COMP319 Object Pattern Notes

MVC Model/View/Controller

This is an architectural pattern which splits the code into 3 parts, model, view and controller.

Model

This stores application data as well as handle all the business logic, rules (e.g. who can access a student's transcript) for the system. It is also responsible for validation (can also be in controller), data persistence, application state (for example keeping a reference to a user’s shopping cart in an online session).

View

This part of the code renders the data to a format that can be presented to the user, for example for a web application the view code would generate HTML code that was sent to the user’s browser.

Controller

This part of the code interprets user input (such as mouse clicks or keyboard input) and sends it to the model. For GUI interfaces, each on screen widget capable of input, has typically an associated piece of controller code.

MVC benefits

Code security
Makes it easier to change the user interface, (the VC) code without having the modify the model code. This is because the VC code can be changed without exposing the critical business logic to change.

Multiple interfaces
The MVC architecture makes it simpler to support multiple interfaces to the application. The module can present a standard public interface that the different VCs can be connected into. Using MVC the software can be developed by two separate teams, one’s with skills in GUI and others in skills in working with database technologies and the application area.

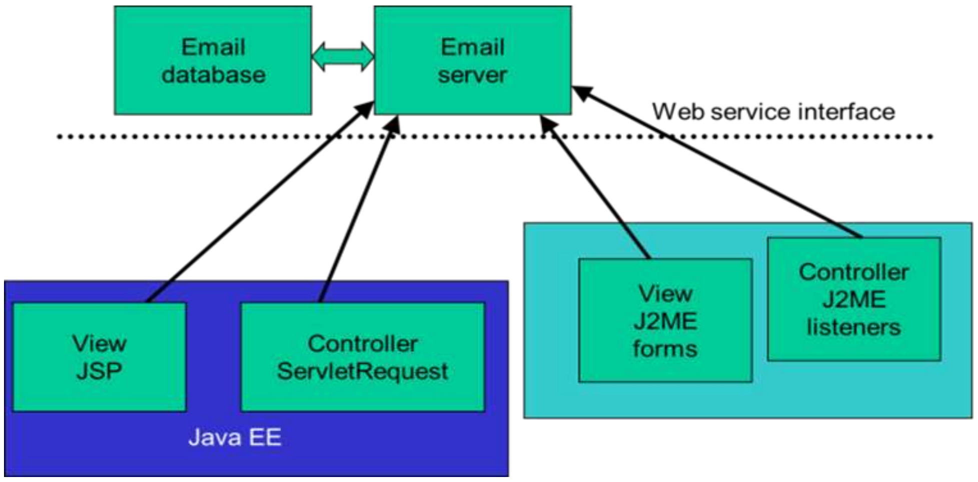

Figure 1 MVC Example Architecture

MVC web-mail work flow example

See figure 1, for archtecture. The user goes to the home URL for the mail service and the VIEW sends a user a login page in Javascript/HTML to the user's browser. The view code can render different versions of the HTML or use CSS to work with different sizes screens etc. The user types in their user/name and password and presses the submit button. The controller software (Javascript+servlet) intercepts the submitted Form page, constructs a Login command object and sends it to the Model command handling code. The Model checks the user's credentials against the database and then makes a request to the view If the user's credentials fail, the VIEW is requested to generate a failed login page. If the login is successful, the model code retrieves the user's current email inbox from the database and makes a request to the View to display this data to the user. The VIEW generates the HTML which is sent back to the user's browser.

Memento

The memento pattern is a software design pattern that provides the ability to restore an object to a previously saved state, this can be used for example to do undo via functions. The memento does not allow external classes to view the state but keeps it encapsulated.

The memento pattern consists of 3 parts an originator, this is the class that has internal state, a memento object which stores the state and a care taker which manipulates the originator object.

The originator can be asked to generate a memento class for its current state and restore its state from a memento object. The importance of the memento pattern is that is allows the object to be persisted without breaking its encapsulation.

Simple code example for Memento

```
class TextBox {

        private int maxlen=0;
        private String text=””;

        class Memento {    // memento defined as inner class

            private String state="";

            public Memento(String state) {

                this.state=state;

            }

            private String getSavedState() {

                return(state);

            }
        }

        public Memento saveToMemento() {

            return(new Memento(“”+maxlen+”:”+text);

        }

        public void restoreFromMemento(Memento memento) {

            String state= memento.getSavedState();

            maxlen=Integer.parseInt(state.subString(0,state.indexOf(“:”)));

            text= state.subString(state.indexOf(“:”)+1);

        }

    }
```

Chain of responsibility pattern

In this pattern a number of classes work together as a chain to work out which one of them is to handle a message or process a command or request. If the class doesn’t want to handle the message, it passes the messages down to next class in the chain. A class also could handle a message and pass it down the list as well for other classes to handle.

