Design Pattern for Java ME Canvas based applications

Problem

Everyone working in Java ME must have faced this problem of “flicker” while switching displayable objects i.e. setting a new Canvas as current from one Canvas already in View, so it is advised that displayable Canvas should not be set again and again. We should use only One Canvas set as current throughout the application, this results in a single long unmanageable Canvas class with long Key press/release, paint methods and lot of Switch case / if then else etc which effects the performance and readability of the code.

Solution

We have to design a solution in such a way that only one Canvas (*BaseCanvas* – see the code snippet below) is set as current displayable only once by the MIDlet and using object oriented approach of creating different Canvas Screens for different modules of the application extending (*CanvasScreen* – see code snippet below) a common interface class objects which can be passed over the *BaseCanvas* without any flicker. Code for handing Paint and events of each of the screen is separated in such different screen classes and hence become more readable and efficient.

Advantages

- Easy to extend
- More efficient code
- More readable code
- Lower memory footprint

Source Code

***BaseCanvas* will be as under**

```
import javax.microedition.lcdui.*;
public class BaseCanvas extends Canvas
{
        /**Singleton Instance**/
        private static BaseCanvas mCanvas;

        /** Singleton Accessor Method with lazy initialization**/
        public static Canvas getCanvas()
        {if(mCanvas==null)
               mCanvas=new BaseCanvas();
         return mCanvas;
        }

        public CanvasScreen setCurrentScreen(CanvasScreen aCs)
        {
                 CanvasScreen old=mCs;
                 mCs=aCs;
                 return old;
        }

 public void paint(Graphics g)
 {
  if(null!=mCs) mCs.paint(g);
 }

 public void keyPressed(int keyCode)
 {
```

```
  if(null!=mCs) mCs.keyPressed(keyCode);
 }

 public void keyRepeated(int keyCode)
 {
  if(null!=mCs) mCs.keyRepeated(keyCode);
 }

        /**Stratergy Object **/
 private CanvasScreen mCs=null;

 private BaseCanvas()
 {
  setFullScreenMode(true);
 }
}
```

CanvasScreen Interface will be as under

```
import javax.microedition.lcdui.*;
public interface CanvasScreen
{
 public void paint(Graphics g);
 public void keyPressed(int keyCode);
 public void keyRepeated(int keyCode);
}
```

All other screens or canvas classes for different modules of the application will be as under (e.g. SplashScreen)

```
import javax.microedition.lcdui.*;
public class SplashScreen implements CanvasScreen
{
 BaseCanvas bc;
 public SplashScreen()
 {
  this.bc=BaseCanvas.getCanvas();
                bc.setCurrentScreen(this);
                // do what you want to paint on splash screen and load your assests.
 }

 public void keyRepeated(int keyCode){ }

 public void paint(Graphics g)
 {
  // paint your Splash Screen
  // paint Loader(g);
 }


 public void keyPressed(int keyCode)
 {
                // do what you need
```

```
  }

  private void repaint()
  {
   bc.repaint();
  }
}
```

All other screen classes will implement *CanvasScreen* interface, do their own painting and event handling within themselves.

Threading Safety

Access to BaseCanvas class above is not thread safe. Simultaneous access from multiple threads can give unpredictable results. So if an application access this class from multiple threads then access to it should be synchronized.