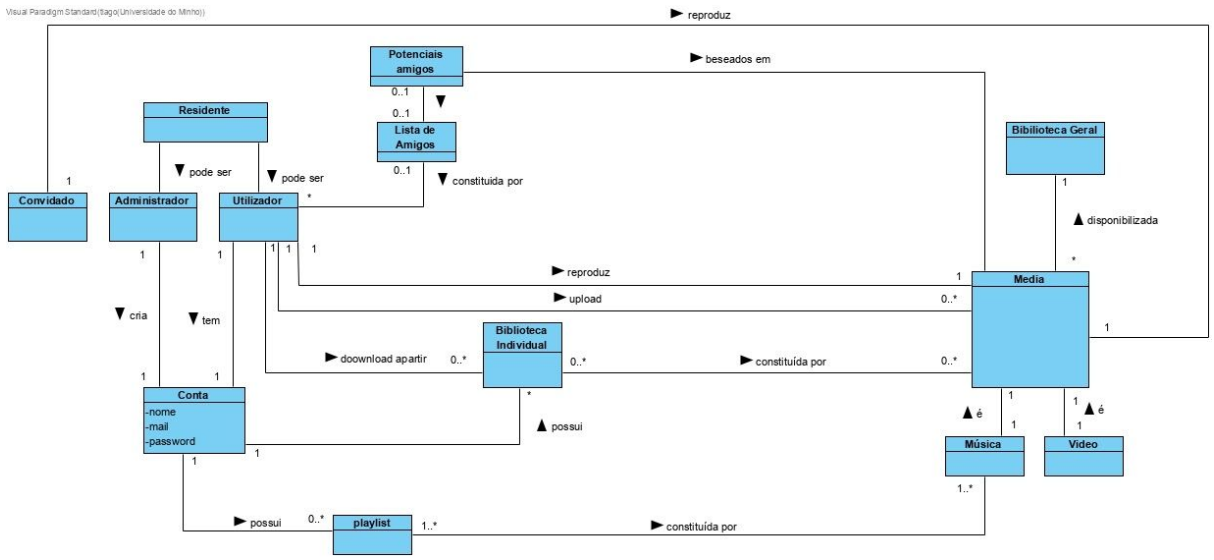


Fig 2-Modelo domínio 1ª fase alterado.

2.Modelo de Use Cases

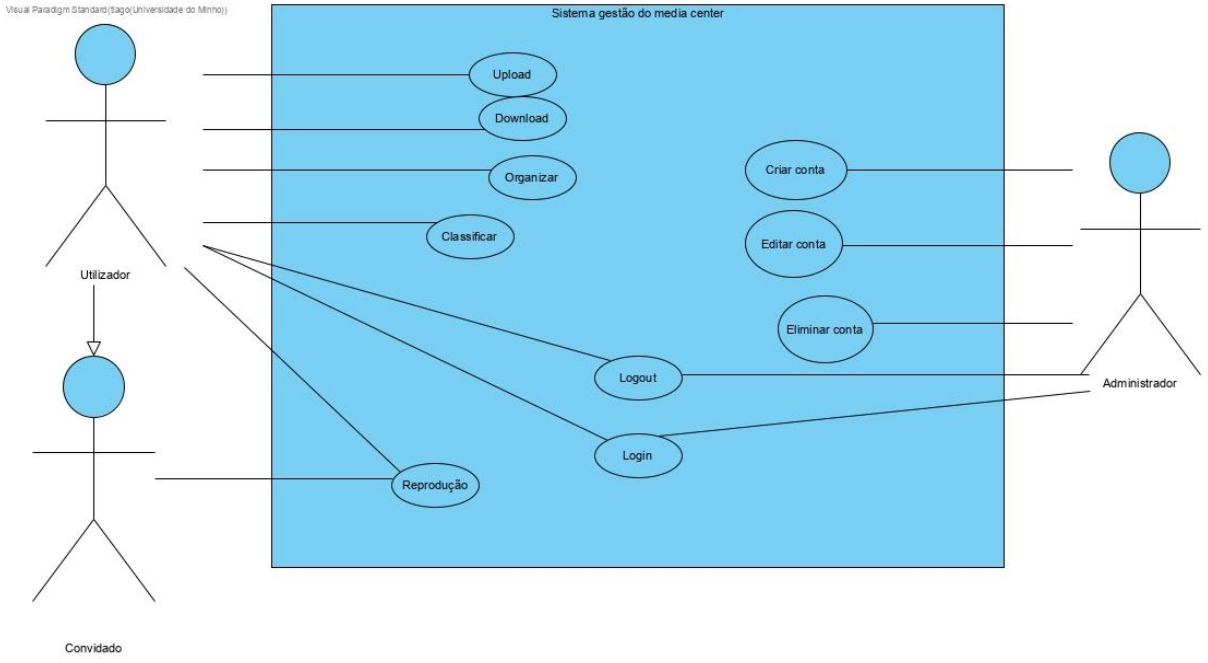


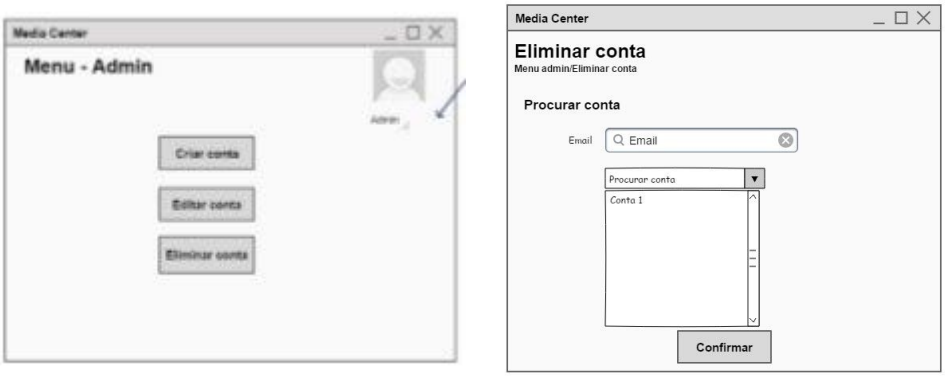
Fig 2-Modelo de Use Cases 2ª fase.

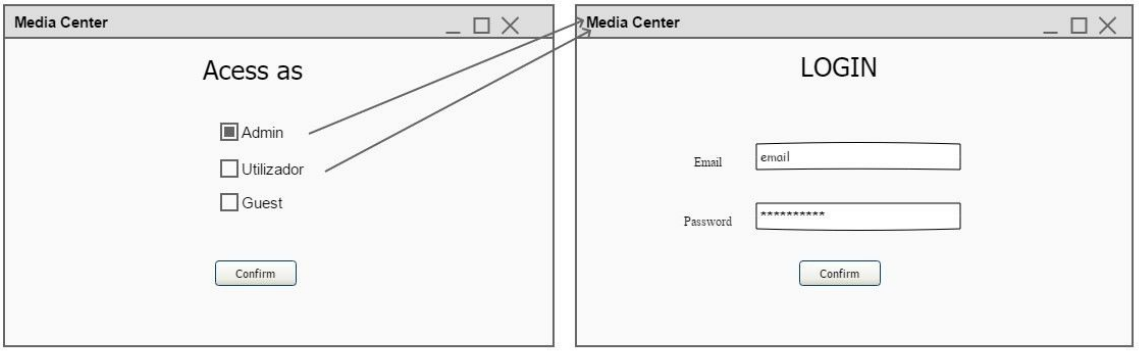
De acordo com o modelo de domínio, identificamos serviços que deveríamos fornecer para elaborar o modelo de use cases. Primeiramente identificamos quais os atores que iriam fazer parte do sistema. Assim, verificamos que iria existir um Administrador com funções exclusivamente administrativas, um utilizador e um convidado. Depois

identificamos para cada um dos atores os use cases que nos pareceram necessários para cada tipo de ator.

3.Prótotipo da interface

Nesta secção encontra-se um protótipo de interface, com baixa fidelidade, simples e rápido. (É de notar que não houve um comprometimento fixo ao formato que esta interface apresenta desde o estado prematuro deste projeto até à sua implementação). Á parte disso, permitiu-nos conhecer informação que o ator deve fornecer e conhecer.





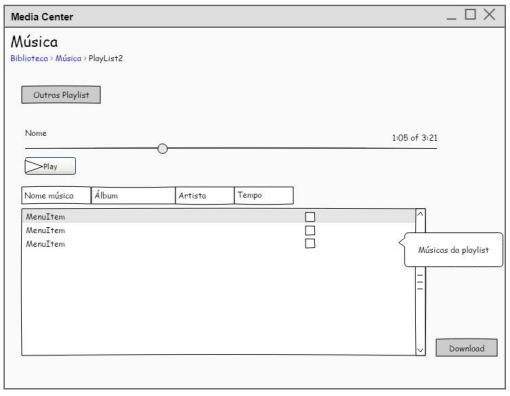


Fig 3-Mockups

4. Use cases diagramas

1.Diagrama de sequência com subsistemas para cada use case

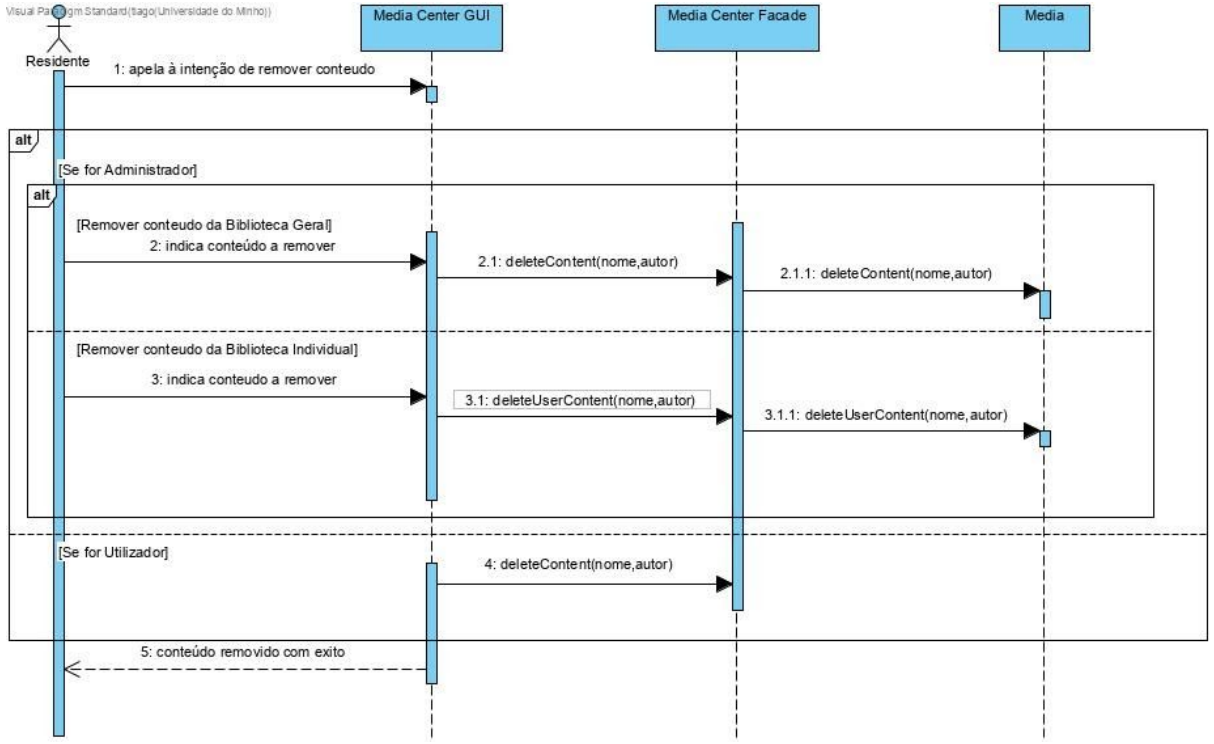


Fig 4-Diagrama de sequência com subsistema para usecase remover conteúdo.

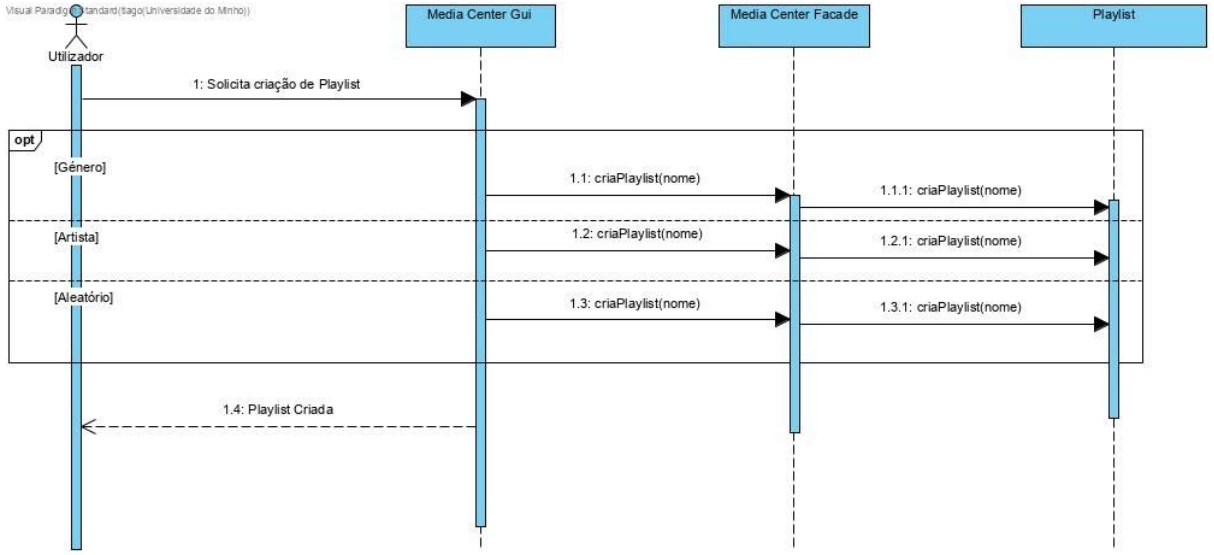


Fig 5-Diagrama de sequência com subsistema para usecase criar playlist.

