**UNIVERSIDADE FEDERAL DO PARÁ**

**INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS**

**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

ESTHER CARDOSO DA SILVA

**PROGRAMAÇÃO II:**

REQUISITOS DO TRABALHO FINAL

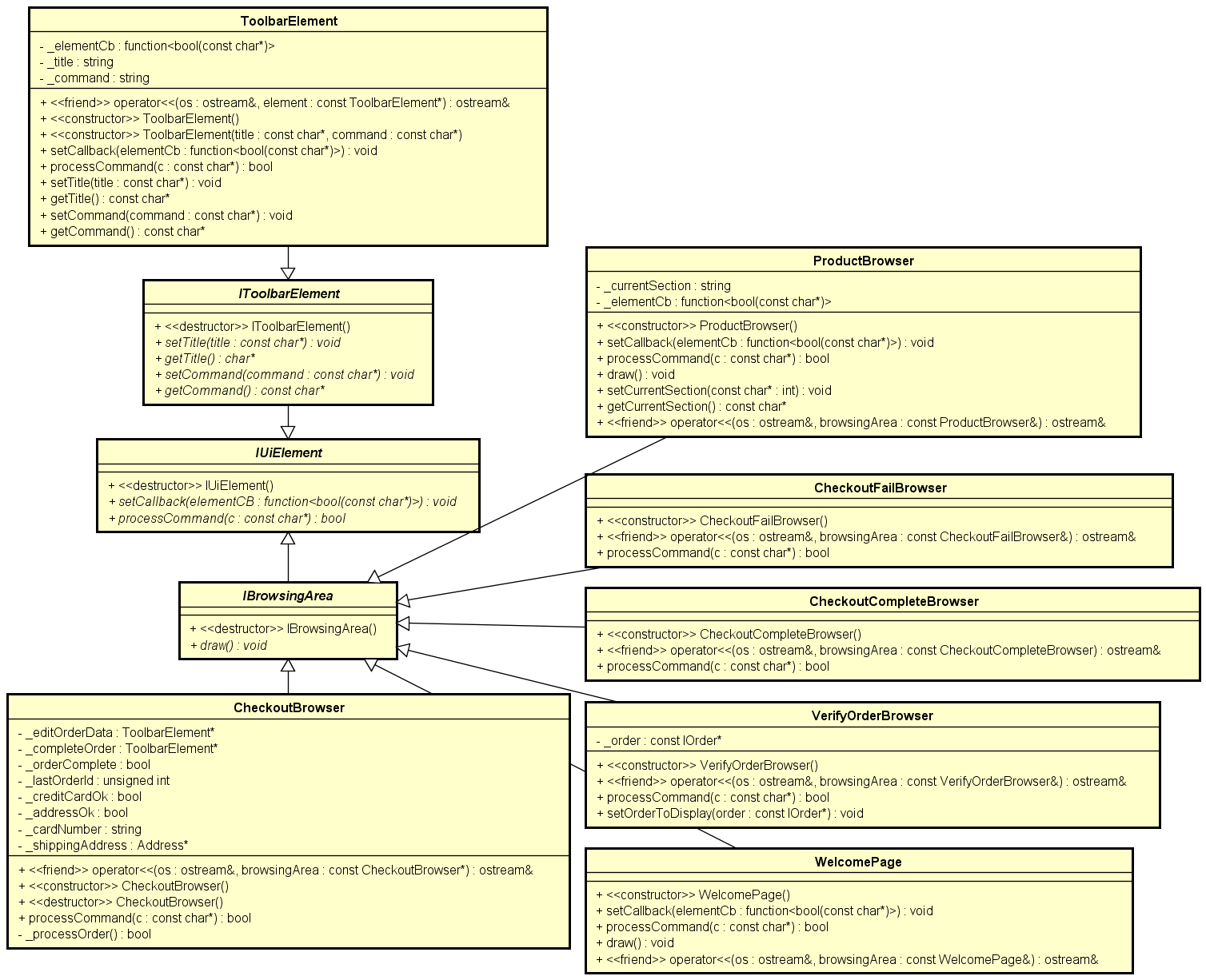
BELÉM

2015

1. **Todas as classes concretas devem vir de classes abstratas. Pelo menos três hierarquias de classes. Uma das hierarquias deve ter três níveis.**

Com exceção da classe *Manager*, todas as classes concretas vêm de classes abstratas com todos os métodos virtuais.

