Architecture Analysis and Improvement of Workflow-Centric Scientific Social Network

SAD - Team9

- Jiakai Zhang
- Qiaoyi Chen
- Ziming Wang
- Nan Wang

Table of Contents

1. Introduction

Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system. Each structure comprises software elements, relations among them, and properties of both elements and relations. The architecture of a software system is a metaphor, analogous to the architecture of a building.

As a result, software architecture has become increasingly important in the last two decades in software engineering community. At the heart of every well-engineered software system is always its software architecture. Software architecture deals with the high-level building blocks that represent an underlying software system. Software architectures that have been proved to be useful for families of systems are often codified into architectural styles (patterns). Also, a good architecture of a software system can lead to better quality attributes including reusability, extensibility, scalability, maintainability, and ect.

A software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can

be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

As a result, good designers do not solve every problem from first principles. They reuse design patterns to help them solve architectural design problems. Design patterns are preferred because they are common design vocabularies, thus allowing engineers to abstract a problem and talk about that abstraction in isolation from its implementation. Design patterns also capture design expertise and allow that expertise to be communicated, thus promoting design reuse and avoiding mistakes. What's more, design patterns help to improve documentation (less is needed) and understandability (patterns are described well once) for software architecture design.

For this workflow-centric scientific social network project, we firstly analyzed the architecture of the legacy system by using ATAM (Architecture Tradeoff Analysis Method) systematically. After that, we carefully abstracted and analyzed the architecture and architectural approaches that the legacy system has adopted. After we had a clear and intensive understanding of the architecture of the whole project, we tried to improve the existing architectural design of the legacy system by applying some useful design patterns. For each design pattern, we carefully analyzed its definition, benefits, constraints, the place where we adopted a specific design pattern, and also detailed information about how to implement a specific design pattern using class diagram and real code.

2. Motivation

Usually, scientists have to meet and mingle with other scientists to discuss scientific researches and workflows. However, more and more scientists now choose to attend their academic discussions online, and in most cases they work as a team to finish an academic issue. Because of the increasing needs for that, an open workflow-centric social network is built, aimed specifically at scientists and researchers. To the scientists and researchers, they want to have a platform that they can share ideas and interact with their peers conveniently, especially focusing on the workflow part of the scientific researches which is the instrumental needs for scientists to collaborate, In a scientific workflow like experiments, they should be able to publish their process and development, have a look at other team members logs and ideas and make comments or interact with each other conveniently.

This Workflow-Centric Scientific Social Network is focused on experiment workflow discussion, collaboration, and search functionalities of actual scientific researches, which differentiates it from other scientific social networks in the market which acts more like a regular social network. The goal of this application is to make it easy for the next generation of scientists to contribute to scientific methods, build communities, reduce time-to-experiment, share expertise and experiences, and avoid reinvention.

To improve the system, we choose to analyze the architecture of the existing software system and make several improvements on it. By applying good software design patterns, the system can have better performance, maintainability, reliability, reusability, extensibility, interoperability, scalability, etc. By doing analysis and improvement, we aim to achieve the goal of the software system, which is stated above.

3. Related Work

To analyze and improve this workflow-centric scientific social network project, we have done make related work as below.

- We analyzed the architecture of the CMDA project by applying ATAM(Architecture Tradeoff Analysis Method).
- We carefully studied all the 23 design patterns in details, including creational patterns, structural patterns and behavioral patterns.
- After we gained a thorough understanding of every design patterns, we actively adopted 8 different useful design patterns for the legacy system by analyzing their definition, pros & cons and detailed implementation(UML and code).
- During the process of improving, we utilized the docker as a powerful tool for easy development, integration and deployment between different machines.
- We also pushed our improved project into github for better sharing and version control.

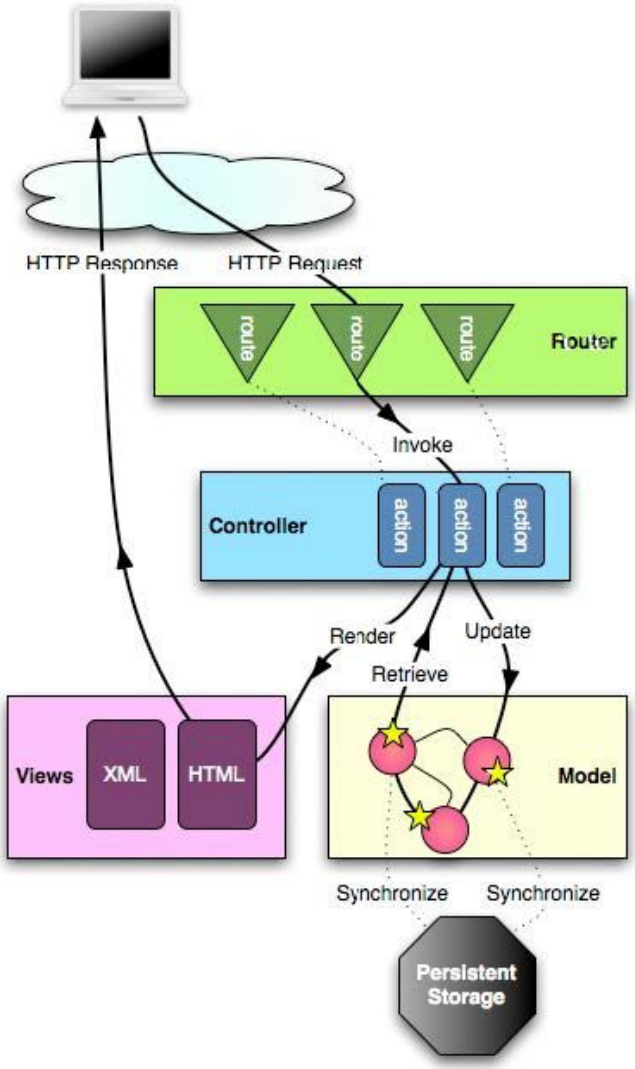
4. System Design and Implementation

The design of the system follows Model-View-Controller (MVC) architectural pattern in general using Play framework, with support of MySQL database at the back-end, and Bootstrap and jQuery at the front-end.

