INSTUTO FEDERAL DO ESPÍRITO SANTO

BRUNO DELL'ORTO MERGH

GABRIELA PONTES BREDER

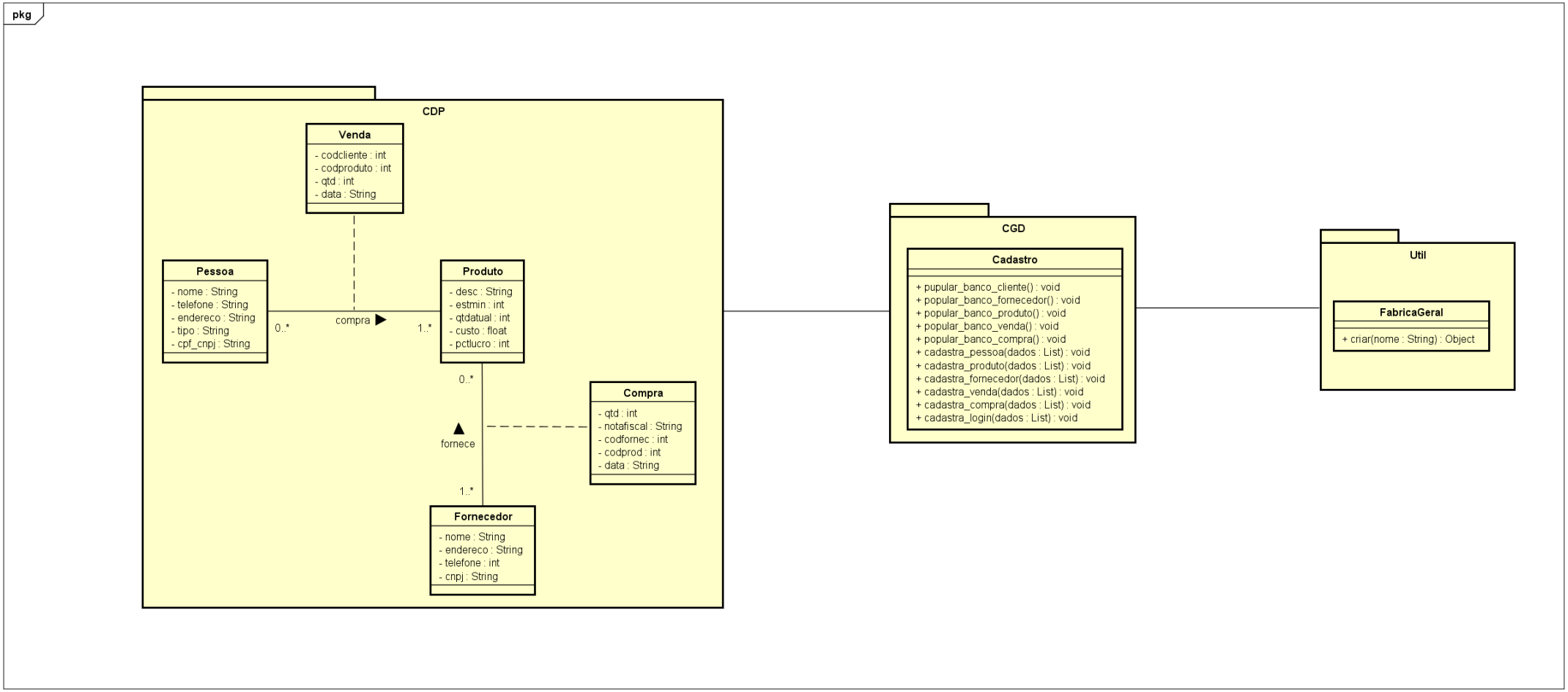
TRABALHO DE PROGRAMAÇÃO ORIENTADA A OBJETOS 2

SERRA

2015

Sumário

DIAGRAMA DE PADRÕES DE PROJETO



powered by Astah

OBS: A relação entre as classes do CDP e a fábrica é feita através da classe *Cadastro*. O método fábrica é utilizado para criar objetos.

