**Контрольное задание**

Используя прикладную задачу л.р. №1 на языке Java выполнить индивидуальное задания №1 (см. п.1) и индивидуальное задание №2 (см. п.4). Оформить отчёт о решении согласно п.2 л.р. №1.

# 1  Индивидуальное задание №1

Тема: "Параллельное выполнение. Шаблон Worker Thread"

Цель: ознакомиться с методами управления параллельно выполняющимися задачами; реализовать и продемонстрировать механизм обслуживания очереди задач, применив соответствующие шаблоны проектирования.

Продемонстрировать возможность параллельной обработки элементов коллекции (поиск минимума, максимума, вычисление среднего значения, отбор по критерию, статистическая обработка и т.д.).

Управление очередью задач (команд) реализовать с помощью шаблона Worker Thread.

Обеспечить диалоговый интерфейс с пользователем.

Разработать класс для тестирования функциональности приложения.

Использовать комментарии для автоматической генерации документации средствами javadoc.

# 2  Пример проекта заданиЯ №1

## 2.1  Разработка программы

Реализуем классы, структура которых соответствует схеме п.2.1.2.

Разработаем класс MainTest для проведения теста разработанных классов –CommandQueue, MaxCommand, AvgCommand, MinMaxCommand. Реализуем методы:

setUpBeforeClass() – выполняется первым;

tearDownAfterClass() – выполняется последним;

testMax() – проверка основной функциональности класса MaxCommand;

testAvg() – проверка основной функциональности класса AvgCommand;

testMin() – проверка основной функциональности класса MinMaxCommand;

testMaxQueue() – проверка основной функциональности класса CommandQueue с задачей MaxCommand;

testAvgQueue() – проверка основной функциональности класса CommandQueue с задачей AvgCommand;

testMinQueue() – проверка основной функциональности класса CommandQueue с задачей MinMaxCommand.

В процессе разработки необходимо обеспечить прохождение всех тестов.

### 2.1.1  Используемые средства ООП

Для некоторых задач удобно организовать параллельное выполнение нескольких частей программы. Каждая из этих самостоятельных подзадач называется потоком (thread). Существует системный механизм, который обеспечивает совместное использованием процессора.

Модель потоков в языке Java является программным механизмом, упрощающим одновременное выполнение нескольких операций в одной и той же программе. Процессор периодически выделяет каждому потоку некоторый отрезок времени. Для каждого потока все выглядит так, словно процессор используется в монопольном режиме, но на самом деле время процессора разделяется между всеми существующими в программе потоками. Ускорение можно получить на многопроцессорном компьютере. При использовании потоков нет нужды учитывать эти тонкости – код не зависит от того, на скольких процессорах будет исполняться. Таким образом, потоки предоставляют механизм масштабирования производительности – если программа работает слишком медленно, можно добиться ускорения, используя многопроцессорную систему, при этом, не переписывая программу заново.

При программировании параллельно исполняемых потоков нужно учитывать следующие моменты:

1)       Программу можно разделить на несколько независимых задач.

2)       Необходимо заранее предусмотреть всевозможные проблемы, возникающие при завершении задач.

3)       Задачи, работающие с общими ресурсами, могут мешать друг другу. Основным средством предотвращения конфликтов является блокировка.

4)       В неаккуратно спроектированных многозадачных системах возможны взаимные блокировки.

Основные причины использования параллельного выполнения:

         управление несколькими подзадачами, одновременное выполнение которых позволяет эффективнее распоряжаться ресурсами вычислительной системы (включая возможность распределения этих задач по нескольким процессорам);

         улучшенная организация кода;

         удобство для пользователя.

Обрабатывающий  шаблон Worker Thread (Background Thread, Thread Pool) предназначен для улучшение пропускной способности и минимизация средней задержки при реализации параллельного выполнения.

Для того, чтобы реализовать средства поддержки многопоточности корректно, необходимо обладать определенными навыками. Один из способов максимизации эффективности многопоточного приложения состоит в использовании того факта, что не все выполненные в виде потоков задачи приложения имеют одинаковый приоритет. Для некоторых задач на первом месте стоит время выполнения. Другие же просто должны быть выполнены, а когда именно произойдет это выполнение – не столь важно.

Разработчик может отделить подобные задачи от приложения и воспользоваться шаблоном Worker Thread. Созданный в соответствии с этим шаблоном обработчик потока будет выбирать из очереди задачи и выполнять их в отдельном потоке. По окончании очередной задачи обработчик выбирает из очереди следующую задачу и все повторяется сначала.

Создание многопоточного приложения при использовании шаблона Worker Thread значительно упрощается, поскольку в тех случаях, когда не важно, как скоро будет выполнена задача, разработчику достаточно просто поместить ее в очередь, а все остальное сделает обработчик потока. Программный код такого приложения также упрощается, так как все объекты, работающие с потоками, скрыты внутри обработчика потока и очереди.

Шаблон Worker Thread рекомендуется использовать когда:

         нужно повысить пропускную способность приложения;

         необходимо обеспечить одновременное выполнение разных фрагментов кода.

Для реализации потоков в приложении можно создать новый объект Thread и запустить его на выполнение. Поток, представленный этим объектом, выполнит всю порученную ему работу и автоматически завершится. Однако создание экземпляра потока – это расточительный процесс с точки зрения производительности, он требует немало времени и позволяет выполнить только одну задачу. Более эффективный способ заключается в создании объекта-"долгожителя" – специального обработчика потока, который будет выполнять одну задачу за другой.

В выполнении такой работы и состоит сущность шаблона Worker Thread. Обработчик потока, реализованный в соответствии с этим шаблоном, выполняет одну за другой множество не связанных друг с другом задач. Не нужно создавать новый поток при каждом запуске новой задачи – достаточно лишь передать задачу уже существующему обработчику потока, который позаботится об остальном.

Возможна ситуация, когда обработчик потока занят выполнением очередной задачи, а приложение уже подготовило следующую задачу. В такой ситуации можно предложить одно из следующих решений.

         Приложение ждет до тех пор, пока обработчик потока не освободится от текущей задачи. Это решение очевидно, но оно практически сводит на нет все преимущества, предоставляемые многопоточностью.

         Приложение создает новый экземпляр обработчика потока всякий раз, когда текущий обработчик потока недоступен. Такое решение по сути, является возвратом к традиционной технологии, так как возможны ситуации, при которых для каждой новой задачи будет создаваться отдельный поток.

Наилучшее решение проблемы временно недоступного обработчика потока состоит в том, чтобы сохранить задачи в очереди до тех пор, пока обработчик потока не освободится. Приложение помещает каждую новую задачу в очередь, а обработчик потока, закончив выполнение очередной задачи, проверяет, есть ли в очереди новые задачи, и если таковые имеются, запускает следующую задачу на выполнение. Это не дает преимущества относительно скорости выполнения задач, но освобождает приложение от необходимости ожидания, пока освободится обработчик потока.

Если нет задач для выполнения, обработчик периодически проверяет очередь. Помещение задачи в очередь – гораздо менее требовательный с точки зрения производительности процесс, чем создание нового потока.

Шаблон Worker Thread оказывает влияние на производительность в нескольких направлениях.

1)       Клиенту для запуска различных заданий не нужно создавать несколько объектов потоков. Нужно лишь поместить задачу в очередь, что требует значительно меньших накладных расходов, чем создание объекта потока.

2)       Существующий, но не выполняющийся поток, снижает производительность, так как планировщик выделяет часть машинного времени для выполнения потока, находящегося в состоянии готовности для выполнения. Создание и запуск потока для каждой задачи означает, что планировщик должен выделять ресурсы каждому такому потоку индивидуально. В сумме потери времени на такое планирование значительно больше, чем потери, которые возникают при наличии постоянно работающего обработчика потока. Иными словами, чем больше потоков, тем выше накладные расходы на их планирование. Если же задание находится в очереди и, соответственно, не выполняется, оно вообще не расходует машинного времени.

