Architecture Analysis and Improvement

Faculty Advisor - Jia Zhang
Team - 9

[18-653] Software Architecture and Design, Spring 2016

Roadmap

Introduction

- Software architecture has become increasingly **important** in the software engineering community.

- Software architecture deals with the **high-level building blocks** that represent an underlying software system.

- **Analyze** existing legacy architecture, and try to **improve** it by using some useful **design patterns** to make the whole system more **reusable**, **extensible** and **maintainable**.

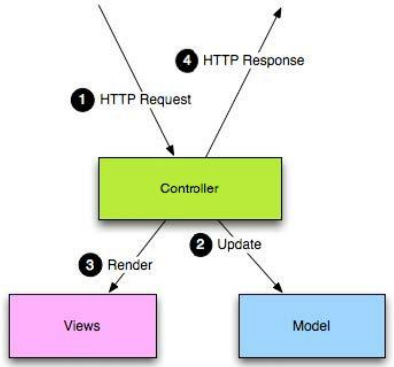
Motivation

- Provide scientists a platform to
  - Share ideas and interact with peers
  - Focus on workflow part of scientific researches

- Goals
  - Easier contribution to scientific methods
  - Build communities
  - Reduce time-to-experiment
  - Share expertise
  - Share experiences
  - Avoid reinvention

System Structure

Play framework (MVC)



- Front-end
  - View
    - Bootstrap
    - jQuery
  - Controller
    - RESTful APIs
- Back-end
  - Controller
    - Data operation support for Model
  - Model
    - MySQL database

Design Pattern

- **Factory**
- **Flyweight**
- **Façade**
- **Strategy**
- **Builder**
- **Visitor**
- **Template**
- **Command**

