Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

**Контрольная работа № 1**

Методы и средства эффективного проектирования типовых решений модулей программных приложений.

Выполнил:

ст.гр. 672302

Морочко А.В.

Проверила:

Унучек Т.М.

Минск 2018

**Задание:**

В соответствии с вариантом предметной области необходимо предложить архитектуру автоматизированной системы с использованием указанного в задании паттерна. Архитектуру представить в виде диаграммы классов в стандарте UML, сгенерировать программный код.

**Предметная область:**

«Разработка автоматизированной системы учета пациентов медицинского учреждения»

**Ход выполнения задания:**

**Паттерн проектирования** — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

**Типы паттернов**:

**—**Порождающие(предоставляют механизмы инициализации, позволяя создавать объекты удобным способом)

**—**Структурные(определяют отношения между классами и объектами, позволяя им работать совместно)

**—**Поведенческие(используются для того, чтобы упростить взаимодействие между сущностями).

В своей контрольной работе я использую паттерн **Adapter** - группа structural. **Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1.Адаптер имеет интерфейс, который совместим с одним из объектов.

2.Поэтому этот объект может свободно вызывать методы адаптера.

3.Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

**Диаграмма классов** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Static Structure diagram) — диаграмма, демонстрирующая [классы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) системы, их [атрибуты](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0), [методы](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [взаимосвязи](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) между ними. Входит в [UML](https://ru.wikipedia.org/wiki/UML).Диаграмма классов – прeдставлeна на рисункe 1.

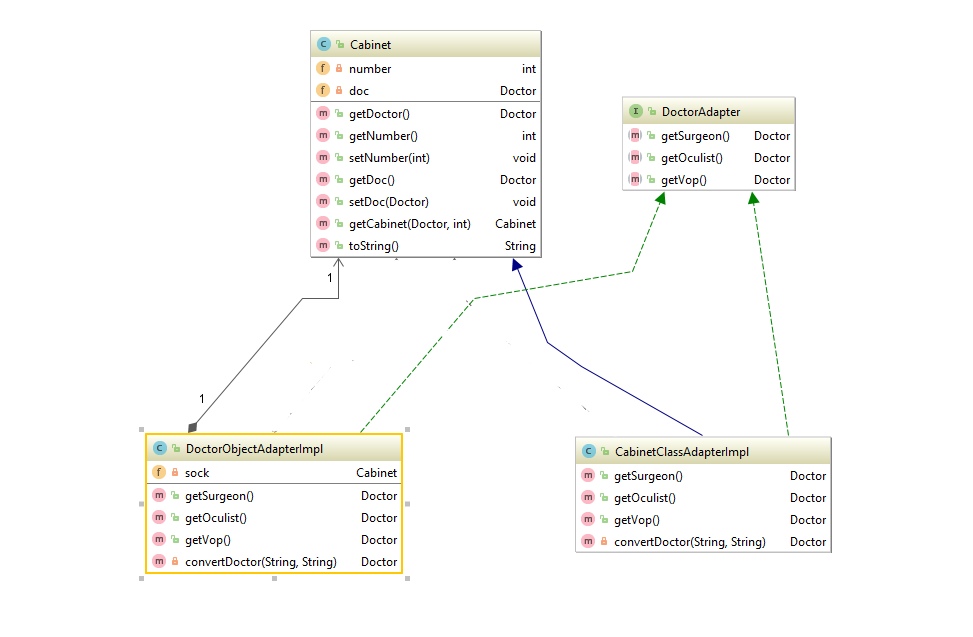
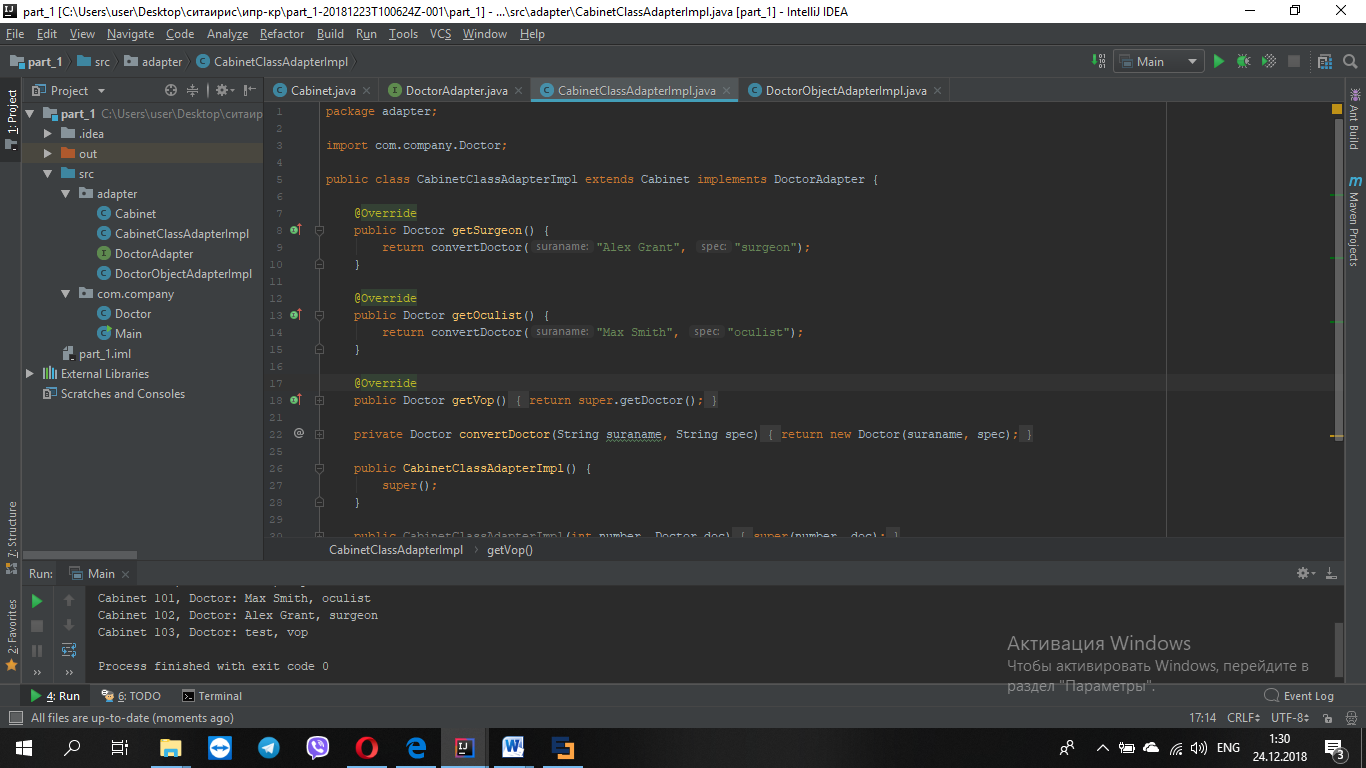
****