An example of chain of responsibility would be logging in an application, each logger can send message to the next logger in the list and depending on the level of the message and the current logging level the logger would deal with the message. The loggers can be organized into a linked list.

Simple code example for Chain of responsibility

```
abstract class Logger {
protected Logger next;
        public Logger(int level) {
                this.logger_level=level;
                setNext();
        }
        private void setNext() {
                if (lastLogger!=null) {
                        lastLogger.next=this; // add this into chain
                }
                lastLogger=this;
        }

public void message(String msg, int priority) {
                if (priority <= logger_level) {
                        writeMessage(msg);
                }
                if (next != null) {
                        next.message(msg, priority);
                }
        }
}
```

Singleton

A singleton is a class which only supports a single instance. The singleton class has to control its own creation and not allow client code to make new instances external to the class. This allows multiple threads to share the data in the class instance.   The control creation of instances of the class, the singleton class keeps its constructor private and gives access to the instance via a static method.

One example of a singleton usage would be a helper object which is used to connect to an SQL database.

```
public class DatabaseHelper {

    private static final DatabaseHelper INSTANCE = new DatabaseHelper();


    private DatabaseHelper() {}


    public static DatabaseHelper getInstance() {

        return INSTANCE;

    }

}
```

In the example code, the instance of the singleton instance is created when the class definition is loaded. Note the use of the private keyword on the class constructor this makes it impossible for clients of this class to create instances of it using the new keyword.

Double lock checking

Sometimes it is important to keep access to certain code exclusive. This means only 1 thread can run the code at once. One way of doing this is to use the key word synchronous in Java.

However usage of the keyword synchronous is very slow and therefore we want to be able to keep its usage to a minimum. Double-lock checking first checks that the lock is needed BEFORE applying the synchronous key word. This makes the code run faster and reduces the number of synchronous calls, speeding up the code.

Simple example of Double lock checking in Singleton with lazy creation (instances created at runtime not class load time)

```
public class DbaseConnector2 {

    private static DbaseConnector2 instance = null;

    public static DbaseConnector2 getConnector() {

        if (instance == null) {

            synchronized (DbaseConnector2.class) {

                instance = new DbaseConnector2();

            }

        }

        return instance;

    }

}
```

Factory pattern

The factory pattern is used then an objects exact type is no know at compile time but will ideally be decided later on at run time. Consider the example of processing an image file to create an image object in memory. The image file could be in PNG image, JPG image or a BMP image format. Since the ideally you need to use an appropriate class for each of these images, so you code might look like this.

```
ImageBMP image1=new ImageBMP("image1.bmp");

ImagePNG image2=new ImagePNG("image1.png");

ImageJPG image1=new ImageJPG("image1.jpg");
```

This code is clumsy and of course will not work if we don't know the type of the image file before the program is running.

The answer to this problem is to create an abstract class which supports the general interface of Images but is not specific to any particular type. So it will have methods such as getWidth and getHeight which will apply to all image files.

```
abstract class Image() {

        private int width,height;

        int getWidth() {

                return(width);

        }

        int getHeight() {

                return(height);

        }

}
```

The image types will inherit from this super class, for example.

public class ImagePNG extends Image {

}

public class ImageJPG extends Image {

}

To create the class, a static method is added to the Image abstract class.

```
public abstract class Image {
	private int height,width;

	public int getWidth() {
		return(width);
	}

	public int getHeight() {
		return(height);
	}

	public static Image getInstance(String fileName) throws Exception {
		if (fileName.endsWith(".png")) {
			return( (Image)new ImagePNG(fileName));
		}
		if (fileName.endsWith(".bmp")) {
			return( (Image)new ImageBMP(fileName));
		}
		if (fileName.endsWith(".jpg")) {
			return( (Image)new ImageJPG(fileName));
		}



		throw new Exception("File type not supported for "+fileName);

	}

}
```

Builder pattern

This pattern separates abstract definition of an object from its representation, this can be used to help build things like SQL commands.

One example of this would be a builder which produced SQL query strings or output of XML or HTML.

In this example, different sub-classes could be used to build different types of queries for example, SELECTS, DELETES, INSERTs etc.

The benefits of this are many:

The client of the builder will not have to remember the syntax of the underlying expression.

Accidental mistakes in the output format can be eliminated, this improves code quality.

The builder can be adapted for different syntax formats, for example between MySQL and MySQL.

Extra functionality can be added, for example to produce queries which can be sharded across different tables or to provide a layer of database security.

The builder can validate arguments such as table names to ensure validation before the query is executed on the database.