Factory

- **Definition:**

  Create obje

  with a comı

- **Pros:**

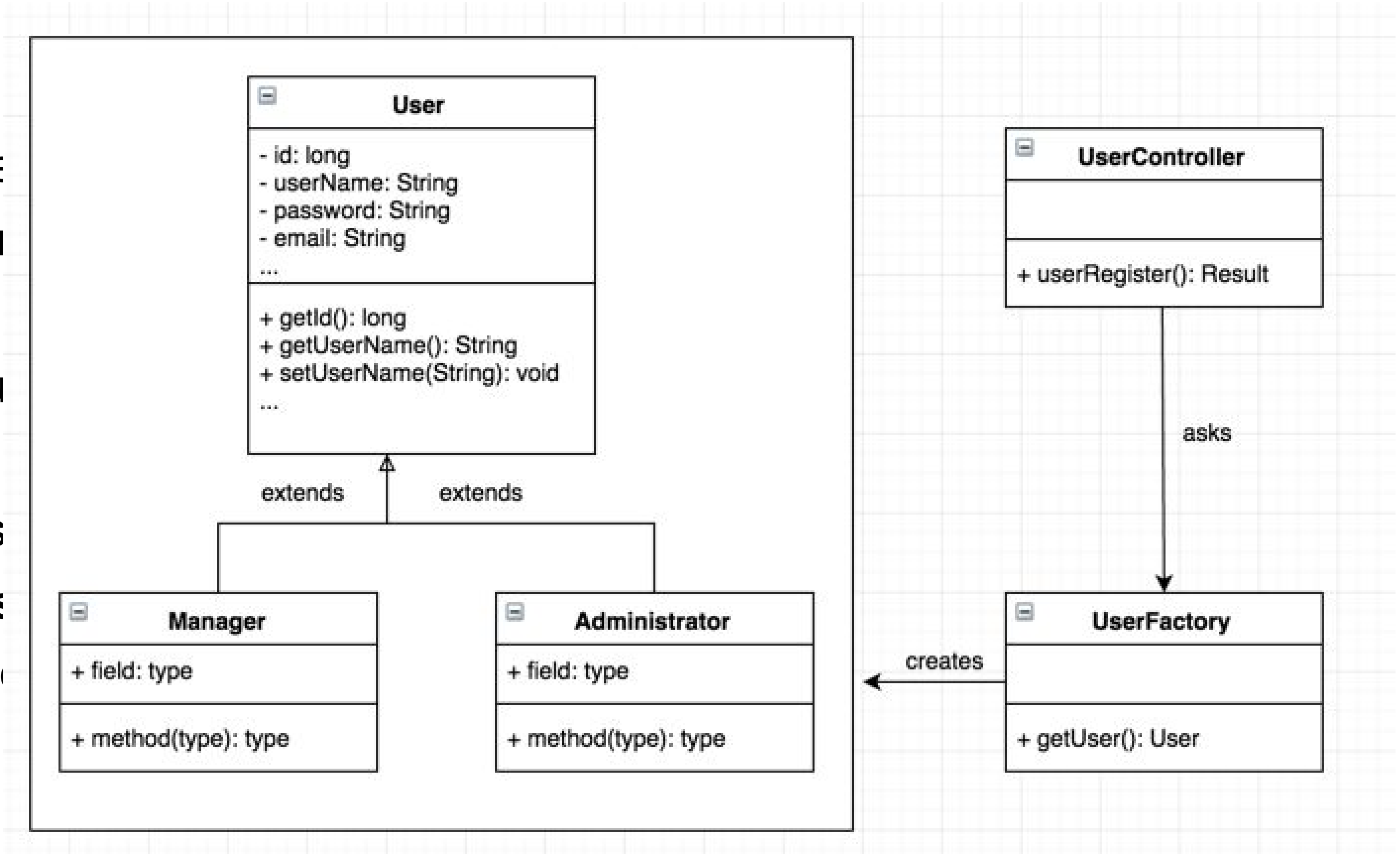
  Act as a **su**

- **Cons:**

  Sometimes

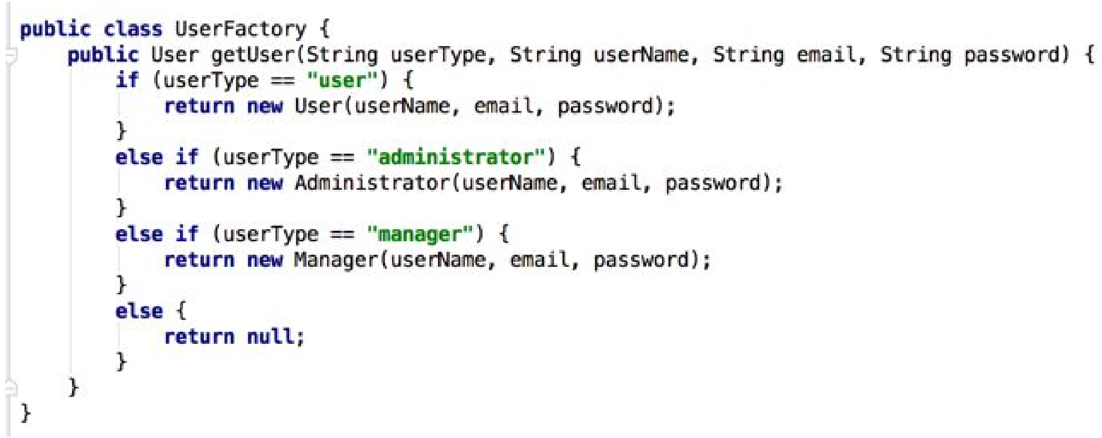
- **Where to us**

  Register/cr



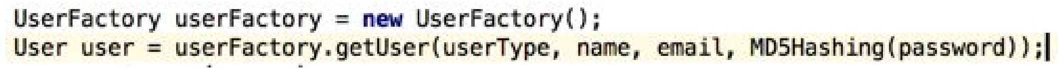
Factory

- **How:**
  - Define a UserFactory



```java
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.



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

Flyweight

- **Definition:**

  Flyweight pattern is primarily used to reduce the nu

- **Pros:**

  Decrease memory footprint and increase performa

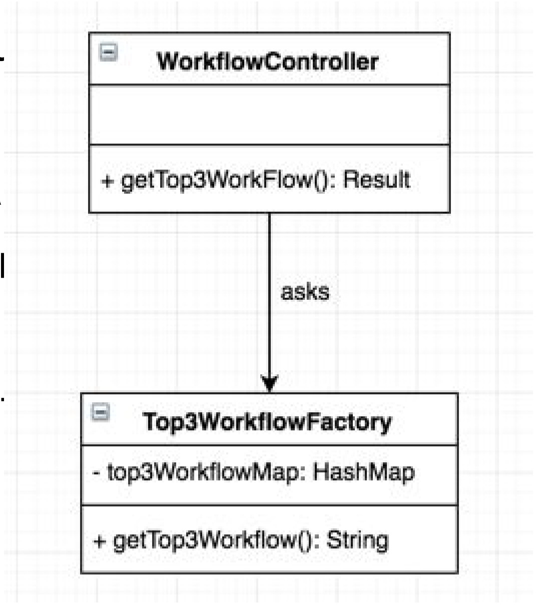
  by **reusing** already existing similar objects when cı

- **Cons:**

  Flyweight is not preferred when objects being crea

- **Where to use in CMDA:**

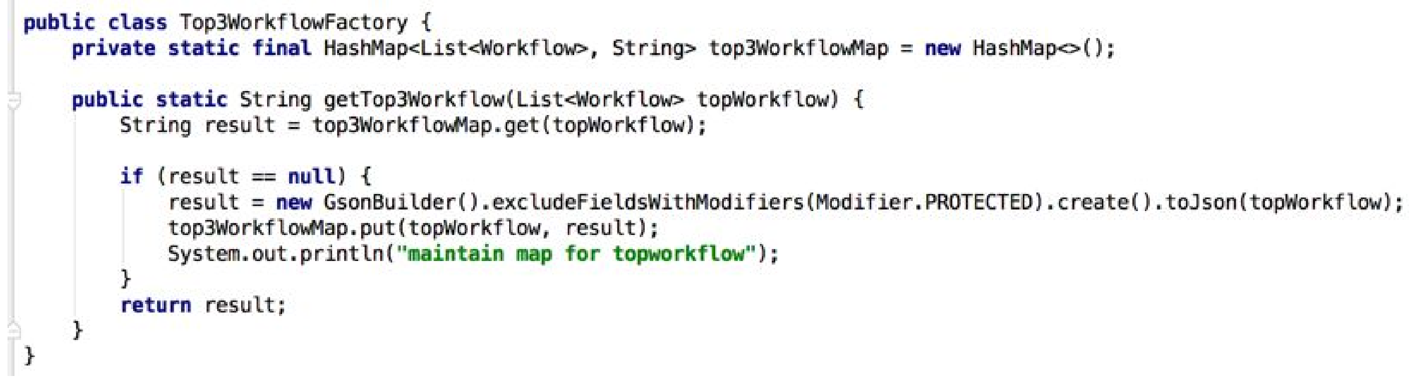
  Create top-3 workflow object, quite stable.


WorkflowController
+ getTop3WorkFlow(): Result
asks
Top3WorkflowFactory
- top3WorkflowMap: HashMap
+ getTop3Workflow(): String

Flyweight

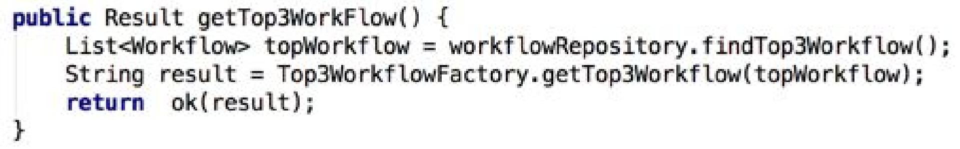
- **How:**

  - Define a Top3WorkflowFactory with a HashMap.