A primeira situação é quando o usuário deseja fazer o backup do banco de dados. As funções de popular_banco no CGD lêem os arquivos, chamam a *FábricaGeral*, que retorna um objeto do tipo requerido. A partir daí, ele é inserido no banco e o backup feito.

Além disso, a fábrica também é utilizada para criar objetos na hora de cadastrar, mas esse cadastro é feito por intermédio do padão Facade, que explicaremos mais à frente.

EXPLICAÇÃO DE USO DOS PADRÕES

1) Composite

Esse padrão de projeto realiza a composição de objetos em uma estrutura de árvore e é utilizado quando queremos representar uma hierarquia parte-todo de objetos. Os clientes então ignoram a diferença entre composições de objetos e objetos individuais. Eles tratarão todos os objetos na estrutura composta de maneira uniforme.

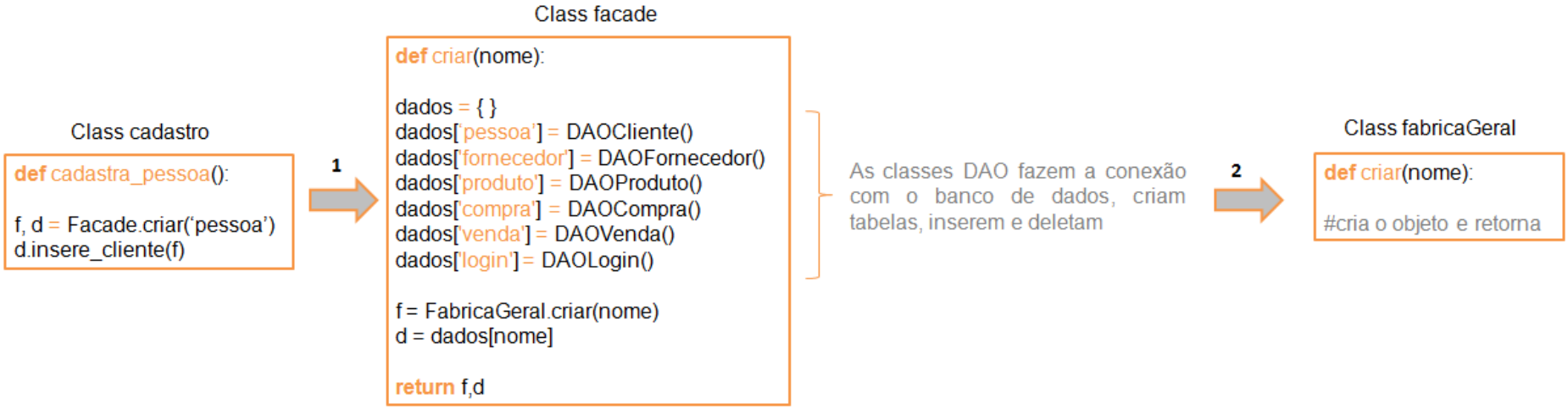
O Composite foi aplicado para a utilização das telas. Quando o programa é executado e a interface exibida, o usuário não sabe se o que está vendo é uma montagem de objetos, ou apenas um individual. O resultado final que é mostrado ao cliente é uma composição de vários outros elementos, no caso, imagens, que sobrepostas formam o todo.

2) Decorator

O objetivo do Decorator é dinamicamente agregar responsabilidades adicionais a objetos. Pelo fato do nosso minimundo estar muito simples, não foi possível implementar esse padrão. Para o próximo trabalho, serão criadas novas regras de negócio, viabilizando a implementação do Decorator.

3) Facade

É utilizado quando necessita-se executar uma sequência de atividades para realizar determinada tarefa. Define uma interface de alto nível para acessar os subsistemas. Como dito na observação do nosso diagrama de padrões de projeto, quando o usuário deseja cadastrar uma pessoa, fornecedor, produto, compra ou venda, a classe *Cadastro* no CDP chama a *Facade* no CGD e está chamará a *fabricaGeral.* O exemplo abaixo ilustra o funcionamento do código, caso queiramos cadastrar uma pessoa

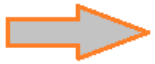


Dessa forma, a sequência de atividades que temos que realizar pelo método facade é invocar as classes DAO, para que consigamos em *cadastro* inserir o cliente, por exemplo. Nesse caso, a variável f retornada na classe *facade* será um objeto do tipo *pessoa*, criado pela fábrica, e d será do tipo *DAOCliente*. A partir daí é possível inserir, na classe *cadastro*, o cliente no banco de dados.