Рисунок 1 – Диаграмма классов

**Результат парсирования:**



**Исходный код:**

**class Cabinet**

package adapter;  
import com.company.Doctor;  
public class Cabinet {  
 private int number;  
 private Doctor doc;  
 public Cabinet() {  
 }  
 public Cabinet(int number, Doctor doc) {  
 this.number = number;  
 this.doc = doc;  
 }  
 public Doctor getDoctor(){  
 return new Doctor("test", "vop");  
 }  
 public int getNumber() {  
 return number;  
 }  
 public void setNumber(int number) {  
 this.number = number;  
 }  
 public Doctor getDoc() {  
 return doc;  
 }  
 public void setDoc(Doctor doc) {  
 this.doc = doc;  
 }  
 public Cabinet getCabinet(Doctor d, int number) {  
 return new Cabinet(number, d);  
 }  
 @Override  
 public String toString() {  
 return "Cabinet "+ number +  
 ", Doctor: " + doc +  
 ", " + doc.getSpeciality();  
 }  
}

**class CabinetClassAdapterImpl**

package adapter;  
import com.company.Doctor;  
public class CabinetClassAdapterImpl extends Cabinet implements DoctorAdapter {  
 @Override  
 public Doctor getSurgeon() {  
 return convertDoctor("Alex Grant", "surgeon");  
 }  
 @Override  
 public Doctor getOculist() {  
 return convertDoctor("Max Smith", "oculist");  
 }  
 @Override  
 public Doctor getVop() {  
 return super.getDoctor();  
 }  
 private Doctor convertDoctor(String suraname, String spec) {  
 return new Doctor(suraname, spec);  
 }  
 public CabinetClassAdapterImpl() {  
 super();  
 }  
 public CabinetClassAdapterImpl(int number, Doctor doc) {  
 super(number, doc);  
 }  
}

**class DoctorObjectAdapterImpl**

package adapter;  
import com.company.Doctor;  
public class DoctorObjectAdapterImpl implements DoctorAdapter {  
 private Cabinet sock = new Cabinet();  
 @Override  
 public Doctor getSurgeon() {  
 return convertDoctor("Alex Grant", "surgeon");  
 }  
 @Override  
 public Doctor getOculist() {  
 return convertDoctor("Max Smith", "oculist");  
 }  
 @Override  
 public Doctor getVop() {  
 return sock.getDoctor();  
 }  
 private Doctor convertDoctor(String suraname, String spec) {  
 return new Doctor(suraname, spec);  
 }  
}

**interface DoctorAdapter**

package adapter;  
import com.company.Doctor;  
public interface DoctorAdapter {  
 Doctor getSurgeon();  
 Doctor getOculist();  
 Doctor getVop();  
}