

# 18653 project report

Team 14

Run Yang  
Huanwen Chen  
Varun Vijayraghavan  
Ge Jin

## 1. Introduction

In this project, we would like to apply multiple design pattern to optimize a social network and forum that scientists collaborate and communicate on scientific workflows or scientific experiments. In this platform, a scientist publishes and shares her experiments, and other scientists may either communicate or comment on the entire workflow. In other words, the project provides a platform for scientists to post their scientific experiments and workflows, so that other scientists who are interested in the workflow can redo the experiments and make comments.

In this whole project, we mainly focus on implementing design pattern on workflow system and user system. Based on the former work, we make a more specific and powerful system. Until now, we have optimized several functions, like: user login and signup functionality, user profile, user group functionality, user access control, subscription functionality, workflow list, adding and deleting workflow, commenting, marking answers and so on.

The project can be divided into two parts - the frontend and the backend.

We made modification on both parts. We have implemented nine different design patterns on it - Iterator Pattern, Builder Pattern, Chains of Responsibility Pattern, Flight Weight Pattern, Factory Pattern, Adapter Pattern, Decorator Pattern, Criteria Pattern, State Pattern. In the whole process of this project, we also use many technique support to help the project work better. We use Docker as a virtualization platform to run the project. In this project, we use MVC framework to connect each part. And more specific, we use Hibernate for mapping all the object-oriented domain models to database in the backend. And we use scala language to change programs in the front end.

## 2. Motivation

In the whole process of doing a software project, design is at the most essential and upmost place. There are two directions of design, architectural styles and design patterns. Architectural styles will decide a macro direction, while design patterns will decide a micro direction. In our project, we mainly deal with how to apply different design patterns to a workflow project.

When a designer wants to do a project, the first thing must be design. Good designers do not solve every problem from first principles. They reuse solutions. However, practioners do not do a good job of recording experience in software design for others to use. Patterns help solve the problem. And there are benefits of using design patterns. Patterns are a common design vocabulary that allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation. What's more, patterns capture design expertise and allow that

expertise to be communicated, which promotes design reuse and avoid mistakes. It will also improve documentation and understandability.

Thus, design pattern is an essential part of design and we will make a practical use and deeper understanding of it through this project.

### **3. Related work**

This project is basically related to the Apache project - Apache Climate Model Diagnostic Analyzer (CMDA), which provides web services for multi-aspect physics-based and phenomenon-oriented climate model performance evaluation and diagnosis through the comprehensive and synergistic use of multiple observational data, reanalysis data, and model outputs.

Since CMDA is an incubation project, we develop our project based on what have done in CMDA, indicating that we optimized the workflows platform for the scientists of CMDA into their workflow discussion.

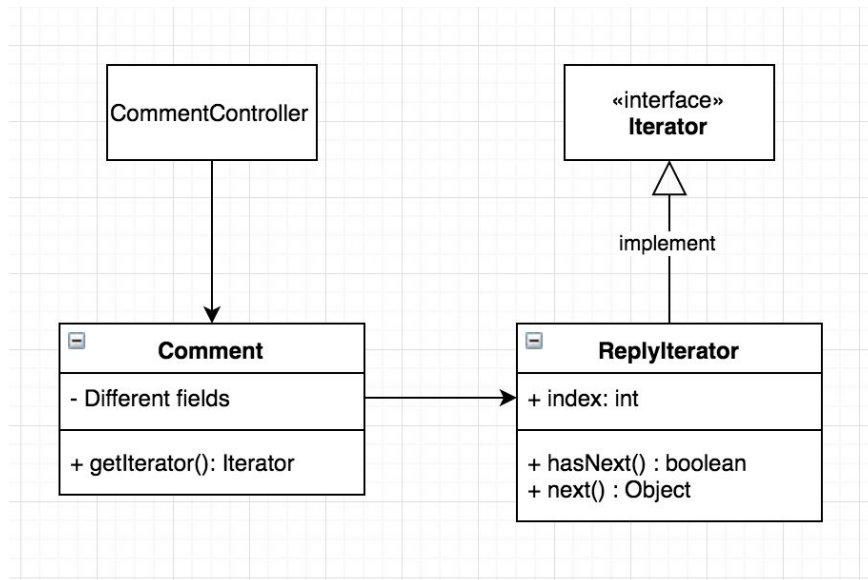
## **4. System design and implementation**

### **Iterator Pattern**

Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers.