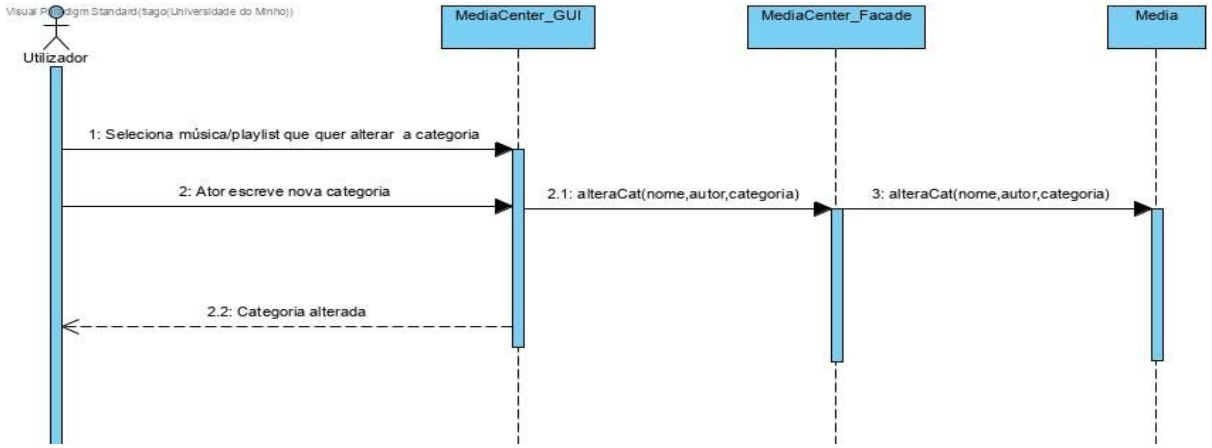


Fig 6-Diagrama de sequência com subsistema para usecase alterar categoria do conteúdo.

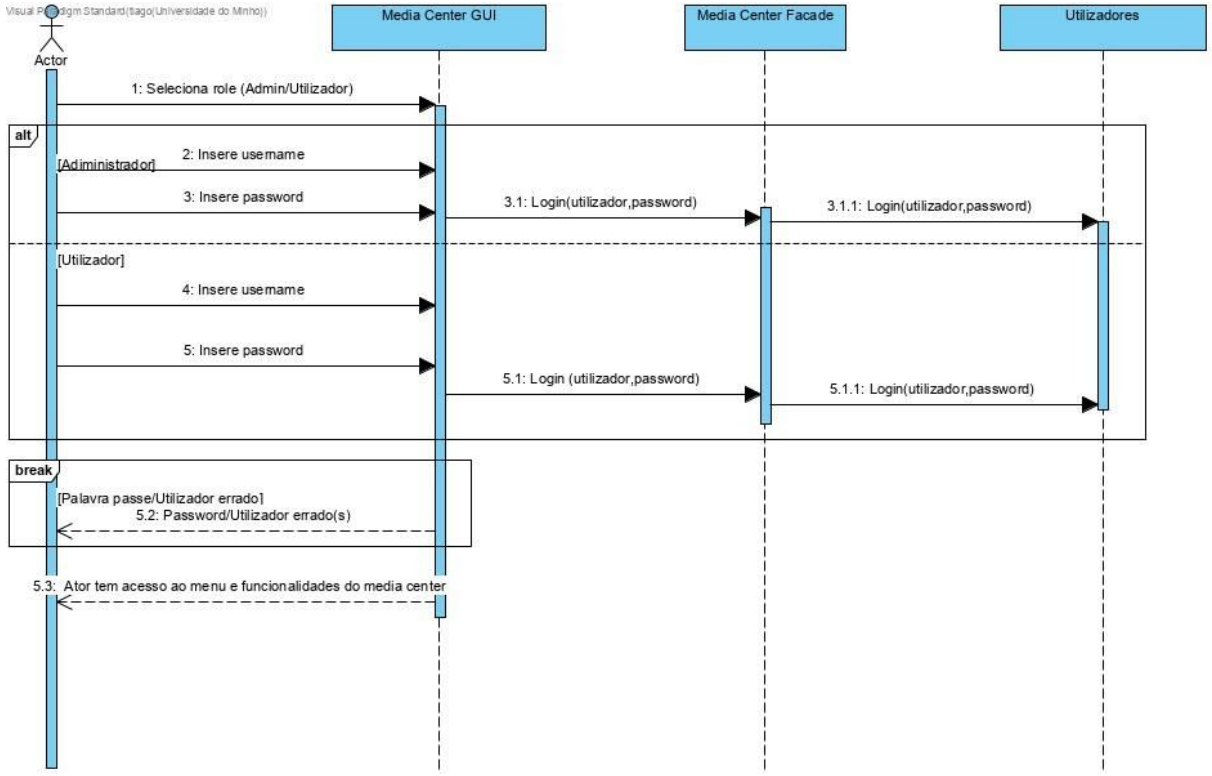


Fig 7-Diagrama de sequência com subsistema para usecase login..

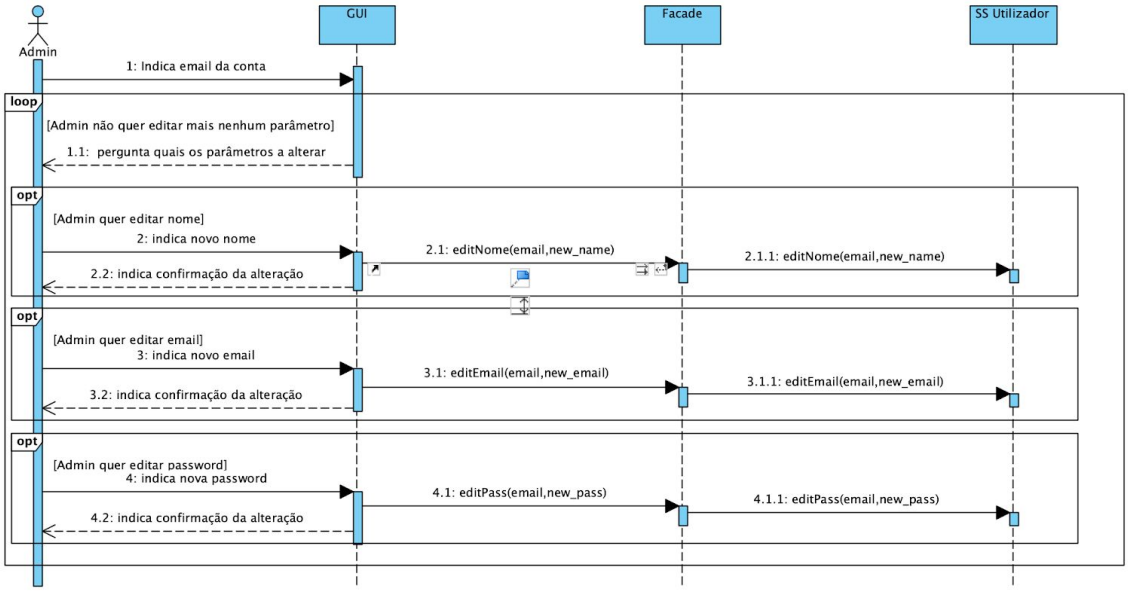


Fig 8-Diagrama de sequência com subsistema para usecase editar utilizador.

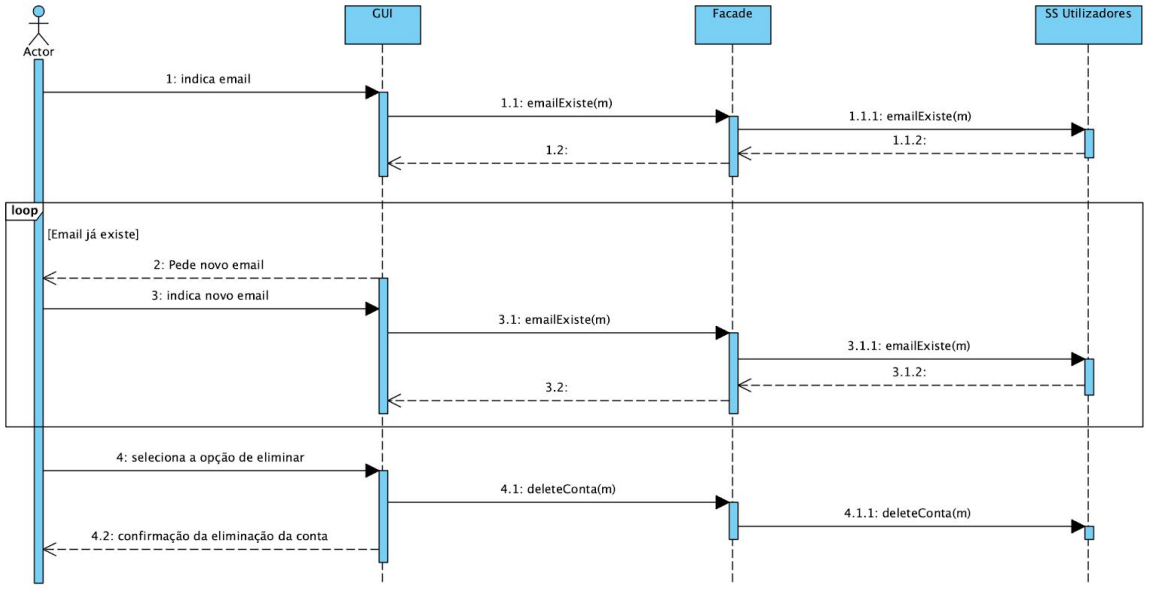


Fig 9-Diagrama de sequência com subsistema para usecase eliminar utilizador.

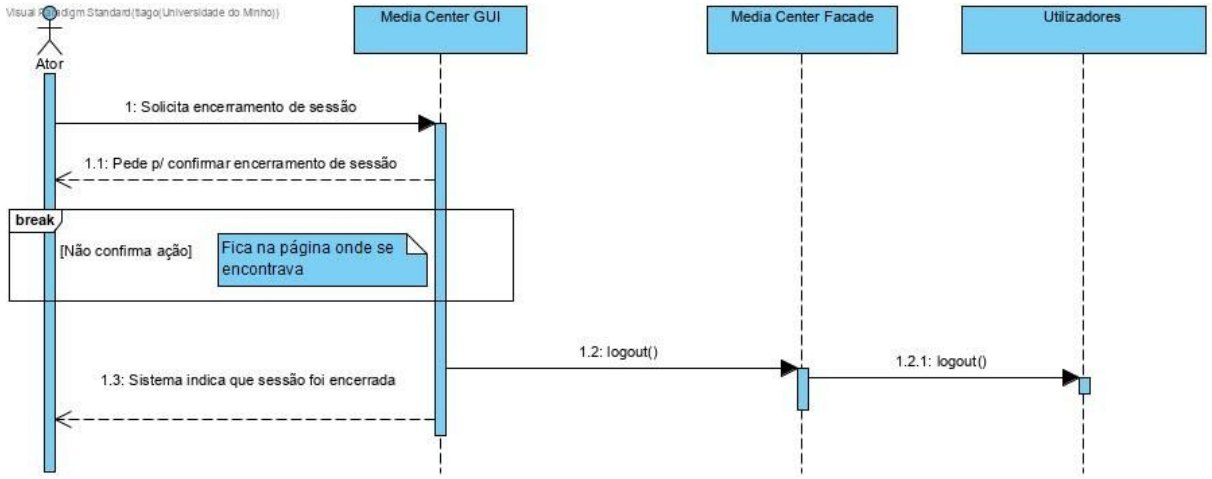


Fig 10-Diagrama de sequência com subsistema para usecase terminar sessão.

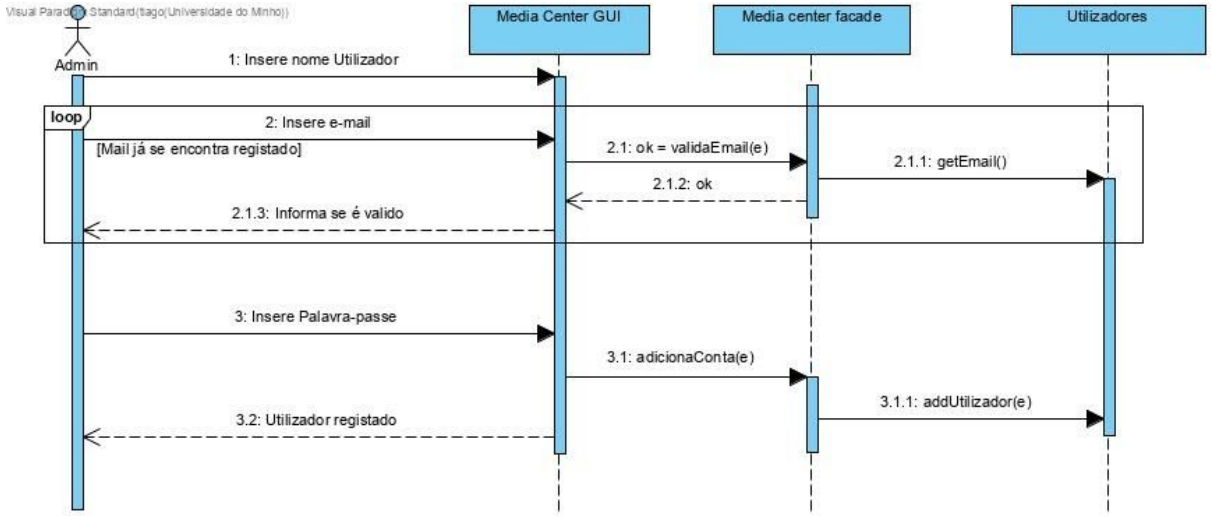


Fig 11-Diagrama de sequência com subsistema para usecase registar utilizador.