```java
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.



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

Façade

- **Definition:**

  Hides the complexity of the system and provides an interface to the client.
- **Pros:**

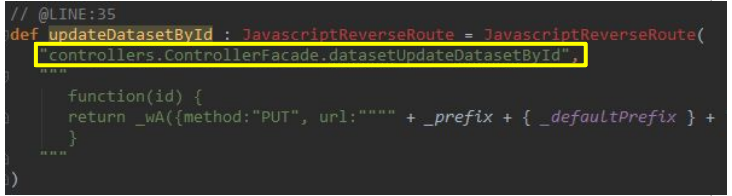
  Hides complexity, clearer code, better maintainability and extensibility.
- **Cons:**

  Can have too many APIs if there are lots of underlying classes.
- **Where to use in CMDA:**

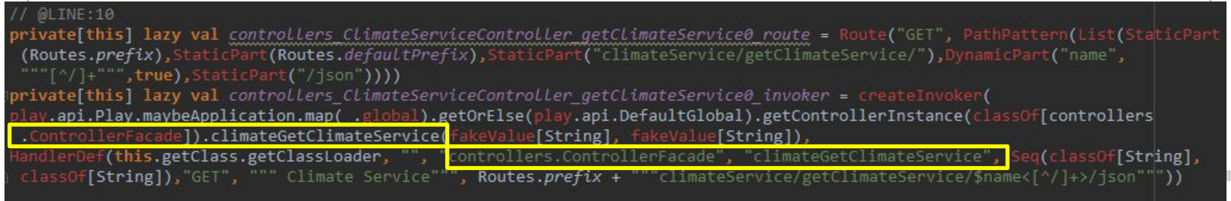
  Provides a single simplified interface for all Controllers.

Façade

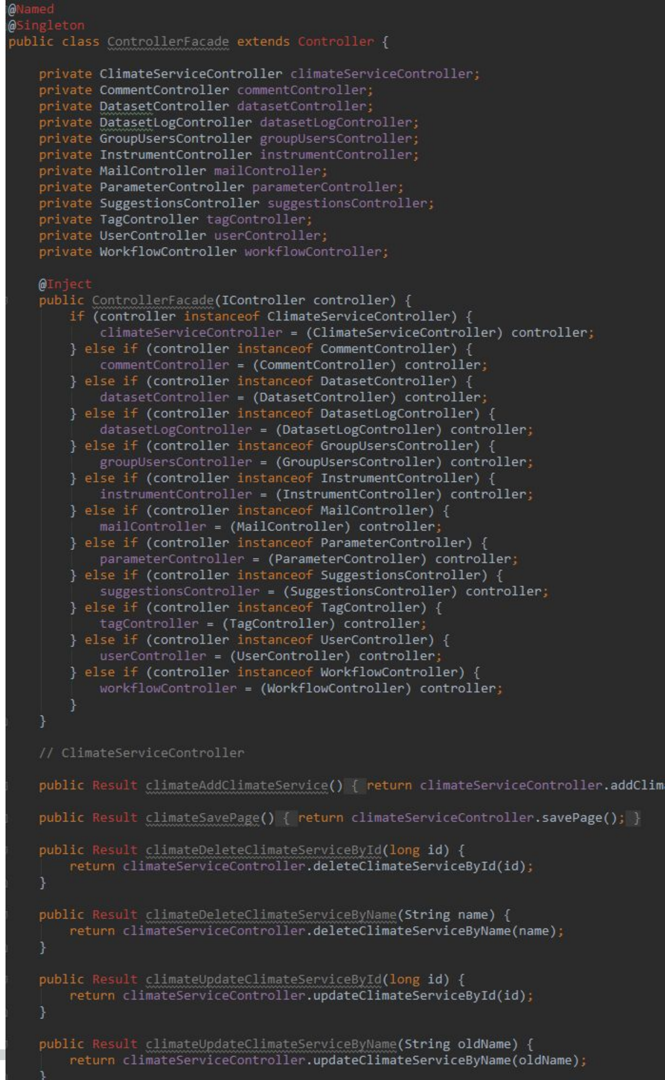
- Provides a single simplified interface for all Controllers
  - Hides complexity
  - Clearer code
  - Better maintainability and extensibility
  - Difficult to apply the new pattern