Here in our back-end code, we used Iterator pattern to help traverse all the replies in Comment class. First of all, I defined an ReplyIterator class in Comment class which implements Iterator interface. I rewrote the hasNext() and next() method to help us go through all replies in reply list. In comment controller, when we need to get all the replies in reply list, we just use ReplyIterator to do that thing.

The UML graph is shown as below:

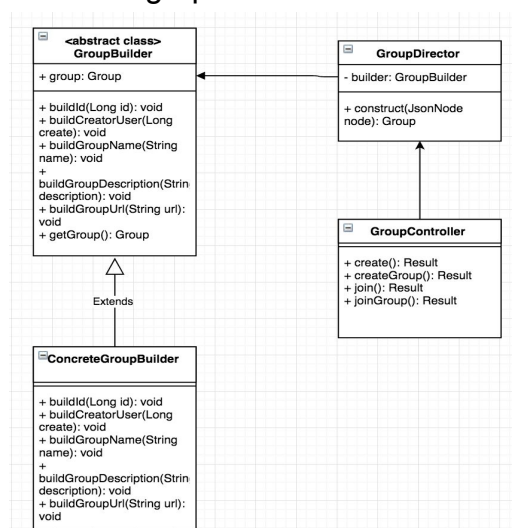


## Builder Pattern

Builder Pattern is a design pattern used to build complex pattern by using simple objects and using step by step approach. A builder class is needed in this pattern. This builder class builds the final object step by step. And this builder is independent of other objects.

In our implementation, we used builder pattern to help create new group. Group is divided into 5 different parts - id, creatoruser, groupname, groupdescription, groupurl. Abstract class GroupBuilder contains methods to create those fields step by step. Concrete is the class to extend GroupBuilder and implement abstract methods in detail. We also made a GroupDirector to manage the use of GroupBuilder. Whenever a new Group is needed to be created, I just call GroupDirector to make a new Group.

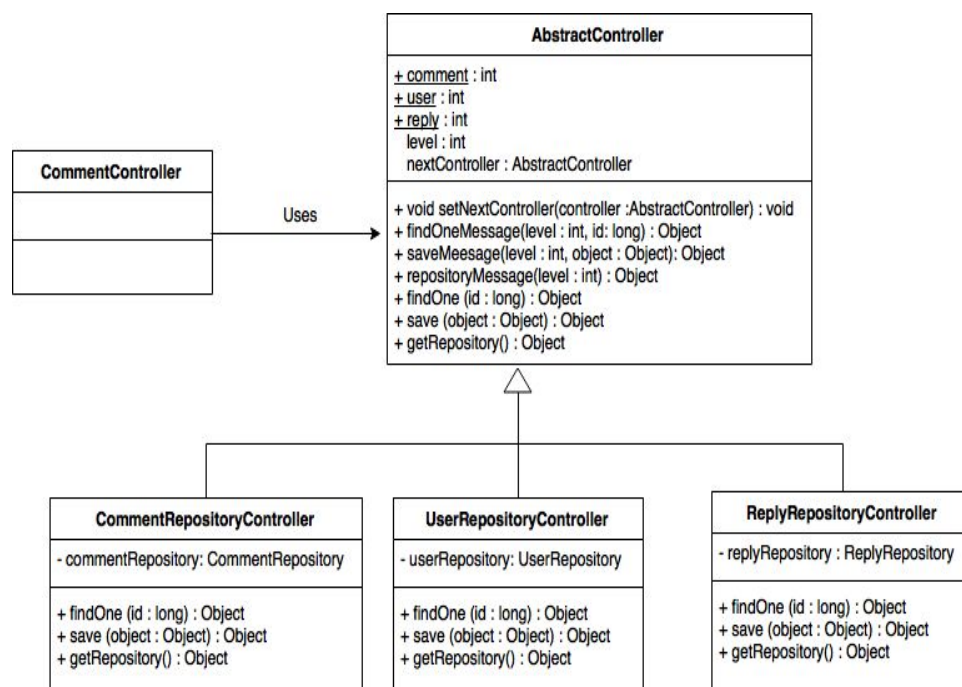
The UML graph is shown as below:



## Chains of Responsibility Pattern

Chains Of Responsibility is used to avoid coupling together between request sender and receiver, enable multiple objects have possibility to receive request. These object will be connect into a chain and pass the request along the chain until an object handles the request.

