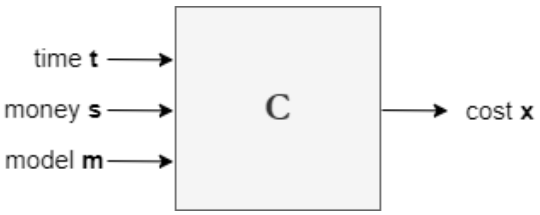
Patiro - Test

By: Iñigo Sanz

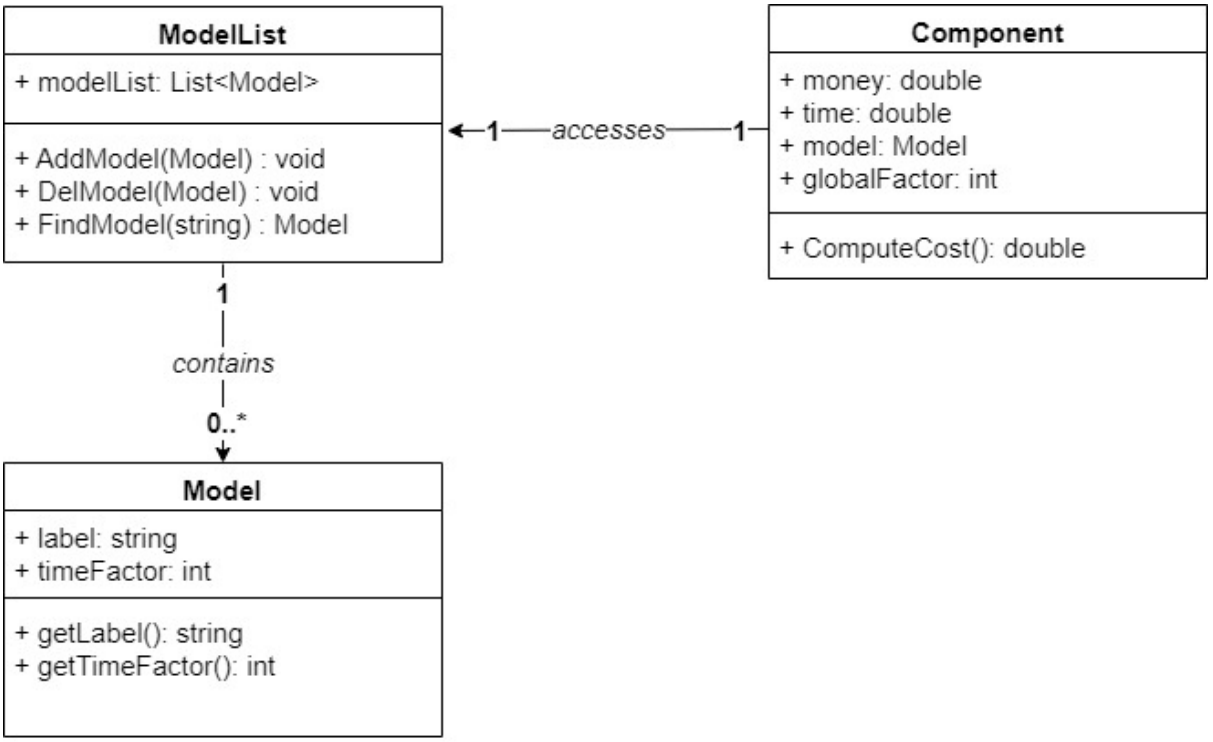
Task 1:

The general view of our system is the following:



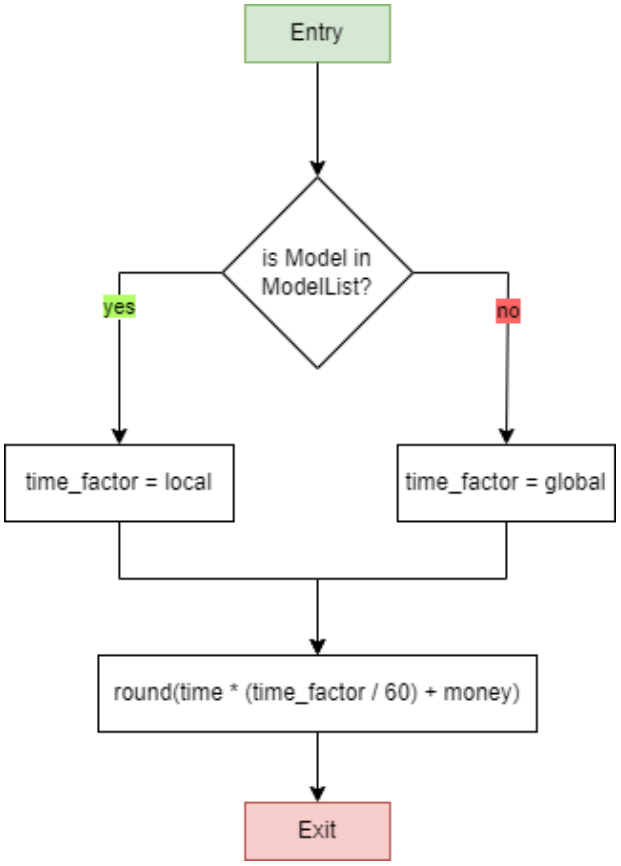
where a component C takes as input the time, money and a model ID and returns a cost result.