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



```
// @LINE:10
private[this] lazy val controllers_ClimateServiceController_getClimateService0_route = Route("GET", PathPattern(List(StaticPart
 (Routes.prefix),StaticPart(Routes.defaultPrefix),StaticPart("climateService/getClimateService/"),DynamicPart("name",
 """[^/]+""",true),StaticPart("/json"))))
private[this] lazy val controllers_ClimateServiceController_getClimateService0_invoker = createInvoker(
play.api.Play.maybeApplication.map(_.global).getOrElse(play.api.DefaultGlobal).getControllerInstance(classOf[controllers
 .ControllerFacade]).climateGetClimateService(fakeValue[String], fakeValue[String]),
HandlerDef(this.getClass.getClassLoader, "", "controllers.ControllerFacade", "climateGetClimateService", Seq(classOf[String],
 classOf[String]),"GET", """ Climate Service""", Routes.prefix + """climateService/getClimateService/$name<[^/]+>/json"""))
```



```
@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);
    }
```

Strategy

- **Definition:**

  A class behavior or its algorithm can be change at run time.
- **Pros:**

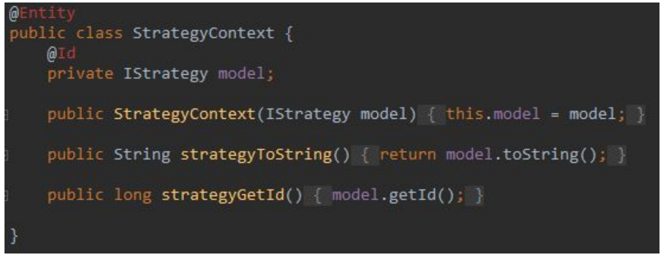
  One API multiple functionalities, better customizability and flexibility
- **Cons:**

  Sometimes unnecessary when we also need to call other APIs
- **Where to use in CMDA:**

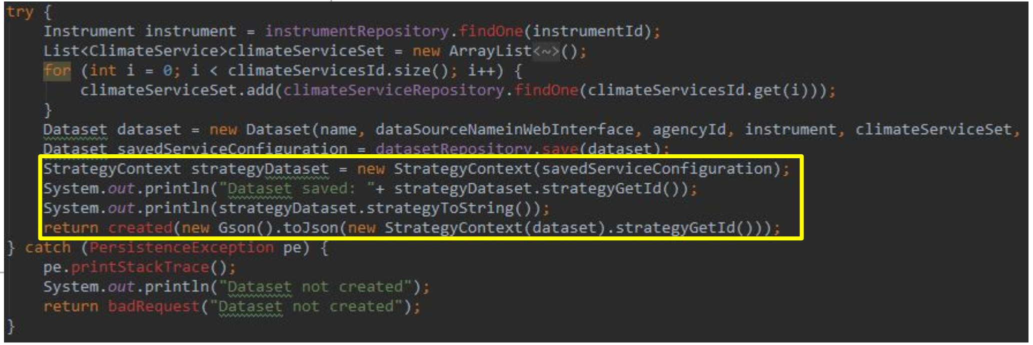
  Method behavior (toString, getId) can/need to be changed at run time in Model classes

Strategy

- Method behavior can be changed at run time in Model classes
  - Using one Strategy class instead of calling different methods from different classes
  - Better customizability, flexibility, maintainability, and reusability
  - Can only be used for methods in common, have to use original classes for other methods



```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(); }

}
```



```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");
}
```

Builder

- **Definition:**

  Builder design pattern is to separate the construction of a complex object from its representation

- **Pros:**

  Making the same construction process create different representations

  Provides control over steps of construction process

  Allows you to vary a product’s internal representation

- **Cons:**

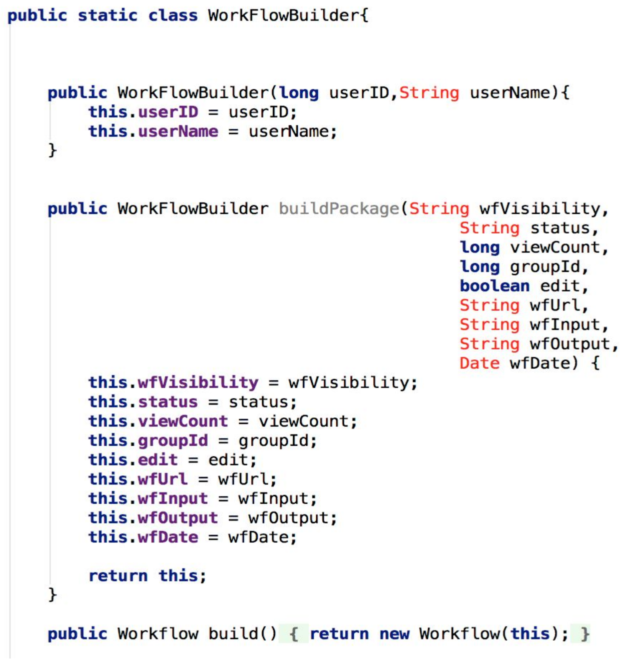
  Builder is not preferred when the construction of the object is simple

- **Where to use in CMDA:**

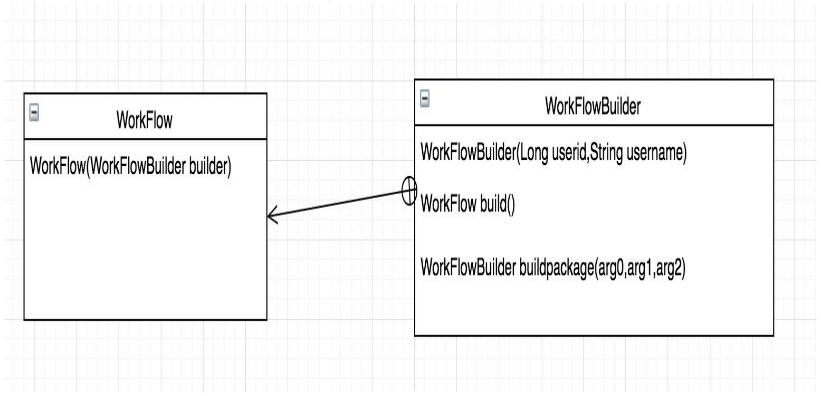
  Create an **nested static class** **WorkFlowBuilder** in WorkFlow Class