3)       Когда задачи являются взаимозависимыми и если очередь последовательна, такая ситуация может привести к блокировке системы. Для решения этой проблемы можно использовать несколько подходов:

         Создается столько обработчиков потоков, сколько задач необходимо выполнять одновременно. В приложении нужно организовать расширяемый пул потоков.

         В очередь разрешается помещать только те задачи, которые не зависят от других задач. В таких случаях клиент должен не помещать задачу в очередь, а создать экземпляр собственного потока или запустить отдельную очередь с обработчиком потока.

         Создается интеллектуальная очередь, которая может установить, какие задачи работают совместно, и принять решение, когда ту или иную задачу передать обработчику потока. Этот подход нужно применять только тогда, когда не осталось других возможностей, поскольку такая интеллектуальная очередь должна быть тесно связана с приложением, а сопровождение ее программного кода может стать очень трудоемким занятием.

### 2.1.2  Иерархия и структура классов

Структура классов и схема их отношений приведена на рис.1.

### 2.1.3  Описание программы

Для реализации параллельного выполнения при обработке коллекции объектов использовались шаблоны Worker Thread, Command и Factory Method.

Для хранения задач используется очередь, представленная интерфейсом Queue. Интерфейс Queue определяет два базовых метода: put и take. Эти методы используются для добавления в очередь и удаления из нее задач, представленных интерфейсом Command.

Класс CommandQueue реализует интерфейс Queue и создает обработчик потока, который выполняет объекты, реализующие интерфейс Command. Класс Worker, который является внутренним классом класса CommandQueue, реализует метод run, предназначенный для периодической проверки очереди на наличие в ней готовых к выполнению задач. Когда такая задача появляется, обработчик потока извлекает ее из очереди и запускает метод execute.

Для параллельного выполнения предназначены три класса, реализующие интерфейс Command:

MaxCommand – предназначен для обработки коллекции объектов и нахождения максимального значения.

MinMaxCommand – предназначен для обработки коллекции объектов, выполняет поиск минимального положительного и максимального отрицательного значения.

AvgCommand – предназначен для обработки коллекции объектов, вычисляет среднее арифметическое значение.

Класс ExecuteConsoleCommand реализует консольную команду "Execute all threads" (шаблон Command). При исполнении метода execute() создаются две очереди задач и выполняется ожидание завершения их обработки.

При написании исходного кода используем стиль комментариев документации javadoc.

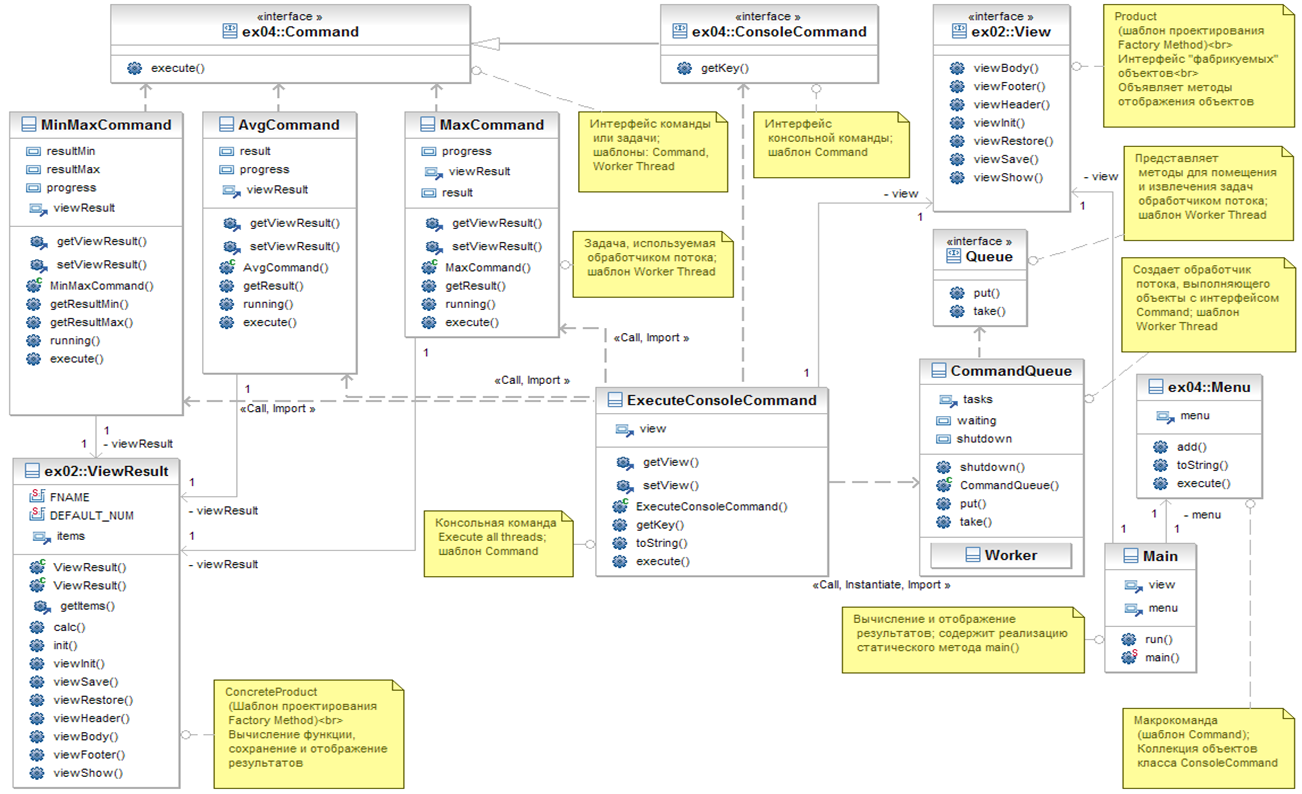
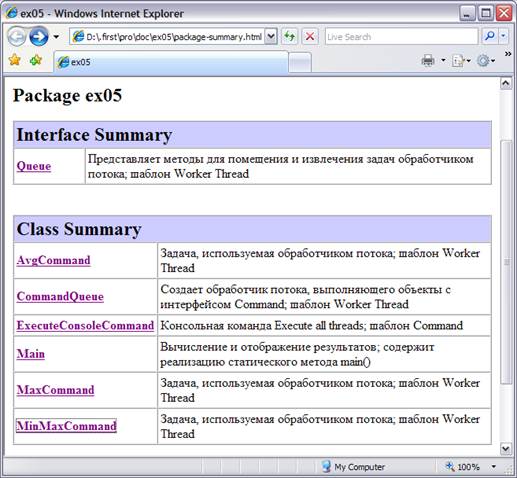


Рис.1. Схема классов и их отношений

Структура проекта:

|  |  |
| --- | --- |
| Папка src  Untitled-1 | Папка test  Untitled-2 copy |

Выполним генерацию документации:



После проверки работоспособности готовой программы, создадим исполняемый JAR файл ex05.jar

## 2.2  Текст программы

### 2.2.1  AvgCommand.java

**package** ex05;

**import** java.util.concurrent.TimeUnit;

**import** ex01.Item2d;

**import** ex02.ViewResult;

**import** ex04.Command;

/\*\* Задача, используемая

 \* обработчиком потока;