4) Flyweight

Esse padrão aplica-se quando precisamos criar uma grande quantidade de objetos de um tipo que podem consumir grande quantidade de memória. Antes de criar um objeto, a implementação do flyweight verifica se o mesmo já não foi criado uma vez, pois se foi, ele é retornado para o cliente. Somente em caso contrário é criado um novo objeto, evitando o gasto excessivo de memória.

```
    self.flys['ControleTelaCadFornecedor'] = ControleTelaCadFornecedor()
    self.flys['ControleTelaCadProduto'] = ControleTelaCadProduto()
    self.flys['ControleTelaCadCompra'] = ControleTelaCadCompra()
    self.flys['ControleTelaCadVenda'] = ControleTelaCadVenda()
    self.flys['ControleTelaRelatorio'] = ControleTelaRelatorio()
    self.flys['ControleTelaRelatorioGerado'] = ControleTelaRelatorioGerado()


def FlyweightTela(self,nome):

    return self.flys[nome]
```



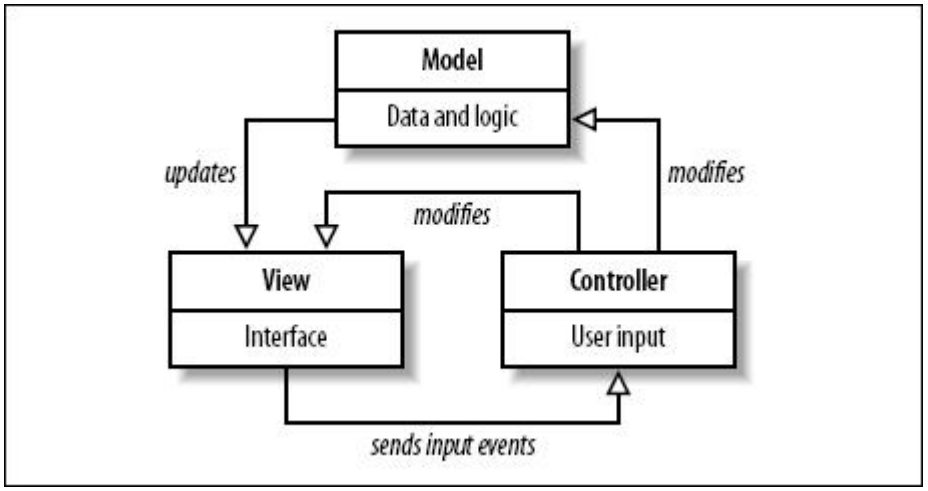
```
flyfactory = FlyweightFactory()

while True:

    janela.screen.fill(0) #limpa a tela
    main_clock.tick(8)

    if opcao == TelaLogin:
        if str(type(tela)) != "<class 'ifes.cih.telaLogin.TelaLogin'>":
            tela = flyfactory.FlyweightTela('TelaLogin')
            controle = flyfactory.FlyweightTela('ControleTelaLogin')
        opcao, listaCadastral = controle.controle()
        tela.imprime(janela,controle.posicao, listaCadastral)
```

5) Proxy, adapter e bridge

Não foram aplicados para essa parte do trabalho. Pendentes para a próxima parte.

DESCRIÇÃO DA UTILIZAÇÃO DO MVC

O padrão Modelo-Visão-Controlador (MVC) considera três papéis relacionados à interação humano-computador. O modelo consiste nos dados da aplicação, regras de negócios, lógica e funções. A visão refere-se à entrada e a exibição de informações na IU. Pode ser qualquer saída de representação de dados. Qualquer requisição é tratada pelo terceiro papel: o controlador. Este pega a entrada do usuário, envia uma requisição para a camada de lógica de negócio, receber sua resposta e solicita que a visão se atualize conforme apropriado. É então a mediação da entrada, convertendo-a em comandos para o modelo ou visão.



