**Fundamental Components Of An Event-Driven Architecture**

My favorite type of architecture is event-driven because of the trivial train of thought as action-reaction. Consider what happens when an incoming tornado approaches; what do we do? Notify the locals. Consider what happens when an online purchase occurs; what do we do? Ship it. Consider what happens when an event occurs; what do we do? Handle it.

## No Less Fundamental Than These Components

An event-driven architecture comprises of a simple set of components to run yet it can represent a variety of situations and how to handle them.

**Events**

An occurrence to handle.

**Event Handlers**

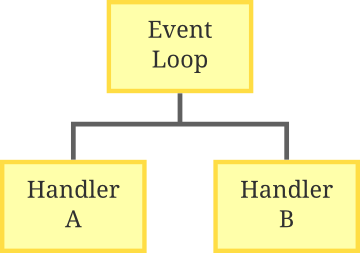
A method to handle an occurence.

**Event Loop**

Maintains the flow of interaction between events and event handlers.

### ***The Flow of Events***

The flow of events is straightforward: events are sent to the event loop and the event loop dispatches each event to their respective event handler.

[](http://www.giocc.com/wp-content/uploads/EventLoop1.png)

Dispatching Events to Handlers

Consequently, events of type A will be handled by Handler A and events of type B will be handled by Handler B.

*Interestingly, the flow of events is often modeled through simple language just as the introduction to this article does. Abstractly, all events and their handling can be described the following manner: if event p occurs, then execute q.*

## Events

The heart of an event-driven architecture.

Events contain two necessary properties: **type** and **data**. The type of the event determines which handlers handle the event. The data of the event is for the handler to use.

*Conceptually, we can conceive an event object as a container for data which contains meta data such as the event type to determine how to handle the data.*

We can then model a simple event in Java:

public static class Event {

public char type;

public String data;

public Event(char type, String data) {

this.type = type;

this.data = data;

}

}

## Event Handlers

Handlers are specific ways to deal with an event. Common operations include filtering or transforming the data associated with the event.

public static void printEventA(Event e) {

System.out.println(e.data);

}

The example code above is a handler for events of type A. Simply, the code prints the data of the event to the console.

public static void printEventB(Event e) {

System.out.println(e.data.toUpperCase());

}

This second example transforms the event data to upper case before the data appears in the console.

## Event Loop

This processes all events as they arrive by dispatching them to their respective handlers until a terminating condition occurs.

public static void main(String[] args) {

Queue<Event> events = new LinkedList<Event>();

events.add(new Event('A', "Hello"));

events.add(new Event('B', "event-driven"));

events.add(new Event('A', "world!"));

Event e;

while (!events.isEmpty()) {

e = events.remove();

if (e.type == 'A')

printEventA(e);

if (e.type == 'B')

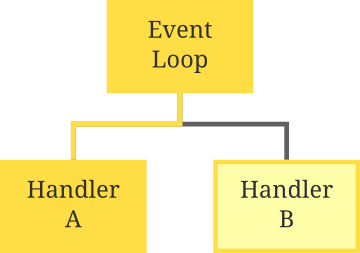
printEventB(e);

}

}

Before initializing our event loop, we create a Queue of event objects to schedule a set of events for processing so that we can see some output. Consequently, the event loop here begins at the while-loop. Yes, the event loop is frequently a loop.

In the event loop, each event is removed from the queue and handled according to their type until there are no more events to process. In total, we should process the three events in order of which they were added. Consider how the first event is handled:

[](http://www.giocc.com/wp-content/uploads/EventLoopPathA.png)

Path for Handling Event A

It’s easy to see now that the events are simply data that are passed down a specific path towards a handler. Consider the first event processed: the event is passed into the event loop where it is appropriately routed to Handler A.

**Putting It All Together**

In this Java code example, I’ve created concrete implementations of a simple simulation of events through an event-driven architecture.

* **Events** are defined by the Event class.
* **Event handlers** are methods (printEventA and printEventB) which accept events as parameters.
* The **event loop** is implemented within the main method which passes schedules events into their respective event handlers.

import java.util.LinkedList;

import java.util.Queue;

public class EventMachine {

// Event definition

public static class Event {

public char type;

public String data;

public Event(char type, String data) {

this.type = type;

this.data = data;

}

}

// Event handler A

public static void printEventA(Event e) {

System.out.println(e.data);

}

// Event handler B

public static void printEventB(Event e) {

System.out.println(e.data.toUpperCase());

}

public static void main(String[] args) {

Queue<Event> events = new LinkedList<Event>();

events.add(new Event('A', "Hello"));

events.add(new Event('B', "event-driven"));

events.add(new Event('A', "world!"));

// Event loop

Event e;

while (!events.isEmpty()) {

e = events.remove();

if (e.type == 'A')

printEventA(e);

if (e.type == 'B')

printEventB(e);

}

}

}

The following output should then be:

Hello

EVENT-DRIVEN

world!