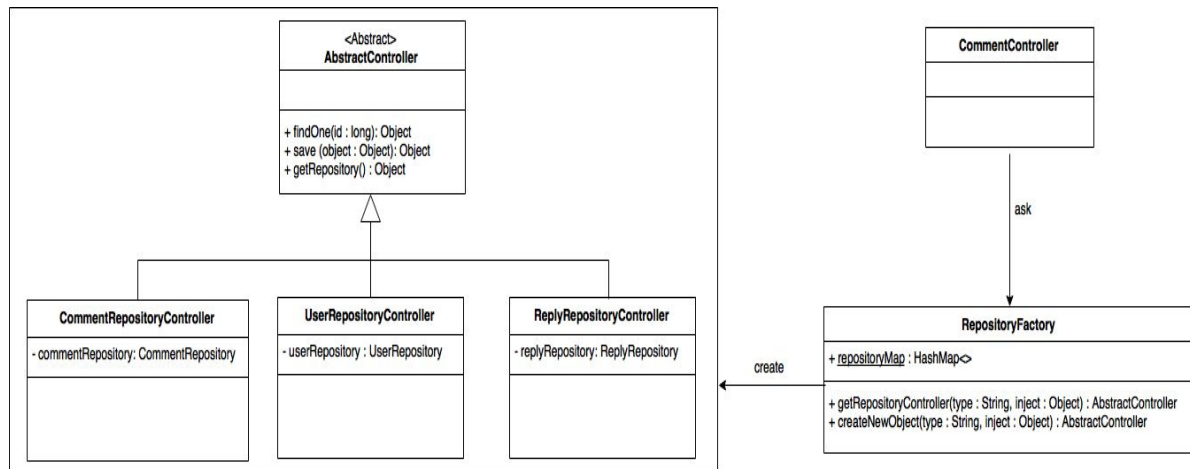
In our implementation, different repositories are different object. Thus I connected each repositories us into a chain, and handle multiple requests.



## Flight Weight Pattern

Flyweight Pattern is mainly used to reduce the number of object created so as to reduce memory usage and improve performance.

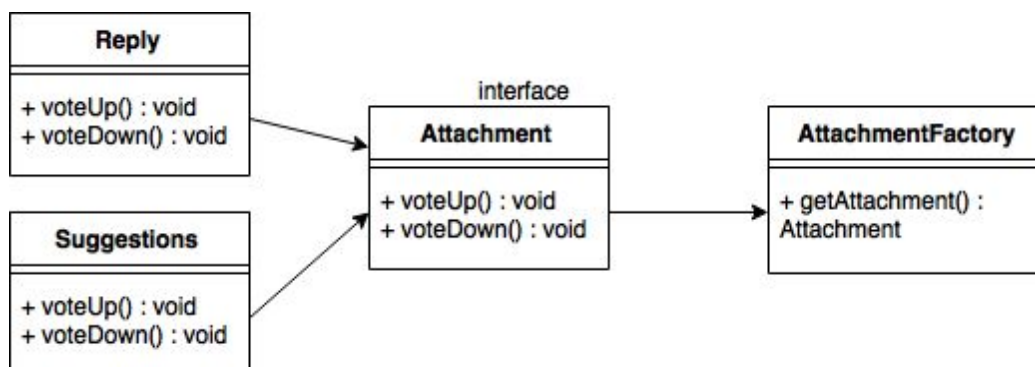
According to our design, we need to used the repository objects in different controller classes frequently. Therefore, we can use Flyweight pattern with static hashmap to store different repository object.



