UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
TÉCNICAS DE CONSTRUÇÃO DE PROGRAMAS
PROFs.ÉRIKA COTA & STÉFANO DRIMOS KURZ MOR

ETAPA FINAL - RELATÓRIO

ANDY RUIZ GARRAMONES
DAVID MEES KNIJNIK
JOÃO VITOR DE CAMARGO
PEDRO SALGADO PERRONE

PORTO ALEGRE, DEZEMBRO DE 2017

2. PROPOSTA DE TRABALHO

O trabalho proposto no presente relatório trabalha com os custos e ganhos de uma organização, tendo como objetivo desenvolver uma plataforma para análise e previsão de gastos. Ao ler uma planilha de despesas e ganhos da organização em um período anual (separado por meses), o programa deve permitir ao gestor a entrada de previsões para o próximo ano. Essas previsões são feitas por mês e cada rubrica recebe um valor. Nesse universo de discurso, entende-se rubrica como uma “categoria de conta” (e.g. despesas com pessoal, receita de produtos, etc.).

Além disso, a previsão pode ser feita de três maneiras: copiando o valor já existente da rubrica para o mês em questão, definindo uma porcentagem do valor já existente para o dado mês ou definindo um valor absoluto. Cabe ao gestor escolher como o valor da previsão será definido para cada rubrica. Após isso, no ano seguinte, mensalmente, o gestor irá disponibilizar uma planilha com os valores reais do mês. O programa então, deve fazer uma comparação com o valor previsto e, para cada rubrica, analisar e categorizar a confiabilidade da previsão.Uma previsão pode ser categorizada como ruim (quando o que gasto mais ou lucrado menos que o esperado) ou boa (quando foi gasto menos ou lucrado mais).. A avaliação da previsão deve ser mostrada ao gestor de forma intuitiva.

Por último, ainda é pedido que o gestor possa alterar valores do ano corrente, caso necessário. A alteração do valor de previsão em um determinado mês de uma rubrica deve influenciar no valor anual da mesma. Além disso, deve ser possível congelar previsões, ou seja, não permitir alterações depois de uma determinada data.

3. DIAGRAMA DE CLASSES

O diagrama de classes disponível em anexo a este relatório serviu como base e apoio para o desenvolvimento da aplicação. É importante salientar que durante o desenvolvimento da aplicação, houve necessidade de mudanças no diagrama, pois constatou-se que, em certos aspectos, sua montagem não favorecia/respeitava os conceitos de modularidade definidos por Meyer e estudados na cadeira. Todavia, o diagrama em anexo está completamente atualizado. Abaixo, a explicação do diagrama de classes divididos por pacote. Para a leitura das explicações, é interessante acompanhar olhando o diagrama.

3.1: pacote Interfaces (pacote cinza):

O pacote Interfaces contém as interfaces utilizadas na aplicação. As interfaces são estruturas que visam descrever comportamentos e se abstém de detalhes de programação. O padrão interface-based diz que um sistema deve basear suas operações em tipos abstratos (interfaces) se preocupando com comportamentos e não implementações. Dessa forma, foram desenvolvidas três interfaces, que servirão como objeto de trabalho para os controladores futuramente desenvolvidos.

A primeira interface a ser explicada é a BudgetItem, que representa uma linha de orçamento. No universo de discurso em questão, essa interface define os comportamentos de uma rubrica, portanto possui dados métodos de acesso e escrita para o código da rubrica, sua classificação, entre outros. É importante comentar que, conforme consulta realizada aos professores, a definição de métodos de acesso (getters e setters) não é muito comum em interfaces, porém podem acontecer quando esses métodos são necessários para outras classes, como controladores, por exemplo.

Para guardar os valores anuais de uma rubrica, foi pensado numa segunda interface nomeada BudgetItemAnnualValues. Essa separação foi realizada pois o conceito de como os valores de uma rubrica são armazenados podem alterar sem que as rubricas em si altere. Portanto, essa interface possui métodos de acesso a conjuntos que salvam os valores da rubrica em um determinado ano. Assim, existem três conjuntos de 12 posições: um de valores previstos, um de valores realizados e um de valores resultantes (resultado da comparação entre previstos e realizados). Para relacionar a BudgetItem (comportamento da rubrica) com a BudgetItemAnnualValues (comportamento dos valores da rubrica), é adicionado um Map (sem implementação definida) na BudgetItem, em que o índice é o ano e o objeto é um BudgetItemAnnualValues. Assim, para cada ano, uma rubrica possui conjuntos de valores (previstos, realizados e resultantes) novos. Isso caracteriza uma composição, pois não há sentido em existir valores de rubrica sem uma rubrica em si.