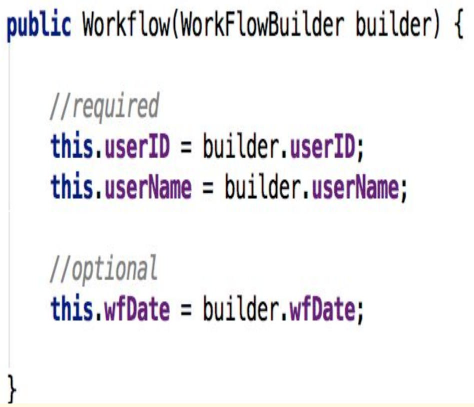
Builder

- **How:**

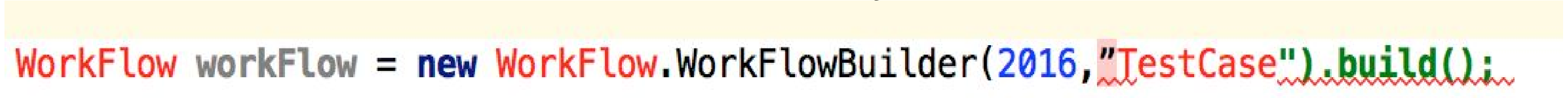


```java
public static class WorkFlowBuilder{


    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); }
```


WorkFlow
WorkFlow(WorkFlowBuilder builder)
WorkFlowBuilder
WorkFlowBuilder(Long userid,String username)
WorkFlow build()
WorkFlowBuilder buildpackage(arg0,arg1,arg2)



```java
public Workflow(WorkFlowBuilder builder) {

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

    //optional
    this.wfDate = builder.wfDate;

}
```



```java
WorkFlow workFlow = new WorkFlow.WorkFlowBuilder(2016,"TestCase").build();
```

Visitor

- **Definition:**

  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
- **Pros:**

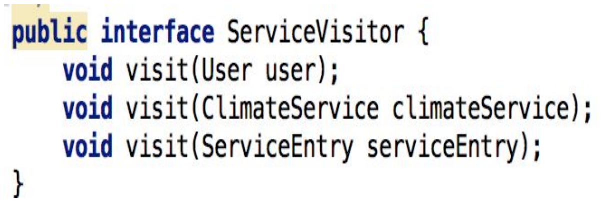
  The nature of the Visitor makes it an ideal pattern to visit the special part of an object without interfere with the whole part
- **Cons:**

  Visitor is not preferred when object is quite simple has no composite parts
- **Where to use in CMDA:**

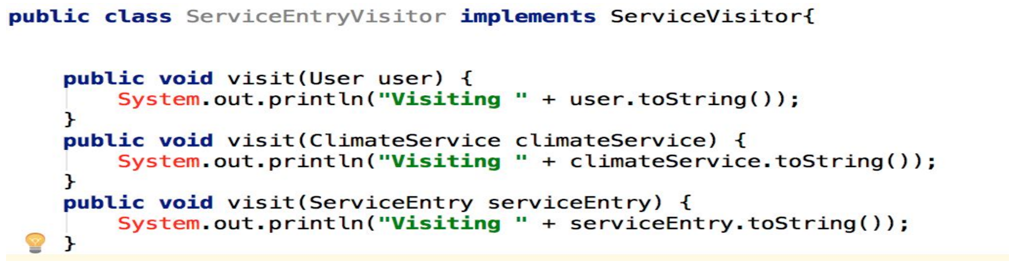
  Create an interface ServiceVisitor for ServiceEntry class

Visitor

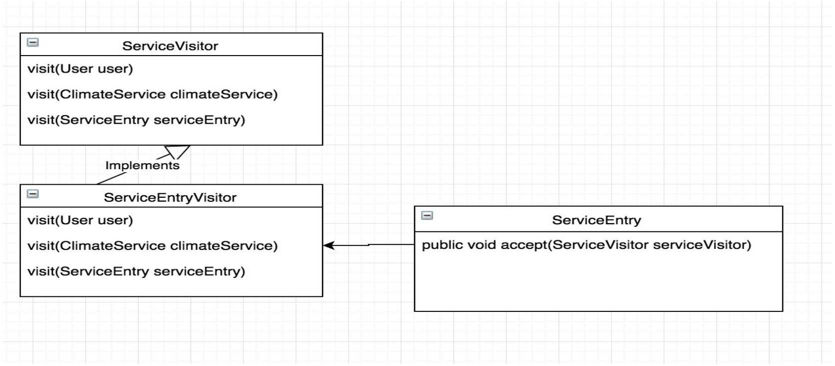
- **How:**

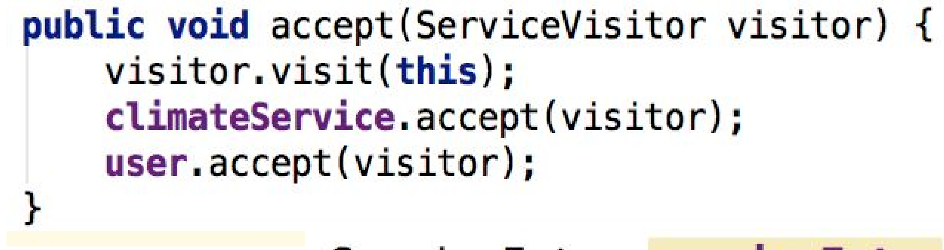


```
public interface ServiceVisitor {
    void visit(User user);
    void visit(ClimateService climateService);
    void visit(ServiceEntry serviceEntry);
}
```

