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

по дисциплине

Шаблоны проектирования программного обеспечения

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Грачев Д. С.,   
 Сорокин Е. А.,

Балашов М. А.

(подпись) (фамилия, и.,о.)

\_\_\_18-ИВТ-1\_\_\_\_\_\_\_\_

(шифр группы)

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

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

Нижний Новгород 2020

Вариант 15.

Разработайте и реализуйте объектно-ориентированную модель фабрики с

поддержкой многоуровневого кэширования объектов классов "синглетонов".

Кэш-память первого уровня хранит наиболее часто запрашиваемые объекты,

кэш-память второго уровня хранит объекты, которые запрашиваются реже и

т.д. На последнем уровне вызываются фабричные методы, которые

возвращают ссылки на запрашиваемые объекты. Максимально возможное

количество объектов, хранящихся на каждом уровне, задается пользователем.

Также необходимо вести учет частоты запросов объектов, на основании

которой принимается решение о кэшировании объекта, а также перемещении

его между кэш-памятью разных уровней или уничтожения их кэш-памяти

при снижении частоты запросов.

**Обоснование выбора паттернов:**

По условию задания, были предложены паттерны fabric, singleton. Также исходя из условий был выбран паттерн proxy.

Паттерн singleton реализован в классах Singleton\_0 и Singleton\_1.

Паттерн proxy реализован с помощью интерфейса ICache и классов TopCache, Cache и Fabric: объекты классов TopCache и Cache могут являться заместителями для объектов классов Cache или Fabric, для чего все они реализуют интерфейс ICache.

Паттерн fabric реализован с помощью классов Fabric, Singleton\_0 и Singleton\_1 и интерфейса ISingleton: Fabric вызывает методы get() классов-синглтонов, а интерфейс необходим для поддержания логики паттерна proxy и непосредственно для выполнения задания.

**По 3-й лабораторной:**

Потоки – это клиенты, запрашивающие объекты из кэша. Их моделирует класс ClientThread (клиентский поток).   
Разделяемые ресурсы – это кэш (реализованный в первой лабораторной).

Для управления доступом к кэшу был добавлен класс CacheManager – обёртка над кэшем, содержащая мьютекс. Всё взаимодействие контролирует CacheManager.

После создания поток-клиент входит в бесконечный цикл, в котором с некоторой периодичностью запрашивает определённый объект. Поток завершает работу, когда будет вызван метод interrupt.

Поток создаётся, инициализируется, запускается и уничтожается из главного потока, выполнение реализовано методом run.

Для управления межпоточным взаимодействием используется мьютекс, который не позволяет обратиться к разделяемому ресурсу (кэшу) более чем одному потоку-клиенту.

Код модулей:

AppThreads.java

**package** apps;

**import** cacheFabricThreads.CacheManager;

**import** cacheFabricThreads.ClientThread;

**public** **class** AppThreads {

**public** **static** **void** main(String[] args) {

// создаём кэш через кэш-менеджер

CacheManager cache = **new** CacheManager(1, 1);

// создаём два потока-клиента

Thread thread\_0 = **new** Thread(**new** ClientThread(cache, 2000, "0"));

Thread thread\_1 = **new** Thread(**new** ClientThread(cache, 2000, "1"));

// запускаем потоки

thread\_0.start();

thread\_1.start();

// ждём 5000 миллисекунд

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// создаём ещё два потока-клиента

Thread thread\_2 = **new** Thread(**new** ClientThread(cache, 3000, "0"));

Thread thread\_3 = **new** Thread(**new** ClientThread(cache, 1000, "1"));

// запускаем потоки

thread\_2.start();

thread\_3.start();

// ждём 5000 миллисекунд

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// закрываем потоки

thread\_0.interrupt();

thread\_1.interrupt();

thread\_2.interrupt();

thread\_3.interrupt();

System.***out***.println("Конец");

}

}

Пакет: CacheFabricThreads

CacheManager.java

**package** cacheFabricThreads;

**import** java.util.concurrent.Semaphore;

**import** cacheFabricJava.ICache;

**import** cacheFabricJava.ISingleton;

**import** cacheFabricJava.TopCache;

**public** **class** CacheManager {

**private** ICache cache; // ссылка на кэш

**private** Semaphore mutex = **new** Semaphore(1); // семафор мьютекс

**public** CacheManager(**int** countLevel, **int** memorySize){

// создаём кэш

cache = **new** TopCache(countLevel, memorySize);

}

**public** ISingleton get(String name){

// пытаемся войти в разделяемую секцию

**try** {

mutex.acquire();

} **catch** (InterruptedException e) {

Thread.*currentThread*().interrupt();

}

// запрашиваем нужный объект у кэша

ISingleton res = cache.get(name);