A última interface definida foi a ExpirationDate. Essa interface contém o comportamento de uma estrutura que controla as datas limites para alteração de previsões. Novamente há um Map, cujo índice é um ano e os objetos são datas. Foi decidido não vincular essa informação diretamente às outras interfaces, pois alterações no modo como é realizado o limite da data não deve acarretar em alterações nas rubricas e seus valores (e vice-versa). A interface em questão contém métodos que definem acesso e gerência sobre esse Map.

3.2: pacote Implementations (pacote verde):

Este pacote contém possíveis implementações para os comportamentos definidos nas interfaces do pacote Interfaces. Importante comentar que ideia de sistema proposta aceita com que uma nova implementação de uma determinada interface seja desenvolvida e passada para o sistema usar sem que mais alterações sejam necessárias.

Assim, as classes Rubric, RubricPerYear e PredictionExpirationDate são sugestões de implementações para as interfaces BudgetItem, BudgetItemAnnualValues e ExpirationDate, respectivamente. Uma outra ressalva é que a única classe que vê essas implementações é o ApplicationController, explicada em breve nesse documento, que define quais implementações serão usadas, umas vez que os controladores, explicados a seguir, só conhecem as interfaces.

3.3: pacote Controllers (pacote azul):

Este pacote contém classes genéricas que servirão para realizar operações sobre os modelos definidos nas interfaces. A ideia aplicada nas classes desse pacote (e validada com a professora) é que eles trabalhem genericamente com instâncias que implementam as interfaces previamente definidas. Assim, o controlador, em sua definição, não possui conhecimento de com qual implementação ele está trabalhando, visto que as implementações são apenas passadas pelo ApplicationController na criação dos controladores. É possível fazer um paralelo com a classe List, do Java, que recebe com qual tipo trabalhará da seguinte forma: List<String>. Os controladores definidos nesse pacote trabalham da mesma forma, ou seja, para chama-los, deve-se usar: NomeDoControlador<ImplementaçãoEscolhida>.

Explicando os controladores em si, agora, existem quatro. O primeiro, chamado de BudgetItemAnnualValuesController, intuitivamente trabalha com a interface BudgetItemAnnualValues, assim, ele possui operações para trabalhar com objetos de classes que implementam essa interface.. Esse controlador conhece apenas a interface falada acima. O segundo controlador se chama BudgetItemController e trabalha com as interfaces BudgetItem e BudgetItemAnnualValues. Essa classe possui operações que trabalham com o comportamento de rubricas (por isso a BudgetItem) e, por necessidades, com seus valores (por isso a BudgetItemAnnualValues).

O próximo controlador é o PredictionController e ele faz a intermediação entre as interações com o usuário e as duas interfaces também usadas no parágrafo anterior. Como não há um comportamento definido para previsões - visto que da forma como foi montada a arquitetura, uma previsão é apenas o ato de definir um valor para uma rubrica em um determinado mês e ano (baseado no mesmo mês do ano anterior) -, o PredictionController simplesmente trabalha com as rubricas e seus valores. Importante comentar que esse controlador dispara exceções que serão abordadas a seguir. Por último, há o ExpirationDateController, que trabalha com a interface ExpirationDate - única ainda não usada pelos controladores - e possui operações com a data limite definida pelo gestor.

3.4: pacote Exceptions (pacote branco):

Esse pacote possui exceções desenvolvidas especialmente para o universo de discurso em questão. Basicamente, foram desenvolvidas três exceções, sendo as duas primeiras pensadas para interagir diretamente com o gestor e a terceira mais uma exceção por um possível mal comportamento do sistema.

A primeira exceção pensada foi a InvalidCodeException. Dependendo de como será desenvolvido o módulo de interação com o usuário, pode ser que exista a possibilidade de ele inserir um código para buscar ou realizar a previsão de uma rubrica. Caso o código não é encontrado, a InvalidCodeException é disparada. Depois existe a InvalidPredictionDateException, disparada quando o usuário tenta realizar uma previsão depois da data limite. Por último, a exceção que não é diretamente vista pelo usuario, a InvalidPredictionTypeException, que é disparada quando uma maneira incorreta de previsão é selecionada - isso fará mais sentido ao ler o pacote a seguir e ver que os tipos de previsão são salvos em um Enum.

3.5: pacote Utils (pacote rosa claro):

Esse pacote contém classes úteis para o sistema. São elas uma classe para leitura e interpretação de arquivos (FileInterpreter), dois Enums (Months e PredictionTypes) e uma classe para trabalhar com Enum de meses (MonthsController).