### ***Extending the Fundamentals***

The implementation of these components are not limited to this tutorial. Event handlers can pass the events to further handlers and event loops can route events to their respective handler using a map data structure for matching handlers in logarithmic time instead of linear time as used in this tutorial.

Next time, I’ll show you how to create an event-driven framework in Java with some design patterns.

# Writing an Event-Driven Framework With Java

## 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();

}

### ***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.

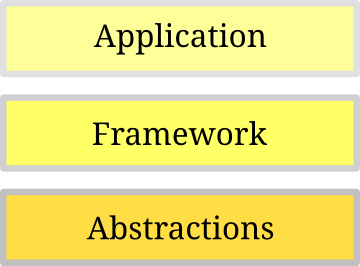
**Extensibility**

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.

[](http://www.giocc.com/wp-content/uploads/FrameworkLayers1.png)

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).*

# MinDispatch: Event-Driven Framework In Java Part 1

We’ve gotten our feet wet with event-driven programming by [developing a framework](http://www.giocc.com/writing-an-event-driven-framework-with-java.html) 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](https://github.com/Hydrotoast/MinDispatch/tree/master/edu/giocc/MinDispatch" \o "MinDispatch).

## 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.

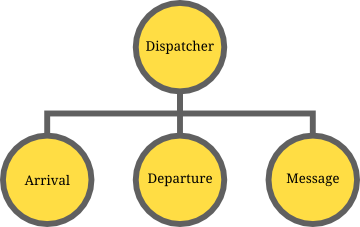
Through the observations above, we should be able to quickly design our program in two steps: model the events and process the events. Finally, we can dispatch events to test our design.

### ***Prerequisites***

First, you must [download the MinDispatch source](https://github.com/Hydrotoast/MinDispatch/tree/master/edu/giocc/MinDispatch) 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.

[](http://www.giocc.com/wp-content/uploads/MinDispatcher.png)

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 room.");

}

});

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 room.");

}

});

dispatcher.registerChannel(UserMessage.class, new Handler() {

@Override

public void dispatch(Event evt) {

UserMessage message = (UserMessage)evt;

String userMessage = String.format("%s: %s", message.user.name, message.message);

System.out.println(userMessage);

}

});

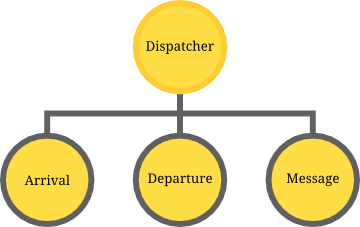
}

}

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.

[](http://www.giocc.com/wp-content/uploads/MinDispatcherControl.png)

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 room.");

}

});

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 room.");

}

});

dispatcher.registerChannel(UserMessage.class, new Handler() {

@Override

public void dispatch(Event evt) {

UserMessage message = (UserMessage)evt;

String userMessage = String.format("%s: %s", message.user.name, message.message);

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));

}

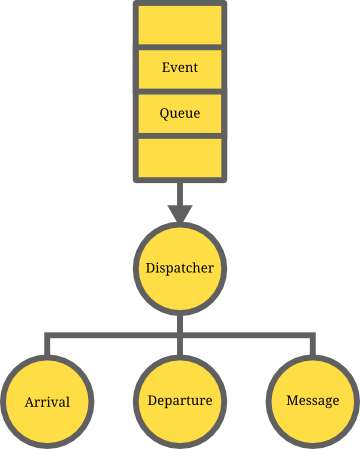
}

## 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:

[](http://www.giocc.com/wp-content/uploads/MinDispatcherWithQueue.png)

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!

# MinDispatch: Event-Driven Framework In Java Part 2

From the last post in this series, we developed a fixed, event-driven chat simulation. In this post, we will extend this example by refactoring. The objective of this tutorial is to teach effective design patterns in an event-driven model.

