UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

SERVIÇO DE AUXÍLIO À ORGANIZAÇÃO DE EVENTOS

SCC0240 BASE DE DADOS
Prof.ª Dr.ª Elaine Parros M. de Sousa

Henrique de S. Q. dos Santos – 10819029
Leonardo R. Luiz – 10851691
Paulo H. da Silva – 10734515
Witor M. A. de Oliveira – 10692190

SÃO CARLOS
2020

Sumário

1 Descrição do Problema e dos Requisitos de Dados 3

2 Consultas ao banco de dados 5

3 Modelo entidade-relacionamento 6

3.1. Alterações feitas 6

4 Projeto lógico (Mapeamento MER-Relacional) 8

4.1. Decisões sobre o Mapeamento 8

4.1.1. Agregações 8

4.1.2. Generalizações 8

4.1.3. Atributos multivalorados 10

4.1.4. Atributos derivados 11

4.1.5. Restrições de Integridade 11

4.1.6. Restrições de Relação 13

4.1.7. Alterações feitas 15

5 A base de dados 16

5.1. Normalização 16

5.2. Descrição 17

5.3. Desenvolvimento da base 17

5.4. Desenvolvimento do sistema 17

6. Conclusão 22

1. Descrição do Problema e dos Requisitos de Dados

Visando facilitar a interação entre usuários que buscam organizarem festas ou eventos diversos e pessoas que buscam destino para componentes úteis a este fim, essa ferramenta foi criada com o intuito de estabelecer um compartilhamento rápido e fácil desses recursos disponíveis.

Atualmente, o planejamento e organização de eventos se dá através de empresas que se responsabilizam por este fim. Infelizmente, por geralmente envolver diversas outras empresas, mão de obra, produtos e ferramentas, o custo para se organizar um evento acaba se tornando mais alto do que o esperado. O propósito dessa aplicação é permitir que o usuário a fim de realizar um evento possa visualizar as melhores ofertas de produtos para essa eventualidade, desde o local aos produtos de decoração, infraestrutura etc.

O usuário que desejar divulgar algum tipo de componente que pode ser usado em eventos, deve se cadastrar no sistema informando o próprio **CPF**, **nome**, **telefones** para contato, **senha** e ***e-mail***, sendo que estas duas últimas informações serão usadas para a realização do *login* no sistema. Além desses dados armazenados, será reservado um campo o qual irá conter a **avaliação** geral desse anunciante, isto é, a nota média calculada mediante todas as **notas** classificatórias dadas pelos usuários que usufruírem de seus componentes anunciados, bem como os **comentários** deixados por eles. Após realizar o cadastro, esse usuário terá acesso ao **painel do anunciante**, onde será possível realizar as seguintes operações: inserção de novos anúncios, edição e remoção dos já existentes, bem como confirmação, ou não, de pedidos feitos (isso será explicado posteriormente).

Ao ofertar um novo **componente**, o usuário deve inserir o **preço** que deseja cobrar por esse item, bem como o **tipo**. Para facilitar a interpretação da oferta, o sistema permite que seja feito apenas um anúncio por tipo de cada vez (por exemplo, você não pode tentar inserir, ao mesmo tempo, ofertas de produto do tipo X **e** Y, apenas do tipo X **ou** do tipo Y). As ofertas de componentes são divididas entre três **tipos**, a serem detalhados nos próximos parágrafos.

Na categoria de **serviços**, o usuário deseja oferecer serviços gerais. Dentre eles, o nosso sistema prioriza oferecer serviços de alguns **tipos**: **alimentação** (onde deverá ser informado sua **quantidade** e **tipo** de alimento – se é bebida ou comida, por exemplo –, **transporte** (serão solicitadas as informações sobre **capacidade**, **quantidade** e **tipo** de transporte – se a oferta é sobre um ônibus ou micro-ônibus, por exemplo), **segurança** (e o(s) **turno**(s) o qual aceita trabalhar), **limpeza** e **entretenimento**, este dividido entre **brinquedos** e **apresentações** (subdividido nas **categorias** de apresentações de **mágica**, **teatro**, **dança** e **música** que, por sua vez, é repartido em **banda**, **DJ**, **cantor** e **dançarino**). O sistema entende que outros serviços, bem como outros tipos de divisões, possam ser úteis e, portanto, permite que outras variedades sejam ofertadas. Vale ressaltar que para cada serviço oferecido, uma **ID** única é automaticamente gerada pelo sistema.

Já no anúncio de um **local**, o usuário possui um local ocioso, por exemplo, e deseja emprestá-lo/alugá-lo. Para tal tipo de oferta, o anunciante deverá informar a **tensão elétrica** do local, **metragem**, **capacidade** de pessoas, uma ou mais **fotos**, o **endereço** do imóvel (**composto por rua**, **número**, **bairro** e **CEP**), que será utilizado para realizar a busca por esse tipo de componente, e qual o **tipo** do local a ser oferecido. O sistema trata, a princípio, de dois tipos de locais: **chácara** e **república**, onde deverá ser informado se possuem **garagem**, a **quantidade de banheiros e de quartos** e uma **descrição** sobre a existência, ou não, **de áreas comuns** (lavanderia, por exemplo). Pela mesma justificativa supracitada, permitimos que outros tipos sejam anunciados (um galpão, por exemplo).

Por fim, ao ofertar **produtos**, entende-se que o usuário possui produtos que possam ser úteis para alguns eventos. Ao optar por anunciar um ou mais produtos (anúncio de vários produtos de uma vez estará disponível apenas para produtos do mesmo tipo), o usuário deverá informar a **quantidade** que deseja anunciar, além da **categoria** do produto. Os produtos estão distribuídos entre **decorações**, que podem ser, por exemplo, **globo(s) espelhado(s)** (onde deverão ser informadas a **tensão** e **potência** do motor desse item) e **bexigas** (deverá ser informada a(s) **cor**(es) e a **forma** dela(s)), **áudio**, **iluminação** (deverão ser informadas a **tensão** e **potência**), categorizada em lâmpadas (dos **tipos LED**, **neon**, **fluorescente** – informando a cor para os 3 tipos citados – e **incandescente**, a priori) e **refletores** (o usuário deve informar a cor), e **infraestrutura**, fragmentado em alguns **tipos**: **montagem** (este dividido em **divisórias** e **ferramentas**), **mesas** e **assentos**, sendo que o **material** desses dois últimos itens deverá ser informado.

