МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

ОТЧЕТ

по лабораторной работе №3

по дисциплине

Шаблоны проектирования программного обеспечения

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_ Жевнерчук Д.В,\_\_

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_Сапожников В.О.\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_ Сухоруков В.А.\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_Мосташов В.С.\_\_\_

(подпись) (фамилия, и.,о.)

\_\_\_19-ИВТ-3\_\_\_\_\_\_\_\_

(шифр группы)

Работа защищена «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2021

# Вариант 8.

Реализуйте консольную утилиту, позволяющую создавать графовые структуры, добавлять узлы и связи, причем каждый узел определяется именем и типом, а каждая связь — именем узла источника, именем узла приемника, типом. Для узлов определены следующие типы: класс, индивид, атрибут, значение. Для связей определены следующие типы: объектное свойство, свойство данных.

Приложение должно позволять:

1. Создавать одиночные узлы-классы.
2. Создавать нескольких индивидов одного класса, при этом свойство данных «ИмеетИндивида» должно формироваться автоматически и для каждого индивида автоматически создается атрибут «Идентификатор» с уникальным номером в пределах всех узлов-атрибутов текущего индивида.
3. Создание нескольких подклассов одного класса, при этом объектное свойство «Подкласс» должно формироваться автоматически

Полученный граф необходимо распечатать в консоли в произвольной, но понятной текстовой форме.

# Проектное решение

## Обоснование выбора паттернов

Поскольку целью работы является создание графовой структуры, то в качестве основного паттерна проектирования был выбран компоновщик. Основным классом является абстрактный класс Node, который содержит основные поля и методы необходимые для работы с узлами.

Производными классами являются:

1. ContainerNode -абстрактные класс - узел компонент, который может содержать потомков
2. Leaf - конечный узел.

В ходе работы было реализовано 3 вида узлов - контейнеров, унаследованных от ContainerNode: ClassNode - может являться корнем, классом или подклассом графа, IndividualNode узел-индивид, AttributeNode- содержит название одного атрибута. Так же был реализован один конечный узел ValueNode - содержит одно значение атрибута.

Для связи между узлами созданы 2 вида связей: DataProperty (связи: IndividNode → AttributeNode, AttributeNode → ValueNode) и ObjectProperty (ClassNode → ClassNode, ClassNode → IndividNode).

**Многопоточность**

Поскольку создание графа происходит последовательно для каждого узла и присутствует четкая иерархия родителей и наследников, то для использования многопоточности было решено имитировать клиент-серверное взаимодействия на основе задачи о спящем брадобрее.

Для создания графовой структуры реализован паттерн Строитель. От основного интерфейса Builder, унаследован GraphBuilder, который реализует методы необходимые для создания графа.

Для генерации сервером графов был создан интерфейс Generator и реализующий его класс G raphGenerator. Генератору на создание поступает запрос, содержащий имя корневого узла графа. Генератор приступает к созданию графа в отдельном потоке, для каждого последующего сгенерированного узла добавляется от 0 до n (где n меняется в зависимости от типа узла) наследников с небольшой задержкой.

Для генерации запросов был создан класс Client, который в отдельном потоке генерирует запросы на создания графов. Взаимодействие клиента и сервера происходит через очередь запросов – разделяемую секцию данных.

Для вывода графа в консоль в удобном для прочтения виде, создан интерфейс Printer и унаследованный от него класс GraphPrinter, который так же является бином и singleton'ом.

……………………………………………………………………………

На рис 1. приведена диаграмма классов.

