**Министерство образования и науки Российской Федерации**

ФГАОУ ВПО «Уральский федеральный университет   
имени первого Президента России Б.Н. Ельцина»

Центр ускоренного обучения

Оценка проекта

Члены комиссии

**РАЗРАБОТКА ПРОГРАММЫ ДЛЯ РЕШЕНИЯ**

**«Задачи о рюкзаке»**

КУРСОВАЯ РАБОТА

по дисциплине «Информатика и программирование»

Пояснительная записка

09.03.04 58.29.29 005 ПЗ

Руководитель доц., к.т.н. С.И. Тимошенко

Студент гр. РИВ-150026у (656) Д.А. Григорьев

Екатеринбург 2016

ФГАОУ ВПО «Уральский федеральный университет

имени первого Президента России Б.Н.Ельцина»

Центр ускоренного обучения

УТВЕРЖДАЮ:

Зав. кафедрой ПСС

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

С.И. Тимошенко

“20” февраля 2016г.

**ЗАДАНИЕ**

на курсовую работу по дисциплине

«Информатика и программирование»

Студент группы РИВ-150026у (656)

Специальность 09.03.04

Фамилия, имя, отчество Григорьев Дмитрий Александрович

Руководитель курсовой работы Тимошенко Сергей Иванович

Срок проектирования с 20.02.16 по 11.06.16

**1 Тема курсовой работы:** *Разработка программы для решения задачи «Задача о рюкзаке».*

**2 Содержание курсовой работы:** Необходимо разработать средствами Java программу со следующими возможностями:

а) ввод/вывод параметров должен осуществляться двумя вариантами (консоль и файл);

б) программа должна содержать тесты с использованием JUnit;

в) при описании программы должны использоваться схема работы системы (ГОСТ 19.701-90) и диаграмма классов по правилам UML;

г) оформление пояснительной записки к курсовой работе должно соответствовать ГОСТ 2.105-90, 7.32-2001 и 7.1-2003;

д) подробное описание задачи должно совпадать с условиями, приведенными на страницах 92-93, 102-105 книги Окулова С.М. Программирование в алгоритмах / С.М.Окулов. – М.: Бином, 2002. – 341 с.

**3 Курсовое проектирование закончено** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4 Оценка проекта** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Члены комиссии\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 3](#_Toc454009902)

[ВВЕДЕНИЕ 5](#_Toc454009903)

[1 Постановка задачи 7](#_Toc454009904)

[2 Анализ поставленной задачи 8](#_Toc454009905)

[2.1 Основные объекты задачи и их взаимодействие 8](#_Toc454009906)

[2.2 Алгоритм 8](#_Toc454009907)

[2.2.1 Полный перебор 9](#_Toc454009908)

[2.2.2 Метод ветвей и границ 10](#_Toc454009909)

[3 Описание результатов разработки 11](#_Toc454009910)

[3.1 Общая архитектура программы 11](#_Toc454009911)

[3.2 Ввод информации 13](#_Toc454009912)

[3.2.1 Класс ConsoleInput() 13](#_Toc454009913)

[3.2.2 Класс FileInput() 14](#_Toc454009914)

[3.3 Вывод информации 16](#_Toc454009915)

[3.3.1 Класс ConsoleOutput 16](#_Toc454009916)

[3.3.2 Класс FileOutput 16](#_Toc454009917)

[3.4 Классы алгоритмов решения задачи 17](#_Toc454009918)

[3.5 Применение паттерна «Фабрика» 18](#_Toc454009919)

[3.6 Вспомогательные классы 20](#_Toc454009920)

[3.7 Тестирование программы 21](#_Toc454009921)

[4 Руководство пользователя 23](#_Toc454009922)

[4.1 Минимальные системные требования 23](#_Toc454009923)

[4.2 Установка и удаление программы 23](#_Toc454009924)

[4.3 Работа с программой 23](#_Toc454009925)

[ЗАКЛЮЧЕНИЕ 26](#_Toc454009926)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 27](#_Toc454009927)

[ПРИЛОЖЕНИЕ А (обязательное) Исходный код программы 28](#_Toc454009928)

[А.1 Основной класс программы 28](#_Toc454009929)

[А.2 Пакет IOData 28](#_Toc454009930)

[А.2.1 Интерфейс InputData 28](#_Toc454009931)

[А.2.2 Реализация интерфейса InputData. Класс ConsoleInput 29](#_Toc454009932)

[А.2.3 Реализация интерфейса InputData. Класс FileInput 31](#_Toc454009933)

[А.2.4 Фабрика InputFactory 33](#_Toc454009934)

[А.2.5 Интерфейс OutputData 33](#_Toc454009935)

[А.2.6 Реализация интерфейса OutputData. Класс ConsoleOutput 33](#_Toc454009936)

[А.2.7 Реализация интерфейса OutputData. Класс FileOutput 34](#_Toc454009937)

[А.2.8 Фабрика OutputFactory 35](#_Toc454009938)

[А.3 Пакет Solve 35](#_Toc454009939)

[А.3.1 Интерфейс Solve 35](#_Toc454009940)

[А.3.2 Реализация интерфейса Solve. Класс ExhaustiveSearch 36](#_Toc454009941)

[А.3.3 Реализация интерфейса Solve. Класс SimpleExhaustiveSearch 37](#_Toc454009942)

[А.3.4 Фабрика SolveFactory 38](#_Toc454009943)

[А.4 Пакет utils. Вспомогательные классы 39](#_Toc454009944)

[А.4.1 Класс CmdParams 39](#_Toc454009945)

[А.4.2 Класс Validator 40](#_Toc454009946)

[А.5 Класс тестирования TestSolves 41](#_Toc454009947)

# ВВЕДЕНИЕ

Целью курсовой работы является разработка программы, позволяющей решить одну из NP-задач комбинаторной оптимизации, известной как «Задача о рюкзаке» (англ. Knapsack problem). Своё название задача получила от максимизационной задачи укладки как можно большего числа ценных вещей в рюкзак при условии, что вместимость рюкзака ограничена. С различными вариациями задачи о ранце можно столкнуться в экономике, прикладной математике, криптографии, генетике и логистике.

При разработке программы, будем руководствоваться описанием задачи в книге Окулова С.М. «Программирование в алгоритмах» [c. 92-93, 102-105]. Для разработки игры будем использовать язык программирования Java.

В общем виде задача ставится следующим образом: в рюкзак загружаются предметы N различных типов (количество предметов каждого типа не ограничено). Максимальный вес рюкзака W. Каждый предмет типа i имеет вес wi и стоимость vi (i = 1,2, …, N). Требуется определить максимальную стоимость груза, вес которого равен W.

Подготовка курсовой работы способствует:

1. закреплению, углублению и обобщению знаний, полученных студентами в процессе изучения лекционных курсов по дисциплине «Информатика и программирование»;
2. развитию умений и навыков, полученных при выполнении лабораторных работ;
3. применению этих знаний, умений и навыков к решению конкретных задач;
4. развитию навыков работы со специальной литературой и разработки программного обеспечения.

Пояснительная записка имеет следующую структуру:

1. во введении описывается краткая постановка задачи и содержание разделов;
2. основная часть состоит из нескольких разделов:
   1. постановка задачи – описание требований к написанию курсовой работы;
   2. анализ поставленной задачи – взгляд на задачу «изнутри», рассмотрение методов решения, предложения по программной реализации задачи (структур данных, алгоритмов);
   3. описание результатов разработки – рассказывается о порядке написания программы, в том числе тестирующих модулей. По мере разработки, программа видоизменяется, избавляясь от первоначальных недостатков. В конце раздела приводится листинг окончательного варианта программы;
3. в заключении проводится анализ проделанной работы;
4. список использованных источников – содержит перечень использованной при разработке литературы;
5. в приложениях содержатся результаты тестирования различных алгоритмов перебора, а так же руководство пользователя.