Após também efetuar o cadastro no sistema e inserir as mesmas informações pedidas para o anunciante (com exceção que, para este usuário, não haverá a criação do campo de avaliação), o contratante poderá fazer uma **solicitação**, com ou sem **detalhes** (como negociação de preço etc.), de um ou mais componentes ao mesmo tempo, isto é, ele pode solicitar N componentes de tipos diversos na mesma solicitação. Ao efetuar uma solicitação de componentes, entende-se que ele planeja criar um **evento** e, portanto, será solicitado, ao término da escolha dos componentes, a **duração do evento** (**composta** pela informação da **data de início e término** do evento) e o **nome** desse evento (que será utilizado como identificação única e chave de busca em conjunto com uma ID que será gerada automaticamente pelo sistema), que serão armazenados para facilitar que um anunciante se organize entre as solicitações de seus anúncios. A solicitação irá constar no painel de pedidos do contratante, onde será possível visualizar todos os pedidos feitos, cancelá-los e visualizar o status da(s) sua(s) solicitação(ões). A resposta à solicitação será retornada para o cliente que terá, então, a informação sobre a disponibilidade dos componentes naquela data. O sistema não possibilitará a comunicação entre cliente e anunciante. O **preço** do evento será calculado de acordo com os componentes solicitados, uma informação que estará disponível apenas para que o solicitante tenha noção dos gastos que terá, já que, como informado, não serão tratadas as operações de pagamento.

2. Consultas ao banco de dados

Para o **cadastro** (tanto de anunciante, quanto para contratante), será necessário consultar se existem usuários cadastrados com o mesmo endereço de email e CPF que estão sendo informados no ato. Se os dados forem válidos, uma inserção será realizada no banco de dados;

Para o **login** (tanto de anunciante, quanto para contratante), uma consulta será feita para verificar se o email informado existe e, se existe, se a senha informada corresponde com a cadastrada;

Na **inserção de um novo anúncio**, uma operação de inserção será feita na base de dados, de modo que o ID deste seja único e maior que o último ID de um anúncio inserido;

Deve ser possível realizar a recuperação dos anúncios de determinado tipo de componentes através da busca realizada pelo usuário. Essa busca será realizada em cima dos tipos de chaves para cada componente:

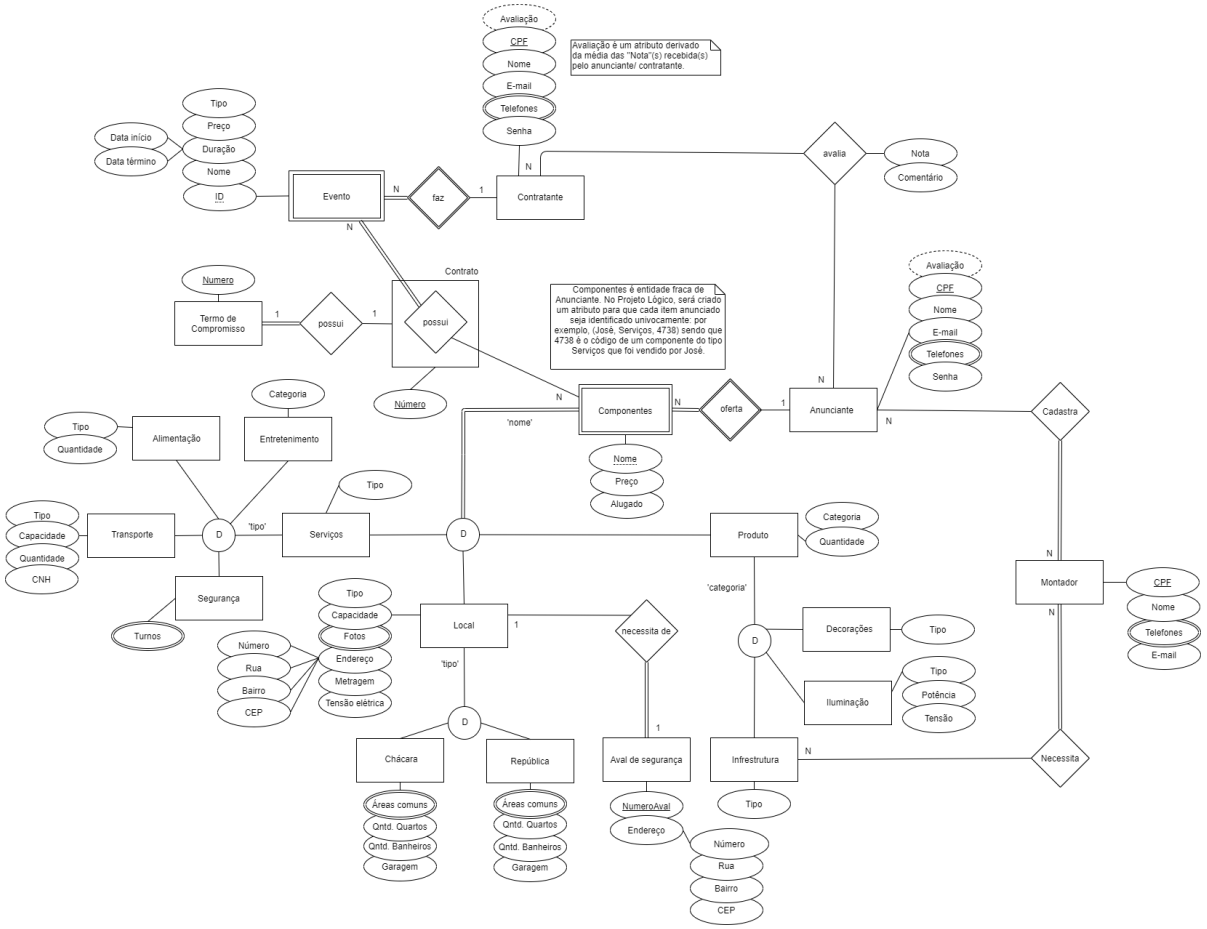
- Serviços: a chave é o ID do serviço inserido;
- Locais: a chave é o endereço do imóvel anunciado; e
- Produtos: a chave é o ID do item cadastrado.

