UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAYLA MEDEIROS ARAÚJO

MODELIGADO: GERANDO DIAGRAMAS DE SEQUÊNCIA

CAMPINA GRANDE - PB

2022

RAYLA MEDEIROS ARAÚJO

MODELIGADO: GERANDO DIAGRAMAS DE SEQUÊNCIA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

Orientador: Professor Dr. Matheus Gaudencio do Rêgo

CAMPINA GRANDE - PB

2022

RAYLA MEDEIROS ARAÚJO

MODELIGADO: GERANDO DIAGRAMAS DE SEQUÊNCIA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharela em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Matheus Gaudencio do Rêgo

Orientador – UASC/CEEI/UFCG

Professora Dra. Eliane Cristina de Araujo

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

RESUMO

O Modeligado é uma ferramenta utilizada em sala de aula que busca auxiliar o usuário na criação e visualização de diagramas de classes, facilitando a correção desses diagramas por parte dos professores e monitores. Porém, o sistema ainda possuía algumas limitações, sendo difícil observar o fluxo de execução ao executar uma tarefa. Pensando nisso, neste projeto foi acrescentada a funcionalidade de gerar um diagrama de sequência a partir da interação com o diagrama de classe. Os resultados obtidos demonstram que os usuários ficaram satisfeitos com a usabilidade do sistema, mas ainda há pontos a serem melhorados, principalmente quanto à responsividade e exibição de erros.

Modeligado: Gerando Diagramas de Sequência

Rayla Medeiros Araújo
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
rayla.araujo@ccc.ufcg.edu.br

Matheus Gaudencio do Rêgo
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
matheusgr@computacao.ufcg.edu.br

RESUMO

O Modeligado é uma ferramenta utilizada em sala de aula que busca auxiliar o usuário na criação e visualização de diagramas de classes, facilitando a correção desses diagramas por parte dos professores e monitores. Porém, o sistema ainda possuía algumas limitações, sendo difícil observar o fluxo de execução ao executar uma tarefa. Pensando nisso, neste projeto foi acrescentada a funcionalidade de gerar um diagrama de sequência a partir da interação com o diagrama de classe. Os resultados obtidos demonstram que os usuários ficaram satisfeitos com a usabilidade do sistema, mas ainda há pontos a serem melhorados, principalmente quanto à responsividade e exibição de erros.

PALAVRAS-CHAVE

Modeligado, diagrama de classes, diagrama de sequência.

REPOSITÓRIO

https://github.com/matheusgr/modeligado

1. INTRODUÇÃO

Um dos mais importantes e utilizados diagramas UML é o diagrama de classe.[1] Esse tipo de diagrama tem como principal objetivo permitir a visualização das classes que formam o sistema, juntamente com seus respectivos atributos e métodos, bem como demonstrar como as classes se relacionam entre si [1]. Com o objetivo de criar e apresentar a estrutura de um sistema de maneira clara, foi desenvolvido o Modeligado[2], plataforma que gera diagramas de classes a partir de uma linguagem própria.

As classes podem se relacionar de diversas formas, mas, em nosso sistema, focaremos em seis principais: herança (*extends*), implementação (*implements*), associação (*association*), agregação (*aggregates*), composição (*composes*) e dependência (*directionalAssociation*). A seguir uma melhor descrição desses relacionamentos:

- Quando temos uma relação em que *A extends B*, dizemos que A é uma subclasse de B e herda sua estrutura e/ou comportamento.
- Quando temos *A implements B*, sendo B uma interface, temos que A implementa os métodos definidos em B.
- Quando temos *A association B*, A e B possuem um relacionamento estrutural entre suas instâncias e seus objetos estão ligados.
- Quando temos *A aggregates B*, indica-se que B é uma parte de ou está contida em A.
- Quando temos *A composes B*, dizemos que é uma agregação em que B constitui A. Se o objeto da classe A for destruído, as classes de B também serão destruídas, já que as mesmas fazem parte dela.
- Quando temos *A directionalAssociation B*, qualquer mudança na especificação de B pode alterar a especificação de A, pois os objetos de A utilizam serviços dos objetos de B.