 \* шаблон Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Command

 \* **@see** CommandQueue

 \*/

**public** **class** AvgCommand **implements** Command /\*, Runnable \*/ {

/\*\* Хранит результат обработки коллекции \*/

**private** **double** result = 0.0;

/\*\* Флаг готовности результата \*/

**private**  **int** progress = 0;

/\*\* Обслуживает коллекцию объектов {@linkplain ex01.Item2d}  \*/

**private** ViewResult viewResult;

/\*\* Возвращает поле {@linkplain MaxCommand#viewResult}

 \* **@return** значение {@linkplain MaxCommand#viewResult}

 \*/

**public** ViewResult getViewResult() {

**return** viewResult;

}

/\*\* Устанавливает поле {@linkplain MaxCommand#viewResult}

 \* **@param** viewResult значение для {@linkplain MaxCommand#viewResult}

 \* **@return** новое значение {@linkplain MaxCommand#viewResult}

 \*/

**public** ViewResult setViewResult(ViewResult viewResult) {

**return** **this**.viewResult = viewResult;

}

/\*\* Инициализирует поле {@linkplain MaxCommand#viewResult}

 \* **@param** viewResult объект класса {@linkplain ViewResult}

 \*/

**public** AvgCommand(ViewResult viewResult) {

**this**.viewResult = viewResult;

}

/\*\* Возвращает результат

 \* **@return** поле {@linkplain MaxCommand#result}

 \*/

**public** **double** getResult() {

**return** result;

}

/\*\* Проверяет готовность результата

 \* **@return** false - если результат найден, иначе - true

 \* **@see** MaxCommand#result

 \*/

**public** **boolean** running() {

**return** progress < 100;

}

/\*\* Используется обработчиком потока {@linkplain CommandQueue};

 \* шаблон Worker Thread

 \*/

@Override

**public** **void** execute() {

       progress = 0;

       System.*out*.println("Average executed...");

       result = 0.0;

**int** idx = 1, size = viewResult.getItems().size();

**for** (Item2d item : viewResult.getItems()) {

              result += item.getY();

              progress = idx \* 100 / size;

**if** (idx++ % (size / 2) == 0) {

                     System.*out*.println("Average " + progress + "%");

              }

**try** {

                     TimeUnit.*MILLISECONDS*.sleep(2000 / size);

              } **catch** (InterruptedException e) {

                     System.*err*.println(e);

              }

       }

       result /= size;

       System.*out*.println("Average done. Result = " + String.*format*("%.2f",result));

       progress = 100;

}

/\*\*

**@Override**

public void run() {

       execute();

}

/\*\*/

}

### 2.2.2  CommandQueue.java

**package** ex05;

**import** java.util.Vector;

**import** ex04.Command;

/\*\* Создает обработчик

 \* потока, выполняющего

 \* объекты с интерфейсом

 \* Command; шаблон

 \* Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Command

 \*/

**public** **class** CommandQueue **implements** Queue {

/\*\* Очередь задач \*/

**private** Vector<Command> tasks;

/\*\* Флаг ожидания \*/

**private** **boolean** waiting;

/\*\* Флаг завершения \*/

**private** **boolean** shutdown;

/\*\* Устанавливает флаг завершения \*/

**public** **void** shutdown() {

       shutdown = **true**;

}

/\*\* Инициализация {@linkplain CommandQueue#tasks}

 \* {@linkplain CommandQueue#waiting}

 \* {@linkplain CommandQueue#waiting};

 \* создает поток для класса {@linkplain CommandQueue.Worker}

 \*/

**public** CommandQueue() {

       tasks = **new** Vector<Command>();

       waiting = **false**;

**new** Thread(**new** Worker()).start();

}

@Override

**public** **void** put(Command r) {

       tasks.add(r);

**if** (waiting) {

**synchronized** (**this**) {

                     notifyAll();

              }

       }

}

@Override

**public** Command take() {

**if** (tasks.isEmpty()) {

**synchronized** (**this**) {

                     waiting = **true**;

**try** {

                            wait();

                     } **catch** (InterruptedException ie) {

                            waiting = **false**;

                     }

              }

       }

**return** (Command)tasks.remove(0);

}

/\*\* Обслуживает очередь

 \* задач; шаблон

 \* Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Runnable

 \*/

**private** **class** Worker **implements** Runnable {

       /\*\* Извлекает из очереди

        \* готовые к выполнению

        \* задачи; шаблон

        \* Worker Thread \*/

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

**while** (!shutdown) {

                     Command r = take();

                     r.execute();

              }

       }

}

}

### 2.2.3  ExecuteConsoleCommand.java

**package** ex05;

/\*\*

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*/

**import** java.util.concurrent.TimeUnit;

**import** ex02.View;

**import** ex02.ViewResult;

**import** ex04.ConsoleCommand;

/\*\* Консольная команда

 \* Execute all threads;

 \* шаблон Command

 \* **@author** xone

 \* **@version** 1.0

 \*/

**public** **class** ExecuteConsoleCommand **implements** ConsoleCommand {

/\*\* Объект, реализующий интерфейс {@linkplain View};

 \* обслуживает коллекцию объектов {@linkplain ex01.Item2d}

 \*/

**private** View view;

/\*\* Возвращает поле {@linkplain ExecuteConsoleCommand#view}

 \* **@return** значение {@linkplain ExecuteConsoleCommand#view}

 \*/

**public** View getView() {

**return** view;

}

/\*\* Устанавливает поле {@linkplain ExecuteConsoleCommand#view}

 \* **@param** view значение для {@linkplain ExecuteConsoleCommand#view}

 \* **@return** новое значение {@linkplain ExecuteConsoleCommand#view}

 \*/

**public** View setView(View view) {

**return** **this**.view = view;

}

/\*\* Инициализирует поле {@linkplain ExecuteConsoleCommand#view}

 \* **@param** view объект, реализующий {@linkplain View}

 \*/

**public** ExecuteConsoleCommand(View view) {

**this**.view = view;

}

@Override

**public** **char** getKey() {

**return** 'e';

}

@Override

**public** String toString() {

**return** "'e'xecute";

}

@Override

**public** **void** execute() {

       /\*\*/

       CommandQueue queue1 = **new** CommandQueue();

       CommandQueue queue2 = **new** CommandQueue();

       /\*\*

       ExecutorService exec1 = Executors.newSingleThreadExecutor();

       ExecutorService exec2 = Executors.newSingleThreadExecutor();

       /\*\*/

       MaxCommand maxCommand = **new** MaxCommand((ViewResult)view);

       AvgCommand avgCommand = **new** AvgCommand((ViewResult)view);

       MinMaxCommand minMaxCommand = **new** MinMaxCommand((ViewResult)view);

       System.*out*.println("Execute all threads...");

       /\*\*

       exec1.execute(minMaxCommand);

       exec2.execute(maxCommand);

       exec2.execute(avgCommand);

       /\*\*/

       queue1.put(minMaxCommand);

       queue2.put(maxCommand);

       queue2.put(avgCommand);

       /\*\*/

**try** {

**while** (avgCommand.running() ||

                            maxCommand.running() ||

                            minMaxCommand.running()) {

                     TimeUnit.*MILLISECONDS*.sleep(100);

              }

              /\*\*

              exec1.shutdown();

              exec2.shutdown();

              /\*\*/

              queue1.shutdown();

              queue2.shutdown();

              /\*\*/

              TimeUnit.*SECONDS*.sleep(1);

       } **catch** (InterruptedException e) {

              System.*err*.println(e);

       }

       System.*out*.println("All done.");

}

}

### 2.2.4  Main.java

**package** ex05;

**import** ex02.View;

**import** ex02.ViewableResult;

**import** ex04.ChangeConsoleCommand;

**import** ex04.GenerateConsoleCommand;

**import** ex04.Menu;

**import** ex04.ViewConsoleCommand;

/\*\* Вычисление и отображение

 \* результатов; содержит реализацию

 \* статического метода main()

 \* **@author** xone

 \* **@version** 5.0

 \* **@see** Main#main

 \*/

**public** **class** Main {

/\*\* Объект, реализующий интерфейс {@linkplain View};

 \* обслуживает коллекцию объектов {@linkplain ex01.Item2d};

 \* инициализируется с помощью Factory Method

 \*/

**private** View view = **new** ViewableResult().getView();

/\*\* Объект класса {@linkplain Menu};

 \* макрокоманда (шаблон Command)

 \*/

**private** Menu menu = **new** Menu();

/\*\* Обработка команд пользователя \*/

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

       menu.add(**new** ViewConsoleCommand(view));

       menu.add(**new** GenerateConsoleCommand(view));

       menu.add(**new** ChangeConsoleCommand(view));

       menu.add(**new** ExecuteConsoleCommand(view));

       menu.execute();

