CSE 5322 – Software Design Patterns - Fall 2014

Team 7 – Iteration 3

Total Number of Patterns Used: 10

Design Problem (Bridge Pattern)

If MUTO tagging has to be used instead of Stanford NLP tagger, we have to change all the client implementations as MUTO tagging will have its own implementation technique. Client is affected by those changes.

Pattern Identified and Reason

Bridge: It gives a stable client interface through which client is not affected by above changes and he is unaware of which tagging method is implemented. Also in the future adding new class for different tagging method will not affect any client implementation.

Benefits

Client does not know whether tagging is done by NLP or MUTO. Can add other techniques in future without changing client interface.

High Level Idea on how to Implement Bridge on our Tool (High Level Code Snippet)

Client

```
public class IdentificationController {
        public String parseText(String input) throws ClassNotFoundException, IOException {
                Tagger tagger = new Tagger();    String taggedText = tagger.tagText(input);
                colorText(taggedText);          return taggedText;          }       }
```

1. Client Interface Class:

```
public class Tagger {      TaggerImplInterface imp = new NLPImpl();        public      String      tagText(String      input)      throws
ClassNotFoundException, IOException {              String taggedText = imp.tagText(input);          return taggedText;  }    }
```

2. Implementation Interface Class:

```
public interface TaggerImplInterface{
                                public String tagText(String input);        }
```

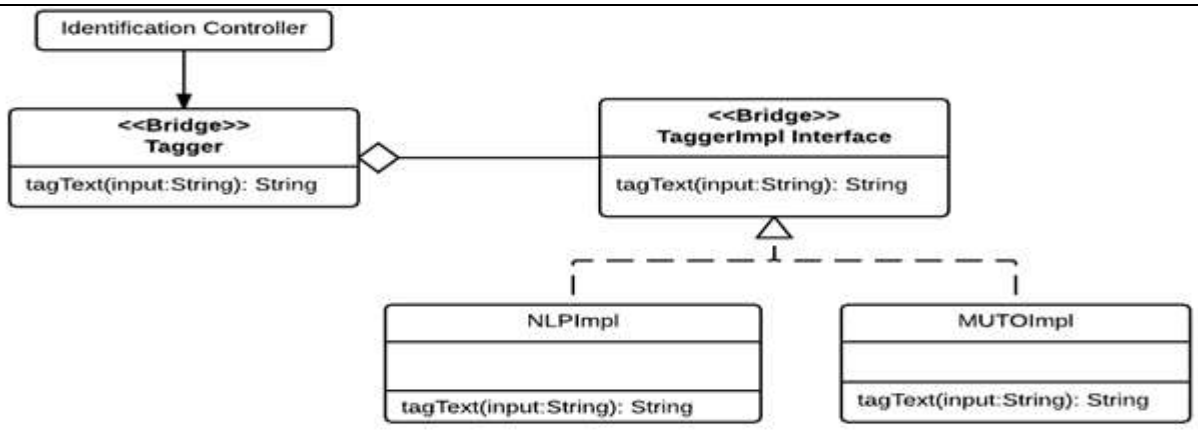
3. Concrete Implementation 1 Class:

```
public class NLPImpl implements TaggerImplInterface {
                        public String tagText(String input) throws ClassNotFoundException, IOException {
                                MaxentTagger tagger = new MaxentTagger("english-left3words-distsim.tagger");
                                        String taggedText = tagger.tagString(input);       return taggedText;       }
        }
```

4. Concrete Implementation 2 Class: (Modular Unified Taggig Ontology)

```
public class MUTOImpl implements TaggerImplInterface {
                public String tagText(String input) throws ClassNotFoundException, IOException {
                                        }         }
```

Class Diagram



Design Problem (Memento Pattern)

The domain model tool requests to save its entire state as a project, and retreive the project at a later time as and when required (Usually when application is re-open). This feature is needed for a smooth checkpoint/rollback mechanism in the tool so that the user can save various project he worked and refer them at a later time.

Pattern Identified & Why

Memento: The **Memento Pattern** provides the ability to restore an object to its previous saved state. This can be used to keep the state of current application always ready and available to use when needed. Memento is "opaque" to the tool and all other objects. If the tool needs to rollback the source object's state, it hands the Memento back to the source object for reinstatement. The source object initializes the Memento with a characterization of its state. The tool is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento also called the “originator”.

Benefits

User can easily save and restore its state whenever required. Reduces the application logic for the controller to take care of it. Reduces coupling for the controller by making it independent from storing and retreiving state of the application.

High Level Idea on how to Implement Memento on our Tool (High Level Code Snippet)

1. GUI Class (Originator)

```
/* New Code*/
public Memento createMemento()      {       ...       }
public void setMemento(Memento m)   {       ...       }

/* Existing Code, to be replaced*/
public void windowClosing(WindowEvent e) {  ...       saveState();     ...       }

private void saveState() {          ...
      pw.write(EditorTextPane.getText());
      DM_Memento.save(dmConcepts, "tmp/conceptlist.dat");                    ...   }
private Gui() {   ...                              //Constructor to load state
ArrayList concepts=(ArrayList) DM_Memento.load(new ArrayList(),
       "tmp/conceptlist.dat");                     ...       }
```

