DESIGN DESCRIPTION

The design prepared for the use case consisted on:
- A client that requests an ordered books list (a class called "Client");
- A class that does that interaction between the client and the right book orderer (called "OrderingService", a singleton class);
- A set of ordering classes that implements a interface to order (the interface "BookOrderer"). Each orderer can aggregate functionalities belonging to other orderer, according to the order rules specified by the client (these rules are set in the configuration file "config.properties"). It was used the design pattern Decorator in order to reach the special requirements;
- A class that represents the book entity (called "Book");
- A class that represents an order rule, formed by an attribute and a direction to order (called "Rule");
- Some helper classes to read and write files, to store constants and properties. For the data input (not ordered list of books), it was chosen a text file called "book_in.txt", having each property separated by semicolon (each line is a different book). For the data output (ordered list of books), it was chosen a text file called "book_out.txt". Both files are inside the directory "client_files" in the project;
- A test case to the class "OrderingService" with jUnit 4. The libraries to run the test case are inside the directory "lib". It was just used libraries to do this test case.

It was chosen the design pattern "Decorator" in order to become the application scalable and in an effort to not to require creating a class for each ordering combination. By using this pattern, it's possible to add functionalities at runtime. In doing so, depending on the configuration file settings, the orderer object will have different behaviors to serve all the client necessities. For each type of ordering, it was created a decorator class. As a result, multiple decorators can be stacked on top of each other, adding a new way to order the books to the overridden method called "order".

Thânia Clair de Souza Vargas - 03/04/2009