Os atributos e métodos definidos nas classes podem ter diferentes níveis de acesso. Temos os seguintes tipos de acesso definidos em nosso sistema: público (+), no qual qualquer outra classe pode acessar; privado (-), em que apenas a classe que possui o atributo ou o método pode acessá-lo; e protegido (#), no qual, além da própria classe, apenas subclasses têm acesso.

Pensando em melhorar a visualização do fluxo de execução quando uma tarefa é executada, neste trabalho adicionamos ao Modeligado a funcionalidade de gerar um diagrama de sequência a partir da interação com o diagrama de classes. O diagrama de sequência é outro importante diagrama UML que tem como objetivo ilustrar a interação entre os componentes de um sistema ao realizar uma função [2]. Este tipo de diagrama é composto basicamente por duas partes: os *objetos*, que indicam quem está executando a ação, e as

[1] "Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original."

[2] https://matheusgr.github.io/modeligado/edit.html

mensagens, que indicam qual ação está sendo executada. Em nosso sistema, definimos os objetos como as classes do sistema, e as mensagens como os métodos chamados e os seus retornos.

O restante deste documento está dividido como descrito a seguir. Na Seção 2 descrevemos o estado atual da ferramenta e a solução adotada. Na Seção 3 apresentamos a arquitetura do sistema. Na Seção 4 apresentamos os resultados obtidos na avaliação da ferramenta. Na Seção 5 discutimos sobre a experiência de desenvolvimento e os principais desafios na construção da solução. E na Seção 6, concluímos o artigo com uma discussão mais abrangente sobre as limitações da solução e os possíveis trabalhos futuros.

2. MODELIGADO

O Modeligado é uma aplicação *web* que pode ser acessada diretamente pelo navegador, através do *link* disponibilizado no repositório do projeto. Logo ao abrir a página inicial, como é mostrado na Figura 1, o usuário tem a possibilidade de utilizar as funcionalidades de criação, visualização e exportação de um diagrama de classes. Inicialmente, a página já traz um código de exemplo e o diagrama resultante.

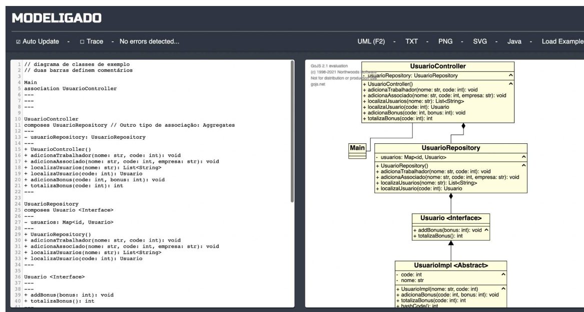

Figura 1 - Página inicial do Modeligado

2.1 Solução Existente

A ferramenta apresenta duas funcionalidades bem definidas que serão discutidas aqui: criação e visualização do diagrama de classes, e Trace, que trata da parte do fluxo de execução do diagrama.

2.1.1 Criação e Visualização de Diagrama de Classes

Foi definida uma sintaxe básica para que fosse possível transformar o texto escrito pelo usuário em diagrama de classe. A definição da classe é dividida em três partes, separadas por traços (---). Na primeira parte, definimos na primeira linha o nome e nas linhas seguintes as relações com outras classes. Na segunda parte, temos os atributos, definindo a visibilidade, o nome e o tipo de cada um. E na terceira e última parte, temos a definição dos métodos, com visibilidade, nome, parâmetros e tipo de retorno; no caso de ser o construtor da classe, o tipo de retorno não é necessário. Finalmente, a definição da classe é encerrada com traços (---).