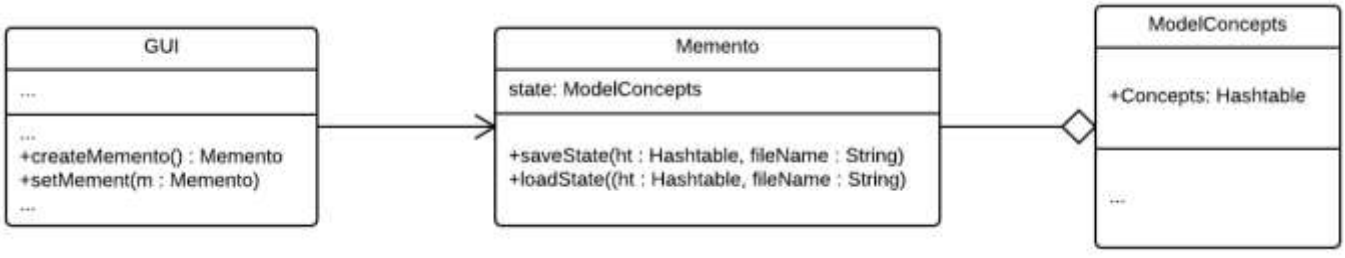
2. DM_Memento Class (Memento)

```
public static void saveState(Hashtable ht, String fileName)
public static void loadState(Hashtable ht, String fileName)
```

3. ModelConcept (Caretaker)

```
      /* Default Methods to create and store ModelConcepts */
```

Class Diagram



Design Problem (Controller Pattern)
How to delegate the request from the UI layer objects to domain layer objects. How to reduce the load over the UI to handle processing task?

Patterns Identified and Reason
Controller: When a request comes from UI layer object, **Controller** pattern helps us in determining what is that first object that receive the message from the UI layer objects. This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.

Benefits
Controller helps to take away all the system events processing or any internal system level processing from the UI and do it from the controller. Controllers can also be re used and helps control the sequence of activities.

In our project we use various controllers (Each controller doing handling a specific system event)
Some examples are (Although we explain just one in the code snippet and digram section):
IdentificationController, ClassificationController, ClassAttributeTableController, ClassRelationshipTableController and so on.

High Level Idea on how to Implement Controller on our Tool (High Level Code Snippet)

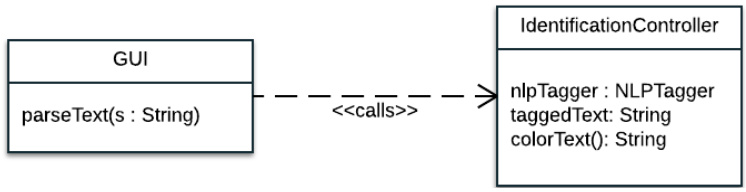
1. GUI Class

```
private void identifyText() throws ClassNotFoundException, IOException {
        String input = getEditorTextPane().getText().toLowerCase();
        IdentificationController it = new IdentificationController();
        it.parseText(input);            }              //Task Performed
```

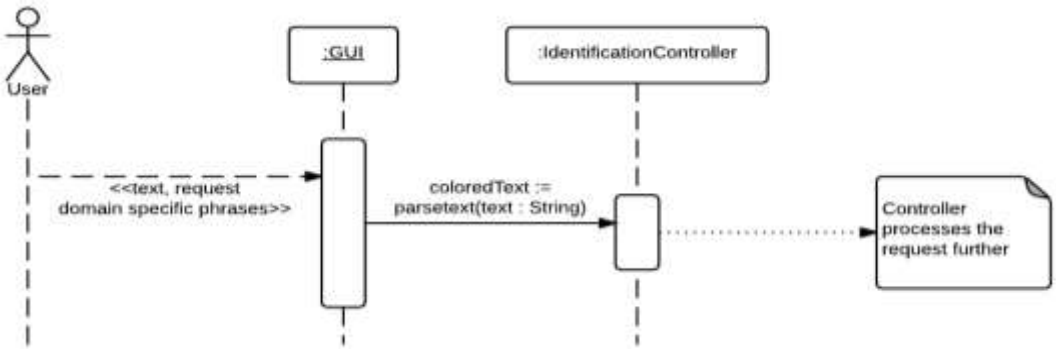
2. IdentificationController Class

```
public String parseText(String input) throws ClassNotFoundException, IOException {
                NLPExpert nlpTagger = new NLPExpert();
                String taggedText = nlpTagger.tagText(input);
                colorText(taggedText);
            return taggedText;              }
```

Class Diagram



Sequence Diagram



Design Problem (Information Expert as well as Creator Pattern)

1. Given; we have a task that needs to be performed. The task's entire responsibility can be done by a entity since it has all the information and resources for performing it. Should we assign this entity to a controller or a handler? Or make a new class for that particular entity and use that whenever required?
2. Also, who should be responsible for assigning this responsibility or performing the task and how?

Patterns Identified and Reason

1. **Expert**: For the first design problem, we identify the **Expert Pattern**; since the Expert Pattern principle says, assign those responsibilities to object for which object has the information to fulfill that responsibility.
2. **Creator**: In this case, we use **Creator Pattern**; the controller object creates the expert pattern to use the services offered by the Expert.

Benefits

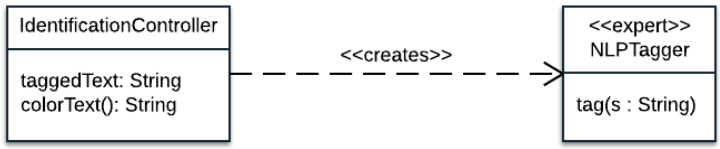
1. This reduces the coupling on the controller object, since it becomes independent of the task performed by the expert. Whenever the expert object changes, it has no effect on the controller and vice versa.
2. Based on the objects association and their interaction, we can create a object exactly when it is required and terminate after use.

