Event Driven Architecture for Skribbage Battle Royale

Alex Carney

Contents

Color Code:

Blue: Concrete class (UserLoginEvent)

Blue Italicized: Abstract Class (AccountEvent)

Green: Interface (Payload)

Orange: Enum (EventType)

Purple: Java keyword (instanceof)

Abstract:

One of the most important elements of good software design is the prospect of "orthogonality". Essentially, a change in code should not affect unrelated components of the project. While it is usually infeasible to make every single operation in a product completely independent (dependencies in data will almost always be required), one can try to limit as many unnecessary dependencies as possible, using a time-tested design pattern to achieve data flow or a desired behavior without strongly coupling a project. In this report the architecture design is introduced in **Section 1**, with examples of its potential implementation in the Skribbage project in **Section 2.** Note that while the architecture is in a working state, it can always benefit from new ideas and refactoring. This has been a huge learning experience for us so far, so we would greatly appreciate feedback on the design itself, its effectiveness, ease of use, etc.

It is also important to recognize when this system should be used to handle communication. Not every single request between layers needs an event/listener combination. Synchronous calls to a

module's façade are usually sufficient, especially between modules on the back end. Any request that needs to be asynchronous (meaning it should not lock up the application while it is being processed) should be controlled by this event system. This includes most front-end requests to the back end systems.

Section 1: Design

1.1 Payloads:

Our design aims to solve a problem: Data must be shared, and potentially mutated, across multiple tiers of our application. It is easiest to explain the design with a concrete example, so I chose an account management system to base the UML design around. Our goal is to set up an architecture where the front-end team can handle login information from the user, send that payload to the logic tier, who then processes the request and sends it back to the front end to display the success or failure. (Note that in the actual process, the persistence layer would need to be involved as well, but for our toy example we can safely assume communication is just between two modules).

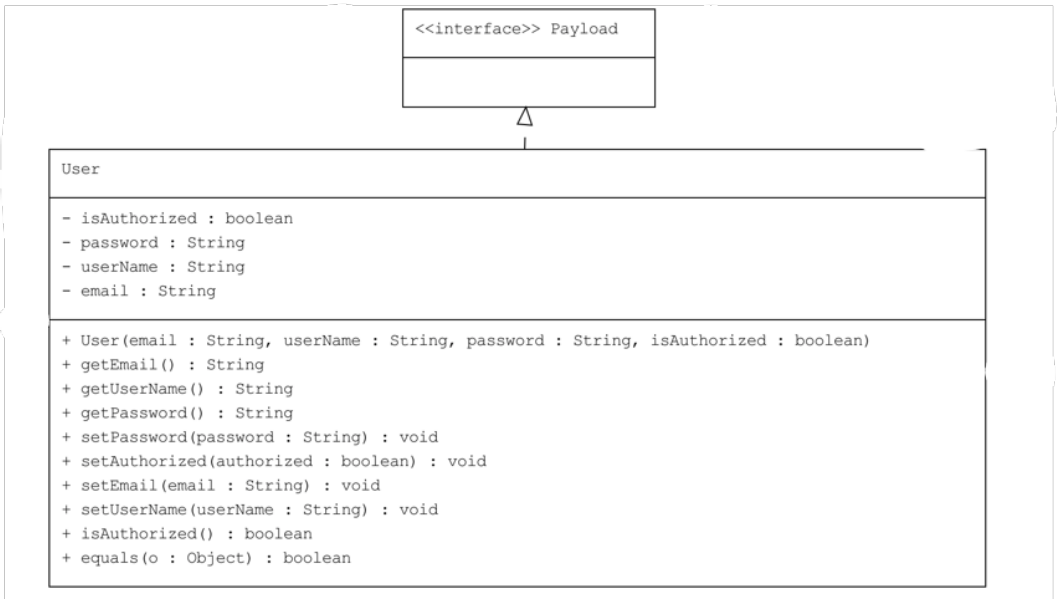

Figure) A concrete User class should implement Payload

A class that represents data to be sent between multiple layers implements the Payload interface, a marker interface that tags a class as a valid member of an event. This is mainly done to discourage usage of primitive types as data to be shared between teams. Data classes that are passed around the entire project (like the User, Player, Lobby, Game, etc.) should be stored in the Common directory and available to all layers.