Diagrama das hierarquias de 3 níveis:



1. **Em todas as classes: construtor de cópia, operatores<< e +=, e construtor default. Fazer o máximo de reaproveitamento de código usando *static\_cast*.**

Classe *CartBrowser*:

ostream& operator<<(ostream& os, const CartBrowser\* browsingArea)

{

os << "Products in cart:" << endl << endl;

vector< pair<IProduct\*, unsigned int>> productsInCart =

Manager::instance().productDataManager()->getProductsInCart();

if (productsInCart.empty())

{

os << "There are no items in your cart right now." << endl << endl;

}

else

{

for (unsigned int i = 0; i < productsInCart.size(); i++)

{

pair<IProduct\*, unsigned int> p = productsInCart.at(i);

os << p.first->getName() << endl

<< "Quantity: " << p.second << endl

<< "Price: US$ " << p.first->getDiscountedPrice() \* p.second << endl

<< "[" << i + 1 << "] Remove from cart." << endl << endl;

}

}

os << "Order total: US$ " << Manager::instance().productDataManager()->getTotalCartValue() << endl

<< endl << browsingArea->\_checkout;

return os;

}

Classe *CheckoutBrowser*:

ostream& operator<<(ostream& os, const CheckoutBrowser\* browsingArea)

{

os << "Payment information:" << endl << endl

<< "Credit card number: " << browsingArea->\_cardNumber << endl << endl

<< "Address information:" << endl << endl

<< browsingArea->\_shippingAddress << endl << endl

<< browsingArea->\_editOrderData << endl << endl;

if (browsingArea->\_creditCardOk && browsingArea->\_addressOk)

{

os << browsingArea->\_completeOrder;

}

else

{

os << "Please enter your credit card and address information to proceed.";

}

return os;

}

Classe *CheckoutCompleteBrowser*:

ostream& operator<<(ostream& os, const CheckoutCompleteBrowser\* browsingArea)

{

os << "Thank you! Your order has been placed!" << endl << endl << endl

<< "Order details can be verified on the Verify Orders page." << endl << endl

<< "Your order Id: " << Manager::instance().orderManager()->getLastOrderId() << endl << endl

<< "[H] Return to Home.";

return os;

}

Classe *CheckoutFailBrowser*:

ostream& operator<<( ostream& os, const CheckoutFailBrowser\* browsingArea)

{

os << "Sorry, your order could not be placed." << endl << endl

<< "The transaction has been declined by your bank." << endl << endl

<< "Please enter another credit card and try again." << endl << endl

<< "[C] Return to Cart." << endl << endl

<< "[H] Return to Home.";

return os;

}

Classe *ProductBrowser*:

ostream& operator<<(ostream& os, const ProductBrowser\* browsingArea)

{

string section = browsingArea->getCurrentSection();

vector<IProduct\*> sectionProducts = Manager::instance().productDataManager()->getProductsForSection(section.c\_str());

os << "List of products in section " << section << ":"

<< endl << endl;

for (unsigned int i = 0; i < sectionProducts.size(); i++)

{

IProduct\* p = sectionProducts.at(i);

os << endl << dynamic\_cast<Product\*>(p) << endl

<< "Items in stock: " << p->getItemStock() << endl;

// Removes the "add to cart" command label if there are no items in stock.

if (p->getItemStock() > 0)

{

os << "[" << i + 1 << "] Add to cart." << endl;

}

else

{

os << "Item currently unavailable." << endl;

}

}

return os;

}

Classe *VerifyOrderBrowser*:

ostream& operator<<( ostream& os, const VerifyOrderBrowser\* browsingArea)

{

if (browsingArea->\_order != nullptr)

{

os << "Details for order " << browsingArea->\_order->getOrderId() << ":" << endl << endl

<< dynamic\_cast<const Order\*>(browsingArea->\_order);

}

else

{

os << "There are no orders with the provided Order Id or it is invalid." << endl << endl

<< "[V] Try again." << endl << endl

<< "[H] Return to Home.";

}

return os;

}

Classe *WelcomePage*:

ostream& operator<<(

ostream& os, const WelcomePage\* browsingArea)

{

return os << endl

<< " Welcome!"

