Sistema 4S

Simple Store, Sale System

Introdução **3**

Implementação **3**

Lista de arquivos 3

Fluxograma Usuário (Operador de Venda) 4

Fluxograma Administrador 5

Módulos 7

UiModule 7

AdminModule 8

Common 8

DataBase 9

OrderModule 12

UserModule 12

Introdução

Documento tem por finalidade a documentação do código do sistema 4S, como ele está estruturado, auxiliando no entendimento de sua estrutura.

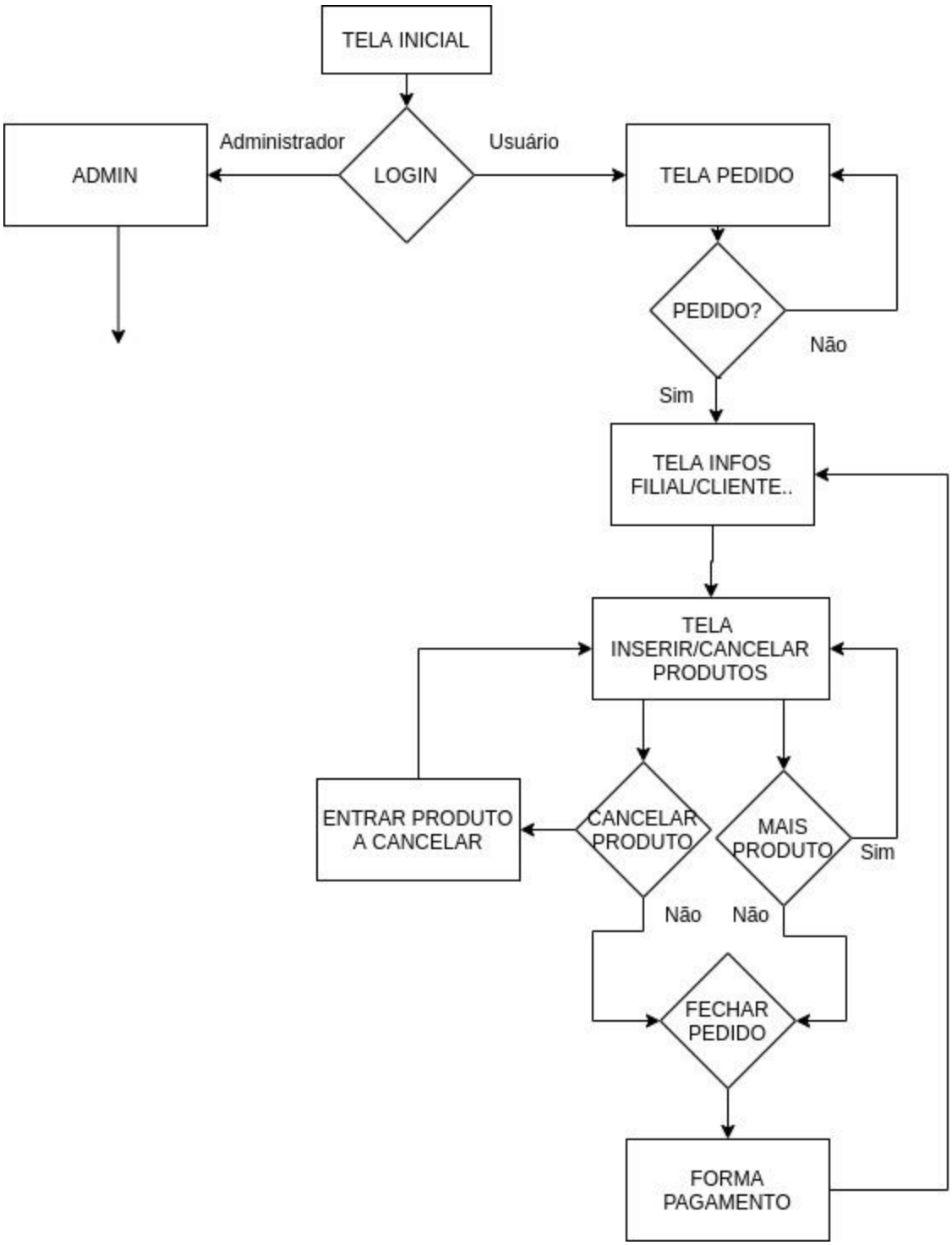
Implementação

Lista de arquivos

AdminModule.cpp
AdminModule.h
build
CMakeLists.txt
Common.cpp
Common.h
DataBase.cpp
DataBase.h
InitialWindow.cpp
InitialWindow.h
main.cpp
ManagerWindow.cpp
ManagerWindow.h
OrderModule.cpp
OrderModule.h
README.md
UserModule.cpp
UserModule.h

Fluxograma Usuário (Operador de Venda)