Using Object-Oriented Programming to implement the design, our class diagram is as follows:



where the component class accesses a system in order to obtain the Model object it has to use which is indicated by the model ID received through the command line by the user. Since we do not know much about the system, in this example it's a class collecting multiple usable models, named ModelList.

The following diagram shows the control flow of the program:



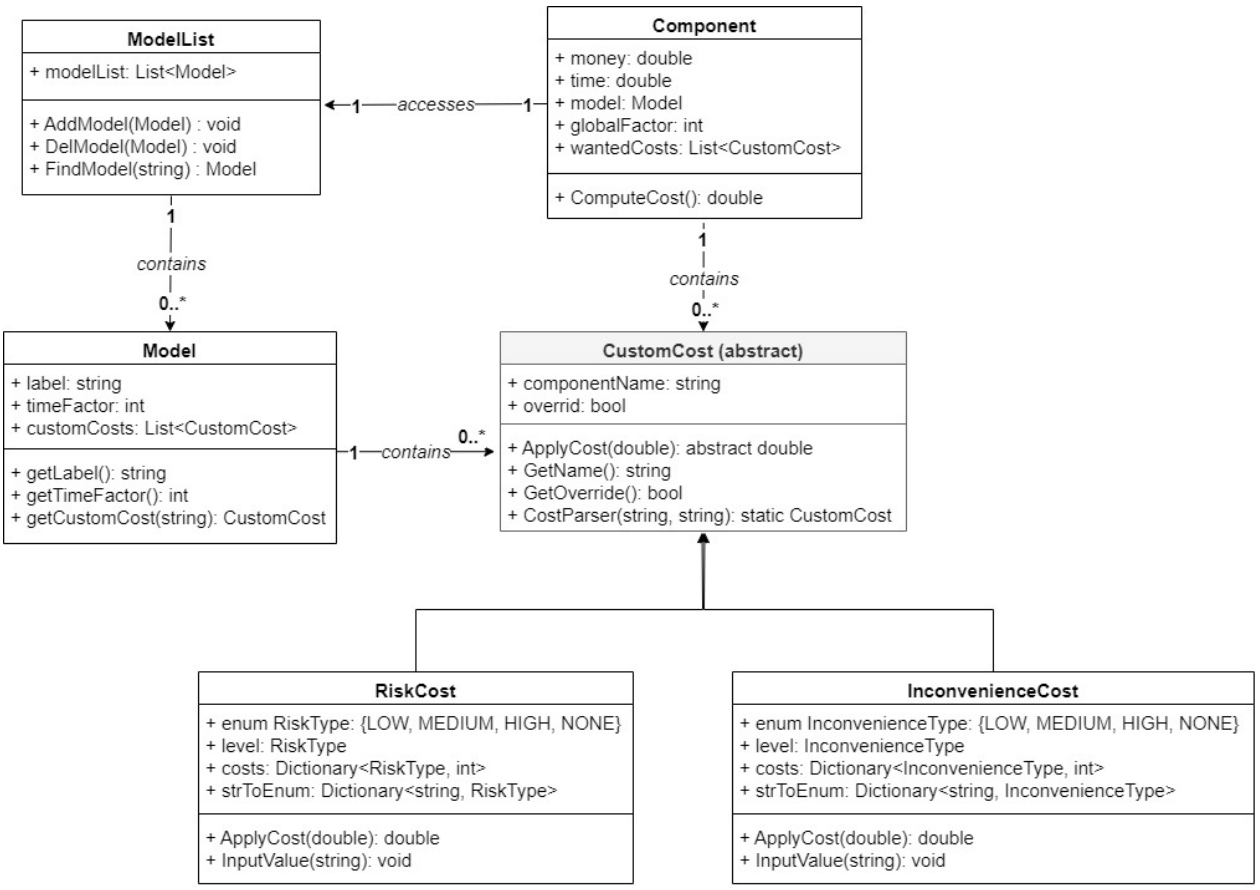
For this first task, the control flow remains simple, as the only changing factor is whether the model that we want to use is within the list of models.

Note that when we calculate the cost in the last statement, the time factor must be divided by 60 as its value is in hours, whereas the time value we were given through the command line is specified in minutes.

Task 2:

The general view of our system remains the same, but certain classes are modified to integrate the custom costs.