A partir da linguagem definida é possível criar a visualização de um diagrama de classes. Podemos ver um exemplo desse código na Figura 2, na qual definimos a classe Aluno, com atributos matrícula e nome, e com métodos getMatricula, getNome e setNome.

```
1 Aluno
2 ---
3 - matricula: str
4 - nome: str
5 ---
6 + Aluno(nome: str)
7 + getMatricula(): str
8 + getNome(): str
9 + setNome(nome: str): void
```

Figura 2 - Exemplo de código utilizando sintaxe definida pela plataforma

Na Figura 3 podemos ver o resultado do código, o diagrama de classes gerado. No Modeligado, a visualização é atualizada automaticamente se a opção de *Auto Update*, localizada no canto superior direito da página, estiver marcada, ou pode ser gerada manualmente clicando no botão UML (ou pela tecla de atalho F2). Caso haja algum erro de sintaxe no código, o diagrama de classe não será atualizado e o erro será exibido logo ao lado do botão *Trace*, no canto superior esquerdo, como é mostrado na Figura 4.

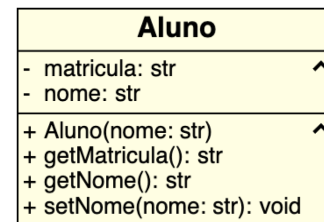

Figura 3 - Resultado do código escrito pelo usuário

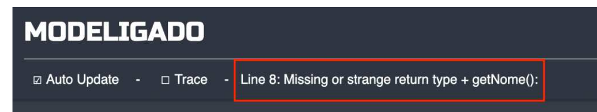

Figura 4 - Página de diagrama de classe com erro de sintaxe

2.1.2 Exportação

Para facilitar a utilização do diagrama fora da plataforma, há a possibilidade de exportá-lo em diferentes formatos. No canto superior direito da página há quatro botões que permitem fazer a exportação do diagrama de classes nos seguintes formatos: TXT, um arquivo de texto do código

usado, PNG e SVG, arquivos de imagem contendo a representação gráfica do diagrama de classes, e Java, que gera um ZIP com as classes Java correspondentes às descritas no diagrama.

2.1.3 Fluxo de Execução

Ativando a opção de *Trace,* presente no canto superior esquerdo da página, é possível interagir com o diagrama de classes e testar seu fluxo de execução. Como podemos ver na Figura 5, ao clicar em qualquer método de uma classe, é mostrada no lado esquerdo da página a ação correspondente, sendo possível interagir com o fluxo de execução para mudar a ordem das ações, através dos botões de *UP* e *DOWN*, e até mesmo deletá-las, através do botão de *DELETE*.

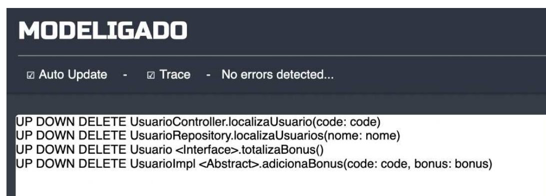

Figura 5 - Página de fluxo de execução antiga

2.2 Solução Proposta

Como podemos ver na Figura 6, nessa evolução adicionamos a visualização do diagrama de sequência, que é gerado automaticamente a partir da interação com o diagrama de classes, permitindo assim acompanhar de maneira mais clara e fácil a comunicação que ocorre entre as classes quando chamamos algum método. Além disso, tivemos o acréscimo da opção de RETORNAR, que possibilita indicar quando um método devolve um resultado para a classe de origem.

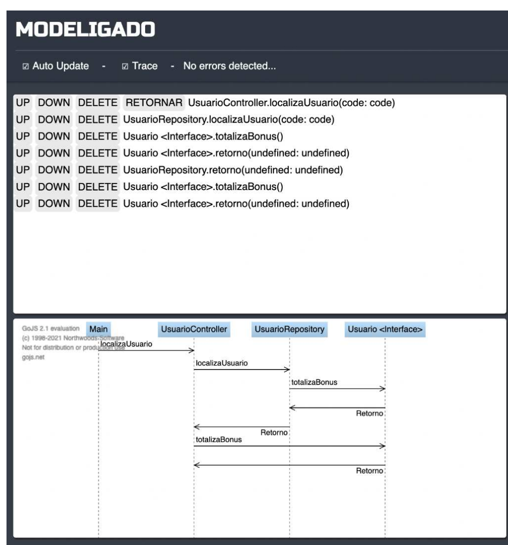

Figura 6 - Página de fluxo de execução nova

3. ARQUITETURA

O Modeligado possui uma arquitetura baseada apenas em *frontend*, tendo em vista que que não é necessário nenhum armazenamento de dados e as operações realizadas possuem baixa complexidade. A página foi toda desenvolvida utilizando HTML, CSS e JavaScript. Para a criação dos diagramas de classes e de sequência, foi utilizada a biblioteca GoJS [3], cujo funcionamento é explicado na Seção 3.1.