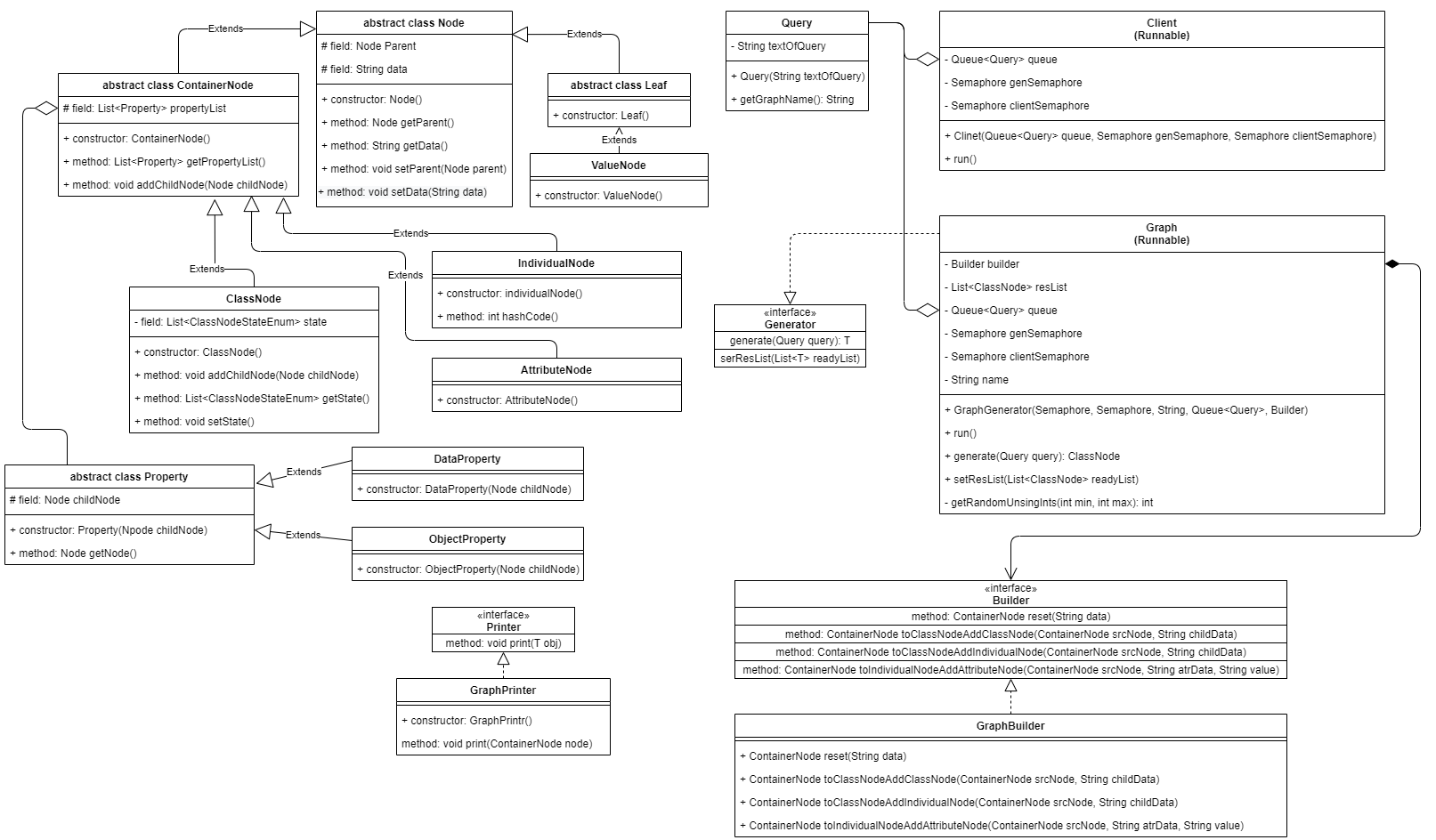


Рис. 1 – Диаграмма классов

# Вывод

Использование многопоточности позволяет запускать в разных потоках независимые друг от друга части программы, что позволяет сократить время выполнения задачи.

Однако к использованию многопоточности стоит подходить с осторожностью, чтобы избежать конфликтов и блокировок при работе с разделяемыми ресурсами, так же не стоит забывать, что каждый поток занимает свое кол-во памяти.

# Приложение 1

**Программный код**

GraphBuilder.java

package com.ngtu.sdp.loboratory\_worl3.builder;  
  
  
import com.ngtu.sdp.loboratory\_worl3.nodes.\*;  
  
*/\*\*  
 \* Частная релизация bilder'a  
 \* Bilder для графа.  
 \*  
 \* @see Builder  
 \*/*public class GraphBuilder implements Builder  
{  
 public GraphBuilder() {  
 }  
  
 */\*\*  
 \* Метод, для добавления к узлу ClassNode потомка ClassNode  
 \*  
 \* @param childNodeData - данные дочернего узла  
 \* @param srcNode - узел родитель, к которому необходимо добавить*

*\* потомка.  
\* @return полученный узел  
\*/*@Override  
public ContainerNode toClassNodeAddClassNode(ContainerNode srcNode, String childNodeData)

{  
 *//Создание нового дочернего узла* ClassNode childNode = new ClassNode(srcNode, childNodeData,

ClassNodeStateEnum.*SUBCLASS*);  
  