}

/\*\* Выполняется при запуске программы

 \* **@param** args параметры запуска программы

 \*/

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

       Main main = **new** Main();

       main.run();

}

}

### 2.2.5  MaxCommand.java

**package** ex05;

**import** java.util.concurrent.TimeUnit;

**import** ex02.ViewResult;

**import** ex04.Command;

/\*\* Задача, используемая

 \* обработчиком потока;

 \* шаблон Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Command

 \* **@see** CommandQueue

 \*/

**public** **class** MaxCommand **implements** Command /\*, Runnable \*/ {

/\*\* Хранит результат обработки коллекции \*/

**private** **int** result = -1;

/\*\* Флаг готовности результата \*/

**private** **int** progress = 0;

/\*\* Обслуживает коллекцию объектов {@linkplain ex01.Item2d}  \*/

**private** ViewResult viewResult;

/\*\* Возвращает поле {@linkplain MaxCommand#viewResult}

 \* **@return** значение {@linkplain MaxCommand#viewResult}

 \*/

**public** ViewResult getViewResult() {

**return** viewResult;

}

/\*\* Устанавливает поле {@linkplain MaxCommand#viewResult}

 \* **@param** viewResult значение для {@linkplain MaxCommand#viewResult}

 \* **@return** новое значение {@linkplain MaxCommand#viewResult}

 \*/

**public** ViewResult setViewResult(ViewResult viewResult) {

**return** **this**.viewResult = viewResult;

}

/\*\* Инициализирует поле {@linkplain MaxCommand#viewResult}

 \* **@param** viewResult объект класса {@linkplain ViewResult}

 \*/

**public** MaxCommand(ViewResult viewResult) {

**this**.viewResult = viewResult;

}

/\*\* Возвращает результат

 \* **@return** поле {@linkplain MaxCommand#result}

 \*/

**public** **int** getResult() {

**return** result;

}

/\*\* Проверяет готовность результата

 \* **@return** false - если результат найден, иначе - true

 \* **@see** MaxCommand#result

 \*/

**public** **boolean** running() {

**return** progress < 100;

}

/\*\* Используется обработчиком потока {@linkplain CommandQueue};

 \* шаблон Worker Thread

 \*/

@Override

**public** **void** execute() {

       progress = 0;

       System.*out*.println("Max executed...");

**int** size = viewResult.getItems().size();

       result = 0;

**for** (**int** idx = 1; idx < size; idx++) {

**if** (viewResult.getItems().get(result).getY() <

                            viewResult.getItems().get(idx).getY()) {

                     result = idx;

              }

              progress = idx \* 100 / size;

**if** (idx % (size / 3) == 0) {

                     System.*out*.println("Max " + progress + "%");

              }

**try** {

                     TimeUnit.*MILLISECONDS*.sleep(3000 / size);

              } **catch** (InterruptedException e) {

                     System.*err*.println(e);

              }

       }

       System.*out*.println("Max done. Item #" + result +

                     " found: " + viewResult.getItems().get(result));

       progress = 100;

}

/\*\*

**@Override**

public void run() {

       execute();

}

/\*\*/

}

### 2.2.6  MinMaxCommand.java

**package** ex05;

**import** java.util.concurrent.TimeUnit;

**import** ex01.Item2d;

**import** ex02.ViewResult;

**import** ex04.Command;

/\*\* Задача, используемая

 \* обработчиком потока;

 \* шаблон Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Command

 \* **@see** CommandQueue

 \*/

**public** **class** MinMaxCommand **implements** Command /\*, Runnable \*/ {

/\*\* Хранит результат обработки коллекции \*/

**private** **int** resultMin = -1;

/\*\* Хранит результат обработки коллекции \*/

**private** **int** resultMax = -1;

/\*\* Флаг готовности результата \*/

**private** **int** progress = 0;

/\*\* Обслуживает коллекцию объектов {@linkplain ex01.Item2d}  \*/

**private** ViewResult viewResult;

/\*\* Возвращает поле {@linkplain MinMaxCommand#viewResult}

 \* **@return** значение {@linkplain MinMaxCommand#viewResult}

 \*/

**public** ViewResult getViewResult() {

**return** viewResult;

}

/\*\* Устанавливает поле {@linkplain MinMaxCommand#viewResult}

 \* **@param** viewResult значение для {@linkplain MinMaxCommand#viewResult}

 \* **@return** новое значение {@linkplain MinMaxCommand#viewResult}

 \*/

**public** ViewResult setViewResult(ViewResult viewResult) {

**return** **this**.viewResult = viewResult;

}

/\*\* Инициализирует поле {@linkplain MinMaxCommand#viewResult}

 \* **@param** viewResult объект класса {@linkplain ViewResult}

 \*/

**public** MinMaxCommand(ViewResult viewResult) {

**this**.viewResult = viewResult;

}

/\*\* Возвращает результат

 \* **@return** поле {@linkplain MinMaxCommand#resultMin}

 \*/

**public** **int** getResultMin() {

**return** resultMin;

}

/\*\* Возвращает результат

 \* **@return** поле {@linkplain MinMaxCommand#resultMax}

 \*/

**public** **int** getResultMax() {

**return** resultMax;

}

/\*\* Проверяет готовность результата

 \* **@return** false - если результат найден, иначе - true

 \*/

**public** **boolean** running() {

**return** progress < 100;

}

/\*\* Используется обработчиком потока {@linkplain CommandQueue};

 \* шаблон Worker Thread

 \*/

@Override

**public** **void** execute() {

       progress = 0;

       System.*out*.println("MinMax executed...");

**int** idx = 0, size = viewResult.getItems().size();

**for** (Item2d item : viewResult.getItems()) {

**if** (item.getY() < 0) {

**if** ((resultMax == -1) ||

                                  (viewResult.getItems().get(resultMax).getY() < item.getY())) {

                            resultMax = idx;

                     }

              } **else** {

**if** ((resultMin == -1) ||

                                  (viewResult.getItems().get(resultMin).getY() > item.getY())) {

                            resultMin = idx;

                     }

              }

              idx++;

              progress = idx \* 100 / size;

**if** (idx % (size / 5) == 0) {

                     System.*out*.println("MinMax " + progress + "%");

              }

**try** {

                     TimeUnit.*MILLISECONDS*.sleep(5000 / size);

              } **catch** (InterruptedException e) {

                     System.*err*.println(e);

              }

       }

       System.*out*.print("MinMax done. ");

**if** (resultMin > -1) {

              System.*out*.print("Min positive #" + resultMin + " found: " +

                            String.*format*("%.2f.", viewResult.getItems().get(resultMin).getY()));

       } **else** {

              System.*out*.print("Min positive not found.");

       }

**if** (resultMax > -1) {

              System.*out*.println(" Max negative #" + resultMax + " found: " +

                            String.*format*("%.2f.", viewResult.getItems().get(resultMax).getY()));

       } **else** {

              System.*out*.println(" Max negative item not found.");

       }

       progress = 100;

}

/\*\*

**@Override**

public void run() {

       execute();

}

/\*\*/

}

### 2.2.7  Queue.java

**package** ex05;

**import** ex04.Command;

/\*\* Представляет

 \* методы для помещения

 \* и извлечения задач

 \* обработчиком потока;

 \* шаблон Worker Thread

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Command

 \*/

**public** **interface** Queue {

/\*\* Добавляет новую задачу в очередь;

 \* шаблон Worker Thread

 \* **@param** cmd задача

 \*/

**void** put(Command cmd);

/\*\* Удаляет задачу из очереди;

 \* шаблон Worker Thread

 \* **@return** удаляемая задача

 \*/

Command take();

