The Memo.

Members

- Thanakit Phanthawit (st120695)
- Rangtiwa Fordswngnoen (st120534)
- Nang San Hom (st120225)

Role of each member

- Thanakit Phanthawit : develop the project using “Facade” Design Pattern
- Rangtiwa Fordswngnoen : develop the project using “Factory” Design Pattern
- Nang San Hom : develop the project using “Builder” Design Pattern

The patterns which we used.

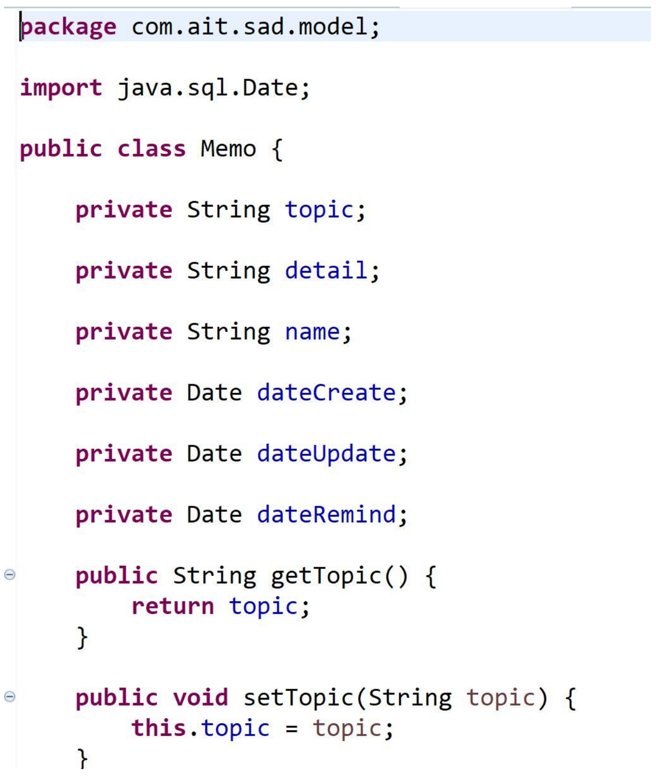
1. Builder: It separates object construction from its representation.
2. Factory: It creates an instance of several derived classes and lets a class defer instantiation to subclass.
3. Facade: It defines a higher-level interface that makes the subsystem easier to use.

When to use this design patterns.

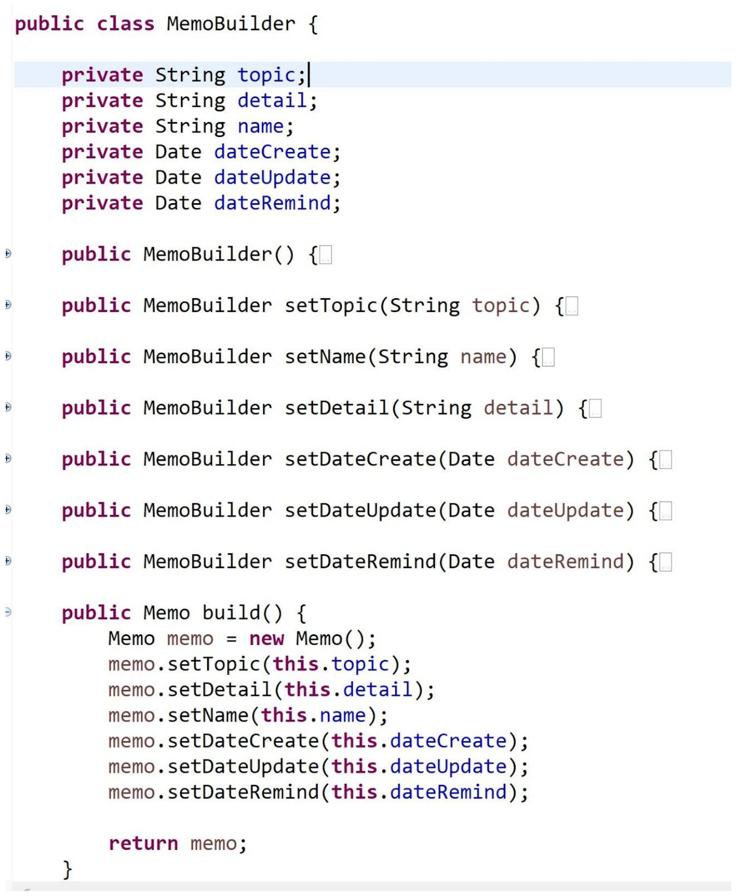
1. Builder: when the objects that you need to create are complex, made up of several sub-objects or require an elaborate construction process
2. Factory: when several possible classes need to be returned that share a common super class.
3. Facade: when working with a large number of interdependent classes, or with classes that require the use of multiple methods.