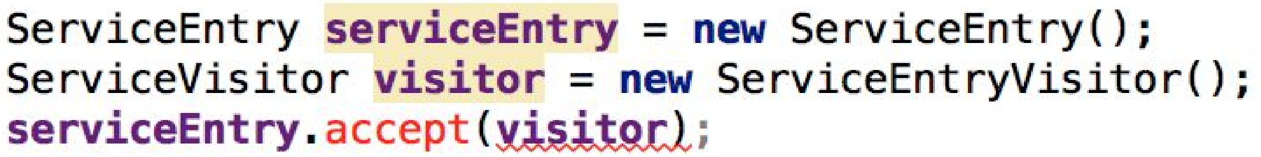


```
public class ServiceEntryVisitor implements ServiceVisitor{

    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());
    }
```


ServiceVisitor
visit(User user)
visit(ClimateService climateService)
visit(ServiceEntry serviceEntry)
Implements
ServiceEntryVisitor
visit(User user)
visit(ClimateService climateService)
visit(ServiceEntry serviceEntry)
ServiceEntry
public void accept(ServiceVisitor serviceVisitor)



```
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);
```

Template

- **Definition:**

  A behavioral design pattern that defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses.

- **Pros:**

  Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm.

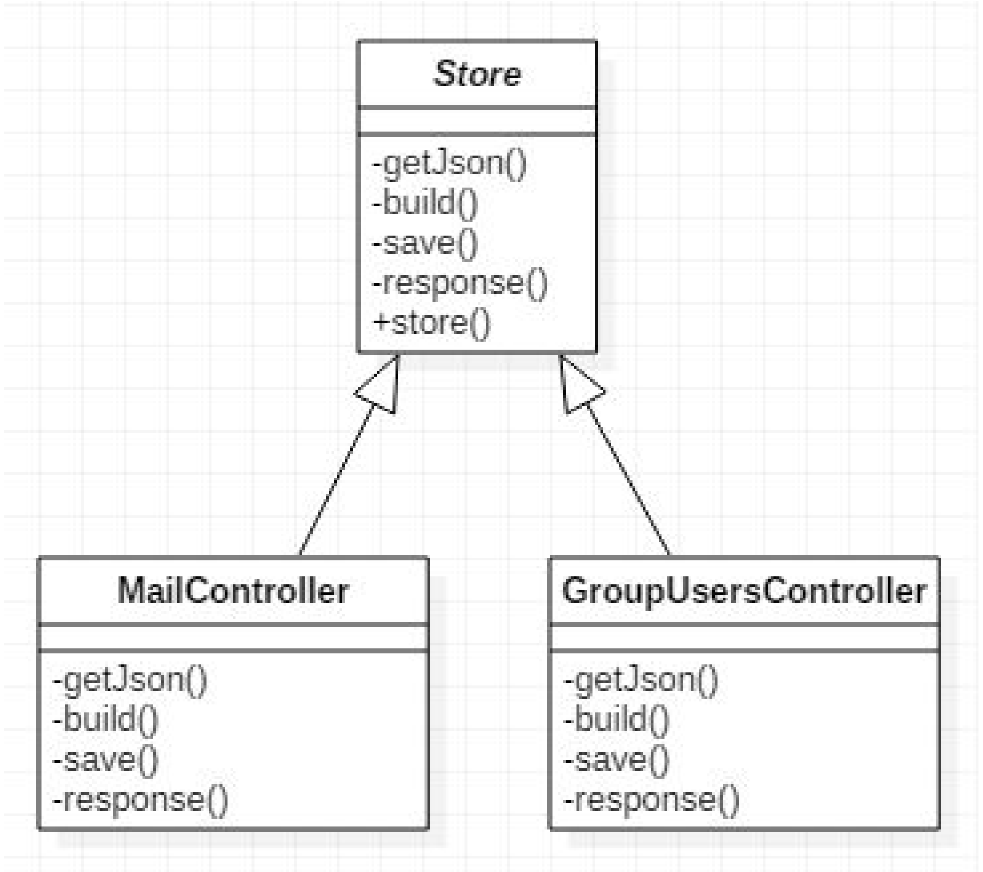
- **Cons:**

  Inheritance bring strong coupling relation between classes

- **Where to use in CMDA:**

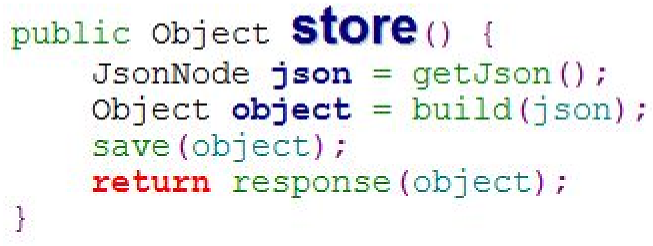
  MailController and GroupUsersController extends Store

