MinDispatch: Event-Driven Framework In Java Part 1

published: May 18, 2012

We've gotten our feet wet with event-driven programming by developing a framework which controls the flow of data through our system. Effectively, I've made my framework available on Github for use by anyone: MinDispatch framework on GitHub.

The Chat Application Simulation Revisited

Our objective once again is to simulate the events of a chat application by explicitly specifying a set of events.

However, we will no longer define the flow of data through our system for we have a framework which handles this. Instead, we can simply define the set of events and processes that our system must simulate.

Events of a Chat Application

User Arrival
Occurs when a user arrives to a room.

User Departure
Occurs when a user departs from a room.

User Message
Occurs when a user sends a message to a room.

We want to process such events by simply printing out the standard output.

```
foo has entered the room.
bar has entered the room.
foo: hello, bar!
bar: hello, foo!
foo: goodbye, bar!
foo has left the room.
```

Through the observations above, we should be able to quickly design our program in two steps: model the events and process the events. Finally, we can dispatch events to test our design.

Prerequisites

First, you must download the MinDispatch source and package it with your current project which we will name **ChatEventMachine**.

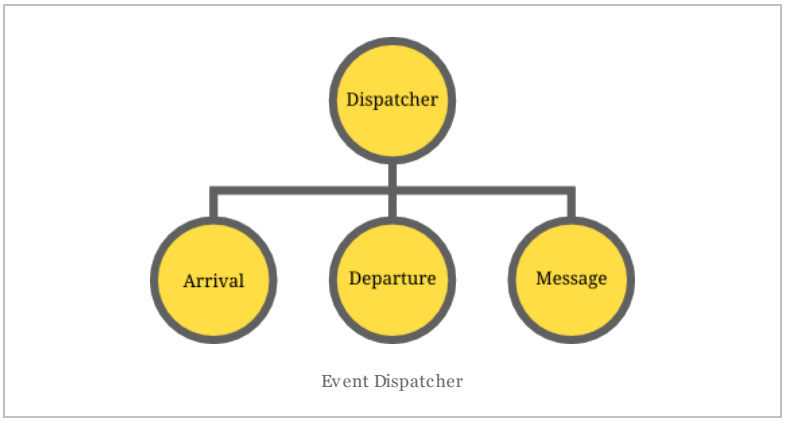
Coding the Simulation with MinDispatch

We modeled our events and now we must consider how they will be routed by the event dispatcher of our framework.

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.



Event Dispatcher

Since our dispatcher on our framework is responsible for routing events to their handlers, we start by identifying the events necessary as we have above. Afterwards, we simply need to register the events to a respective set of event handlers which process each event accordingly.

First: Model the Events

Let's begin with modeling the events specified:

```
public class ChatEventMachine {
    private static class User {
        public String name;

        public User(String name) {
            this.name = name;
        }
    }

    private static class UserArrival extends Event {
        public User user;

        public UserArrival(User user) {
            this.user = user;
        }
    }

    private static class UserDeparture extends Event {
        public User user;

        public UserDeparture(User user) {
            this.user = user;
        }
    }

    private static class UserMessage extends Event {
        public User user;
        public String message;

        public UserMessage(User user, String message) {
            this.user = user;
            this.message = message;
        }
    }
}
```

In our code above, we use an auxilliary `User` class which helps us encapsulate data which is only associated to the user such as the user's name.

User classes frequently associate with many more properties such as a unique

user id, password, email and more depending on the application.

Each of our events provide sufficient data for their handlers to use. The constructors of the new events, though unnecessary, makes dispatching easier as will be seen later in this tutorial.