Na **solicitação**, através do anunciante, de um componente, deverá ser inserido na tabela de solicitações o ID único (e maior que o último ID de um evento inserido) dessa solicitação, bem como o nome do evento, que serão utilizados como chave de busca para o evento em questão;

Para o **envio da solicitação** de um componente, deverá ser possível realizar a busca do CPF (chave) do anunciante do componente, para que seja possível enviar a confirmação para o painel dele. Além disso, deverá ser **possível atualizar o status da solicitação** (na tabela de solicitações), dada a confirmação ou não da disponibilidade do componente, de acordo com a resposta do anunciante;

Para o anunciante, deverá ser possível **recuperar as informações** acerca de um de seus anúncios (por vez), **realizar alterações no estado** do anúncio (bem como edições nas informações ou remoção do anúncio) e **atualizar a tabela com as novas informações**.

3. Modelo entidade-relacionamento



3.1. Alterações feitas

Após solicitações feitas visando o aumento da complexidade do modelo relacional, acrescentamos mais relações no diagrama e entidades no modelo. Abaixo, enumeramos as alterações feitas no modelo relacional:

1. Acrescentamos uma relação **avalia** entre as entidades Contratante e Anunciante, além de termos tornado o atributo **Avaliação** em derivado dessa relação (será uma média simples de todas as instâncias do novo atributo **Nota** na relação avalia). Vale ressaltar que, como é possível ver, criamos um **ciclo** entre essas duas entidades (anunciante -> avalia -> contratante e contratante -> avalia -> anunciante) que não pode ser quebrado, já que o atributo "avaliação" é dependente e bidirecional às duas partes, enquanto as mesmas partes também são necessárias para criação de um "Evento";

2. **Evento** não é mais uma agregação e, sim, uma entidade fraca de **Contratante**, pois consideramos que um evento só será cadastrado no sistema se existir uma instância de Contratante dona (entidade *Owner*) da instância de Evento. Além disso, Evento possui um relacionamento total com **Componentes** (já que, além de não existir uma instância de Evento sem um Contratante, não existirá essa mesma instância sem que haja o relacionamento com algum dos Componentes cadastrados no sistema) e, desse relacionamento, surgiu a

agregação **Contrato**, que se relaciona com **Termo de Compromisso**, instanciado, obrigatoriamente, toda vez que um Contrato também for criado.

3. A entidade **Componente** se tornou uma entidade fraca da entidade Anunciante. Por tal, o atributo **Nome** se tornou chave fraca da entidade Componente;

4. Anunciante possui uma nova relação **"cadastra"** com participação total com uma nova entidade **Montador.** Este possui uma nova relação **"necessita"** de participação total com Infraestrutura, para realizar a montagem da mesma solicitada pelo contratante. A entidade Montador possui os mesmos atributos que Anunciante e Contratante possuem (com exceção do atributo Avaliação), mas não os mesmos dados. A inserção dos relacionamentos **"necessita de"** e **"cadastra"** por causa da entidade montador acabou por gerar um ciclo (Anunciante -> Cadastra -> Montador -> Necessita de -> Infraestrutura -> Produto -> Componente -> Anuncia), o qual não conseguimos desfazer, pois o Anunciante cadastra tanto a Infraestrutura quanto o Montador, enquanto o montador só existe na dependência de alguém que o cadastre e uma infraestrutura a ser construída.

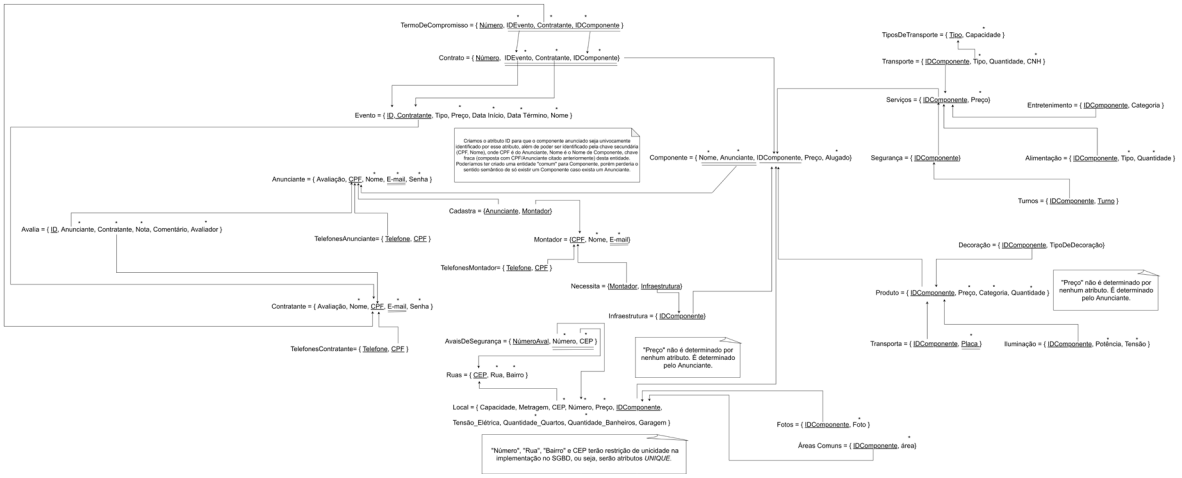
5. Local possui um novo relacionamento **"necessita de"** com participação total com uma nova entidade intitulada **Aval de segurança**, que possui os atributos NumeroAval (chave primária) e Endereço (chave secundária), o mesmo da entidade genérica "Local"**;**

6. Removemos algumas generalizações que consideramos desnecessárias para o entendimento do escopo do projeto;

7. Adicionado atributo "Alugado" para a entidade Componente, que indicará se a relação entre o Componente e o Contratante é de aluguel ou compra.

Além disso, efetuamos alterações na descrição do projeto de forma a tornar mais simples a especificação de uma funcionalidade do sistema, já que o foco era apenas a modelagem do banco. Por fim, vale ressaltar que as principais alterações feitas no modelo relacional foram visando explorar dois dos três casos de mapeamento relacional quando estamos lidando com o projeto lógico e, por isso, as novas relações foram focadas nas generalizações (conceito bastante presente no nosso modelo).

4. Projeto lógico (Mapeamento MER-Relacional)



4.1. Decisões sobre o Mapeamento

4.1.1. Agregações

- **Agregação "Contrato"** entre Evento e Componentes, entidade fraca de Anunciante

    - Solução Adotada:

        - Optamos por mapear a agregação em uma tabela separada de Evento e Componentes (entidades a quais se relaciona), sem mapear a relação "Possui" englobada por esta.

          A vantagem de se possuir essa tabela é que, para o foco da aplicação, fica mais fácil identificar um Contrato e, consequentemente, a existência, ou não, de Componentes relacionados à um Evento específico.