High Level Idea on how to Implement Expert and Creator on our Tool (High Level Code Snippet)

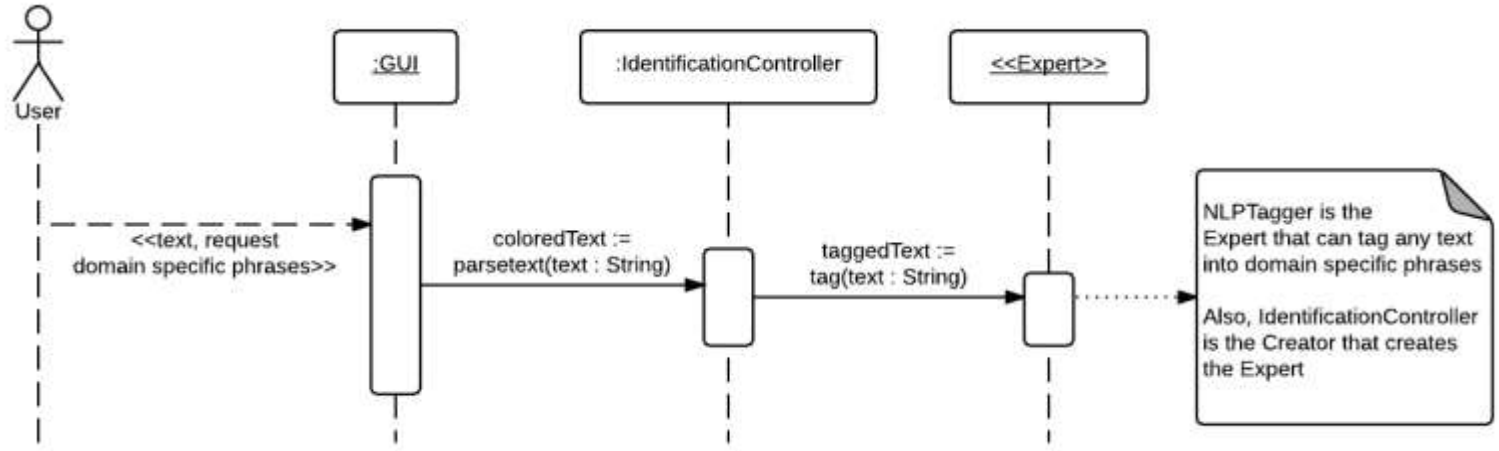
1. IdentificationController Class

```
public class IdentificationController {
        public String parseText(String input) throws ClassNotFoundException, IOException {
                NLPExpert nlpTagger = new NLPExpert();       //Creator(IdentificationController) creating the Expert
                String taggedText = nlpTagger.tagText(input);
                colorText(taggedText);
              return taggedText;          }               }
```

Class Diagram



Sequence Diagram



Design Problem (Strategy Pattern)

We are providing two layouts for domain model diagram, Horizontal and Vertical. How client will select and execute one of the available layouts.

Pattern Identified and Reason

Strategy: We are creating one abstract Layout class to provide common interface to make the layouts interchangeable. Each subclass, Horizontal or Vertical implement an available Layout. The client can now select any of the Layout and delegate the work to the specific subclass.

Benefits

It easily removed conditional statements to select the layouts. It made software easy to understand, implement and maintain. We can simply add or remove layouts as per need.

Liabilities

Client needs to select which layout he/she wants.

High Level Idea on how to Implement Strategy on our Tool (High Level Code Snippet)

1. Context Class

```
Public class DomainModelDiagram{      Data data;                  …
        Public Data getDomainModelDiagramData() {return data;}              }
```

2. Strategy Class

```
Public abstract class Layout{      DomainModel Diagram DMG;
        Public void setDomainModelDaigram(DomainModelDiagram  D)        {        DMG = D;        }
        Public abstract void apply();              }
```

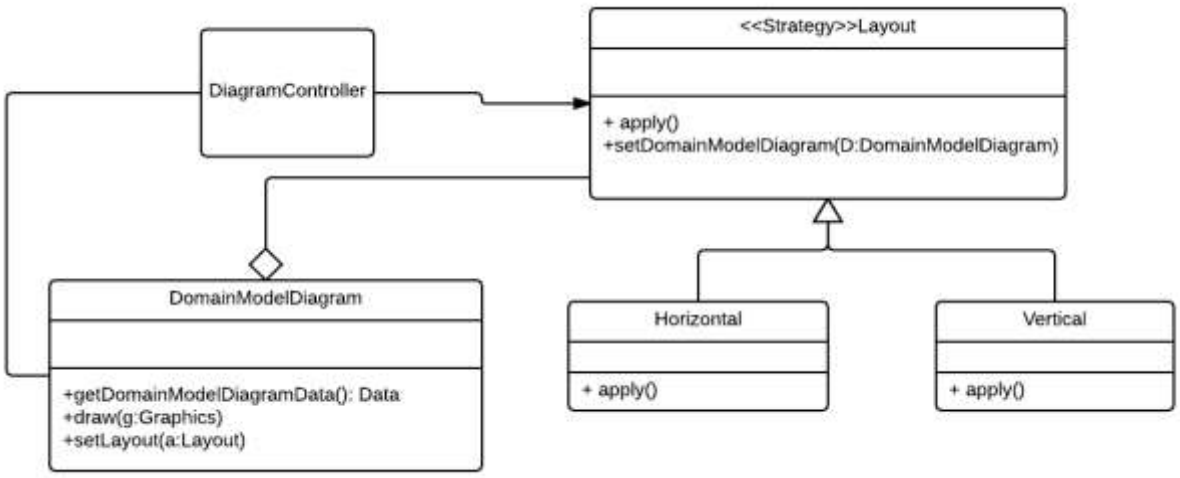
3. StrategyK Class

```
Public class Horizontal extends Layout{          Public void apply()              {
                //It will display DMG in horizontal Layout          }        }
Public class Vertical extends Layout{            Public void apply()      {
                //It will display DMG in vertical Layout            }        }
```