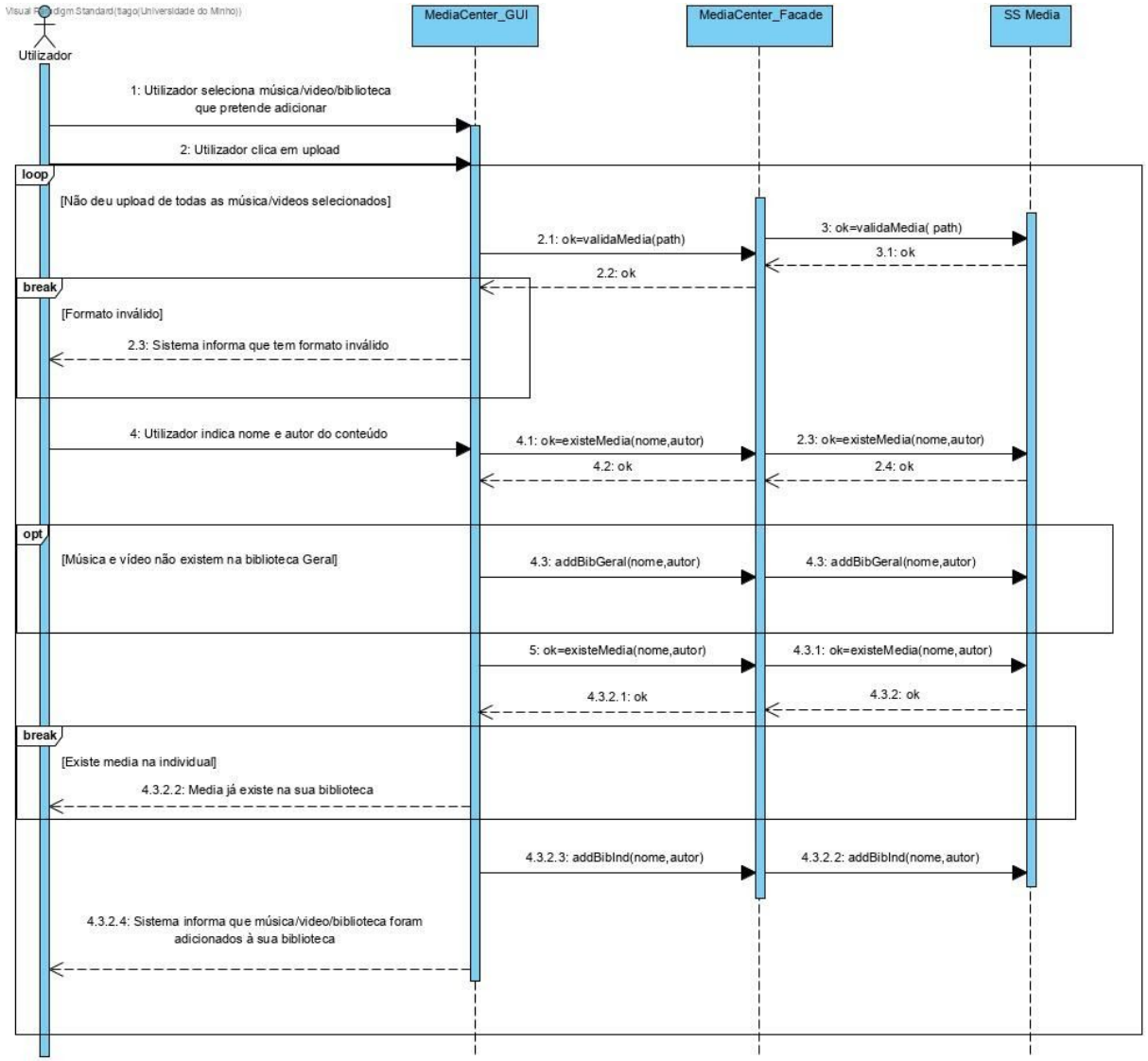


Fig 12-Diagrama de sequência com subsistema para usecase fazer upload conteúdo.

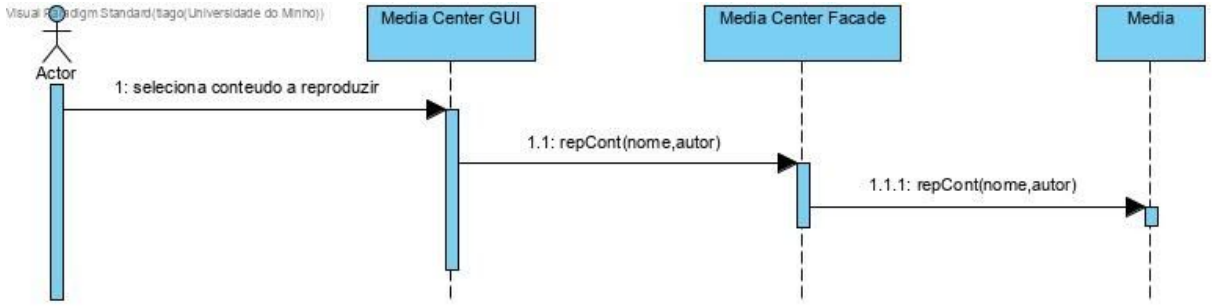


Fig 13-Diagrama de sequência com subsistema para usecase reproduzir conteúdo.

2.Diagrama de packages

O diagrama de packages permitiu-nos ajudar a começar a estruturar o programa de acordo com a arquitetura MVC, bem como estabelecer relações entre as entidades dos modelos de domínio. Este diagrama começou a tornar-se mais nítido quando estávamos a realizar os diagramas de sequência com subsistemas para cada use case.

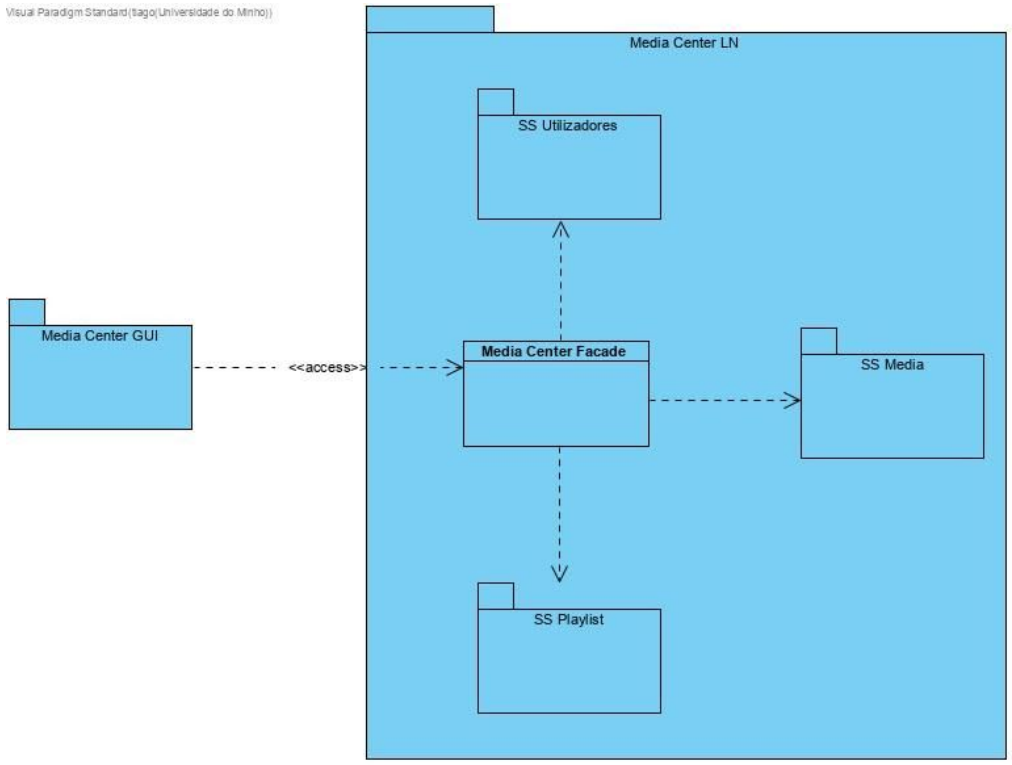


Fig 14-Diagrama de packages

3.Diagrama de classes

A estruturação deste diagrama,ajudou-nos a começar a pensar já como poderia ficar em código o projeto, desde as estruturas que iremos utilizar, classes, objetos

