Item Packer Challenge – Low Level Design

6/25/2021
Yusuf Shoair

Table of Contents

1 Item Packer

1.1 Requirements

1.1.1 Functional Requirements

1- Read Input packages from file and parse them to respective models.

2- Develop packing algorithm that would take each package carrying list of items and selects the most value items while not exceeding the package capacity and if items have equal values select the lower weight item.

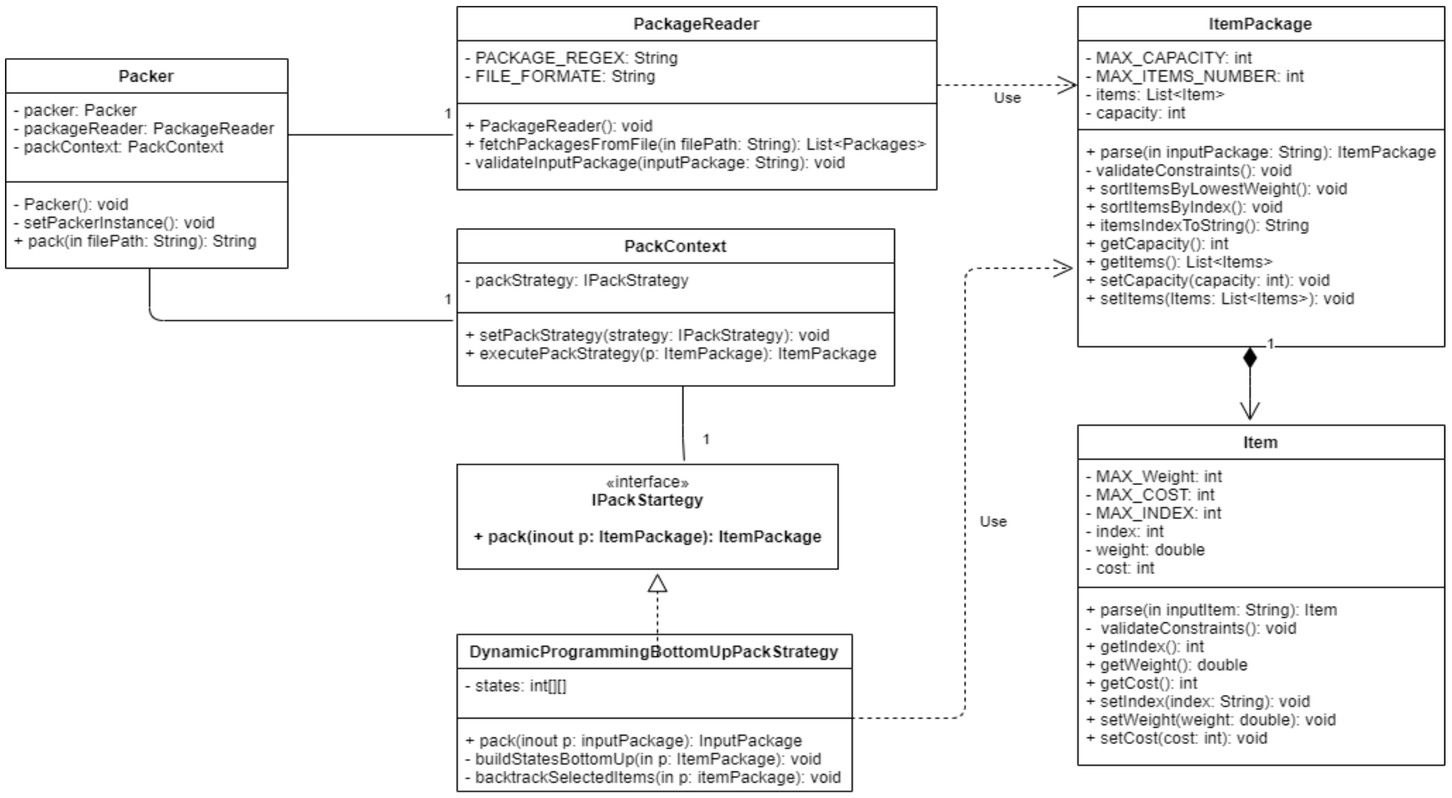
3- Return the selected items comma separated and – if no items exist in the package.

