Оглавление

[ВВЕДЕНИЕ 3](#_Toc41153861)

[1.АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 4](#_Toc41153862)

[1.1 Устройство склада 4](#_Toc41153863)

[1.2 Выбор языка 6](#_Toc41153864)

[1.3 Выбор среды разработки 6](#_Toc41153865)

[1.4 Постановка задачи 7](#_Toc41153866)

[2.РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА 8](#_Toc41153867)

[2.1 Разработка основного алгоритма программы 8](#_Toc41153868)

[2.2 Многопоточность 8](#_Toc41153869)

[2.2.1 Общие сведения 8](#_Toc41153870)

[2.2.2 Поток погрузчика 9](#_Toc41153871)

[2.2.3 Поток покупателя 12](#_Toc41153872)

[2.3 Реализация основных блоков кода 13](#_Toc41153873)

[2.3.1 Класс ConcurrentUtils 13](#_Toc41153874)

[2.3.2 Класс Customer 14](#_Toc41153875)

[2.3.3 Класс Loader 16](#_Toc41153876)

[2.3.4 Класс Storage 17](#_Toc41153877)

[2.4 Визуальная составляющая приложения 20](#_Toc41153878)

[2.4.1 Графический интерфейс 20](#_Toc41153879)

[2.4.2 Компоненты 21](#_Toc41153880)

[2.4.3 Формы 23](#_Toc41153881)

[3. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 26](#_Toc41153882)

[3.1 Концепция объектно-ориентированного программирования 26](#_Toc41153883)

[3.2 Принципы SOLID 28](#_Toc41153884)

[3.3 Паттерн декоратор 29](#_Toc41153885)

[3.3.1 Структура 29](#_Toc41153886)

[3.3.2 Класс LoggingLoaderDecorator 29](#_Toc41153887)

[3.3.3 Классы ViewModelCustomerDecorator и ViewModelLoaderDecorator 31](#_Toc41153888)

[ЗАКЛЮЧЕНИЕ 34](#_Toc41153889)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 35](#_Toc41153890)

# ВВЕДЕНИЕ

Симулятор — имитатор (обычно механический или компьютерный), задача которого состоит в имитации управления каким-либо процессом, аппаратом или транспортным средством.

Чаще всего сейчас слово «симулятор» используется применительно к компьютерным программам (обычно играм). С помощью компьютерно-механических симуляторов, абсолютно точно воспроизводящих интерьер кабины аппарата, тренируются пилоты, космонавты, машинисты высокоскоростных поездов.

Строительство и оснащение современных складов необходимыми оборудованием и техникой требуют значительных капиталовложений. Планирование склада — ключевой этап в этом процессе. Ведь ошибки, допущенные при планировании, могут сделать невозможным эффективное использования склада и привести к существенным финансовым потерям.

Симуляция позволяет с минимальным риском проверить различные гипотезы и получить результаты практически моментально, так как в симулятор может быть встроен движок реального времени, который можно ускорить или замедлить. Так, например, в симуляции 1 день может длиться всего несколько секунд реального времени.

Данный симулятор имитирует работу склада с сильно ограниченными ресурсами, а именно всего одним погрузчиком. С помощью этой программы можно понять, насколько быстро склад сможет обслужить данное количество клиентов.

# 1.АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## Устройство склада

Любое помещение для хранения должно быть оборудовано согласно принятым нормам. Особенности устройства складов напрямую зависят от задач, которые ставятся перед хозяйством. Здания для хранения грузов, передачи товаров от производителя к розничным реализаторам, переработки запасов будут различаться по конструктивным особенностям и предлагаемым технологиям складских операций.

Каждый склад должен обеспечивать необходимый режим хранения для определенного типа продукции, эффективную работу без простоев и перебоев, количественную и качественную сохранность оставленных товаров. Также любое хранилище должно обеспечивать нормальные условия труда для всех сотрудников.

К каждому зданию для хранения предъявляются требования нескольких групп:

* общетехнические — по освещению, вентиляции, отоплению, к конструктивным элементам;
* противопожарные — обеспечение безопасности и возможности оперативного тушения возгораний;
* технологические — соотношение площадей и объемов помещений и проводимых там технологических операций;
* экономические — соответствие бюджету при строительстве и эксплуатации зданий;
* санитарно-гигиенические — необходимые условия для труда всех сотрудников;
* эстетические — привлекательное оформление внутренней и внешней отделки.

Если складское помещение планируется построить, а не взять в аренду, нужно учесть требования к участку. Он должен быть ровным, расположенным на возвышении, сухим или с возможностью дренажа. К каждому участку должны быть обеспечены подходящие подъездные пути для всех используемых хранилищем видов транспорта (автомобили, железная дорога и т.д.). Для помещений любого класса должна быть возможность подведения необходимых для труда коммуникаций: канализация, водоснабжение, электричество.

Для оказания всех типов складских услуг, принятия грузов на ответственное хранение помещение должно быть оборудовано соответственно стандартам. Все оборудование для хранения товаров на складе можно разделить на две крупные группы: механическое и немеханическое.