Figura 1 - Fluxograma de modo usuário



Fluxograma Administrador

Figura 2 - Tela inicial - Modo administrador

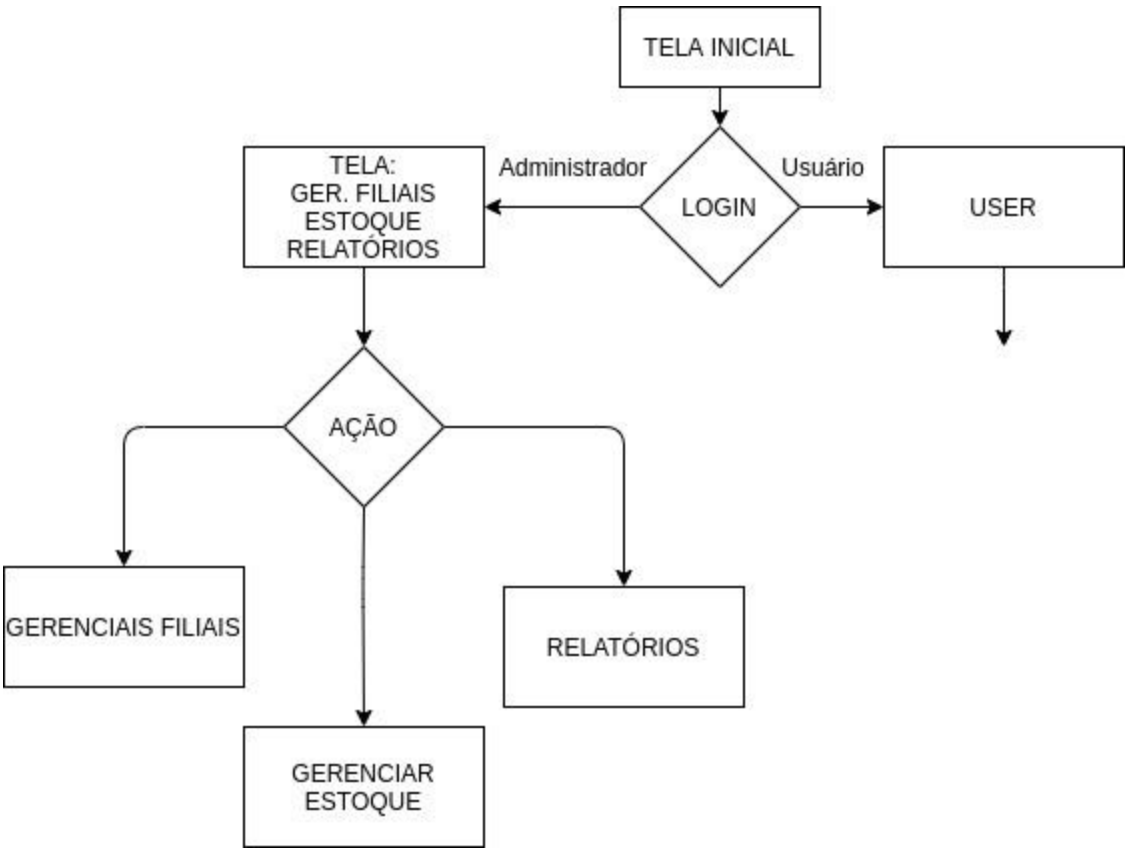


Figura 3 - Cadastro de filiais

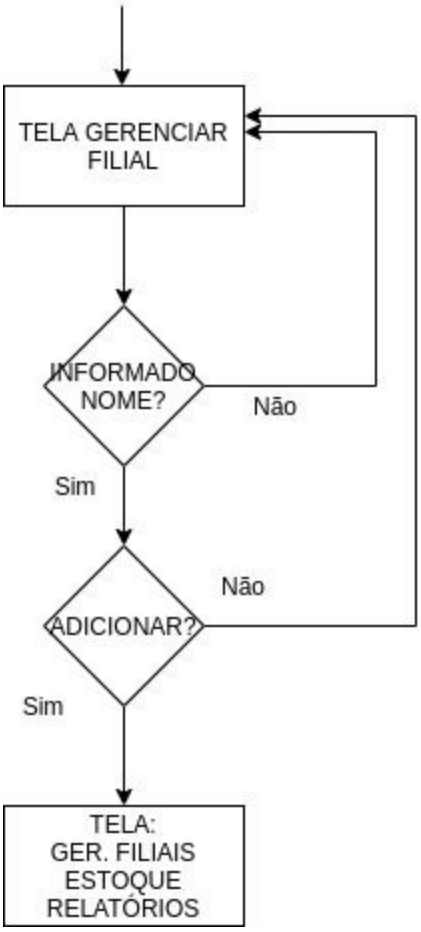
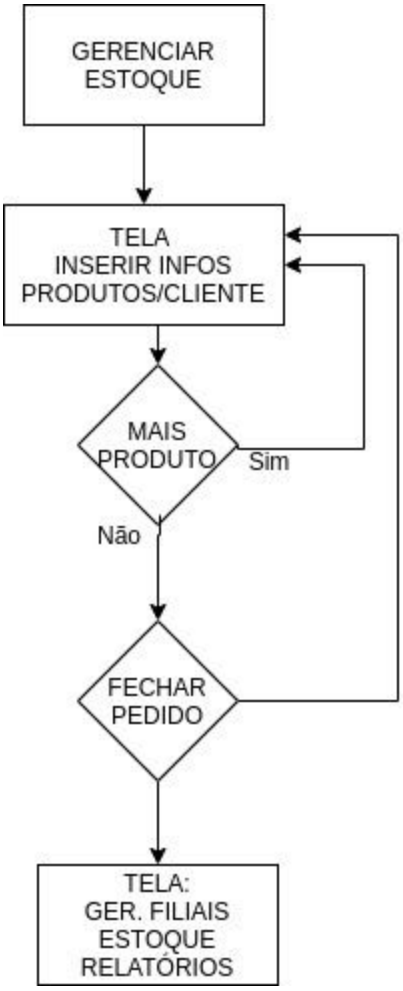


Figura 4 - Cadastro de novos itens



Módulos

UiModule

Esse é um módulo implícito. O ideal seria criar um micro-serviço que comunicasse com a aplicação *core*, via (REST ou DBUS dependendo da plataforma), em que o core enviaria para o outro serviço as informações (JSON) de:

- Qual tela mostrar;
- Que dados popular;

Para esse tipo de aplicação também trocaria a tecnologia para Python e no ambiente gráfico utilizaria Kivy, que tem um design mais moderno.

No entanto, esse módulo está junto ao core, e a tecnologia utilizada é Qt, o que implica uma certa mistura de funções nos métodos.

AdminModule

Módulo com as implementações de cadastro de filiais, estoque e relatórios.

O ciclo básico desse módulo é permanecer em uma tela, até que alguma ação seja tomada.

Cada tela é montada com específicos Widget, portanto, sempre ao mudar de tela, a anterior é destruída, para que a nova seja apresentada. Em caso de finalizar a operação ou de retornar para a tela anterior, novamente a atual é destruída.