## Factory Pattern

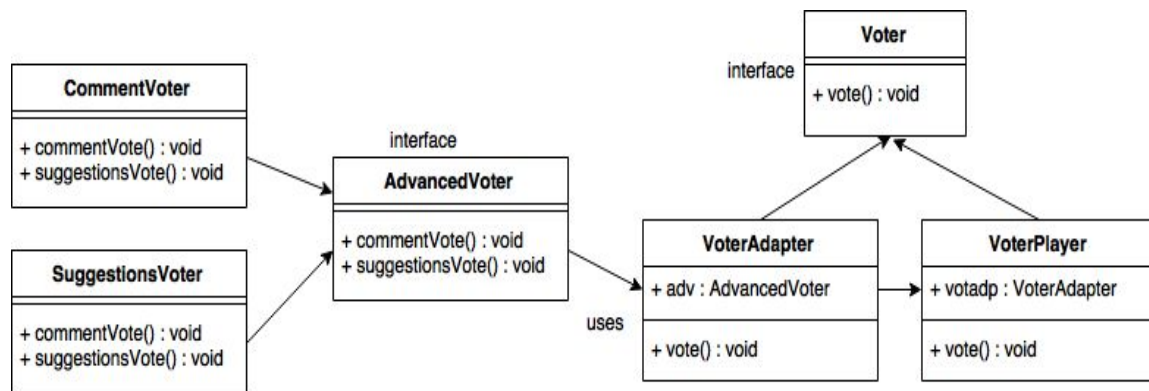
Factory pattern is one of the creational patterns that can be used to this project. It's a design pattern for creating object using a common interface and without exposing the creation logic to the client.

We create Attachment interface containing voting functions, and making it implemented by Reply and Suggestions concrete classes. Then we create AttachmentFactory class to generate objects of the concrete classes based on parameters we send. Then when we want to create Reply or Suggestions object, we can directly use the factory and send different parameters.



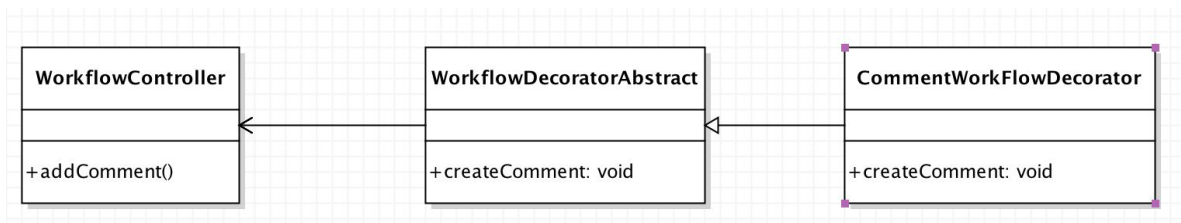
## Adapter Pattern

Adapter pattern is one of the structural patterns used as a bridge between two incompatible interfaces. So we create Voter and AdvancedVoter interface to join functionalities, making them implemented by CommentVoter and SuggestionsVoter concrete classes. Then we create VoterAdapter class to implement Voter interface. And at last we create VoterPlayer class to implement AdvancedVoter and receiving different parameters.



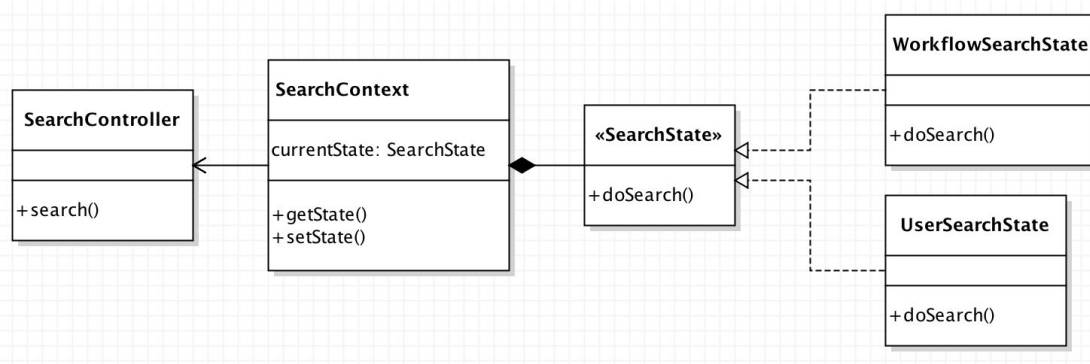
## Decorator Pattern

The decorator pattern can be used to enhance the functionality of an existing object. Here, we use the decorator pattern to enhance the Workflow object when a Comment is added to it. The WorkflowDecorator Abstract Class defines the functionality of a workflow decorator, with the specific implementation in the concrete class CommentWorkFlowDecorator. This, when a comment is added in the WorkflowController class, the decorator pattern takes care of it.