Основным видом немеханического оборудования являются стеллажи. Они нужны для размещения грузов и товаров в паллетах или коробках, также возможно расположение их россыпью. В зависимости от способа размещения на них вещей выделяют фронтальные, полочные и консольные варианты. На специфических складах есть хранилища формата cash and carry, архивы, мезонины и т.д. Этот тип оборудования можно также классифицировать по следующим признакам:

* виды емкостей —полностью закрытые (в том числе герметичная закупорка), частично закрытые, открытые;
* назначению — могут быть оборудованы для размещения товаров насыпным или наливным способом, штучного хранения;
* материалу — стойки и держатели могут быть выполнены из металла, дерева, пластика, часто используют комбинированные варианты.
* Помимо стеллажей в хранилищах используют ящики, поддоны и контейнеры.
* К основным видам механического оборудования, применяемого на складах, относятся:
* тележки — используются в небольших хранилищах для транспортировки мелких партий товаров и грузов. Могут быть ручные или самоходные;
* погрузчики — техника, которая облегчает размещение любых товарных единиц в разных концах помещений. Техническая составляющая зависит от грузоподъемности. На небольших складах уместны мини-погрузчики на 300 – 500 кг. В крупных логистических центрах для подъема контейнеров используют ричстакеры. Техника различается по типам питания. Погрузчики могут работать на бензине, газе, дизеле или электричестве, также есть комбинированные варианты;
* штабелеры — устройства для поднятия грузов на высокие полки или стеллажи. Механизм также может использоваться для перемещения грузов по территории склада, но он ездит медленнее обычного погрузчика. Устройства могут быть пневматическими, ручными или самоходными;
* подъемники и подъемные столы — стационарные механизмы для размещения грузов на высоту. Не предусмотрена возможность передвижения их по территории;
* весы — для работы с грузами обычно используют платформенные устройства повышенной прочности;
* упаковочное оборудование — не является обязательным, используется опционально. Часто такие механизмы устанавливают на складах, где оказывают услуги по сортировке, разбору крупных партий на небольшие грузы. К этому типу механизмов относятся полуавтоматические и ручные устройства для обвязки товаров лентой, оброллеры, упаковщики паллетов;
* приборы для автоматизации логистики – устанавливают в больших зданиях, на крупных транспортных узлах. Это сканеры кодов, терминалы для сбора и фиксации данных, принтеры для этикеток.

Нужно понимать, что в хранилищах должна быть четкая организация пространства. Техника и иные конструкции устанавливаются в зависимости от потребностей производства, загрузки и перегрузки, иных логистических операций.

## 1.2 Выбор языка

Для написания этой курсовой работы выбран язык Java.

Версия 11.0.6

Java — строго типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process, язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle.

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

## 1.3 Выбор среды разработки

Выбрана интегрированная среда разработки программного обеспечения IntelliJ IDEA

IntelliJ IDEA — интегрированная среда разработки программного обеспечения для многих языков программирования, в частности Java, JavaScript, Python, разработанная компанией JetBrains.

Первая версия появилась в январе 2001 года и быстро приобрела популярность как первая среда для Java с широким набором интегрированных инструментов для рефакторинга, которые позволяли программистам быстро реорганизовывать исходные тексты программ. Дизайн среды ориентирован на продуктивность работы программистов, позволяя сконцентрироваться на функциональных задачах, в то время как IntelliJ IDEA берёт на себя выполнение рутинных операций.

## 1.4 Постановка задачи

Разработать многопотоковую программу, моделирующую работу товарного склада. Программа, реализует модель работы склада, отвечающего за хранение и продажу некоторого товара (одного). Склад содержит М помещений, каждый из которых может хранить определённое количество единиц товара. Поступающий товар помещается в помещения специальным погрузчиком. За товаром прибыло К покупателей, каждому из которых требуется по ГК единиц товара. Площадка перед складом мала и на ней может в один момент времени находиться либо погрузчик, либо один из покупателей. Если покупателям требуется больше товара, чем имеется на складе, то они ждут новых поступлений, периодически проверяя склад. Время работы склада ограниченно.

# 2.РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

## 2.1 Разработка основного алгоритма программы

Симулятор склада использует две основные сущности. склад и покупатель.

Склад занимается погрузкой товара и обслуживанием покупателей. По условию склад не может делать это одновременно, поэтому ещё одной его задачей является понять, что нужно делать в данный момент для максимально эффективной работы. Для этого склад в программе представлен тремя классами:

* Class Storage. Сущность, отвечающая за хранение товара, выдачу его клиентам, а также выполняет функцию приёма товара с поставки. Размер хранилища ограничен.
* Сlass Provider. Поставщик отвечает за предоставления товара. Привозит товара по одинаковым партиям. Если склад полон, прекращает работу.
* Сlass Loader. Погрузчик может выполнять две функции. Работа с поставщиком, то есть погружает товар в хранилище и обслуживание покупателей, а именно выгружать товар из хранилища и отдавать покупателям.

Покупатель более простая сущность. Она содержит данные о покупатели и его запрос. А именно, имя покупателя и количество товара, которое ему нужно. Самая важная функция покупателя – это узнавать у склада хватает ли продуктов, чтобы удовлетворить его запрос. В программе представлено классом Customer.

Основной алгоритм:

1. Запуск поставщика и погрузчика.
2. Запуск хранилища.
3. Генерация клиентов
4. Обслуживание всех клиентов.
5. Ожидание новых клиентов.

## 2.2 Многопоточность

### 2.2.1 Общие сведения

Многопоточность — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Сутью многопоточности является квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса. Кроме этого, все потоки процесса имеют не только общее адресное пространство, но и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

К достоинствам многопоточной реализации той или иной системы перед многозадачной можно отнести следующее:

* Упрощение программы в некоторых случаях за счёт использования общего адресного пространства.
* Меньшие относительно процесса временные затраты на создание потока.

К достоинствам многопоточной реализации той или иной системы перед однопоточной можно отнести следующее:

* Упрощение программы в некоторых случаях, за счёт вынесения механизмов чередования выполнения различных слабо взаимосвязанных подзадач, требующих одновременного выполнения, в отдельную подсистему многопоточности.
* Повышение производительности процесса за счёт распараллеливания процессорных вычислений и операций ввода-вывода.

В случае, если потоки выполнения требуют относительно сложного взаимодействия друг с другом, возможно проявление проблем многозадачности, таких как взаимные блокировки.

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

Для реализации основного алгоритма использовано многопоточность.