Assim, foi criado o conceito de pre_SOMETASK, que consiste em: quando o usuário clica em um botão, que acionará a mudança de tela, um método pré-tela é chamado para destruir a anterior. Ex:

```
void AdminModule::pre_ResumeSperlativeManagement_clickedSlot()
{
	destroyOptionsGeneralListManagementScreen();
	ResumeSuperlativeManagement_clickedSlot();
}
void AdminModule::destroyOptionsGeneralListManagementScreen()
{
	mGridLayout->removeWidget(listManagPaymentButton);
	mGridLayout->removeWidget(resumeSperlativeButton);
	...
	delete listManagPaymentButton;
	delete resumeSperlativeButton;
}
```

Common

Esse módulo foi criado apenas para manter funções comumente utilizada em vários módulos, evitando assim repetição de código. Ex:

```
int isSettedVariable(QString var, QString msgErr);
```

Esse método verifica se a variável “*var*” está setada, caso contrário, é mostrado a mensagem “*msgErr*”.

DataBase

O banco de dados utilizado nessa aplicação é o SQLite.

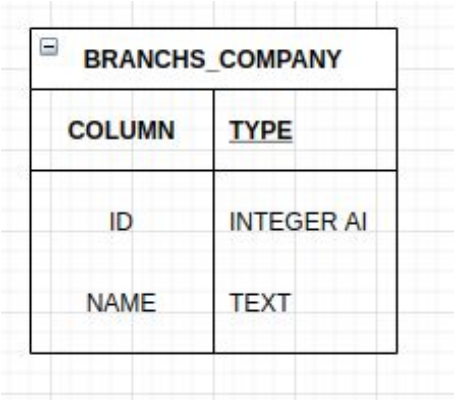
O módulo do banco de dados contém os protótipos, métodos e querys que permitem a criação, persistência e recuperação de informações.

O banco de dados é o arquivo "*database.db*" localizado no diretório de execução do software.

Ao instanciar a classe DataBase, caso esse banco de dados não exista, ele será criado "*limpo*". Além disso, quando instanciada, essa classe é do tipo "***singleton***", fazendo que toda tentativa de nova instanciação retorna o mesmo ponteiro.

É criado uma tabela contendo os nomes das filiais.

Figura 5 - Tabela de filiais

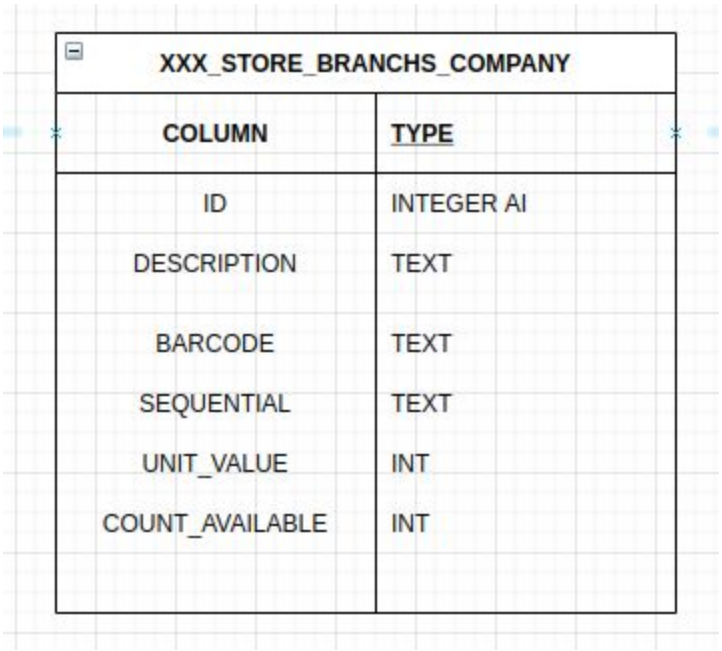


No processo de criação de filial é criado, para cada nova filial, 3 tabelas:

- Estoque (XXX_STORE_BRANCHS_COMPANY). Fig. 6;
- Pedidos (XXX_ORDERS_BRANCHS_COMPAN). Fig. 7;
- Itens do Pedido (XXX_ORDERS_PRODUCTS_BRANCHS_COMPANY). Fig. 8;

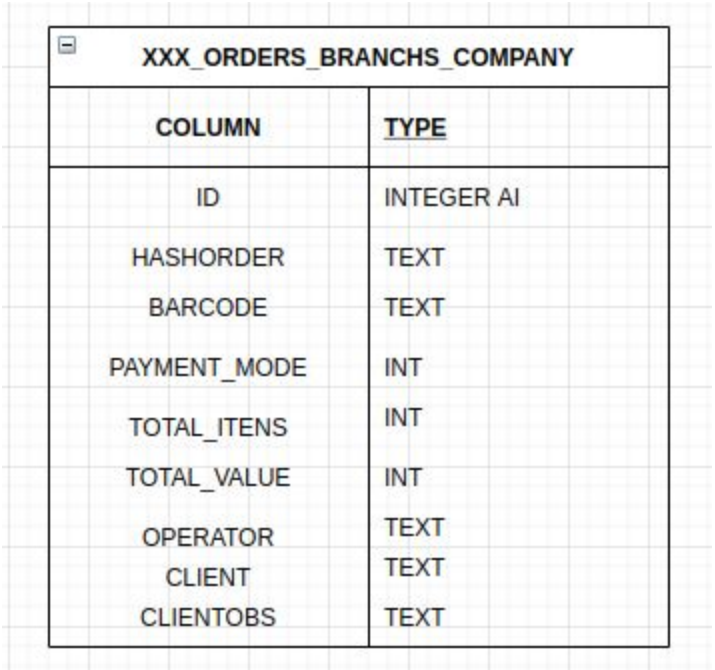
void DataBase::createBranchCompanyTable(string branch_name)

Figura 6 - Tabela de estoque da filial XXX



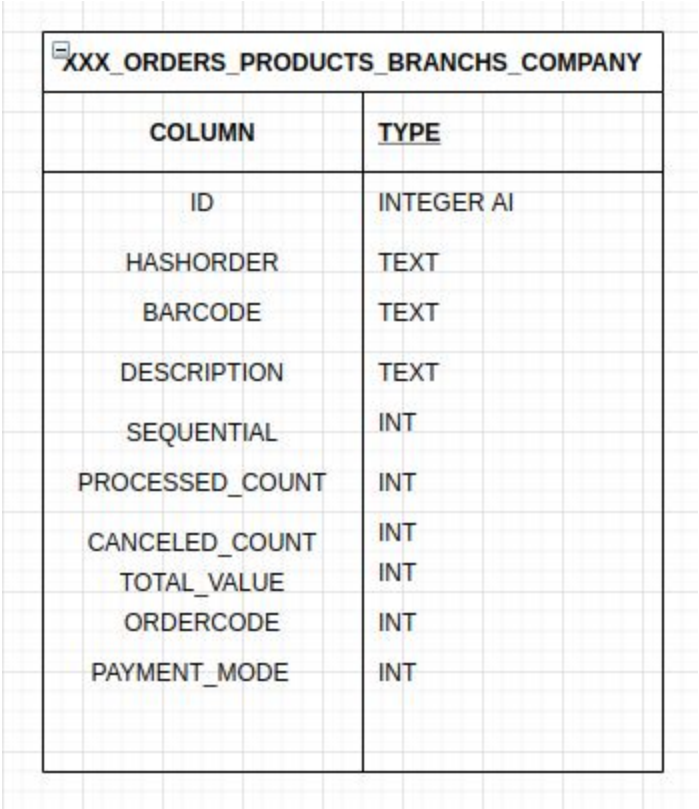
XXX_STORE_BRANCHS_COMPANY	
COLUMN	**TYPE**
ID	INTEGER AI
DESCRIPTION	TEXT
BARCODE	TEXT
SEQUENTIAL	TEXT
UNIT_VALUE	INT
COUNT_AVAILABLE	INT

