**Friends Package**

**Solution/Design Approach**

Version 1.0

**Nov** 22, 2020

Table of Contents

[Introduction 1](#__RefHeading___Toc1345_2061943338)

[Purpose 1](#__RefHeading___Toc1347_2061943338)

[Document Conventions 1](#__RefHeading___Toc1349_2061943338)

[Requirement 2](#__RefHeading___Toc1351_2061943338)

[Constraints 2](#__RefHeading___Toc1353_2061943338)

[1. Class Digram 3](#__RefHeading___Toc1355_2061943338)

[Java Principals and Design Pattern Applied 4](#__RefHeading___Toc1357_2061943338)

# Introduction

## Purpose

This document describes the solution or design approach for the friends package requirement.

## Document Conventions

This document contains two main types of information: requirements statements and explanatory text. Requirements statements are numbered, and describe a specific feature or behaviour of the system. In each functional requirement statement, one of the following words must be used, and their usage is similar to the conventions of standards document writing:

* The words **must** or **shall** (either is acceptable) indicate a requirement that must be met in the final product.
* The word **should** indicate a requirement that is desirable in the final product, and every effort should be made to achieve this requirement in the product design and implementation.
* The word **may** indicate a requirement that has been discussed, but that may or may not be included in the final product. As the specification document is reviewed and revised, it is desirable to try to change may statements to **should** or **shall**, to remove ambiguity as the feature moves into design, development, and testing.

Explanatory text is any text that is not a requirement statement. Explanatory text can appear at the beginning of document sections (for general information about that section), or following a requirement (for additional detail about a specific requirement).

# Requirement

You want to send your friend a package with different items. You can choose from a number of `N` items. The items are numbered from 1 to `N`. Each one of these items has a given weight and a given cost (in €), where the weights and costs of the items might be different. The package itself has a weight limit. The combined weight of the items you put in the package must not exceed the weight limit of the package, otherwise the package would be too heavy.

Your goal is to determine which items to put in the package so that the total cost of the items you put inside is as large as possible. In case the total cost the of the packaged items is the same for two sets of items, you should prefer the combination of items which has a lower total weight.

## **Constraints**

1. The maximum weight that a package can hold must be <= 100.

2. There may be up to 15 items you can to choose from.

3. The maximum weight of an item should be <= 100.

4. The maximum cost of an item should be <= €100.

Write a program, preferably in Java, which can be run on the command line in order to solve this problem. The program should take one command line argument, which contains the path to a text file. This text file should contain several lines, each line describing one test case for the problem.

Each line starts with the maximum weight of the package for this test case. It is followed by ` : ` and then the list of descriptions of the items available for packaging. Each item description contains, in parentheses, the item's number, starting at 1, its weight and its cost (preceded by a € sign).

In case of a constraint violation, your program should indicate this fact to the user, for example by throwing an exception with a descriptive message, allowing the user to address this problem.

**Sample Input**

**```**

**81 : (1,53.38,€45) (2,88.62,€98) (3,78.48,€3) (4,72.30,€76) (5,30.18,€9) (6,46.34,€48)**

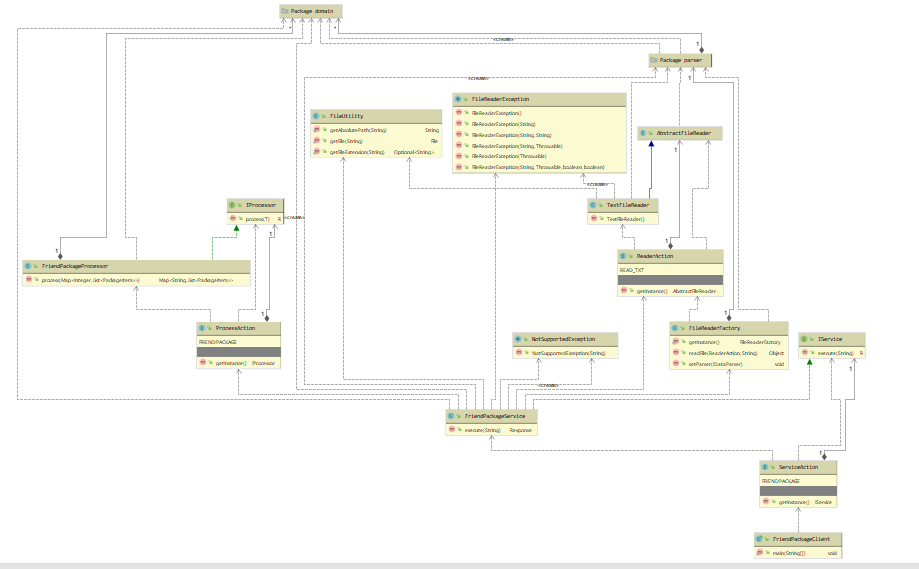
**8 : (1,15.3,€34)**

**75 : (1,85.31,€29) (2,14.55,€74) (3,3.98,€16) (4,26.24,€55) (5,63.69,€52) (6,76.25,€75) (7,60.02,€74) (8,93.18,€35) (9,89.95,€78)**

**56 : (1,90.72,€13) (2,33.80,€40) (3,43.15,€10) (4,37.97,€16) (5,46.81,€36) (6,48.77,€79) (7,81.80,€45) (8,19.36,€79) (9,6.76,€64)**

**```**

# **Class Digram**



# Java Principals and Design Pattern Applied

Applied all S.O.L.D principles approach which is would be easier to develop or add new things and can be managed easily. Below are the S.O.L.D principles used:

Single responsibility principles

Open-closed principles

Liskov Substitution Principle (Not Applied)

Interface Segregation Principle

Dependency Inversion Principle

Following design patterns are approached:

Singleton

Factory (Static Method & Enum)

Command

To calculate package all subset items followed:

Knapsack algorithm