### 2.2.2 Поток погрузчика

Схема погрузчика в программе представлена так:

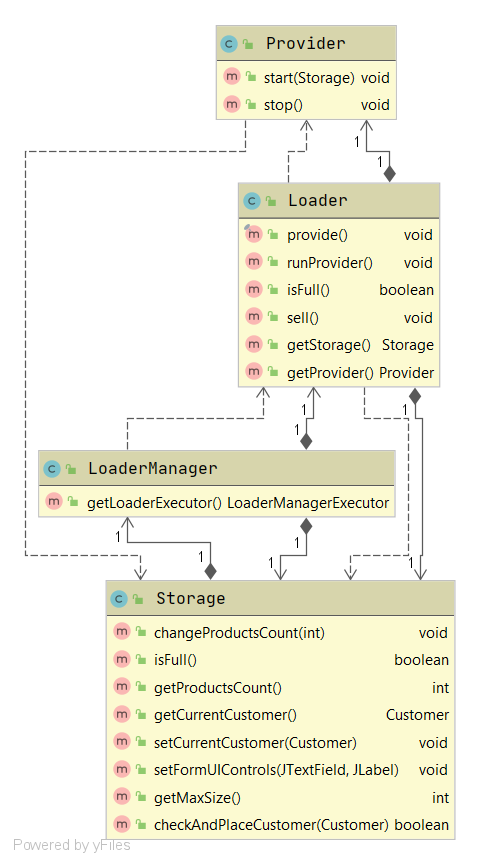


Рисунок 1 - UML диаграмма погрузчика

На диаграмме видно, что погрузчик (класс Loader) агрегирует хранилище и поставщика, а класс LoaderManager хранит самого погрузчика. Это сделано для того, чтобы разделить обязанности. Обязанность выбора, что делать погрузчику сейчас, лежит на классе LoaderManager, а обязанности самого погрузчика инкапсулированы в классе Loader.

LoaderManager имеет один метод, который возвращает объект класса LoaderManagerExecutor.

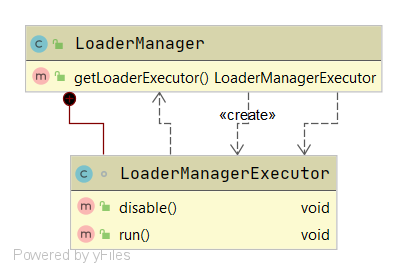


Рисунок 2 - LoaderManagerExecutor

LoaderManagerExecutor – это поток. Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. Этот поток отвечает каждые три секунды проверяет, есть ли покупатель, которого можно обслужить. Если покупатель есть, то он его обслуживает, если же нет, то погрузчик начинает заполнять склад.

class LoaderManagerExecutor implements Runnable {

private boolean isActive;

public void disable() {

isActive = false;

}

public LoaderManagerExecutor(Loader loader) {

isActive = true;

}

public void run() {

while (isActive) {

ConcurrentUtils.sleep(3);

if (storage.getCurrentCustomer() == null) {

loader.provide();

} else {

loader.sell();

}

}

}

}

### 2.2.3 Поток покупателя

Схема покупателя в программе представлена так:

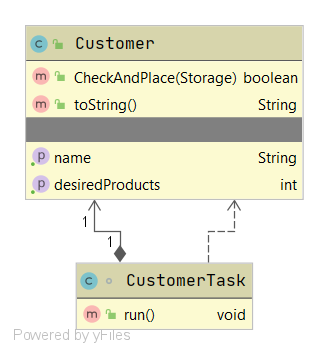


Рисунок 3 - UML диаграмма покупателя

Класс Customer содержит всю информацию о покупателе. CustomerTask – это поток, который содержит в себе объект покупателя. Этот поток каждые сто миллисекунд, проверяет хватает ли покупателю товара на складе. Если хватает, то забирает его, когда погрузчик свободен, иначе проверяет до тех пор, пока не заберёт свой заказ. Таких потоков в симуляторе можно запустить неограниченное количество.

class CustomerTask implements Runnable {

private boolean isActive;

private Customer customer;

private Storage storage;

public CustomerTask(Customer customer, Storage storage) {

this.customer = customer;

this.storage = storage;

isActive = true;

}

public void run() {

while (isActive) {

ConcurrentUtils.sleepMilliseconds(100);

isActive = customer.CheckAndPlace(storage);

}

}

}

## 2.3 Реализация основных блоков кода

Так как код программы достаточно большой, то все он разделён на файлы. Каждый класс находится в отдельном файле.

### 2.3.1 Класс ConcurrentUtils

Вспомогательный статический класс, который отвечает за прерывания в потоках. Обладает методами остановки и задержки потоков. Используется очти во всех классах.

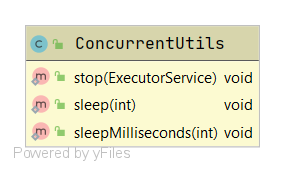


Рисунок 4 – ConcurrentUtils

public class ConcurrentUtils {

public static void stop(ExecutorService executor) {

try {

executor.shutdown();

executor.awaitTermination(10, TimeUnit.SECONDS);

}

catch (InterruptedException e) {

System.err.println("termination interrupted");

}

finally {

if (!executor.isTerminated()) {

System.out.println("Некоторый процессы не завершены");

}

executor.shutdownNow();

}

}

public static void sleep(int seconds) {

try {

TimeUnit.SECONDS.sleep(seconds);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

}

public static void sleepMilliseconds(int milliseconds) {

try {

TimeUnit.MILLISECONDS.sleep(milliseconds);

} catch (InterruptedException e) {

throw new IllegalStateException(e);

}

}

}

### 2.3.2 Класс Customer

Customer представляет собой модель покупателя. Хранит данные о нём и умеет проверять склад.