}

### 2.2.8  MainTest.java

**package** ex05;

**import** **static** org.junit.Assert.\*;

**import** java.util.concurrent.TimeUnit;

**import** org.junit.AfterClass;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**import** ex02.ViewResult;

/\*\* Тестирование

 \* разработанных классов

 \* **@author** xone

 \* **@version** 5.0

 \* **@see** CommandQueue

 \* **@see** MaxCommand

 \* **@see** AvgCommand

 \* **@see** MinMaxCommand

 \*/

**public** **class** MainTest {

**private** **final** **static** **int** *N* = 1000;

**private** **static** ViewResult *view* = **new** ViewResult(*N*);

**private** **static** MaxCommand *max1* = **new** MaxCommand(*view*);

**private** **static** MaxCommand *max2* = **new** MaxCommand(*view*);

**private** **static** AvgCommand *avg1* = **new** AvgCommand(*view*);

**private** **static** AvgCommand *avg2* = **new** AvgCommand(*view*);

**private** **static** MinMaxCommand *min1* = **new** MinMaxCommand(*view*);

**private** **static** MinMaxCommand *min2* = **new** MinMaxCommand(*view*);

**private** CommandQueue queue = **new** CommandQueue();

/\*\* Выполняется первым \*/

@BeforeClass

**public** **static** **void** setUpBeforeClass() {

*view*.viewInit();

*assertEquals*(*N*, *view*.getItems().size());

}

/\*\* Выполняется последним \*/

@AfterClass

**public** **static** **void** tearDownAfterClass() {

*assertEquals*(*max1*.getResult(), *max2*.getResult());

*assertEquals*(*avg1*.getResult(), *avg2*.getResult(), .1e-10);

*assertEquals*(*min1*.getResultMax(), *min2*.getResultMax());

*assertEquals*(*min1*.getResultMin(), *min2*.getResultMin());

}

/\*\* Проверка основной функциональности класса {@linkplain MaxCommand} \*/

@Test

**public** **void**   testMax() {

*max1*.execute();

*assertTrue*( *max1*.getResult() > -1);

}

/\*\* Проверка основной функциональности класса {@linkplain AvgCommand} \*/

@Test

**public** **void**   testAvg() {

*avg1*.execute();

*assertTrue*( *avg1*.getResult() != 0.0);

}

/\*\* Проверка основной функциональности класса {@linkplain MinMaxCommand} \*/

@Test

**public** **void**   testMin() {

*min1*.execute();

*assertTrue*( *min1*.getResultMin() > -1);

*assertTrue*( *min1*.getResultMax() > -1);

}

/\*\* Проверка основной функциональности класса

 \* {@linkplain CommandQueue} с задачей {@linkplain MaxCommand}

 \*/

@Test

**public** **void**   testMaxQueue() {

       queue.put(*max2*);

**try** {

**while** (*max2*.running()) {

                     TimeUnit.*MILLISECONDS*.sleep(100);

              }

              queue.shutdown();

              TimeUnit.*SECONDS*.sleep(1);

       } **catch** (InterruptedException e) {

*fail*(e.toString());

       }

}

/\*\* Проверка основной функциональности класса

 \* {@linkplain CommandQueue} с задачей {@linkplain AvgCommand}

 \*/

@Test

**public** **void**   testAvgQueue() {

       queue.put(*avg2*);

**try** {

**while** (*avg2*.running()) {

                     TimeUnit.*MILLISECONDS*.sleep(100);

              }

              queue.shutdown();

              TimeUnit.*SECONDS*.sleep(1);

       } **catch** (InterruptedException e) {

*fail*(e.toString());

       }

}

/\*\* Проверка основной функциональности класса

 \* {@linkplain CommandQueue} с задачей {@linkplain MinMaxCommand}

 \*/

@Test

**public** **void**   testMinQueue() {

       queue.put(*min2*);

**try** {

**while** (*min2*.running()) {

                     TimeUnit.*MILLISECONDS*.sleep(100);

              }

              queue.shutdown();

              TimeUnit.*SECONDS*.sleep(1);

       } **catch** (InterruptedException e) {

*fail*(e.toString());

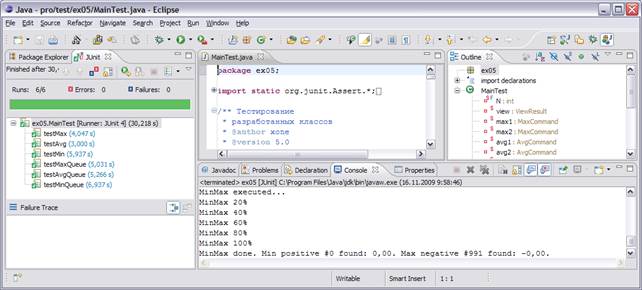
       }

}

}

## 2.3  Результаты тестирования

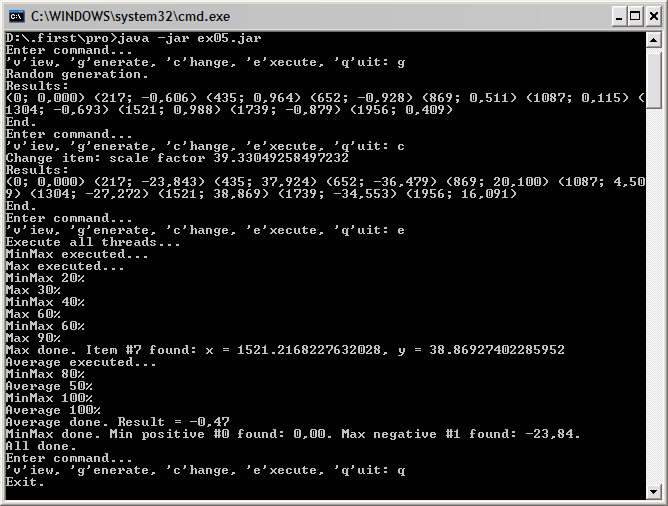
Выполним ex05.MainTest как JUnit Test



Выполним запуск программы из командной строки:

java -jar ex05.jar

В результате выполнения получим:



# 3  Заключение по результату решения задания №1

Разработали программу решения задачи индивидуального задания. Результаты тестирования подтверждают корректность используемых алгоритмов.

Для реализации параллельного выполнения при обработке коллекции объектов использовались шаблоны Worker Thread, Command и Factory Method.

Продемонстрирована возможность управления очередью параллельно исполняемых задач на примере реализации алгоритмов обработки коллекции объектов.

Для тестирования программы использовались средства JUnit.

# 4  Индивидуальное задание №2

Тема: "Использование рефлексии. Аннотации. Шаблон Observer"

Цель: ознакомиться с механизмом рефлексии; приобрести навыки использования средств аннотирования; реализовать обслуживание объектов на основе шаблона проектирования Observer.

Разработать иерархию классов согласно шаблону Observer и продемонстрировать возможность обслуживания разработанной ранее коллекции (наблюдаемый объект, Observable) различными (не менее двух) наблюдателями (Observers) – отслеживание изменений, упорядочивание, вывод, отображение и т.д. При реализации иерархии классов использовать средства аннотирования (Annotation). Отметить особенности различных политик удержания аннотаций (annotation retention policies). Продемонстрировать поддержку классами концепции рефлексии (Reflection).

Разработать класс для тестирования функциональности приложения.

Использовать комментарии для автоматической генерации документации средствами javadoc.

# 5  Пример проекта РЕШЕНИЯ ЗАДАНИЯ №2

## 5.1  Разработка программы

Реализуем классы, структура которых соответствует схеме п.2.1.2.

Разработаем класс MainTest для проведения теста разработанных классов –ItemsGenerator, ItemsSorter, Items. Реализуем методы:

setUpBeforeClass() – выполняется первым;

testAdd() – тестирует операцию добавления объектов в коллекцию;

testAddDel() – тестирует операции добавления и удаления объектов;

testSort() – тестирует операцию сортировки объектов.

В процессе разработки необходимо обеспечить прохождение всех тестов.

### 5.1.1  Используемые средства ООП

Поведенческий шаблон Observer предоставляет компоненту возможность гибкой рассылки сообщений зарегистрированным получателям.

На некотором объекте сконцентрировано внимание наблюдателей, заинтересованных в получении от него какой-то информации. Потребовав от наблюдающих объектов, чтобы они устанавливали сеансы связи с центральным объектом, можно значительно снизить накладные расходы на коммуникацию, т.к. устанавливать связь будут только объекты, заинтересованные в получении обновленной информации.