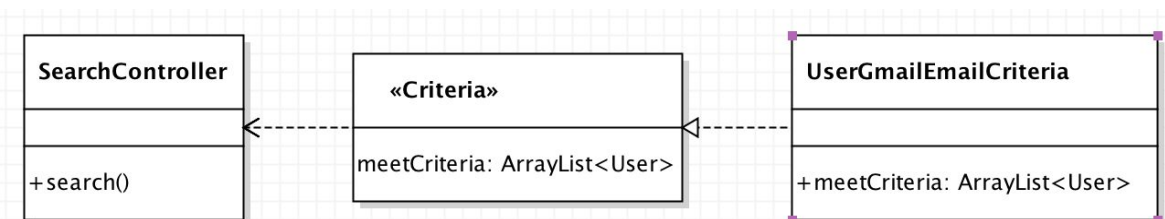
## State Pattern

The state design pattern can be used to control the flow of an application based on its current state. In this application, we use it to control the Search functionality based on different states: User search or Workflow Search. The SearchContext maintains the current state, with the State and operations that can be performed in a state described by the SearchState interface and WorkflowSearchState and UserSearchState concrete classes. The search context is set and specific search performed in the SearchController.



## Criteria Pattern

The criteria design pattern can be used to filter a set of objects based on a particular criteria. Here, we use it to filter the search and only display GMail users. This is done by specifying a Criteria interface to define the function that filters a list of users, and the `UserGmailEmailCriteria` concrete class to implement the function that filters GMail users.



## 5. Experiments and analysis

### Iterator Pattern

A specific `ReplyIterator` is designed and implemented in `Comment` class. `hasNext()` and `next()` method are implemented in the `Iterator`. We can see that in `commentcontroller`, this `Iterator` is used to traverse all replies in the reply list. When we add comments and show all comments, the application run normally, which proves that our implementation is valid.

Some of the implement code sections are listed as below :



```

@Override
public Iterator getIterator() { return new ReplyIterator(); }

private class ReplyIterator implements Iterator {
    int index = 0;

    @Override
    public boolean hasNext() {
        if (index < replies.size()) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return replies.get(index++);
        } else {
            return null;
        }
    }
}

//
//      Reply reply = new Reply(fromUser, toUser, timestamp, content);
AttachmentFactory af= new AttachmentFactory();
Reply reply= (Reply)af.getAttachment("reply");
reply.setFromUser(fromUser);
reply.setToUser(toUser);
reply.setTimestamp(timestamp);
reply.setContent(content);
Reply savedReply = (Reply)replyRepositoryController.save(reply);
//
//      List<Reply> replyList = comment.getReplies();
List<Reply> replyList = new ArrayList();
Iterator iter = comment.getIterator();
while (iter.hasNext()) {
    replyList.add((Reply)iter.next());
}
replyList.add(reply);
comment.setReplies(replyList);
commentRepositoryController.save((Object)comment);

return ok(new Gson().toJson(savedReply.getId()));

```

## Builder Pattern

A specific Builder class is used to create the group for user. The code segments are listed as below:

```

import models.Group;
import com.fasterxml.jackson.databind.JsonNode;

public class GroupDirector {
    private GroupBuilder builder;

    public GroupDirector(GroupBuilder builder) { this.builder = builder; }

    public Group construct(JsonNode node) {
        if (node.get("id")!=null)
            builder.buildId(node.get("id").asLong());
        if (node.get("creatorUser")!=null)
            builder.buildCreatorUser(node.get("creatorUser").asLong());
        if (node.get("groupName")!=null)
            builder.buildGroupName(node.get("groupName").asText());
        if (node.get("groupUrl")!=null)
            builder.buildGroupUrl(node.get("groupUrl").asText());
        if (node.get("groupDescription")!=null)
            builder.buildGroupDescription(node.get("groupDescription").asText());

        return builder.getGroup();
    }
}