    - Alternativa:

A alternativa seria mapear a tabela que representa a relação. Essa solução é ruim para o foco do projeto já que a agregação Contrato é mais importante, semanticamente, do que a relação "Possui".

4.1.2. Generalizações

- Generalização da **entidade Componentes**

    - Solução Adotada:

Optamos por mapear tanto "Componente" (entidade genérica) quanto "Serviço", "Local" e "Produto" (entidades

específicas) em relações diferentes. A justificativa segue em problemas das alternativas citadas abaixo.

- Alternativa:

  - Uma alternativa seria mapear apenas “Componente”, o que acarretaria problemas de armazenamento, visto que teríamos que trazer todos os atributos em uma única tabela e isto geraria informações nulas em extrema abundância e acarretaria problemas relacionais visto que “Local” faz relação com outra entidade. Mapeando todas as generalizações em apenas uma tabela, cada consulta iria trabalhar com valores nulos em atributos específicos para tipos de Componente que não aquele que estaríamos buscando, bem como contaria com muito mais valores por consulta do que o caso adotado (mapeamento em várias tabelas), o que acarretaria custo adicional de busca. Ainda, por conter todos os valores numa tabela só, o tempo e custo de busca por tuplas seria maior.

  - Outra seria mapear apenas “Serviço”, “Local” e “Produto” (entidades específicas), o que geraria incoerência semântica no modelo, visto que “Componente” faz relação com “Anunciante”.

- Generalização das **entidades Serviço e Produto**

  - Solução Adotada:
    - Optamos por mapear, tanto as entidades genéricas, quanto as entidades específicas.

      A vantagem dessa solução é que acabamos por armazenar apenas as informações necessárias para cada entidade específica, bem como nos referenciamos a cada uma delas quando necessário (uma consulta em um conjunto específico, por exemplo). Além disso, julgamos que as entidades específicas possuem muitos atributos específicos que, se juntos na mesma tabela, acarretariam grande quantidade de valores nulos. No caso da entidade “Infraestrutura”, ela participa de relações com outro conjunto de entidade, mais um motivo para desmembrar as suas generalizações em tabelas.

  - Alternativa:

    A alternativa seria criar somente as tabelas específicas ou somente as tabelas genéricas. Neste, como dito anteriormente, haveria muitos valores nulos. Além disso, os relacionamentos perderiam o sentido semântico no projeto.

- Generalização **da entidade Local**

    - Solução Adotada:

        - Optamos por criar apenas a entidade genérica.

          Optamos por essa solução pois, como os atributos específicos de ambas as entidades específicas são iguais, não haveria muitas ocorrências de valores nulos ocasionados por estarmos instanciando um tipo específico ao invés de outro. Por isso, criar apenas uma tabela ao invés de 2 (mapeamento de apenas as entidades específicas) ou 3 (mapeamento da entidade específicas e genérica). E, por fim, a entidade genérica possui um relacionamento com outra entidade e, por isso, se faz necessário mantê-la.

    - Alternativa:

A alternativa, como mencionado anteriormente, era criar apenas as tabelas específicas (inviável, pois a genérica participa em relacionamentos com outras entidades) ou criarmos as três (genérica e as 2 específicas), o que seria desnecessário já que as entidades específicas possuem os mesmos atributos e não participam de nenhuma relação.

4.1.3. Atributos multivalorados

- Solução Adotada:

    - Optamos por mapear **todos** os atributos multivalorados do projeto relacional (“Telefones” das entidades Anunciante, Contratante e Montador, “Turnos” de Segurança, “Fotos” de Local e “Áreas comuns” das entidades generalizadas de Local).

      Essa solução foi adotada pois, em todos os casos, os atributos podem receber mais de um valor por chave. Ou seja, para uma mesma chave, esses atributos podem receber vários valores pois não há limite (exceto em “Turnos” de Segurança, onde este pode receber apenas um máximo de quatro valores que são os valores Matutino, Diurno, Vespertino e Noturno) de valores que podem ser atribuídos para uma mesma chave.

- Alternativa
    - A única alternativa era limitar a quantidade de atributos que poderiam ser inseridos por chave e mapear todos os atributos multivalorados na mesma tabela de origem, o que acarretaria uma tabela a menos para cada atributo multivalorado no

MER. Tirada essa limitação, que é a ideia do projeto, haveriam várias tuplas com os mesmos valores inseridos porém com apenas esse atributo multivalorado diferente. Isso poderia gerar inconsistências na hora de efetuar a busca ou remoção de tuplas.

4.1.4. Atributos derivados

- Atributo derivado “Avaliação”
    - Solução Adotada:
        - Optamos por mapeá-lo como um atributo na tabela de Anunciante e Contratante, já que, como a média de avaliações das duas entidades é algo frequentemente mostrado para o usuário, é interessante que apenas haja consulta para verificar se esse valor foi alterado.

          A vantagem é que exigirá menor esforço computacional para calcular esse atributo, já que ele é consultado com frequência. A desvantagem é que haverá custo de espaço por termos criados mais um atributo.
    - Alternativa:

      A alternativa é não mapear o atributo e calculá-lo sob demanda. Ou seja, sempre que fosse necessário mostrarmos a média de avaliações do Anunciante/ Contratante, iríamos realizar a média simples de todas as instâncias do atributo “Nota”, o que demanda maior esforço computacional.

4.1.5. Restrições de Integridade

As restrições de integridade foram aplicadas de acordo com as regras explicitadas nas aulas.

- **Chave primária** da entidade **Componentes**
    - Solução Adotada:
        - Para chave primária dos componentes foi utilizado o atributo “ID” que será gerado pelo SGBD durante a operação de inserção de um componente.

          Essa solução foi adotada para que fosse possível identificar univocamente cada Componente instanciado pelo banco sem que fosse necessário efetuar a composição das chaves (fraca de

Componentes + primária de Anunciante, entidade *owner* de Componentes).

- Alternativa:

  - Uma alternativa seria usar a chave de “Componente” como sendo “Nome” e “CPF” (chave primária de “Anunciante”) combinado com esse “ID”, pois “Componente” é entidade fraca de “Anunciante”. Porém isso acarretaria aumento desnecessário de complexidade, visto que o “ID” basta para que os elementos inseridos sejam únicos e diferenciados. Além disso, todas as outras tabelas que fazem referência de chave composta para “Componente” precisariam referenciar estes três atributos e isso acarretaria armazenamento redundante, tornando mais complexo e denso seu acesso.