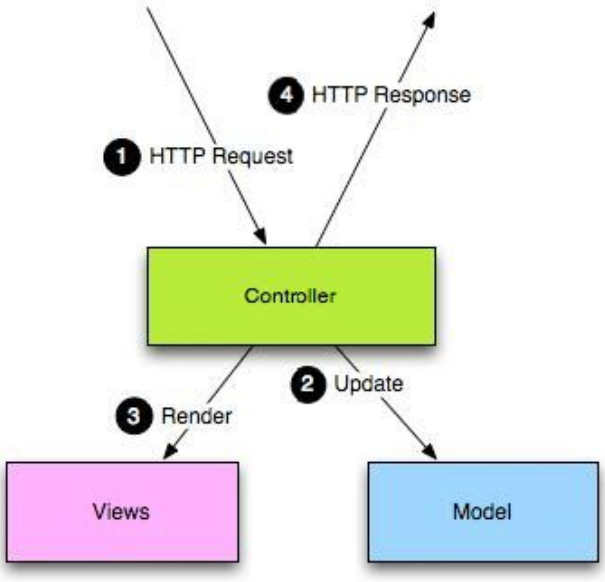
The general code structure is constructed using the Play framework. Play is an open source web application framework, written in Java and Scala, which strictly follows the MVC architectural pattern. It aims to optimize developer productivity by using convention over configuration, hot code reloading and display of errors in the browser. It is fully stateless and only Request/Response oriented. All HTTP Requests follow the same path:

- An HTTP request is received by the framework.
- The router component tries to find the most specific route able to accept this request.
- The corresponding action method in the router is invoked.
- The application code is executed.
- If a complex view needs to be generated, a template file is rendered.
- The result of the action method (HTTP response code and content) is then written as an HTTP response.

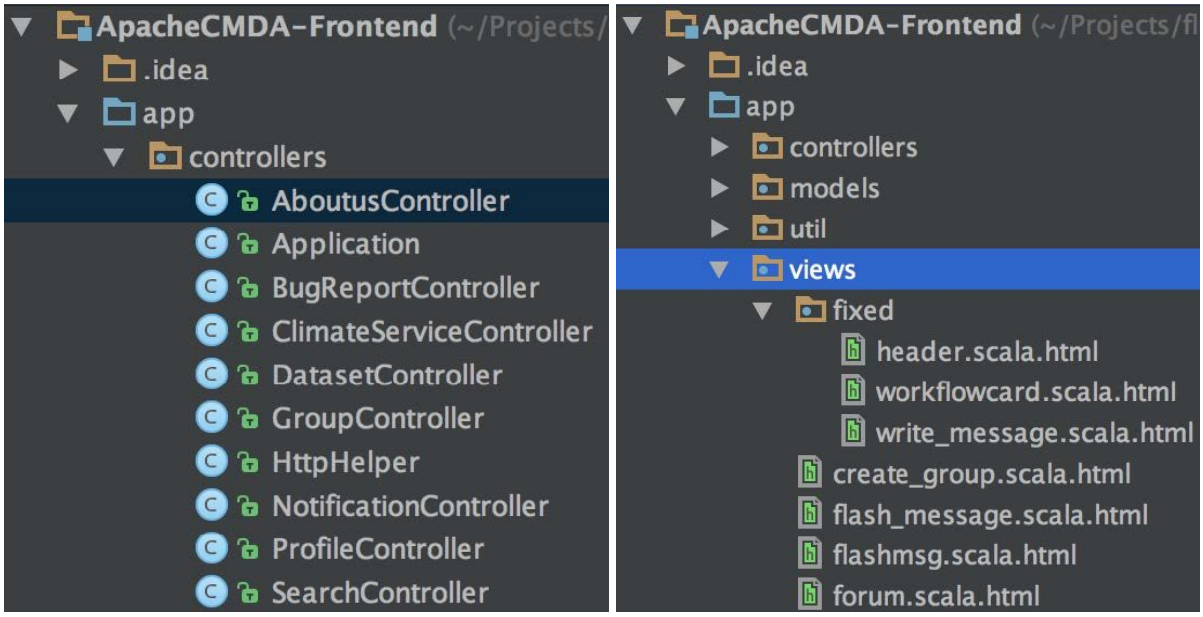
The following diagram summarizes the Play framework and the HTTP request path:



Play framework splits the application into separate layers: the Presentation layer (front-end) and the Model layer (back-end). The Presentation layer is further splitted into View and Controller. The Model layer is further splitted into Controller and Model. The diagram below demonstrates the relationships between Model, View, and Controller:



The front-end is coded using Scala (router), jQuery (JavaScript library) and Bootstrap (HTML, CSS, JavaScript framework). The back-end is coded in Java and MySQL (database). The project structures of front-end and back-end is shown below:



5. Experiment and Analysis

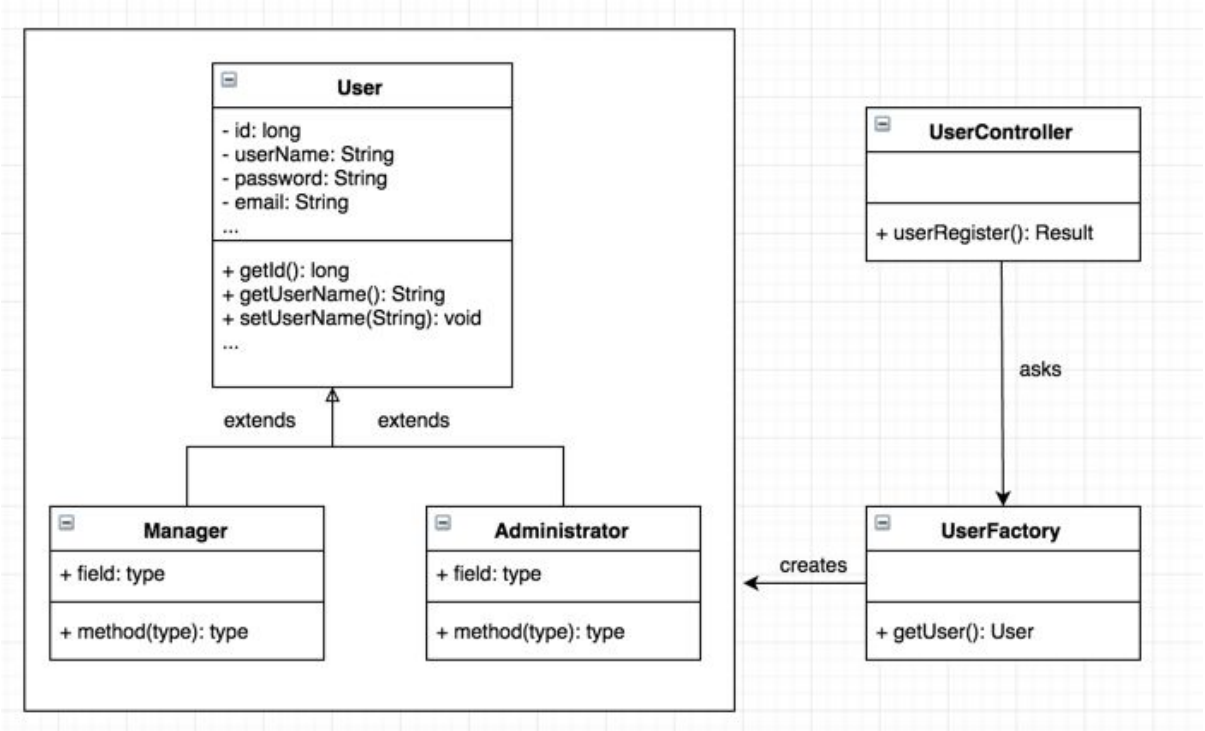
5.1 Factory

Factory pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method, either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes, rather than by calling a constructor.

The benefit of using factory is that this pattern provides a superclass for creation while hiding the creation logic to the client. The constraint of this design pattern is that sometimes this pattern is unnecessary when creating logic is simple.

So for this CMDA project, we can use this factory design pattern to register/create user with different types. Currently the legacy system can only support general users, but in future we may need to specify different user types like common user, manager, administrator, and etc. So we can use this factory pattern to register/create different types of user with just one single factory interface, and hide the creation logic to the client.

The class diagram and parts of the code for this Factory pattern are shown as below.



Class diagram

- Define a UserFactory

```
public class UserFactory {
    public User getUser(String userType, String userName, String email, String password) {
        if (userType == "user") {
            return new User(userName, email, password);
        }
        else if (userType == "administrator") {
            return new Administrator(userName, email, password);
        }
        else if (userType == "manager") {
            return new Manager(userName, email, password);
        }
        else {
            return null;
        }
    }
}
```

- Create an user object based on specific user type.

```
UserFactory userFactory = new UserFactory();
User user = userFactory.getUser(userType, name, email, MD5Hashing(password));
```

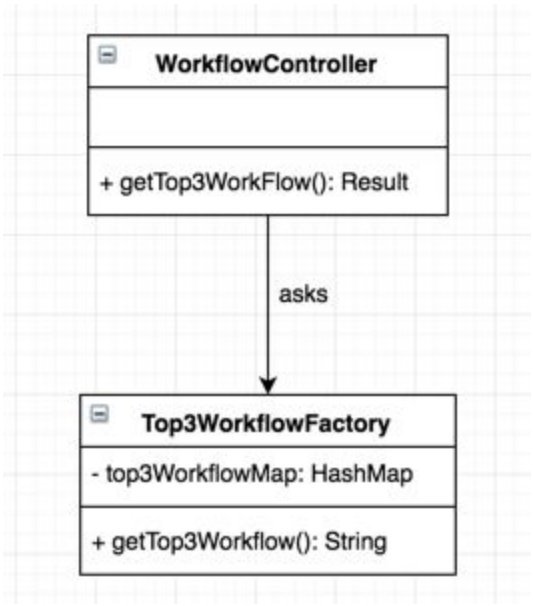
5.2 Flyweight

A flyweight is a structural design pattern that minimizes memory use by sharing as much data as possible with other similar objects. it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the flyweight objects temporarily when they are used.

The main benefit of using this design pattern is to reduce the number of objects created. Flyweight pattern is good for decreasing memory footprint and increasing performance by reusing already existing similar objects when creating objects. One constraint of using this design pattern is that it is not preferred when objects being created have no or little similarity.

So for this CMDA project, we could use this Flyweight pattern when creating top-3 workflow object. The reason why we chose this scenario is that the top-3 workflow object is quite stable and similar giving a period of time. So it is a good way to use the flyweight pattern to reuse the existing similar top-3 workflow object each time when creating new top-3 workflow object.

The key technology of implementing this flyweight pattern is to use a kind of data structure named HashMap. The class diagram and parts of the code for this Flyweight pattern are shown as below.



Class Diagram

- Define a Top3WorkflowFactory with a HashMap.

```
public class Top3WorkflowFactory {
    private static final HashMap<List<Workflow>, String> top3WorkflowMap = new HashMap<>();

    public static String getTop3Workflow(List<Workflow> topWorkflow) {
        String result = top3WorkflowMap.get(topWorkflow);

        if (result == null) {
            result = new GsonBuilder().excludeFieldsWithModifiers(Modifier.PROTECTED).create().toJson(topWorkflow);
            top3WorkflowMap.put(topWorkflow, result);
            System.out.println("maintain map for topworkflow");
        }
        return result;
    }
}
```

- Get String result by reusing the similar objects stored in the HashMap, instead of creating from scratch every time.

```
public Result getTop3WorkFlow() {
    List<Workflow> topWorkflow = workflowRepository.findTop3Workflow();
    String result = Top3WorkflowFactory.getTop3Workflow(topWorkflow);
    return  ok(result);
}
```