```

Unused import statement

```

import models.Group;
import com.fasterxml.jackson.databind.JsonNode;

public class GroupDirector {
    private GroupBuilder builder;

    public GroupDirector(GroupBuilder builder) { this.builder = builder; }


    public Group construct(JsonNode node) {
        if (node.get("id") != null)
            builder.buildId(node.get("id").asLong());
        if (node.get("creatorUser") != null)
            builder.buildCreatorUser(node.get("creatorUser").asLong());
        if (node.get("groupName") != null)
            builder.buildGroupName(node.get("groupName").asText());
        if (node.get("groupUrl") != null)
            builder.buildGroupUrl(node.get("groupUrl").asText());
        if (node.get("groupDescription") != null)
            builder.buildGroupDescription(node.get("groupDescription").asText());

        return builder.getGroup();
    }
}

```

Unused import statement

Also, experiments are done to test if group creation function is normal. From the test result, we can see that our implementation is successful.


**Jet Propulsion Laboratory**  
 California Institute of Technology

[Home](#)
[Search](#)
[Workflow](#)
[Timeline](#)
[Notification](#)
[Forum](#)

Congratulations, the pass for the group is 2abb7b4b-6356-484a-ad5b-7c2032f1cd5d !

### Create new group

## Your Groups

Group Name	Pass code
mynewgroup	eebc504b-830b-4f7d-a0f6-43d763fad128
mynewgroup2	2abb7b4b-6356-484a-ad5b-7c2032f1cd5d

## Chains of Responsibility Pattern

```

public abstract class AbstractController {
    public static int comment = 1;
    public static int user = 2;
    public static int reply = 3;

    protected int level;

    protected AbstractController nextController;

    public void setNextController(AbstractController nextController) {
        this.nextController = nextController;
    }

    public Object findOneMessage(int level, long id) {
        if (this.level <= level) {
            return findOne(id);
        } else {
            return nextController.findOneMessage(level, id);
        }
    }

    public Object saveMessage(int level, Object object) {
        if (this.level <= level) {
            return save(object);
        } else {
            return nextController.saveMessage(level, object);
        }
    }

    public Object repositoryMessage(int level) {
        if (this.level <= level) {
            return getRepository();
        } else {
            return nextController.repositoryMessage(level);
        }
    }
}

```

Above is part of our code in chains of responsibility pattern. As we can see, there are 3 levels of request handler ---- comment, user and reply. Each controller point to the next level controller. And the Abstractcontroller handles different request according to its level and request income. If the request level equals to the controller level itself, it will handle the request. Otherwise, it pass the request to the next level controller. Since the comment and reply can be successfully added while using the thie chains of responsibility, indicating that the pattern works well in the code.

## Flight Weight Pattern

```

package controllers;

import models.*;
import java.util.HashMap;

public class RepositoryFactory {
    public static final HashMap<String, AbstractController> map = new HashMap<>();

    public static AbstractController getRepositoryController (String type, Object injectObject) {
        AbstractController controller = map.get(type);
        if (controller == null) {
            controller = createNewObject(type, injectObject);
            map.put(type, controller);
        }
        return controller;
    }

    private static AbstractController createNewObject(String type, Object injectObject) {
        switch(type) {
            case "comment": return new CommentRepositoryController(((CommentRepository)injectObject));
            case "reply": return new ReplyRepositoryController(((ReplyRepository)injectObject));
            case "user": return new UserRepositoryController(((UserRepository)injectObject));
            default: return null;
        }
    }
}
}

```

Above is part of codes for the Flight weight pattern. In the Repository Factory class, all the related repository controller objects are stored in the hashmap for reuse. So that there is only one object for each controller during the system running. Since the comment and reply can be successfully added while using the this chains of responsibility, indicating that the pattern works well in the code.

## Factory Pattern

```

AttachmentFactory.java
1 package models;
2
3
4 public class AttachmentFactory {
5     public AttachmentFactory() {
6
7     }
8
9     public Attachment getAttachment(String attachmentType) {
10         if(attachmentType == null) {
11             return null;
12         }
13         if(attachmentType.equalsIgnoreCase("reply")) {
14             return new Reply();
15         } else if(attachmentType.equalsIgnoreCase("suggestion")) {
16             return new Suggestions();
17         }
18         return null;
19     }
20 }

```

```
AttachmentFactory af= new AttachmentFactory();
Reply reply= (Reply)af.getAttachment("reply");
reply.setFromUser(fromUser);
reply.setToUser(toUser);
reply.setTimestamp(timestamp);
reply.setContent(content);
```

Above is the code for factory pattern. When we want to encapsulate the implementation details and implementation can be changed without any impact on caller API. So when we want to create new Reply or Suggestions object, we can create a factory object and get that object.