Implementation

1. Builder: We create the ‘MemoBuilder’ class to create instance of Memo object having many constructors in easiest way.

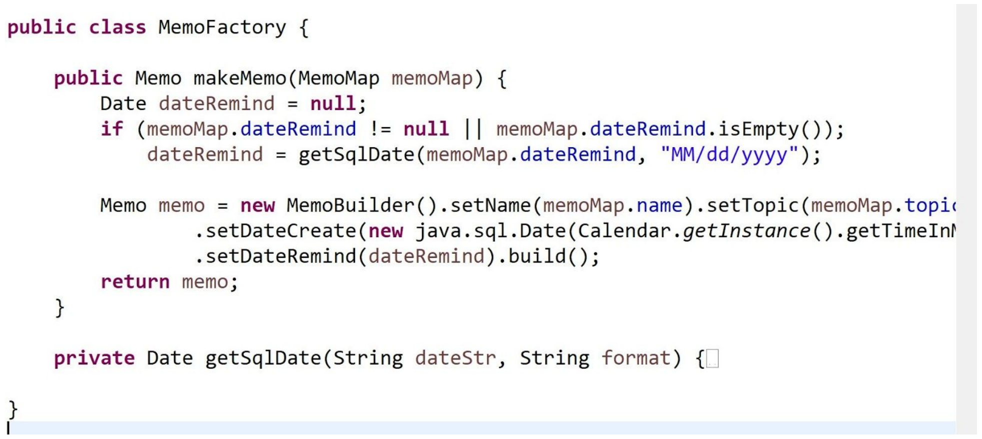


```java
package com.ait.sad.model;

import java.sql.Date;

public class Memo {

    private String topic;

    private String detail;

    private String name;

    private Date dateCreate;

    private Date dateUpdate;

    private Date dateRemind;

    public String getTopic() {
        return topic;
    }

    public void setTopic(String topic) {
        this.topic = topic;
    }
```



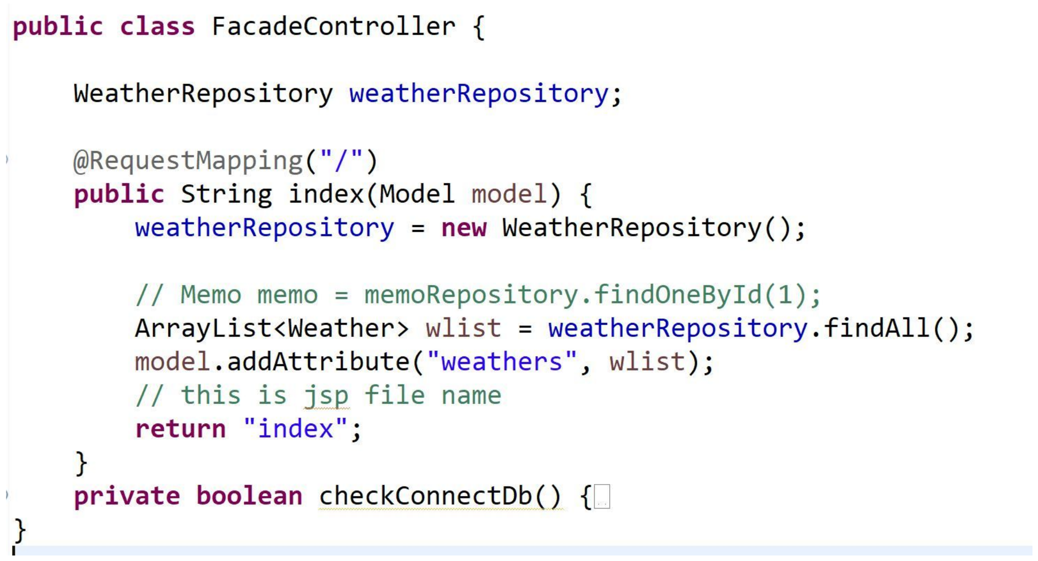
```java
public class MemoBuilder {

    private String topic;
    private String detail;
    private String name;
    private Date dateCreate;
    private Date dateUpdate;
    private Date dateRemind;

    public MemoBuilder() {

    public MemoBuilder setTopic(String topic) {

    public MemoBuilder setName(String name) {

    public MemoBuilder setDetail(String detail) {

    public MemoBuilder setDateCreate(Date dateCreate) {

    public MemoBuilder setDateUpdate(Date dateUpdate) {

    public MemoBuilder setDateRemind(Date dateRemind) {

    public Memo build() {
        Memo memo = new Memo();
        memo.setTopic(this.topic);
        memo.setDetail(this.detail);
        memo.setName(this.name);
        memo.setDateCreate(this.dateCreate);
        memo.setDateUpdate(this.dateUpdate);
        memo.setDateRemind(this.dateRemind);

        return memo;
    }
```