Na Figura 7, podemos observar como ocorre a organização da aplicação. A página se comunica com os arquivos sequence_diagram.js, modeligado.js, modeligado_storage.js e go.js para executar as funcionalidades desejadas.

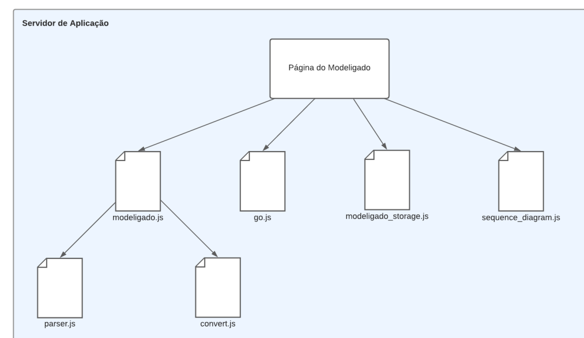

Figura 7 - Arquitetura do Modeligado

Primeiramente, o navegador baixa os arquivos disponibilizados pelo servidor de aplicação e trata de interpretar o código desses arquivos. A página do Modeligado é onde estão concentrados os componentes visuais do sistema, construídos utilizando HTML e CSS, e também é a responsável por fazer a ligação do componente visual à funcionalidade desejada. Essas funcionalidades foram desenvolvidas utilizando JavaScript e separadas em arquivos diferentes para uma maior legibilidade e coesão, como descrito a seguir:

- Quando o usuário escreve um código, este é salvo no *local storage* do navegador. O arquivo modeligado_storage.js é responsável por salvar esse código e carregá-lo na aplicação quando ela é executada. Se não houver nenhum código salvo, ele carrega o código de exemplo da aplicação.
- O arquivo go.js é responsável por criar a estrutura do diagrama de classes, permitir sua atualização e controlar as ações feitas no diagrama.
- O arquivo sequence_diagram.js é responsável por criar a estrutura do diagrama de sequência e permitir sua atualização.
- O arquivo parser.js é responsável por definir as regras da linguagem própria utilizada no sistema para o

desenvolvimento do diagrama de classes e converter esta linguagem no objeto aceito pela biblioteca GoJS.

- O arquivo convert.js é responsável por converter o código UML em código Java, extraindo as classes com seus respectivos atributos, métodos e relacionamentos.
- O arquivo modeligado.js é responsável por fazer a conversão do diagrama em um dos formatos escolhidos. No caso dos arquivos de imagem, o próprio diagrama é convertido. No arquivo TXT é exportado o código escrito pelo usuário. Já na opção Java, o código é convertido para UML e então para Java.

3.1 GoJS

O GoJS é uma biblioteca JavaScript e TypeScript para criação e manipulação de diagramas e gráficos HTML. É bastante flexível, possuindo em sua documentação diversos exemplos de diagramas interativos e editores de gráficos. Pode ser executada pelo próprio navegador ou do lado do servidor em *Node* ou *Puppeteer*; em nosso caso, optamos por executá-la no navegador.

O diagrama é renderizado na página como um componente *canvas* do HTML. Sua estrutura é definida por modelos, que são salvos e carregados como JSON. Esse modelo define os objetos do diagrama e seus relacionamentos, com seus atributos variando de acordo com o tipo de diagrama escolhido para ser visualizado.

Na Figura 8 temos um exemplo do modelo usado na criação de diagramas de classe. No campo nodeDataArray são definidas as classes, indicando seus atributos e métodos. Já no linkDataArray são determinados os relacionamentos entre as classes definidas anteriormente, indicando também o tipo do relacionamento.