# Постановка задачи

Рассмотрим поставленную задачу более подробно. В рюкзак загружаются предметы *N* различных типов (количество предметов каждого типа не ограничено). Максимальный вес рюкзака равен *W*. Каждый предмет типа i имеет вес *wi* и стоимость *vi* (*i* = 1,2, …, *N*). Требуется определить максимальную стоимость груза, вес которого равен *W*. Обозначим количество предметов типа *i* через *ki*, тогда требуется максимизировать

*v1 \* k1 + v2 \* k2 + … + vN \* kN*

при ограничениях

*w1 \* k1 + w2 \* k2 + … + wN \* kN = W,*

где *ki* – целые (0 ≤ *ki* ≤ [*W / wi*]), квадратные скобки означают целую часть числа.

Формализуем задачу следующим образом. Шаг *i* ставится в соответствие типу предмета *i*=1,2, …, *N*. Состояние *yi*на шаге *i* выражает суммарный вес предметов, решение о загрузке которых принято на шагах 0,1, …, *i*. При этом, *yn = W*, *yi* = 0,1, …, *W* при *i*=1,2,…,*N*-1. Варианты решения *ki*на шаге *i* описываются количеством предметов типа *i*, 0 ≤ *ki* ≤ [*W / wi*].

Необходимо разработать программу, которая находит наилучший вариант загрузки рюкзака. Ввод/вывод параметров программы осуществлять двумя вариантами – консоль и файл. Программа должна быть написана на языке программирования Java, в среде разработки Eclipse. При разработке обязательно активное использование методов TDD и тестовых средств JUnit. Описание результатов разработки оформить в виде пояснительной записки, в соответствии с ГОСТ 2.105-90, 7.32-2001 и 7.1-2003. При описании программы использовать схемы работы системы (ГОСТ 19.701-90) и диаграммы классов по правилам UML. Подробное описание сути задачи должно соответствовать страницам 92-93, 102-105 книги «Программирование в алгоритмах» / С.М.Окулов. – М.: Бином, 2002. – 341 с.

# Анализ поставленной задачи

## Основные объекты задачи и их взаимодействие

Рассмотрим поставленную задачу. Программе на вход (через интерфейс ввода) подается массив набора значений, описывающий набор предметов различных типов, Каждый предмет имеет характеристики вес и стоимость. Также на вход программе передается максимальный вес рюкзака. После завершения работы программы, на выходе (через интерфейс вывода) требуется получить число, определяющее максимальную стоимость груза, при данных типах предметов, а также массив чисел, определяющий наилучший вариант загрузки рюкзака (сколько и каких типов предметов требуется загрузить в рюкзак для достижения максимальной стоимости груза).

«Задача о рюкзаке» относится к классу NP-полных, для неё нет полиномиального алгоритма, решающего её за разумное время. Поэтому при решении задачи о ранце всегда нужно выбирать между точными алгоритмами, которые не применимы для «больших» рюкзаков, и приближенными, которые работают быстро, но не обеспечивают оптимального решения задачи. Примером точных алгоритмов могут служить «полный перебор», «метод ветвей и границ», «динамического программирования», приближенными алгоритмами является «жадный» и «генетический».

Для решения поставленной задачи Окулов С.М. в книге «Программирование в алгоритмах» предлагает для решения данной задачи использовать два различных подхода – воспользоваться алгоритмом полного перебора и методом динамического программирования.

## Алгоритм

Решение данной задачи предполагает использование различных алгоритмов. В частности, возможно использовать «полный перебор», «метод ветвей и границ», «жадный алгоритм», метаалгоритмы, в частности «генетический алгоритм».

В данном курсовом проекте рассмотрим решение данной задачи через полный перебор, и методом ветвей и границ.

### Полный перебор

Пусть в рюкзак загружаются предметы *N* разных типов. Рассмотрим задачу, когда количество предметов каждого типа не ограничено. Нужно определить максимальную стоимость груза, вес которого равен *P*. Для получения решения алгоритмом полного перебора осуществляется перебор всех вариантов загрузки рюкзака.

Временная сложность алгоритма *O*(*N*!), т.е он работоспособен для небольших значений *N*. С ростом *N* задача становится неразрешимой данным методом за приемлемое время.

На рисунке 2.1 показано четырёхуровневое дерево перебора. Корень дерева соответствует нулевому весу (рюкзак пуст), в кружках показан вес предмета. Первый предмет возможно выбрать четырьмя способами, второй тремя и т. д.

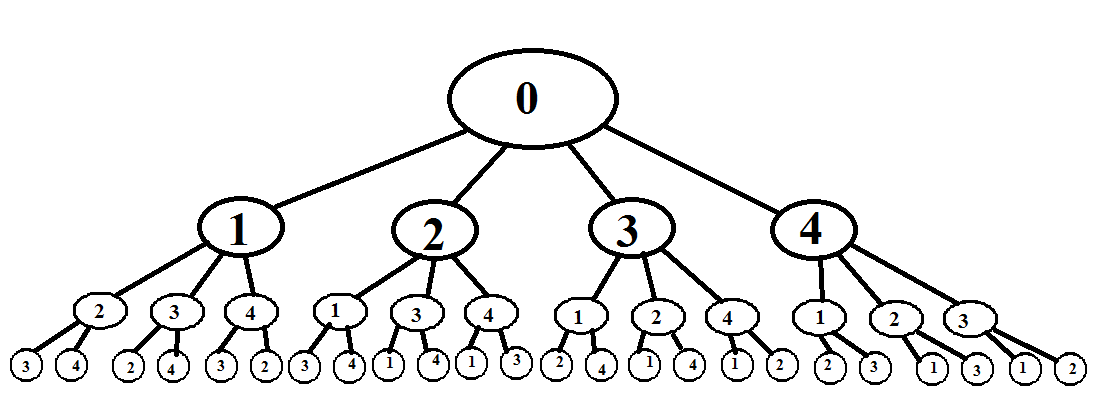


Рисунок 2.1 – Дерево полного перебора

Полный перебор позволяет получить гарантированно верный результат, однако, ввиду своей сложности, для решения данной задачи при больших значениях *N*, требуется воспользоваться оптимизированным алгоритмом. Остановимся на «методе ветвей и границ», представленным на страницах 102-105 книги «Программирование в алгоритмах» / С.М.Окулов. – М.: Бином, 2002. – 341 с.

### Метод ветвей и границ

Метод ветвей и границ является вариацией метода полного перебора с той разницей, что мы сразу исключаем заведомо неоптимальные решения. Как и метод полного перебора, он позволяет найти оптимальное решение и поэтому относится к точным алгоритмам.

Пусть есть оптимальное решение *R*. Попытаемся его улучшить, рассмотрев решение на другой ветви. Если на рассматриваемой в данной момент ветви решение становится хуже (с какого-то шага), чем *R*, то прекращаем его исследование и выбираем другую ветвь дерева.

На рисунке 2.2 показано четырёхуровневое дерево перебора, упрощенное методом ветвей и границ.

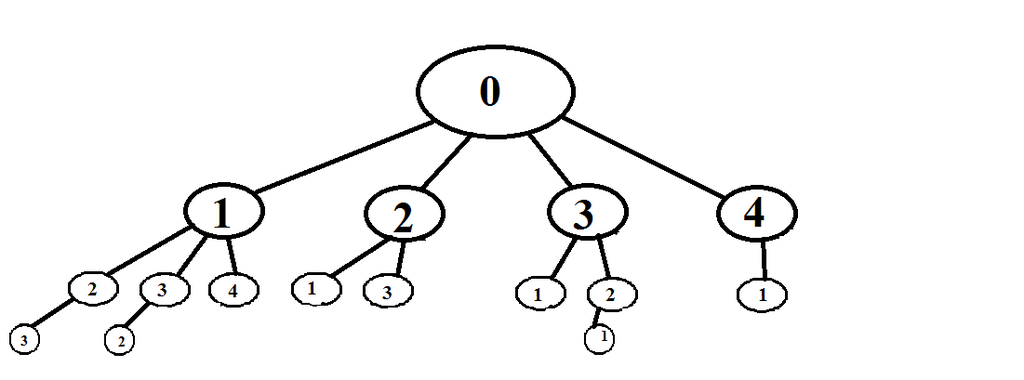


Рисунок 2.2 – Дерево, упрощенное методом ветвей и границ

При использовании метода ветвей и границ строится сеть. По оси *X* откладывается количество предметов, по оси *Y* — их вес. На первом шаге из начала координат строятся две линии: горизонтальная, соответствующая тому, что первый предмет не был взят, и наклонная, соответствующая взятому первому предмету. Их проекции на ось *Y* равны весу предмета. На втором шаге опять строятся 2 линии, горизонтальная (второй предмет не был взят) или наклонная (второй предмет взят). Положим длину горизонтальных дуг равной нулю, а наклонных — ценности предмета.