Template

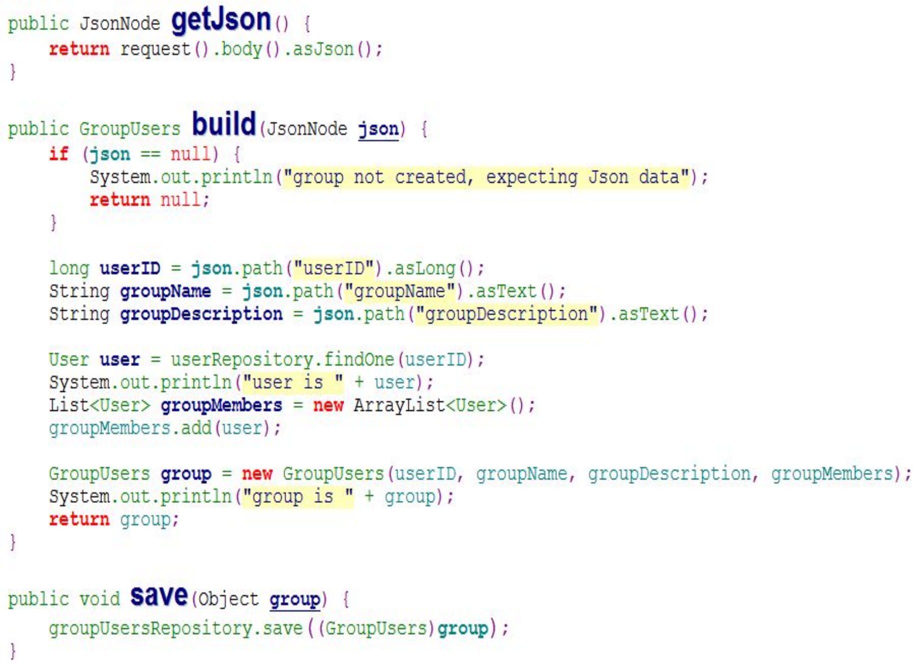

Store
-getJson()
-build()
-save()
-response()
+store()
MailController
-getJson()
-build()
-save()
-response()
GroupUsersController
-getJson()
-build()
-save()
-response()

Template

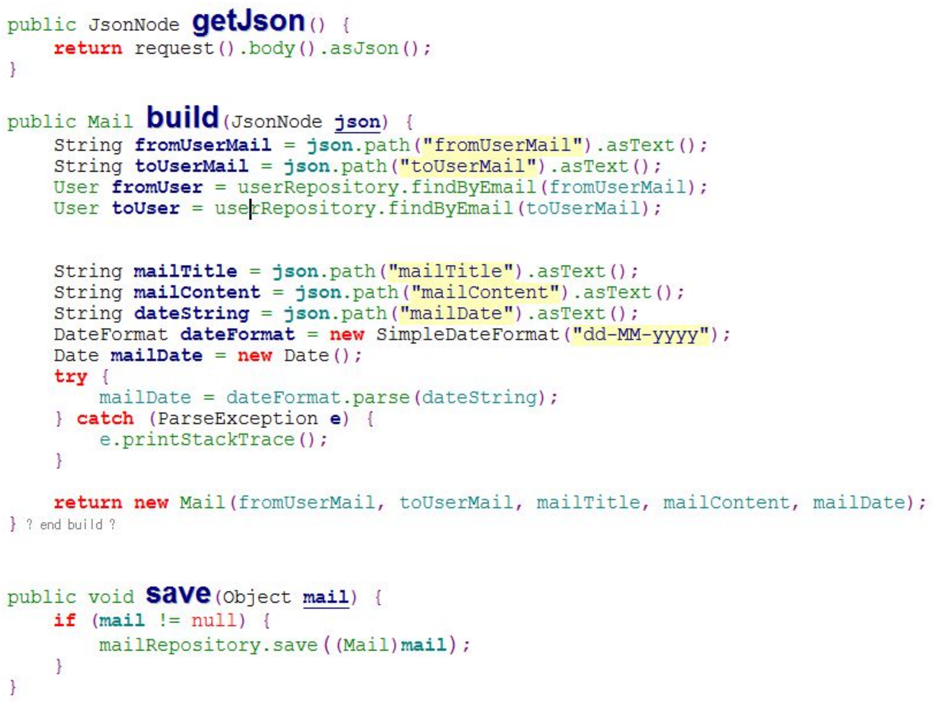
- **How:**



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



```java
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);
}
```



```java
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);
    }
}
```

Command

- **Definition:**

  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 object that owns the method and values for the method parameters.

- **Pros:**

  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.

