PMS: Portable Messaging System

January 28, 2008
Revision $\alpha 2$ (2)

Contents

1 Introduction

PMS is a small, but highly useful Java library with a stupid name (yes, we aware). It is somewhat alike a lightweight, in-JVM implementation of JMS, though with different architecture and API. It enables you to make the Observer design pattern ubiquitous in your code, without all the related pains of creating multiple listeners, handlers and add/remove methods for every message type you want to handle. It is especially useful with GUI projects or server-side code which handles lots of events, and it can change dramatically the way you design your applications, aiming for more simplicity and flexibility, both at the same time.

PMS helps with the following tasks:

- Handling events (messages) of multiple types with a single handler object (which can be an instance of anonymous class) in a uniform way;

- Adding/removing multiple message listeners, each handling messages of multiple types, and broadcasting messages to them—by delegating this task to `MessageListenerDelegate` objects;
- Executing message handling code in a separate thread or thread pool (thanks to `java.util.concurrent.Executor`)—as simple as adding @Asynchronous annotation to your handler method.

OK, we may talk the talk, but let's code the code.

2 The Hello World Example

Yes, the Observer pattern can be applied to the unfamous "Hello, World!" program and even make it somewhat interesting (compared to the original). Let's say we should welcome the world when the morning comes and say a polite goodbye when the evening takes over. So, let's define the morning and the evening.

Listing 1: Morning.java

```java
package com.miriamlaurel.pms.examples.helloworld;

public class Morning {
    public String toString() {
        return "Sun rises, birds sing!";
    }
}
```

Listing 2: Evening.java

```java
package com.miriamlaurel.pms.examples.helloworld;

public class Evening {
    public String toString() {
        return "Sun sets, birds sleep!";
    }
}
```

Now the time to implement our message handler—let's call it, without any originality, `HelloWorldHandler`.

Listing 3: HelloWorldHandler.java

```java
package com.miriamlaurel.pms.examples.helloworld;

import com.miriamlaurel.pms.Listener;

public class HelloWorldHandler {
    @Listener void $(Morning morning) {
        System.out.println(morning);
        System.out.println("Hello, world!");
    }

    @Listener void $(Evening evening) {
        System.out.println(evening);
        System.out.println("Goodbye, world!");
    }
}
```

Looks strange, huh? Let's make some clarifications.

The methods called `$` are *message handlers*. They will be called by dispatch listener when the time comes. You can use any method name you like, but if you don't intend to call these methods directly (which is normally the case)—it's recommended to stick with the `$`. This is the shortest and the less distractive method name available, and we like to be minimalists. The methods don't have any scope modifier (`public`, `private`, whatever)—default package visibility is OK for the same reason, though you may opt to change it.

Now it's actually the time to rise and set the sun, and see how this piece of code respond to it. Here we are:

Listing 4: Main.java

```
package com.miriamlaurel.pms.examples.helloworld;

import com.miriamlaurel.pms.listeners.dispatch.DispatchListener;

public class Main {
    public static void main(String[] args) {
        HelloWorldHandler handler
                = new HelloWorldHandler();
        DispatchListener listener
                = new DispatchListener(handler);
        listener.processMessage(new Morning());
        listener.processMessage(new Evening());
    }
}
```

And here is the output:

```
Sun rises, birds sing!
Hello, world!
Sun sets, birds sleep!
Goodbye, world!
```

Here we create a `DispatchListener`—the main workhorse of the framework. It accepts any object acting as a message (in our case, `Morning` and `Evening`). The `DispatchListener` takes our message handler as argument and is now ready to process messages. The instances of `Morning` and `Evening` are passed via `processMessage` method and routed to the appropriate handler method. Et voila! Things are done.

It was simple enough, wasn't it? This is the most simple and useful usage pattern of the PMS framework: typed message handling. Please note: if you are OK with extending another class (often not the case because your handler may already extend some other class up the hierarchy), you can make the handler a `DispatchListener` itself—by extending it. In that case, the `processMessage` method will be available right from the handler class.

3 Message Broadcasting

There are awful lots of situations when multiple listeners should react on the same message and fire appropriate handlers. Almost everyone wrote his share of events delegating/broadcasting code while developing any non-trivial event handling in Swing. Fear not, my friends, the quest is over. Now all you need to do is implement the `HasMessageListeners` interface:

Listing 5: HasMessageListeners.java

```
package com.miriamlaurel.pms.listeners;

import java.util.List;

public interface HasMessageListeners {
    public List<MessageListener> listeners();
}
```

Not a big deal, you see. But you have to forward incoming messages to all listeners in this list. Or have you? The `MessageListenerDelegate` class will happily do all the dirty work for you. You just need to create and initialize the delegate object as a class field and delegate the implementation of `HasMessageListeners` interface to it.

Consider the following example. Along with saying hello to the world at mornings, you have to do a lot of other stuff: make bed, shave, do a morning exercise (if you prefer to stay in the shape) etc. Here is how it could be done.

Let's define a handler object responsible for shaving, brushing the teeth, etc—let's call it `DailyRoutineHandler`.

Listing 6: DailyRoutineHandler.java

```
package com.miriamlaurel.pms.examples.routine;

// imports skipped for clarity

public class DailyRoutineHandler extends DispatchListener {
    @Listener void $(Morning morning) {
        System.out.println("Making bed...");
        System.out.println("Brushing teeth...");
        System.out.println("Shaving...");
    }

    @Listener void $(Evening evening) {
        System.out.println("Taking shower...");
        System.out.println("Going to sleep...");
    }
}
```

For simplicity's sake, here we making our message handler extending `DispatchListener` directly, allowing it to process messages itself.

