**[Object-Oriented Dev w/Components](https://colstate.view.usg.edu/d2l/lp/ouHome/home.d2l?ou=1218642" \o "Enter Obj-Orient Dev w/Components Section V01 Fall Semester 2016 CO)**

**Description of Design Patterns for Term Project**

In this term project I employ three design patterns, respectively. They are one structural pattern - decorator pattern, and two behavioral patterns - iterator pattern and strategy pattern. Firstly, the decorator pattern helps clients/users to add as many components as they want to an object, thus this looks like a “wrapper”. Secondly, in this project iterator pattern helps to navigate through a collection of object using a common interface, and without knowing about the underlying implementations. Thirdly and lastly, there are common situations when classes differ only in their behaviors, in these cases strategy pattern lets the algorithm vary independently from the clients/users that use it. The following paragraphs demonstrate these three design patterns in details and how they are used in the project.

1. As stated in previous proposal, a toy factory needs an application to help their customers build their train set, which, could include several trains as per customers’ requirements. Each train in this train set can be customized according to customers’ needs. Every single train starts from basic mode, which, is one locomotive only. The customers, or application users are able to add components dynamically, for example, fuel tender, powered tender, brake tender, and water cart to the object (the train) whenever there is a need to do so. In this case, the decorator pattern is more convenient for adding functionalities at runtime. The participants classes in the decorator pattern in the project are as follows:
2. Class **MyTrain** is an abstract class (Component) for objects which, can have functionalities added to them dynamically. In this class the method describe() returns description of current train object, the abstract method cost() returns estimate for current train, the method display() prints the description of current train object and its price.
3. Class **BasicTrain** extends **MyTrain**, thus **BasicTrain** can be viewed as Concrete Componnet here, it defines and object to which additional responsibilities can be added. Its constructor and method cost() define the description (Basic Train) and cost ($100) for basic model.
4. Class **TrainDecorator** extends **MyTrain**, it is an abstract class and the concrete implementations should be derived from it. **TrainDecorator** can be viewed as a Decorator here, this Decorator maintains a reference to a Component object. In this abstract class only one abstract method describe(), that returns description for current object.
5. Four classes **FuelTender**, **PoweredTender**, **BrakeTender**, and **WaterCart** are classes extends **TrainDecorator**, they can be view as concrete decorators, which extend the functionality of the component by adding state or components/behaviors. In these classes variable myTrain is declared as class MyTrain, and is initialized by constructors. Method describe() overrides method in Decorator and method cost() updates new price.
6. According to project proposal, the step 1 helps users build their first train and print out the description and price for that train. In this step a collection is used to store more customized train per user’s requirements. A collection is a grouping of a set of objects (trains), it should provide a way to access its elements (train aggregate objects) sequentially without exposing its internal structure. In the project the Array List data structure is used and a mechanism to traverse this Array List is needed. The iterator pattern can help take the responsibility of accessing and passing the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what element is next to be iterated. The participants classes in the iterator pattern in the project are as follows:
7. Interface **iContainer** defines the Aggregate. Method createIterator() returns iIterator.
8. Interface **iIterator** defines the Iterator. There are two methods in the interface: hasNext() and next();
9. Class **TrainCollection** implements **iContainer**. In this class, method createIterator() overrides method in **iContainer**; and method addTrain() adds objects whose type is **MyTrain** into Aggregate. Aggregate class provides a method that generates an iterator object for the data it aggregates.
10. Class **TrainIterator** implements **iIterator**. The methods hasNext() and next() are the implementation of **iIterator** methods.
11. According to project proposal users need to test run each of trains stored in collection. One toy train can move left, right, up, and down in a x-y coordinate plain. For this case strategy pattern can help split these behaviors of a class from the class itself (better encapsulation and decoupling). There are common situations when classes differ only in their behavior, strategy pattern defines an interface common to all supported algorithms, and specific classes implement the interface. The participants classes in the strategy pattern in the project are as follows:
12. Class **MyTrain** is an abstract class, which is involved in step 1 a). For strategy pattern design, a variable moveTrain is declared as interface **MoveTrainInterface** in step 3b). New objects are created by four methods moveLeft(), moveRight(), moveUp(), and moveDown(). Variables x and y are declared, they represent the train’s position in a x-y coordinate plane.
13. Interface **MoveTrainInterface** has only one method: move()
14. Four classes **MoveLeft**, **MoveRight**, **MoveUp**, **MoveDown** implement **MoveTrainInterface**, and override the method move().
15. **TrainIterator** in step 2 d) creates an iterator through **TrainCollection** in step 2 c). With help of iterator every toy train in a customer/user’s train set can be fully tested before purchase.