Figura 7 - Tabela de pedidos da filial XXX



XXX_ORDERS_BRANCHS_COMPANY	
COLUMN	**TYPE**
ID	INTEGER AI
HASHORDER	TEXT
BARCODE	TEXT
PAYMENT_MODE	INT
TOTAL_ITENS	INT
TOTAL_VALUE	INT
OPERATOR	TEXT
CLIENT	TEXT
CLIENTOBS	TEXT

Figura 8 - Tabela de Itens dos pedidos



XXX_ORDERS_PRODUCTS_BRANCHS_COMPANY

COLUMN	TYPE
ID	INTEGER AI
HASHORDER	TEXT
BARCODE	TEXT
DESCRIPTION	TEXT
SEQUENTIAL	INT
PROCESSED_COUNT	INT
CANCELED_COUNT	INT
TOTAL_VALUE	INT
ORDERCODE	INT
PAYMENT_MODE	INT

Para cada busca em uma filial, é buscado o nome da mesma, de forma a montar corretamente os nomes das tabelas de produtos (PRODUCTS), pedidos (ORDERS) e itens (PRODUCTS_ORDER)

As querys de busca dinâmica dentro do sistema, possuem estruturas básicas, que são preenchidas com as variáveis corretas em tempo de execução. Ex:

```
"SELECT * FROM '%s_STORE_BRANCHS_COMPANY' WHERE %s == '%s';
```

Nesse exemplo, é esperado três parâmetros para que a query possa ser executada, e todos os três são do tipo string.

Para busca de dados complexos, como por exemplo lista de produtos ou itens, é utilizado como armazenador a estrutura hashtable. Ex:

```
QHash<QString, Product *> HTProducts;
QHash<QString, ProductOfOrder *> HTProductOfOrder;
```

No exemplo acima, as hashtable possuem como chave uma QString e como valor ponteiros de classes (*Product *, ProductOfOrder *)* . Esse tipo de estrutura de armazenamento facilita a iteração sobre o mesmo, otimizando esse processo.

Há também a utilização de estrutura de armazenamento mais simples como a QList, quando é preciso buscar apenas uma lista de string. Ex:

```
QList<QString> * DataBase::searchOrdersHashofBranch(string branch)
```

Neste método, é retornado uma QList com todas os pedidos de uma filial.

OrderModule

Módulo com classes de produtos (PRODUCTS), pedidos (ORDERS) e itens (PRODUCTOFORDER).

A função principal desse módulo é fazer o *parser* dos dados de cada domínio oriundo dos módulos que estão interagindo com usuário, como UserModule e AdminModule, inserindo essas requisições no banco de dados.

UserModule

Módulo que implementa as funcionálidades de vendas, realizadas pelo operador.

O ciclo básico desse módulo é permanecer em uma tela, até que alguma ação seja tomada.

Cada tela é montada com específicos Widget, portanto, sempre ao mudar de tela, a anterior é destruída, para que a nova seja apresentada. Em caso de finalizar a operação ou de retornar para a tela anterior, novamente a atual é destruída.

Assim, foi criado o conceito de pre_SOMETASK, que consiste em: quando o usuário clica em um botão, que acionará a mudança de tela, um método pré-tela é chamado para destruir a anterior. Ex:

```
void UserModule::returnProcessingOrder()
{
	destroyOrder();
	Execute();
}

void UserModule::destroyOrder()
{

	mGridLayout->removeWidget(returnButton);
	mGridLayout->removeWidget(searchModeLabel);

	…

	delete searchModeLabel;
	delete returnButton;
}
```

Nesse caso, o usuário clicou no ***PushButton*** para retornar e o method de destruir é chamada antes da nova tela.