The updated class diagram is the following:



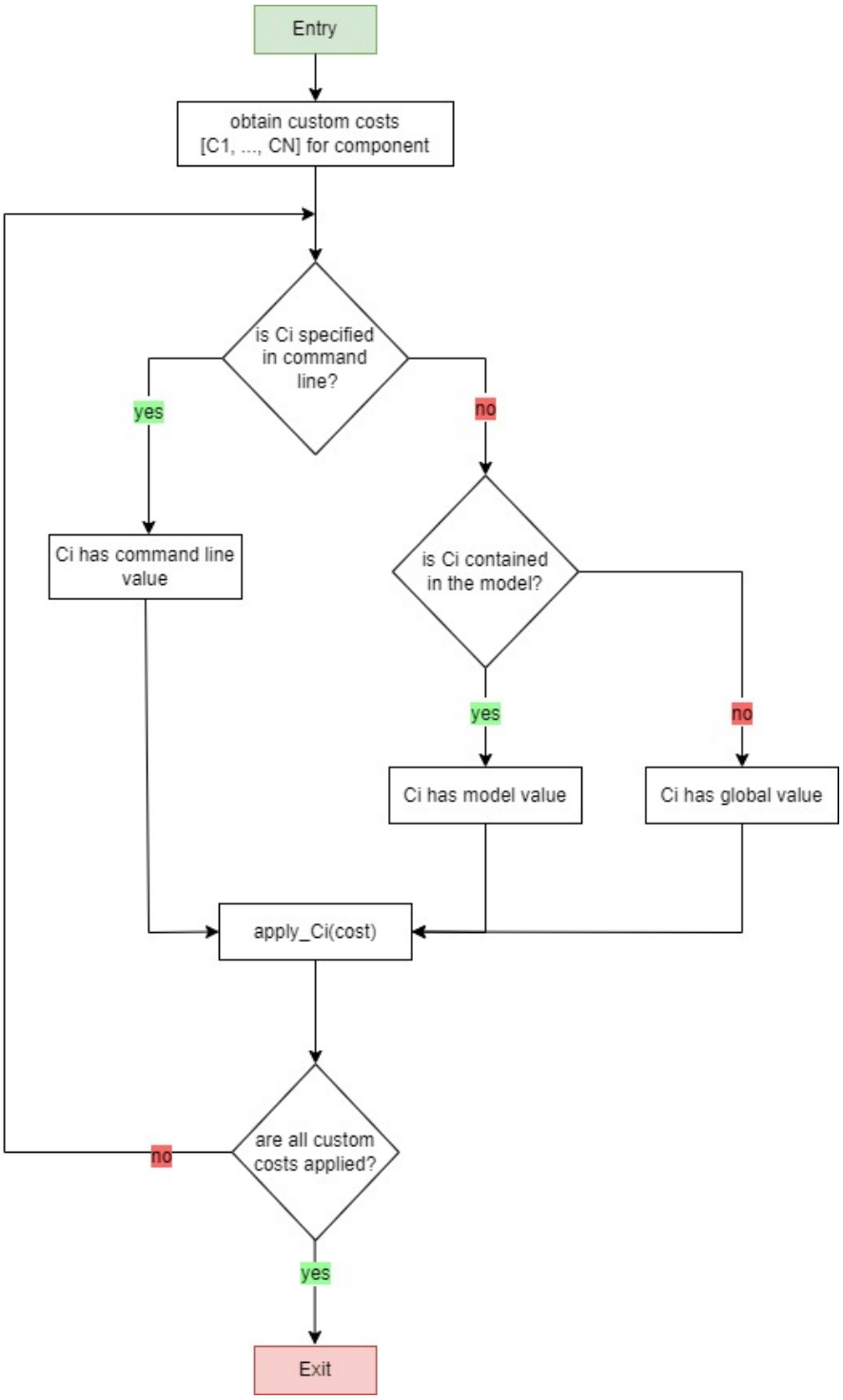
The abstract class CustomCost provides a blueprint for the other custom costs that will be used to modify the initial cost. With it, scalability should be easy to handle since all custom costs follow the same behavior.

The classes RiskCost and InconvenienceCost inherit from their parent class CustomCost, where each one must specify a specific enumerator with the possible values that each custom cost can contain. The dictionary that maps RiskType or InconvenienceType to integers allows us to define the adjustment that each value will make.
Meanwhile, the dictionary that maps strings to RiskType or InconvenienceType allows us to convert the user input in the command line to the values within the enumerators.
The reason why the enumerators are not located in the CustomCost class is because each custom cost can have different values and adjustments.

As for the changes in the initial model, the Model and Component classes now contain a list of elements of CustomCost type.

Each Model object will have their own local factors for each custom cost, just like “model1234” indicates specific factors for the Risk and Inconvenience costs. Moreover, each Component object uses a specific set of custom costs, therefore the list within the object specifies which custom costs are going to be applied.

Regarding the control flow of the program, it builds on top of the previous one:



First and foremost, we need to know which custom costs are going to be applied with the component that we are using. Therefore, for each one of them, we first check if it has been

specified by the user in the command line, as it would take maximum priority. If it can’t be found, then we must check whether the model object that the user specified considers said custom cost. If it can’t be found, then the global value of the custom cost must be used. After the original cost from Part 1 has been calculated, we apply the relative adjustment for each custom cost on the original cost.