- CDP

Na Camada de Domínio do Problema estão as classes *compra*, *fornecedor*, *pessoa*, *produto* e *venda*. Todas essas classes são identificadas na fase de análise. As classes pfisica e pjuridica da parte 1 do trabalho foram retiradas, pois ao adicionarmos o uso do banco de dados colocamos um atributo tipo na classe pessoa. Além disso, criamos uma nova classe login, utilizada por conta da interface.

- CGD

Na Camada de Gerência de Dados estão as classes cadastro e relatorio. Como o nome diz, essa camada compreende a definição das classes gerenciadoras de tarefas. Na classe *cadastro*, estão as funções de popular o banco, que fazem a leitura/backup dos arquivos existentes. Além disso, há as funções de cadastro de pessoa, produto, fornecedor, venda e compra. Já na classe *relatorio* são gerados os arquivos WriteApagar (valor devido aos fornecedores), WriteAreceber (valor a receber por cliente), WriteVendasPorProduto (vendas e lucro por produto) e WriteEstoque (gerenciamento de todos os produtos do estoque). Nesta camada então estão as classes que requerem o armazenamento de dados.

A descrição acima já estava implementada na parte 1. O que mudou agora foi a adição das classes DAO, relacionadas ao uso do banco de dados: *daocliente*, *daocompra*, *daofornecedor*, *daologin*, *daoproduto* e *daovenda*. Elas se conectam ao banco e possuem as funções de inserir, retornar (cliente, compra, fornecedor...), e deletar. Outra mudança foi a criação da classe facade, para a implementação desse padrão de projeto.

- CIH

O Componente de Interação Humana trata do projeto da interação humano-computador. Essa camada foi a que mais sofreu modificações, por conta da criação de uma interface. Antigamente, possuía somente a classe menu, que imprimia as opções de entrada ao cliente e estabelecia então a comunicação entre o usuário e o computador. Foram criadas no CIH classes que exibem as telas de cadastro, tela inicial, de login, relatórios, enfim, tudo que a interface exibe ao usuário.

- CCI

O Componente de Controle de Interação trata das classes responsáveis por controlar a interação (ativação/desativação dos objetos do CIH). Na parte 1, o CCI possuía apenas a classe *control* - que, dado o valor do input do usuário, chamava as classes e funções responsáveis para executar o que ele desejava.

Nesta camada também foram adicionadas muitas outras classes em decorrência das telas. Para cada uma delas foi criada uma classe de controle respectiva que faz basicamente o mesmo que a *control*, porém agora através da interface. Por exemplo, se na tela de cadastro o usuário aperta enter na opção cliente, a classe *controleTelaCadastro* do CCI chama a classe *TelaCliente* que está no CIH e a partir daí o usuário será redirecionado para a tela de cadastro do cliente e poderá fazer o que deseja.

- UTIL

No pacote Util fizemos uma modificação nas classes do Método Fábrica. Na primeira parte, o código possuía uma fábrica para cada classe do nosso diagrama (*pessoa* – *fabricaPessoa*, *produto* – *fabricaProduto*...). Refatorando o código, deixamos apenas uma classe *fabricaGeral* que possui o método criar e a *flyweightFactory* para a aplicação do padrão Flyweight.

- APPLICATION

Já no pacote Application está a classe *main*, que cria os objetos DAO e as tabelas necessárias. A partir do main, há o gerenciamento das telas e todo o programa é executado.

- ARQUIVOS

Neste pacote estão todos os arquivos de entrada necessários caso o usuário queira fazer o backup do banco de dados. Além deles, os relatórios gerados também são salvos neste pacote.

- TESTE

Esta classe, como o nome já diz, é para aplicar o TDD (Desenvolvimento Orientado a Testes). Para cada funcionalidade do sistema é criado um teste antes. Essa boa prática garante um software com código mais limpo e coeso. Em nosso trabalho, por exemplo, fazemos testes nos atributos de pessoa, produto, fornecedor etc e também para testar o resultado dos relatórios.