Na Figura 9 temos um exemplo do modelo usado na criação de diagramas de sequência. No campo nodeDataArray são definidos os objetos, indicando a posição do objeto e o quanto dura sua linha de vida. No linkDataArray são estabelecidos os relacionamentos entre os objetos definidos anteriormente, indicando também a ação e o tempo em que essa ação ocorre.

```json
{
  "class": "go.GraphLinksModel",
  "copiesArrays": true,
  "copiesArrayObjects": true,
  "nodeDataArray": [
    {
      "key": 1,
      "name": "Aluno",
      "properties": [
        { "name": "id", "type": "int", "visibility": "private" },
        { "name": "nome", "type": "str", "visibility": "private" },
        { "name": "turmas", "type": "List<Turma>", "visibility": "private" }
      ],
      "methods": [
        { "name": "Aluno", "parameters": [{ "name": "nome", "type": "str" }], "visibility": "public" },
        { "name": "getId", "type": "int", "visibility": "public" },
        { "name": "getNome", "type": "str", "visibility": "public" },
        { "name": "setNome", "parameters": [{ "name": "nome", "type": "str" }], "type": "void", "visibility": "public" }
      ]
    },
    {
      "key": 2,
      "name": "Turma",
      "properties": [
        { "name": "id", "type": "int", "visibility": "private" },
        { "name": "nome", "type": "str", "visibility": "private" }
      ],
      "methods": [
        { "name": "Turma", "parameters": [{ "name": "nome", "type": "str" }], "visibility": "public" },
        { "name": "getId", "type": "int", "visibility": "public" },
        { "name": "getNome", "type": "str", "visibility": "public" },
        { "name": "setNome", "parameters": [{ "name": "nome", "type": "str" }], "type": "void", "visibility": "public" }
      ]
    }
  ],
  "linkDataArray": [
    { "from": 1, "to": 2, "relationship": "association" }
  ]
}
```

Figura 8 - Exemplo de código para criação de diagrama de classes

```json
{
    "class": "go.GraphLinksModel",
    "nodeDataArray": [
        { "key": "classe1", "text": "Classe 1", "isGroup": true, "loc": "0 0", "duration": 5},
        { "key": "classe2", "text": "Classe 2", "isGroup": true, "loc": "100 0", "duration": 5 }
    ],
    "linkDataArray": [
        { "from": "classe1", "to": "classe2", "text": "Ação", "time": 0 }
    ]
}
```

Figura 9 - Exemplo de código para criação de diagrama de sequência

3.2 Lógica da Solução

Quando a página é carregada, o diagrama de sequência já é inicializado com a classe Main e é atualizado a partir do histórico de ações feitas no diagrama de classe. Quando o usuário clica em algum método do diagrama de classe, primeiramente é verificado se a classe à qual pertence aquele método já faz parte do diagrama de sequência, para que, caso ainda não faça, ela seja adicionada. Após isso, é percorrido o histórico de ações para que sejam construídas as interações entre os objetos do diagrama. A lógica para definição da origem e do destino dessas interações é descrita a seguir.

Se a ação for um retorno, sua origem vai ser a classe à qual pertence seu método de origem; se não for um retorno, a origem vai ser a classe à qual pertence o método da ação anterior. Caso seja a primeira ação a ser realizada ou os métodos anteriores já tenham sido retornados, a origem será a classe Main.

Se a ação for um retorno, é considerado que seu destino será a classe à qual pertence o último método sem retorno ou o Main, para casos nos quais todos métodos anteriores já tenham sido retornados; se não for um retorno, seu destino vai ser a classe à qual pertence o método selecionado.

Para visualizarmos melhor a lógica descrita anteriormente, vamos analisar o exemplo de fluxo de execução da Figura 10

e, em seguida, na Figura 11, o diagrama de sequência gerado a partir dele.