<< endl << endl

<< "Please choose a store section to begin."

<< endl << endl;

}

Classe *ToolbarElement*:

ostream& operator<<(ostream& os, const ToolbarElement& element)

{

return os << "[" << element.getCommand() << "] " << element.getTitle();

}

ostream& operator<<( ostream& os, const ToolbarElement\* element)

{

return os << "[" << element->getCommand() << "] " << element->getTitle();

}

Classe *Product*:

ostream& operator<<(ostream& os, const Product\* product)

{

// Display only two decimal digits.

os << setprecision(2) << fixed;

os << product->getName() << " (" << product->getManufacturer() << ")" << endl

<< product->getDescription() << endl

<< "Price: US$ " << product->getDiscountedPrice();

// Discount information is only added if product is on sale.

if (product->onSale())

{

os << " (" << (product->getDiscount()) \* 100.0f << "% OFF of US$ " << product->getPrice() << ")";

}

return os;

}

Classe *Address*:

ostream& operator<<( ostream& os, const Address\* address)

{

os << "Address Line 1: " << address->getAddressLine1() << endl

<< "Address Line 2: " << address->getAddressLine2() << endl

<< "City: " << address->getCity() << endl

<< "State/Province: " << address->getState() << endl

<< "ZIP Code: " << address->getZip();

return os;

}

Classe *Order*:

Operator<<:

ostream& operator<<( ostream& os, const Order\* o)

{

os << "Order date: " << o->getDate() << endl

<< "Products in this order:" << endl;

vector<const IProduct\*> products = o->getItems();

for (unsigned int i = 0; i < products.size(); i++)

{

os << endl

<< dynamic\_cast<const Product\*>(products.at(i)) << endl;

}

os << endl

<< "Order value: US$" << o->getTotal() << endl;

os << endl

<< "Shipping address:" << endl << endl

<< dynamic\_cast<const Address\*>(o->getShippingAddress());

return os;

}

Operator+=:

1. **Todas as hierarquias devem ter classes Concretas, e em uma das hierarquias, três classes Concretas relacionadas.**

As hierarquias são: (*Classes Concretas*)

* *Iapplication → Application*;
* *IMainWindow → MainWindow*;
* *IUiElement → BrowsingArea → [Browsers]*;
* *IUiElement → IToolbarElement → ToolbarElement*;
* *IOrderManager → OrderManager*;
* *IProductDataManager → ProductDataManager*;
* *IProduct → Product*;
* *ISection → Section*;
* *IAddress → Address*;
* *IOrder → Order*;

1. **Atributos *static* e *const static* em todas as hierarquias de classe.**

Atributos *static* nas classes *Product* e *Order*:

static unsigned int s\_globalId;

Atributos *const static* na classe *MainWindow:*

const static string s\_windowSeparator;

const static string s\_sectionSeparator;

1. **Método *static* em todas as hierarquias de classe.**

Método *static* na classe *Manager:*

// Returns the global Manager instance.

static Manager& instance()

{

static Manager instance;

return instance;

}

1. **Construtores em todas as classes, e três para todas as classes da hierarquia principal.**

Construtor default: todas as classes abstratas.

Construtor vazio: *CheckoutCompleteBrowser*, *CheckoutFailBrowser*, *ProductBrowser*, *WelcomePage* e *ProductDataManager* .