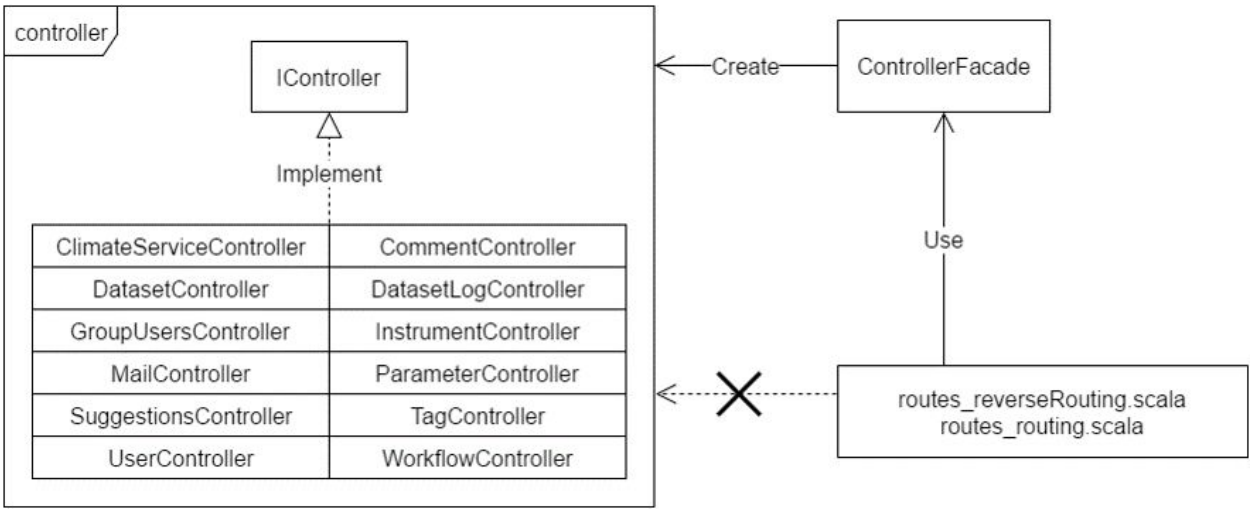
5.3 Façade

Facade pattern aims to hide the complexity of the system and provides an interface to the client. In the project, it can be used to provide a single simple interface for all Controllers in the back-end. The benefit of this is to hide complexity of the Controller code, and makes the code

clean. By changing the Controller to a Facade pattern and letting the client to use Facade, it can have better maintainability and extensibility.

In the “controller” package of the back-end code, there are 12 Controller classes which have methods to control all functionalities of the application. We create an IController interface and let all the Controller classes implement it. Then we create a ControllerFacade class so that it will take in any Controller class during initialization and the client code can directly access method in ControllerFacade. By making this change, the client code can directly use a single interface instead of using different Controller classes which is not very convenient. The client code is the 2 router Scala files.

The class diagram and parts of the code in Facade class is shown below:



```java
@Named
@Singleton
public class ControllerFacade extends Controller {

    private ClimateServiceController climateServiceController;
    private CommentController commentController;
    private DatasetController datasetController;
    private DatasetLogController datasetLogController;
    private GroupUsersController groupUsersController;
    private InstrumentController instrumentController;
    private MailController mailController;
    private ParameterController parameterController;
    private SuggestionsController suggestionsController;
    private TagController tagController;
    private UserController userController;
    private WorkflowController workflowController;

    @Inject
    public ControllerFacade(IController controller) {
        if (controller instanceof ClimateServiceController) {
            climateServiceController = (ClimateServiceController) controller;
        } else if (controller instanceof CommentController) {
            commentController = (CommentController) controller;
        } else if (controller instanceof DatasetController) {
            datasetController = (DatasetController) controller;
        } else if (controller instanceof DatasetLogController) {
            datasetLogController = (DatasetLogController) controller;
        } else if (controller instanceof GroupUsersController) {
            groupUsersController = (GroupUsersController) controller;
        } else if (controller instanceof InstrumentController) {
            instrumentController = (InstrumentController) controller;
        } else if (controller instanceof MailController) {
            mailController = (MailController) controller;
        } else if (controller instanceof ParameterController) {
            parameterController = (ParameterController) controller;
        } else if (controller instanceof SuggestionsController) {
            suggestionsController = (SuggestionsController) controller;
        } else if (controller instanceof TagController) {
            tagController = (TagController) controller;
        } else if (controller instanceof UserController) {
            userController = (UserController) controller;
        } else if (controller instanceof WorkflowController) {
            workflowController = (WorkflowController) controller;
        }
    }

    // ClimateServiceController

    public Result climateAddClimateService() { return climateServiceController.addClim

    public Result climateSavePage() { return climateServiceController.savePage(); }

    public Result climateDeleteClimateServiceById(long id) {
        return climateServiceController.deleteClimateServiceById(id);
    }

    public Result climateDeleteClimateServiceByName(String name) {
        return climateServiceController.deleteClimateServiceByName(name);
    }

    public Result climateUpdateClimateServiceById(long id) {
        return climateServiceController.updateClimateServiceById(id);
    }

    public Result climateUpdateClimateServiceByName(String oldName) {
        return climateServiceController.updateClimateServiceByName(oldName);
    }
```

There is one disadvantage of applying this design pattern. The client code (2 Scala routers) has more than a thousand lines and the places where it uses Controller classes are all scattered. So, it's very difficult and time-consuming to change all the client code to use the Facade class. Examples of places where the 2 Scala routers need to be changed are shown below:

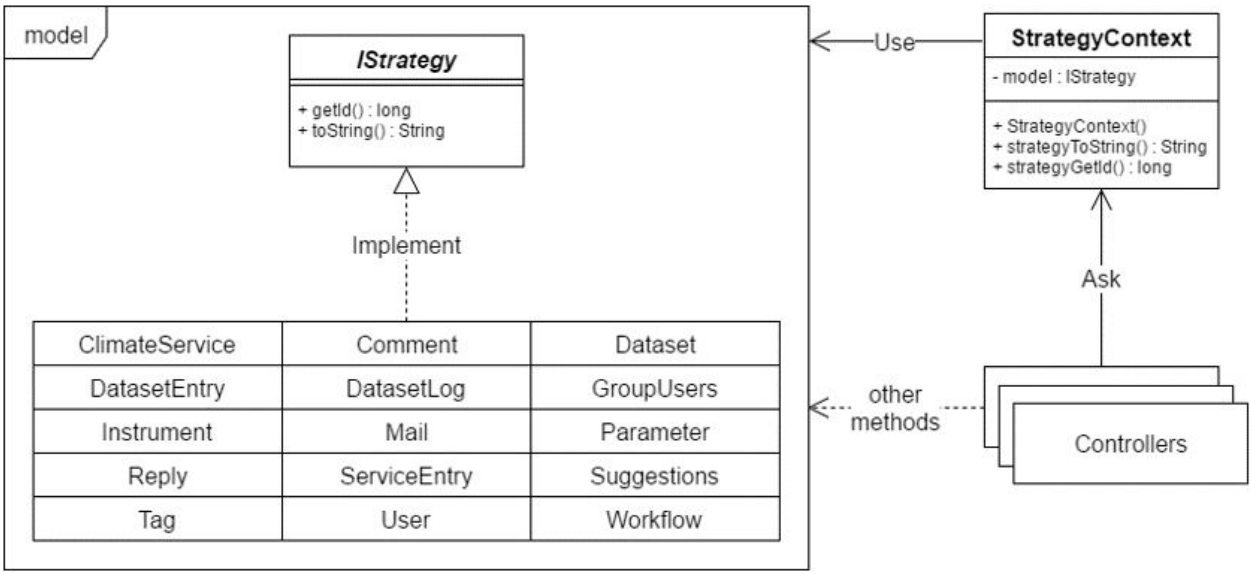
```scala
// @LINE:35
def updateDatasetById : JavascriptReverseRoute = JavascriptReverseRoute(
    "controllers.ControllerFacade.datasetUpdateDatasetById",
    """
        function(id) {
        return _wA({method:"PUT", url:""" + _prefix + { _defaultPrefix } +
        }
    """
)
```

5.4 Strategy

Strategy pattern can be applied so that a class behavior or its algorithm can be changed at run time. In the project, it can be applied to Model classes in the back-end so that some behaviors (such as methods toString(), and getId()) can be changed at run time. Originally, the client class (Controller classes, which use the Model classes) need to initialize different Model classes and call the same methods (such as toString() and getId()). By using the Strategy pattern, the client can only use a Strategy class, call a single method in the class, and achieve different behaviors. This approach allows one API to have multiple functionalities, which have better customizability and flexibility than the original implementation. It also has better maintainability and reusability.

In the “model” package of the back-end code, there are 15 Model classes which have some common methods which have different behaviors. The best example is the toString() method. Every Model class have a different toString() method so that they can generate strings describing their own class. We create an IStrategy interface which contains common APIs of all the Model classes. We make all the Model classes implement the interface. Then, a StrategyContext class is created so that it contains an IStrategy instance and some common APIs such as strategyToString(). By calling this API in the client, it can generate strings according to the instance type in StrategyContext class. So, when the client (Controller classes) need to call toString() methods of many different Model classes, it only need to call one single method (strategyToString()) to achieve this now. The client only need to pass an instance of a particular Model class into StrategyContext object when it needs to change the behavior of strategyToString().

The class diagram and the code for StrategyContext implementation is shown below:




```java
@Entity
public class StrategyContext {
    @Id
    private IStrategy model;

    public StrategyContext(IStrategy model) { this.model = model; }

    public String strategyToString() { return model.toString(); }

    public long strategyGetId() { model.getId(); }

}
```

One disadvantage of the Strategy pattern here (or inability) is that if the client need to call other methods which is unique for a particular Model class, the strategy pattern does not work. In this situation, the client still needs to use the original Model classes and call their methods individually. The following picture shows the place where it uses the Strategy class:

```java
try {
    Instrument instrument = instrumentRepository.findOne(instrumentId);
    List<ClimateService>climateServiceSet = new ArrayList<~>();
    for (int i = 0; i < climateServicesId.size(); i++) {
        climateServiceSet.add(climateServiceRepository.findOne(climateServicesId.get(i)));
    }
    Dataset dataset = new Dataset(name, dataSourceNameinWebInterface, agencyId, instrument, climateServiceSet,
    Dataset savedServiceConfiguration = datasetRepository.save(dataset);
    StrategyContext strategyDataset = new StrategyContext(savedServiceConfiguration);
    System.out.println("Dataset saved: "+ strategyDataset.strategyGetId());
    System.out.println(strategyDataset.strategyToString());
    return created(new Gson().toJson(new StrategyContext(dataset).strategyGetId()));
} catch (PersistenceException pe) {
    pe.printStackTrace();
    System.out.println("Dataset not created");
    return badRequest("Dataset not created");
}
```

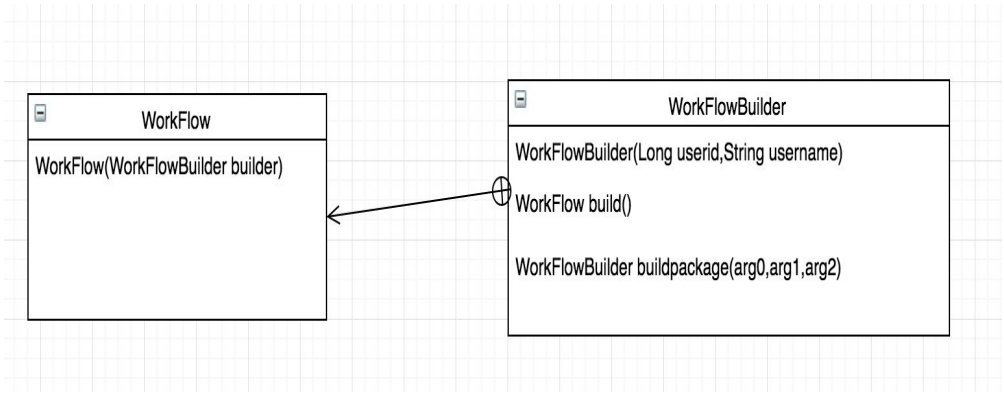
5.5 Builder

Build pattern is Builder design pattern is to separate the construction of a complex object from its representation and it handles the situation when the object is in complexity which makes the construction too complicated, then build pattern could focus on the contraction of specific part without concern remain part.The benefits of using it is Making the same construction process create different representations,Provides control over steps of construction process and allows you to vary a product's internal representation,while the disadvantage is that Builder is not preferred when the construction of the object is simple.