Second: Route Events to Handlers

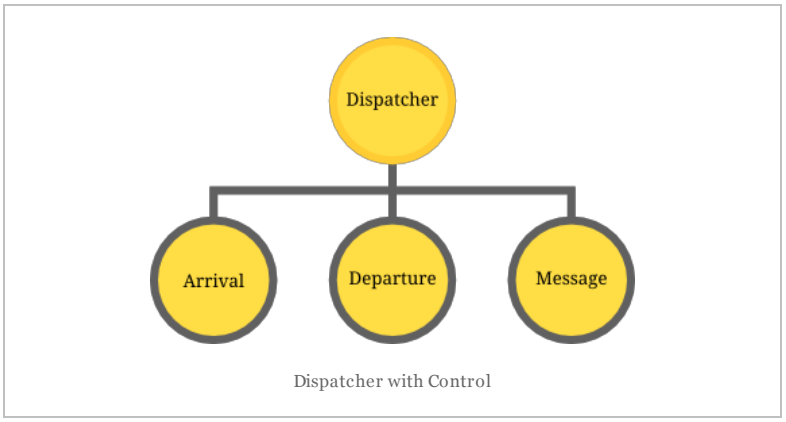
After our events have been successfully modelled, we can easily route events to handlers through the event dispatcher within our framework.

```
public class ChatEventMachine {
    // Event models

    public static void main(String[] args) {
        EventDispatcher dispatcher = new EventDispatcher();

        dispatcher.registerChannel(UserArrival.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserArrival arrival = (UserArrival)evt;

                System.out.println(arrival.user.name + " has entered the
            }
        });

        dispatcher.registerChannel(UserDeparture.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserDeparture departure = (UserDeparture)evt;

                System.out.println(departure.user.name + " has left the
            }
        });

        dispatcher.registerChannel(UserMessage.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserMessage message = (UserMessage)evt;
                String userMessage = String.format("%s: %s", message.use
                System.out.println(userMessage);
            }
        });
    }
}
```

In the code above, we simply register a set of channels by mapping an event subclass to a new handler through a `HashMap` according to our framework implementation. Note here that I simply create a new, unique event handler for each event since each event requires a unique process.

Third: Dispatch Events to Simulate

Since our event dispatcher has now been set up at this point, we can start dispatching a set of events to simulate a chat application. This marks the beginning of control starting from the event dispatcher.



Dispatcher with Control

In this simple example, I will hard-code dispatched events by manually writing the constructors for the necessary objects and dispatching them through the framework's dispatcher.

```
public class ChatEventMachine {
    // Event models

    public static void main(String[] args) {
        // Dispatcher and handler definitions

        User foo = new User("foo");
        User bar = new User("bar");
        dispatcher.dispatch(new UserArrival(foo));
        dispatcher.dispatch(new UserArrival(bar));
        dispatcher.dispatch(new UserMessage(foo, "hello, bar!"));
        dispatcher.dispatch(new UserMessage(bar, "hello, foo!"));
        dispatcher.dispatch(new UserMessage(foo, "goodbye, bar!"));
        dispatcher.dispatch(new UserDeparture(foo));
    }
}
```

At this point, our chat application simulator is complete. When we execute the main method of this program, we achieve the following:

```
foo has entered the room.
bar has entered the room.
foo: hello, bar!
bar: hello, foo!
foo: goodbye, bar!
foo has left the room.
```

The full source at the end of our tutorial:

```
public class ChatEventMachine {
    private static class User {
        public String name;

        public User(String name) {
            this.name = name;
        }
    }

    private static class UserArrival extends Event {
        public User user;

        public UserArrival(User user) {
            this.user = user;
        }
```