Таким образом, любому решению задачи соответствует некоторый путь в сети. Задача сводится к нахождению пути максимальной длины.

# Описание результатов разработки

## Общая архитектура программы

В соответствии с требованиями задания, программа должна осуществлять ввод и вывод данных как через консоль, так и через файл. Соответственно, требуется написать классы ввода и вывода информации для, соответственно, консоли и файла. Помимо ввода и вывода информации требуется создать отдельный класс для решения непосредственно задачи. Следовательно, условно программа должна состоять из трех блоков: ввод информации (исходного набора данных), решение задачи, вывод информации (результат решения задачи).

Рисунок 3.1 – Общая структура программы

Поскольку ввод и вывод требуется осуществлять через консоль и файл, создадим интерфейсы ввода и вывода информации: InputData и OutputData, соответственно, для ввода и вывода информации.

Интерфейс InputData содержит методы:

* public int getWMax() – максимальный объем рюкзака;
* public int[] getWeights() – массив набора весов;
* public int[] getPrices() – массив набора стоимостей.

Интерфейс OutputData содержит метод public void OutputResult(Solve knapsackSolve), в котрый передается решение (Solve).

Поскольку решение задачи возможно несколькими способами, для алгоритма решения также создадим интерфейс Solve. Данный интерфейс содержит следующие методы:

1. public void KnapsackResolve(int wMax, int[] weights, int[] prices) – метод (алгоритм) решения задачи, который принимает параметры:
   1. wMax – Макс вместимость рюкзака;
   2. weights – Массив весов;
   3. prices – Массив стоимостей;
2. public int getMaxPrice() – метод, через который мы запрашиваем (получаем) максимизированную стоимость решения.
3. public int[] getResolveArray() – метод, возвращающий массив выбранных предметов.

Также, для программы потребуется класс, содержащий точку входа в программу. Данный класс будет называться Knapsack, в соответствии с англоязычным названием данной задачи.

UML диаграмма для интерфейсов показана на рисунке 3.2.

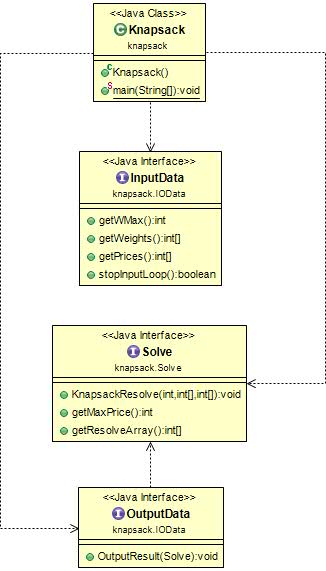


Рисунок 3.2 – UML диаграмма интерфейсов

Для определения режима работы программы («диалоговый» режим, когда ввод и вывод информации осуществляется через консоль, и «пакетный» режим, когда ввод и вывод информации осуществляется через файл), создадим вспомогательный класс CmdParams, с помощью которого будем обрабатывать параметры командной строки при запуске программы.

При диалоговом режиме работы программы за один запуск программы пользователь решает данную задачу однократно. При необходимости решения задачи при другом наборе исходных данных пользователь запустит программу повторно. Однако, при решении задачи в пакетном режиме, когда чтение информации происходит из файла, файл может содержать несколько наборов исходных данных. Соответственно, для многократного решения задачи на разных исходных данных, требуется зациклить работу программы. Для возможности остановки цикла, дополнительно введем в интерфейс InputData метод public boolean stopInputLoop(), который будет останавливать цикл работы программы при прочтении всех наборов исходных данных.

## Ввод информации

Для удобства, классы относящиеся к вводу-выводу информации, вынесены в отдельный пакет – knapsack.IOData.

### Класс ConsoleInput()

Для ввода информации с консоли, написан класс ConsoleInput, реализующий интерфейс InputData.

Поскольку при вводе информации с консоли (диалоговый режим) нет необходимости многократного повторения ввода исходных данных, метод stopInputLoop() будет всегда возвращать true.

В методе getWMax() происходит получение максимальной вместимости рюкзака через private метод inputConsoleInt(String msg).

**int** wmax = 0;

**while** (wmax == 0) {

wmax = inputConsoleInt("Введите максимальную вместимость рюкзака");

}

**return** wmax;

Данная конструкция позволяет остановить обработку данного метода только в том случае, если метод inputConsoleInt(String msg) вернул корректное значение. В private методе inputConsoleInt(String msg) происходит печать сообщения msg в консоль и чтение введенной информации с консоли. При этом данный метод может получить с консоли только число (int):

consoleInput = **new** Scanner(System.***in***);

res = consoleInput.nextInt();

в противном случае, метод inputConsoleInt(String msg) не вернет ничего, однако, метод getWMax() будет выполняться до тех пор, пока число типа int не будет получено.

В соответствии с исходными данными требуется также получить два массива: набор стоимостей и набор весов. Для этого, соответственно, служат методы public int[] getPrices() и public int[] getWeights(). Поскольку тип данных одинаков для обеих методов (массив int), целесообразно создать отдельный private метод private int[] inputConsoleIntArr(String msg), который будет получать строку msg с сообщением для вывода в консоль приглашения ввести исходные данные, получать исходные данные. При этом допустим ввод только массива типа int, в противном случае, программа запросит повторный ввод информации, аналогично методу getWMax().

### Класс FileInput()

Для ввода (чтения) информации из файла написан класс FileInput, реализующий интерфейс InputData, аналогично классу ConsoleInput.

Важным отличием данного класса от ConsoleInput, является вычисление значения метода stopInputLoop().

При чтении информации из файла, создается список всех считанных из файла строк. Типа ArrayList<String>(). При этом, игнорируются пустые строки и строки, первым символом которой идет «#» (комметарий). Дополнительно, начальные и конечные пробелы строки обрезаются методом trim().

**while** ((line = reader.readLine()) != **null**) {

line = line.trim();

**if** (!line.isEmpty() && line.charAt(0)!='#') fileData.add(line);

}

*linesOfData* = fileData.size();

В переменной linesOfData содержится количество строк с исходными данными.

Процесс получения исходных наборов данных методами getWMax(), getWeights() и getPrice() похож на реализацию аналогичных методов класса ConsoleInput. Однако, при каждом вызове этих методов, увеличивается число прочитанных строк. Для этого написан дополнительный метод startLn(), увеличивающий количество прочитанных строк, а также дополнительно проверяющий, чтобы количество строк исходных данных было не меньше количества прочитанных строк. Если достигнут лимит прочитанных строк, то поле класса Boolean stopInputLoop меняется на true, и метод stopInputLoop() также будет возвращать true, что приведет к остановке в вызывающем классе цикла чтения исходных данных из файла.

UML диаграмма классов ввода информации показана на рисунке 3.3.