Гибкость шаблона позволяет применять его для рассылки информации как отдельным, так и всем компонентам системы.

В соответствии с этим шаблоном, генераторы сообщений (наблюдаемые компоненты) рассылают сообщения, которые представляют события в системе. Эти события обрабатываются одним или несколькими получателями сообщений (компоненты-наблюдатели). Наблюдаемые компоненты отвечают за доставку событий всем заинтересованным наблюдателям (т.е. тем, которые установили сеансы связи). Интерфейс передачи сообщений позволяет наблюдаемым компонентам детализировать события для наблюдателей.

Если наблюдаемый объект многопоточный, он может поддерживать очередь сообщений, приоритеты сообщений, перекрытие сообщений и т.д.

Шаблон можно изменить, чтобы наблюдатели самостоятельно получали сообщения: наблюдаемый объект извещает о том, что событие произошло, а заинтересованные наблюдатели вызывают метод наблюдаемого для получения дополнительной информации о событии.

Аннотации – средство, которое позволяет встроить некоторую информацию (метаданные) в исходные и исполняемые файлы.

Аннотироваться могут классы, методы, поля, параметры, константы enum и сами аннотации.

Политика удержания аннотации определяет, на каком этапе аннотация отбрасывается. Определены три политики:

SOURCE – аннотации удерживаются только в исходном файле,

CLASS – аннотации сохраняются в файле .class во время компиляции, но недоступны JVM во время выполнения,

RUNTIME – аннотации сохраняются в файле .class и доступны во время выполнения.

Механизм рефлексии – позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы.

Рефлексия и наличие логически целостной модели выдачи информации об ошибках дает возможность создавать корректный динамический код.

RTTI позволяет получить информацию о точном типе объекта, когда имеется лишь ссылка базового типа. Использование этой информации подразумевает отказ от всех преимуществ полиморфизма. Рекомендуется использовать именно полиморфные методы, а к RTTI обращаться только в крайнем случае.

Различие между механизмом RTTI и рефлексией состоит в том, что при использовании RTTI файл .class открывается и анализируется компилятором, а при использовании рефлексии файл .class открывается и обрабатывается системой выполнения.

### 5.1.2  Иерархия и структура классов

Структура классов и схема их отношений приведена на рис.1.

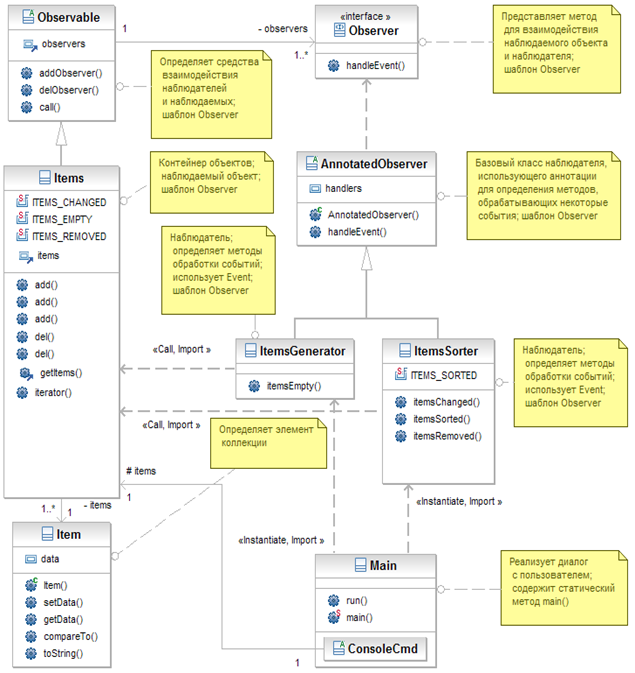


Рис.1. Схема классов и их отношений

### 5.1.3  Описание программы

Для реализации шаблона Observer использовались следующие классы и интерфейсы:

Observer – интерфейс, представляет метод для взаимодействия наблюдаемого объекта и наблюдателя; шаблон Observer.

Observable – абстрактный класс, определяет средства взаимодействия наблюдателей и наблюдаемых; шаблон Observer.

AnnotatedObserver – абстрактный класс, реализует Observer – базовый класс наблюдателя, использующего аннотации для определения методов, обрабатывающих некоторые события; шаблон Observer.

Event – аннотация времени выполнения для назначения методам наблюдателя конкретных событий.

Item – класс, определяет элемент коллекции.

Items – класс, расширяет Observable, контейнер объектов; наблюдаемый объект; шаблон Observer.

ItemsGenerator – класс, расширяет AnnotatedObserver, наблюдатель; определяет методы обработки событий; использует Event; шаблон Observer.

ItemsSorter – класс, расширяет AnnotatedObserver, наблюдатель; определяет методы обработки событий; использует Event; шаблон Observer.

Main – класс, реализует диалог с пользователем; содержит статический метод main().

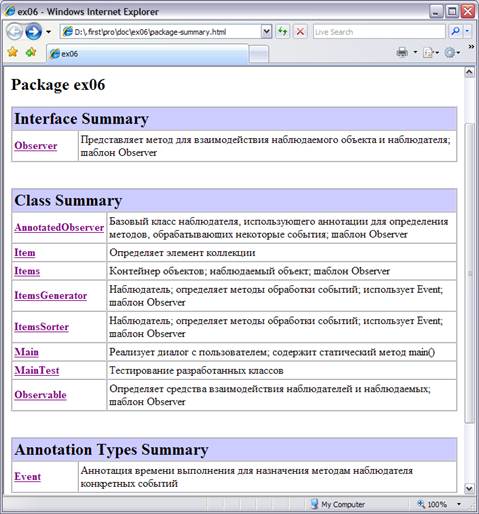
В данном примере показано, как наблюдаемый объект рассылает всем наблюдателям информацию об обновленном состоянии объекта Items. Наблюдатели при создании заполняют ассоциативный массив парами событие-обработчик. При этом для нахождения обработчиков используется механизм рефлексии – в конструкторе класса AnnotatedObserver просматриваются все методы и, отмеченные аннотацией Event, помещаются в массив. Параметр аннотации определяет идентификатор события. При получении сообщения вызывается обработчик, соответствующий идентификатору.

При написании исходного кода используем стиль комментариев документации javadoc.

Структура проекта:

|  |  |
| --- | --- |
| Папка src  Untitled-1 | Папка test  Untitled-2 |

Выполним генерацию документации:



После проверки работоспособности готовой программы, создадим исполняемый JAR файл ex06.jar

## 5.2  Текст программы

### 5.2.1  AnnotatedObserver.java

**package** ex06;

**import** java.lang.reflect.Method;

**import** java.util.HashMap;

**import** java.util.Map;

/\*\* Базовый класс наблюдателя,

 \* использующего аннотации

 \* для определения методов,

 \* обрабатывающих некоторые

 \* события; шаблон Observer

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Observer

 \* **@see** Observable

 \*/

**public** **abstract** **class** AnnotatedObserver **implements** Observer {

/\*\* Ассоциативный массив обработчиков событий; содержит пары событие-обработчик \*/

**private** Map<Object, Method> handlers = **new** HashMap<Object, Method>();

/\*\* Заполняет {@linkplain AnnotatedObserver#handlers} ссылками на методы,

 \* отмеченные аннотацией {@linkplain Event}

 \*/

**public** AnnotatedObserver() {

**for** (Method m : **this**.getClass().getMethods()) {

**if** (m.isAnnotationPresent(Event.**class**)) {

                     handlers.put(m.getAnnotation(Event.**class**).value(), m);

              }

       }

}

@Override

**public** **void** handleEvent(Observable observable, Object event) {

       Method m = handlers.get(event);

**try** {

**if** (m != **null**) m.invoke(**this**, observable);

       } **catch** (Exception e) {

              System.*err*.println(e);