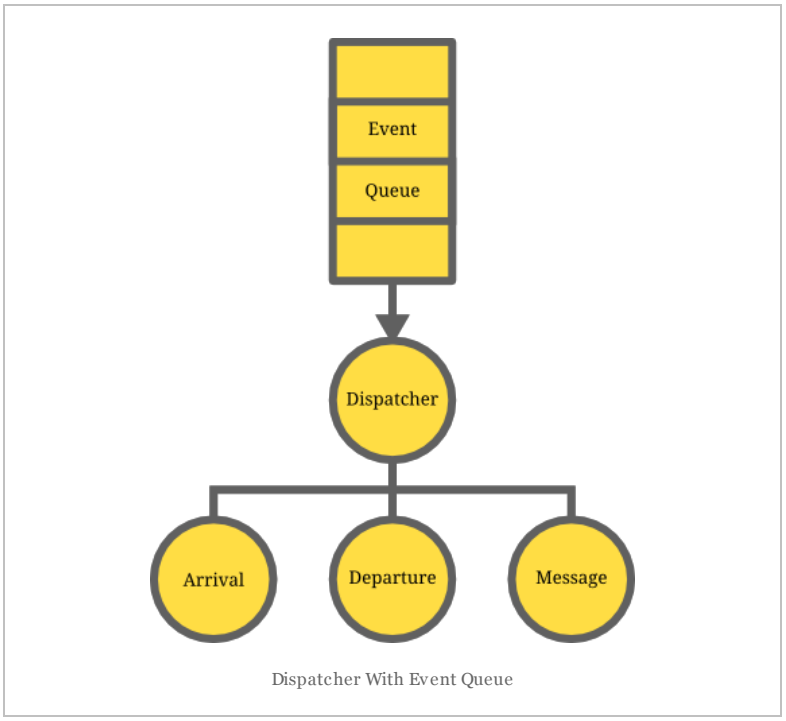
```
    }

    private static class UserDeparture extends Event {
        public User user;

        public UserDeparture(User user) {
            this.user = user;
        }
    }

    private static class UserMessage extends Event {
        public User user;
        public String message;

        public UserMessage(User user, String message) {
            this.user = user;
            this.message = message;
        }
    }

    public static void main(String[] args) {
        EventDispatcher dispatcher = new EventDispatcher();

        dispatcher.registerChannel(UserArrival.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserArrival arrival = (UserArrival)evt;

                System.out.println(arrival.user.name + " has entered the
            }
        });

        dispatcher.registerChannel(UserDeparture.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserDeparture departure = (UserDeparture)evt;

                System.out.println(departure.user.name + " has left the
            }
        });

        dispatcher.registerChannel(UserMessage.class, new Handler() {
            @Override
            public void dispatch(Event evt) {
                UserMessage message = (UserMessage)evt;
                String userMessage = String.format("%s: %s", message.use
                System.out.println(userMessage);
            }
        });

        User foo = new User("foo");
        User bar = new User("bar");
        dispatcher.dispatch(new UserArrival(foo));
        dispatcher.dispatch(new UserArrival(bar));
        dispatcher.dispatch(new UserMessage(foo, "hello, bar!"));
        dispatcher.dispatch(new UserMessage(bar, "hello, foo!"));
        dispatcher.dispatch(new UserMessage(foo, "goodbye, bar!"));
        dispatcher.dispatch(new UserDeparture(foo));
    }
}
```

Conclusion

I was able to write this code within ten minutes; consequently, we see the power of a simple, event-driven framework, namely, MinDispatch, in quickly modelling event-based system and handling it accordingly.

> *A few people may notice that this approach is capable of modeling various live systems such as multiplayer gaming where a set of players may emit events to interact with the world e.g. opening chests, communicating with others and fight sequences.*

I purposely ommitted a parse file or dynamic input stream; however, I plan to continue this tutorial by extending our application to include a dynamic stream of events as shown below:



Dispatcher With Event Queue

With the above diagram as a treat for the next post in this series, please look forward to reading about building a simple AI to respond to chat events!

This entry was posted in Innovation and tagged chat application, event dispatcher, event-driven, java, MinDispatch, simulation. Bookmark the permalink. Post a comment or leave a trackback: Trackback URL.

« Analysis of a Plaintext Representation of Adjacency Lists

Prelude into Underscorejs: Higher-Order Functions »

One Comment

Devontrae Walls Posted May 18, 2012 | *Permalink*

Very informative read and really useful learning experience for anyone really trying to dive into the world of event-driven programming.

Reply

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