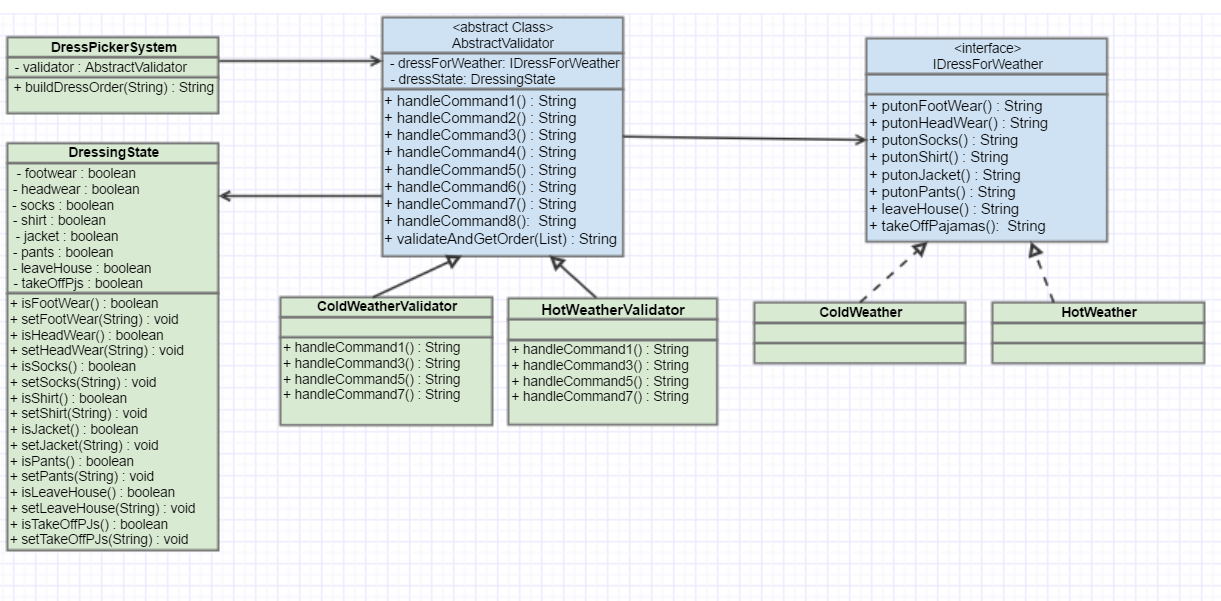
**Class Design:**



**Design Decisions:**

The decision of using an **IDressForWeather** interface allows the **extensible** feature of the system to work easier. This can be extended to other different kinds of temperature such as Rainy, Snowy and Moderate. Also, a new type of clothing could be easily added to the interface allowing the classes implementing the interface to define it.

**AbstractValidator** is a class to validate the commands to be executed and to get the right clothing. It is designed as an abstract class to ensure that the classes inheriting this abstract class can override the abstract methods and can use the common methods from this class.

**DressingState** is a class to store the current state of the input being parsed and the types of clothing that are put on based on the input parsed. A class with methods to put on a clothing was designed to know the previous clothing that are put on to **implement the business rules** when a particular command is executed.

One more design decision that I made was when I needed a mapping between an input state and a command. A switch case would have been done it. The disadvantage of using a switch is that when new commands are added as part of an extension, the switch will be a big block with hardcoded mappings to functions. I had used Java's **Reflection** API to get the corresponding method for a state to handle a command. Although it is not as fast as a switch statement, I had used Reflection for an **ease of maintenance**.

**DressPickerSystemTest** is the class that handles the possible test cases that test the business rules of the dress picker system. All the possible failure and success test cases are being handled in this class.