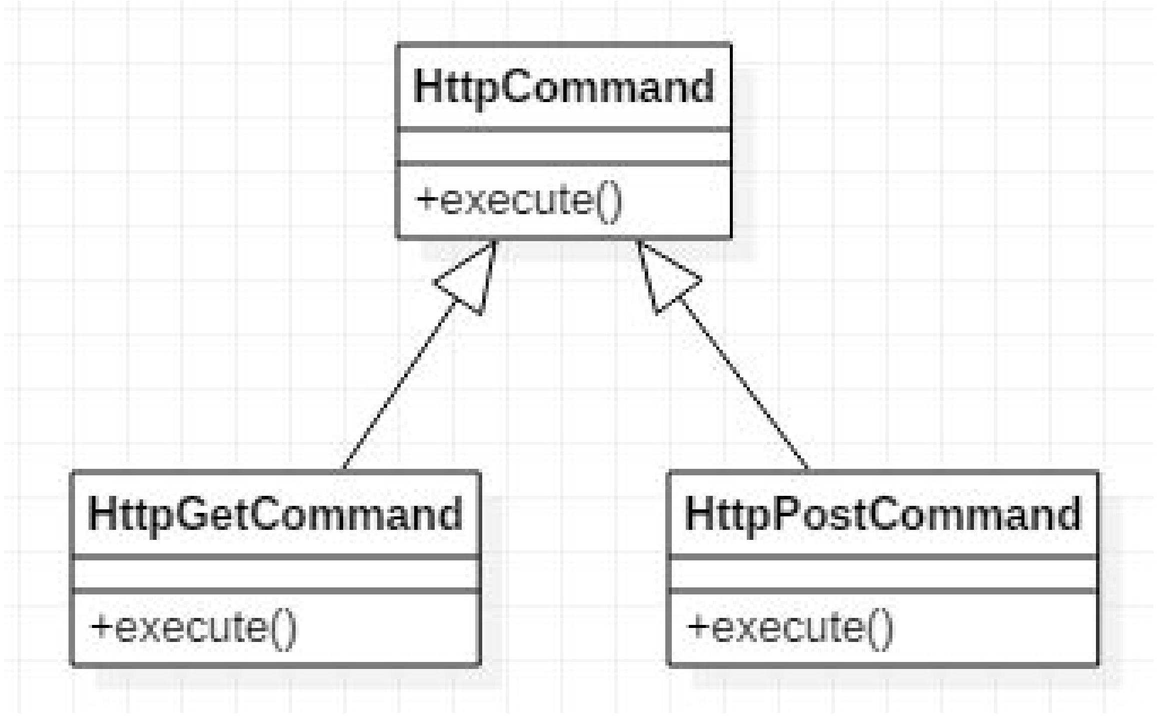
- **Cons:**

  Command Pattern is not preferred when delegation is not required

- **Where to use in CMDA:**

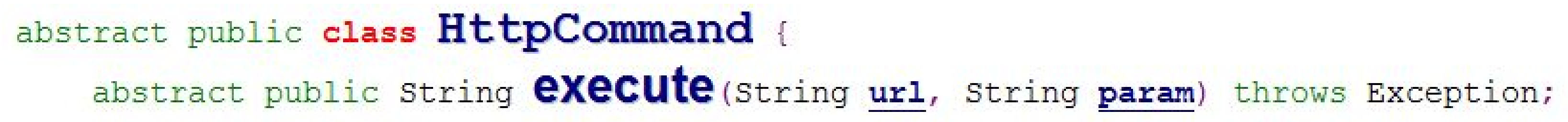
  HTTP Helper

Command

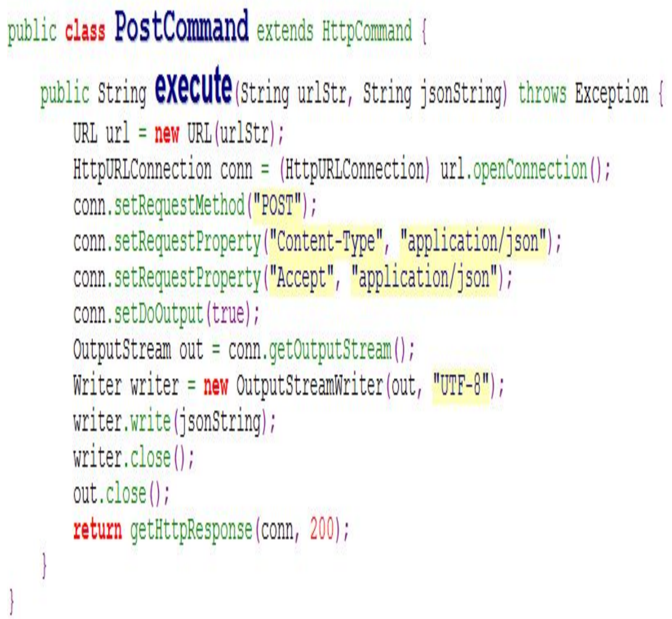

HttpCommand
+execute()
HttpGetCommand
+execute()
HttpPostCommand
+execute()

Command

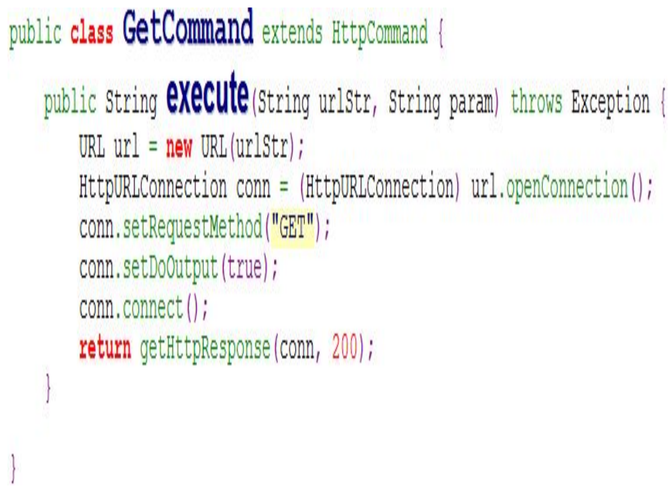
- **How:**



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



```
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);
    }

}
```

Conclusion

- Benefits:

  Provide a stronger **reusability**, **extensibility**, and **maintainability** for the system, and make it **well-structured** by applying some useful design patterns.

- Constraints:
  - Increased effort to apply
  - More classes and interfaces to maintain
  - Poor performance due to indirection

- Suggestion:

  Try to apply more useful design patterns after carefully analyzing their **tradeoff**.

Questions?