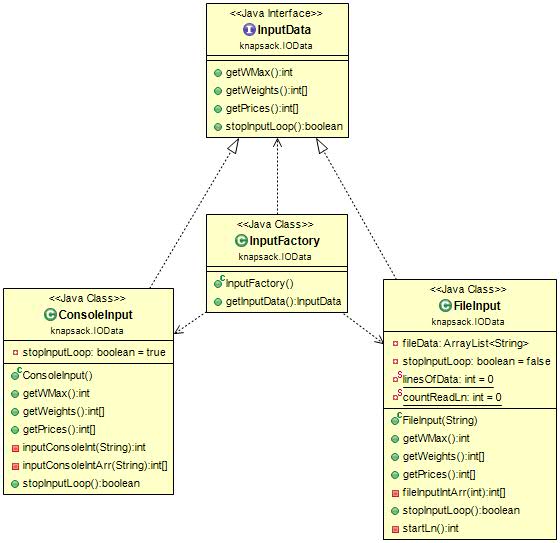


Рисунок 3.3 – UML диаграмма классов ввода информации

## Вывод информации

Для вывода информации созданы классы ConsoleOutput иFileOutput для вывода информации, соответственно, в консоль и файл. Данные классы реализуют интерфейс OutputData, который принимает в качестве параметра объект класса интерфейса Solve, содержащий методы getResolveArray(), содержащий массив искомых предметов и getMaxPrice(), содержащий максимизированную стоимость предметов.

### Класс ConsoleOutput

Получает объект класса, реализующего интерфейс Solve, после чего получает через методы getResolveArray() и getMaxPrice() данного объекта массив выбранных предметов и максимизированную стоимость. Вывод полученных значений осуществляется в консоль вывода стандартными средствами.

### Класс FileOutput

Класс FileOutput, получает данные через методы getResolveArray() и getMaxPrice() объекта класса, реализующего интерфейс класса Solve аналогично классу ConsoleOutput. Дополнительно, данный класс получает через конструктор строку, содержащую путь к файлу вывода информации. И записывает результат решения задачи в файл.

UML диаграмма классов вывода информации показана на рисунке 3.4.

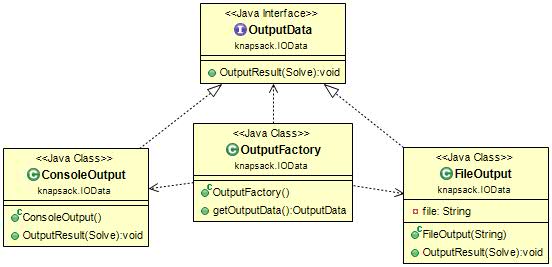


Рисунок 3.4 – UML диаграмма классов вывода информации

## Классы алгоритмов решения задачи

Поскольку алгоритмов решения данной задачи может быть несколько, написан интерфейс Solve, и два класса реализации данного интерфейса: ExhaustiveSearch (полный перебор) и SimpleExhaustiveSearch (метод ветвей и границ).

Интерфейс содержит методы:

* 1. public void KnapsackResolve(int wMax, int[] weights, int[] prices), через параметры в который передаются исходные данные, а именно: wMax Макс вместимость рюкзака, weights Массив весов, prices Массив стоимостей;
  2. public int getMaxPrice(), получающий максимизированную стоимость найденного решения;
  3. public int[] getResolveArray(), получающий массив выбранных предметов найденного решения.

Класс ExhaustiveSearch реализован в соответствии с примером, представленным на страницах 102-105 книги «Программирование в алгоритмах» / С.М.Окулов. – М.: Бином, 2002. – 341 с.

UML диаграмма классов поиска решения показана на рисунке 3.5.

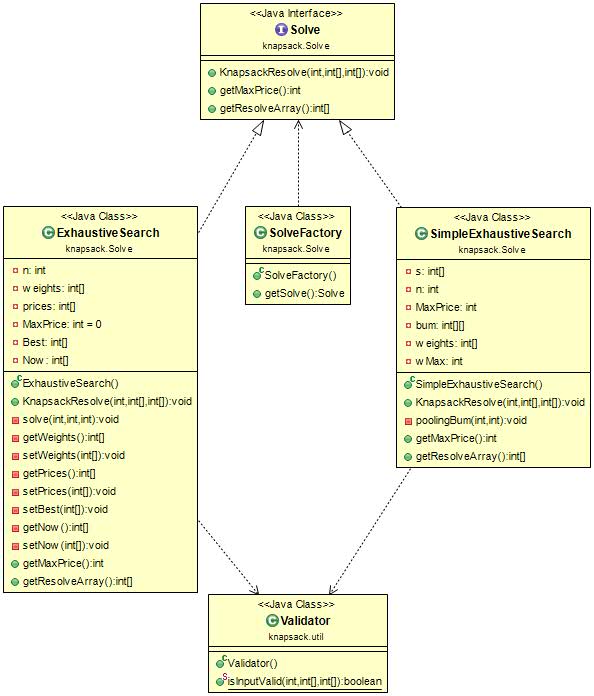


Рисунок 3.5 – UML диаграмма классов поиска решения

## Применение паттерна «Фабрика»

После запуска программы, в классе CmdParams записываются параметры запуска программы. Для выбора реализации интерфейсов InputData и OutputData, применяется паттерн проектирования «Фабрика», реализованный в классах InputFactory и OutputFactory, для данных интерфейсов.

Фабрика InputFactory содержит метод public InputData getInputData(), читающий параметры из класса CmdParams и возвращающий, в зависимости от параметров, конкретную реализацию интерфейса InputData. Если параметр fileIn класса CmdParam не пустой, то возвращается реализация FileInput, при этом в конструктор передается строка, содержащая путь к файлу с исходными данными. Если параметр CmdParams.fileIn не задан, то возвращается экземпляр класса ConsoleInput.

**public** InputData getInputData(){

**if** (CmdParams.*fileIn* != **null**) {

**return** **new** FileInput(CmdParams.*fileIn*);

}**else**{

**return** **new** ConsoleInput();

}

}

Фабрика OutputFactory содержит метод public OutputData getOutputData(), читающий параметры из класса CmdParams и возвращающий, в зависимости от параметров, конкретную реализацию интерфейса OutputData. Если параметр fileOut класса CmdParam не пустой, то возвращается реализация FileOutput, при этом в конструктор передается строка, содержащая путь к файлу для сохранения туда результатов работы программы. Если параметр CmdParams.fileOut не задан, то возвращается экземпляр класса ConsoleOutput.

**public** OutputData getOutputData(){

**if** (CmdParams.*fileOut* != **null**) {

**return** **new** FileOutput(CmdParams.*fileOut*);

}**else**{

**return** **new** ConsoleOutput();

}

}

Поскольку для решения задачи о рюкзаке может быть использовано множество решений, для выбора реализации интерфейса Solve, так же применен паттерн «Фабрика». Однако, в отличие от реализации данного паттерна в InputFactory и OutputFactory, здесь, в фабрике SolveFactory применяется динамическая загрузка классов, реализующих интерфейс Solve. Имя класса для загрузки вводится при старте программы через параметры командной строки.

Для динамической загрузки класса применена следующая конструкция:

Class<?> solveClass = Class.*forName*(path + "." + method);

**return** (Solve) solveClass.newInstance();

## Вспомогательные классы

В данном курсовом проекте применяются два вспомогательных класса CmdParams и Validator, вынесенные в отдельный пакет knapsack.utils.

Класс CmdParams вызывается сразу после запуска программы, после чего получает и хранит параметры запуска программы из командной строки. При необходимости, остальные части программы (через классы-фабрики) обращаются к параметрам сохраненным параметрам через статические поля fileIn, fileOut и solveMethod, которые содержат (если параметры заданы) путь к файлу с исходными данными, путь к файлу, в который требуется записать результат вычислений и способ (класс) решения, соответственно.

Для обработки параметров командной строки используется сторонняя библиотека Commons CLI, являющаяся частью Apache Commons, который в свою очередь является проектом фонда Apache Software Foundation, имеющий своей целью разработку и поддержку открытого программного обеспечения повторного использования на языке Java, то есть библиотек Java. pache Commons содержит набор библиотек утилит Java самого разного назначения, доступных по лицензии Apache License, и использующихся во многих других проектах с открытым исходным кодом. Утилиты проекта Apache Commons лежат в основе таких проектов как Apache Tomcat, Struts, Hibernate и др.