A classe mais complexa desse pacote é a FileInterpreter. A função dela é ler os arquivos de entrada e mapear eles para as interfaces definidas. Entretanto, ele não contata diretamente as interfaces, pois para evitar acoplamento, foi decidido montar a arquitetura de forma com que o FileInterpreter apenas realizasse operações presentes nos controladores. Assim, os únicos que acessam diretamente as interfaces são os controladores delas e as classes mais acima (como a FileInterpreter e a PredictionController) apenas acessam esses controladores. Importante comentar que a FileInterpreter possui operações tanto para ler o arquivo de histórico de valores quanto os arquivos de realizado mensal.

A Months é uma Enum que possui os meses para qual é possível fazer a previsão. No caso, são todos os meses do ano, porém a ideia é fazer com que apenas um lugar do código (a MonthsController) possua uma lógica para realizar, por exemplo, conversões entre um dado nome de mês e seu índice (o vetor de valores necessita acessar a posição 0 para Janeiro, por exemplo). Isso foi pensado para respeitar o conceito de single-choice, definido por Meyer.

O mesmo conceito motivou a criação do PredictionTypes, que possui as maneiras nas quais é possível realizar uma previsão. Apenas a PredictionController deve ver esse Enum. Caso uma nova forma de prever for adicionada, basta adicioná-la à PredictionTypes e colocar uma forma de acessá-la na PredictionController.

Nesse pacote também se encontra a classe Triple e Quintuple. Ambas classes nada mais são do que uma tupla, de tamanho 3 e 5 respectivamente. Foram criadas para poder agrupar diferentes tipos de informações em elementos, tuplas, de forma a permitir uma maior flexibilidade entre classes que comunicam informações. Desta forma não é necessário sequer saber os tipos de cada elemento da tupla, apenas repassar a informação.

3.6: pacote Interaction (pacote amarelo):

Esse pacote possui classes que servem para realizar a interação do sistema previamente descrito. A principal classe desse pacote é a ApplicationController, que possui o papel de orquestrador do sistema. Um possível módulo de interação com o usuário pode ser adicionado e a única classe que ele precisa conhecer do sistema é a ApplicationController, pois é ela que realiza a comunicação entre os dados inseridos pelo usuário e os controladores previamente explicados. Foi criada uma classe

DesktopClientController, que estende a ApplicationController, para que aplicações client em computadores usem ela para se comunicar e usufruir do nosso sistema. Outra classe presente nesse pacote é, por exemplo, a UIConsole, uma classe praticamente sem lógica, que apenas recebe dados do usuário pelo console e usa a DesktopClientController para gerenciá-los.

No nosso caso foi implementado também uma pequena aplicação cliente através de Interfaces Gráficas de Usuário .Como explicado acima, a parte do “client” não precisa saber nada da aplicação além do que a DesktopClientController fornece e é com ela únicamente que se interage para usar o nosso sistema .Por isso, a classe UIGraphical não tem lógica alguma, apenas instância um GUIManager que gerencia toda a parte gráfica do cliente se comunicando com a DesktopClientController.

3.8: pacote Tests (clases azul celeste espalhadas):

O pacote Tests contém as classes de testes unitários. Com exceção de TestsHelper, cada classe deste pacote se refere a testes de uma determinada classe de implementação ou de um determinado controlador presente na aplicação. Por sua vez, a TestsHelper é uma classe com métodos comuns à mais de uma classe de teste, como a codificação de datas para texto (e vice-versa) ou o retorno de estruturas de dados utilizadas em stubs (pedaços de código que simulam outra funcionalidade). O objetivo dessa classe é evitar duplicação de código e centralizar comportamentos comuns, buscando modularidade.

As experiências acumuladas demonstram que o início da atividade geral de formação de atitudes cumpre um papel essencial na formulação dos índices pretendidos. Por outro lado, o desafiador cenário globalizado causa impacto indireto na reavaliação do retorno esperado a longo prazo. A prática cotidiana prova que o julgamento imparcial das eventualidades estimula a padronização de alternativas às soluções ortodoxas. Podemos já vislumbrar o modo pelo qual a execução dos pontos do programa auxilia a preparação e a composição de todos os recursos funcionais envolvidos.

3.7: pacote GUI (pacote rosa escuro):

Aqui estão todas as Classes relacionadas com o client gráfico. A classe central e que é responsável por tudo é a GUIManager, pois ela que instancia todos os Frames (janelas gráficas) e se relaciona com eles, além de interagir com o nosso sistema através do DesktopClientController.