Now let's create a class called `Life`, which will be able to handle such unfortunate events as coming of mornings and evenings by invoking all attached message handlers. This is achieved by delegating messages broadcasting to `MessageListenerDelegate`. OK, here is our `Life`:

Listing 7: Life.java

```
package com.miriamlaurel.pms.examples.routine;

// imports skipped for clarity

public class Life implements MessageListener,
                                    HasMessageListeners {

    private MessageListenerDelegate delegate
            = new MessageListenerDelegate();

    // Delegated methods...

    public void processMessage(Object message) {
        delegate.processMessage(message);
    }

    public List<MessageListener> listeners() {
        return delegate.listeners();
    }

    // Some other methods which may be useful in Life...
    // ...
}
```

Now, let's create a `Life`, add the both handlers we have to it, fire the events and see what happens.

Listing 8: Main.java

```
package com.miriamlaurel.pms.examples.routine;

// imports skipped for clarity

public class Main {
    public static void main(String[] args) {
        Life life = new Life();
        DispatchListener helloWorld
                = new DispatchListener(new HelloWorldHandler());
        DailyRoutineHandler dailyRoutine
                = new DailyRoutineHandler();
        life.listeners().add(helloWorld);
        life.listeners().add(dailyRoutine);
        life.processMessage(new Morning());
        life.processMessage(new Evening());
    }
}
```

Here is the expected output:

```
Making bed...
Brushing teeth...
Shaving...
Sun rises, birds sing!
Hello, world!
Taking shower...
Going to sleep...
Sun sets, birds sleep!
Goodbye, world!
```

Not bad, not bad. Please note that message handlers are always invoked in the LIFO (last-in-first-out) order.

That's all we have to do with the PMS framework. Now go code and sin no more. Oh wait...

4 Here Comes The Threads

All handler methods we have written to date are synchronous, i.e. *blocking.* That means if the code in one of these methods takes veeeeery long to execute, the caller will have to wait until it's done, and only then it can resume normal execution flow. It's oh-so-nineties. Is there any way to schedule message handling somewhere in a parallel thread, return right away and resume other important tasks down the caller's code?

Of course it is. To achieve this unprecedented advantage over your competitors, you'll have to do the following:

- Supply an implementation of `java.util.concurrent.Executor` as the argument of the constructor in `DispatchListener`;
- Mark handler methods which will be executed within this `Executor` as `@Asynchronous`.

There is no step three. Let's illustrate it by creating a message handler which responds to the morning sun unusually slow—the `SleepyMorningHandler`.

Listing 9: SleepyMorningHandler.java

```
package com.miriamlaurel.pms.examples.async;

// imports skipped for clarity

public class SleepyMorningHandler {
    @Asynchronous @Listener void $(Morning morning) {
        try {
            System.out.println("RING! RING!");
            System.out.println("Grrr... Have to " +
                    "snooze the alarm clock...");
            Thread.sleep(500);
            System.out.println("WHAT?! I'm late! Oh noes!");
        } catch (InterruptedException e) {
            System.out.println("OK, I'm awake...");
        }
    }
}
```

This method takes some time to execute, and therefore marked with `@Asynchronous` annotation. If there is an `Executor` set up in `DispatchListener`, it will be used to run the code within this method. Let's create it and emulate a normal daily workflow:

Listing 10: Main.java

```java
package com.miriamlaurel.pms.examples.async;

// imports skipped for clarity

public class Main {
    public static void main(String[] args) throws Exception {
        SleepyMorningHandler sleepyMorning
                = new SleepyMorningHandler();
        ExecutorService threadPool
                = Executors.newFixedThreadPool(2);
        DispatchListener listener
                = new DispatchListener(
                sleepyMorning, threadPool);
        listener.processMessage(new Morning());
        Thread.sleep(50);
        System.out.println("Meanwhile...");
        System.out.println("A work day started...");
        System.out.println("Where are my TPS reports?");
        threadPool.shutdown();
    }
}
```

Here's the expected tragic output:

```
RING! RING!
Grrr... Have to snooze the alarm clock...
Meanwhile...
A work day started...
Where are my TPS reports?
WHAT?! I'm late! Oh noes!
```

As you can see, the code in the handler method was properly executed in the newly created thread pool. Determining the count of lines of code in the equivalent implementation without the PMS is left as an exercise for the reader.

5 Going Swing

There is a certain place where event handling is a commonplace: GUI projects, yes, those with all the fancy windows, buttons and stuff. Most of the GUI development in Java is done with Swing (there are also people using SWT, but let's forget about them for now). And there is a certain caveat in Swing: all code actually doing something with the GUI should be executed from within a very special thread, called the AWT Event Dispatch Thread, or EDT.

Of course you could write a special kind of executor which forwards the actual execution to EDT by `SwingUtils.invokeLater` method (OK, OK, it's even already included in the PMS: see `AwtDispatchThreadExecutor` class). However, it's a pain to mark all methods `@Asynchronous`—it's quite easy to occasionally forget to do this, and then all the hell may break loose. Instead, the solution is to use `AwtDispatchListener` instead of plain vanilla `DispatchListener` in the Swing environment, and it will execute all `@Listener` methods in EDT automagically. As an additional bonus, you can still specify another executor in the constructor, and methods marked with `@Asynchronous` will be executed right there. This makes creating so-called "responsive GUIs" a complete no-brainer.

6 Conclusion

Now we have covered the entire PMS framework. Example code aside, it consists only of nine classes and interfaces, and it's implementation is wonderfully clean and simple. You are highly encouraged to use it while it's at version 1.0, before we decide to add lots of irrelevant features and bloat it beyond practical use.

Happy coding!