Выбор данной библиотеки существенно упрощает и стандартизирует работу с параметрами запуска программы. Данная библиотека обрабатывает параметры запуска программы, написанные в POSIX (к примеру, tar -zxvf foo.tar.gz) и GNU (du --human-readable --max-depth=1) стилях.

Взаимодействие класса CmdParams с классами-фабриками показано на UML диаграмме на рисунке 3.6.

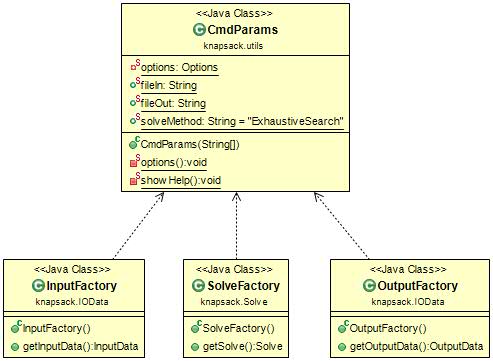


Рисунок 3.6 – UML диаграмма класса CmdParams и классов-фабрик

При этом, остальные классы, за исключением класса Knapsack (точка входа в программу) напрямую не взаимодействуют с классом CmdParams.

Класс Validator служит для проверки передаваемых в методы параметров. В данном курсовом проекте в классе Validator реализован метод isInputValid, проверяющий правильность полученных данных методами классов, реализующих интерфейс Solve.

## Тестирование программы

Для тестирования программы написан класс тестирования TestSolves, реализующий org.junit.Assert.\*.

В данном тесте происходит тестирование классов решения задачи о рюкзаке.

На вход тестируемому классу подаются следующие данные:

* Максимальный объем рюкзака;
* Массив с указанием весов предметов;
* Массив с указанием стоимостей предметов.

На выходе из программы мы должны получить:

* Максимальную стоимость загруженных в рюкзак предметов;
* Массив с указанием какие предметы и в каком количестве загружены в рюкзак.

Поскольку для создания корректных исходных данных требуется вначале так или иначе решить задачу о рюкзаке, воспользуемся готовыми исходными данными, опубликованными в свободном доступе в сети Интернет под лицензией GNU LGPL.

Для теста возьмем единственный набор исходных данных, на котором тестируемая программа должна быть отлажена.

Максимальный объем: 26.

Массив стоимостей: 24, 13, 23, 15, 16

Массив весов: 12, 7, 11, 8, 9.

Ожидаемый результат:

Массив выбранных предметов: 0, 1, 1, 1, 0

Максимизированная стоимость: 51.

В качестве примера приведен тест для класса ExhaustiveSearch

@Test

**public** **void** testExhaustiveSearch() {

**for** (**int** i = 0; i < *weightMax*.length; i++) {

ExhaustiveSearch testKnapsack = **new** ExhaustiveSearch();

testKnapsack.KnapsackResolve(*weightMax*[i], *arrWeights*[i], *arrPrices*[i]);

*assertArrayEquals*(testKnapsack.getResolveArray(), *arrCheck*[i]);

*assertEquals*(testKnapsack.getMaxPrice(), *priceMaxCheck*[i]);

}

}

В данном тесте применяется цикл для последовательного тестирования нескольких наборов исходных данных.

# Руководство пользователя

## Минимальные системные требования

Для установки и запуска программы «Задача о рюкзаке» необходимоа установленная виртуальная машина Java (Java 2 Platform, Standard Edition), не ниже 8 версии. Установочные файлы Java можно загрузить с сервера компании Sun, по адресу: http://java.sun.com.

## Установка и удаление программы

Для установки программы необходимо скопировать файл knapsack.jar в любую директорию.

Для запуска программы необходимо ввести следующую команду в консоли:

**java -jar %path% knapsack.jar <params>, где:**

%path% - путь к файлу knapsack.jar, если вы уже находитесь в этой папке, то путь указывать не надо;

<params> - параметры запуска программы.

Для удаления программы, удалите файл knapsack.jar.

## Работа с программой

Программа предусматривает в себе два режима работы: консольный и файловый (пакетный).

Консольный режим работы.

При запуске программы без указания параметров, программа запустится в консольном режиме работы. В данном режиме работы реализован диалоговый режим работы. Программа запрашивает ввод исходных данных:

1. Ввод максимальной вместимости рюкзака;
2. Ввод набора весов предметов. Веса вводятся через пробел. Количество предметов соответствует количеству весов.
3. Ввод набора стоимостей предметов. Стоимости вводятся через пробел. Количество вводимых стоимостей должно соответствовать количеству введенных весов.

После ввода максимальной вместимости рюкзака, наборов весов и стоимостей, следует нажимать клавишу Enter на устройстве ввода.

После ввода всех исходных данных в консоль пользователю выводится результат. А именно: количество выбранных предметов и максимизированная стоимость.

Запуск программы без параметров в диалоговом режиме показан на рисунке 4.1.

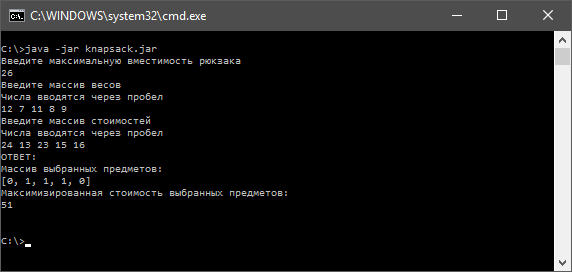


Рисунок 4.1 – Запуск программы без параметров

При запуске программы можно указать следующие параметры:

-a,--algorithm <arg> Алгоритм для решения задачи.

-i,--fileIn <path> Файл-источник данных

-o,--fileOut <path> Файл для записи результатов

Пример запуска программы в пакетном режиме с использованием файла с исходными данными input.txt, файла вывода результатов output.txt и указанием алгоритма решения задачи в GNU-стиле:

Java –jar knapsack.jar --fileIn="input.txt" --fileOut="output.txt" --algorithm= SimpleExhaustiveSearch

Аналогично можно запустить программу с параметрами в POSIX-стиле:

Java –jar knapsack.jar -i="input.txt" -o="output.txt" –a= SimpleExhaustiveSearch

Каждый параметр запуска программы не является обязательным. Например, при указании только параметра --fileIn, вывод результата работы программы будет осуществлен в консоль. Если помимо параметра --fileIn будет указан параметр --fileOut, то результат выполнения программы будет записан в файл. При необходимости файл будет создан.

Пример запуска программы с указанием исходного файла через параметр --fileIn и альтернативного алгоритма через параметр --algorithm, приведен на рисунке 4.2. В данном примере файл input.txt содержал два набора исходных данных, при решении задачи был применен алгоритм SimpleExhaustiveSearch, а поскольку файл вывода не указан, то результат решения для каждого набора исходных данных выводится пользователю в консоль.