A classe GUIManager implementa 3 interfaces: UserMenu, BudgetCalculator e BudgetViewer. A ideia foi abstrair qualquer lógica dos JFrames de tal forma que eles pudessem ser utilizados até por outras aplicações e clients distintos, pois eles só enxergam uma Interface e alguma classe implementará todas aqueles métodos conhecidos definidos (botão foi clicado, foi selecionado tal coisa, mudou o ano etc...). Portanto a única lógica destas janelas gráficas é avisar as ações que aconteceram e a classe que implementa a interface tratará os eventos conforme a aplicação necessitar.

4. DIAGRAMAS DE SEQUÊNCIA

O diagramas de sequências estão dividido em 4, um para cada operação principal da interface de comando. Esses diagramas foram baseados na classe UIConsole, que simula uma simples interação entre o usuário e o sistema, como explicado anteriormente.

4.1: Sequence_Prediction:

Diagrama que representa as operações que o software faz para fazer a previsão de valores para o próximo ano.

A aplicação começa chamando a função predict da classe ApplicationController. Essa função testa se o mês é válido por meio da operação 1.2.1, calcula o índice do mês dado com a operação 1.2.3, depois pega a lista de Rubricas da classe BudgetItemController(1.2.4), e usa esta lista na função FindByCode (1.2.5) da mesma classe, que retorna a rúbrica de código igual ao dado pelo usuário. A função então atualiza os parâmetros para a previsão em 1.2.6 usando a classe recebida, e chama a função predict da classe PredictionController (1.2.7). Essa última função pega os valores da previsão com a operação 1.2.7.1, pega os valores anuais de Rubricas com 1.7.2.4 e calcula a Previsão por mês com 1.2.7.5.

4.2: Sequence_Read:

Este diagrama representa a operação de ler um arquivo CSV de valores reais.

A função principal chamada para efetuar essa operação, readRealValuesFile da classe interpretadora de arquivos FileInterpreter é chamada na operação 1.2.1. Seguindo, o programa iterativamente pega a Rubrica por seu código em 1.2.1.5 e, se a Rubrica não é nula, o programa atualiza os seus dados em 1.2.1.8.

4.3: Sequence_Compare:

Este diagrama mostra as operações que mostram ao usuário as diferenças entre as previsões de orçamentos e os orçamentos reais.

O programa começa pegando o nome e o código da rubrica (1.2.4 e 1.2.5) e mandando-os ao usuário (1.2.6), depois recebe os valores anuais da rubrica em 1.2.8, e imprime na tela, iterativamente, os valores de previsões (1.2.11), orçamentos (1.2.12) e resultados (1.2.14) da rubrica.

4.4: Sequence_Show:

Diagrama que representa as operações que o software faz para mostrar ao usuário os anos nos quais há previsões de orçamento.

O programa imprime todos os anos recebidos da função getYears(). Esta função função, chamada em 1.2 pega todas as rubricas em 1.2.1, e busca a rúbrica com o maior número de projeções anuais, retornando todos os anos das projeções.

5. IMPLEMENTAÇÃO

Nesse capítulo serão explicados detalhes de implementação do projeto. É necessário comentar que boa parte da lógica usada já foi explicada no diagrama de classes, visto que o objetivo desse recurso é definir, o mais claramente possível, como será a implementação. Dessa forma, além do anexo contendo o projeto em Java, apenas considerações não contidas no capítulo sobre o diagrama de classes serão abordadas aqui.

5.1: sobre o relacionamento entre rubricas:

No cenário passado ao grupo, mostrou-se a existência de rubricas que eram compostas por um conjunto de outras rubricas. Por isso, existe, na interface BudgetItem e na implementação Rubric, uma lista do tipo BudgetItem. Assim, a estrutura de dados idealizada foi a de uma floresta. Cada rubrica pode ter de 0 a n rubricas filhas. Essas rubricas filhas, por sua vez, podem ou não possuir outras rubricas filhas. O fato de não possuir uma raíz única, mas sim um conjunto de raízes faz com que a ED se caracterize como uma floresta n-ária e não uma árvore n-ária.

A localização de uma rubrica dentro da floresta depende de sua classificação, campo contido no arquivo de exemplo. Se a primeira rubrica lida possuir classificação 1, por exemplo, ela será adicionada como uma das raízes da floresta. Uma próxima rubrica com classificação 1.1 seria colocada coma filha da rubrica 1 e assim por diante. Assim, as raízes da floresta são rubricas que não devem fazem parte de nenhuma outra parte da própria floresta. Importante comentar também que rubricas sem classificação são adicionadas como raízes; rubricas sem código são adicionadas na localização dada pela sua classificação com um código aleatório único e rubricas sem ambos (código e classificação) serão raízes com código aleatório único.