1.2: Events

Events are the backbone of this architecture. Each layer's API will essentially include a list of events that are available for other modules to subscribe to. Because of this, events should be standardized, implemented properly, and most importantly, well documented. A developer from front-end should not have to ask a member of the logic team when one of their events is fired, or what data it contains. This architecture will work its best when each team can clearly define the events they broadcast, and not have to worry about who receives them.

1.2.1: Abstract Events

All events in the Skribbage project inherit from the java.beans.PropertyChangeEvent class. However, it is required that any events that a team writes subclasses from some abstract event class that represents shared functionality among related events. For example, all events related to account management (login, change password, delete account) contain information about the user attempting the action, and are related in some way. Therefore, **none of these events subclass** PropertyChangeEvent directly. Instead, we define an abstract *AccountEvent* as part of a hierarchical tree.

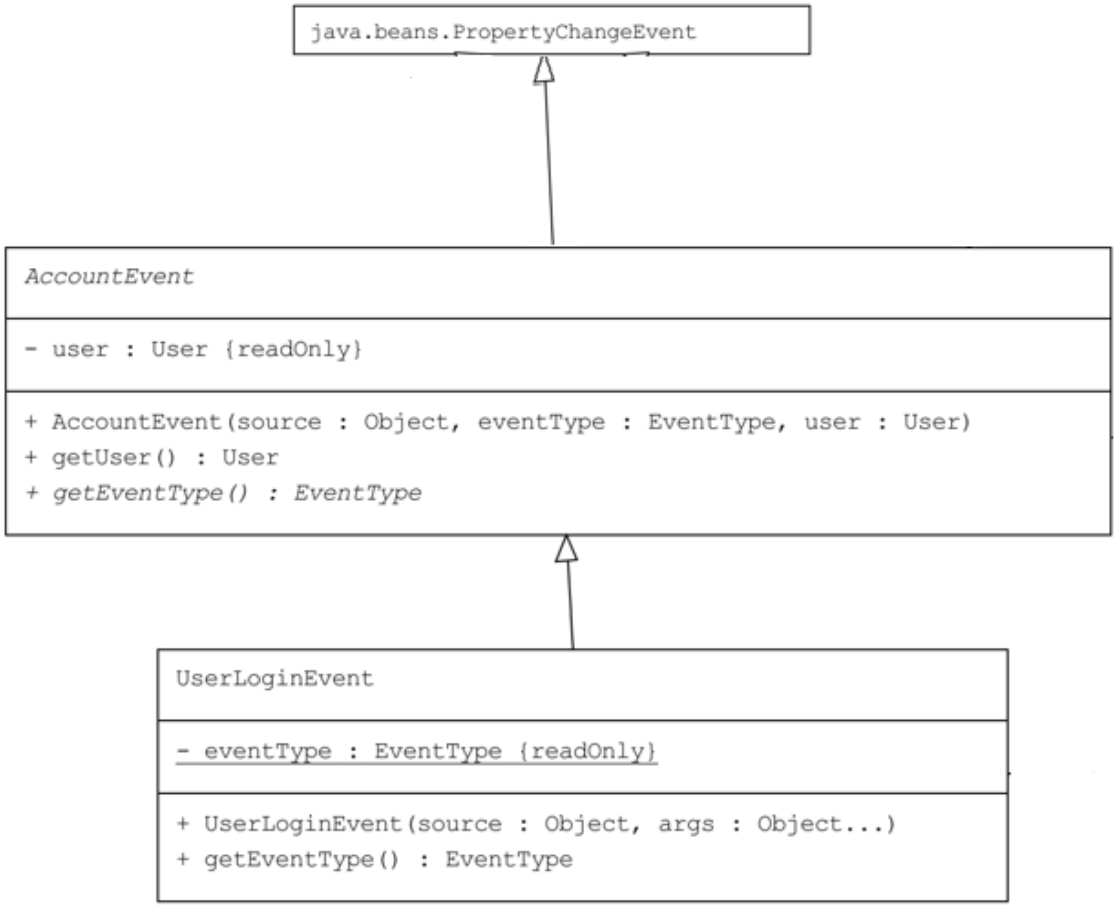

Figure) Concrete events should not extend PropertyChangeEvent directly, rather teams should design hierarchies of related events.

```java
public abstract class AccountEvent extends PropertyChangeEvent {
    /**
     * User executing operation.
     */
    private final User user;

    /**
     * Constructs a new {@code PropertyChangeEvent}.
     *
     * @param source        the bean that fired the event
     * @param eventType     the EventType enum entry that corresponds to this
     *                      event
     * @param user          User executing operation
     * @throws IllegalArgumentException if {@code source} is {@code null}
     */
    protected AccountEvent(Object source, EventType eventType, User user) {
        super(source, eventType.toString(), null, null);
        this.user = user;
    }

    /**
     * Shared by all concrete account events
     * @return User executing an account management operation
     */
    public User getUser() {
        return user;
    }

    /**
     * Abstract method to be overridden by concrete subclasses.
     * @return The EventType enum associated with the event.
     */
    public abstract EventType getEventType();
}
```

Figure) Implementation of abstract *AccountEvent* class. As we will see, this allows listener classes to reduce repeated code across related events.

1.2.2: Concrete Events

The concrete account management events directly subclass *AccountEvent*. All concrete events must store their EventType, which is part of the EventType enum we will see soon. Additionally, all concrete events must pass whatever shared data they contain with their superclass into a super call, then implement any additional information they contain. Since the UserLoginEvent only stores a user

(from its superclass), we will use an example that stores a user and additional information as our example of a concrete class:

```java
/**
 * A concrete implementation of an Event, representing the event associated