First we will begin by designing the structure and behavior of the user and chat to describe our application. Second, we will bind the aforementioned chat state to the event handlers to fix constant parameters. Finally, we will use an event queue for separation of concerns.

### ***Continuation***

We will continue with our previous example which you can find on [GitHub](https://github.com/Hydrotoast/MinDispatch" \o "MinDispatch). It will be easier to follow along with the source code.

Beware that I will no longer provide a full example of the code. It is expected that you are aware of where the highlighted code modifications occur because the code file has grown large.

## Structural Design

We begin by specifying the structural requirements of our chat. That is, we must specify the fields associated with our two primary data structures: User and ChatState.

### ***User State Structure***

We begin with the atomic data structure, the User. A user in a chat has a name.

private static class User {

public String name;

public User(String name) {

this.name = name;

}

}

Simple enough.

### ***Chat State Structure***

Next,the ChatState must model the configuration of a chat room: it should maintain the list of users currently in the chat.

private static class ChatState {

private ArrayList<User> users;

public ChatState() {}

}

Now to move on to the behaviors of our design.

## Behavioral Design

The structure of our design has been outlined. Now we must enable interactions between these structures relative the possible events that may occur. Recall the 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.*

Both the Chat and User must respond to these events accordingly. The following two sections will describe their behavioral implementation respectively.

### ***Chat State Behaviors***

Beginning with the ChatState this time, we must handle each of the aforementioned events. Specifically, we must support the following operations:

* the addition of users to the chat,
* the removal of users from the chat,
* and broadcasting messages to all users currently in the chat.

Broadcasting must be handled in a special manner. For broadcasting, we must specify our recipients: all users should receive a copy of an event dispatched. Since we have outlined the design, consider the following interface which expresses our intent,

private static class ChatState {

private ArrayList<User> users;

public ChatState() {}

public void broadcast(Event evt) {}

// Mutators

public void addUser(User user) {}

public void removeUser(User user) {}

}

The implementation is relatively straightforward: the users list maintains the current users in the chat and events can be broadcasted to subsequent users.

private static class ChatState {

private ArrayList users;

public ChatState() {

this.users = new ArrayList();

}

public void broadcast(Event evt) {

for (User recipient : users)

recipient.dispatch(evt);

}

// Mutators

public void addUser(User user) {

users.add(user);

}

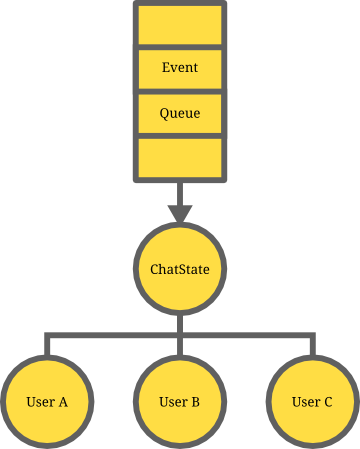
public void removeUser(User user) {

users.remove(user);

}

}

The ChatState is responsible for broadcasting events to each of the registered users. So, theChatState must broadcast the message to individual users so that they may handle the messages individually. See the diagram for more information.

[](http://www.giocc.com/wp-content/uploads/ChatStateBroadcasting.png)

Event broadcasting

When we construct the event queue, the ChatState will act as an event forwarding mechanism for UserMessage events.

Beware that there is a dependency the above code. The broadcast method dispatches events to users by calling the User.dispatch method which doesn’t exist yet. So, let us continue onto theUser behaviors.

### ***User State Behaviors***

We will outline the behavioral implementation of our User class now. Since the user is capable of receiving events, we should demultiplex the incoming events and handle them appropriately. Specifically, we want to know if a user received a message. Consider the implementation then:

private static class User {

public String name;

public User(String name) {

this.name = name;

}

// Event demultiplexing

public void dispatch(Event evt) {

if (evt.getClass() == UserMessage.class) {

UserMessage message = (UserMessage) evt;

processMessage(message.user, message.message);

}

}

// Event processing

public void processMessage(User user, String userMessage) {

// Ignore messages by me

if (user.equals(this))

return;

System.out.println(

name + " received message from " +

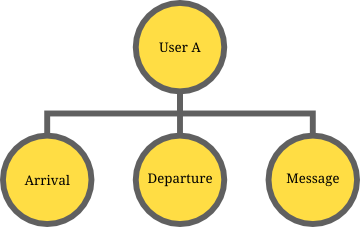
user.name

);

}

}

Take a look at the broadcast method. The type of the event argument is compared againstUserMessage.class. This if-ladder is an example of event demultiplexing.