1. Há uma chamada para o construtor da classe Cliente; como é a primeira ação e há apenas a classe Main no diagrama de sequência, é criado um objeto para Cliente. Também é criada uma relação na qual o Main é a origem e o Cliente é o destino.
2. Há uma chamada para o método solicitaCartao da classe CaixaEletronico. Como a classe ainda não existe no diagrama, ela é adicionada. Também é criada uma relação na qual o Cliente é a origem e o CaixaEletronico é o destino.
3. O método solicitaCartao é retornado, sendo assim criada uma relação na qual o CaixaEletronico é a origem e o Cliente é o destino.
4. Há uma chamada para o método solicitaSenha da classe CaixaEletronico; como a classe já existe no diagrama, é criada apenas uma relação na qual o Cliente é a origem e o CaixaEletronico é o destino.
5. O método solicitaSenha é retornado, sendo assim criada uma relação na qual o CaixaEletronico é a origem e o Cliente é o destino.

UP DOWN DELETE RETORNAR Cliente.Cliente()
UP DOWN DELETE CaixaEletronico.solicitaCartao(cartao: cartao)
UP DOWN DELETE CaixaEletronico.retorno(undefined: undefined)
UP DOWN DELETE CaixaEletronico.solicitaSenha(senha: senha)
UP DOWN DELETE CaixaEletronico.retorno(undefined: undefined)

Figura 10 - Exemplo de fluxo de execução

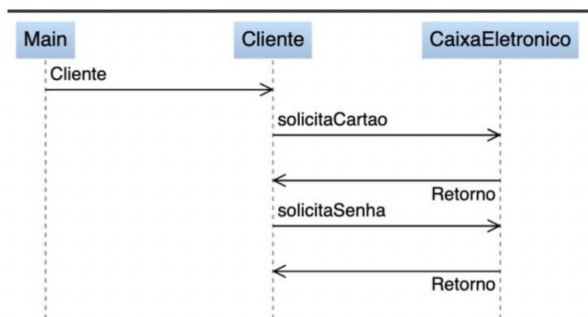

Figura 11 - Diagrama de sequência gerado a partir do exemplo

4. TESTES DE USABILIDADE

Com o objetivo de medir a usabilidade e a satisfação dos usuários em relação ao sistema, foi realizado um teste utilizando o *Computer System Usability Questionnaire* (CSUQ) [4]. Trata-se de um questionário composto por 19 afirmações, nas quais o usuário escolhe entre valores de 1 a 7 para representar a sua concordância com cada afirmação, sendo 1 equivalente a "Discordo totalmente" e 7 equivalente a "Concordo totalmente".

O teste foi realizado com 15 alunos e ex-alunos do curso de Ciência da Computação da UFCG que já haviam cursado a disciplina de Análise de Sistemas. Esse requisito foi colocado devido ao fato do sistema abordar conceitos de diagrama de sequência que são abordados nessa disciplina. Na Seção 4.1 será descrito como se deu a aplicação do teste e seus resultados.

4.1 Experimento

Antes de responder ao teste, foi pedido para que os respondentes realizassem duas tarefas no sistema[3], utilizando como base o diagrama de classe mostrado na Figura 12. O usuário foi guiado a interagir com o diagrama de classe e observar o fluxo de execução e o diagrama de sequência gerado, para analisar se o comportamento era o esperado. Além de interagir com o próprio fluxo de execução para observar se a ação era refletida de maneira adequada no diagrama de sequência.

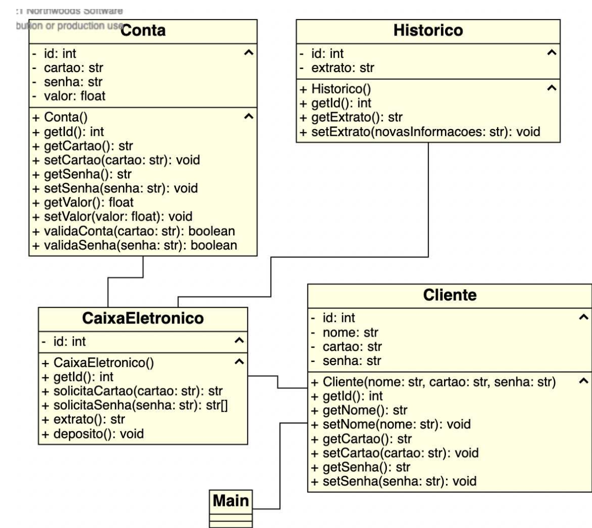

Figura 12 - Diagrama de classe da atividade proposta

Para cada tarefa havia uma situação a ser simulada, quais os passos a serem seguidos e quais métodos deveriam ser usados. Na primeira tarefa foi solicitado que o respondente interagisse com o diagrama de classe a fim de gerar o fluxo de execução para simular um depósito num caixa eletrônico, sem se preocupar com retornos. Era recomendado que seguisse os seguintes passos: o cliente acessa o caixa eletrônico para fazer um depósito; o caixa eletrônico atualiza