Classe *CartBrowser*:

CartBrowser::CartBrowser()

{

// Checkout

\_checkout = new ToolbarElement("Checkout", "T");

std::function<bool(const char\*)> checkoutCb = [](const char\* c)

{

if (Manager::instance().productDataManager()->getProductsInCart().size() > 0)

{

Manager::instance().mainWindow()->setCurrentBrowsingArea(PageType::CheckoutPage);

}

return true;

};

\_checkout->setCallback(checkoutCb);

}

Classe *CheckoutBrowser*:

CheckoutBrowser::CheckoutBrowser()

{

\_cardNumber = "";

\_shippingAddress = new Address();

\_creditCardOk = false;

\_addressOk = false;

\_editOrderData = new ToolbarElement("Edit credit card and address", "E");

std::function<bool(const char\*)> orderDataCb = [this](const char\*)

{

bool dataOk = false;

// Get credit card number.

\_creditCardOk = false;

\_cardNumber = "";

while (!dataOk)

{

dataOk = true;

std::cout << std::endl

<< "Please enter your credit card number ([B] to abort): ";

std::string cardNumber;

std::cin >> cardNumber;

if (strcmp(cardNumber.c\_str(), "B") == 0)

{

return true;

}

// Check if string lenght is valid.

if (cardNumber.length() != 16)

{

dataOk = false;

std::cout << std::endl

<< "Invalid card number." << std::endl;

}

// Check if provided string is a number

if (!std::all\_of(cardNumber.begin(), cardNumber.end(), ::isdigit))

{

dataOk = false;

std::cout << std::endl

<< "Invalid card number." << std::endl;

}

if (dataOk)

{

\_creditCardOk = true;

\_cardNumber = cardNumber;

}

}

dataOk = false;

// Get address data.

\_addressOk = false;

delete \_shippingAddress;

std::string addressLine1, addressLine2, city, state, zipCode;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << std::endl

<< "Please enter your address line 1: ";

std::getline(std::cin, addressLine1);

std::cout << std::endl

<< "Please enter your address line 2: ";

std::getline(std::cin, addressLine2);

std::cout << std::endl

<< "Please enter your city: ";

std::getline(std::cin, city);

std::cout << std::endl

<< "Please enter your state/province: ";

std::getline(std::cin, state);

std::cout << std::endl

<< "Please enter your ZIP Code: ";

std::getline(std::cin, zipCode);

\_addressOk = true;

\_shippingAddress = new Address(addressLine1.c\_str(),

addressLine2.c\_str(),

city.c\_str(),

state.c\_str(),

zipCode.c\_str());

return true;

};

\_editOrderData->setCallback(orderDataCb);

\_completeOrder = new ToolbarElement("Complete order", "O");

std::function<bool(const char\*)> completeCb = [this](const char\*)

{

if (\_creditCardOk && \_addressOk && \_processOrder())

{

// Order date.

std::time\_t t = std::time(0);

char d[30];

ctime\_s(d, sizeof(d), &t);

std::string date = d;

float orderValue = Manager::instance().productDataManager()->getTotalCartValue();

IOrder\* order = new Order(date.c\_str(), new Address(\*\_shippingAddress), orderValue);

// Add products to order.

std::vector<std::pair<IProduct\*, unsigned int>> orderProducts = Manager::instance().productDataManager()->getProductsInCart();

for (unsigned int i = 0; i < orderProducts.size(); i++)

{

// TODO: Update stock amounts after this order.

IProduct\* p = orderProducts.at(i).first;

order->addItem(p);

}

// Add it to the order manager.

Manager::instance().orderManager()->addOrder(order);

// Clear cart.

Manager::instance().productDataManager()->removeAllItemsFromCart();

// Update the browsing area with the checkout complete page.

Manager::instance().mainWindow()->setCurrentBrowsingArea(PageType::CheckoutCompletePage);

}

else

{

// Update the browsing area with the checkout failed page.