4. Client Class

```
Public class DiagramController{          Layout L;        Public void use()        {
                DomainModelDiagram DMG = new DomainModelDiagram();      Layout = new Horizontal();
                Layout.setDomainModelDiagram(DMG);          Layout.apply();          }        }
```

Class diagram:



Design Problem (Composite Pattern)

Domain Model Diagram is complex class structure consisting of class, attributes and different types of relations. All these objects need to be stored in some data structure and adding or removing in a later stage should be easy. All the class, attributes and relationship class does the work of writing into file before generating a diagram. So the general approach is to check for if condition and write implementation for each different operation. But adding some more components in a later stage will be problematic as more if conditions has to be added and it becomes very tight coupling.

Pattern Identified and Reason

Composite: It will give uniform interface to access all domain model diagram elements and to extend the editor to support other types of uml diagrams. It also helps in traversing complex structures of domain model diagram. It will eliminate if conditions and writing operation in different classes will be called dynamically.

Benefits

In the future if more components need to be added or composite component i.e. different domain model diagram, then it becomes very easy to add those classes in this pattern. Client can treat all of these components i.e. classes, attributes and relationship uniformly as there is single interface for accessing these components.

Liabilities

If the attribute component is deleted or removed from the pattern then client should also remove all of its relationship to class component. Client has to take are of those integrity constraint.

High Level Idea on how to Implement Composite on our Tool (High Level Code Snippet)

1. Component Class

```
public abstract class ModelConcept{
  public abstract void writeFile();          }
```

2. Composite Component Class

```
public class DomainModel extends {
   ArrayList concepts=new ArrayList<ModelConcept>();
        public void add(ModelConcept concept) {
                if (!concepts.contains(concept)) {  concepts.add(concept);     }  }
        public void remove(ModelConcept concept) {
                concepts.remove(concept);      concepts.trimToSize();   }
        public void writeFile() {          classes=new StringBuilder();      attributes=new StringBuilder();
                relationships=new StringBuilder();                  for (int i=0; i<concepts.size(); i++) {
                        ((ModelConcept)concepts.get(i)).writeFile();     }          }          }
```

3. Primitive Component Class

```
public class ClassObj extends ModelConcept{
        public void writeFile() {                  DomainModel domainModel=DomainModel.getInstance();
                String name=getName();                  Iterator it_attr=getAttributes().iterator();
                while (it_attr.hasNext()) {        String aName=((Attribute)it_attr.next()).getName();
                        if (aName==null) continue;
                                domainModel.attributes.append("\""+aName+"\"\n");     }  }    }
```