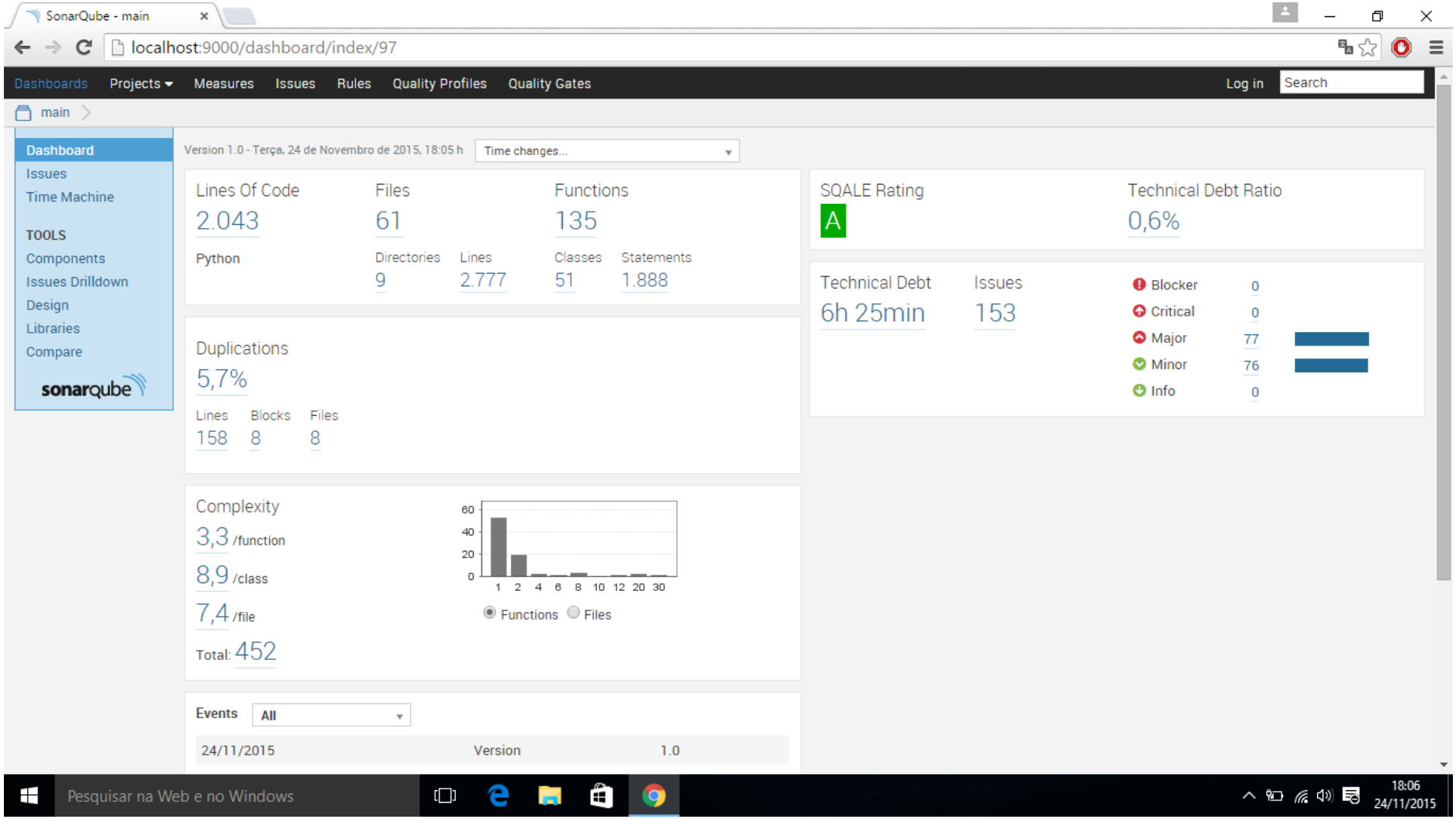

SonarQube - main
localhost:9000/dashboard/index/97
Dashboards
Projects
Measures
Issues
Rules
Quality Profiles
Quality Gates
Log in
Search
main
Dashboard
Issues
Time Machine
TOOLS
Components
Issues Drilldown
Design
Libraries
Compare
sonarqube
Version 1.0 - Terça, 24 de Novembro de 2015, 18:05 h
Time changes...
Lines Of Code
2.043
Python
Files
61
Directories
9
Lines
2.777
Functions
135
Classes
51
Statements
1.888
SQALE Rating
A
Technical Debt Ratio
0,6%
Technical Debt
6h 25min
Issues
153
Blocker 0
Critical 0
Major 77
Minor 76
Info 0
Duplications
5,7%
Lines
158
Blocks
8
Files
8
Complexity
3,3 /function
8,9 /class
7,4 /file
Total: 452
60
40
20
0
1 2 4 6 8 10 12 20 30
Functions
Files
Events
All
24/11/2015
Version
1.0
Pesquisar na Web e no Windows
18:06
24/11/2015

Figura 1

Com a criação de novas classes, é visível que a quantidade de linhas de código, funções e complexidade aumentaram relativamente. A complexidade por função passou de 2,0 para 3,3. Em contrapartida, a complexidade por classe e por arquivo caíram. No total, houve o aumento de 192 para 452, o que não é bom. O campo de Technical Debt, também nos desapontou de 1h 41min para 6h 45 min. A quantidade de erros subiu bastante, de 10 para 153. Entretanto, este último é justificável, pois os erros *Majors* e *Minors* não foram tratados até o momento. Esperamos reduzir bastante depois que isso for feito.

Um fato que nos chamou atenção é que, como explicado no relatório acima, antigamente, na camada CCI existia apenas uma classe control – que recebia o input do usuário e chamava as classes responsáveis pelo que ele queria fazer (cadastrar, gerar relatório...). Por conta da interface, na CIH tivemos que criar várias classes representando uma tela distinta. Consequentemente, no CCI, precisamos então criar uma classe que controlasse cada uma dessas telas (por exemplo, para a classe *telaCadastro* existe a correspondente *controleTelaCadastro*. Assim como para a *telaRelatorio* existe a *controleTelaRelatorio*). Além de aumentar a complexidade pelo fato de serem criadas muitas classes, em cada uma delas existem muitos comandos condicionais, que mudam a posição da seta na interface, chamam a tela respectiva ao comando do usuário, etc. O Sonar acusou justamente uma complexidade muito alta nessas classes, como pode ser visto na Figura 2.

Nas classes de controle também foi onde houve duplicação de código (Figura 3), talvez por todas elas precisarem fazer coisas iguais, como ler do teclado os inputs, atualizarem e escreverem na tela. Essa duplicação é algo que deve ser tratada e a complexidade diminuída. Realmente o código precisa da diminuição dos comandos condicionais e aninhados.