- **Chave secundária** da entidade **Componentes**

  - Solução Adotada:

    - Para chave secundária dos componentes foram utilizados os atributos “Nome” e “Anunciante” (“CPF” de “Anunciante”) combinados, pois essa também é uma forma única de se referir a um componente.

  - Alternativa

    - A alternativa era deixarmos a tabela de componentes sem chave secundária, tendo apenas o “ID” para a referenciar, o que poderia se tornar um problema visto que para realizar as buscas por componentes de forma mais intuitiva, do ponto de vista do usuário, seria mais difícil, visto que ele não conhece o ID do componente que fora armazenado.

- Restrição da **agregação “Contrato”**

  - Solução Adotada
    - Para um Contrato ser criado, deve-se ter sido solicitado um componente para um Evento (através da relação “possui”). Decidimos definir uma chave primária para a agregação Evento para que esta passasse a ser identificada univocamente por essa cave (Número de Contrato).

  - Alternativa

    - Utilizar a chave composta das chaves de Evento + Componentes. A chave da agregação Contrato seria muito

grande se usássemos a composição das chaves de Evento (ID + CPF do Contratante, pois Evento é entidade fraca de Contratante) composta com ID de Componentes (ou Nome + CPF do Anunciante pois Componentes é entidade fraca de Anunciante).

- Restrição de **unicidade para o atributo "Endereço" de Local**

    - Solução Adotada
        - Como não é possível garantir em nível lógico, o atributo **Endereço** (Rua, número, CEP e Bairro) de **Local** será tratado como *UNIQUE* no SGBD para que a chave secundária de **Avais De Segurança**, que também é estrangeira e uma referência ao atributo Endereço de Local, não seja violada.

    - Alternativa

        - Criar uma chave secundária na entidade Local composta pelos atributos de **Endereço** que, obrigatoriamente, seria única e não-nula. Mas isto não é prático, já que teríamos uma chave secundária para uma entidade específica (Local) da genérica (Componentes).

4.1.6. Restrições de Relação

- Relacionamento "Necessita" (cardinalidade N) entre Montador (participação total) e Infraestrutura e relacionamento "cadastra" (cardinalidade N) entre Montador (participação total) e Anunciante

    - Solução Adotada:

        - Optamos por criar duas tabelas, uma para cada relação, pois é um relacionamento N – N e para esse tipo de relação só existe essa única forma de representar essas relações. **Não dá para garantir a participação total do modelo relacional no esquema de banco**. Para garantir, teríamos que utilizar *trigger* ou *procedure* na aplicação, já que o relacionamento é importante para a funcionalidade o qual está exposto.

- Relacionamento "Possui" entre Contrato e Termo de Compromisso (total nessa ponta) e relacionamento "Necessita de" entre Local e Aval de Segurança (total nessa ponta), ambos de cardinalidade 1-1

    - Solução Adotada:

- No caso do relacionamento “Possui”, optamos por manter o atributo “número” como sendo a chave primária de Termo de Compromisso e adicionar uma chave secundária composta por “IDEvento” (chave fraca de Evento), “Contratante” (CPF de Contratante, *owner* da entidade Evento) e “IDComponente” (chave primária da tabela Componentes), aplicando uma restrição de nulidade nessa chave secundária. Com isso, foi garantida a restrição, pois toda instância de Termo de Compromisso terá um Contrato associado e ele não se repetirá na tabela.

- No caso do relacionamento “Necessita de”, a mesma solução foi adotada, mantendo o atributo “NumeroDoAval” como sendo chave primária e adicionando “Endereço” como chave secundária, também com restrição de nulidade, que referencia “Endereço” em “Local”. Com isso, cada instância de “Aval De segurança” estará ligada em uma instância de “Local” e este não se repetirá na tabela.

- Especialização total da entidade “Componentes”

  - Solução Adotada:

    - A restrição será tratada a nível de banco com a restrição ***check*****,** realizada no atributo “tipo” da tabela de entidade genérica “Componentes” onde só serão permitidos os valores: “Serviços”, “Local” e “Produto”.

  - Alternativa:

    Mapear apenas as tabelas das entidades especificas, porém, como os componentes terão alta taxa de busca, preferiu-se criar uma tabela genérica para evitar **operações de junções** nas consultas.

- Restrição de entidade fraca “Componentes” com “Anunciante”

  - Solução Adotada:

    - Criar tabela de componentes e fazer sua chave primária ser a combinação da chave primária de “Anunciante” com a chave parcial “nome” de “Componentes”, ou seja, CPF (chave estrangeira para o CPF de Anunciante) + “Nome”.

- Restrição de entidade fraca “Evento” com “Contratante”

- Solução Adotada:

    - Criar tabela de Evento e fazer sua chave primária ser a combinação da chave primária de “Anunciante” com a chave parcial “ID” de “Evento”, ou seja, CPF (chave estrangeira para o CPF do Contratante) + “ID”.

4.1.7. Alterações feitas

1. A principal e mais visível alteração feita é referente à agregação Contrato que, antes, se chamava Evento e era em cima da relação de Contratante e Componentes. Agora, a agregação Contrato é em cima da relação entre Evento (uma entidade) e Componentes;

2. Foram removidas as chaves primárias das entidades específicas (Serviço, Local e Produto) já que não é necessário (e recomendado) que as entidades específicas possuam chaves primárias, já que elas a herdam da entidade genérica (Componentes, neste caso);

3. Foi criada uma chave secundária para a entidade Avais de Segurança. Essa chave é uma referência ao atributo Endereço da entidade Local;

4. A entidade Componentes agora possui uma chave primária que será usada para identificar cada univocamente suas instâncias. Assim, a antiga chave primária composta pelo atributo “Tipo” (que agora é “Nome”) e CPF do Anunciante se tornou uma chave secundária;

5. Por conta da **normalização da base** (especificamente a terceira forma normal para chaves primárias), **foi criada uma tabela “Ruas”** contendo os atributos CEP (chave primária), Rua e bairro (atributos não nulos). Isso aconteceu devido a esses atributos dependerem de CEP (um atributo não-primário) e somente dependerem da chave primária transitivamente (CEP -> Rua e Bairro e IDComponente -> CEP), o que fere a 3ª F.N;

6. Foram removidos os atributos Número e CEP das tabelas Fotos e Áreas Comuns. Além disso, foram removidos os atributos Rua e Bairro das tabelas Local, Fotos, Áreas Comuns e AvaisDeSegurança de forma que o atributo CEP faça uma referência ao CEP na nova tabela criada;