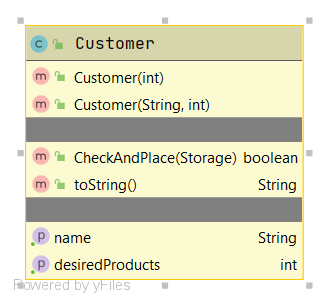


Рисунок 5 - Customer

public class Customer {

private final String name;

private final int desiredProducts;

public Customer(int desiredProducts) {

this(String.valueOf(desiredProducts), desiredProducts);

}

public Customer(String name, int desiredProducts) {

this.name = name;

this.desiredProducts = desiredProducts;

}

public boolean CheckAndPlace(Storage storage){

return !storage.checkAndPlaceCustomer(this);

}

public String getName() {

return name;

}

public int getDesiredProducts() {

return desiredProducts;

}

@Override

public String toString() {

return "Покупатель: " + getName() + ". Требуется продуктов: " + String.valueOf(getDesiredProducts()) + ". ";

}

}

### 2.3.3 Класс Loader

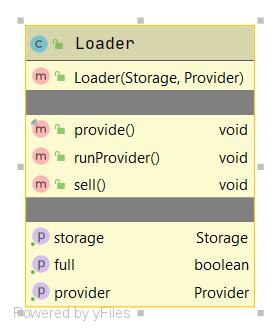
Этот класс симулирует работу погрузчика, может либо погружать товар в хранилище, либо отдавать его покупателям.

Рисунок 6 - Loader

public class Loader {

private Storage storage;

private Provider provider;

public Loader(Storage storage, Provider provider) {

this.storage = storage;

this.provider = provider;

}

public final void provide() {

if (isFull()) {

return;

}

runProvider();

}

public void runProvider(){

provider.start(storage);

ConcurrentUtils.sleep(3);

provider.stop();

}

public boolean isFull() {

if (storage.isFull()) {

return true;

}

return false;

}

public void sell() {

var customer = storage.getCurrentCustomer();

storage.changeProductsCount(-(customer.getDesiredProducts()));

storage.setCurrentCustomer(null);

}

public Storage getStorage() {

return storage;

}

public Provider getProvider() {

return provider;

}

}

### 2.3.4 Класс Storage

Данный класс представляет собой хранилище. Может отдавать и принимать товары. Центральный класс программы.

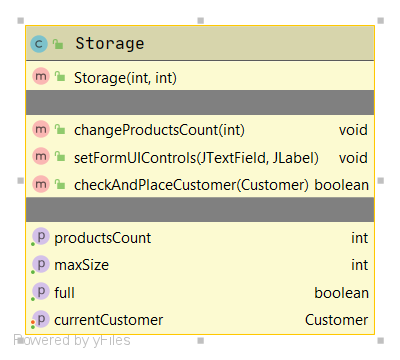


Рисунок 7 - Storage

public class Storage {

private int productsCount;

private JTextField formProductsCount;

private JLabel formIsFull;

private final ReadWriteLock lock;

private final ReentrantLock reentrantLock;

private LoaderManager loaderManager;

private Customer currentCustomer;

private final int maxSize;

public Storage(int productsCount, int maxSize) {

lock = new ReentrantReadWriteLock();

this.maxSize = maxSize;

this.productsCount = productsCount;

this.currentCustomer = null;

reentrantLock = new ReentrantLock();

}

public void changeProductsCount(int delta) {

lock.writeLock().lock();

try {

if (isFull() && delta > 0) {

formProductsCount.setText(String.valueOf(maxSize));

formIsFull.setVisible(true);

return;

}

if (productsCount + delta >= maxSize) {

delta = maxSize - productsCount;

productsCount = maxSize;

System.out.println("Количество товара изменилось на: " + delta);

System.out.println("Склад заполнен");

formProductsCount.setText(String.valueOf(maxSize));

formIsFull.setVisible(true);

} else {

productsCount += delta;

System.out.println("Количество товара изменилось на: " + delta);

formProductsCount.setText(String.valueOf(productsCount));

formIsFull.setVisible(false);

}

} finally {

lock.writeLock().unlock();

}

}

public boolean isFull() {

return productsCount == maxSize;

}

public int getProductsCount() {

lock.readLock().lock();

try {

return productsCount;

} finally {

lock.readLock().unlock();

}

}

public Customer getCurrentCustomer() {

return currentCustomer;

}

public void setCurrentCustomer(Customer currentCustomer) {

this.currentCustomer = currentCustomer;

}

public void setFormUIControls(JTextField formProductsCount, JLabel isFull) {

this.formProductsCount = formProductsCount;

this.formIsFull = isFull;

}

public int getMaxSize() {

return maxSize;

}

public boolean checkAndPlaceCustomer(Customer customer) {

reentrantLock.lock();

try {

if (customer.getDesiredProducts() < productsCount && currentCustomer == null) {

setCurrentCustomer(customer);

return true;

} else {

return false;

}

} finally {

reentrantLock.unlock();

}

}

}

## 2.4 Визуальная составляющая приложения

### 2.4.1 Графический интерфейс

Для создания графического интерфейса была использована технология Swing.

Swing — библиотека для создания графического интерфейса для программ на языке Java. Swing был разработан компанией Sun Microsystems. Он содержит ряд графических компонентов (англ. Swing widgets), таких как кнопки, поля ввода, таблицы и т. д.

Swing относится к библиотеке классов JFC, которая представляет собой набор библиотек для разработки графических оболочек. К этим библиотекам относятся Java 2D, Accessibility-API, Drag & Drop-API и AWT.

Вслед за AWT Sun разработала графическую библиотеку компонентов Swing, полностью написанную на Java. Для отрисовки используется 2D, что принесло с собой сразу несколько преимуществ. Набор стандартных компонентов значительно превосходит AWT по разнообразию и функциональности. Swing позволяет легко создавать новые компоненты, наследуясь от существующих, и поддерживает различные стили и скины.