* with a user attempting to change their password.
 * @author Alex Carney
 */
public class UserChangePasswordEvent extends AccountEvent {

    /**
     * Proposed new password for user.
     */
    private final Password newPassword;

    /**
     * EventType enum for this class
     */
    private static final EventType eventType =
        EventType.USER_CHANGE_PASSWORD;

    /**
     * Constructs a new {@code PropertyChangeEvent}.
     *
     * @param source       The bean that fired the event
     * @param user         The un-authorized user associated with the event
     * @param newPassword  The proposed new password for the user
     * @throws IllegalArgumentException if {@code source} is {@code null}
     */
    public UserChangePasswordEvent(Object source, Object... args) {
        super(source, eventType, (User) args[0]);
        this.newPassword = (Password) args[1];

    }

    @Override
    public EventType getEventType() {
        return eventType;
    }

    /**
     * Get the proposed new password of a user, unique to this concrete
     * implementation
     * @return new password object
     */
    public Password getNewPassword() {
        return newPassword;
    }
}
```

Figure) An implementation of a concrete event. Ignore the ‘args’ argument for now. Also note that we use Password (which implements Payload) instead of java.lang.String to store a password

1.2.3: Event Type Enum

All **concrete** events must appear in the EventType enum class. For this architecture, we used enums as more than just a list of constants. Instead, the enum not only holds names of events, but also the *class types required to instantiate that event.* This means we can use a single factory class, along with the Java reflection API, to instantiate all events.

```
public enum EventType {
    /**
     * Fired when a user attempts to login from the LoginPage
     */
    USER_LOGIN(User.class),

    /**
     * Fired when a logged in user attempts to change their password from the
     * AccountPage
     */
    USER_CHANGE_PASSWORD(User.class, Password.class),

    /**
     * Fired when a user attempts to create an account from the
     * AccountPage
     */
    USER_CREATE_ACCOUNT(User.class),

    /**
     * Fired when a logged in user attempts to delete their account from the
     * AccountPage
     */
    USER_DELETE_ACCOUNT(User.class),

    /**
     * Stores the varargs for required data types for instantiation
     */
    private Object[] args;

    EventType(Object... args) {
        this.args = args;
    }

    /**
     * @return A list of class objects that represent required types for
     * event instantiation
     */
    public Object[] getArgumentList() {
        return args;
    }
```

Figure) The EventType enum class. Must be updated accordingly when teams add new events

Looking at the enum, we can start to make sense of the odd syntax inside the constructors of concrete events. This is a pivotal part of the architecture, which we will explore in more detail in Section 2.

1.3: Event Management System

Now that we have a class hierarchy of events, we want to use them to communicate information between different tiers. The common event management system takes care of this.