7. Foi removido o atributo “Áreas Comuns” da tabela Local, já que foi necessário criar uma tabela específica para Áreas Comuns, pois esse atributo é multivalorado;

8. Foi criada a tabela “TiposDeTransporte” **por conta da normalização da base** (3ª FN para chaves primárias). Como o atributo “capacidade” (de um tipo de transporte) depende do seu tipo, temos um atributo dependendo de outro não primário, mesmo que dependa

transitivamente do primário (IDComponente -> Capacidade e Capacidade -> Tipo). Assim, na tabela TiposDeTransporte, temos a chave primária "Tipo", que serve como referência (chave estrangeira) para "tipo" da tabela Transporte;

9. O projeto inicial era criar apenas uma tabela contendo todos os **Telefones** cadastrados no sistema (tanto de Anunciante, Contratante e Montador). Esbarramos em dificuldades na criação da base pois a chave estrangeira estava referenciando três valores distintos (três tabelas diferentes). Por questões operacionais, isso poderia acarretar erros de compatibilidade e, por tal, resolvemos dividir essa tabela em três: TelefonesAnunciante, TelefonesContratante e TelefonesMontador;

10. Foi adicionado o atributo "Alugado" na tabela Componentes, mediante a alteração feita no MER citada acima.

5. A base de dados

5.1. Normalização

Adentrando um pouco nas alterações citadas em 4.1.7, listamos a seguir os motivos que nos levaram a normalizar a base, bem como as implicações dessas normalizações e o resultado obtido através delas:

1. Foi percebido que a tabela "Local" tinha uma transitividade em suas dependências funcionais (IDComponente (chave primária) -> CEP, IDComponente (chave primária) -> Rua e Bairro e CEP -> Rua e Bairro), o que fere a 3ª F.N para chaves primárias. Para normalizar essa tabela e fazer ela atender à 3ª F.N para chaves primária e genérica (visto que a tabela só possue uma única chave) **foi criada uma tabela "Ruas"** contendo os atributos CEP (chave primária), Rua e bairro (atributos não nulos) que anteriormente estavam armazenados na tabela "Local", e foi mantido na tabela "Local" apenas o atributo CEP, como chave estrangeira que referencia a tabela "Ruas";

2. Foi criada a tabela "TiposDeTransporte" por não seguir a 3ª FN para chaves primárias. Como o atributo "capacidade" (de um tipo de transporte) depende do seu tipo, temos um atributo dependendo de outro não primário, mesmo que dependa transitivamente do primário (IDComponente -> Capacidade, IDComponente -> Tipo e Capacidade -> Tipo). Assim, na tabela TiposDeTransporte, temos a chave primária "Tipo", que serve como referência (chave estrangeira) para "tipo" da tabela Transporte;

Conseguimos, com essas alterações, deixar todas tabelas do projeto atendendo até a forma BCNF e as tabelas que possuem atributos multivalorados também atendendo à quarta forma normal.

5.2. Descrição

Neste projeto optamos por utilizar o **Oracle** como SGBD, **C++** para programar a interface e a comunicação com o SGBD foi feita utilizando a **biblioteca Ocilib** fornecida pela própria Oracle. O SGBD Oracle e a linguagem C++ foram escolhidos pois nós já possuíamos familiaridade, já a biblioteca Ocilib foi selecionada por ser uma interface fornecida pela própria empresa criadora do SGBD utilizado. Todo o desenvolvimento da aplicação foi feito utilizando o Visual Studio 19, que fornece integração padrão com o gerenciador de pacotes do Windows (VCPKG) que fora utilizado para baixar e instalar a biblioteca Ocilib.

5.3. Desenvolvimento da base

Desenvolvemos toda a base de dados seguindo estritamente o que foi planejado no Projeto lógico desenvolvido. O script para criação do banco de dados com todos os comandos DDL utilizados e comentados estão no arquivo *Oracle.sql.* Para a criação da base de dados, tivemos que trabalhar com *Procedures* e *Triggers* para inserir algumas funcionalidades planejadas para o projeto. Utilizamos os *procedures* principalmente para controlar as Ids únicas e automaticamente incrementais geradas automaticamente para algumas de nossas tabelas (como, por exemplo, a tabela de Componentes). Já para os *triggers*, tivemos que utilizá-los pois alguns atributos eram derivados de outros (como o atributo "preço" da tabela Evento, que é a soma de todos os preços dos componentes comprados por um Contratante para tal Evento). Apesar de não fazer parte do escopo da disciplina, tentamos trabalhar com os dois para tornar o banco mais parecido com o que foi especificado no início do projeto. Quanto aos *procedures*, tudo ocorreu conforme o esperado, já para os *triggers*, tivemos problemas com o atributo "Avaliação" de Contratante de Contratante, que é derivado do atributo "Nota" recebido na relação "Avalia" entre Contratante e Anunciante, onde a "avaliação" dos dois é dada pela média (função **AVG** no Oracle) dos atributos "Nota" recebidos para aquela entidade. Não sabemos ao certo o(s) motivo(s), mas, aparentemente, os *triggers* não funcionam na primeira execução dos *inserts* que os ativam mas, ao compilarmos os *triggers* após dado o erro, a inserção funciona normalmente.

Posteriormente fizemos a alimentação do banco apenas para finalidade de testes das consultas, que serão comentadas mais a frente, e para utilização do protótipo. Todos os comandos DML para alimentação da base se encontram no arquivo *inserts.sql.*

5.4. Desenvolvimento do sistema

A aplicação foi feita na linguagem C++ utilizando a biblioteca OCILIB para conexão e interação com o banco de dados Oracle. O desenvolvimento do sistema foi feito para ser de simples acesso e entendimento para quem está usando. Por mais que não tenha um sistema de interface, e sim o terminal, o usuário não terá problemas para interagir com a aplicação, sendo instruído por um menu a cada avanço na utilização do programa.

Sobre os arquivos criados e toda sua desenvoltura, a divisão ficou como segue:

1. ***SystemInterface*.h**

- Arquivo onde se pode compreender toda a estrutura da nossa aplicação. No arquivo .h declaramos todas variáveis, classes e métodos utilizados pelo sistema. No .cpp, elas são definidas, excluindo os métodos das *Nested Classes* (classes dentro de outra classe com métodos que são definidos pelos seus próprios arquivos .cpp).

