Tutorial 3 Answers Comp319 Software Engineering

Object patterns

Model View Controller

Splitting the pattern into the model (this stores and manipulates the data and executes all business rules).

View **Displays data to the user on request from the model.**

Controller **Accepts input from the user and pushes input to the model.**

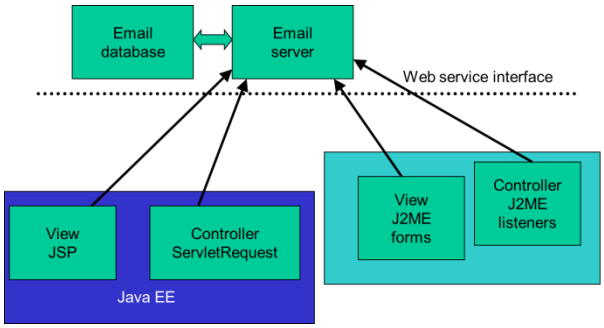
Detail

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.

The model view controller can be used to model most applications, here is an example of a email system.



Detail of process

The user would open their browser and type in the URL of the mail service. This would send a request to the Java EE application which handles the view, which generate login page in html and send it to the mail subscribers browser. The mail subscriber would type in their user name and password into the html page and then click the login button. The controller part of the J2EE application would pick up on the login click and push the request with the username and password to the model part of the application, via the web service interface. The web service

interface would determine the authenticity of the request and send a reply back to the J2EE application, the reply could contain a rejection of the login or could contain the information needed to render the opening page of the email, including details of the users inbox in most cases formatted in XML. The Java EE view part of the application can take the model data returned and format it back into HTML for the user to read.

There are a number of advantages to using the MVC approach.

It is simpler to change the user's interface since we don't need to change the model critical code when doing this. The view and controller code can be changed without the critical business logic undergoing the risk of change. It is also simpler to engineer different interfaces to the same underlying business application. The model can be developed so that is provides a standard interface for example using XML requests and each VC client can be adapted to connect to this interface. MVC also allows you to use separate teams to develop different views and the model code, so staff skilled in different areas can be used efficiently.

Singletons

These are classes which have only 1 instance. This is a constructor object pattern, the instance of the class is made available via a static interface.

This instance of the class can be created in 1 of 2 ways.

On class load, in this case the constructor is called as the class is loaded up by the class loader.

private static SingletonConnector instance=new SingletonConnect();

This is a very simple approach, but makes it harder to have parameters loaded up from some resource or configured dynamically into the software.

On demand or lazy initialization involves only creating the instance when it is required, this has the benefit of allowing the constructor to take arguments and be called as part of a defined start up process.

One example of the use of a singleton pattern is something like a database connector, for example if you wanted to manage a pool of connections to a remote database to improve performance.

Since singletons are shared they need to have thread safety built into their design, since the code can involve multiple threads sharing the same data.

One part of thread safety is the construction of the object itself, it is important that 1 and only 1 copy of the object is created and served to each of the calling threads.

One way of doing this is to make the method creating the object synchronized, since this will ensure only 1 thread can call it at a time. However this is time consuming since synchronized methods are a lot slower (than methods without synchronization). Another approach is where the method uses a double locking approach, this involves only applying the lock as and when it is needed, when the object it being constructed for the 1st time.

Class Adapter interface and providers

This type of structure is useful when wanting to adapt a number of different external APIs to provide a standard interface to the application software.

Examples could be

Different interfaces to different

Database services, SMS services, Card payment services.

In each of these cases the details will be different but the basic service is the same.

Think of an example for the SMS service, we might have 3 different service providers each who provide a different API to connect to.

There are a number of things that we need to do:

Define a standard interface to the service (define in an object interface).
Here is an example of a standard method to send an SMS message.