In the project, it can be used to build an inner class and main class can call it in the construction functions.Thus the initialization of other attributes is handled by build class and we could just focus on the construction of the part we concerned.

The class diagram and parts of the code in Builder pattern is shown below:

Create an inner workflowbuilder class in the workflow class.



In the "model" package of the back-end code, there is a build class in the workflow class. Thus,the work-flow could call the build function build to build the object and just initialize the username and id.And the other attributes are initialized in the build class.

```
public static class WorkFlowBuilder{



    public WorkFlowBuilder(long userID,String userName){
        this.userID = userID;
        this.userName = userName;
    }


    public WorkFlowBuilder buildPackage(String wfVisibility,
                                        String status,
                                        long viewCount,
                                        long groupId,
                                        boolean edit,
                                        String wfUrl,
                                        String wfInput,
                                        String wfOutput,
                                        Date wfDate) {
        this.wfVisibility = wfVisibility;
        this.status = status;
        this.viewCount = viewCount;
        this.groupId = groupId;
        this.edit = edit;
        this.wfUrl = wfUrl;
        this.wfInput = wfInput;
        this.wfOutput = wfOutput;
        this.wfDate = wfDate;

        return this;
    }

    public Workflow build() { return new Workflow(this); }
```

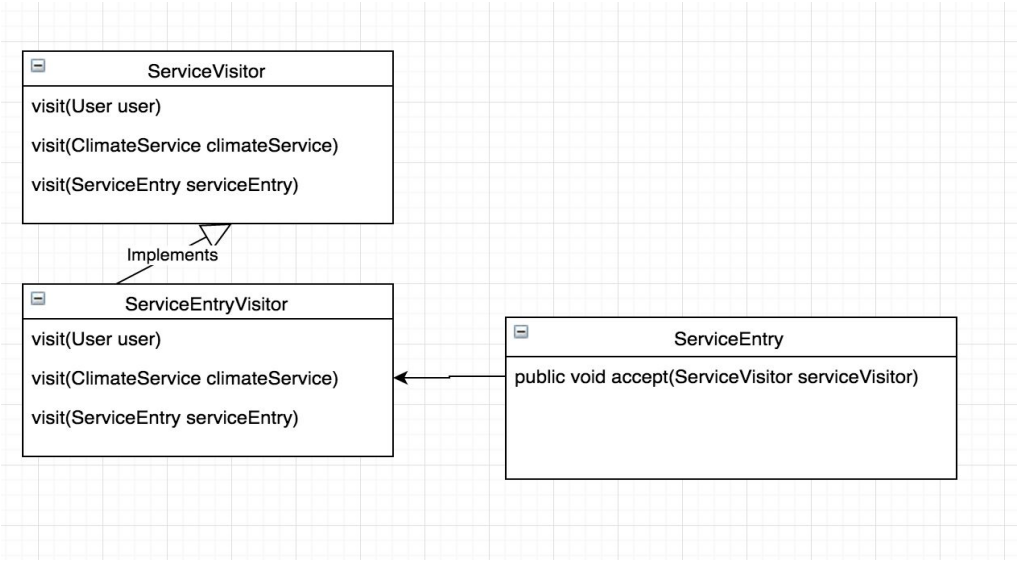
```
public Workflow(WorkFlowBuilder builder) {

    //required
    this.userID = builder.userID;
    this.userName = builder.userName;

    //optional
    this.wfDate = builder.wfDate;

}
```

5.6 Visitor

Visitor pattern is Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates and it handles the situation when the object is in complexity and be composed of other objects which makes the visit and call of it too complicated, then visitor pattern could focus on the calling and visit of specific part without focusing on other part.The benefits of applying visitor pattern is the nature of the Visitor makes it an ideal pattern to visit the special part of an object without interfere with the whole part, while the disadvantage of it is that visitor is not preferred when object is quite simple has no composite parts.

In the project, we could define a visitor interface and implemented it regarding to the target objects. And then define the accept function in each part of the target objects to accept the visit.

The class diagram and parts of the code in Visitor pattern is shown below:



In the “model” package of the back-end code, there is a Service Visitor Interface and we create a service entry visitor to implemented it. The target class service entry class including the user and climate service.Then after define the accept method in these object class we could use the method in visitor to visit the user and climate service in service entry objects and don’t need to focus on other part in service entry object.

```
public interface ServiceVisitor {
    void visit(User user);
    void visit(ClimateService climateService);
    void visit(ServiceEntry serviceEntry);
}
```

```
public class ServiceEntryVisitor implements ServiceVisitor{

    public void visit(User user) {
        System.out.println("Visiting " + user.toString());
    }
    public void visit(ClimateService climateService) {
        System.out.println("Visiting " + climateService.toString());
    }
    public void visit(ServiceEntry serviceEntry) {
        System.out.println("Visiting " + serviceEntry.toString());
    }
```

```
public void accept(ServiceVisitor visitor) {
    visitor.visit(this);
    climateService.accept(visitor);
    user.accept(visitor);
}
```

```
ServiceEntry serviceEntry = new ServiceEntry();
ServiceVisitor visitor = new ServiceEntryVisitor();
serviceEntry.accept(visitor);
```

5.7 Template

Template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure

In the template method of this design pattern, one or more algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed.

