

***Introduction of MVC  
structure in J2ME  
client***

March 1, 2006

*WHITE PAPER*

---

## Introduction of MVC structure in J2ME client

By

Motocoder Staff

---

**M**VC (Model-View-Controller) is one of the most classic patterns for UI structure design. The MVC pattern is a way to divide the application, or even part of the application, into three parts: Model, which is the body of the application and includes the business logic; View, which represents the user interface; Controller, whose job is process the user input and system event, delegate the job to model service and update the view accordingly.

MVC structure has different scales, even in a single class; the structure can be implemented with different functions. Several examples will show -- from simple to complex -- how an all-in-one class J2ME[awv1] client transformed to a J2ME[awv2] client with integrated MVC structure; how three classes are organized and how the functions distributed among these classes. The pros and cons in every client type will also be discussed, so as to help the developers select the proper client type.

---

### **All code in one class client**

The concept is simple: just put all the code in one class with the startApp(), pauseApp(), destroyApp() like the HelloWorld example. I sometimes write code in this way to test some features of the handset, but no more than that. Sure it's not even a pattern, but it gives us a way to understand how the MVC pattern is generated from the very beginning.

```
//Code example, items shared in different views
public void commandAction(Command cmd,Displayable disp)
{
    if(cmd == getCommand &&currentView==mainForm) then
        {... }
    else if(cmd == getCommand &&currentView==listForm) then
        {... }
    else {...}
}
```

#### **Pros:**

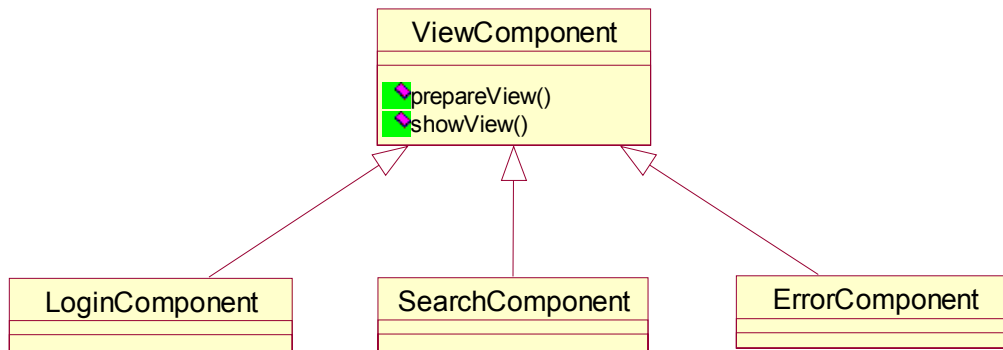
This has the shortest code length, for there's no duplicated code if we abstract them to methods. In fact, some items can even be shared among different views.

#### **Cons:**

With more views, the complex class structure and the jump here and there style in the control logic will confuse most developers.

## View based Component

View base components implements classes in this way: every view (or several similar view) of the application is designed as a class; in a single class, the MVC structure can be implemented with different functions.



**Figure 1:** Class diagram of view based component

```
public class LoginComponent extends ViewComponent implements CommandListener {
    public Display display;
    ...

    // View related functions
    public Displayable getView( ... );
    public Displayable prepareView () { ... }
    public void showView() { ... }

    // Model related functions
    public boolean loginService(String name, String password){
        ...
    }

    // Controller
    public void commandAction(Command c, Displayable s){
        ...
    }
}
```

### Pros:

The structure is simple and straight forward; it consists with the screen view and is easy to understand by the developers. With the super class, we can put some common interfaces in it, and the code will become more reusable.

### Cons:

Some of the view base component may have similar functions; this makes duplicated codes in different classes. Duplicate code is a bad feature of the class design, it makes the application hard to maintain.

## Model-View

The model-view structure is the most common structure for a J2ME<sup>[awv3]</sup> client. The entity data and the business logic functions are taken out as the model classes.



**Figure 2:** Class diagram of view-model structure

```
// View example, use form as super class of a view
class UnitView extends Form implements CommandListener {
    private LoginEngine engine;
    private String result; [awv4]
    ...
    UnitView(String name; LoginEngine engine){
        Super(name);
        this.engine = engine;
    }

    String login(String name, String password){
        result=model.getService(name, password);
        ...
    }
    ...
}

//Model example
class LoginEngine{
    public String getService(String name,String password){
        ...
        return result;
    }
}
```

### Pros:

Two or more view objects could share the functions in a model; this abstracts similar code from different classes.