1.1.2 Non-Functional Requirements

1- Validate Input file and throw custom exception in case of validation error.

2- Validate package and item data according to input constraints.

1.2 UML Diagram


Packer
- packer: Packer
- packageReader: PackageReader
- packContext: PackContext
- Packer(): void
- setPackerInstance(): void
+ pack(in filePath: String): String
1
PackageReader
- PACKAGE_REGEX: String
- FILE_FORMATE: String
+ PackageReader(): void
+ fetchPackagesFromFile(in filePath: String): List<Packages>
- validateInputPackage(inputPackage: String): void
Use
ItemPackage
- MAX_CAPACITY: int
- MAX_ITEMS_NUMBER: int
- items: List<Item>
- capacity: int
+ parse(in inputPackage: String): ItemPackage
- validateConstraints(): void
+ sortItemsByLowestWeight(): void
+ sortItemsByIndex(): void
+ itemsIndexToString(): String
+ getCapacity(): int
+ getItems(): List<Items>
+ setCapacity(capacity: int): void
+ setItems(Items: List<Items>): void
1
PackContext
- packStrategy: IPackStrategy
+ setPackStrategy(strategy: IPackStrategy): void
+ executePackStrategy(p: ItemPackage): ItemPackage
1
«interface»
IPackStartegy
+ pack(inout p: ItemPackage): ItemPackage
DynamicProgrammingBottomUpPackStrategy
- states: int[][]
+ pack(inout p: inputPackage): InputPackage
- buildStatesBottomUp(in p: ItemPackage): void
- backtrackSelectedItems(in p: itemPackage): void
Use
Item
- MAX_Weight: int
- MAX_COST: int
- MAX_INDEX: int
- index: int
- weight: double
- cost: int
+ parse(in inputItem: String): Item
- validateConstraints(): void
+ getIndex(): int
+ getWeight(): double
+ getCost(): int
+ setIndex(index: String): void
+ setWeight(weight: double): void
+ setCost(cost: int): void

1.3 Algorithm

1.3.1 Dynamic Programming Bottom Up approach

1- Define state for each item you either take it or leave it out to maximize highest value or cost which can be represented by this function V [n, c] = max {V [n - 1, c], vi + V [n - 1, c -wi]. where n: number of items, c: package capacity, v: item value, w: item weight.

2- Define special cases wi > c then the item will be excluded
3- Define base case c = 0 or n = 0.
4- Construct state[][] in a bottom-up manner with columns representing all possible weights from 0 to C and rows representing all items and state[n][c] will denote maximum value of c-weight considering all values from 1 to nth.
5- After building the array backtrack it from the max value node to get the included items according to this formula state[n][c] != state[n-1][c] then include the item and subtract its weight and value else continue traversing the upper rows.
6- To make the algorithm choose items of lower weight if value is equal the items must be sorted by their weights in ascending order.
7- Item weights must be rounded if item is included when subtracting its weight from the total capacity.

Note: In Case of real number weights the dynamic programming approach is not the best approach; the best approach would be using branch and bound algorithm but when I realized that I have already invested in the DP approach so there was no time to develop another algorithm.

1.4 Design Pattern

1- Singleton design pattern.
2- Strategy Design patterns to be able to add packing strategy seamlessly and achieve open close principle.

1.5 Development Practice

1- Used TDD.

1.6 Future Improvements

1- Using caching to return problem combinations solved before.
2- Use branch and bound algorithm.
3- Use Coding to interfaces for package and item classes to be able to introduce different types of packages and items in the future.