2. Factory: We create the ‘MemoFactory’ class to create an Memo object without exposing the creation logic to the client and refer to the newly-created object using a common interface.

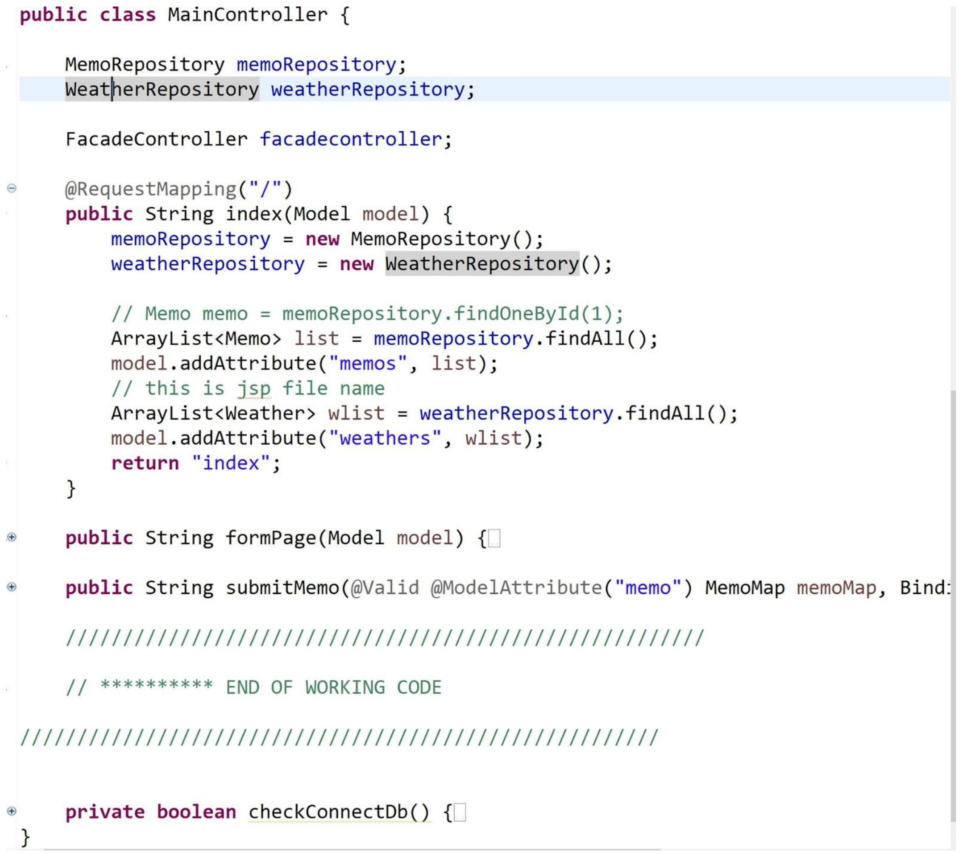


```
public class MemoFactory {

    public Memo makeMemo(MemoMap memoMap) {
        Date dateRemind = null;
        if (memoMap.dateRemind != null || memoMap.dateRemind.isEmpty());
            dateRemind = getSqlDate(memoMap.dateRemind, "MM/dd/yyyy");

        Memo memo = new MemoBuilder().setName(memoMap.name).setTopic(memoMap.topic
                .setDateCreate(new java.sql.Date(Calendar.getInstance().getTimeInM
                .setDateRemind(dateRemind).build();
        return memo;
    }

    private Date getSqlDate(String dateStr, String format) {

}
```