[](http://www.giocc.com/wp-content/uploads/ChatStateDemultiplexing.png)

Event demultiplexing

*Event demultiplexing occurs when a stream of events split its channels thereby processing the events individually rather than as a stream. This may warrant a need for an event dispatcher within a*User*.*

When demultiplexing events, we route events to their respective handlers. Specifically, we route all of the UserMessage events to the processMessage handler and ignore the rest (arrival and departure are ignored). Once the events have been handled after demultiplexing, the behaviors of the data structure are complete.

## Binding Chat State to Event Handlers

Unfortunately, now that a ChatState exists, we must pass the object, as a parameter, to the each of the event handlers so that they may change the state of the object. Consider the event handler setup for UserArrival

state.registerChannel(UserArrival.class, new ChatHandler() {

@Override

public void dispatch(Event evt) {

UserArrival arrival = (UserArrival) evt;

arrival.state.addUser(arrival.user);

System.out.println(

arrival.user.name + " has entered the room."

);

}

});

Passing state along with each event may cause significant code duplication as well as unnecessary runtime overhead. With the current design of event handlers, each of the previously designed handlers to the user events, UserArrival, UserDeparture and UserMessage must store a reference to the ChatState that they operate on.

There exists a solution which removes the code duplication and the runtime overhead. We can push the responsibility of maintaining state to the event handlers by binding the ChatState to a custom event handler. We know that this is feasible because ChatState is the same throughout the execution of this simulation.

### ***Application-specific Chat Handlers***

We will implement our own event handlers, ChatHandler, specifically for handling chat-specific events on a ChatState. Simply, this custom handler should fix the parameter common to all of our handlers, ChatState.

private static class ChatHandler extends Handler {

protected ChatState state;

public ChatHandler(ChatState state) {

this.state = state;

}

}

Afterwards, we may access the state of the chat for each subsequent ChatHandler. So, the handler registration will be slightly different with an inherited handler.

public static void registerHandlers(EventDispatcher dispatcher, ChatState state) {

dispatcher.registerChannel(UserArrival.class, new ChatHandler(state) {

@Override

public void dispatch(Event evt) {

UserArrival arrival = (UserArrival) evt;

state.addUser(arrival.user);

System.out.println(

arrival.user.name + " has entered the room."

);

}

});

dispatcher.registerChannel(UserDeparture.class, new ChatHandler(state) {

@Override

public void dispatch(Event evt) {

UserDeparture departure = (UserDeparture) evt;

state.removeUser(departure.user);

System.out.println(

departure.user.name + " has left the room."

);

}

});

dispatcher.registerChannel(UserMessage.class, new ChatHandler(state) {

@Override

public void dispatch(Event evt) {

UserMessage message = (UserMessage) evt;

String userMessage =

String.format(

"%s: %s",

message.user.name,

message.message

);

System.out.println(userMessage);

// Broadcast messages

state.broadcast(message);

}

});

}

*In our new implementation, we will use a*registerHandlers*helper function to initialize our event handlers with a specified event dispatcher and*ChatState*.*

The state of the chat is updated in the above event handlers using the behavioral design that we have previously specified. Hence, we have effectively decoupled the state of the chat from the event dispatching.

## Using an Event Queue

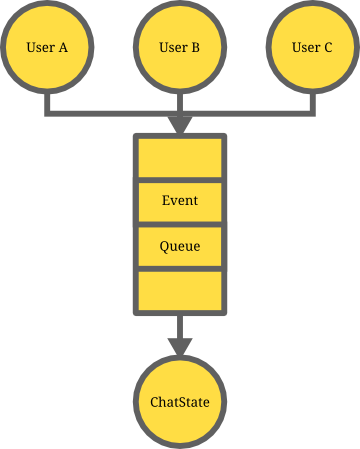
Next, we will utilize an event queue to separate concerns.

*In computer science,*separation of concerns*(SoC) is the process of breaking a computer program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program.*

The event queue will enable us to separate the event dispatcher from the application-specific users and the chat state. That is, users should be unaware of the existence of a dispatcher especially when generating events themselves.

*Additionally, when we introduce concurrency into this application (hint), the queue will also serve as a shared buffer between distributed users and the server which decouples application-independent concurrency mechanisms from our application-specific method functionality.*

Conceptually, the event queue acts as a multiplexed channel which interleaves events from individual users since there is no particular order in which users may send messages. The event queue’s only concern is event multiplexing.