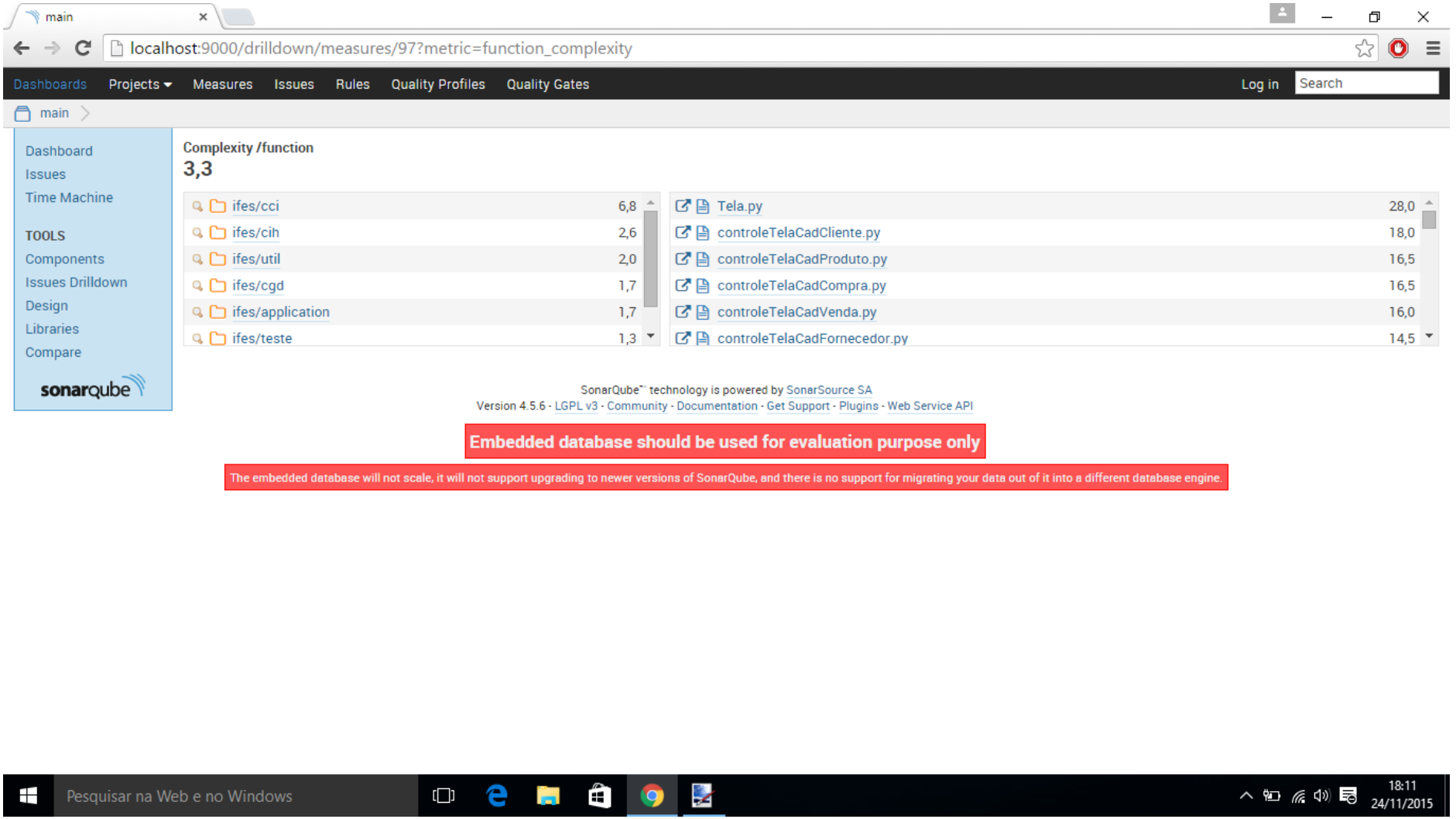

main
localhost:9000/drilldown/measures/97?metric=function_complexity
Dashboards
Projects
Measures
Issues
Rules
Quality Profiles
Quality Gates
Log in
Search
main
Dashboard
Issues
Time Machine
TOOLS
Components
Issues Drilldown
Design
Libraries
Compare
sonarqube
Complexity /function
3,3
ifes/cci 6,8
ifes/cih 2,6
ifes/util 2,0
ifes/cgd 1,7
ifes/application 1,7
ifes/teste 1,3
Tela.py 28,0
controleTelaCadCliente.py 18,0
controleTelaCadProduto.py 16,5
controleTelaCadCompra.py 16,5
controleTelaCadVenda.py 16,0
controleTelaCadFornecedor.py 14,5
SonarQube™ technology is powered by SonarSource SA
Version 4.5.6 - LGPL v3 - Community - Documentation - Get Support - Plugins - Web Service API
Embedded database should be used for evaluation purpose only
The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.
Pesquisar na Web e no Windows
18:11
24/11/2015