       }

}

}

### 5.2.2  Event.java

**package** ex06;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

/\*\* Аннотация времени

 \* выполнения для

 \* назначения

 \* методам наблюдателя

 \* конкретных событий

 \* **@author** xone

 \* **@see** AnnotatedObserver

 \*/

@Retention(RetentionPolicy.*RUNTIME*)

**public** **@interface** Event {

String value();

}

### 5.2.3  Item.java

**package** ex06;

/\*\* Определяет элемент

 \* коллекции

 \* **@author** xone

 \* **@see** Items

 \*/

**public** **class** Item **implements** Comparable<Item> {

/\*\* Информационное поле \*/

**private** String data;

/\*\* Инициализирует {@linkplain Item#data}

 \* **@param** data значение для поля {@linkplain Item#data}

 \*/

**public** Item(String data) {

**this**.data = data;

}

/\*\* Устанавливает поле {@linkplain Item#data}

 \* **@param** data значение для поля {@linkplain Item#data}

 \* **@return** значение поля {@linkplain Item#data}

 \*/

**public** String setData(String data) {

**return** **this**.data = data;

}

/\*\* Возвращает поле {@linkplain Item#data}

 \* **@return** значение поля {@linkplain Item#data}

 \*/

**public** String getData() {

**return** data;

}

@Override

**public** **int** compareTo(Item o) {

**return** data.compareTo(o.data);

}

@Override

**public** String toString() {

**return** data;

}

}

### 5.2.4  Items.java

**package** ex06;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

/\*\* Контейнер объектов;

 \* наблюдаемый объект;

 \* шаблон Observer

 \* **@author** xone

 \* **@see** Observable

 \* **@see** Observer

 \* **@see** Item

 \*/

**public** **class** Items **extends** Observable **implements** Iterable<Item> {

/\*\* Константа-идентификатор события, обрабатываемого наблюдателями \*/

**public** **static** **final** String *ITEMS\_CHANGED* = "ITEMS\_CHANGED";

/\*\* Константа-идентификатор события, обрабатываемого наблюдателями \*/

**public** **static** **final** String *ITEMS\_EMPTY* = "ITEMS\_EMPTY";

/\*\* Константа-идентификатор события, обрабатываемого наблюдателями \*/

**public** **static** **final** String *ITEMS\_REMOVED* = "ITEMS\_REMOVED";

/\*\* Коллекция объектов класса {@linkplain Item} \*/

**private** List<Item> items = **new** ArrayList<Item>();

/\*\* Добавляет объект в коллекцию и извещает наблюдателей

 \* **@param** item объект класса {@linkplain Item}

 \*/

**public** **void** add(Item item) {

       items.add(item);

**if** (item.getData().isEmpty()) call(*ITEMS\_EMPTY*);

**else** call(*ITEMS\_CHANGED*);

}

/\*\* Добавляет объект в коллекцию

 \* **@param** s передается конструктору {@linkplain Item#Item(String)}

 \*/

**public** **void** add(String s) {

       add(**new** Item(s));

}

/\*\* Добавляет несколько объектов в коллекцию и извещает наблюдателей

 \* **@param** n количество добавляемых объектов класса {@linkplain Item}

 \*/

**public** **void** add(**int** n) {

**if** (n > 0) {

**while** (n-- > 0) items.add(**new** Item(""));

              call(*ITEMS\_EMPTY*);

       }

}

/\*\* Удаляет объект из коллекции и извещает наблюдателей

 \* **@param** item удаляемый объект

 \*/

**public** **void** del(Item item) {

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

              items.remove(item);

              call(*ITEMS\_REMOVED*);

       }

}

/\*\* Удаляет объект из коллекции и извещает наблюдателей

 \* **@param** index индекс удаляемого объекта

 \*/

**public** **void** del(**int** index) {

**if** ((index >= 0) && (index < items.size())) {

              items.remove(index);

              call(*ITEMS\_REMOVED*);

       }

}

/\*\* Возвращает ссылку на коллекцию

 \* **@return** ссылка на коллекцию объектов класса {@linkplain Item}

 \*/

**public** List<Item> getItems() {

**return** items;

}

@Override

**public** Iterator<Item> iterator() {

**return** items.iterator();

}

}

### 5.2.5  ItemsGenerator.java

**package** ex06;

/\*\* Наблюдатель;

 \* определяет методы

 \* обработки событий;

 \* использует Event;

 \* шаблон Observer

 \* **@author** xone

 \* **@see** AnnotatedObserver

 \* **@see** Event

 \*/

**public** **class** ItemsGenerator **extends** AnnotatedObserver {

/\*\* Обработчик события {@linkplain Items#ITEMS\_EMPTY};

 \* извещает наблюдателей; шаблон Observer

 \* **@param** observable наблюдаемый объект класса {@linkplain Items}

 \* **@see** Observable

 \*/

@Event(Items.*ITEMS\_EMPTY*)

**public** **void** itemsEmpty(Items observable) {

**for** (Item item : observable) {

**if** (item.getData().isEmpty()) {

**int** len = (**int**)(Math.*random*() \* 10) + 1;

                     String data = "";

**for** (**int** n = 1; n <= len; n++) {

                            data += (**char**)((**int**)(Math.*random*() \* 26) + 'A');

                     }

                     item.setData(data);

              }

       }

       observable.call(Items.*ITEMS\_CHANGED*);

}

}

### 5.2.6  ItemsSorter.java

**package** ex06;

**import** java.util.Collections;

/\*\* Наблюдатель;

 \* определяет методы

 \* обработки событий;

 \* использует Event;

 \* шаблон Observer

 \* **@author** xone

 \* **@see** AnnotatedObserver

 \* **@see** Event

 \*/

**public** **class** ItemsSorter **extends** AnnotatedObserver {

/\*\* Константа-идентификатор события, обрабатываемого наблюдателями \*/

**public** **static** **final** String *ITEMS\_SORTED* = "ITEMS\_SORTED";

/\*\* Обработчик события {@linkplain Items#ITEMS\_CHANGED};

 \* извещает наблюдателей; шаблон Observer

 \* **@param** observable наблюдаемый объект класса {@linkplain Items}

 \* **@see** Observable

 \*/

@Event(Items.*ITEMS\_CHANGED*)

**public** **void** itemsChanged(Items observable) {

       Collections.*sort*(observable.getItems());

       observable.call(*ITEMS\_SORTED*);

}

/\*\* Обработчик события {@linkplain Items#ITEMS\_SORTED}; шаблон Observer

 \* **@param** observable наблюдаемый объект класса {@linkplain Items}

 \* **@see** Observable

 \*/

@Event(*ITEMS\_SORTED*)

**public** **void** itemsSorted(Items observable) {

       System.*out*.println(observable.getItems());

}

/\*\* Обработчик события {@linkplain Items#ITEMS\_REMOVED}; шаблон Observer

 \* **@param** observable наблюдаемый объект класса {@linkplain Items}

 \* **@see** Observable

 \*/

@Event(Items.*ITEMS\_REMOVED*)

**public** **void** itemsRemoved(Items observable) {

       System.*out*.println(observable.getItems());

}

}

### 5.2.7  Main.java

**package** ex06;

**import** ex04.ConsoleCommand;

**import** ex04.Menu;

/\*\* Реализует диалог

 \* с пользователем;

 \* содержит статический

 \* метод main()

 \* **@author** xone

 \* **@version** 6.0

 \* **@see** Main#main

 \*/

**public** **class** Main {

/\*\* Консольная команда;

 \* используется при

 \* создании анонимных

 \* экземпляров команд

 \* пользовательского

 \* интерфейса;

 \* шаблон Command

 \* **@author** xone

 \* **@see** ConsoleCommand

 \*/

**abstract** **class** ConsoleCmd **implements** ConsoleCommand {

       /\*\* Коллекция объектов {@linkplain Items} \*/

**protected** Items items;

       /\*\* Отображаемое название команды \*/

**private** String name;

       /\*\* Символ горячей клавиши команды \*/

**private** **char** key;

       /\*\* Инициализирует поля консольной команды

        \* **@param** items {@linkplain ConsoleCmd#items}

        \* **@param** name {@linkplain ConsoleCmd#name}

        \* **@param** key {@linkplain ConsoleCmd#key}

        \*/

       ConsoleCmd(Items items, String name, **char** key) {

**this**.items = items;

**this**.name = name;

**this**.key = key;

       }

       @Override

**public** **char** getKey() {

**return** key;

       }

       @Override

**public** String toString() {

**return** name;

       }

}

/\*\* Устанавливает связь наблюдателей с наблюдаемыми объектами;