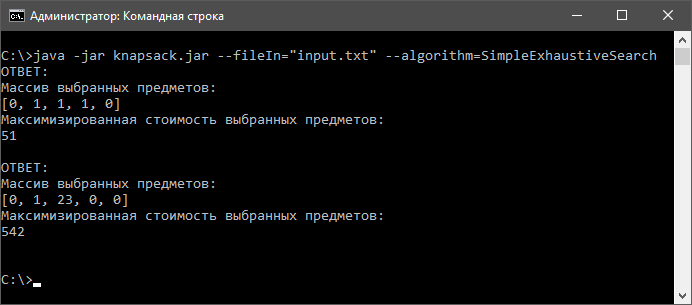


Рисунок 4.2 – Запуск программы с параметрами

# ЗАКЛЮЧЕНИЕ

В курсовой работе описан процесс разработки программы «Задача о рюкзаке». Исследован полный перебор (рекурсивно) и динамическое программирование. Результаты всех исследованных алгоритмов приведены в приложении А. Раздел 4 включает в себя руководство пользователя, описывающее: системные требования, процессы установки, настройки и удаления программы. Окончательная версия программы – приложение для Java платформы. В ходе выполнения, программа использует консольный и файловый ввод/вывод.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

x

|  |  |
| --- | --- |
| 1 | Окулов С.М. Программирование в алгоритмах. Москва: Бином, 2004. 341 с. |
| 2 | Burkardt J. KNAPSACK\_01 // Dept. of Scientific Computing - Florida State University. URL: https:/​/​people.sc.fsu.edu/​~jburkardt/​datasets/​knapsack\_01/​knapsack\_01.html (дата обращения: 23.Февраль.2016). |
| 3 | Тимошенко С.И., Кичигин В.Н., Мясников И.Е. Оформление курсовых и дипломных проектов: методические указания для студентов технических специальностей. Екатеринбург. 2005. 80 с. |
| 4 | Тимошенко С.И. Выполнение курсовой работы по дисциплине "программирование на языке высокого уровня": методические указания для студентов специальности 230150 "Программное обеспечение вычислительной техники и автоматизированных систем". Екатеринбург. 2008. 21 с. |
| 5 | Шилдт Г. Java 8. Полное руководство, 9-е изд.: Пер. с англ. - М. : ООО "И.Д. Вильямс". Москва: Издательский дом "Вильямс", 2015. 1376 с. |
| 6 | Селтинг С., Маасен О. Применение шаблонов Java. Библиотека профессионала. : Пер. с англ. — М.: Издательский дом "Вильямс". Москва. 2002. 576 с. |

x

# ПРИЛОЖЕНИЕ А (обязательное) Исходный код программы

## А.1 Основной класс программы

/\*\*

\*

\*/

**package** knapsack;

**import** knapsack.IOData.\*;

**import** knapsack.Solve.\*;

**import** knapsack.utils.CmdParams;

/\*\*

\* Задача о рюкзаке

\* **@author** Дмитрий

\*/

**public** **class** Knapsack {

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

**new** CmdParams(args);

InputData in = **new** InputFactory().getInputData();

**do** {

**int** wMax = in.getWMax();

**int**[] weights = in.getWeights();

**int**[] prices = in.getPrices();

Solve solve = **new** SolveFactory().getSolve();

solve.KnapsackResolve(wMax, weights, prices);

OutputData outSolve = **new** OutputFactory().getOutputData();

outSolve.OutputResult(solve);

} **while** (!in.stopInputLoop());

}

}

## А.2 Пакет IOData

### А.2.1 Интерфейс InputData

**package** knapsack.IOData;

/\*\*

\* Интерфейс ввода (получения) исходных данных

\* **@author** Дмитрий

\*

\*/

**public** **interface** InputData {

/\*\*

\*

\* **@return** Максимальный объем рюкзака

\*/

**public** **int** getWMax();

/\*\*

\*

\* **@return** массив набора весов

\*/

**public** **int**[] getWeights();

/\*\*

\*

\* **@return** массив набора стоимостей

\*/

**public** **int**[] getPrices();

/\*\*

\* Метод, возвращающий false, если необходимо

\* читать сразу много исходных данных (ввод из файла)

\* **@return** true, если больше не нужно повторять ввод данных

\*/

**public** **boolean** stopInputLoop();

}

### А.2.2 Реализация интерфейса InputData. Класс ConsoleInput

**package** knapsack.IOData;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

**public** **class** ConsoleInput **implements** InputData {

//читаем исходные данные с консоли всегда 1 раз.

**private** **boolean** stopInputLoop = **true**;

@Override

**public** **int** getWMax() {

**int** wmax = 0;

**while** (wmax == 0) {

wmax = inputConsoleInt("Введите максимальную вместимость рюкзака");

}

**return** wmax;

}

@Override

**public** **int**[] getWeights() {

**int**[] arr = **null**;

**while** (arr == **null**) {

arr = inputConsoleIntArr("Введите массив весов\n"

+ "Числа вводятся через пробел");

}

**return** arr;

}

@Override

**public** **int**[] getPrices() {

**int**[] arr = **null**;

**while** (arr == **null**) {

arr = inputConsoleIntArr("Введите массив стоимостей\n"

+ "Числа вводятся через пробел");

}

**return** arr;

}

**private** **int** inputConsoleInt(String msg) {

**int** res = 0;

System.***out***.println(msg);

Scanner consoleInput = **null**;

**try**{

consoleInput = **new** Scanner(System.***in***);

res = consoleInput.nextInt();

}**catch** (InputMismatchException e){

System.***err***.println("Ошибка ввода числа. повторите ввод.");

}**catch** (Exception e){

e.printStackTrace();

} **finally** {

// **TODO** при раскомментировании данной строки метод inputConsoleIntArr не работает.

//consoleInput.close();

}

**return** res;

}

/\*\*

\* Ввод массива чисел с консоли

\* **@param** msg строка приглашения для ввода

\* **@return** массив чисел типа Int

\* **@throws** FileNotFoundException

\*/

**private** **int**[] inputConsoleIntArr(String msg){

System.***out***.println(msg);

**int**[] arr = **null**;

Scanner consoleInput = **null**;

Scanner input = **null**;

**try** {

ArrayList<Integer> arrInput = **new** ArrayList<Integer>();

consoleInput = **new** Scanner(System.***in***);

input = **new** Scanner(consoleInput.nextLine());

**while** (input.hasNext()) {

arrInput.add(input.nextInt());

}

arr = **new** **int**[arrInput.size()];

**for** (**int** i = 0; i < arr.length; i++) {

arr[i] = arrInput.get(i);

}

}**catch** (InputMismatchException e){

System.***err***.println("Ошибка ввода. повторите ввод.");

}**catch** (Exception e) {

e.printStackTrace();

}**finally**{

// **TODO** при расскоментировании прога падает.

// consoleInput.close();

// input.close();

}

**return** arr;

}

@Override

**public** **boolean** stopInputLoop() {

**return** stopInputLoop;

}

}

### А.2.3 Реализация интерфейса InputData. Класс FileInput

**package** knapsack.IOData;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

**public** **class** FileInput **implements** InputData {

**private** ArrayList<String> fileData;

**private** **boolean** stopInputLoop = **false**;

**private** **static** **int** *linesOfData* = 0;

**private** **static** **int** *countReadLn* = 0;

**public** FileInput(String file){

BufferedReader reader = **null**;

**try** {

reader = **new** BufferedReader(**new** FileReader(file));

} **catch** (FileNotFoundException e) {

System.***out***.println("Файл с данными не обнаружен");

//throw new IllegalArgumentException("файл с данными не обнаружен");

**return**;

}

String line;

fileData = **new** ArrayList<String>();

**try** {

**while** ((line = reader.readLine()) != **null**) {

line = line.trim();

**if** (!line.isEmpty() && line.charAt(0)!='#') fileData.add(line);

}

*linesOfData* = fileData.size();

} **catch** (IOException e) {

System.***out***.println("Ошибка чтения данных из файла");

} **finally** {

**try** {

reader.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

@Override

**public** **int** getWMax() {

Scanner fileInt = **new** Scanner(fileData.get(startLn()));

**int** res = 0;

**try**{

res = fileInt.nextInt();

}**catch** (InputMismatchException e){

System.***err***.println("Ошибка чтения числа из файла. Проверьте формат файла");

}**catch** (Exception e){

**throw** **new** Error();

}**finally**{

fileInt.close();

}

**return** res;

}

@Override

**public** **int**[] getWeights() {

**int**[] arr = **null**;

arr = fileInputIntArr(startLn());

**return** arr;

}

@Override

**public** **int**[] getPrices() {

**int**[] arr = **null**;

arr = fileInputIntArr(startLn());

**return** arr;

}

**private** **int**[] fileInputIntArr(**int** startLn){

**int**[] arr = **null**;

Scanner fileinput = **null**;

Scanner input = **null**;

**try** {

ArrayList<Integer> arrInput = **new** ArrayList<Integer>();

fileinput = **new** Scanner(fileData.get(startLn));

input = **new** Scanner(fileinput.nextLine());