5.2: sobre a leitura de arquivos:

Um dos objetivos durante o desenvolvimento do módulo de leitura e interpretação de arquivos foi fazer com que todas as informações necessárias para o sistema fossem extraídas dos arquivos dados como exemplo no Moodle. Entretanto, houve um empecilho: haviam rubricas cujo valor era definido por fórmulas, que estavam presentes no arquivo .XSLX, mas não presentes no .CSV.

Devido a isso, houve a necessidade de adicionar no .CSV de histórico de rubricas mais uma coluna, que é respectiva a fórmula que determina seu valor. Visto que havia essa necessidade, foram definidos três padrões de fórmula para as rubricas. Rubricas filhas na floresta descrita no subcapítulo anterior não têm seus valores dados por ninguém além delas mesmas, pois não têm filhas, portanto sua fórmula é vazia. Rubricas cujo valor é definido pelo somatório dos valores dos filhos, netos e afins, possuem a fórmula “SUM”. Caso o programa encontre essa fórmula, ele definirá que o valor da rubrica em questão é simplesmente a soma de todas as rubricas abaixo dela na floresta. O terceiro tipo de fórmula são aquelas começadas com “F:”. Essas fórmulas foram criadas de forma com que o programa conseguisse interpretar a função descrita nela. Se, por exemplo, uma rubrica tem a fórmula “F: 104 + 108 - 114”, significa que o valor daquela rubrica deve ser a soma dos valores das rubricas de código 104 e 108 menos o valor da rubrica de código 114. Dessa forma, após a leitura do arquivo de histórico, é chamado um método que realiza o cálculo do valor de todas as rubricas novamente, buscando evitar inconsistências (como

uma rubrica possuir valor R$50,00, sendo composta por duas rubricas cujo o valor é R$100,00 cada).

Uma outra coluna foi adicionada. Seu objetivo é identificar se a rubrica se caracteriza como um custo (despesas, pagamentos, entre outros) ou não. Essa coluna facilita na hora de visualização, pois ela como a avaliação deve ser feita. Por exemplo, se para uma determinada rubrica eu previ R$50 e o realizado foi 70R$, quer dizer que houve uma discrepância de R$20. Porém se essa rubrica é uma despesa, foram gastos R$20 a mais, o que é ruim, mas, por outro lado, se for um ganho, quer dizer que houve um ganho de R$20 a mais que o previsto, o que é bom.

Concluindo, as duas colunas adicionais abordadas acima foram as únicas modificações realizadas no CSV de histórico de entrada. Para o CSV de realizados, nenhuma alteração foi necessária.

5.3: sobre controladores genéricos:

O cenário ideal é que um controlador apenas conheça comportamentos, definidos através de interfaces. Esse conceito, chamado de interface-base - já comentado anteriormente no presente trabalho -, foi aplicado nos controladores desenvolvidos. Entretanto, como em alguns momentos, a instanciação de objetos das classes de modelo era necessária (e não há como instanciar um comportamento), foram observadas duas opções: atrelar as instâncias a implementações específicas (aumentando acoplamento e perdendo os benefícios do padrão interface-based) ou aprender e implementar controladores genéricos. Visto que o objetivo do trabalho era aplicar conceitos de modularidade e boas práticas, optou-se pela segunda opção.

Assim, como explicado anteriormente, os controladores devem receber quais implementações usarão. A ApplicationController, responsável por orquestrar o sistema, é a classe que diz a todos os controladores que implementações serão usadas. Assim, caso uma nova implementação para a interface BudgetItem fosse criada, precisaríamos apenas definir será usada e os controladores devem entendê-la.

5.4: sobre testes:

No desenvolvimento dos testes, buscou-se evitar redundâncias e repetições de códigos desnecessárias. Assim, se uma determinada classe de alto nível A chama métodos de uma classe B (presente em um nível mais abaixo na arquitetura), não foram criados testes para as chamadas em si, afinal os métodos já possuem testes descritos na sua respectiva classe de testes.

Dessa forma, a ApplicationController - que basicamente apenas cria instâncias dos controladores e realiza o intermédio entre eles e as entradas do usuário - não possui uma classe de testes específica para ela, uma vez que os métodos que ela usa já foram testados nas classes de testes dos controladores.

5.5: sobre arquivos:

O arquivo de histórico no padrão modificado está junto com o ZIP do projeto em Java e se chama a.csv.