For this assignment both the command and memento design patterns are implemented differently this is due to the fact the command pattern emoticon generator makes use of the builder design pattern while the my memento design pattern does not. My memento emojitcon generator only makes use of the factory design pattern and the memento design pattern itself, being similar to assignment 4.

Comparing the two I found the command pattern to be easier to implement, this was because I felt the command pattern was more suited towards this assigment and implementing the builder desgin pattern saved development time. Originally I tried to get the memento pattern to implement the builder design pattern but since I was making no progress I decided to try without using the design pattern.

The big difference between the two is how they store commands given; for each commands given by the user, the command pattern only stores the each individaul command while the memento stores an entire copy of the object (at the time that command was issued).

The command pattern storing commands is what made it more suitable since the emoji features could be changed once they were added to the base emoji. So if a user wanted to undo something, the command pattern could just unexecute the previous command (undo the change made to the emoji feature) , to redo it just had to execute the command that was just undone.

Since the memento pattern stores a copy of the object (emoticon in this case), if a user wanted to add, remove or change (move, rotate) an emoji feature the entire emoticon object must be changed for that one command. This need to change the entire memento object stored it what makes the undo/redo functionality cumbersome. To undo, the program has to get the previous version of the entire object and to redo it must get the entire object that was last undone.

The command builder pattern works as follows.

The Product class, “Emoji”, contains the paramters each emoji will have such as their (x,y) values and styling, these paramers are used for the SVG represenation of each emoji feature. The parameters are not set in the emoji class but in a concrete builder, one exists for each emoji feature that can be added. Since concrete builders exist for each feature, this implies a builder interface for all concrete builders, containing abstract setter functions for the emoticons paramters such as x, y and styling values. Each emoji feature is actually implemented as a Concrete Builder, they initialise the parameters in the Emoji class with default values, using the setter methods defined in the builder interface. It should be noted the interface also contains a “getObject” method, which each concrete builder uses to return a copy of itself, the emoji feature object, containing all its parameters.

Finally the Director class, called “IemojiBuilder” determines which interface set methods will be executed. In my program all the interface set methods are ran when creation an emoji feature even though the circle (eyes) and the rectangle (mouth and brows) have different parameters their builder interfaces parameters resolve this issue in a hackish way, storing the radius as the width parameter.

The command pattern works well with the builder pattern as the builder pattern allows each emoji feature have its own values by each being their own concrete builder object, thus each emojis features different parameters can be changed directly. Since all the emoji features are of the same interface the entire command pattern is more felxible since what works for one feature, being able to change its style or move it, works for all features, no feature has their own specific field, although this lack of specilisation means the hackish methods mentioned above, the best part when the director creates the emoji feature object, they can all assign their own initial values, thus when a

new feature is added, depending on what it is it will be in the correct position on the base emoji, also this allows for default styling.

The command pattern does not have to be built around specific emoji feature class, they are all the same interface type. Thus the receiver class (which contains the methods to be executed/unexectued logic) could store all the emoji feature (remember these are all of builder interface type) on the emoji in a single dictionary. The execute and unexecute methods in the actuall “Command” class could thus call methods from the receiver class, only needing to pass the emoji name, since the receiver dicionary can use this name to preform the operation.

The memento pattern does not have this luxury of using the builder pattern, I tried to implement it but since the builder pattern but got confused due to what I said, earlier, I felt the command suited this assignment more.

The memento design pattern works as a tradional memento design pattern would work. The originator stores the current state of the emoji using a dictionary, containing each feature (like the command ), it is dynamic, when a feature is added the originator adds it to the dictionary, when this feature is changed the originator changes the features value using the dictionary to reference the values paramers, and when deleted it is removed from the dictionary. After each command a string represenation of the originater (the svg value for each feature on the emoji) dictionary is stored as a memento and added to the caretaker. Thus the caretaker stores the entire emoji shape.

An abstract class “EmojiFeature” contains all the parameters each emoji feature will (x, y, styling or may not have (width or height, radius for circles).Each emoji feature uses this interface to set their own default values in their constructor (much like the builder interface concrete implementations). The factory pattern is used to create the emoji feature when the user requests one be created. The emoji features the factory returns is sent to the originator, stored in said dicionary.

When a change is made to the shape, the shape is changed by setting the parameter to be changed to a new value, and then getting the memento of the entire originator values svg lines, thus storing the change made.

Overall I am happy I got the undo and redo working for both but their are a few missing features, such as rotate and reset.