// выходим из разделяемой секции

mutex.release();

**return** res;

}

}

ClientThread.java

**package** cacheFabricThreads;

**public** **class** ClientThread **implements** Runnable {

**private** CacheManager cache; // ссылка на кэш-менеджер

**private** **int** time; // время сна

**private** String name; // имя запрашиваемого объекта

**public** ClientThread(CacheManager c, **int** t, String n){

cache = c;

time = t;

name = n;

}

**public** **void** run() {

**while**(**true**) {

**if** (Thread.*currentThread*().isInterrupted()) **break**;

// обращаемся к кэш-менеджеру

cache.get(name);

System.***out***.println("Получен "+name);

**try**{

Thread.*sleep*(time);

}**catch**(InterruptedException e) {

**break**;

}

}

}

}

Пакет: CacheFabricJava

ICache.java

**package** cacheFabricJava;

**public** **interface** ICache {

ISingleton get(String name); // получение объекта по имени

**void** take(ISingleton obj); // помещение объекта в память кэша

**void** printer(); // выводим память кэша

}

TopCache.java

**package** cacheFabricJava;

**import** java.util.ArrayList;

**public** **class** TopCache **implements** ICache {

**private** ICache down; // Ссылка на уровень ниже

**private** ArrayList<ISingleton> memory; // Массив синглтонов в кэше

**private** **int** memorySize; // Длинна массива

**private** **int** minCounter = 0; // Счетчик самого редкого

**private** **int** maxCounter = 0; // Счетчик самого частого

**public** TopCache(**int** countLevel, **int** memorySize) {

System.***out***.println("Вызван конструктор TopCache");

memory = **new** ArrayList<ISingleton>(memorySize); // создаём память кэша

**this**.memorySize = memorySize;

// создаём кэши нижнего уровня

**if** (countLevel > 0) {

down = **new** Cache(countLevel - 1, memorySize);

}

**else** {

down = **new** Fabric();

}

System.***out***.println("Создан TopCache");

}

**public** ISingleton get(String name) {

System.***out***.println("Вызван TopCache.get()");

ISingleton res = **null**;

// просматриваем память кэша, если она не пуста

**for** (**int** i = 0; i < memory.size(); i++) {

**if** (memory.get(i).getName() == name) {

res = memory.remove(i);

res.upCounter();

**break**;

}

}

// если объект не был найден в памяти кэша, то ищем на следующем уровне кэша

**if** (res == **null**) res = down.get(name);

take(res); // take(взять) - пытаемся разместить найденный объект в памяти кэша

**return** res;

}

**public** **void** take(ISingleton obj) {

System.***out***.println("Вызван TopCache.take()");

ISingleton trash = **null**;

**boolean** objAdded = **false**;

// проверяем подходит ли объект к данному уровню кэша

**if** (obj.getCounter() >= minCounter) {

**if** (memory.size() == memorySize) {

// последний объект в памяти кэша вытаскиваем, дабы освободить место для нового объекта

trash = memory.remove(memorySize - 1);

}

// проходимся по памяти кэша

**for** (**int** i = 0; i < memory.size(); i++) {

**if** (memory.get(i).getCounter() <= obj.getCounter()) {

memory.add(i, obj);

objAdded = **true**;

**break**;

}

}

// добавляем объект в конец памяти кэша, если он не был ранее добавлен в память

**if** (!objAdded) {

memory.add(obj);

objAdded = **true**;

}

// обновляем значения minCounter

**if** (memory.size() == memorySize) {

minCounter = memory.get(memorySize - 1).getCounter();

}

// лишний объект, если таковой имеется, отдаём кэшу следующего уровня

**if** (trash != **null**) {

down.take(trash);

}

}

// отдаём объект кэшу следующего уровня, так как объект не подходит для данного уровня по частоте запросов

**else** down.take(obj);

}

// выводим память в консоль

**public** **void** printer() {

System.***out***.print("\nthis lvl -> ");

**for** (**int** i = 0; i < memory.size();i++) {

System.***out***.print(memory.get(i).getName() + " ");

}

System.***out***.print("\nnext lvl -> ");

down.printer();

}

}

Cache.java

**package** cacheFabricJava;

**import** java.util.ArrayList;

**public** **class** Cache **implements** ICache {

**private** ICache down; // Ссылка на уровень ниже

**private** ArrayList<ISingleton> memory; // Массив синглтонов в кэше

**private** **int** memorySize; // Длинна массива

**private** **int** minCounter = 0; // Счетчик самого редкого

**private** **int** maxCounter = 0; // Счетчик самого частого

**public** Cache(**int** countLevel, **int** memorySize) {

System.***out***.println("Вызван конструктор Cache");

memory = **new** ArrayList<ISingleton>(memorySize); // создаём память кэша

**this**.memorySize = memorySize;

// создаём кэши нижнего уровня

**if** (countLevel > 0) {

down = **new** Cache(countLevel - 1, memorySize);