**while** (input.hasNext()) {

arrInput.add(input.nextInt());

}

arr = **new** **int**[arrInput.size()];

**for** (**int** i = 0; i < arr.length; i++) {

arr[i] = arrInput.get(i);

}

}**catch** (InputMismatchException e){

System.***err***.println("Ошибка чтения массива из файла. Проверьте формат файла");

}**catch** (Exception e) {

**throw** **new** Error();

} **finally** {

fileinput.close();

input.close();

}

**return** arr;

}

@Override

**public** **boolean** stopInputLoop() {

**return** stopInputLoop;

}

**private** **int** startLn(){

**if**(*countReadLn* == *linesOfData* -3){

stopInputLoop = **true**;

}

**if**(*countReadLn* < *linesOfData*){

*countReadLn*++;

**return** *countReadLn*-1;

}

**return** 0;

}

}

### А.2.4 Фабрика InputFactory

**package** knapsack.IOData;

**import** knapsack.utils.CmdParams;

/\*\*

\* Фабрика выбора конкретной реализации интерфейса

\* получения (ввода) данных, в зависимости от

\* установленных параметров командной строки

\* **@author** Дмитрий

\*

\*/

**public** **class** InputFactory {

/\*\*

\* выбор реализации интерфейса ввода данных.

\* Если параметр fileIn класса CmdParam не пустой,

\* то возвращается реализация FileInput.

\* **@return** экземпляр реализации интерфейса InputData

\*/

**public** InputData getInputData(){

**if** (CmdParams.*fileIn* != **null**) {

**return** **new** FileInput(CmdParams.*fileIn*);

}**else**{

**return** **new** ConsoleInput();

}

}

}

### А.2.5 Интерфейс OutputData

**package** knapsack.IOData;

**import** knapsack.Solve.Solve;

/\*\*

\* Интерфейс вывода (сохранения в файл) результатов решения задачи

\* **@author** Дмитрий

\*

\*/

**public** **interface** OutputData {

/\*\*

\* Вывод (сохранение) результатов

\* **@param** knapsackSolve Экземпляр решения задачи

\*/

**public** **void** OutputResult(Solve knapsackSolve);

}

### А.2.6 Реализация интерфейса OutputData. Класс ConsoleOutput

**package** knapsack.IOData;

**import** java.util.Arrays;

**import** knapsack.Solve.Solve;

**public** **class** ConsoleOutput **implements** OutputData {

@Override

**public** **void** OutputResult(Solve knapsackSolve) {

System.***out***.println("ОТВЕТ:");

System.***out***.println("Массив выбранных предметов:");

System.***out***.println(Arrays.*toString*(knapsackSolve.getResolveArray()));

System.***out***.println("Максимизированная стоимость выбранных предметов:");

System.***out***.println(knapsackSolve.getMaxPrice());

}

}

### А.2.7 Реализация интерфейса OutputData. Класс FileOutput

**package** knapsack.IOData;

**import** java.io.BufferedWriter;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.OutputStreamWriter;

**import** java.io.UnsupportedEncodingException;

**import** java.io.Writer;

**import** java.util.Arrays;

**import** knapsack.Solve.Solve;

**public** **class** FileOutput **implements** OutputData {

**private** String file;

**public** FileOutput(String file) {

**this**.file = file;

}

@Override

**public** **void** OutputResult(Solve knapsackSolve) {

Writer writer = **null**;

**try** {

FileOutputStream fileout = **new** FileOutputStream(file, **true**);

OutputStreamWriter outWriter = **new** OutputStreamWriter(fileout, "utf-8");

writer = **new** BufferedWriter(outWriter);

writer.write(knapsackSolve.getMaxPrice() + "\n");

writer.write(Arrays.*toString*(knapsackSolve.getResolveArray()) + "\n");

} **catch** (FileNotFoundException | UnsupportedEncodingException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

writer.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

### А.2.8 Фабрика OutputFactory

**package** knapsack.IOData;

**import** knapsack.utils.CmdParams;

/\*\*

\* Фабрика выбора конкретной реализации интерфейса

\* вывода (сохранения) результатов решения задачи, в зависимости от

\* установленных параметров командной строки

\* **@author** Дмитрий

\*

\*/

**public** **class** OutputFactory {

/\*\*

\* Выбор реализации в зависимости от установленного

\* параметра CmdParam.fileOut Если параметр не пустой -

\* возвращается экземпляр FileOutput.

\* **@return** Экземпляр реализации интерфейса OutputData

\*/

**public** OutputData getOutputData(){

**if** (CmdParams.*fileOut* != **null**) {

**return** **new** FileOutput(CmdParams.*fileOut*);

}**else**{

**return** **new** ConsoleOutput();

}

}

}

## А.3 Пакет Solve

### А.3.1 Интерфейс Solve

**package** knapsack.Solve;

/\*\*

\* Интерфейс решения (алгоритм) задачи о рюкзаке

\* **@author** Дмитрий

\*

\*/

**public** **interface** Solve {

/\*\*

\* метод (алгоритм) решения задачи о рюкзаке

\* **@param** wMax Макс вместимость рюкзака

\* **@param** weights Массив весов

\* **@param** prices Массив стоимостей

\*/

**public** **void** KnapsackResolve(**int** wMax, **int**[] weights, **int**[] prices);

/\*\*

\* Получение решения задачи о рюкзаке

\* **@return** максимизированная стоимость

\*/

**public** **int** getMaxPrice();

/\*\*

\* Получение решения задачи о рюкзаке

\* **@return** Массив выбранных предметов

\*/

**public** **int**[] getResolveArray();

}

### А.3.2 Реализация интерфейса Solve. Класс ExhaustiveSearch

**package** knapsack.Solve;

**import** knapsack.utils.Validator;

/\*\*

\* Класс для решения задачи о рюкзаке методом полного перебора (рекурсивно).

\* Код (адаптированный) из книжки Окулов Программирование в алгоритмах 2004 с.92-93

\* **@author** Дмитрий

\*/

**public** **class** ExhaustiveSearch **implements** Solve{

**private** **int** n; //кол-во типов предметов

**private** **int**[] weights; //набор весов

**private** **int**[] prices; //набор стоимостей

**private** **int** MaxPrice = 0; //Максимальная стоимость предметов

**private** **int**[] Best; //Лучшее решение

**private** **int**[] Now; //Текущее решение

/\*\*

\* Рекурсивный полный перебор

\* **@param** wMax максимальная вместимость рюкзака

\* **@param** weights массив весов предметов

\* **@param** prices массив стоимостей предметов

\*/

**public** **void** KnapsackResolve(**int** wMax, **int**[] weights, **int**[] prices) {

**if** (!Validator.*isInputValid*(wMax, weights, prices)) {

System.*exit*(-1);

}

setWeights(weights.clone());

setPrices(prices);

n = weights.length;

setNow(**new** **int**[n]);

setBest(**new** **int**[n]);

solve(0,wMax,0);

}

/\*\*

\* Рекурсивный полный перебор

\* **@param** k порядковый номер группы предметов

\* **@param** w вес, который требуется набрать из оставшихся предметов

\* **@param** st Стоимость текущего решения

\*/

**private** **void** solve(**int** k, **int** w, **int** st){

**if** ((k >= n && st > MaxPrice) ) {

setBest(getNow().clone());

MaxPrice = st;

} **else** **if** (k < n){

**for** (**int** i = 0; i <= (w / getWeights()[k]); i++) {

getNow()[k] = i;

solve(k+1, w-i\*getWeights()[k], st+i\*getPrices()[k]);

}

}

}

**private** **int**[] getWeights() {

**return** weights;

}

**private** **void** setWeights(**int**[] weights) {

**this**.weights = weights;

}

**private** **int**[] getPrices() {

**return** prices;

}

**private** **void** setPrices(**int**[] prices) {

**this**.prices = prices;

}

**private** **void** setBest(**int**[] best) {

Best = best;