[3]https://gist.github.com/RaylaMedeiros/e0e9ac5a389ff17ee4659ac9334cf7b2#file-atividade-md

o valor da conta de acordo com o depósito. Na segunda tarefa foi solicitado que o respondente gerasse um fluxo de execução para simular a solicitação de um extrato num caixa eletrônico, dessa vez se preocupando com retornos. Era recomendado que seguisse os seguintes passos: o cliente acessa o caixa eletrônico que solicita seu cartão para validar sua conta; após a validação da conta, o caixa eletrônico solicita sua senha para validá-la; após a validação da senha, o caixa eletrônico exibe as opções e o cliente solicita o extrato.

Na Figura 13 podemos ver a média das respostas do questionário, em que a média da maioria das perguntas ficou acima de 5. A pergunta nove, que obteve a menor média, tinha como enunciado "O Modeligado exibe mensagens de erro que claramente me dizem como corrigir/solucionar os problemas". Também houve comentários sobre esse aspecto, nos quais as pessoas descreveram que não testaram nenhum fluxo com erros e por isso não sabiam informar a respeito disso.

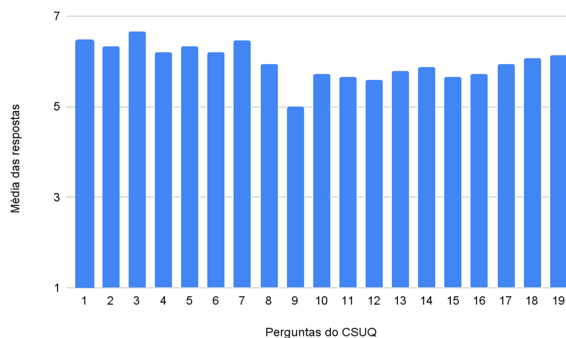

Figura 13 - Gráfico com média das respostas do CSUQ

Um outro problema apontado foi a questão da responsividade do site, já que sua visualização fica bastante prejudicada em telas menores, principalmente quando o diagrama de sequência vai aumentando, sendo necessário utilizar a barra de rolagem para vê-lo por completo. Além disso, comentou-se que o retorno poderia indicar de maneira mais clara na listagem de ações o que está sendo retornado.

Também foi sugerido que houvesse um link para uma documentação no próprio site, como um guia de como utilizar o Modeligado, pois algumas funções ficaram confusas, como o botão *Trace* que foi considerado pouco intuitivo. Além disso, recomendou-se adicionar uma função de exportar o diagrama de sequência assim como há para o diagrama de classe.

Os pontos positivos destacados pelos usuários foram a facilidade para criar tanto o diagrama de classe quanto o diagrama de sequência e a relação estabelecida entre os dois, tida como algo muito importante, já que o diagrama de sequência deixou ainda mais clara as relações entre as classes definidas no diagrama de classe. Além disso, apesar de alguns conceitos estarem confusos no início, a interface foi bastante elogiada em relação a sua simplicidade.

5. EXPERIÊNCIA

Nesta seção, é discutido sobre como foi realizado o processo de desenvolvimento da evolução do sistema, bem como são descritos os principais desafios encontrados ao longo do processo.

5.1 Processo de Desenvolvimento

O modelo de desenvolvimento adotado neste trabalho se baseou no método *Scrum* [5]. Em conjunto com o professor orientador, foram levantadas as funcionalidades desejadas para a melhoria do sistema, descrevendo e dividindo-as em atividades menores. As tarefas foram agrupadas em subgrupos para que fossem desenvolvidas ao longo dos ciclos de desenvolvimento do projeto, denominados *Sprints*.