[](http://www.giocc.com/wp-content/uploads/ChatStateMultiplexing.png)

Event multiplexing

For now, we will simply use a simple Queue<Event> in the Java standard library to express our intent. So, to instantiate this, we use a java.util.LinkedList.

import java.util.LinkedList;

// ChatState declaration here

public static void main(String[] args) {

EventDispatcher dispatcher = new Dispatcher();

ChatState state = new ChatState();

Queue<Event> eventQueue = new LinkedList<Event>();

// Further simulation code such as event handler registration

}

### ***Integration with Dispatcher***

Since the Dispatcher is responsible for dispatching events, we should dispatch all of the events in queue when flushing the buffer.

import java.util.LinkedList;

// ChatState declaration here

public static void main(String[] args) {

EventDispatcher dispatcher = new Dispatcher();

ChatState state = new ChatState();

Queue<Event> eventQueue = new LinkedList<Event>();

// Further simulation code such as event handler registration

// Possibly generate events beforehand

// Dispatch all queued events

while (!eventQueue.isEmpty()) {

Event evt = eventQueue.remove();

dispatcher.dispatch(evt);

}

}

Furthermore, notice that the event queue does not interact with the ChatState. This is a highlight of separated concerns because event queues are application-independent.

### ***Integration with Users***

Before dispatching events with users, we must connect users to the event queue. Simply, we enable each individual user to reference the event queue in the implementation.

private static class User {

public Queue<Event> eventQueue;

public String name;

public User(Queue<Event> eventQueue, String name) {

this.eventQueue = eventQueue;

this.name = name;

}

// Behavioral methods

}

Once users have a reference to the event queue, they are able to generate events. Specifically, we want to enable users to send messages to the chat, thereby sending a message to all other users currently in the chat.

private static class User {

public Queue<Event> eventQueue;

public String name;

public User(Queue<Event> eventQueue, String name) {

this.eventQueue = eventQueue;

this.name = name;

}

// Event demultiplexing and handling methods

// Event generation

public void sendMessage(String message) {

eventQueue.add(new UserMessage(this, message));

}

}

Thus, users are now capable of sending messages without being aware of the event dispatcher and the chat state. Effectively, this is a highlight of modularity where modifications to a user’s capability in the system is independent of the modifications to the chat state and event dispatcher.

## Testing the Simulation

Finally, the final source should be similar to my source code on [GitHub](https://github.com/Hydrotoast/MinDispatch/blob/master/edu/giocc/EventMachine/StatefulChatEventMachine.java). Now, we can test the simulation using the following main and hardcoded events:

public static void main(String[] args) {

EventDispatcher dispatcher = new EventDispatcher();

ChatState state = new ChatState();

Queue<Event> eventQueue = new LinkedList<Event>();

registerHandlers(dispatcher, state);

// Initialize users

User foo = new User(eventQueue, "foo");

User bar = new User(eventQueue, "bar");

dispatcher.dispatch(new UserArrival(foo));

dispatcher.dispatch(new UserArrival(bar));

// Enqueue events from individual users

foo.sendMessage("hello, bar!");

bar.sendMessage("hello, foo!");

foo.sendMessage("goodbye, bar!");

// Dispatch all queued events

while (!eventQueue.isEmpty()) {

Event evt = eventQueue.remove();

dispatcher.dispatch(evt);

}

// Finish up simulation

dispatcher.dispatch(new UserDeparture(foo));

dispatcher.dispatch(new UserDeparture(bar));

}

The following output should be produced:

foo has entered the room.

bar has entered the room.

foo: hello, bar!

bar: hello, foo!

foo: goodbye, bar!

foo has left the room.

bar has left the room.

Thus, our chat simulation is complete.

## Conclusion

Effective application design coupled with an event queue makes modification of the code far easier simply because we have a separation of concerns and modularity. That is, modifications to our application-specific handlers or data structures are independent of modifications to the application-independent event-driven framework, [MinDispatch framework on GitHub](https://github.com/Hydrotoast/MinDispatch/tree/master/edu/giocc/MinDispatch" \o "MinDispatch)..

It is easy to see that using the MinDispatch framework significantly simplifies the design for an event-driven application by handling the application-independent work.

### ***Another Continuation***

There are two paths we can take from here:

1. Enabling concurrency and distributed computing
2. Enabling dynamic user and input and developing a chat AI