Manager::instance().mainWindow()->setCurrentBrowsingArea(PageType::CheckoutFailPage);

}

\_cardNumber = "";

\_shippingAddress->setAddressLine1("");

\_shippingAddress->setAddressLine2("");

\_shippingAddress->setCity("");

\_shippingAddress->setState("");

\_shippingAddress->setZip("");

\_creditCardOk = false;

\_addressOk = false;

return true;

};

\_completeOrder->setCallback(completeCb);

}

Classe *VerifyOrderBrowser*:

VerifyOrderBrowser::VerifyOrderBrowser()

{

\_order = nullptr;

}

Classe *ToolbarElement*:

ToolbarElement::ToolbarElement()

{

\_title = "No title";

\_command = "";

\_elementCb = nullptr;

}

ToolbarElement::ToolbarElement(const char\* title, const char\* command)

{

assert(title);

assert(command);

\_title = title;

\_command = command;

\_elementCb = nullptr;

}

Classe *Manager*:

Manager::Manager()

{

\_orderManager = new OrderManager();

\_productDataManager = new ProductDataManager();

\_mainWindow = new MainWindow();

}

Classe *OrderManager*:

OrderManager::OrderManager()

{

\_lastOrderId = std::numeric\_limits<unsigned int>::max();

}

Classe *Product*:

Product::Product(const char\* name, const char\* description, const char\* manufacturer, float initialPrice, unsigned int initialStock)

{

assert(name != nullptr);

assert(description != nullptr);

assert(manufacturer != nullptr);

\_name = name;

\_description = description;

\_manufacturer = manufacturer;

\_price = initialPrice;

\_itemStock = initialStock;

\_onSale = false;

\_discount = 0.0f;

\_productId = s\_globalId;

s\_globalId++;

}

// Forbidden methods and operators to enforce strict creation of products with necessary data.

Product() = delete;

Product(const Product& p) = delete;

Classe *Section*:

Section::Section(const char\* name)

{

assert(name != nullptr);

\_name = name;

\_onSale = false;

}

// Forbidden methods and operators to enforce strict creation of sections with necessary data.

Section() = delete;

Section(const Section& p) = delete;

Classe *Address*: \*Três construtores

Address::Address()

{

\_addressLine1 = "";

\_addressLine2 = "";

\_city = "";

\_state = "";

\_zip = "";

}

Address::Address(const char\* addressLine1, const char\* addressLine2, const char\* city, const char\* state, const char\* zip)

{

assert(addressLine1);

assert(addressLine2);

assert(city);

assert(state);

assert(zip);

\_addressLine1 = addressLine1;

\_addressLine2 = addressLine2;

\_city = city;

\_state = state;

\_zip = zip;

}

Address::Address(const Address& a)

{

\_addressLine1 = a.\_addressLine1;

\_addressLine2 = a.\_addressLine2;

\_city = a.\_city;

\_state = a.\_state;

\_zip = a.\_zip;

}

Classe *Order*:

Order::Order(const char\* orderDate, const IAddress\* shippingAddress, float orderValue)

{

assert(orderDate && std::strlen(orderDate) != 0);

assert(shippingAddress);

\_orderId = s\_globalId;

\_orderDate = orderDate;

\_shippingAddress = shippingAddress;

\_orderValue = orderValue;

s\_globalId++;

}

// Forbidden methods.

Order() = delete;

Order(const Order& o) = delete;

1. ***Vector* em todas em todas as hierarquias de classe.**

Vector na classe *Section*:

vector<IProduct\*> \_products;

Vector na classe *Order*:

vector<const IProduct\*> \_orderItems;

Vector na classe *ProductDataManager*:

vector<pair<IProduct\*, unsigned int>> \_cart;

vector<ISection\*> \_productSections;

Vector na classe *OrderManager*:

vector<const IOrder\*> \_orders;

Vector na classe *MainWindow*:

vector<IToolbarElement\*> \_toolbarElements;

1. ***ENUM* na hierarquia principal.**

*Enum* utilizado na classe *IMainWindow* para determinar os tipos de janelas.