3. Facade: We create ‘FacadeController’ class to encapsulate subsystems behind a simple interface.

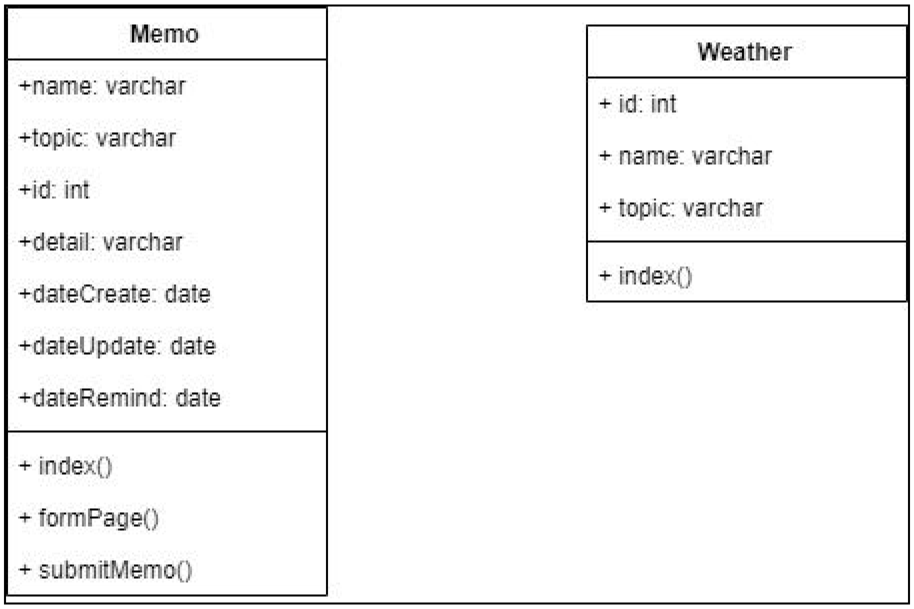


```
public class FacadeController {

    WeatherRepository weatherRepository;

    @RequestMapping("/")
    public String index(Model model) {
        weatherRepository = new WeatherRepository();

        // Memo memo = memoRepository.findOneById(1);
        ArrayList<Weather> wlist = weatherRepository.findAll();
        model.addAttribute("weathers", wlist);
        // this is jsp file name
        return "index";
    }
    private boolean checkConnectDb() {
}
```



```
public class MainController {

    MemoRepository memoRepository;
    WeatherRepository weatherRepository;

    FacadeController facadecontroller;

    @RequestMapping("/")
    public String index(Model model) {
        memoRepository = new MemoRepository();
        weatherRepository = new WeatherRepository();

        // Memo memo = memoRepository.findOneById(1);
        ArrayList<Memo> list = memoRepository.findAll();
        model.addAttribute("memos", list);
        // this is jsp file name
        ArrayList<Weather> wlist = weatherRepository.findAll();
        model.addAttribute("weathers", wlist);
        return "index";
    }

    public String formPage(Model model) {

    public String submitMemo(@Valid @ModelAttribute("memo") MemoMap memoMap, Bind

    ////////////////////////////////////////////////////////

    // ********** END OF WORKING CODE

////////////////////////////////////////////////////////


    private boolean checkConnectDb() {
}
```

The UML of the app architecture.



Should I be obsessed with Design Patterns and always apply when possible? When should one apply or not apply Design Patterns?

Design patterns should not be always apply since it makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions. We should apply them when designing an complex application or system with a huge amount of codes. Design patterns should not be apply when design on a simple application and implementation is not needed for a long term.

Are you satisfied with your code implementations? Discuss in terms of future maintainability. (Imagine your app will grow to 10 million users)

We have not satisfied yet as we should using the framework for the part of the code that connecting to database, need to add more services and provide updates, modifications, bug fixes, patches and additional features to existing software solutions to increase performance.

Research what is “anti-patterns”

An “anti-pattern” is a pattern that tells how to go from a problem to a bad solution.Identifying that bad practices can be as valuable as identifying good practices. It is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.

Give two examples of anti-patterns

1. Spaghetti : Spaghetti code is like the noodles. Iit is very long. Although the noodles are delicious, code the longer it gets is not.
2. Interface bloat: Making an interface so powerful that it is extremely difficult to implement.