2. *Application*.cpp

- Arquivo inicial, contém a main. Nela é realizada a conexão com o servidor, através do método *InitializeConnection*. Após isso é chamado nosso sistema de Loops.

(*InitializeConnection*, linha 8)

3. *Loop*.cpp (*SystemInterface*::Loop)

- Arquivo responsável por efetuar os *loops* de exibição para o usuário, além de efetuar os tratamentos e redirecionamentos das escolhas feitas por ele para navegar entre as funções do sistema. O arquivo pode ser interpretado como um organizador hierárquico. Por exemplo: do *Loop*::*Starter* podemos ir para *User*::*Login*, *User*::*Register* ou *Component*::*Search*. Do *User*::*Login*, caso tenha sucesso no *login*, é redirecionado para *Loop*::*Advertiser* ou *Loop*::*Contractor*, e assim por diante.

```
void SystemInterface::Loop::Starter()
{
        // RUNNING INTERFACE
        do {
                Menu.Starter();

                std::cin >> command;
                system("CLS");

                if (CheckCommand(command, 0, 3) == false) continue;

                // MENU OPERATIONS
                if (command == 1) User.Login();
                else if (command == 2) User.Register();
                else if (command == 3) Component.Search();

        } while (command != 0);

        command = -1;
}
```

(*Loop*::*Starter*, exemplo)

4. *Menu*.cpp (*SystemInterface*::Menu)

- Arquivo responsável por armazenar as interfaces de menu. Os menus são gerenciados pelo *SystemInterface*::*Loop*

```cpp
#include "SystemInterface.h"

void SystemInterface::Menu::Starter()
{
	std::cout << "-------------------- MENU --------------------" << std::endl;
	std::cout << "Please, select your action:" << std::endl;
	std::cout << "[1] Login" << std::endl;
	std::cout << "[2] Register" << std::endl;
	std::cout << "[3] Search for a component" << std::endl;
	std::cout << "[0] Exit aplication" << std::endl;
	std::cout << "----------------------------------------------" << std::endl;
}

void SystemInterface::Menu::Advertiser()
{
	std::cout << "----------------- ADVERTISER -----------------" << std::endl;
	std::cout << "Please, select your action:" << std::endl;
	std::cout << "[1] Edit profile" << std::endl;
	std::cout << "[2] Search for a component" << std::endl;
	std::cout << "[3] Announce a component" << std::endl;
	std::cout << "[0] Back" << std::endl;
	std::cout << "----------------------------------------------" << std::endl;
}

void SystemInterface::Menu::Contractor()
{
	std::cout << "----------------- CONTRACTOR -----------------" << std::endl;
	std::cout << "Please, select your action:" << std::endl;
	std::cout << "[1] Edit profile" << std::endl;
	std::cout << "[2] Search for a component" << std::endl;
	std::cout << "[0] Back" << std::endl;
	std::cout << "----------------------------------------------" << std::endl;
}

void SystemInterface::Menu::Profile()
{
	std::cout << "-------------------- INFO --------------------" << std::endl;
	std::cout << "NAME: " << name << std::endl;
	std::cout << "CPF: " << cpf << std::endl;
	std::cout << "E-MAIL: " << email << std::endl;
	std::cout << "PASSWORD: " << password << std::endl;
	std::cout << "------------------ PROFILE -------------------" << std::endl;
	std::cout << "Please, select your action:" << std::endl;
	std::cout << "[1] Name" << std::endl;
	std::cout << "[2] E-mail" << std::endl;
	std::cout << "[3] Password" << std::endl;
	std::cout << "[0] Back" << std::endl;
	std::cout << "----------------------------------------------" << std::endl;
	//std::cout << "Type the information that you want to change: ";
}
```

(SystemInterface::Menu)

5. *User*.cpp (*SystemInterface*::*User*)

Arquivo responsável por cuidar do *login* e *cadastro*.

- ***Login***

  Método responsável por receber e verificar os dados inseridos por um usuário que escolheu a opção de “Login”. Como forma de *login*, o usuário deve inserir o e-mail de cadastro e a senha, após ter informado qual o seu tipo de usuário (anunciante ou contratante). Após isso, para verificar tal integridade, uma busca pelo banco de dados é feita por essas duas informações (os dados são buscados na tabela informada pelo tipo de usuário). Se elas existirem em conjunto, o usuário possui um cadastro no sistema e pode entrar, sendo redirecionado para o *loop* do qual pertence seu tipo. Se o usuário não for encontrado, ou as informações são inválidas, uma mensagem de erro é gerada.

```cpp
// QUERY BUILDER
ocilib::Resultset rs = QueryBuilder("select Nome, CPF, Email, Senha from " + userType[type] + " where upper(Email)=upper('" + email + "') and Senha='" + password + "'");
```

(linha 20)

Acima podemos ver a função “select” sendo usada para buscar as informações de *login* do usuário que está tentando entrar. E-mail e senha são buscados na tabela Anunciante ou Contratante (*userType* é um vetor de strings com dois valores, 0 para “Anunciante” e 1 para “Contratante”).

- ***Register***

Método responsável por cadastrar um usuário no sistema. Caso ele

selecione essa opção, informações de tipo de usuário, nome, CPF, e-mail e senha são solicitadas. Cada informação deve ser inserida após pedida pelo programa. Nesse caso, após o preenchimento, temos algumas considerações para fazer:

1. O CPF deve ser único para garantir inserção no banco, já que esta informação é a chave primária, tanto na tabela “Anunciante”, quanto na tabela “Contratante”. Segue imagem da busca pelo CPF digitado para posterior avaliação:

```
// Check informations
// Verificando se o CPF ja esta em uso
ocilib::Resultset rs = QueryBuilder("select Email, CPF from " + userType[type] + " where CPF='" + cpf + "'");
```

Arquivo *User*.cpp linha 71

2. O e-mail inserido também deve ser único, pois, ao fazer um *login,* para poder checar a informação senha, não pode haver mais um e-mail cadastrado. Sendo assim, segue imagem da busca pelo e-mail digitado para posterior avaliação (porém, um e-mail pode ser cadastrado em Anunciante e em Contratante):

```
// Verificando se o E-mail ja esta em uso
rs = QueryBuilder("select Email, CPF from " + userType[type] + " where CPF='" + cpf + "'");
```