Создатели новой библиотеки пользовательского интерфейса Swing не стали «изобретать велосипед» и в качестве основы для своей библиотеки выбрали AWT. Конечно, речь не шла об использовании конкретных тяжеловесных компонентов AWT (представленных классами Button, Label и им подобными). Нужную степень гибкости и управляемости обеспечивали только легковесные компоненты. На диаграмме наследования представлена связь между AWT и Swing.

Важнейшим отличием Swing от AWT является то, что компоненты Swing вообще не связаны с операционной системой и поэтому гораздо более стабильны и быстры. Такие компоненты в Java называются легковесными (lightweight), и понимание основных принципов их работы во многом объяснит работу Swing.

Для создания графического интерфейса приложения необходимо использовать специальные компоненты библиотеки Swing, называемые контейнерами высшего уровня (top level containers). Они представляют собой окна операционной системы, в которых размещаются компоненты пользовательского интерфейса. К контейнерам высшего уровня относятся окна JFrame и JWindow, диалоговое окно JDialog, а также апплет JApplet (который не является окном, но тоже предназначен для вывода интерфейса в браузере, запускающем этот апплет). Контейнеры высшего уровня Swing представляют собой тяжеловесные компоненты и являются исключением из общего правила. Все остальные компоненты Swing являются легковесными.

### 2.4.2 Компоненты

Компонент Swing следует архитектуре Model-View-Controller для выполнения следующих критериев.

* Одного API должно быть достаточно для поддержки множественного внешнего вида.
* API должен быть ориентирован на модель, чтобы не требовалось, чтобы у API самого высокого уровня были данные.
* API заключается в использовании модели Java Bean, чтобы инструменты Builder и IDE могли предоставлять разработчикам более качественные сервисы для использования.

Использованы компоненты:

* JPanel
* JScrollPane
* JButton
* JTextField
* JFormattedTextField
* JProgressBar
* SyncJList

SyncJList – это наследник JList, в котором определены методы для работы с потоками клиентов.

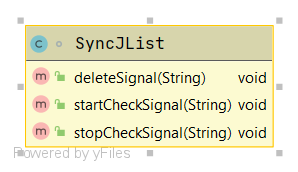


Рисунок 8 – SyncJLIst

class SyncJList extends JList<String> {

public void deleteSignal(String customerInfo) {

synchronized (this) {

DefaultListModel<String> model = (DefaultListModel<String>) this.getModel();

setSelectionBackground(Color.GREEN);

int index = model.indexOf(customerInfo);

setSelectedIndex(index);

ConcurrentUtils.sleepMilliseconds(2000);

model.removeElement(customerInfo);

setSelectionBackground(Color.blue);

}

}

public void startCheckSignal(String customerInfo) {

synchronized (this) {

DefaultListModel<String> model = (DefaultListModel<String>) this.getModel();

int index = model.indexOf(customerInfo);

addSelectionInterval(index, index);

ConcurrentUtils.sleepMilliseconds(100);

}

}

public void stopCheckSignal(String customerInfo) {

synchronized (this) {

DefaultListModel<String> model = (DefaultListModel<String>) this.getModel();

int index = model.indexOf(customerInfo);

removeSelectionInterval(index, index);

}

}

}

### 2.4.3 Формы

Самый главный элемент графических интерфейсов — форма для элементов управления. К форме можно присоединять множество различных элементов управления в любом расположении и в любых комбинациях. Форму можно будет перемещать, при этом все элементы управления, которые к ней присоединены, будут перемещаться вместе с ней.

В программе использована одна форма.

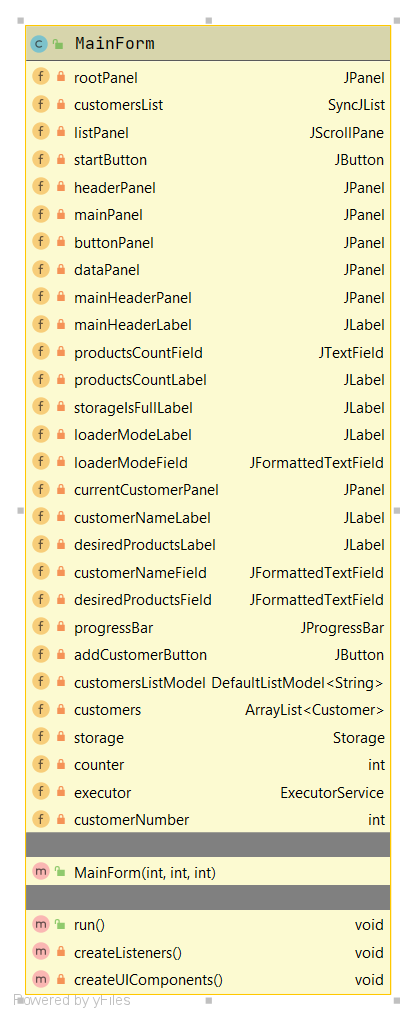


Рисунок 9 – MainForm

Форма выглядит так:

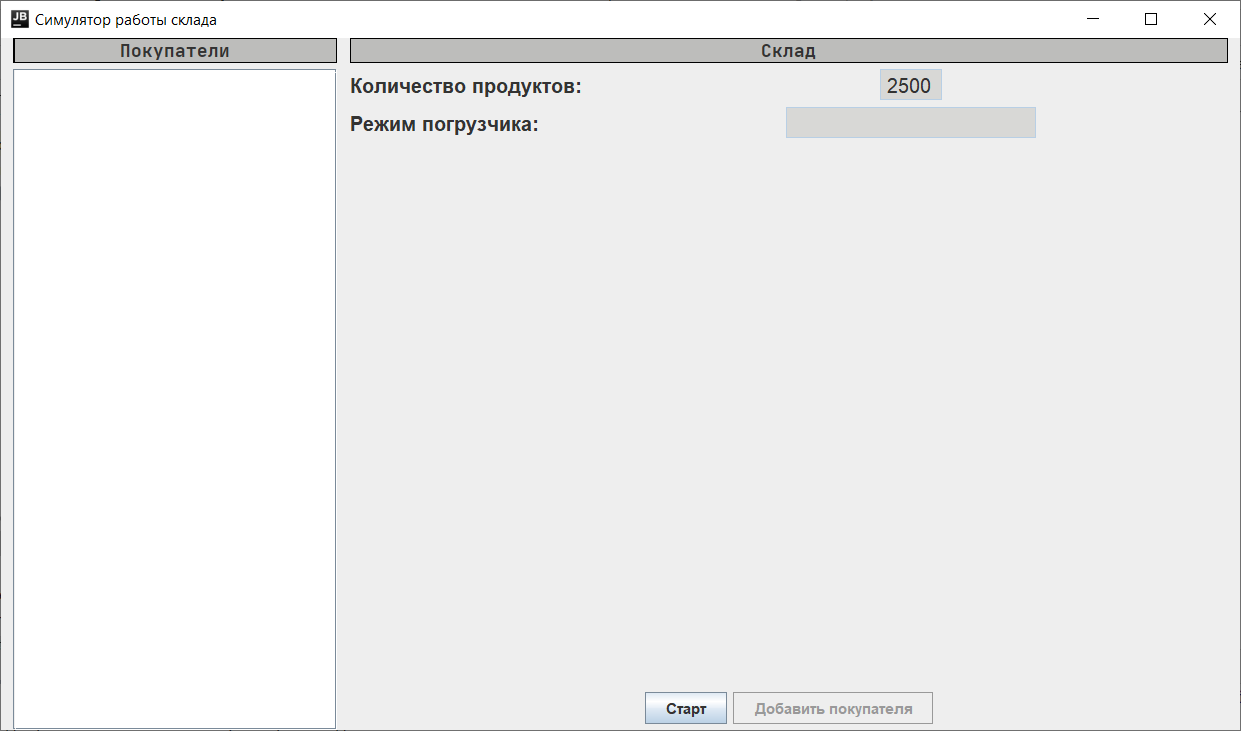


Рисунок 10 - MainForm

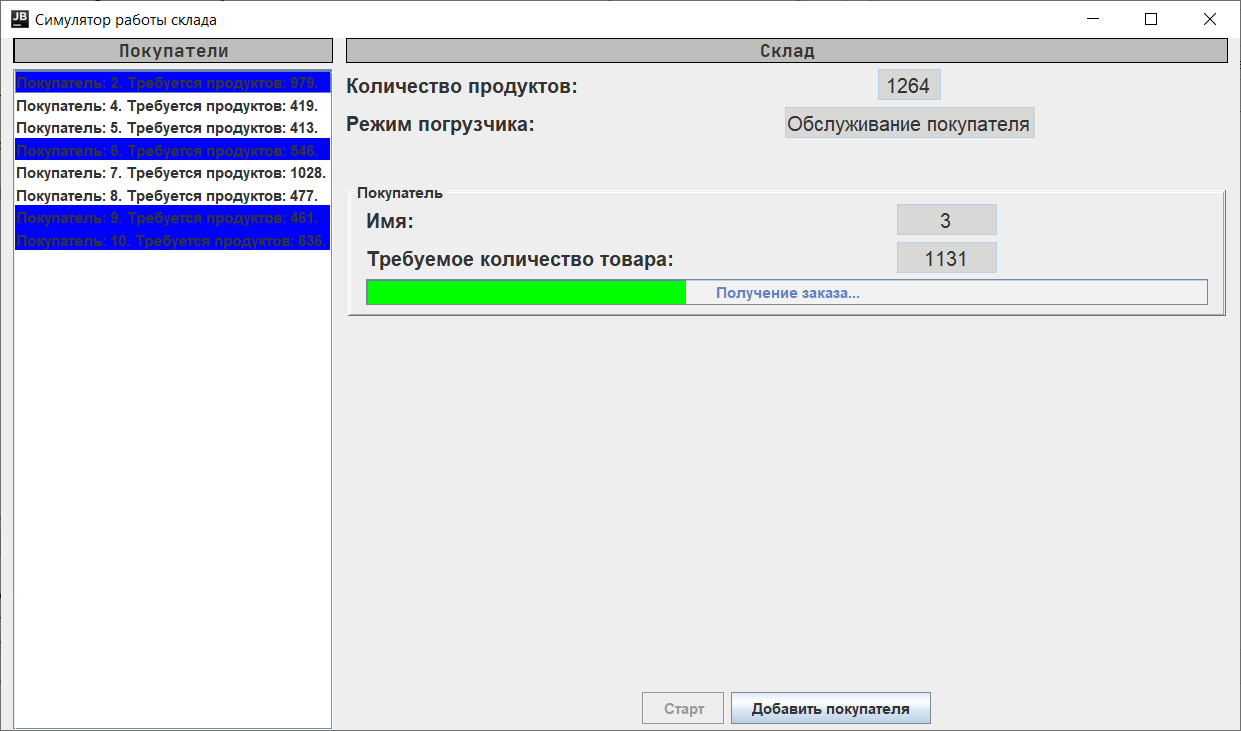


Рисунок 11 - MainForm во время выполнения

# 3. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

## 3.1 Концепция объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Идеологически ООП — подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов.

Управляемость для иерархических систем предполагает минимизацию избыточности данных (аналогичную нормализации) и их целостность, поэтому созданное удобно управляемым — будет и удобно пониматься. Таким образом, через тактическую задачу управляемости решается стратегическая задача — транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

* абстракция для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счёте — контекстное понимание предмета, формализуемое в виде класса;
* инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления;
* наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя всё остальное, учтённое на предыдущих шагах;
* полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот — собрать воедино.