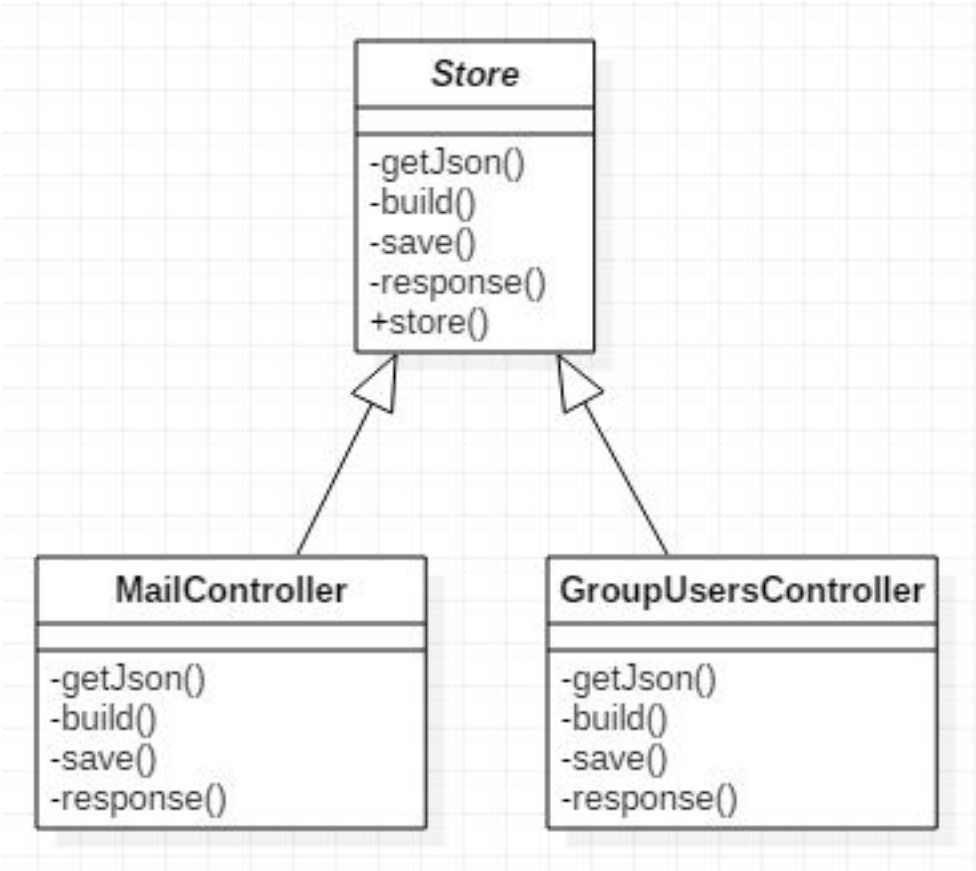
In object-oriented programming, first a class is created that provides the basic steps of an algorithm design. These steps are implemented using abstract methods. Later on, subclasses change the abstract methods to implement real actions. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.

The Template Method pattern thus manages the larger picture of task semantics, and more refined implementation details of selection and sequence of methods. This larger picture calls abstract and non-abstract methods for the task at hand. The non-abstract methods are completely controlled by the template method, but the abstract methods, implemented in subclasses, provide the pattern's expressive power and degree of freedom. Template Method's abstract class may also define hook methods that may be overridden by subclasses. Some or all of the abstract methods can be specialized in a subclass, allowing the writer of the subclass to provide particular behavior with minimal modifications to the larger semantics. The template method (that is non-abstract) remains unchanged in this pattern, ensuring that the subordinate non-abstract methods and abstract methods are called in the originally intended sequence. The Template Method pattern occurs frequently, at least in its simplest case, where a method calls only one abstract method, with object oriented languages. If a software writer uses a polymorphic method at all, this design pattern may be a rather natural consequence. This is because a method calling an abstract or polymorphic function is simply the reason for being of the abstract or polymorphic method. The Template Method pattern may be used to add immediate present value to the software or with a vision to enhancements in the future. It is strongly related to the Non-Virtual Interface (NVI) pattern.

The Template Method pattern implements the Protected Variations GRASP principle, like the Adapter pattern does. The difference is that Adapter gives the same interface for several operations, while Template Method does so only for one.

Template pattern can avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm. As the meanwhile Inheritance bring strong coupling relation between classes.

The class diagram in CMDA system is as below:



The code is in MailController and GroupUsersController extends Store. Both MailController and GroupUsersController extend Store and define specific functionality of store to perform simiar steps.

```
public Object store() {
    JsonNode json = getJson();
    Object object = build(json);
    save(object);
    return response(object);
}

public JsonNode getJson() {
    return request().body().asJson();
}

public Mail build(JsonNode json) {
    String fromUserMail = json.path("fromUserMail").asText();
    String toUserMail = json.path("toUserMail").asText();
    User fromUser = userRepository.findByEmail(fromUserMail);
    User toUser = userRepository.findByEmail(toUserMail);


    String mailTitle = json.path("mailTitle").asText();
    String mailContent = json.path("mailContent").asText();
    String dateString = json.path("mailDate").asText();
    DateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
    Date mailDate = new Date();
    try {
        mailDate = dateFormat.parse(dateString);
    } catch (ParseException e) {
        e.printStackTrace();
    }

    return new Mail(fromUserMail, toUserMail, mailTitle, mailContent, mailDate);
} ? end build ?


public void save(Object mail) {
    if (mail != null) {
        mailRepository.save((Mail)mail);
    }
}

public JsonNode getJson() {
    return request().body().asJson();
}

public GroupUsers build(JsonNode json) {
    if (json == null) {
        System.out.println("group not created, expecting Json data");
        return null;
    }

    long userID = json.path("userID").asLong();
    String groupName = json.path("groupName").asText();
    String groupDescription = json.path("groupDescription").asText();

    User user = userRepository.findOne(userID);
    System.out.println("user is " + user);
    List<User> groupMembers = new ArrayList<User>();
    groupMembers.add(user);

    GroupUsers group = new GroupUsers(userID, groupName, groupDescription, groupMembers);
    System.out.println("group is " + group);
    return group;
}

public void save(Object group) {
    groupUsersRepository.save((GroupUsers)group);
}
```