4. Client

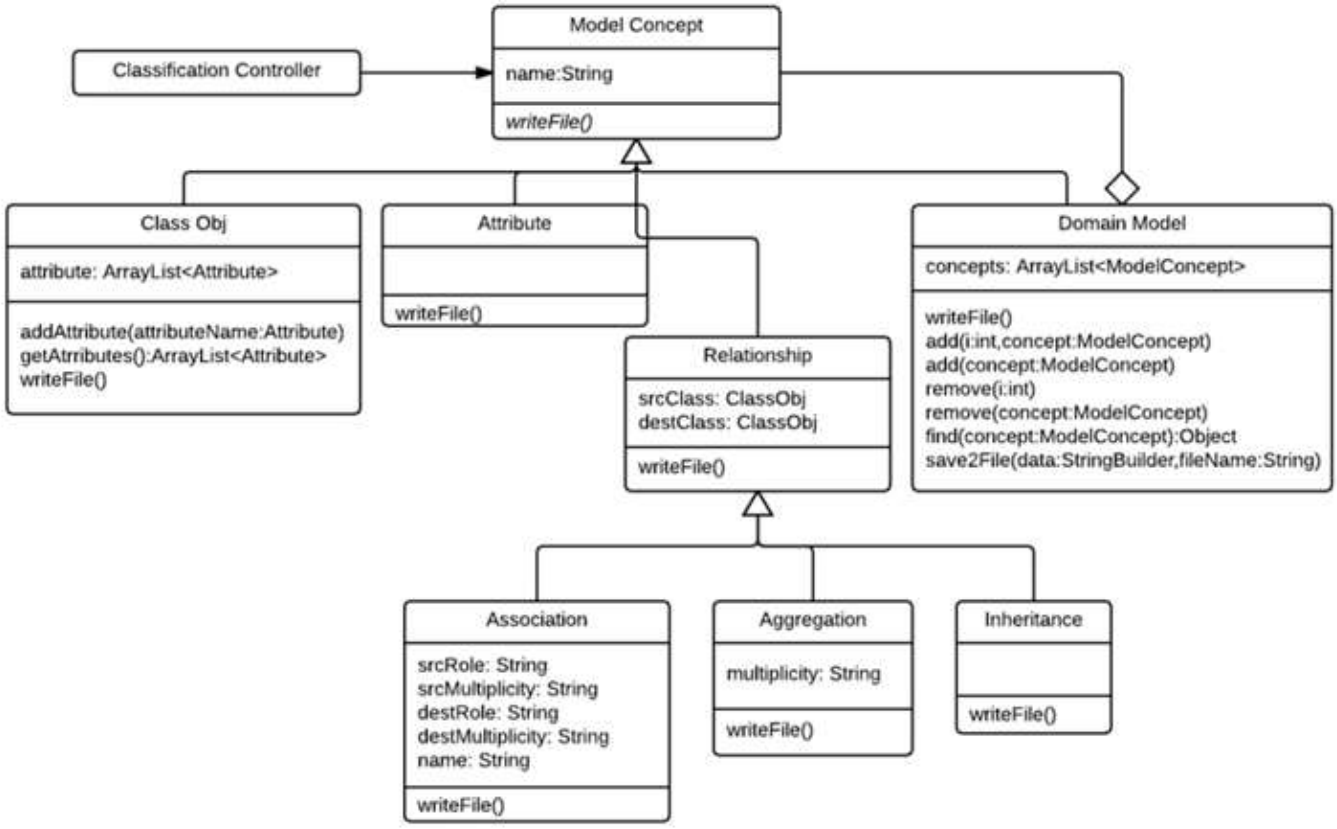
```
public class ClassificationController {              public void DragToClassAttributeTable(JTable table) {
                                table.setTransferHandler(new ClassTableTransferHandler());      }          }
class ClassTableTransferHandler extends StringTransferHandler {
   protected void importString(JComponent c, String str) {
        AddClass addclass = new AddClass(indivValue[i]);          }      }  }
```

```
public class AddClass extends AbstractAction{              public void add(int row, int column) {
                                ClassOkAction ok = new ClassOkAction(classDialog);              classDialog.display();     }
        }

public class ClassOkAction extends OkAction {              public void actionPerformed(ActionEvent e) {
                DomainModel domainModel=DomainModel.getInstance();          domainModel.add(classObj);   }    }

public class GenerateListener implements ActionListener {          public void actionPerformed(ActionEvent e) {
                DomainModel.getInstance().writeFile();     }       }
```

Class Diagram



`

Design Problem (Singleton Pattern)

User is only allowed generates one domain model diagram. If user generates many diagram, user may not know which one is correct.When user edits the diagram, the update will store in the domain concept. If there are many diagrams, we won't know which need be stored.

Pattern Identified and Reason

Singleton: The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.

Benefits

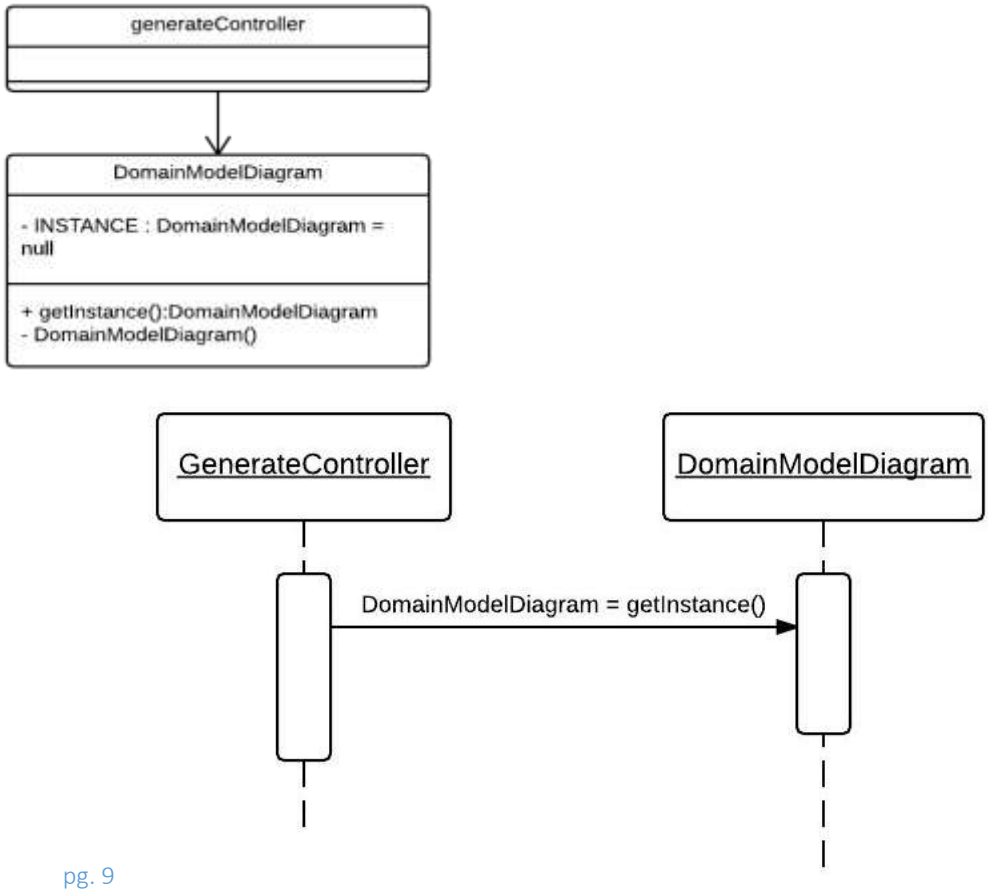
Singleton pattern will prevent other objects instantiated copies of their own singleton object, ensuring that all objects are accessed only instance._Because class controls the instantiation process, so classes can flexibly change the instantiation process.