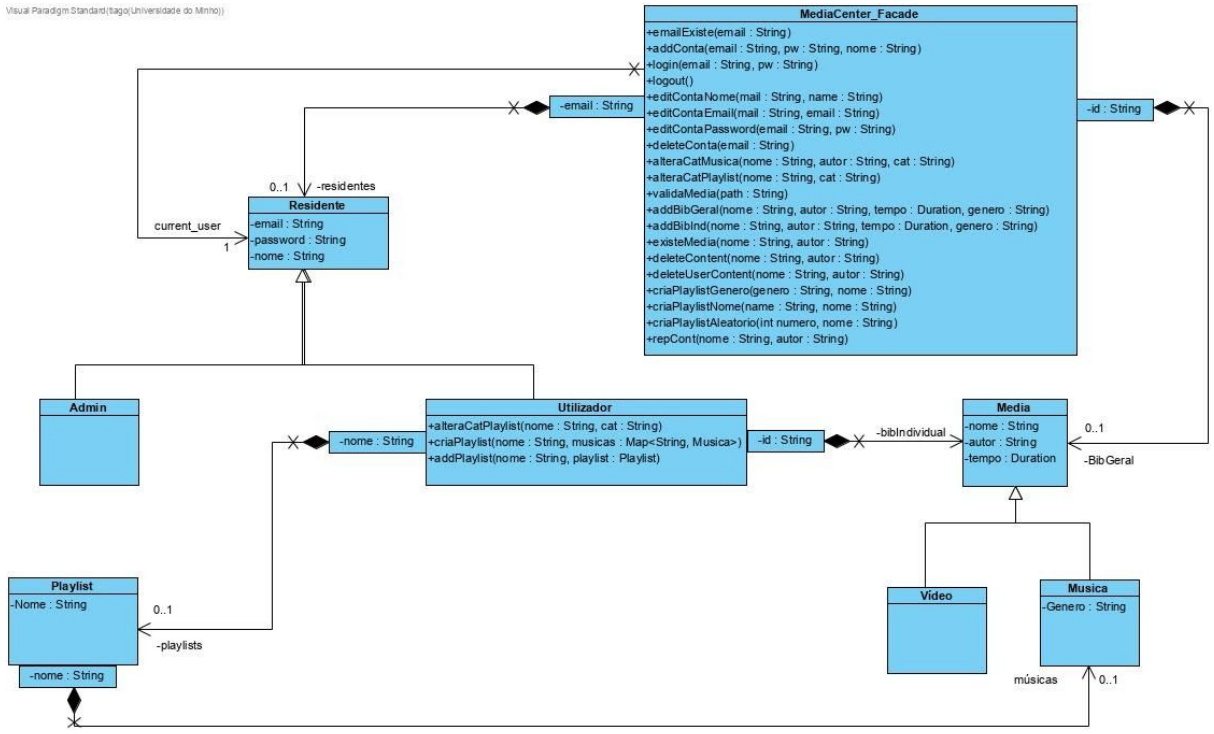


Fig 15-Diagrama de classes

4.Diagrama de sequência das operações de cada subsistemas

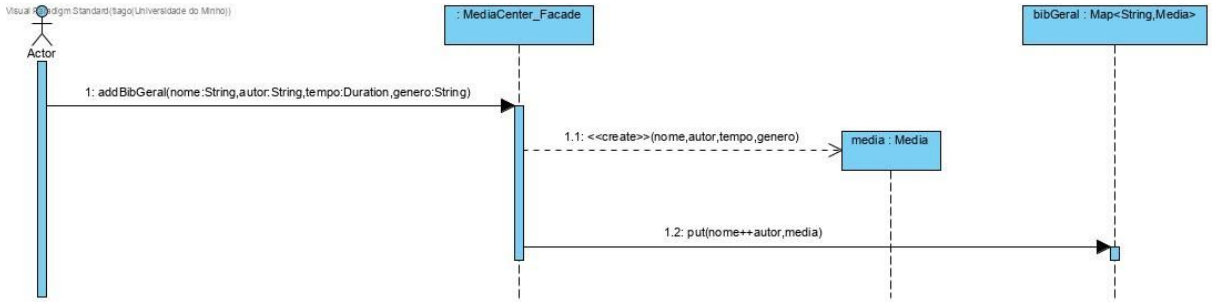


Fig 16-Diagrama de sequência da operação addBibGeral

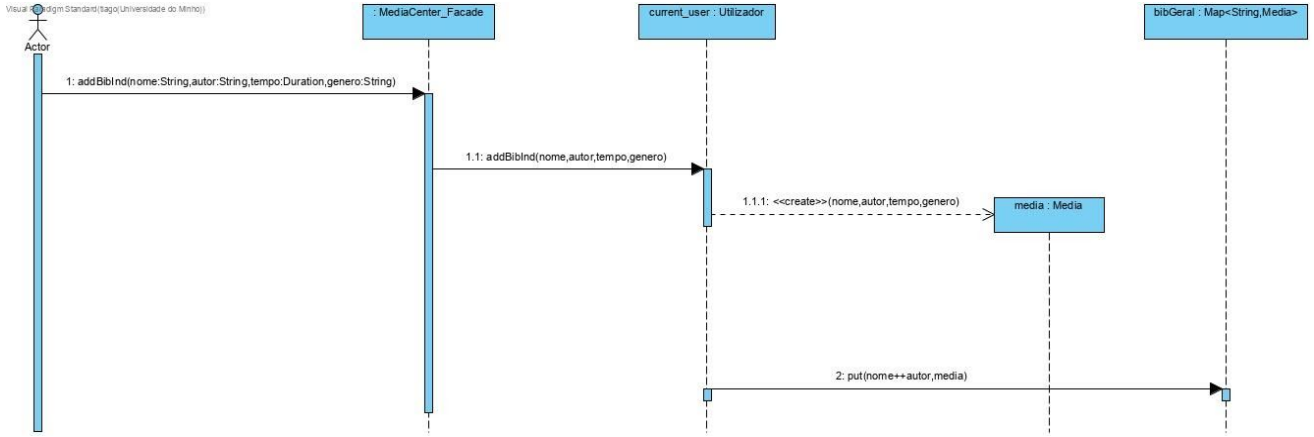


Fig 17-Diagrama de sequência da operação addBibInd

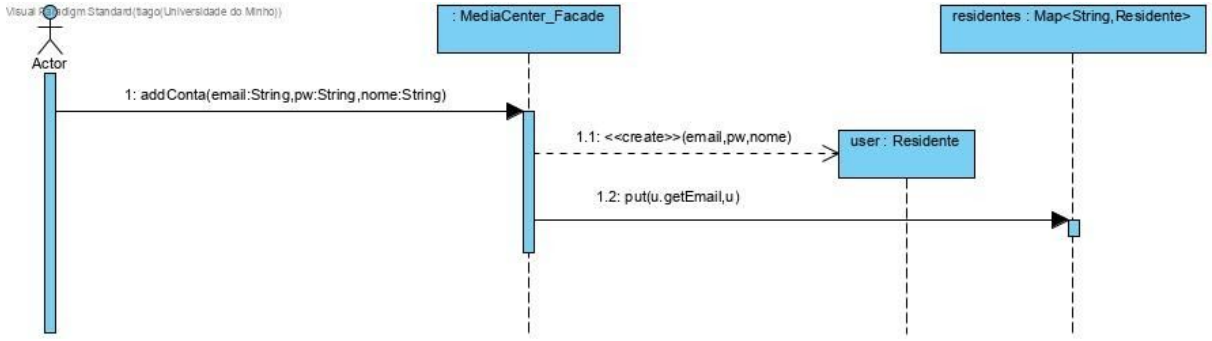


Fig 18-Diagrama de sequência da operação addConta

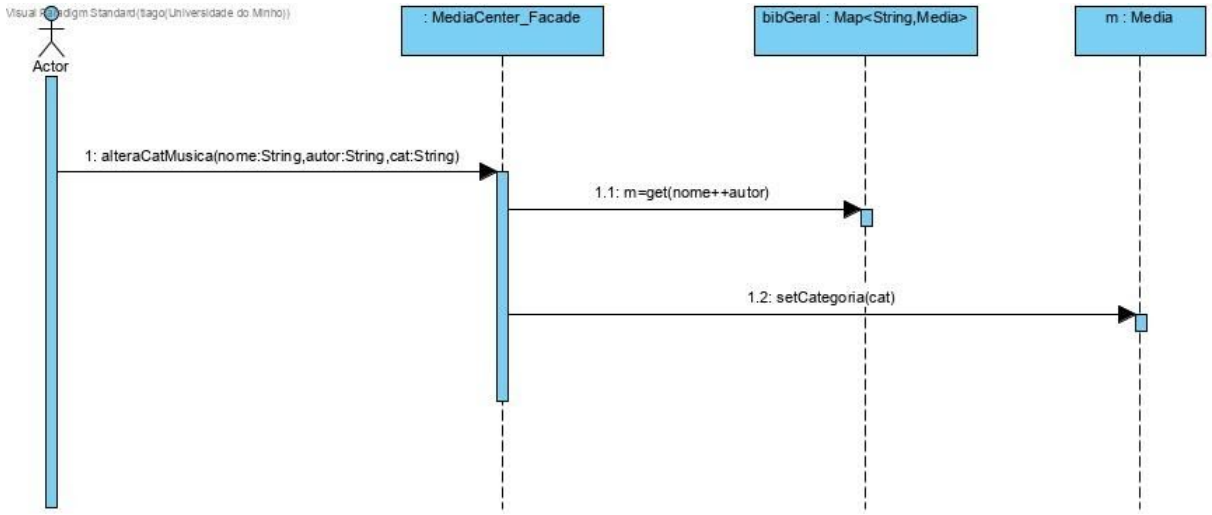


Fig 19-Diagrama de sequência da operação alteraCatMusica

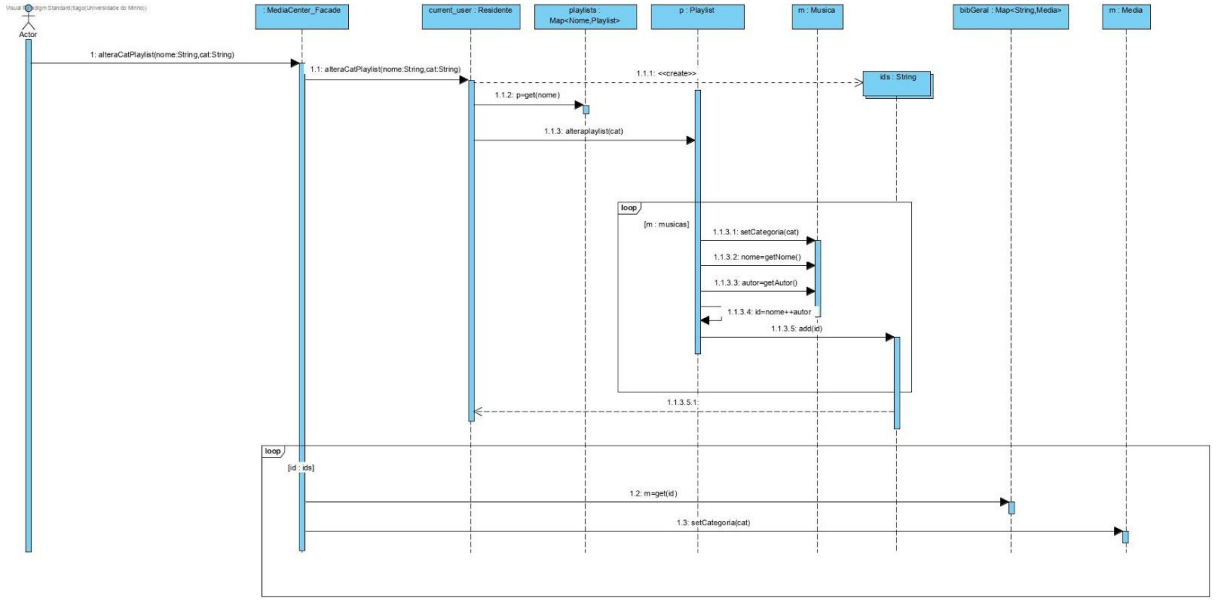


Fig 20-Diagrama de sequência da operação alteraCatPlaylist

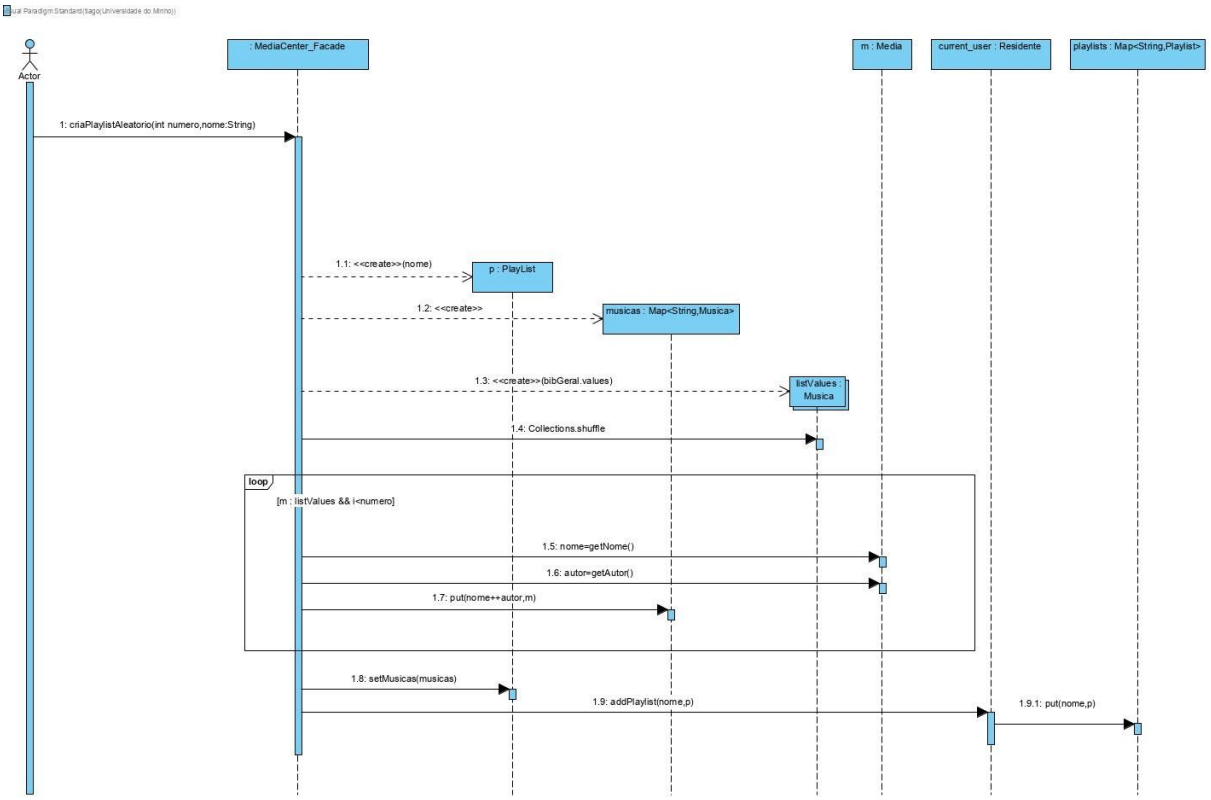


Fig 21-Diagrama de sequência da operação *criaPlaylistAleatoria*

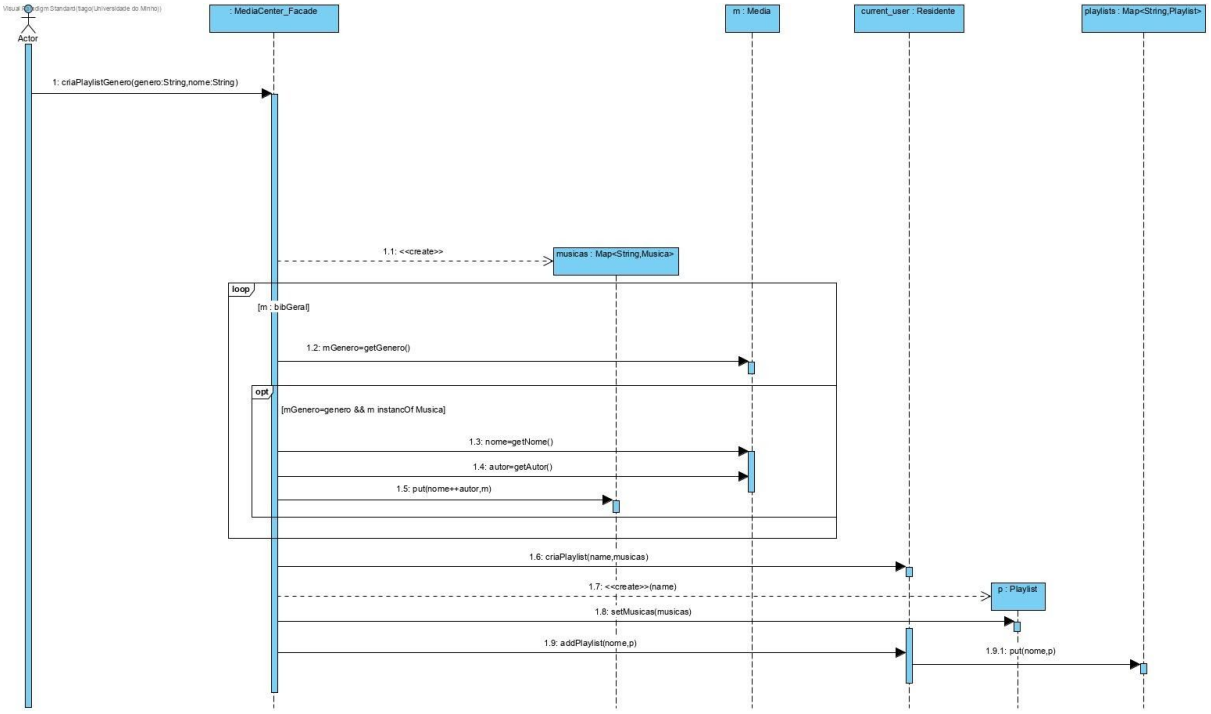