То есть фактически речь идёт о прогрессирующей организации информации согласно первичным семантическим критериям: «важное/неважное», «ключевое/подробности», «родительское/дочернее», «единое/множественное». Прогрессирование, в частности, на последнем этапе даёт возможность перехода на следующий уровень детализации, что замыкает общий процесс.

В проекте использована данная концепция. Это можно увидеть на UML диаграмме.

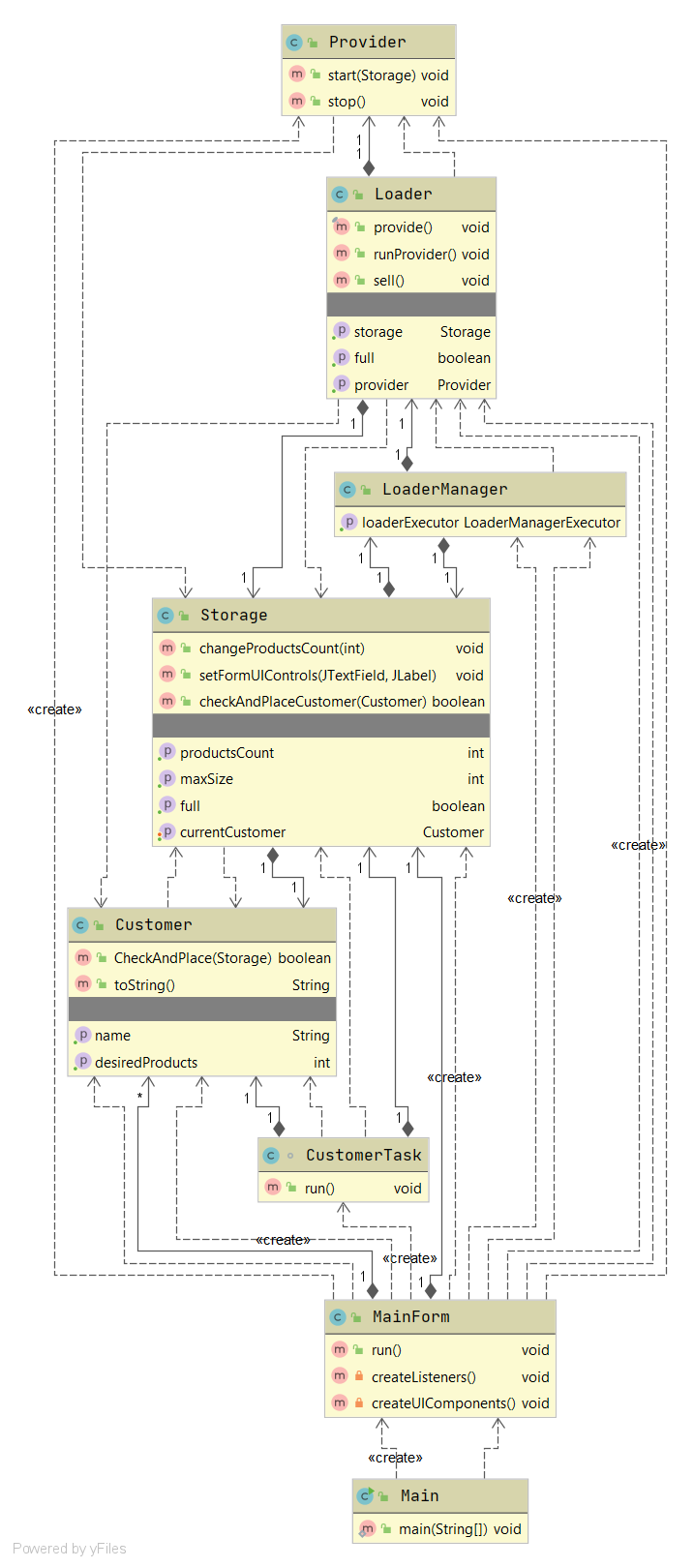


Рисунок 12 - UML Диаграмма проекта

## 3.2 Принципы SOLID

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. В переводе на русский: принципы единственной ответственности, открытости / закрытости, подстановки Барбары Лисков, разделения интерфейса и инверсии зависимостей)

Аббревиатура SOLID была предложена Робертом Мартином, автором нескольких книг, широко известных в сообществе разработчиков. Эти принципы позволяют строить на базе ООП масштабируемые и сопровождаемые программные продукты с понятной бизнес-логикой.

Расшифровка:

* Single responsibility — принцип единственной ответственности
* Open-closed — принцип открытости / закрытости
* Liskov substitution — принцип подстановки Барбары Лисков
* Interface segregation — принцип разделения интерфейса
* Dependency inversion — принцип инверсии зависимостей

Принцип единственной обязанности / ответственности (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности. Подробнее про SRP...

Принцип открытости / закрытости (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода. Подробнее про OCP...

Принцип подстановки Барбары Лисков (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом». Подробнее про LSP...

Принцип разделения интерфейса (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют. Подробнее про ISP...

Принцип инверсии зависимостей (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

Все эти принципы использованы при создание данного проекта.

## 3.3 Паттерн декоратор

### 3.3.1 Структура

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Этот паттерн используется, когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

В данном проекте использованы 2 вида декораторов:

* Декоратор для работы с логами
* Декоратор для работы с формой.