High Level Idea on how to Implement Singleton on our Tool (High Level Code Snippet)

1. DomainDiagram Class:

```
public class DomainDiagram {       private final static DomainDiagram INSTANCE = new DomainDiagram();
   private DomainDiagram() {}       public static DomainDiagram DomainDiagram() {
      if (INSTANCE == null) {          synchronized (DomainDiagramo.class) {
            if (INSTANCE == null) {                INSTANCE = new DomainDiagram();            }         }      }
       return INSTANCE;      }       }
```

Class Diagram



Design Problem (State Pattern)

User highlights or unhighlights the words by clicks it. But the result depends on the state before user's action. When user clicks the words, software needs check the status every time. If user needs more operate and state about the text, software needs to add more if-else or switch-case to check it.

Pattern Identified and Reason

State: Since we highlight and unhighlight words in the GUI, they can be considered as states. Each mouse event on these corresponds to identifying the current state and take action accordingly. These reduces the overhead on the GUI/Controller class instead making the context and state class handle the action.

Benefits

The software will not check the state every time rather the state is always stored and can be used. And we can easily to add new states.

Liabilities

Each state derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

High Level Idea on how to Implement State on our Tool (High Level Code Snippet)

1. TextController Class:

```
public class TextController{        private WordState currentState ;          public void click(Point p){
                currentState = getState(p);                ccurrentState = currentState.click();       }          }
```

2. AbstractState (WordState) Class:

```
public abstract class WordState(){          public WordState click(){          return this;       }          }
```

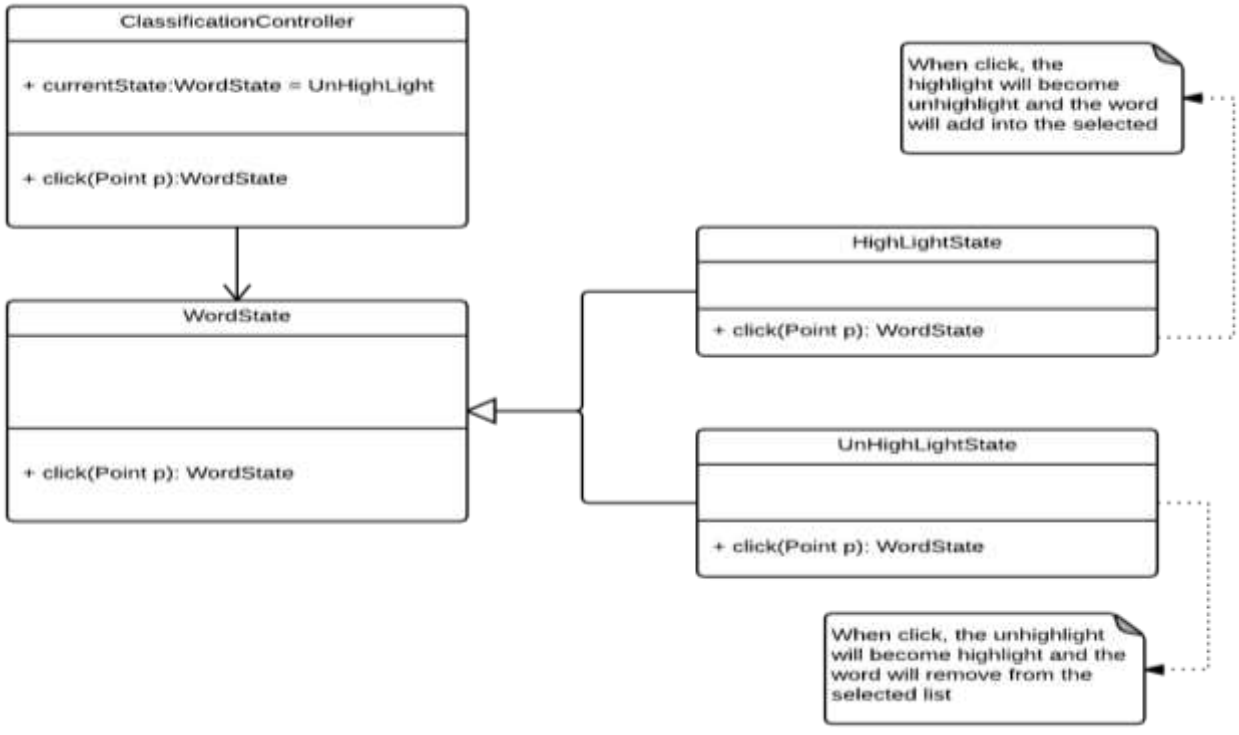
3. HighLight State Class:

```
public class HighLight extends WordState{          public void click(Point p){          Gui.highLightInstance(p);
                return UnHighLight;                }          }
```

4. UnHighLight State Class:

```
public class UnHighLight extends WordState{      public WordState click(Point p){          Gui.unHighLightInstance(p);
        return Highlighte;                }          }
```

Class Diagram



Requirements

How to Add Class to the ClassAttribute Table (JTable)? -> There are two ways to add Class:

1. By dragging from Input TextArea to ClassAttribute Table after selecting single or multiple words. Here the underlying data structure is HashMap
2. By dragging from ClassRelationship Table to the ClassAttribute Table after selecting single or multiple rows. Here the underlying data structure is ArrayList.