}

**private** **int**[] getNow() {

**return** Now;

}

**private** **void** setNow(**int**[] now) {

Now = now;

}

**public** **int** getMaxPrice() {

**return** MaxPrice;

}

**public** **int**[] getResolveArray() {

**return** Best;

}

}

### А.3.3 Реализация интерфейса Solve. Класс SimpleExhaustiveSearch

**package** knapsack.Solve;

**import** knapsack.utils.Validator;

/\*\*

\*

\*/

**public** **class** SimpleExhaustiveSearch **implements** Solve {

**private** **int** s[];

**private** **int** n;

**private** **int** MaxPrice;

**private** **int**[][] bum;

**private** **int**[] weights;

**private** **int** wMax;

@Override

**public** **void** KnapsackResolve(**int** wMax, **int**[] weights, **int**[] prices) {

**if** (!Validator.*isInputValid*(wMax, weights, prices)) {

System.*exit*(-1);

}

**this**.weights = weights;

**this**.wMax = wMax;

n = weights.length;

s = **new** **int**[wMax +1];

bum = **new** **int**[wMax +1][n];

s[0]=0;

**for**(**int** i=1; i<s.length; i++){

s[i]=-1;

}

**int** m;

**for** (**int** i=1; i<s.length; i++){

**int** num = -1;

**for**(**int** j=0;j<n;j++){

**if**((i-weights[j])<0) **continue**;

**if**(s[i-weights[j]]==-1) **continue**;

m = s[i-weights[j]]+prices[j];

**if** (s[i]<m){

s[i]=m;

num = j;

}

}

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

poolingBum(i,num);

bum[i][num]++;

}**else**{

bum[i][0]=0;

}

}

MaxPrice = s[wMax];

}

**private** **void** poolingBum(**int** procW, **int** numer){

**for** (**int** i = 0; i < n; i++) {

bum[procW][i] = bum[procW-weights[numer]][i];

}

}

@Override

**public** **int** getMaxPrice() {

**return** MaxPrice;

}

@Override

**public** **int**[] getResolveArray() {

**return** bum[wMax];

}

}

### А.3.4 Фабрика SolveFactory

**package** knapsack.Solve;

**import** knapsack.utils.CmdParams;

/\*\*

\* Фабрика выбора конкретной реализации интерфейса

\* решения (алгоритм) задачи, в зависимости от

\* установленных параметров командной строки

\* **@author** Дмитрий

\*

\*/

**public** **class** SolveFactory {

/\*\*

\* Выбор реализации в зависимости от установленного

\* параметра CmdParam.solveMethod

\* **@return** Экземпляр реализации интерфейса Solve

\*/

**public** Solve getSolve(){

String method = CmdParams.*solveMethod*;

String path = "knapsack.Solve";

**try** {

Class<?> solveClass = Class.*forName*(path + "." + method);

**return** (Solve) solveClass.newInstance();

} **catch** (Exception e) {

System.***out***.println("Неверное значение параметра запуска программы: --algorithm");

System.*exit*(-1);

}

**return** **null**;

}

}

## А.4 Пакет utils. Вспомогательные классы

### А.4.1 Класс CmdParams

**package** knapsack.utils;

**import** org.apache.commons.cli.\*;

/\*\*

\* Класс для работы с параметрами командной строки при запуске программы.

\* Все статическое, из-за специфики класса (параметры не изменяются во время работы программы).

\* **@author** Дмитрий

\*

\*/

**public** **class** CmdParams {

**private** **static** Options *options*;

**public** **static** String *fileIn*;

**public** **static** String *fileOut*;

**public** **static** String *solveMethod* = "ExhaustiveSearch"; //Алгоритм решения задачи по-умолчанию.

/\*\*

\* Установка (парсинг) опций из командной строки в переменные

\* **@param** args

\*/

**public** CmdParams(String[] args) {

*options*();

CommandLineParser parser = **new** DefaultParser();

**try** {

CommandLine line = parser.parse(*options*, args);

**if** (line.hasOption("fileIn")) {

*fileIn* = line.getOptionValue("fileIn");

}

**if** (line.hasOption("fileOut")) {

*fileOut* = line.getOptionValue("fileOut");

}

**if** (line.hasOption("algorithm")) {

*solveMethod* = line.getOptionValue("algorithm");

}

} **catch** (ParseException e) {

System.***out***.println("Неверные параметры командной строки.");

*showHelp*();

System.*exit*(1);

}

}

/\*\*

\* Задание опций (параметров) командной строки

\*/

**private** **static** **void** options() {

*options* = **new** Options();

*options*.addOption(Option.*builder*("i")

.longOpt("fileIn")

.hasArg()

.desc("Файл-источник данных")

.argName("path")

.build());

*options*.addOption(Option.*builder*("o")

.longOpt("fileOut")

.desc("Файл для записи результатов")

.hasArg()

.argName("path")

.build());

*options*.addOption(Option.*builder*("a")

.longOpt("algorithm")

.desc("Алгоритм для решения задачи.")

.hasArg()

.build());

}

/\*\*

\* Метод отображения справки по программе в командной строке

\*/

**private** **static** **void** showHelp() {

HelpFormatter formatter = **new** HelpFormatter();

String usage = "knapsack --fileIn=\"input.txt\" --fileOut=\"output.txt\"";

String header = "Программа для нахождения решения \"Задачи о рюкзаке\"";

String footer = "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";

formatter.printHelp(usage, header, *options*, footer, **false**);

}

}

### А.4.2 Класс Validator

**package** knapsack.utils;

**public** **class** Validator {

**public** **static** **boolean** isInputValid(**int** wMax, **int**[] weights, **int**[] prices){

**if**(weights.length != prices.length) {

System.***out***.println("Ошибка! Массивы стоимостей и весов должны быть одного размера");

**return** **false**;

}

**if**(weights.length == 0 || prices.length == 0) {

System.***out***.println("Ошибка! Массивы стоимостей и весов имеют нулевой размер");

**return** **false**;

}

**if**(wMax <= 0) {

System.***out***.println("Ошибка! Вместимость рюкзака должна быть больше нуля");

**return** **false**;

}

**return** **true**;

}

}

## А.5 Класс тестирования TestSolves

**package** knapsack.Solve;

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

**import** org.junit.Test;

**import** knapsack.Solve.ExhaustiveSearch;

**public** **class** TestSolves {

**static** **int**[][] *arrPrices* = {

{ 24, 13, 23, 15, 16 },

{ 50, 90, 30, 140 }

};

**static** **int**[][] *arrWeights* = {

{ 12, 7, 11, 8, 9 },

{ 2, 3, 1, 4 }

};

**static** **int**[] *weightMax* = { 26,

6};

**static** **int**[][] *arrCheck* = {

{ 0, 1, 1, 1, 0 },

{ 0, 0, 2, 1 }

};

**static** **int**[] *priceMaxCheck* = { 51,

200};

@Test

**public** **void** testExhaustiveSearch() {

**for** (**int** i = 0; i < *weightMax*.length; i++) {

ExhaustiveSearch testKnapsack = **new** ExhaustiveSearch();

testKnapsack.KnapsackResolve(*weightMax*[i], *arrWeights*[i], *arrPrices*[i]);

*assertArrayEquals*(testKnapsack.getResolveArray(), *arrCheck*[i]);

*assertEquals*(testKnapsack.getMaxPrice(), *priceMaxCheck*[i]);

}

}

@Test

**public** **void** testSimpleExhaustiveSearch() {

**for** (**int** i = 0; i < *weightMax*.length; i++) {

SimpleExhaustiveSearch testKnapsack = **new** SimpleExhaustiveSearch();

testKnapsack.KnapsackResolve(*weightMax*[i], *arrWeights*[i], *arrPrices*[i]);

*assertArrayEquals*(testKnapsack.getResolveArray(), *arrCheck*[i]);

*assertEquals*(testKnapsack.getMaxPrice(), *priceMaxCheck*[i]);

}

}

}