Arquivo *User*.cpp linha 82

Após isso, se todas as informações estão corretas e autenticadas, a inserção desse novo usuário deve ser feita na tabela referente ao tipo de usuário informado. Segue imagem da inserção:

```
// INSERT DATA INTO DATABASE AND COMMIT
st.Execute("insert into " + userType[type] + " values ('" + cpf + "', 0, '" + name + "', '" + email + "', '" + password + "')");
con.Commit();
```

Arquivo User.cpp linha 94

Neste caso, o vetor *userType* funciona como explicado na operação de Login e as outras informações concatenadas na string são as informadas pelo usuário. Por fim, as informações do usuário devem ser salvas para acessos futuros. Para isso, é feito como segue:

```
// RECOVER INSERTED DATA
rs = QueryBuilder("select Nome, CPF, Email, Senha from " + userType[type] + " where Email='" + email + "' and Senha='" + password + "'");

// SET USERINFO
SetUserInfo(rs);
```

Arquivo User.cpp linha 98

6. *ProfileEditor*.cpp (*SystemInterface*::*ProfileEditor*)

Arquivo responsável por editar informações de um usuário cadastrado no sistema. Este usuário pode escolher editar 3 informações:

- **Nome**

  Escolhido editar o nome, basta o usuário digitar o novo nome que deseja

quando solicitado. Após isso, um "update" deve ser feito no banco para que essa informação seja atualizada, como segue:

```
// UPDATE
st.Execute("update " + userType[type] + " set NOME ='" + name + "' where CPF='" + cpf + "'");
con.Commit();
```

Arquivo *ProfileEditor*.cpp linha 10

- **E-mail**

Caso o usuário decida mudar seu e-mail de cadastro, após informá-lo, o sistema verifica se este é único (motivo já explicado neste documento). Sendo assim, uma busca pelo e-mail é feita para e, caso não seja encontrado, um "update" é feito para atualizar o banco.
Vale mencionar que o e-mail é verificado para saber se segue a máscara de corpo de e-mail.

```
// Checar se o email ja existe
ocilib::Resultset rs = QueryBuilder("select Email from " + userType[type] + " where Email='" + tmp + "'");
```

Arquivo ProfileEditor.cpp linha 23

```
// UPDATE
email = tmp;

st.Execute("update " + userType[type] + " set EMAIL = '" + email + "' where CPF='" + cpf + "'");
con.Commit();
```

Arquivo ProfileEditor.cpp linha 36

- **Senha**

Escolhido editar a senha, basta o usuário digitar a nova senha que deseja quando solicitado. Após isso, um "update" deve ser feito no banco para que essa informação seja atualizada, como segue:

```
// UPDATE
st.Execute("update " + userType[type] + " set SENHA = '" + password + "' where CPF='" + cpf + "'");
con.Commit();
```

Arquivo ProfileEditor.cpp linha 47

7. ***Component*.cpp** (*SystemInterface*::Componente)

Arquivo responsável pelas funções de busca e armazenamento de componentes:

- **Busca**

Para este método, o único parâmetro para realizar a busca é o nome do c

```
// Buscar pelos componentes com esse nome na tabela
ocilib::Resultset rs = QueryBuilder("select ID, ANUNCIANTE, PRECO, NOME from COMPONENTE where NOME='" + component + "'");
```

m
ponente, sendo assim temos a query:

((linha 13)

- **Inserção:**

Para inserção, é necessário o usuário estar logado em uma conta do tipo Anunciante. Os parâmetros necessários para inserção são: Nome do componente e preço.

```
// Insercao no banco
st.Execute("insert into Componente values ('" + cpf + "', " + strID + ", '" + componentName + "', " + price + ", 'NAO')");
con.Commit();
```

(linha 60)

O ID é gerado ao pegarmos o maior ID da tabela componentes somado mais um:

```
// Checando ultimo id
ocilib::Resultset rs = QueryBuilder("select MAX(ID) from Componente");

// ID
// Precisamos somar 1 ao ID para que nao tenha restricoes da chave primaria (unica)
int componentID = rs.Get<int>(1) + 1;
```

(linha 50)

- Melhor busca

Para este método nenhuma entrada é necessária. O sistema apenas mostrará os componentes ofertados por anunciantes com avaliação acima da média. Segue a função de banco (imagem seria de difícil visualização, então optamos por mostrar de forma textual – linha 68):

ocilib::Resultset rs = QueryBuilder (

"select anunciante.nome as ANUNCIANTE, componente.nome as NOME_COMPONENTE from componente inner join ANUNCIANTE on Componente.anunciante = Anunciante.cpf where Anunciante.avaliacao >= (SELECT AVG(Anunciante.Avaliacao) FROM Aunciante) GROUP BY Anunciante.nome, componente.nome ORDER BY Anuncainte.nome"

);

6. Conclusão

Analisando o projeto a partir do desenvolvimento das suas três partes, o grupo concorda que o maior ponto de dificuldade aconteceu na primeira parte, onde tivemos dificuldades para elaborar um projeto que fosse de acordo com o tema apresentado. Essa dificuldade desencadeou problemas na elaboração do restante do projeto (MER e projeto lógico). Apesar disso, foi possível se aprofundar nos assuntos apresentados na disciplina e aprender os tópicos conforme foram solicitados.

Como sugestão, o grupo concorda que, para os próximos oferecimentos dessa disciplina, mantendo-se o formato que fora nos apresentado, seria mais interessante que alguns temas de projetos fossem disponibilizados para a escolha de alunos e, dentre esses, ou houvesse uma votação para que um tema fosse escolhido como o único do projeto, ou os alunos seriam livres para elaborar o projeto dentre um dos temas propostos. Assim, a criatividade dos alunos não fica minada de acordo com um tema que fora proposto e que pode acabar por ser bom para alguns e, talvez, não tão legais para outros (nosso caso).

Como crítica, entendemos que a situação que a disciplina se encontrou perante a pandemia não é algo comum, mas achamos que a avaliação foi mais pesada do que o

necessário, principalmente na parte da nota depender, principalmente, da parte 3 do trabalho. Isso acaba tirando um pouco a importância das outras etapas, bem como coloca uma pressão desnecessária em cima dos grupos, que já estão ocupados, também, com outros projetos (já que a parte 3 fora dada no final do semestre, época onde todo o campus está tenso em relação à etapa final do semestre), que não são nem mais, nem menos importantes que outros.

No mais, o grupo gostaria de agradecer ao estagiário PAE, Afonso Matheus, por todo o suporte e atenção oferecidos. A ajuda dele foi essencial (demais) para o desenvolvimento e conclusão do projeto.