 *//К род. узлу добавляем новый узел* srcNode.addChildNode(childNode);

*//Если родиетльский узел еще не имеет состояния "ИМЕЕТ ПОДКЛАСС", то задаем ему это состяние* if (!((ClassNode) srcNode).getState()

.contains(ClassNodeStateEnum.*HAVE\_SUBCLASS*))

{  
 ((ClassNode) srcNode).addState

(ClassNodeStateEnum.*HAVE\_SUBCLASS*);  
 }  
  
 return childNode;  
 }  
  
 */\*\*  
 \* Метод, для добавления к узлу ClassNode потомка IndividualNode  
 \*  
 \* @param childNodeData - данные дочернего узла  
 \* @param srcNode - узел родитель, к которому необходимо*

*\* добавить потомка.  
 \* @return полученный узел  
 \*/* @Override  
 public ContainerNode toClassNodeAddIndividualNode(ContainerNode

srcNode, String childNodeData) {  
 *//Создание нового дочернего узла* IndividualNode childNode = new IndividualNode(srcNode, childNodeData);  
  
 *//К род. узлу добавляем новый узел* srcNode.addChildNode(childNode);  
  
 *//Если родиетльский узел еще не имеет состояния "ИМЕЕТ ИНДИВИДА", то задаем ему это состяние* if (!((ClassNode) srcNode).getState().

contains(ClassNodeStateEnum.*HAVE\_INDIVIDUAL*)) {  
 ((ClassNode) srcNode).addState

(ClassNodeStateEnum.*HAVE\_INDIVIDUAL*);  
 }  
  
 *//Автоматическое задание Атрибута ID* childNode.setID();  
  
 return childNode;  
 }  
  
 */\*\*  
 \* Метод, для добавления к узлу IndividualNode потомка AttributeNode*

*\* с параметром  
 \*  
 \* @param value - значение атрибута  
 \* @param atrNodeData - данные дочернего узла  
 \* @param srcNode - узел родитель, к которому необходимо добавить потомка.  
 \* @return полученный узел  
 \*/* @Override  
 public ContainerNode toIndividualNodeAddAttributeNode(ContainerNode srcNode, String atrNodeData, String value) {  
 *//Создание нового узла значения* ValueNode valueNode = new ValueNode(value);  
  
 *//Создание нового узла атрибута* AttributeNode atrNode = new AttributeNode(srcNode, atrNodeData);  
  
 *//К атрибуту прибавляем значение* atrNode.addChildNode(valueNode);  
  
 *//Значению записываем родителя атрибута* valueNode.setParent(atrNode);  
  
 *//К род. узлу добавляем новый узел* srcNode.addChildNode(atrNode);  
 return atrNode;  
 }  
}

Client.java

package com.ngtu.sdp.loboratory\_worl3.client;  
  
import com.ngtu.sdp.loboratory\_worl3.query.Query;  
  
import java.util.Queue;  
import java.util.concurrent.Semaphore;  
  
*/\*\*  
 \* Класс клиент - генерирует запросы на  
 \* создания графов в отдельном потоке.  
 \* \*/*public class Client implements Runnable  
{

*//ссылка на очередь – разделяемый ресурс между потоками*  
 private final Queue<Query> queue;

*//ссылка на семафор генератора*

private final Semaphore genSemaphore;

*//ссылка на семафор клиента* private final Semaphore clientSemaphore;  */\*\*  
 \* Конструктор с параметрами.  
 \*  
 \* @param clientSemaphore - ссылка на семафор клиента  
 \* @param genSemaphore - ссылка на семафор генератора  
 \* @param queue - ссылка на очередь  
 \* \*/* public Client(Queue<Query> queue, Semaphore clientSemaphore,

Semaphore genSemaphore)  
 {  
 this.queue = queue;  
 this.clientSemaphore = clientSemaphore;  
 this.genSemaphore = genSemaphore;  
 }  
  
 */\*\*  
 \* Метод run() интерфейса Runnable - данный код выполняется  
 \* в дополнительном потоке.  
 \* Алгоритм задачи о спящем брадобрее для клиента  
 \* \*/* @Override  
 public void run()  
 {  
 *//генерация 25 запросов* for (int i = 1; i <= 10; i++)  
 {  
 Query query = new Query("create graph Graph" + i);  
 try  
 {  
 *//Опустили mutex на очередь* synchronized (queue)  
 {  
 System.*out*.println("Получение запроса на создание " +

query.getGraphName());  
  