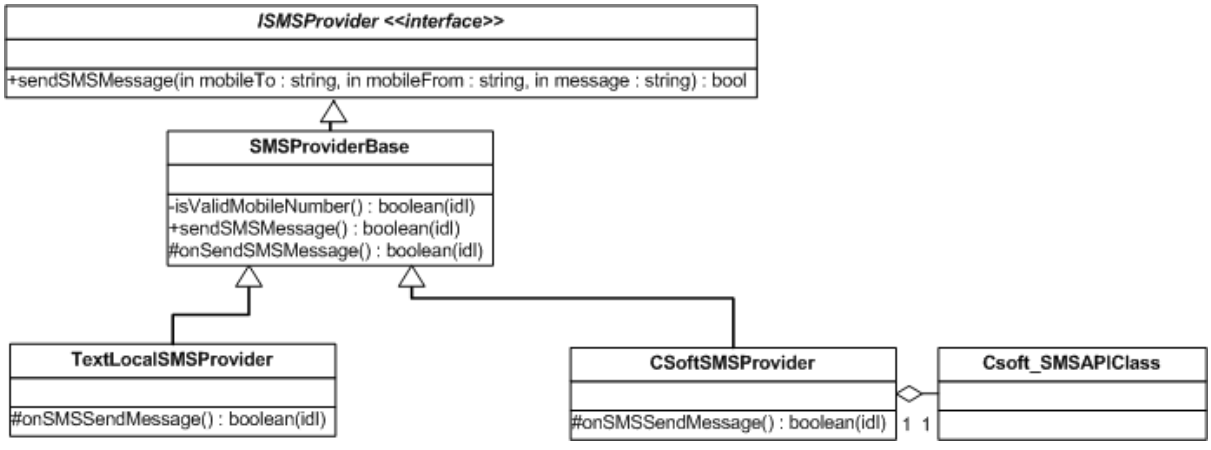
public bool sendSMSMessage(String telephoneTo,String telephoneFrom,String message);

Define a set of standard methods which will be used my all providers, one example for the SMS service could be a mobile number validation routine, this would check that the telephone number provided is valid:

public bool isValidMobileNumber(String telephoneNumber); These standard methods can be stored in an abstract class which implements the interface to the SMS service.

Finally you can have the concrete class, which implement the SMS service for a given provider.

You end up with a class diagram looking like this.



This allows you to connect to all the different providers via 1 standard interface.

Integrating new providers is made simpler since the user only has to implement 1 new concrete provider class each time.

Factory classes

Factory classes are constructor classes that are used to generate instances of a class. The factory class approach has advantage over standard object creation using a constructor, is that is allows the actual object type to be deferred to run-time. For example, imagine you are creating a class which will handle and render image data on a screen. You might want a different class type depending on the image data contained, so for png format you might have a class called ImagePNG for a jpg format you could have a class called ImageJPG. This becomes a problem when creating instances of the classes, since only at runtime you will know the type of the object concerned. The solution is to define an interface, called Image, and this will be the type handled by the calling code, like so:

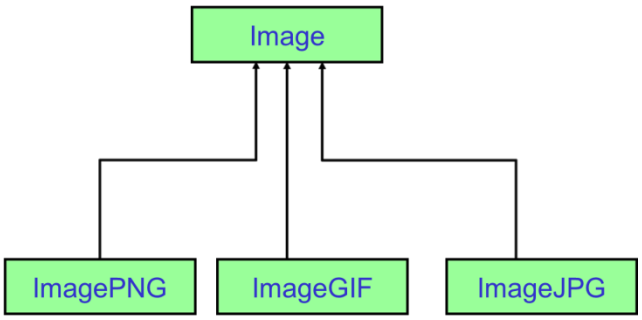
```
public interface Image {

        public Image(String filename) {};

        public drawImage(Graphics g) { };

}
```

Each particular concrete class shall implement this interface, see Figure



We can now declare a static factory method, which will return a different class instance depending on the input parameters.

```
public interface ImageFactory {
```

```
        public static createImage(String fname) throws Exception {
            if (fname.endsWith(“.gif”)) {
                return( (Image) new ImageGIF(fname) );
            }
            if (fname.endsWith(“.png”)) {
                return( (Image) new ImagePNG(fname) );
            }
            if (fname.endsWith(“.jpg”)) {
                return( (Image) new ImageJPG(fname) );
            }
            throw new Exception("Unknown image type for file "+fname);
        }
}
```