# Evaluation of effective mediator pattern in data interpreter

## Introduction

I have implemented the Mediator pattern in my data interpreter. The result has been to add a class to the system which works with the data checkers, making the mediator responsible for the behavior of the colleagues. In my project I have identified InputChecker as Colleague, and RegExp and Enum as ConcreteColleagues.

## Evaluation

Specifying Object Interfaces - An analysis of the system showed that data checkers had a class in common, called InputChecker. This was to determine that both would have a function that allow checking of valid data (isValid) this was the reason I thought the mediator pattern useful for the current system.The use of the interfaces in the mediator pattern are so that reuse is possible. I changed the system so that the mediator handles the checking of data, which i found had the nature of mediator with the colleagues (input checkers) Mediator is supposed to work on its own, handling dependencies of the system. Because of this change, the mediator now can access the checkers and act as a mediator for producing right input.

Programming to an interface, not an implementation - the purpose of interfaces are so that reuse can happen. In the mediator pattern, the mediator acts on behalf of colleagues to make the system handle reuse better, allowing change to happen to the system for next revision. What if you wanted to modify data? The mediator would allow you to check that data regardless of where that was happening. It’s interface means that it allows you to change the mediator and use classes in the system without needing to specify how it does that. This is seen in the way the model handles checking data - it asks the mediator to do this.

Delegation - the way Mediator pattern handles delegation is that requests for the checking are made by asking a checker to check. Mediator does not need to know how, it sends data to the checker to determine this. All the complex methods for determining what checker to check are found in the mediator. This is seen in allowing the mediator to read lines of data and producing a result - either good or bad - from the checkers. If you wanted to, you can accept all data, regardless. This is handled by the mediator too.

Relating run-time and compile-time structures: in the before, the model was dually responsible for the checking and creating data. It was bound to its checkers because it also contained enumerations for each set of data fed to it. This was all done at compile-time - the checkers being instances of a set of data in the model. When I added the mediator, the actual functionality was handled at run-time - when it was accepting input from the program. The model is now unconcerned with those implementation details - it asks the Mediator to implement its request. However, the attributes that are in the mediator are still bound by this data system. It acts as the “data checker” just like the model used to. This is related to the drawback that it prohibits subclassing.

Tight coupling - Before the addition of mediator, the model was tightly coupled to the checking of data, it specified the implementation of the checkers themselves, and related to itself for handling checking. Mediator changes the coupling with dependencies so that a mediator accesses the implementation of its colleagues (dependencies). This means that the functionality required to enable checking has been loosely coupled from the creation of data. When dealing with checking, the model now accesses the mediator, which knows what to do with its dependencies. Only knowing that data is needed to be received back to be processed.

## Recommendations

Adopting subclassing of mediator for handling different requests related to data manipulation. This makes it easy to adopt different behavior regarding manipulating data. This would mean that I would have to redraw the model to be able to handle the manipulation of data. But the mediator would still be used for the sending the data back to the model.