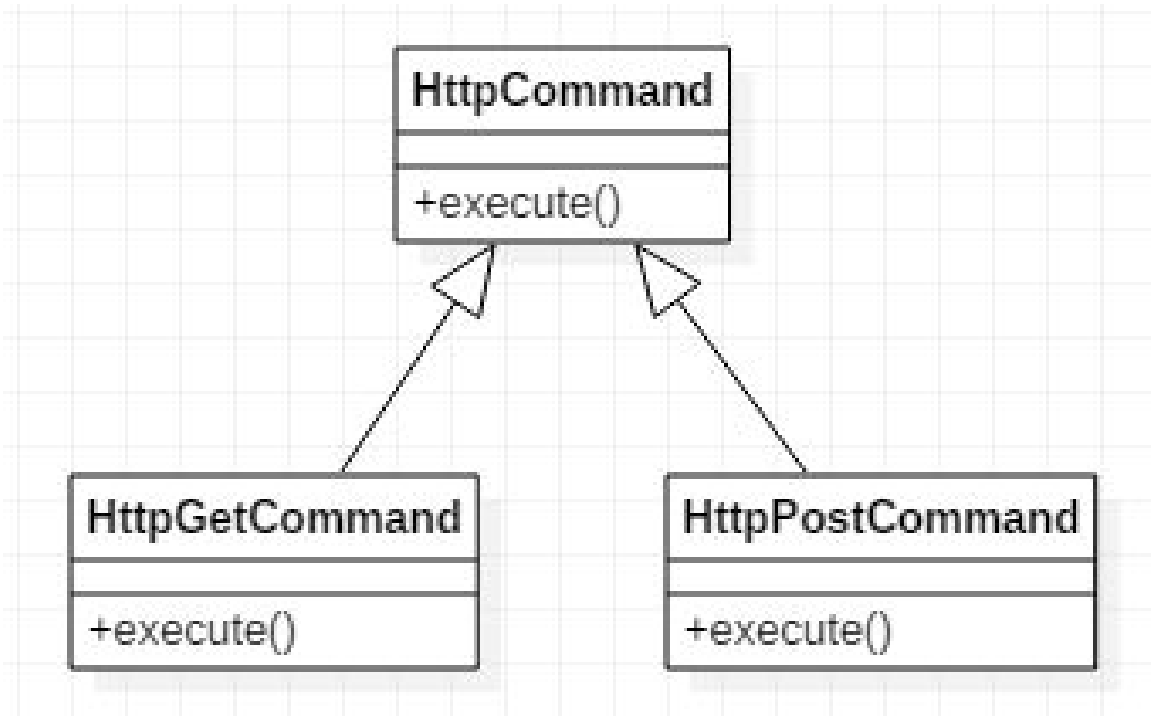
5.8 Command

Command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Command Pattern has advantages to make it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Command Pattern is not preferred when delegation is not required.

Four terms always associated with the command pattern are command, receiver, invoker and client. A command object knows about receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver then does the work. An invoker object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about command interface. Both an invoker object and several command objects are held by a client object. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object. Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.

The class diagram of command pattern used in CMDA is as below:



There is abstract class HttpCommand define the abstract function execute() which needs subclass to inherit.

```
abstract public class HttpCommand {
    abstract public String execute(String url, String param) throws Exception;
```

PostCommand class, GetCommand class are the subclass of HttpCommand which defines their own execute() function for executer to call.

```
public class PostCommand extends HttpCommand {
    public String execute(String urlStr, String jsonString) throws Exception {
        URL url = new URL(urlStr);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");
        conn.setRequestProperty("Accept", "application/json");
        conn.setDoOutput(true);
        OutputStream out = conn.getOutputStream();
        Writer writer = new OutputStreamWriter(out, "UTF-8");
        writer.write(jsonString);
        writer.close();
        out.close();
        return getHttpResponse(conn, 200);
    }
}
```

```
public class GetCommand extends HttpCommand {
    public String execute(String urlStr, String param) throws Exception {
        URL url = new URL(urlStr);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setDoOutput(true);
        conn.connect();
        return getHttpResponse(conn, 200);
    }

}
```

6. Conclusions and Future Work

As a conclusion, we systematically analyzed the architecture design of this work-flow centric scientific social network project. In general, the design of the system follows Model-View-Controller (MVC) architectural pattern by using Play framework, with support of MySQL database at the back-end, and Bootstrap and jQuery at the front-end.

Our team successfully applied 8 different useful design patterns to improve the existing architecture design of the legacy CMDA project. The 8 design patterns are Factory, Flyweight, Façade, Strategy, Builder, Visitor, Template and Command. The benefits of applying these design patterns include supporting a stronger reusability, extensibility, and maintainability for the system, and making the system architecture well-structured. There are also some constraints of using design patterns. For example, design patterns may bring increased effort to apply, and they also define more classes and interfaces to maintain, thus may leading to a poor performance due to indirection, and gaining the complexity of code.

In short, a perfect design patterns for all design problems do not exist. Every design pattern will has its own fitable scenario, benefits and constraints. We need to carefully analyze each design pattern's cons&pros, and then make a trade-off of whether should use it or not given different specific scenarios.

There are also some other good suggestions for improving this CMDA project. Micro-services architecture style and SOA architecture style are also good practices to improve the architecture design of the workflow-centric scientific social network project.

Both of these two styles are the most popular technologies for software development. We could decompose the whole project into multiple sub-systems, then implement micro-services for each of these sub-systems. By using micro-services, we could achieve the good standard of low coupling and high cohesion. What's more, micro-services enable a stronger composition freedom for development, and can make the whole project easier to reuse, extend, scale, compose and manage. And SOA style is mainly focused on enabling business-level programming by using the business processing technologies like BPEL. In SOA, the high-level business service could easily make use of the reusable utility and IT services to achieve a business goal. If we can adopt the SOA style for the CMDA project, it will greatly promote the sharing and communication process of the system with the public.