 *//Добавили запрос в очередь, освободили семафор клиента* queue.add(query);  
 clientSemaphore.release();  
 }  
 *//Подняли mutex, заняли семафор генератора* genSemaphore.acquire();  
 Thread.*sleep*(250);  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
 }  
}

GrahpGenerator.java

package com.ngtu.sdp.loboratory\_worl3.generator;  
  
import com.ngtu.sdp.loboratory\_worl3.query.Query;  
import com.ngtu.sdp.loboratory\_worl3.builder.Builder;  
import com.ngtu.sdp.loboratory\_worl3.nodes.\*;  
  
import java.util.\*;  
import java.util.concurrent.Semaphore;  
  
*/\*\*  
 \* Генератор графовой структуры.  
 \* Созданный граф содержит:  
 \* Корневой узел  
 \* 1-4 подкласса корневого узла  
 \* 0-4 индивидов для каждого подкласса  
 \* 1-5 атрибутов для каждого индивида (включая ID)  
 \*  
 \* @see Generator  
 \* @see Builder  
 \* @see Query  
 \* \*/*public class GraphGenerator implements Generator<ClassNode>, Runnable  
{  
 private final Builder builder; *//ссылка на билдер* private List<ClassNode> resList; *//ссылка на список корневых узлов, в который генераторы будут заносить готовые грфы*

*//ссылка на очередь – разделяемый ресурс между потоками* private final Queue<Query> queue; private final Semaphore genSemaphore; *//семафор генератора*  
 private final Semaphore clientSemaphore; *//семафор клиента* private final String name; *//имя генератора* public GraphGenerator(Semaphore genSemaphore, Semaphore clientSemaphore,

String name, Queue<Query> queue, Builder builder)  
 {  
 this.builder = builder;  
 this.name = name;  
 this.genSemaphore = genSemaphore;  
 this.clientSemaphore = clientSemaphore;  
 this.queue = queue;  
 }  
  
 */\*\*  
 \* Метод run() интерфейса Runnable - данный код выполняется  
 \* в дополнительном потоке.  
 \* Алгоритм задачи о спящем брадобрее для серверной стороны  
 \* \*/* @Override  
 public void run()  
 {  
 try  
 {  
 while (!queue.isEmpty())  
 {  
 clientSemaphore.acquire();  
 Query query;  
 synchronized (queue)  
 {  
 query = queue.remove();  
 genSemaphore.release();  
 }  
 System.*out*.println(name + " начал создание " +

query.getGraphName());  
 resList.add(generate(query));  
 System.*out*.println(name + " закончил создание " +

query.getGraphName());  
 }  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 */\*\*  
 \* Генерация графа.  
 \* Имя графа берется из запроса.  
 \*  
 \* @param query - запрос на создание графа. \*/* @Override  
 public synchronized ClassNode generate(Query query)  
 {

*//очередь для создания узлов*  
 Queue<ContainerNode> nodeQueue = new ArrayDeque<>();

*//переменная для хранения обрабатываемого  
 //(вытащенного из очереди) узла* ContainerNode outputNode;

*//переменная для хранения ссылки на созданный узел* ContainerNode tempNode;

*/\* Из запроса получаем имя графа (корневого узла).  
 Создаем узел с таким именем и помещаем в очередь. \*/* ClassNode rootNode = new ClassNode(query.getGraphName());  
 nodeQueue.add(rootNode);  
  
 *//генерация узлов, пока очередь не будет пуста* while (!nodeQueue.isEmpty())  
 {  
 try  
 {  
 Thread.*sleep*(250);  
 }  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 *//Вытаскиваем из чоереди узел, который будет обрабатывать.* outputNode = nodeQueue.poll();  
  
 *//Обработка, если вытащенный узел принадлежит классу ClassNode* if (outputNode instanceof ClassNode)  
 {  
 */\* Для сокращения созданного графа иметь подклассы может*

*только корневой узел.  
 До первой операции корневой узел не имеет состояний.  
 К корневому узлы добавляем от 1 до 2 подклассов\*/* if (((ClassNode) outputNode).getState().isEmpty())  
 {  
 for (int i = 0; i < getRandomNumberUsingInts(1, 3); i++)  
 {  
 tempNode =builder.toClassNodeAddClassNode(outputNode,

"подкласс" + "№" + (i + 1));  
 nodeQueue.offer(tempNode);  
 }  
 }  
  
 *//Для любого узла типа ClassNode создаем от 0 до 3 индивидов* for (int i = 0; i < getRandomNumberUsingInts(0, 4); i++)  
 {  
 tempNode=builder.toClassNodeAddIndividualNode(outputNode,