Fig 22-Diagrama de sequência da operação *criaPlaylistnome*

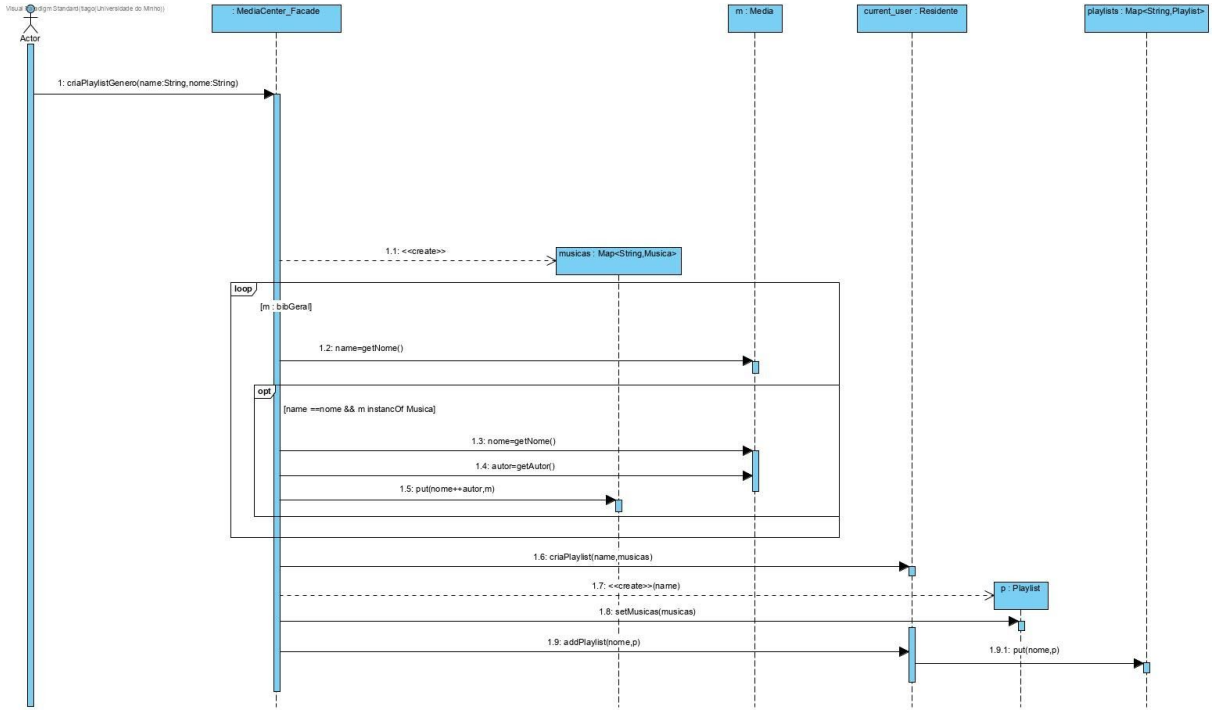


Fig 23-Diagrama de sequência da operação *criaPlaylistGenero*

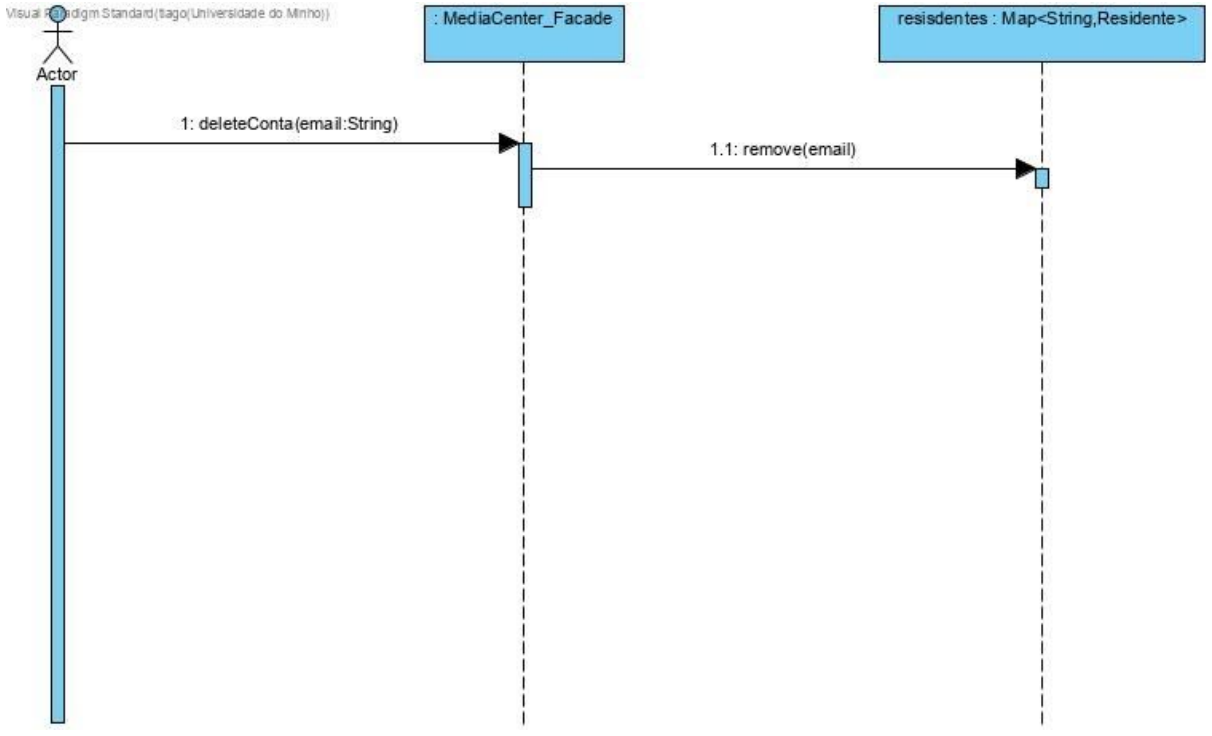


Fig 24-Diagrama de sequência da operação *deleteConta*

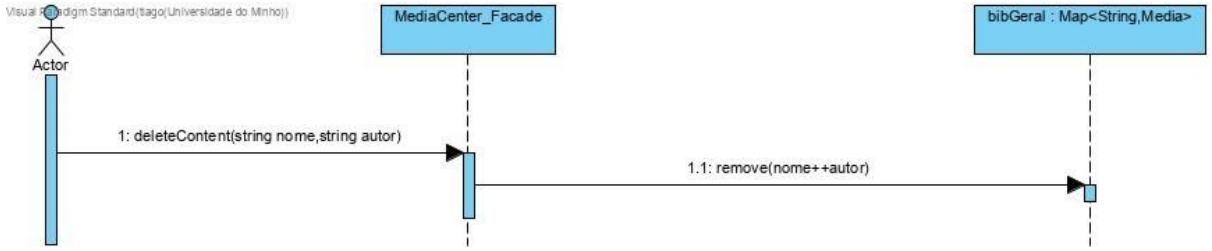


Fig 25-Diagrama de sequência da operação deleteContent

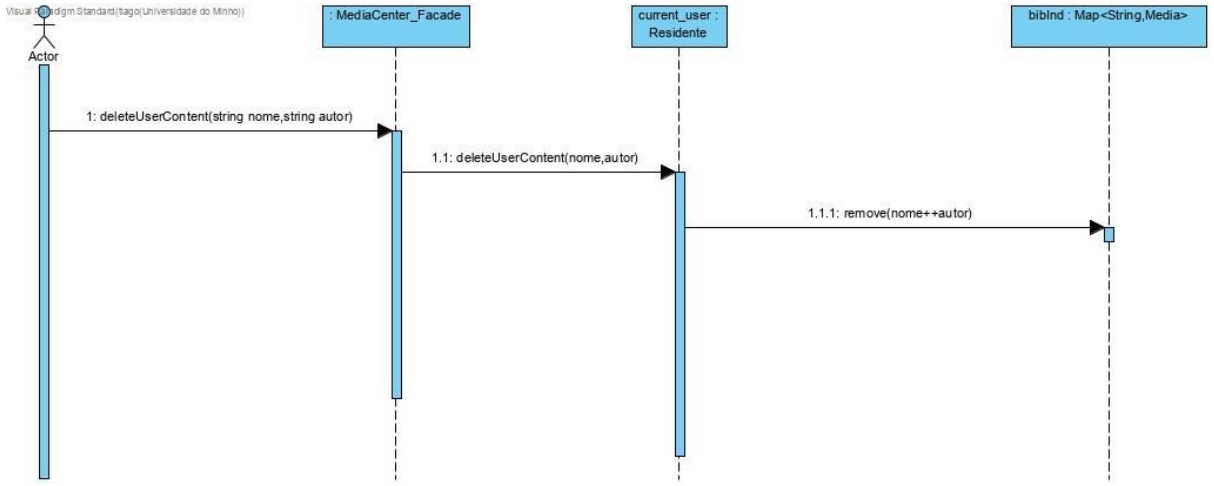


Fig 26-Diagrama de sequência da operação deleteUserContent

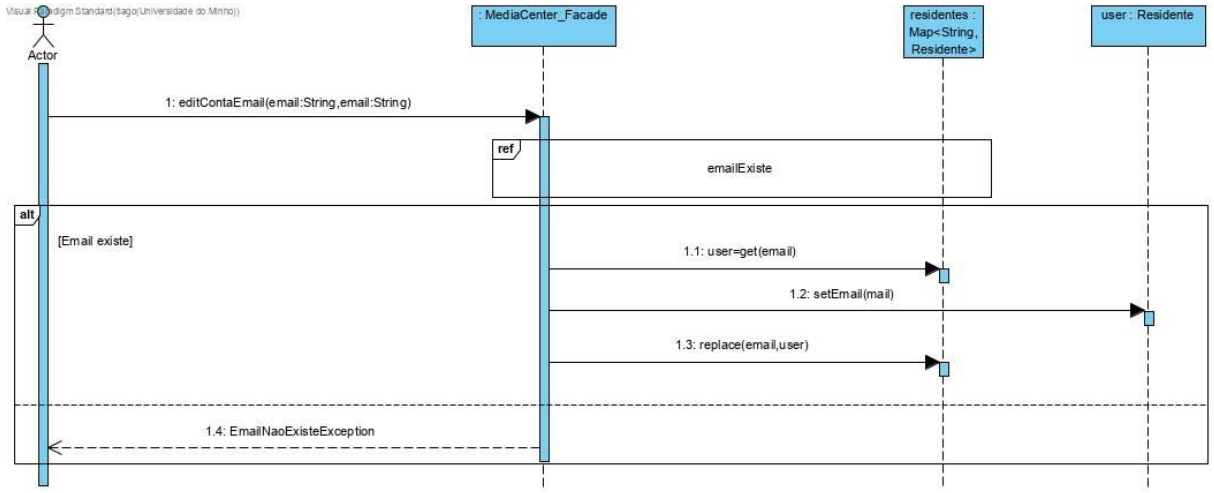


Fig 27-Diagrama de sequência da operação editContaEmail

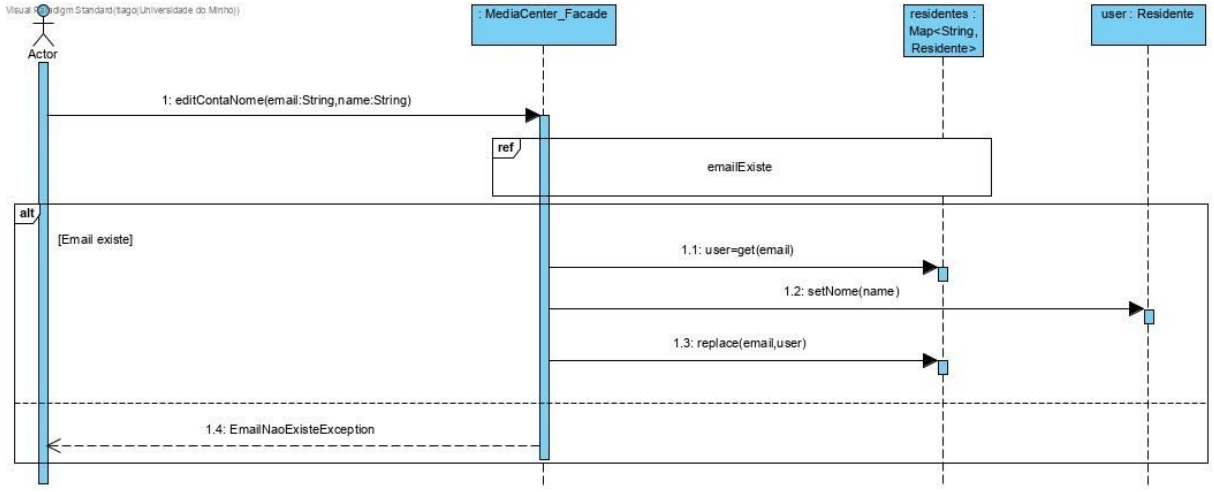


Fig 28-Diagrama de sequência da operação editContaNome

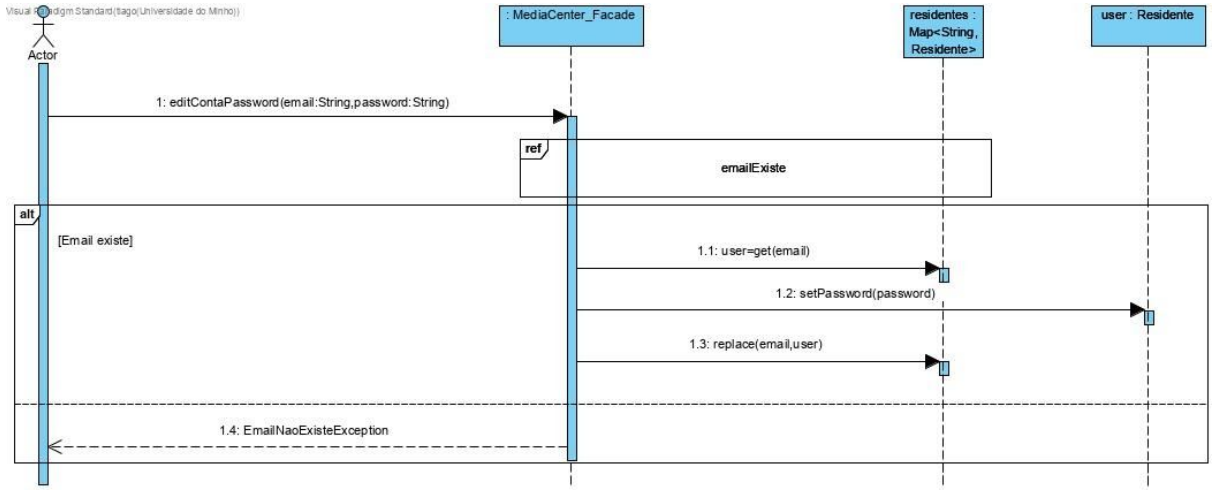


Fig 28-Diagrama de sequência da operação editContaPassword

