Writing An Event-Driven Framework With Java

published: April 16, 2012

Last time, we covered the fundamental components of an event-driven architecture. Now, we'll develop an event-driven framework so that we can get started with event-driven development as soon as possible.

Messaging Systems

Event-driven architecture follows several patterns from messaging systems. Consider the analog: events to **messages**, event handlers to **channels** and event dispatchers to **routers**.

A concrete example of this system is the postman (or the mailman). The postman has a *satchel with letters* which he delivers to several *homes*, each with their own *destination address*; he must deliver them accordingly.

The postman was once a programmer; thus, he constructs an algorithm to represent his thought process:

```
procedure deliver_letters(satchel):
        repeat
                letter := next_letter(satchel)
                for home in homes do:
                        if letter.destination == home:
                                deliver_letter(home)
                        end if
                end for
        until satchel is empty
end procedure
```

This example can be modeled using event-driven programming. Now let's develop a framework for such a model in its most abstract terms.

> *Beware: all content beyond this point is abstract.*

Messages

Recall that the analog for an event in a messaging system is a **message**. Each message has a specified **type** which will be used to associate with a handler.

We can define the interface for a message as thus:

```
public interface Message {
    public Class<? extends Message> getType();
}
```

ABOUT GIO

I am a torrent of ingenuity (or insanity) with a myriad of innovations (sometimes fallacies) and a wealth of inspiration (possibly naiveté). My name is Gio Carlo Cielo Borje and I like puffer fish because they're just cooltalkin', highwalkin' and fastlivin'.

I'm also nineteen and a current student at UC Irvine for Computer Science.

Channels

The second aspect to a messaging based system is the delivery point. Since we have a set of messages, we must define delivery points which we will call **channels**.

Each channel will be responsible for a single **type** of message; consequently, we can **dispatch** messages to its respective channel for processing.

We can then define the interface for a channel:

```
public interface Channel<E extends Message> {
    public void dispatch(E message);
}
```

Dynamic Routers

The harmony of messaging systems occurs through its routers. Routers are responsible for selecting the proper path for a given message.

> *The postman is considered a router.*

When initializing, a router will **register** a message with an associated channel. Afterwards, all messages dispatched by the router should automatically match its type with its associated channel and route the message to it.

Our router's interface:

```
public interface DynamicRouter<E extends Message> {
    public void registerChannel(Class<? extends E> contentType,
            Channel<? extends E> channel);
    public abstract void dispatch(E content);
}
```

All that's left now is to implement each of the interfaces for a complete framework.

Implementing the Framework

Events

Events are simple and we can define each event as a subclass of messages. A quick little trick here is to know that each class of an event is simply a new type of event; consequently, the type of the event is denoted by its class!

```
import Message;

public class Event implements Message {
    @Override
    public Class<? extends Message> getType() {
        return getClass();
    }
}
```

Handlers

Event handlers act as destination points for receiving events as channels do; thus, we implement the channel interface:

```
import Channel;
```

```
public class Handler implements Channel<Event> {
    @Override
    public void dispatch(Event message) {
        System.out.println(message.getClass());
    }
}
```

Event Dispatcher

Now comes the event dispatcher. The duty of the dispatcher is to first register channels with messages; in this case, we register handlers with their associated event class types.

We use the native `HashMap` to associate events with their respective handlers. Afterwards, we can simply query the map to store new (event, handler) associations or to dispatch events to the proper handler.

```
import java.util.HashMap;
import java.util.Map;

import edu.giocc.util.router.Channel;
import edu.giocc.util.router.DynamicRouter;

public class EventDispatcher implements DynamicRouter<Event> {
    private Map<Class<? extends Event>>, Handler> handlers;

    public EventDispatcher() {
        handlers = new HashMap<Class<? extends Event>, Handler>();
    }

    @Override
    public void registerChannel(Class<? extends Event> contentType,
            Channel<? extends Event> channel) {
        handlers.put(contentType, (Handler)channel);
    }

    @Override
    public void dispatch(Event content) {
        handlers.get(content.getClass()).dispatch(content);
    }
}
```

Our framework is complete! We can now test our framework:

```
public class Program {
    public static void main(String[] args) {
        EventDispatcher dispatcher = new EventDispatcher();
        dispatcher.registerChannel(Event.class, new Handler());
        dispatcher.dispatch(new Event());
    }
}
```

The code above should output the class name of the event into the terminal. Now, all applications can simply register further events and handlers and dispatch them.

Extending the Framework into an Application

Now that the framework has been established, we can note a few properties of general frameworks that we must follow:

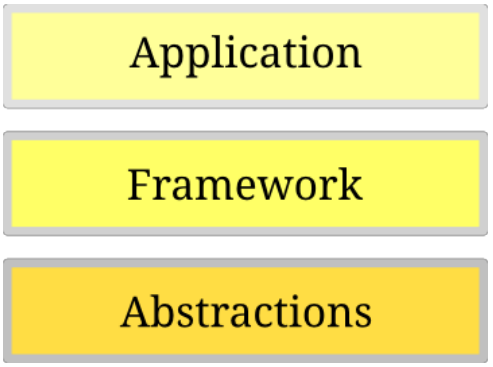
Inversion of control

The framework controls the flow of data throughout the system as a messaging system does.

Extensibility
The framework can be extended for application use.

Non-modifiable framework code
Don't change the framework!

Each of the aforementioned properties explain how the applications interface with the framework. The framework is simply an abstraction of the architecture.



Application Layers

The `Handler` and `Event` classes are part of our framework layer which are not to be touched. All code that extends our framework will exist on the application layer of the above diagram. The duty of our framework is to abstract and scaffold the architectural components of our event-driven architecture and it has been done.

It's easy to see now that we can inherit from the `Handler` class to create our own event handlers and we can also inherit form the `Event` class to create our own events. Additionally, we can **register** those event handlers and events to dispatch them accordingly.

> *The source of events here is arbitrary since it is up to the application at this point where the events are genereted. Commonly, events are generated from standard I/O or from network packets (which requires a bit of socket programming).*

Next time, I'll show you a concrete implementation of this framework for simulating systems.

This entry was posted in Innovation and tagged event dispatcher, event-driven, framework, java, messaging system, router. Bookmark the permalink. Post a comment or leave a trackback: Trackback URL.

« Fundamental Components of an Event-Driven Architecture

Analysis of a Plaintext Representation of Adjacency Lists »

One Trackback

- By Event-Driven Simulations in Java: Chat Application Part 1 on May 18, 2012 at 2:11 am

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name *

Email *

Website

Comment

Post Comment

CATEGORIES

Ingenuity (16)

Innovation (8)

Inspiration (13)

RECENT POSTS

Designing a Linux Resource Manager in C++

MinDispatch: Event-Driven Framework In Java Part 2

Partioning Discussion Sections for Lecture-Hall Sized Classes

Projects Matching Problem of ICS Clubs and Small Organizations

Historical Problems with Closures in JavaScript and Python

PROJECTS

GitHub

MetaZaku

ZeroZaku