"индивид" + "№" + (i + 1));  
 nodeQueue.offer(tempNode);  
 }  
 }  
  
 */\* Обработка, если вытащенный узел принадлежит классу*

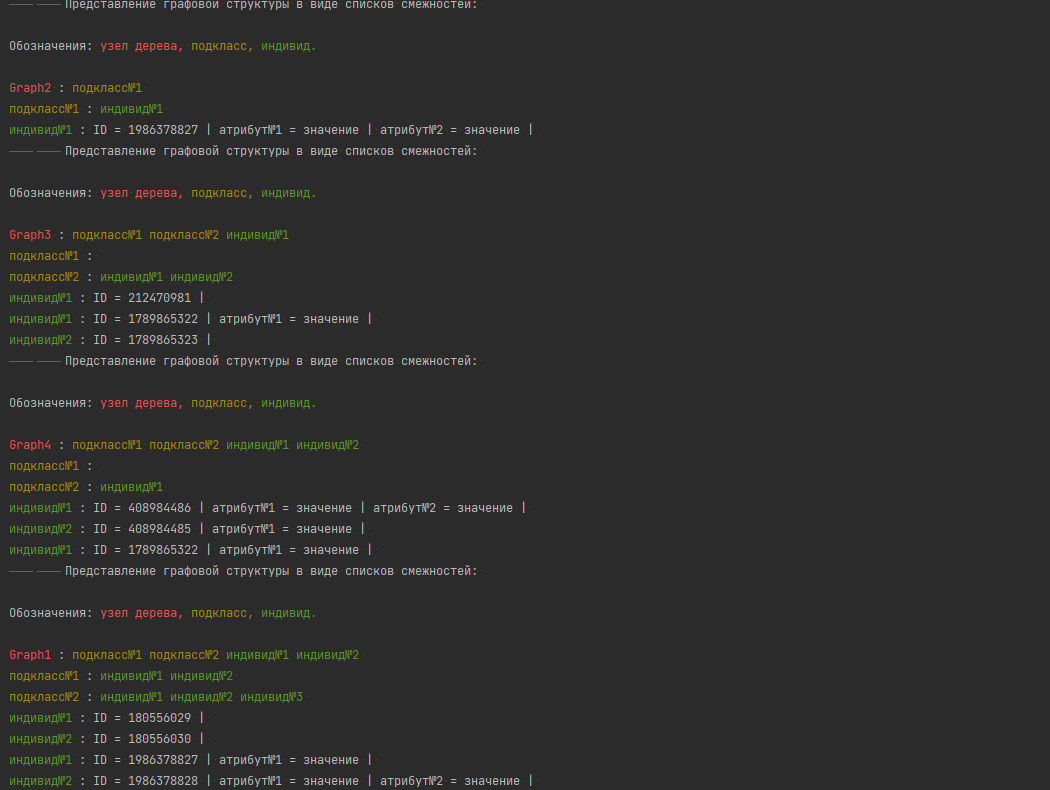
*IndividualNode  
 Создаем для индивида от 0 до 3 атрибутов + Каждый индивид по*

*умолчанию имеет атрибут ID\*/* if (outputNode instanceof IndividualNode)  
 {  
 for (int i = 0; i < getRandomNumberUsingInts(0, 4); i++)  
 {  
 tempNode = builder.toIndividualNodeAddAttributeNode

(outputNode, "атрибут" + "№" + (i + 1), "значение");  
 nodeQueue.offer(tempNode);  
 }  
 }  
 }  
 return rootNode;  
 }  
  
 public void setResList(List<ClassNode> readyList)  
 {  
 this.resList = readyList;  
 }  
  
 */\*\*  
 \* Вспомогательный метод получения случайного значения типа int  
 \* в заданном диапазоне [min; max)  
 \*  
 \* @param min - левая граница диапазона (включительно)  
 \* @param max - правая граница диапазона (не включительно)  
 \* @return случайное значение из заданного диапазона.  
 \* \*/* private int getRandomNumberUsingInts(int min, int max)  
 {  
 Random random = new Random();  
  
 return random.ints(min, max) *//создание бесконечного потока интов*

*//в заданном диапазоне* .findFirst() *//выбираем первый элемент* .getAsInt(); *//получаем его как тип int* }  
}

# Приложение 2

**Результаты тестирования**