Figura 2

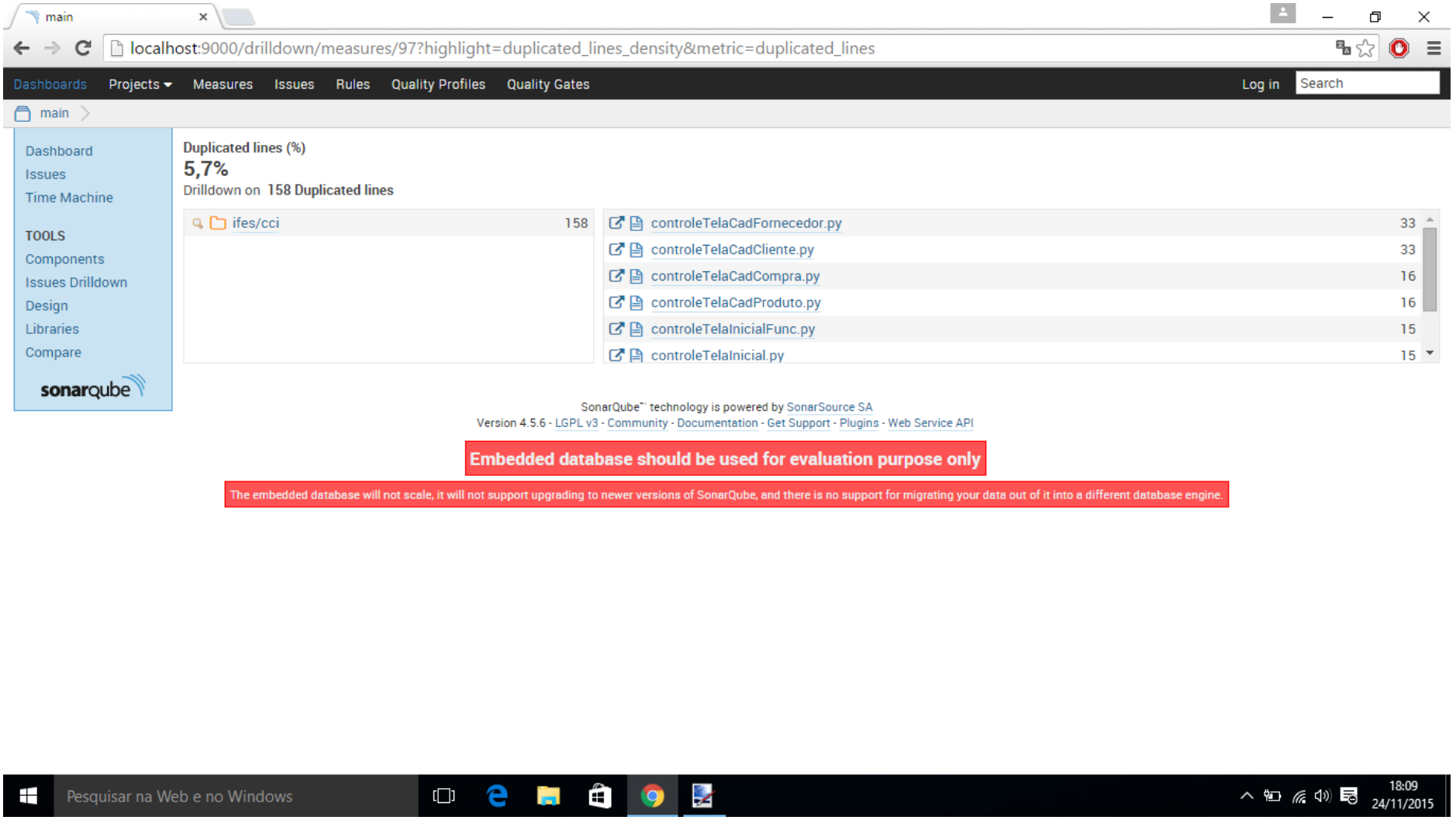

main
localhost:9000/drilldown/measures/97?highlight=duplicated_lines_density&metric=duplicated_lines
Dashboards
Projects
Measures
Issues
Rules
Quality Profiles
Quality Gates
Log in
Search
main
Dashboard
Issues
Time Machine
TOOLS
Components
Issues Drilldown
Design
Libraries
Compare
sonarqube
Duplicated lines (%)
5,7%
Drilldown on 158 Duplicated lines
ifes/cci 158
controleTelaCadFornecedor.py 33
controleTelaCadCliente.py 33
controleTelaCadCompra.py 16
controleTelaCadProduto.py 16
controleTelaInicialFunc.py 15
controleTelaInicial.py 15
SonarQube™ technology is powered by SonarSource SA
Version 4.5.6 - LGPL v3 - Community - Documentation - Get Support - Plugins - Web Service API
Embedded database should be used for evaluation purpose only
The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.
Pesquisar na Web e no Windows
18:09
24/11/2015

Figura 3.