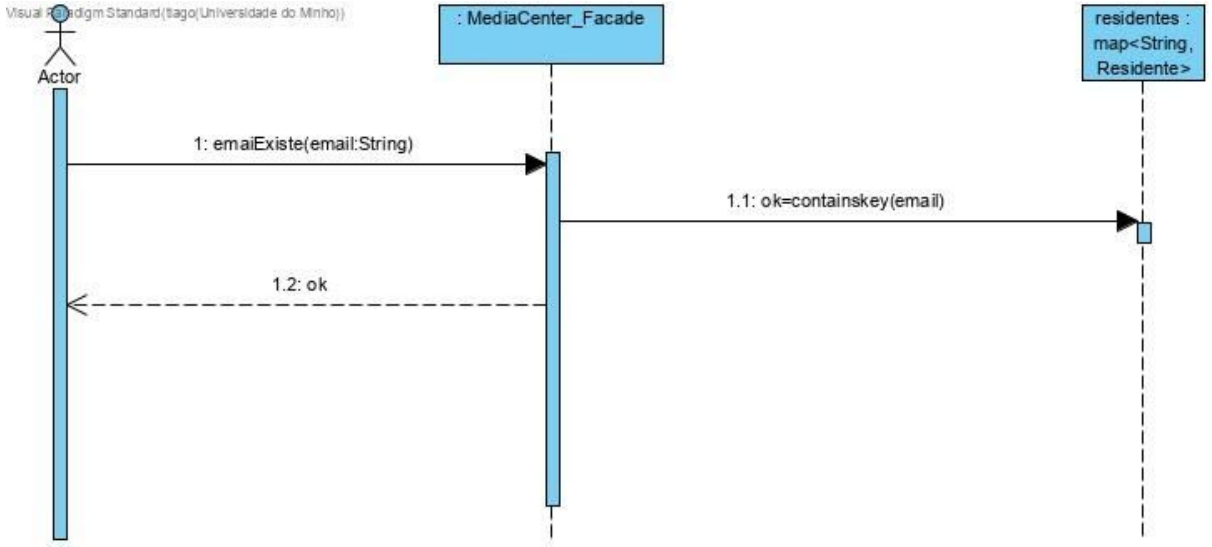


Fig 29-Diagrama de sequência da operação emailExiste

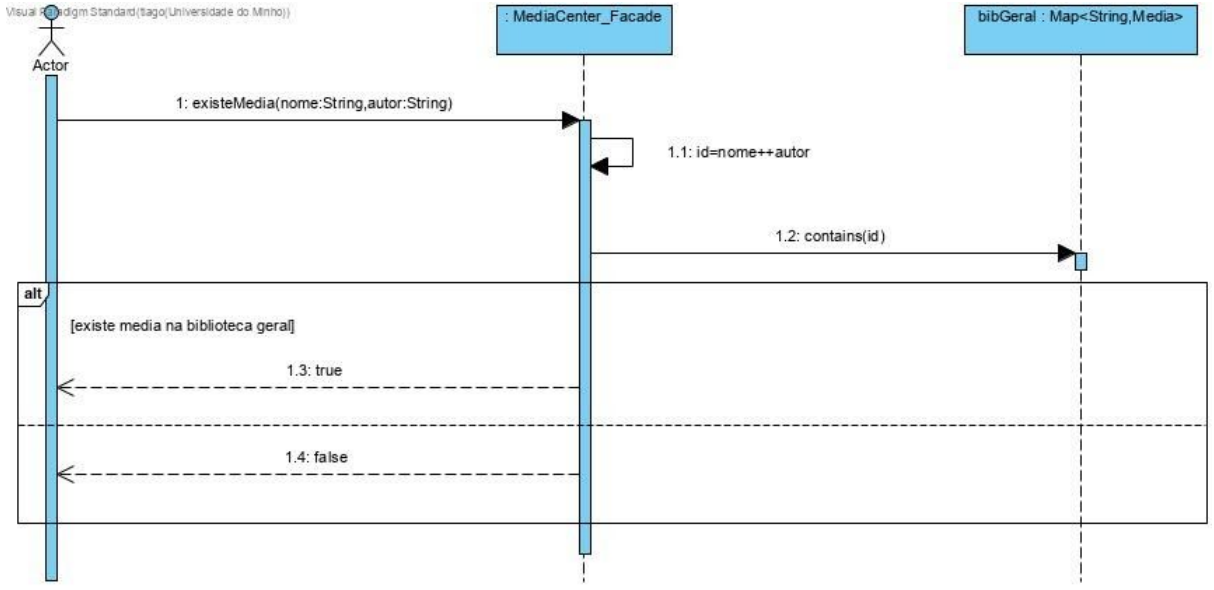


Fig 30-Diagrama de sequência da operação existeMedia

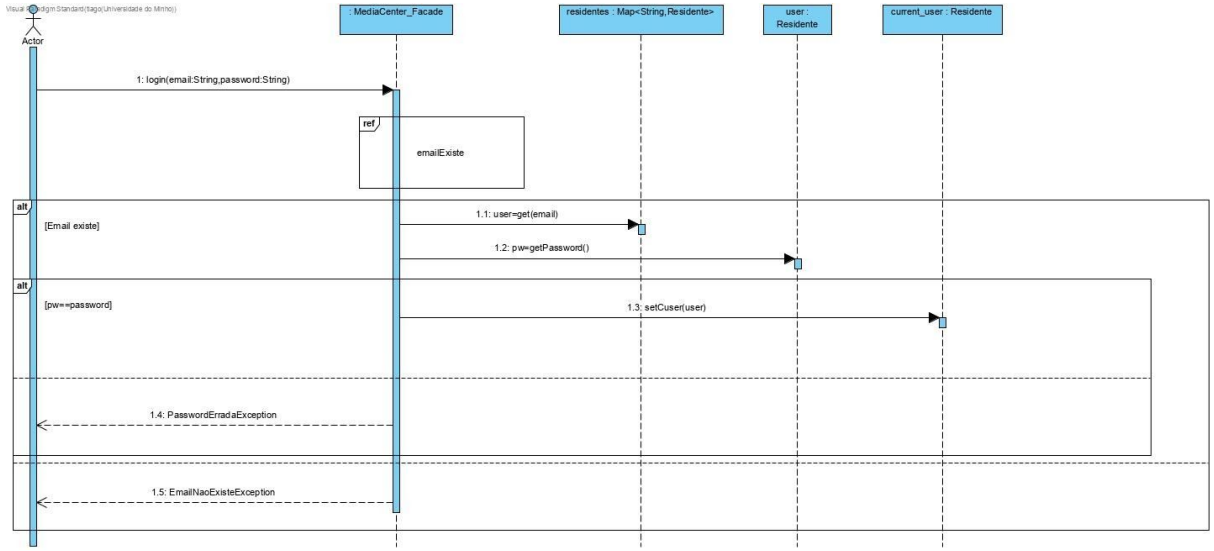


Fig 31-Diagrama de sequência da operação login

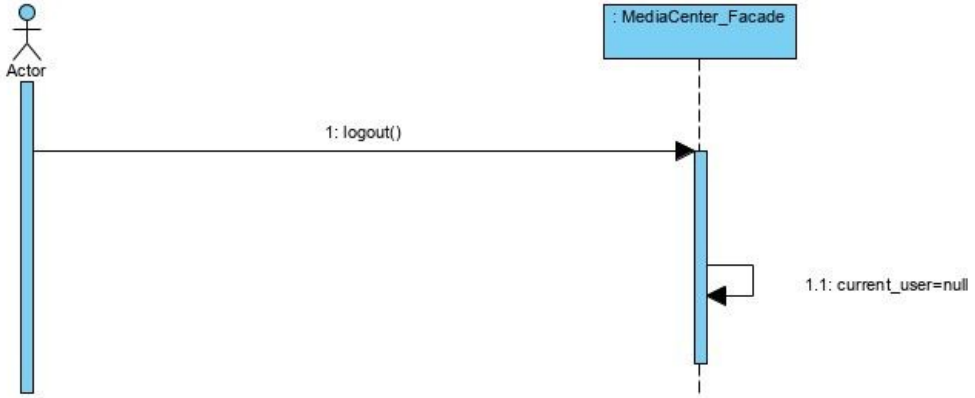


Fig 32-Diagrama de sequência da operação logout

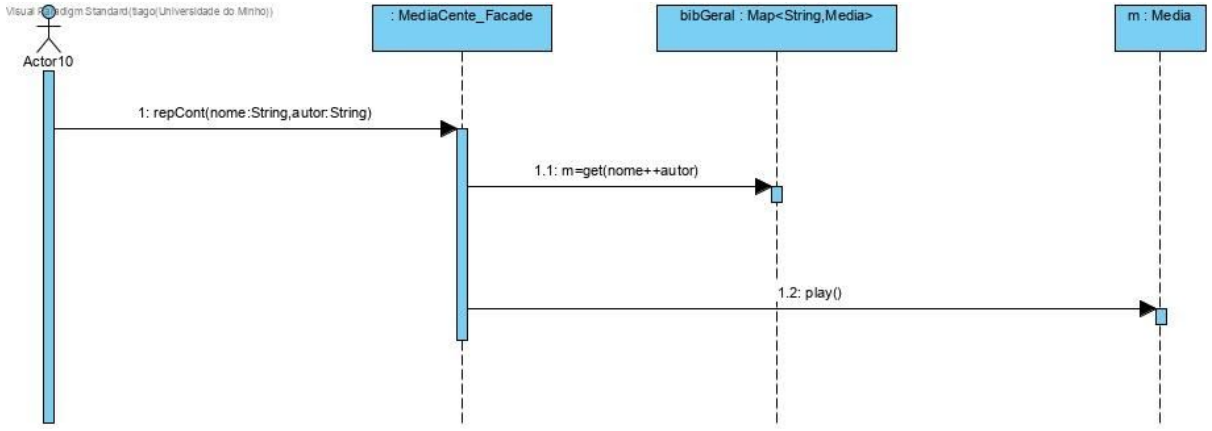


Fig 32-Diagrama de sequência da operação repCont

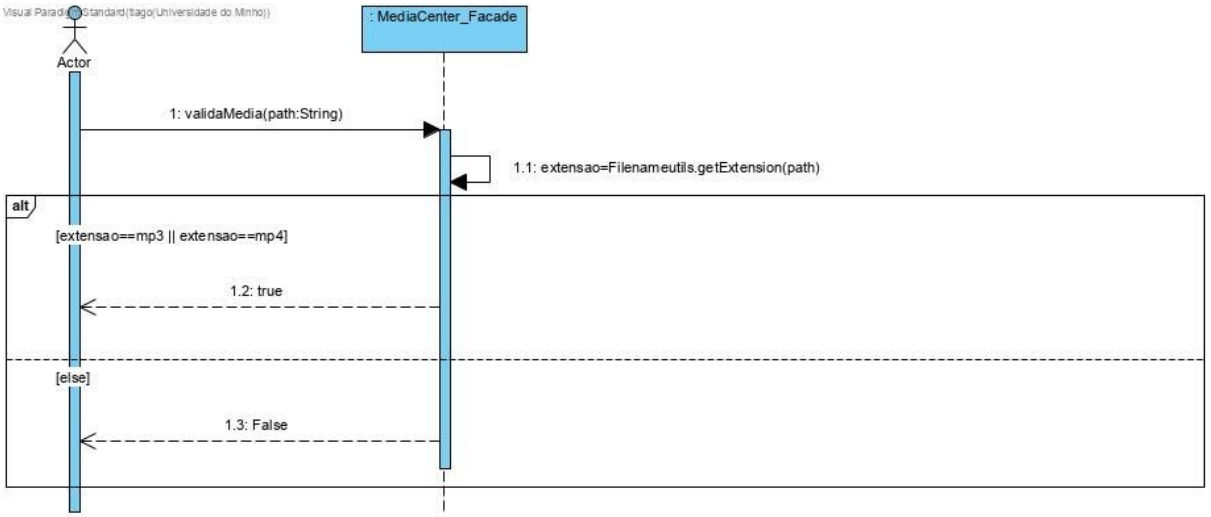
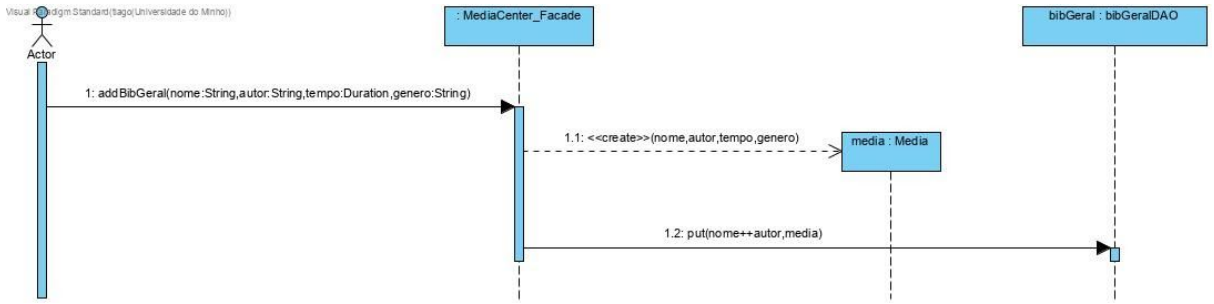
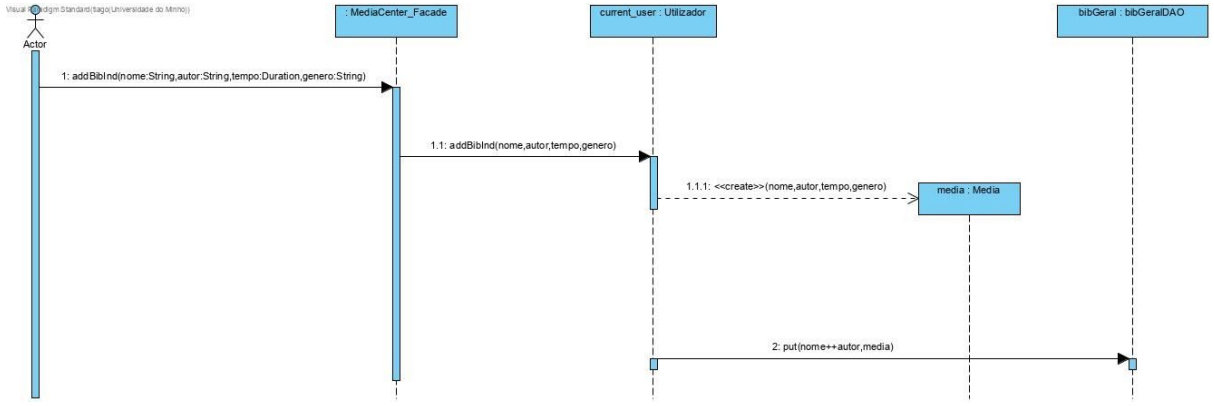
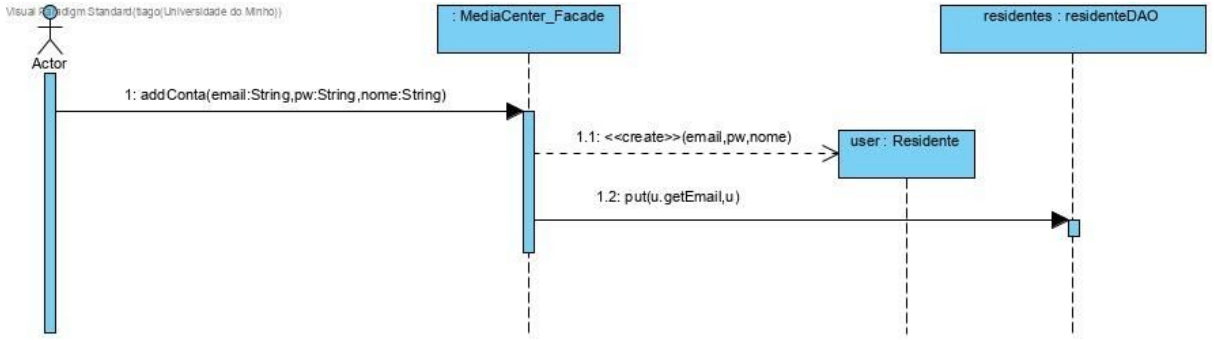


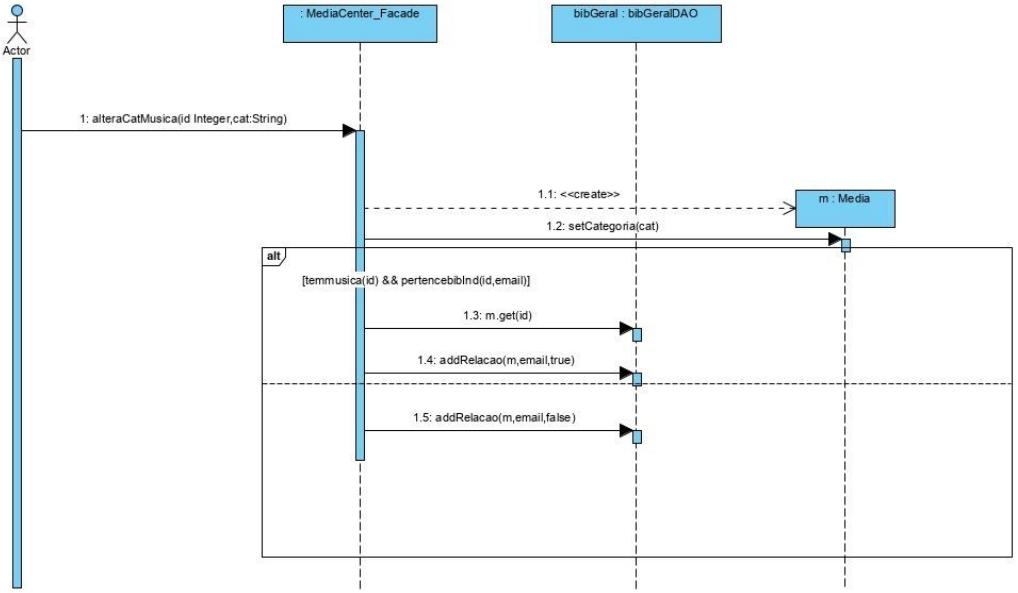
Fig 33-Diagrama de sequência da operação validaMedia