(Our primary GUI will contain 3 major fields, TextArea for Input, JTable 1 for ClassAttribute details, JTable 2 for ClassRelationship details)

Design Problems

How would you process the different data structure in order to add class(es) to the table? Different methods should be used to process different data structures even though both are doing the same action (to add class).

Pattern Identified and Why Reason

Iterator: The Iterator Pattern provides the ability to use a common interface to process a set of data irrespective of underlying data structure. Here both the data structures have a common agenda to add classes to the table. Iterator method helps us to form a concrete iterator from different data structures (HashMap and ArrayList), and a common method (AddClass) can be used with hasNext() and next() to process the set of data and add a class in Jtable for each data.

(Note: The same design problem arises in Add Relationship also, but in this example we only talked about Add Class)

Benefits

Iterator helps to reduce the work load of creating multiple methods to process data sets. It also helps to create a clean and non-repetitive code structure.

High Level idea on how to implement Iterator Pattern in our Tool: (High Level Code Snippets)

1. HashMap Structure

```
Map<String, String> newClass = new HashMap<String, String>();        …… /*Add Some data to the HashMap */ …….
Iterator iterator = newClass.iterator(); ……. // More Code Here       processDataSet(iterator);
```

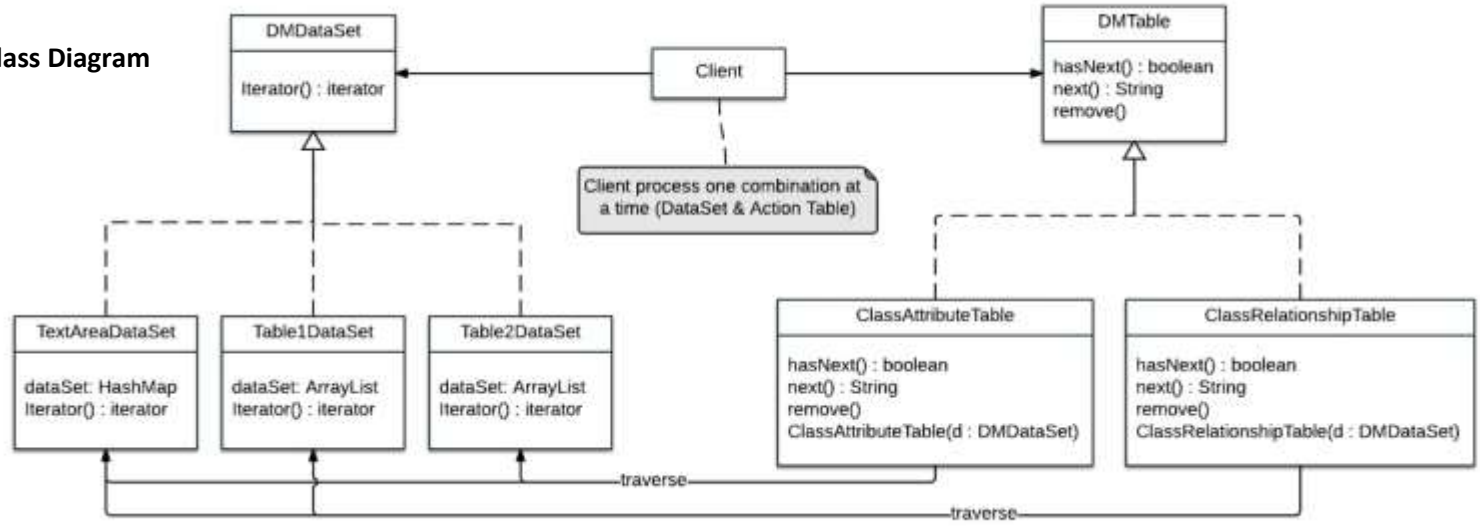
2. ArrayList Structure

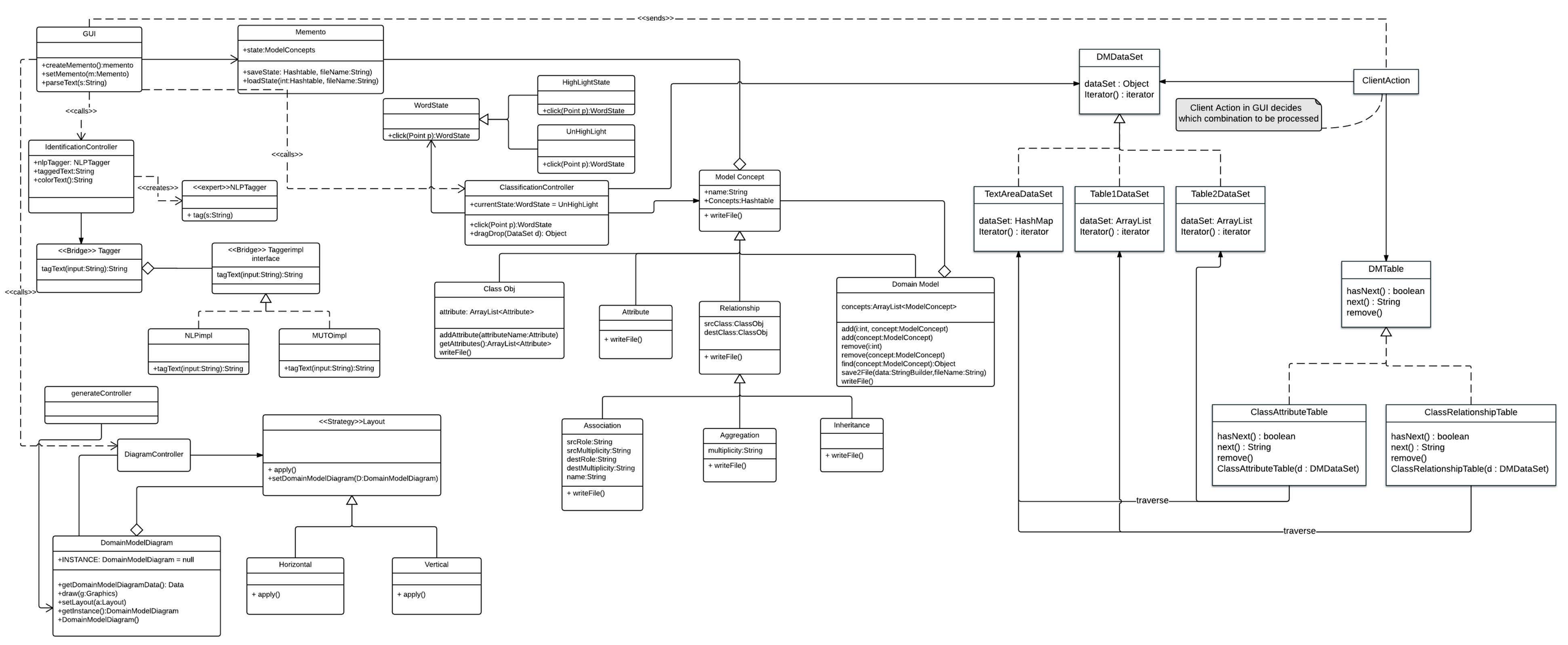
```
ArrayList<String> newClass = new ArrayList<String>();          …… /*Add Some data to the ArrayList */ …….
Iterator iterator = newClass.iterator(); ……. // More Code Here          processDataSet(iterator);
```

3. AddClass

```
public void processDataSet(Iterator iterator) {
        while(iterator.hasNext()) {      …….          AddClass(iterator.next());
            …….  }                  }
```

Class Diagram




GUI
+createMemento():memento
+setMemento(m:Memento)
+parseText(s:String)
Memento
+state:ModelConcepts
+saveState: Hashtable, fileName:String)
+loadState(int:Hashtable, fileName:String)