## Adapter Pattern

```
VoterAdapter.java
1 package models;
2
3 public class VoterAdapter implements Voter{
4
5     AdvancedVoter adv;
6
7     public VoterAdapter(String objectType) {
8         if(objectType.equalsIgnoreCase("comment")) {
9             adv= new CommentVoter();
10        } else if(objectType.equalsIgnoreCase("suggestions")) {
11            adv= new SuggestionsVoter();
12        }
13    }
14
15    @Override
16    public void vote(String type, Object object) {
17        if(type.equalsIgnoreCase("comment")) {
18            adv.commentVote(object);
19        } else if(type.equalsIgnoreCase("suggestions")) {
20            adv.suggestionsVote(object);
21        }
22    }
23
24 }
```

Above is the main code for adapter pattern. We need to create a new Adapter object when we want to vote up or down. It can run successfully.

## Decorator Pattern



Here is the Abstract WorkflowDecorater that can be extended by concrete decorator classes

```
WorkflowDecorator.java > No Selection
1 package controllers;
2
3
4 import com.fasterxml.jackson.databind.node.ArrayNode;
5 import com.fasterxml.jackson.databind.JsonNode;
6 import models.SearchResult;
7 import models.Workflow;
8 import play.api.mvc.*;
9 import play.mvc.Result;
10 import util.APICall;
11 import util.Constants;
12 import views.html.*;
13 import play.mvc.Controller;
14 import java.util.ArrayList;
15 import java.util.List;
16 import models.User;
17 import com.fasterxml.jackson.databind.node.ObjectNode;
18 import util.APICall;
19 import util.Constants;
20
21 public abstract class WorkflowDecorator{
22
23     private final static String CREATE = Constants.NEW_BACKEND + "workflow/addComment";
24
25     // public static JsonNode create(ObjectNode node) {
26     //     JsonNode response = APICall.postAPI(CREATE, node);
27     //     return response;
28     // }
29 }
```

And this is the concrete class WorkflowCommentDecorator that creates a comment and returns a JsonNode

```
CommentWorkflowDecorator.java > No Selection
1 package controllers;
2
3
4 import com.fasterxml.jackson.databind.node.ArrayNode;
5 import com.fasterxml.jackson.databind.JsonNode;
6 import models.SearchResult;
7 import models.Workflow;
8 import play.api.mvc.*;
9 import play.mvc.Result;
10 import util.APICall;
11 import util.Constants;
12 import views.html.*;
13 import play.mvc.Controller;
14 import java.util.ArrayList;
15 import java.util.List;
16 import models.User;
17 import com.fasterxml.jackson.databind.node.ObjectNode;
18 import util.APICall;
19 import util.Constants;
20
21 public class CommentWorkflowDecorator extends WorkflowDecorator{
22
23     private final static String CREATE = Constants.NEW_BACKEND + "workflow/addComment";
24
25
26     public static JsonNode create(ObjectNode node) {
27         JsonNode response = APICall.postAPI(CREATE, node);
28         return response;
29     }
30 }
```

## State Pattern

Here is the SearchContext class used to maintain the current SearchState.

SearchContext.java > class SearchContext

```
1 package controllers;
2
3 import com.fasterxml.jackson.databind.JsonNode;
4 import models.SearchResult;
5 import models.Workflow;
6 import play.api.mvc.*;
7 import play.mvc.Result;
8 import util.APICall;
9 import util.Constants;
10 import views.html.*;
11 import play.mvc.Controller;
12 import java.util.ArrayList;
13 import java.util.List;
14 import models.User;
15
16 public class SearchContext {
17
18     private SearchState state;
19     public SearchContext(){
20         this.state = null;
21     }
22
23     public void setState(SearchState state) {
24         this.state = state;
25     }
26
27     public SearchState getState() {
28         return this.state;
29     }
30 }
```

The SearchState interface is defined as follows. Here we use Java Templating to accomodate Searches of objects of different types.