### Cons:

The drawback of this structure is that the event process function and screen flow logic are distributed in several views. For every view that maintains the relationship with other views, if we have  $N$  view units and every one keeps a relationship with the others, there are  $N*(N-1)$  relationship links, but with a mediator, there are only  $N$  relationship links.

## Single controller MVC

The single controller, which is also called the system controller, is responsible for receiving and processing all the system events. Normally, the controller will delegate most work to the model and just control the screen flow and process and distribute the system event. In Sun's famous J2ME[awv5] blueprint, [smart ticket](#), a large switch function is used as the core of the single controller.

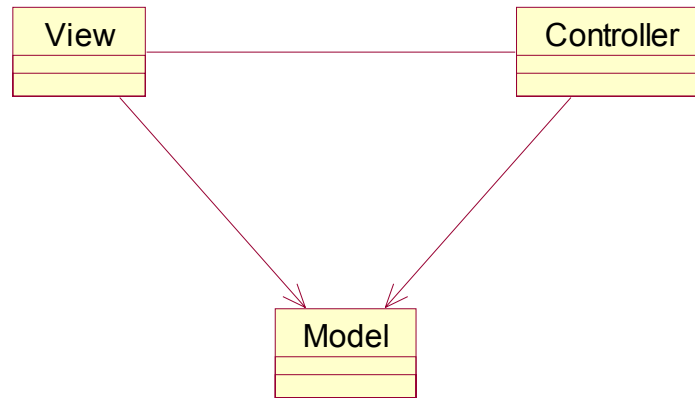


Figure 3: Class diagram of MVC model

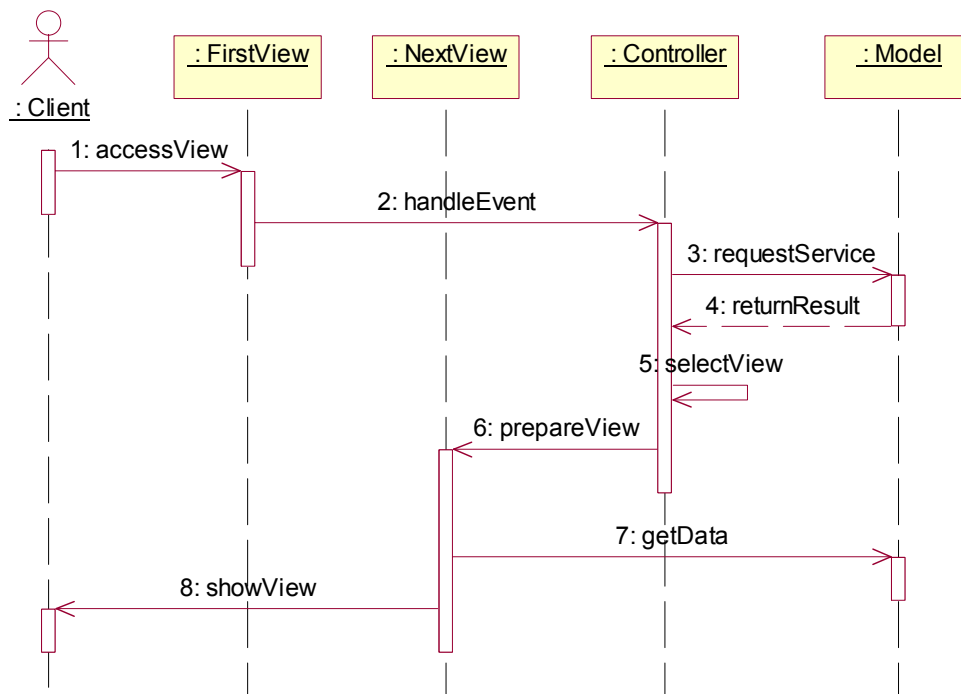


Figure 5: Sequence diagram of MVC objects

```
// login view
public class LoginView extends Form implements CommandListener {
    Controller uniController;
    Display display;
    MVCMidlet mvcMidlet;
```

```

TextField userNameField;
TextField passWordField;
Command loginCommand;

public LoginView(String title){
    super(title);
    uniController=Controller.getInstance();
    ...
    this.append(userNameField);
    this.append(passWordField);
    this.addCommand(loginCommand);
}

public void prepareView(Display display){
    setCommandListener(this);
    this.display=display;
}

public void showView(){
    display.setCurrent(this);
}

public void commandAction(Command c, Displayable d){
    if(c==loginCommand){
        Event event = new Event();
        event.setByName("userName",userNameField.getString());
        event.setByName("password",passWordField.getString());
        uniController.handleEvent(Event.LOGIN_EVENT,event);
    }
    ...
}

}

//Controller
public class Controller {
    private static Controller instance;
    private Display display;
    private LoginEngine loginEngine;
    private HomeView homeView;
    private LoginView loginView;
    private ErrorView errorView;
    ...

    private Controller() {
    ...
    }

    public void init(MIDlet midlet){
        this.display = Display.getDisplay(midlet);