}

**else** {

down = **new** Fabric();

}

System.***out***.println("Создан Cache");

}

**public** ISingleton get(String name) {

System.***out***.println("Вызван Cache.get()");

ISingleton res = **null**;

// просматриваем память кэша, если она не пуста

**for** (**int** i = 0; i < memory.size(); i++) {

**if** (memory.get(i).getName() == name) {

res = memory.remove(i);

res.upCounter();

**break**;

}

}

// если объект не был найден в памяти кэша, то ищем на следующем уровне кэша

**if** (res == **null**) res = down.get(name);

**return** res;

}

**public** **void** take(ISingleton obj) {

System.***out***.println("Вызван Cache.take()");

ISingleton trash = **null**;

**boolean** objAdded = **false**;

// проверяем подходит ли объект к данному уровню кэша

**if** (obj.getCounter() >= minCounter) {

**if** (memory.size() == memorySize) {

// последний объект в памяти кэша вытаскиваем, дабы освободить место для нового объекта

trash = memory.remove(memorySize - 1);

}

// проходимся по памяти кэша

**for** (**int** i = 0; i < memory.size(); i++) {

**if** (memory.get(i).getCounter() <= obj.getCounter()) {

memory.add(i, obj);

objAdded = **true**;

**break**;

}

}

// добавляем объект в конец памяти кэша, если он не был ранее добавлен в память

**if** (!objAdded) {

memory.add(obj);

objAdded = **true**;

}

// обновляем значения minCounter

**if** (memory.size() == memorySize) {

minCounter = memory.get(memorySize - 1).getCounter();

}

// лишний объект, если таковой имеется, отдаём кэшу следующего уровня

**if** (trash != **null**) {

down.take(trash);

}

}

// отдаём объект кэшу следующего уровня, так как объект не подходит для данного уровня по частоте запросов

**else** down.take(obj);

}

// выводим память в консоль

**public** **void** printer() {

**for** (**int** i = 0; i < memory.size();i++) {

System.***out***.print(memory.get(i).getName() + " ");

}

System.***out***.print("\nnext lvl -> ");

down.printer();

}

}

Fabric.java

**package** cacheFabricJava;

**public** **class** Fabric **implements** ICache {

**public** Fabric() {

System.***out***.println("Создан Fabric");

}

**public** ISingleton get(String name) {

System.***out***.println("Вызван Fabric.get()");

ISingleton res = **null**;

// обращаемся к нужному классу

**switch** (name) {

**case**("0"):

res = Singleton\_0.*get*();

**break**;

**case**("1"):

res = Singleton\_1.*get*();

**break**;

}

**return** res;

}

**public** **void** take(ISingleton obj) {

System.***out***.println("Вызван Fabric.take()");

}

**public** **void** printer() {

System.***out***.println("конец!\n");

}

}

ISingleton.java  
**package** cacheFabricJava;

**public** **interface** ISingleton {

**default** **int** getCounter() {

**return** 0;

} // получение популярности объекта

**int** upCounter(); // увеличение популяности объекта

**default** String getName() {

**return** "";

} // получение имени объекта

}

Singleton\_0.java

**package** cacheFabricJava;

**public** **class** Singleton\_0 **implements** ISingleton {

**private** **static** Singleton\_0 *inst* = **null**;

**private** **static** **int** *counter* = 0;

**private** Singleton\_0() {}

// популярность объекта

**public** **int** getCounter() {

**return** *counter*;

}

// увеличиваем популярность объекта

**public** **int** upCounter() {

*counter*++;

**return** *counter*;

}

// возвращает имя объекта

**public** String getName() {

**return** "0";

}

// получение объекта синглтона

**public** **static** Singleton\_0 get() {

**if** (*inst* == **null**) *inst* = **new** Singleton\_0();

*counter*++;

**return** *inst*;

}

}

Singleton\_1.java  
**package** cacheFabricJava;

**public** **class** Singleton\_1 **implements** ISingleton {

**private** **static** Singleton\_1 *inst* = **null**;

**private** **static** **int** *counter* = 0;

**private** Singleton\_1() {}

// популярность объекта

**public** **int** getCounter() {

**return** *counter*;

}

// увеличиваем популярность объекта

**public** **int** upCounter() {

*counter*++;

**return** *counter*;

}

// возвращает имя объекта

**public** String getName() {

**return** "1";

}

// получение объекта синглтона

**public** **static** Singleton\_1 get() {

**if** (*inst* == **null**) *inst* = **new** Singleton\_1();

*counter*++;

**return** *inst*;

}

}