<<sends>>
<<calls>>
<<calls>>
<<calls>>
IdentificationController
+nlpTagger: NLPTagger
+taggedText:String
+colorText():String
<<creates>>
<<expert>>NLPTagger
+ tag(s:String)
WordState
+click(Point p):WordState
HighLightState
+click(Point p):WordState
UnHighLight
+click(Point p):WordState
ClassificationController
+currentState:WordState = UnHighLight
+click(Point p):WordState
+dragDrop(DataSet d): Object
Model Concept
+name:String
+Concepts:Hashtable
+ writeFile()
DMDataSet
dataSet : Object
Iterator() : iterator
ClientAction
Client Action in GUI decides which combination to be processed
TextAreaDataSet
dataSet: HashMap
Iterator() : iterator
Table1DataSet
dataSet: ArrayList
Iterator() : iterator
Table2DataSet
dataSet: ArrayList
Iterator() : iterator
DMTable
hasNext() : boolean
next() : String
remove()
<<Bridge>> Tagger
tagText(input:String):String
<<Bridge>> Taggerimpl interface
tagText(input:String):String
NLPimpl
+tagText(input:String):String
MUTOimpl
+tagText(input:String):String
Class Obj
attribute: ArrayList<Attribute>
addAttribute(attributeName:Attribute)
getAttributes():ArrayList<Attribute>
writeFile()
Attribute
+ writeFile()
Relationship
srcClass:ClassObj
destClass:ClassObj
+ writeFile()
Domain Model
concepts:ArrayList<ModelConcept>
add(i:int, concept:ModelConcept)
add(concept:ModelConcept)
remove(i:int)
remove(concept:ModelConcept)
find(concept:ModelConcept):Object
save2File(data:StringBuilder,fileName:String)
writeFile()
Association
srcRole:String
srcMultiplicity:String
destRole:String
destMultiplicity:String
name:String
+ writeFile()
Aggregation
multiplicity:String
+ writeFile()
Inheritance
+ writeFile()
ClassAttributeTable
hasNext() : boolean
next() : String
remove()
ClassAttributeTable(d : DMDataSet)
ClassRelationshipTable
hasNext() : boolean
next() : String
remove()
ClassRelationshipTable(d : DMDataSet)
traverse
traverse
generateController
DiagramController
<<Strategy>>Layout
+ apply()
+setDomainModelDiagram(D:DomainModelDiagram)
DomainModelDiagram
+INSTANCE: DomainModelDiagram = null
+getDomainModelDiagramData(): Data
+draw(g:Graphics)
+setLayout(a:Layout)
+getInstance():DomainModelDiagram
+DomainModelDiagram()
Horizontal
+ apply()
Vertical
+ apply()