SearchState.java > No Selection

```
1 package controllers;
2
3 import com.fasterxml.jackson.databind.JsonNode;
4 import models.SearchResult;
5 import models.Workflow;
6 import play.api.mvc.*;
7 import play.mvc.Result;
8 import util.APICall;
9 import util.Constants;
10 import views.html.*;
11 import play.mvc.Controller;
12 import java.util.ArrayList;
13 import java.util.List;
14 import models.User;
15
16
17 public interface SearchState<T>{
18
19     public ArrayList<T> doSearch(JsonNode response);
20 }
```

One example of a concrete Search Class that extends this interface is the WorkflowSearcState class shown below.

```

1 package controllers;
2
3 import com.fasterxml.jackson.databind.JsonNode;
4 import models.SearchResult;
5 import models.Workflow;
6 import play.api.mvc.*;
7 import play.mvc.Result;
8 import util.APICall;
9 import util.Constants;
10 import views.html.*;
11 import play.mvc.Controller;
12 import java.util.ArrayList;
13 import java.util.List;
14 import models.User;
15
16 public class WorkflowSearchState implements SearchState<Workflow>{
17
18     @Override
19     public ArrayList<Workflow> doSearch(JsonNode wfresponset){
20         System.out.println("Searching users");
21         ArrayList<Workflow> wfArtrt = new ArrayList<Workflow>();
22
23         for (JsonNode n: wfresponset) {
24             Workflow wf = new Workflow(n);
25             wfArtrt.add(wf);
26         }
27         return wfArtrt;
28     }
29 }
30

```

## Criteria Pattern

The Criteria interface is defined as below

```

1 package controllers;
2
3
4 import com.fasterxml.jackson.databind.node.ArrayNode;
5 import com.fasterxml.jackson.databind.JsonNode;
6 import models.SearchResult;
7 import models.Workflow;
8 import play.api.mvc.*;
9 import play.mvc.Result;
10 import util.APICall;
11 import util.Constants;
12 import views.html.*;
13 import play.mvc.Controller;
14 import java.util.ArrayList;
15 import java.util.List;
16 import models.User;
17
18 public interface Criteria{
19     public ArrayList<User> meetCriteria(ArrayList<User> userList);
20 }
21

```

With the concrete class UserEmailGmailCriteria implementing the interface and defining the search code.



```

1 package controllers;
2
3 import com.fasterxml.jackson.databind.JsonNode;
4 import models.SearchResult;
5 import models.Workflow;
6 import play.api.mvc.*;
7 import play.mvc.Result;
8 import util.APICall;
9 import util.Constants;
10 import views.html.*;
11 import play.mvc.Controller;
12 import java.util.ArrayList;
13 import java.util.List;
14 import models.User;
15
16
17 public class UserEmailGmailCriteria implements Criteria{
18     @Override
19     public ArrayList<User> meetCriteria(ArrayList<User> userList){
20         ArrayList<User> userArr = new ArrayList<User>();
21         for (User u1:userList){
22             if (u1.getEmail().contains("@gmail.com")) {
23                 System.out.println(u1.getEmail());
24                 userArr.add(u1);
25             }
26             else
27                 System.out.println("not gmail ".concat(u1.getEmail()));
28         }
29         return userArr;
30     }
31 }

```

## 6. Conclusions and future work

Although there are many complex functionalities in this project, we complete this job at last. Thanks to the professor Jia and TAs. Each time we have problems or difficulties, we usually search help from them and they could give us huge help. From this project, we have learnt a lot like how to deploy project with docker, how to use design pattern in practical project, how to allocate tasks correctly and conveniently, how to merge two projects together from different versions and so on. And I think it is so useful for our later work in life.

But there still exist some other work that we would prepared to do but did not do, such as: it needs a search page on the workflow list that we can type in the author's name or workflow's title and then it can return a list of workflow. What's more, we can also add one functionality that when leaving comments in one workflow, he can also mention another user like facebook or twitter' sign @. This functionality makes the interoperability between users better and more convenient. And one functionality of sharing could be also added into this project, to make the service more popular to the public.