enum PageType

{

HomePage,

ProductPage,

CartPage,

CheckoutPage,

CheckoutCompletePage,

CheckoutFailPage,

OrdersPage,

VerifyOrderPage

};

1. **Usar o *dynamic\_cast* e *typeid* na *main* junto com as classes concretas. Para uma da classe concreta identificada, chamar um método dessa classe e fazer uma ação.**

O *dynamic\_cast*  foi usado na clase *MainWindow*  para a decisão da classe a ser exibida na execução.

void MainWindow::draw()

{

cout << endl << endl

<< s\_windowSeparator

<< endl << endl;

// Toolbar buttons.

for (unsigned int i = 0; i < \_toolbarElements.size(); i++)

{

cout << dynamic\_cast<ToolbarElement\*>(\_toolbarElements.at(i)) << "\t";

}

cout << endl << endl

<< s\_sectionSeparator

<< endl << endl;

// Section list.

cout << dynamic\_cast<ToolbarElement\*>(\_sectionList);

cout << endl << endl

<< s\_sectionSeparator

<< endl << endl;

// Browsing area.

IBrowsingArea\* browsingArea = \_storePages.at(\_currentStorePage);

switch (\_currentStorePage)

{

case PageType::HomePage:

cout << dynamic\_cast<WelcomePage\*>(browsingArea);

break;

case PageType::ProductPage:

cout << dynamic\_cast<ProductBrowser\*>(browsingArea);

break;

case PageType::CartPage:

cout << dynamic\_cast<CartBrowser\*>(browsingArea);

break;

case PageType::CheckoutPage:

cout << dynamic\_cast<CheckoutBrowser\*>(browsingArea);

break;

case PageType::CheckoutCompletePage:

cout << dynamic\_cast<CheckoutCompleteBrowser\*>(browsingArea);

break;

case PageType::CheckoutFailPage:

cout << dynamic\_cast<CheckoutFailBrowser\*>(browsingArea);

break;

case PageType::VerifyOrderPage:

cout << dynamic\_cast<VerifyOrderBrowser\*>(browsingArea);

break;

default:

// Reset current page to Home.

\_currentStorePage = PageType::HomePage;

cout << dynamic\_cast<WelcomePage\*>(\_storePages.at(PageType::HomePage));

}

cout << endl << endl

<< s\_windowSeparator

<< endl << endl;

}

1. **Usar o *rand*.**

Como o método *rand* é impreciso e defasado, foi utilizada uma forma mais moderna de gerar números pseudo-aleatórios para gerar descontos por seção da loja.

void Section::setOnSale(bool onSale, float min, float max)

{

if (min > max || min < 0.0f ||

max < min || max > 1.0f )

{

return;

}

for (unsigned int i = 0; i < \_products.size(); i++)

{

IProduct\* p = \_products.at(i);

p->setOnSale(onSale);

// Apply randomly calculated discount in between the given interval.

if (onSale)

{

p->setDiscount(\_calculateDiscount(min, max));

}

else

{

p->setDiscount(0.0f);

}

}

}

// Modern C++ approach for generating random numbers within a range.

float Section::\_calculateDiscount(float min, float max)

{

random\_device rd; // Obtain a random number from hardware

mt19937 eng(rd()); // Seed the generator

uniform\_real\_distribution<> distr(min, max); // Define the range.

return static\_cast<float>(distr(eng));

}

1. **Na *main,* o usuário deve fazer entrada via teclado e interagir com a aplicação.**

A classe *Application* recebe comandos do teclado e distribui para a classe que sabe processar.

void Application::run()

{

while (true)

{

// Warning: Windows-only! Clears screen to make browsing less confusing.

system("cls");

Manager::instance().mainWindow()->draw();

std::cout << "[Q] Leave store." << std::endl << std::endl;

std::cout << "Please input store command: ";

std::string command;

std::cin >> command;

if (strcmp("Q", static\_cast<const char\*>(command.c\_str())) == 0)

{

return;

}

Manager::instance().mainWindow()->processCommand(static\_cast<const char\*>(command.c\_str()));

}

}