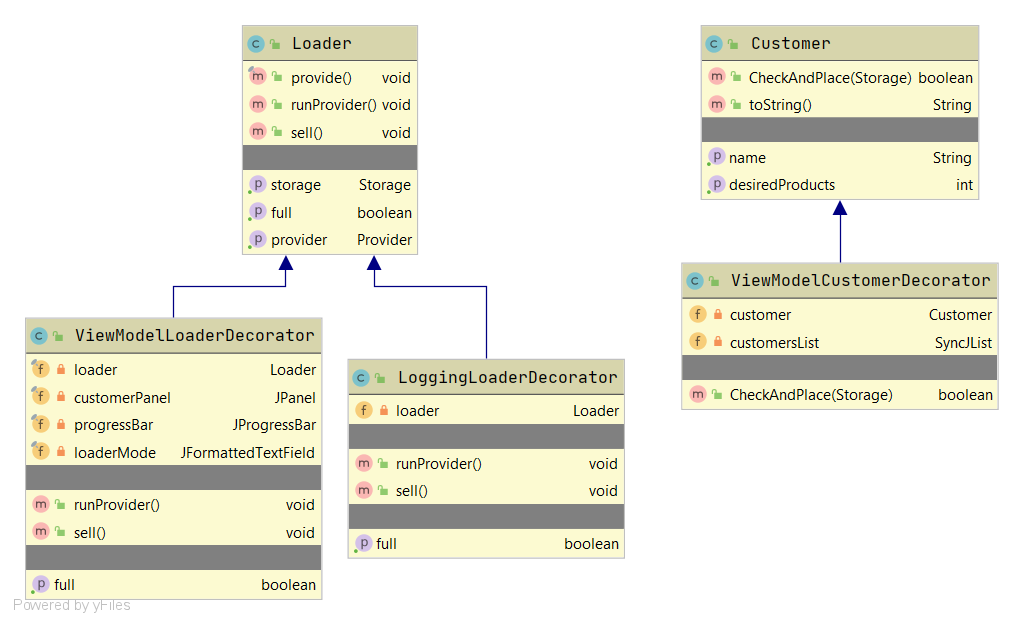


Рисунок 13 - Структура декораторов

### 3.3.2 Класс LoggingLoaderDecorator

Этот декоратор работает с консолью. Он выводит данные о том, чем сейчас занимается погрузчик.

public class LoggingLoaderDecorator extends Loader {

private Loader loader;

public LoggingLoaderDecorator(Loader loader) {

super(loader.getStorage(),loader.getProvider());

this.loader = loader;

}

@Override

public void runProvider() {

System.out.println("Погрузка товара");

loader.runProvider();

}

@Override

public boolean isFull() {

boolean temp = loader.isFull();

if(temp){

System.out.println("Склад заполнен");

}

return temp;

}

@Override

public void sell() {

Customer currentCustomer = getStorage().getCurrentCustomer();

System.out.println("---------------------------");

System.out.println("Покупатель: " + currentCustomer.getName() + " на площадке");

System.out.println("Покупатель " + currentCustomer.getName() + " получает свой заказ...");

loader.sell();

System.out.println("Покупатель " + currentCustomer.getName() + " получил свой заказ");

System.out.println("На складе осталось: " + getStorage().getProductsCount());

System.out.println("---------------------------");

}

}

### 3.3.3 Классы ViewModelCustomerDecorator и ViewModelLoaderDecorator

Эти классы предназначены для работы с MainForm

ViewModelCustomerDecorator:

public class ViewModelCustomerDecorator extends Customer {

private Customer customer;

private SyncJList customersList;

public ViewModelCustomerDecorator(Customer customer, SyncJList customersList) {

super(customer.getName(), customer.getDesiredProducts());

this.customer = customer;

this.customersList = customersList;

}

@Override

public boolean CheckAndPlace(Storage storage) {

customersList.startCheckSignal(customer.toString());

boolean isActive = customer.CheckAndPlace(storage);

if (!isActive) customersList.deleteSignal(customer.toString());

customersList.stopCheckSignal(customer.toString());

return isActive;

}

}

ViewModelLoaderDecorator:

public class ViewModelLoaderDecorator extends Loader {

private final Loader loader;

private final JPanel customerPanel;

private final JProgressBar progressBar;

private final JFormattedTextField loaderMode;

public ViewModelLoaderDecorator(Loader loader, JPanel customerPanel, JProgressBar progressBar, JFormattedTextField loaderMode) {

super(loader.getStorage(), loader.getProvider());

this.loader = loader;

this.customerPanel = customerPanel;

this.progressBar = progressBar;

this.loaderMode = loaderMode;

}

@Override

public void runProvider() {

loaderMode.setText("Погрузка товара");

loader.runProvider();

loaderMode.setText("Ожидание");

}

@Override

public boolean isFull() {

boolean temp = loader.isFull();

if(temp){

loaderMode.setText("Ожидание");

}

return temp;

}

@Override

public void sell() {

Customer currentCustomer = getStorage().getCurrentCustomer();

JFormattedTextField name = (JFormattedTextField) customerPanel.getComponent(1);

JFormattedTextField desiredProducts = (JFormattedTextField) customerPanel.getComponent(3);

JProgressBar progressBar = (JProgressBar) customerPanel.getComponent(4);

name.setText(currentCustomer.getName());

desiredProducts.setText(String.valueOf(currentCustomer.getDesiredProducts()));

loaderMode.setText("Обслуживание покупателя");

customerPanel.setVisible(true);

for (int i = 0; i < 100; i++) {

progressBar.setValue(progressBar.getValue() + 1);

ConcurrentUtils.sleepMilliseconds((int) (Math.random() \* (50 - 10)) + 10);

}

loader.sell();

progressBar.setValue(0);

customerPanel.setVisible(false);

loaderMode.setText("Ожидание");

}

}

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта был разработан симулятор склада. В ходе его разработки были получены теоретические и практические знания по языку программирования Java, а также по принципам объектно-ориентированного программирование, паттернам проектирования и многопоточному программированию.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1.WIKIPEDIA [Электронный ресурс] – WIKIPEDIA: <https://ru.wikipedia.org>