```

```

    }
    ...
}

public static synchronized Controller getInstance(){
    if(instance == null){
        instance = new Controller();
    }
    return instance;
}

public void handleEvent(int eventID, Event e){
    switch(eventID){
        case LOGIN_EVENT:
            String passWord = e.getByName("password");
            String userName = e.getByName("userName");
            Boolean result = loginEngine.loginService(userName,password);
            if(result==true){
                homeView.prepareView(display);
                homeView.showView();
            }
            else{
                errorView.prepareView(display,"loginError");
                errorView.showView();
            }
            break;
        ...
    }
}

//login model
public class LoginEngine {
    public boolean loginService(String userName, String passWord){
        boolean result = ...
        ...
        return result;
    }
}

```

**Pros:**

In the standard MVC structure, business logic (**model**), representation logic (**view**) and event processing logic (**controller**) were elegantly separated. Modification in one part is isolated from the others. This makes the whole application more extensible and scalable.

**Cons:**

With the application size increasing, more and more page flow logic is concentrated in the single controller. The controller becomes very large and hard to maintain. Maybe it's time to divide it into several pieces. The rigorous separation between the model, view and controllers can sometimes make it more difficult for debugging and testing.

## Multi controller MVC

Multi controller structure is usually based on use cases, that is to say, one controller only processes the events in one or several related use cases. With the division of the single controller, the whole application could be divided into several packages with similar structure. This is typical for enterprise level applications, but is rarely used for J2ME[awv6] applications.

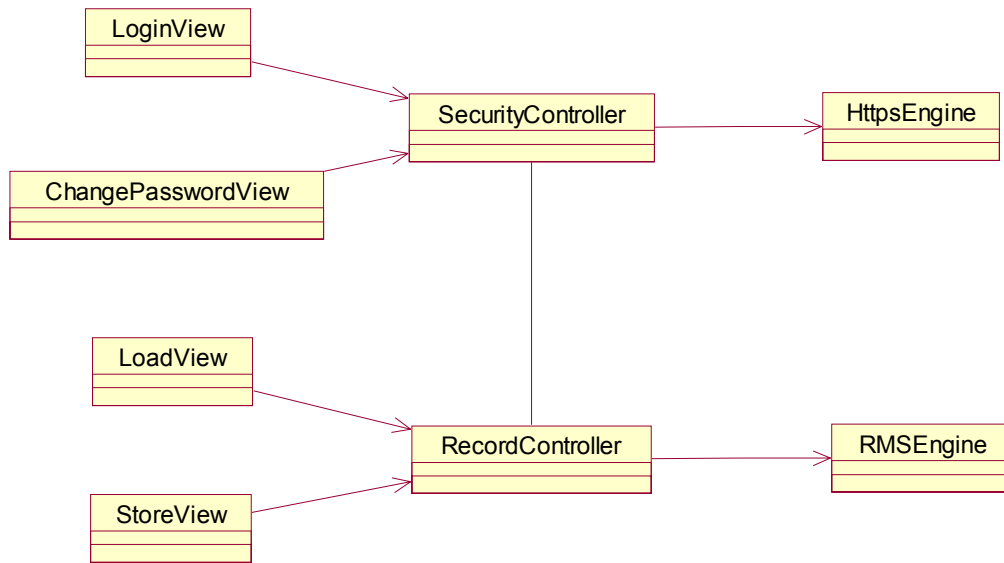


Figure 6: Object diagram of multi-controller MVC structure

### Pros:

With the multi-controller structure, the controller becomes easy to manage and the application can be divided into several parts and given to several developers.

### Cons:

With the increasing number of classes, objects and files, the interaction mechanism among them becomes more complex.

## **Conclusion**

Integrated MVC structure requires careful design and planning, you will have to spend enough time thinking about the interaction mechanism among different parts. The J2ME<sup>[awv7]</sup> client has its own feature: the application is usually small in size and the resources available are also limited. With the small screen size, the J2ME<sup>[awv8]</sup> application would rather like to replace a view than modify it. The best design choice is the one best fit for the application requirements, not the one with the most elegant structure.

## **References**

*Refactoring: Improving the Design of Existing Code*, Martin Fowler

*Applying UML and Patterns*, Craig Larman

Smart Ticket Blue Prints, <http://java.sun.com/blueprints/wireless/>