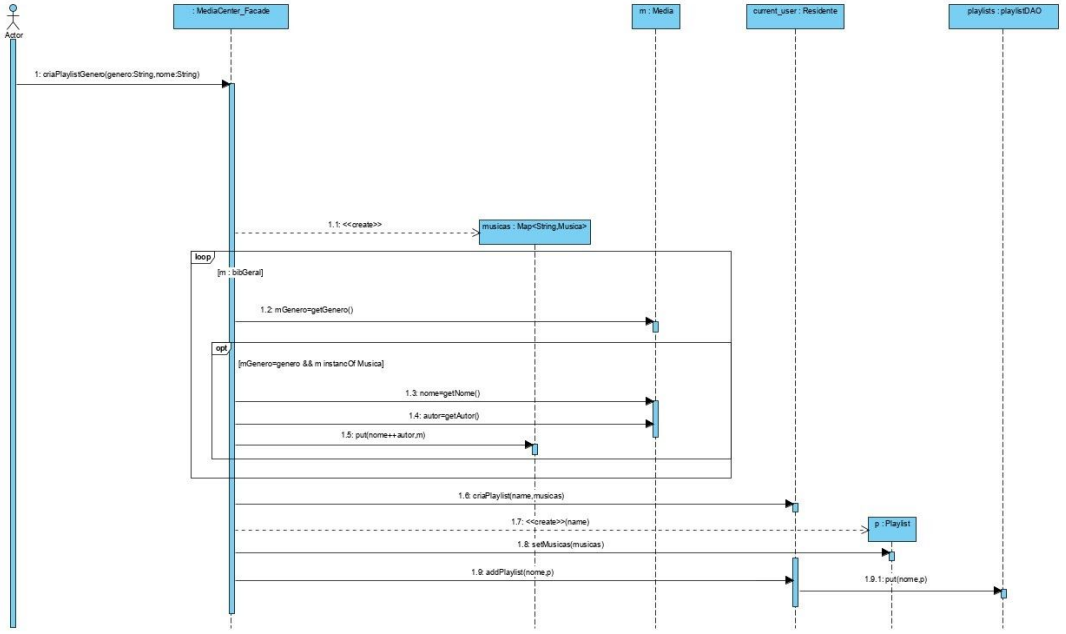
Diagramas com DAOs

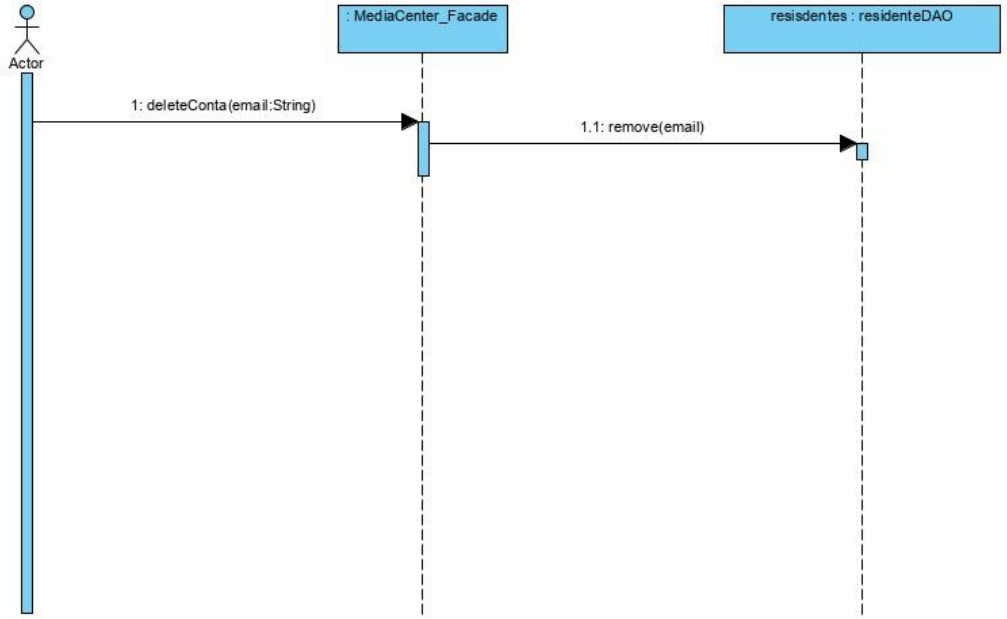


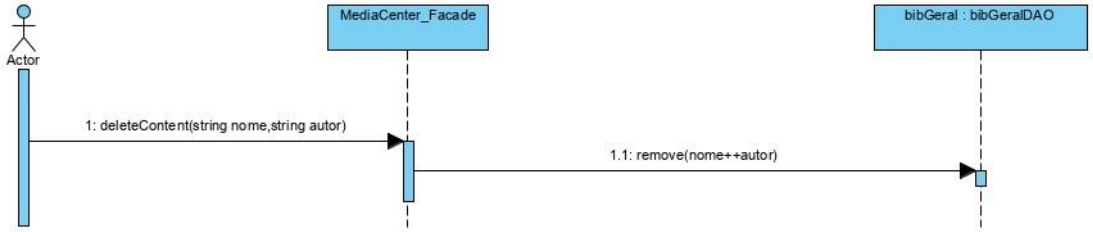


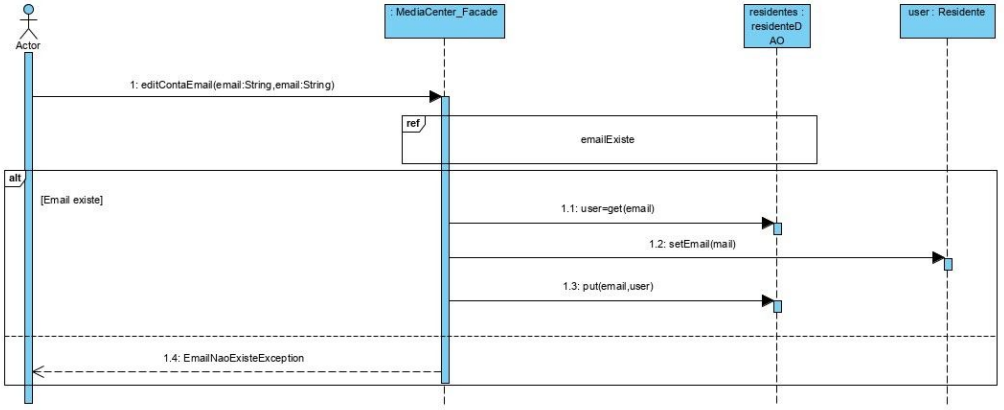


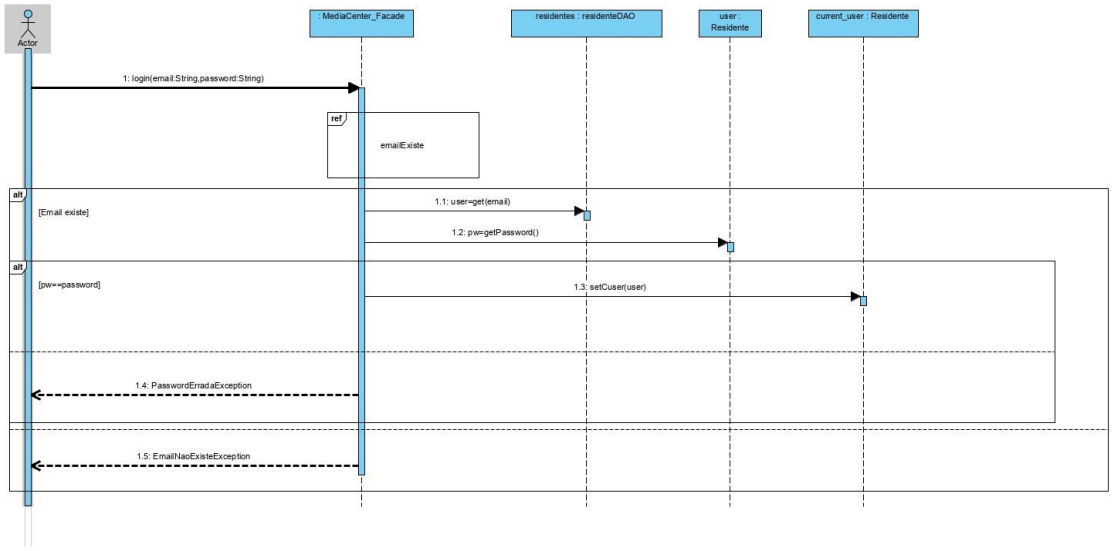

Actor
: MediaCenter_Facade
bibGeral : bibGeralDAO
1: alteraCatMusica(id Integer,cat:String)
1.1: <<create>>
m : Media
1.2: setCategoria(cat)
alt
[temmusica(id) && pertencebibInd(id,email)]
1.3: m.get(id)
1.4: addRelacao(m,email,true)
1.5: addRelacao(m,email,false)

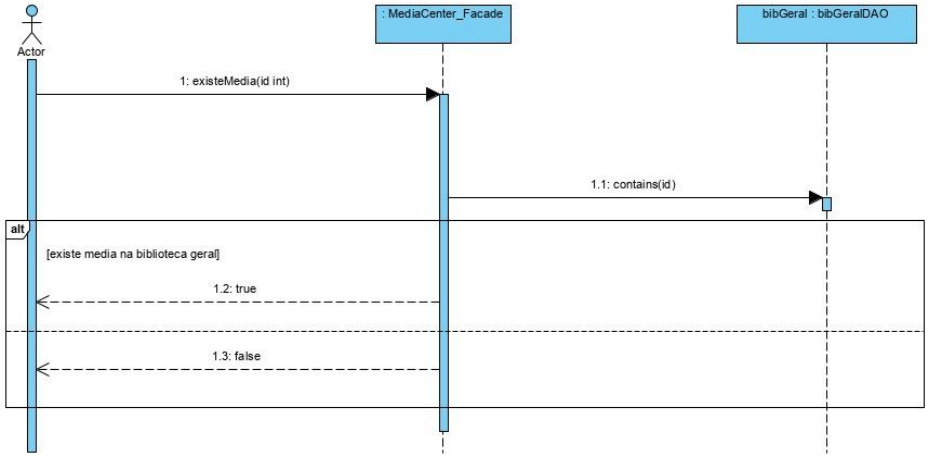

Actor
: MediaCenter_Facade
m : Media
current_user : Residente
playlists : playlistDAO
1: criaPlaylistGenero(genero:String,nome:String)
1.1: <<create>>
musicas : Map<String,Musica>
loop
[m : bibGeral]
1.2: mGenero=getGenero()
opt
[mGenero=genero && m instanceOf Musica]
1.3: nome=getNome()
1.4: autor=getAutor()
1.5: put(nome++autor,m)
1.6: criaPlaylist(name,musicas)
1.7: <<create>>(name)
p : Playlist
1.8: setMusicas(musicas)
1.9: addPlaylist(nome,p)
1.9.1: put(nome,p)

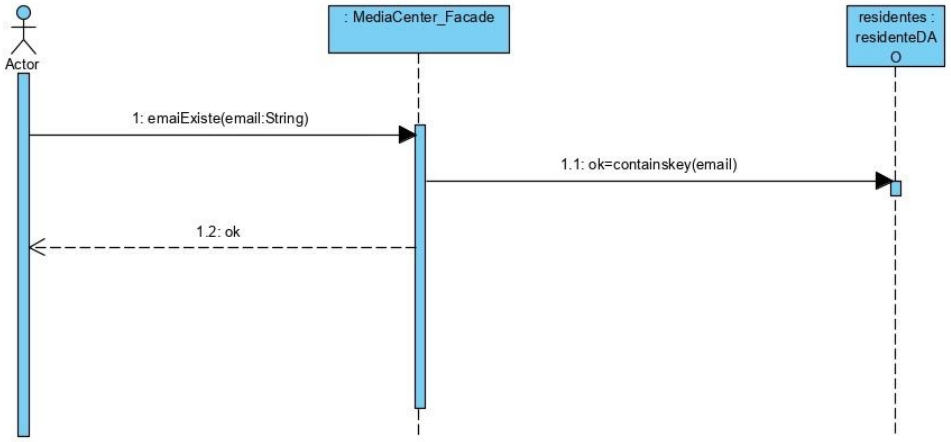

Actor
: MediaCenter_Facade
resisdentes : residenteDAO
1: deleteConta(email:String)
1.1: remove(email)

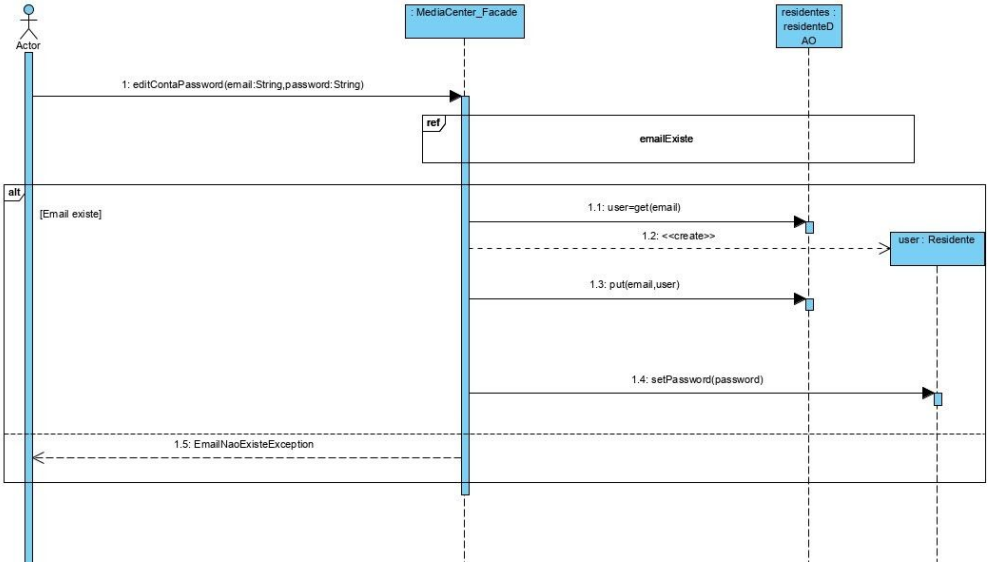

Actor
MediaCenter_Facade
bibGeral : bibGeralDAO
1: deleteContent(string nome,string autor)
1.1: remove(nome++autor)