 \* реализует диалог с пользователем

 \*/

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

       Items items = **new** Items();

       ItemsGenerator generator = **new** ItemsGenerator();

       ItemsSorter sorter = **new** ItemsSorter();

       items.addObserver(generator);

       items.addObserver(sorter);

       Menu menu = **new** Menu();

       menu.add(**new** ConsoleCmd(items, "'v'iew", 'v') {

              @Override

**public** **void** execute() {

                     System.*out*.println(items.getItems());

              }

       });

       menu.add(**new** ConsoleCmd(items, "'a'dd", 'a') {

              @Override

**public** **void** execute() {

                     items.add("");

              }

       });

       menu.add(**new** ConsoleCmd(items, "'d'el", 'd') {

              @Override

**public** **void** execute() {

                     items.del((**int**)Math.*round*(Math.*random*()\*(items.getItems().size()-1)));

              }

       });

       menu.execute();

}

/\*\* Выполняется при запуске программы

 \* **@param** args параметры запуска программы

 \*/

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

**new** Main().run();

}

}

### 5.2.8  Observable.java

**package** ex06;

**import** java.util.HashSet;

**import** java.util.Set;

/\*\* Определяет средства

 \* взаимодействия

 \* наблюдателей

 \* и наблюдаемых;

 \* шаблон Observer

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Observer

 \*/

**public** **abstract** **class** Observable {

/\*\* Множество наблюдателей; шаблон Observer

 \* **@see** Observer

 \*/

**private** Set<Observer> observers = **new** HashSet<Observer>();

/\*\* Добавляет наблюдателя; шаблон Observer

 \* **@param** observer объект-наблюдатель

 \*/

**public** **void** addObserver(Observer observer) {

       observers.add(observer);

}

/\*\* Удаляет наблюдателя; шаблон Observer

 \* **@param** observer объект-наблюдатель

 \*/

**public** **void** delObserver(Observer observer) {

       observers.remove(observer);

}

/\*\* Оповещает наблюдателей о событии; шаблон Observer

 \* **@param** event информация о событии

 \*/

**public** **void** call(Object event) {

**for** (Observer observer : observers) {

              observer.handleEvent(**this**, event);

       }

}

}

### 5.2.9  Observer .java

**package** ex06;

/\*\* Представляет метод

 \* для взаимодействия

 \* наблюдаемого объекта

 \* и наблюдателя;

 \* шаблон Observer

 \* **@author** xone

 \* **@version** 1.0

 \* **@see** Observable

 \*/

**public** **interface** Observer {

/\*\* Вызывается наблюдаемым объектом для каждого наблюдателя; шаблон Observer

 \* **@param** observable ссылка на наблюдаемый объект

 \* **@param** event информация о событии

 \*/

**public** **void** handleEvent(Observable observable, Object event);

}

### 5.2.10  MainTest.java

**package** ex06;

**import** **static** org.junit.Assert.\*;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**import** java.util.Random;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

/\*\* Тестирование

 \* разработанных классов

 \* **@author** xone

 \* **@version** 6.0

 \* **@see** ItemsGenerator

 \* **@see** ItemsSorter

 \* **@see** Items

 \*/

**public** **class** MainTest {

/\*\* Количество элементов в коллекции \*/

**private** **static** **final** **int** *ITEMS\_SIZE* = 1000;

/\*\* Наблюдатель; шаблон Observer \*/

**private** **static** ItemsGenerator *generator* = **new** ItemsGenerator();

/\*\* Наблюдатель; шаблон Observer \*/

**private** **static** ItemsSorter *sorter* = **new** ItemsSorter();

/\*\* Наблюдаемый объект; шаблон Observer \*/

**private** **static** Items *observable* = **new** Items();

/\*\* Выполняется первым \*/

@BeforeClass

**public** **static** **void** setUpBeforeClass() {

*observable*.addObserver(*generator*);

*observable*.addObserver(*sorter*);

}

/\*\* Тестирует операцию добавления объектов в коллекцию \*/

@Test

**public** **void**   testAdd() {

*observable*.getItems().clear();

*observable*.add(**new** Item("AAA"));

*observable*.add("AAA");

*observable*.add("");

*observable*.add(*ITEMS\_SIZE*);

**for** (Item item : *observable*) {

*assertFalse*(item.getData().isEmpty());

       }

*assertEquals*(*ITEMS\_SIZE* + 3, *observable*.getItems().size());

}

/\*\* Тестирует операции добавления и удаления объектов \*/

@Test

**public** **void**   testAddDel() {

       Item tmp;

*observable*.getItems().clear();

*observable*.add("");

*observable*.add(*ITEMS\_SIZE*);

**for** (**int** i = *ITEMS\_SIZE*; i > 0; i--) {

              tmp = *observable*.getItems().get((**new** Random()).nextInt(i));

*observable*.del(tmp);

       }

*assertEquals*(1, *observable*.getItems().size());

}

/\*\* Тестирует операцию сортировки объектов \*/

@Test

**public** **void**   testSort() {

*observable*.getItems().clear();

*observable*.add(*ITEMS\_SIZE*);

       List<Item> items = **new** ArrayList<Item>(*observable*.getItems());

       Collections.*sort*(items);

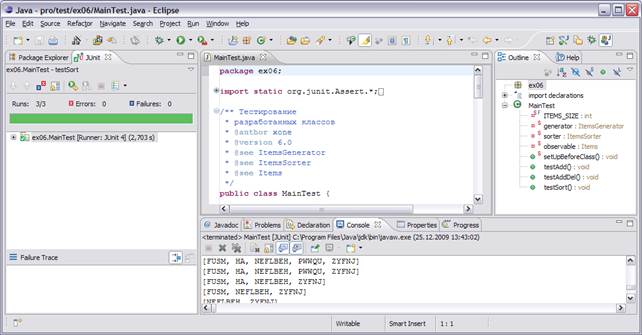
*assertEquals*(items, *observable*.getItems());

}

}

## 5.3  Результаты тестирования

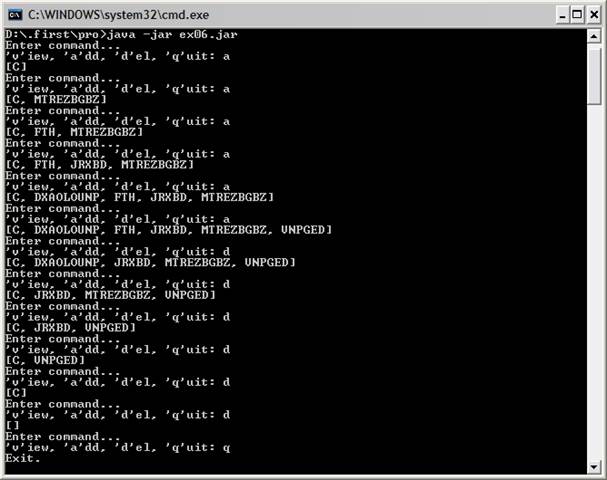
Выполним ex05.MainTest как JUnit Test



Выполним запуск программы из командной строки:

java -jar ex06.jar

В результате выполнения получим:



# 6  Заключение по результату решения заданиЯ №2

Разработали программу решения задачи индивидуального задания. Результаты тестирования подтверждают корректность используемых методов.

Для обслуживания коллекции объектов был реализован шаблон Observer.

При создании наблюдателей использовались средства аннотирования и рефлексии.

Для тестирования программы использовались средства JUnit.

Документация подготовлена на основе комментариев инструментом автоматической генерации javadoc.