A implementação do projeto se deu ao longo dos meses de maio a agosto de 2022 e foi dividida em três *Sprints* de um mês cada. Na primeira *Sprint* foi desenvolvida a funcionalidade de gerar o diagrama de sequência a partir da interação com o diagrama de classe. Na segunda *Sprint* foi iniciado o desenvolvimento da funcionalidade de retorno dos métodos no diagrama de sequência. E na terceira *Sprint* foi feita a refatoração do código, juntamente com a finalização do método de retorno. Ao final de cada *Sprint* era apresentado o último estado do produto ao professor orientador para que houvesse uma validação do que foi produzido.

5.2 Principais Desafios

O primeiro desafio do trabalho foi buscar entender como o código já existente funcionava e como acrescentar essa nova funcionalidade mantendo uma boa legibilidade do código. Para isso, foi entendido que seria melhor manter o código para geração do novo diagrama de sequência em um script separado, assim como já era feito com o diagrama de classe.

Outro desafio foi adicionar o diagrama de sequência na página de forma que sua visualização fosse clara para o usuário. Para isso, foi escolhido utilizar componentes semelhantes aos já utilizados, a fim de facilitar o aprendizado do usuário sobre a nova ferramenta. Ainda, utilizou-se o mesmo estilo de *frame* na renderização e foi escolhido posicioná-lo ao lado do diagrama de classe, logo abaixo da lista de ações executadas.

6. TRABALHOS FUTUROS

Primeiramente, é essencial que se busque melhorar ou resolver os problemas apontados nos resultados do teste de usabilidade. Um grande desafio encontrado neste trabalho foi a definição de como ocorreria o retorno de um método. Esse ponto acabou sendo solucionado de forma parcial neste trabalho, definindo-se que o retorno teria como destino a última chamada de método sem retorno.

Em próximos trabalhos, espera-se que sejam estudados outros casos de teste, para que possa ser desenvolvida uma solução para o retorno dos métodos que funcione de forma mais ampla. Por exemplo, poderia haver uma verificação para bloquear retornos em métodos aninhados, a fim de evitar um retorno antes do esperado.

Outro ponto que poderia ser melhorado em próximas versões do Modeligado seria a questão da interface, para torná-la mais agradável e com uma melhor usabilidade. Para isso, poderiam ser executadas ferramentas que avaliam a acessibilidade da plataforma e feitos testes com profissionais da área de design digital para avaliação da mesma.

7. AGRADECIMENTOS

Agradeço primeiramente a Deus, por permitir que eu tivesse saúde e determinação para não desanimar durante todo o tempo de curso. A minha família, por todo o apoio e pela ajuda ao longo de toda a minha jornada de estudos, que muito contribuíram para que eu chegasse até aqui. Ao meu orientador Matheus Gaudêncio, pela paciência e dedicação no acompanhamento deste trabalho.

Por fim, gostaria de agradecer também aos meus amigos que estiveram comigo em toda essa jornada. Dividimos vários momentos de alegria e desespero que nos fizeram evoluir não só academicamente, como também, e mais importante ainda, como pessoas.

8. REFERÊNCIAS

[1] GUEDES, Gilleanes T. A. 2018. UML 2 - Uma Abordagem Prática. 3. ed. Novatec Editora.

[2] Xiaoshan Li, Zhiming Liu and H. Jifeng, "A formal semantics of UML sequence diagram," 2004 Australian Software Engineering Conference. Proceedings., 2004, pp. 168-177, doi: 10.1109/ASWEC.2004.1290469.

[3] [n. d.]. GoJs: A Web Framework for Rapidly Building Interactive Diagrams. Retrieved August 22, 2022 from https://gojs.net/latest/

[4] Lewis, J. R. 1995. IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. International Journal of Human‐Computer Interaction, 7(1), 57-78.

[5] [n. d.]. Scrum: Saiba como usar o Scrum da melhor forma. Retrieved August 22, 2022 from https://www.atlassian.com/br/agile/scrum