Actor
: MediaCenter_Facade
residentes : residenteDAO
user : Residente
1: editContaEmail(email:String,email:String)
ref
emailExiste
alt
[Email existe]
1.1: user=get(email)
1.2: setEmail(mail)
1.3: put(email,user)
1.4: EmailNaoExisteException


Actor
: MediaCenter_Facade
residentes : residenteDAO
user : Residente
current_user : Residente
1: login(email:String,password:String)
ref
emailExiste
alt
[Email existe]
1.1: user=get(email)
1.2: pw=getPassword()
alt
[pw==password]
1.3: setCuser(user)
1.4: PasswordErradaException
1.5: EmailNaoExisteException


Actor
: MediaCenter_Facade
bibGeral : bibGeralDAO
1: existeMedia(id int)
1.1: contains(id)
alt
[existe media na biblioteca geral]
1.2: true
1.3: false


Actor
: MediaCenter_Facade
residentes : residenteDAO
1: emaiExiste(email:String)
1.1: ok=containskey(email)
1.2: ok


Actor
: MediaCenter_Facade
residentes : residenteDAO
1: editContaPassword(email:String,password:String)
ref
emailExiste
alt
[Email existe]
1.1: user=get(email)
1.2: <<create>>
user : Residente
1.3: put(email,user)
1.4: setPassword(password)
1.5: EmailNaoExisteException

3 - Implementação

1. Diagrama de Classes com DAO's

De modo a tornar os dados persistentes, tivemos de pensar numa forma de relacionar a base dados e a lógica de negócio. Como o paradigma orientado a objetos e paradigma relacional não são diretamente compatíveis, foi necessário estabelecer um mapeamento entre os dois, assim foi necessário definir tabelas (derivar tabelas a partir de objetos(explicação introdução).

Uma das principais decisões tomadas foram quanto às tabelas da base de dados:

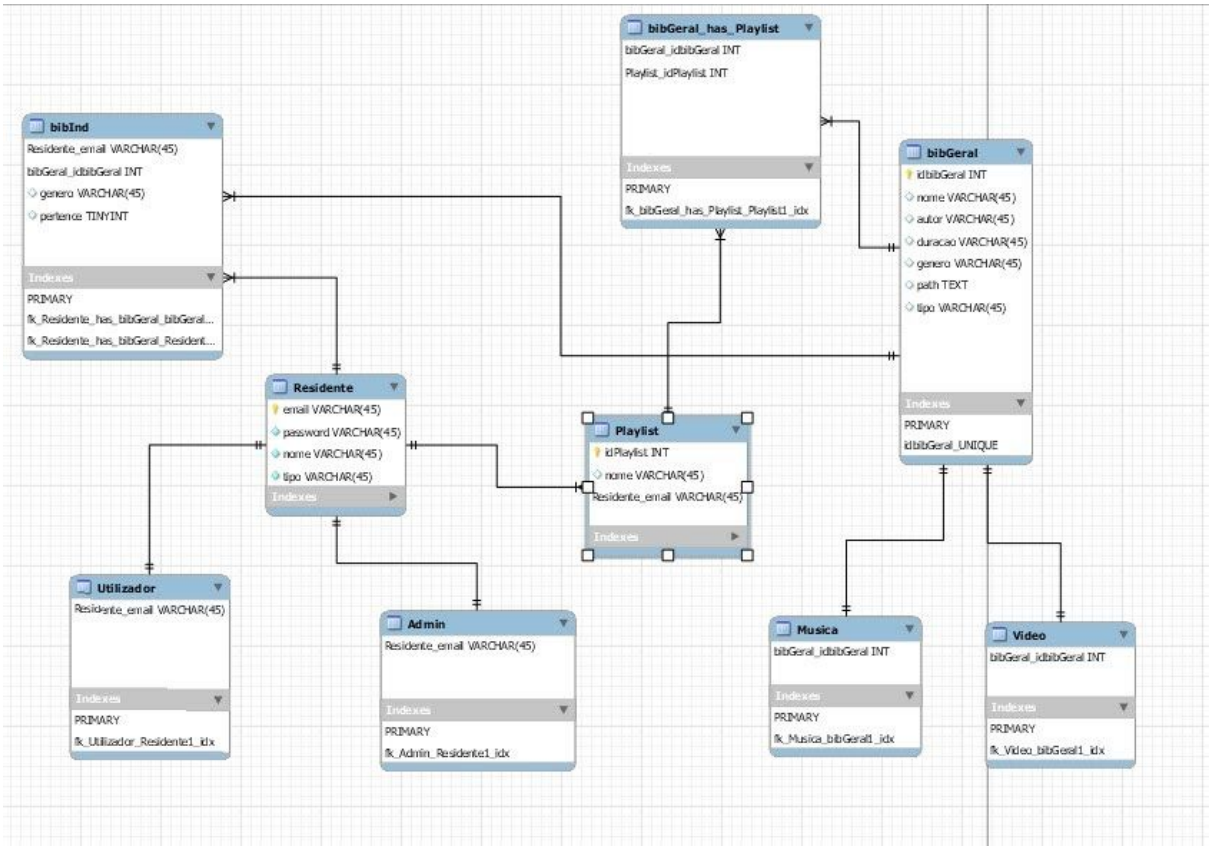


Fig 35-Modelo lógico base dados

Com o auxílio do modelo de domínio e diagrama de classes decidimos criar uma tabela residente com chave primária - o email (único), assim como uma tabela *bibGeral* onde são armazenadas todas as *medias* com chave primária - um id (poderiam existir músicas com mesmo nome e autor diferente e vice-versa). Para além disso, construiu-se uma tabela playlist com chave primária - um id - e uma foreignkey - o email do residente (associação um para muitos). Criamos as tabelas utilizador, admin, música, vídeo e devido à hierarquia de classes,surgiu também a tabela *bibInd* que relaciona músicas com residentes. Graças a esta, poderemos saber se se deu *upload* de uma música ou se alterou a categoria. Esta tabela surge de uma relação muitos para muitos bem como a tabela *bibGeral_has_Playlist* que relaciona as músicas que uma playlist tem.

O diagrama de classes com DAOs é uma adaptação do Diagrama de Classes anterior na qual se substituiu os *maps* da camada de negócio por classes do tipo *Data Access Object* (DAO). Estas classes permitem representar entidades a persistir mapeadas na base de dados, além do mais garante a separação entre a camada de negócio e a camada de dados, fazendo com que não dependam uma da outra.

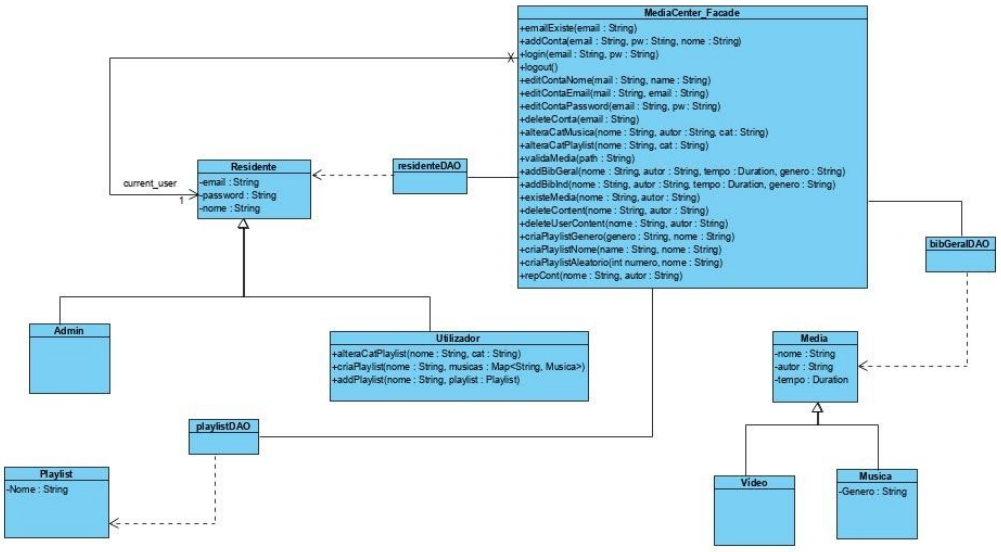


Fig 36-Diagrama de Classes com DAOs

2. Detalhes relevantes da implementação

Um dos detalhes que demoramos mais a fazer foi a de alterar a categoria de um conteúdo em que o utilizador alterava o conteúdo apenas para ele,mas conseguimos resolver isso graças à tabela da relação entre a bibGeral e o residente onde nessa relação ficavam informações como o género com que aquele utilizador definia a música bem como seus uploads.

A seleção da música é feita na Playlist, para reproduzir a música usamos um player externo, que no nosso caso é o VLC.

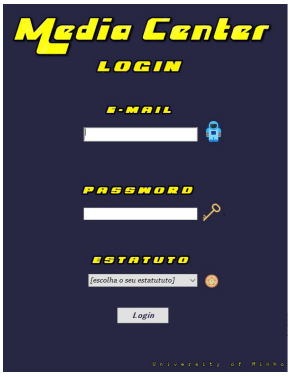
Inicialmente pensamos que a melhor maneira para desenvolver a base de dados para os residentes seria usando os email's de cada um, não usamos a estratégia dos id's auto incrementados da Biblioteca Geral e das Playlists, pois naturalmente os emails de cada residente seria único ao contrário dos outros. Mais tarde viemos a descobrir que não era a melhor ideia, pois teríamos de implementar a possibilidade de editar estes email's e não é uma boa prática alterar as primaries keys de uma tabela. No entanto, usamos a opção CASCADE para atualizar todas as chaves estrangeiras das outras tabelas chegando à solução problema.

3.Descrição da Interface

O grupo decidiu usar o programa *Netbeans* para elaborar a parte gráfica deste projeto. Usando a ferramenta *Swing* conseguiu-se proceder à criação de menus que, aliados aos DAOs, diagramas use case e à base de dados elevam este projeto a um patamar mais profissional.

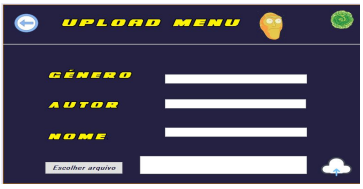
A interface tenta ser o mais user-friendly possível, apresentando menus devidamente identificados, de modo a que haja um correto funcionamento do programa e que todas a suas funcionalidades sejam convenientemente exploradas pelo utilizador. Pode-se dizer que existem três menus globais acessíveis: o de administrador, o de utilizador e o de convidado.

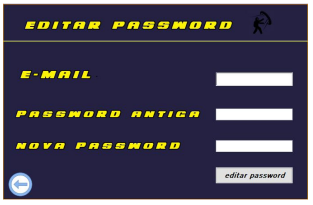
Um utilizador apenas vai ter acesso aos menus de utilizador, sendo que o mesmo se regista para um administrador e para um convidado. É de notar que o menu que o convidado tem acesso faz parte do conjunto de menus que pertencentes ao utilizador, ou seja, o convidado tem acesso restrito às funcionalidades dos menus de utilizador. Em anexo apresenta-se a grande maioria deles.



página inicial

menus de administrador, utilizador, convidado





conjunto de menus admin utilizador

4.Diagrama de instalação

Não tivemos tempo para meter o programa a correr em vários clientes,por isso o programa corre no próprio computador do utilizador.

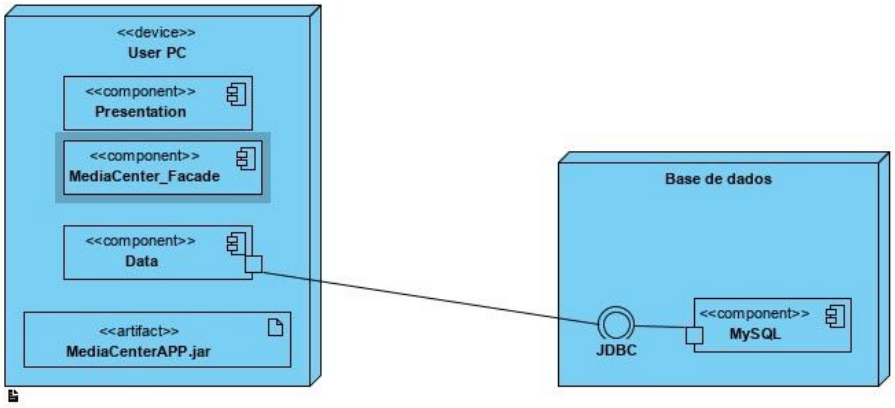


Fig 37-Diagrama de instalação

4 - Conclusões

Ao seguir este processo iterativo e de modelação notámos que a parte de código ao nível da lógica de negócio foi mais rápida e produtiva, sem ter de andar a reescrever código - quer seja por falta de estruturas ou erros de raciocínio. Isto deveu-se ao facto da modelação ter permitido compreender melhor o enunciado e termos tomado as principais decisões (estruturas a implementar) antes de codificar. Nem toda a modelação foi bem concebida desde alguns use cases mal estruturados, e algumas entidades sem sentido no modelo de domínio,bem como nos modelos de diagramas de sequência de operações em que alguma keys dos maps se encontram erradas,podemos constatar isso na realização do modelo de Classes DAO,que apenas foi feitos após os diagramas de sequência .A nossa perceção disto também deve-se ao facto de ser um processo iterativo e das aulas terem tido um papel fulcral na interpretação de algum tipo de informação .Este processo também nos permitiu perceber melhor acerca das arquiteturas usadas (arquitetura MVC), e como o software tem uma mais fácil manutenção, caso fosse preciso adicionar mais funcionalidades não seria preciso adicionar muito código. As estruturas já se encontram estruturadas para isso, o uso controlado de dependências permite também uma maior produtividade, permitindo que várias pessoas possam estar a trabalhar em partes diferentes (View,Data,Business). Caso uma parte seja reestruturada as outras não precisam de mudar