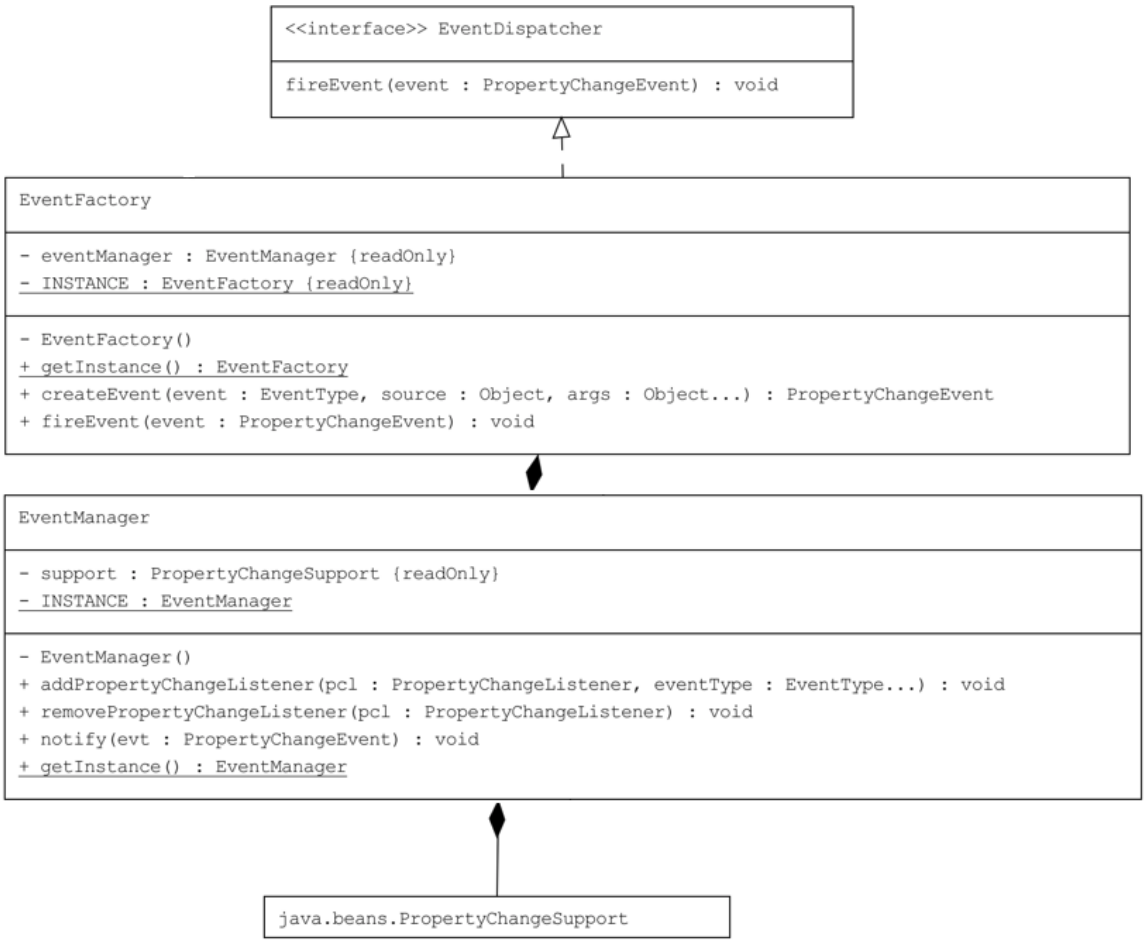

Figure) UML of event management system

In short, the EventFactory handles creation of events, along with notifying the EventManager that an event is to be fired[1]. The EventManager stores listeners according to what events they are subscribed to and distributes incoming events accordingly. Both classes are singletons and instantiated immediately when the program starts running.

1.3.1: Event Factory

Before I go any farther, I'd like to credit Sten Leinasaar for taking the lead on the event factory design and implementation (and its relationship with EventType enum). This was a fun problem to solve, and I'm glad we were able to work on it together.

The main functionality of the EventFactory class is the factory method createEvent(). Since the arguments for each event is stored inside the associated enum, we can instantiate *any* event from a single method. This reduces the amount of code we need to write immensely; however it puts a lot of responsibility on the developer. It is imperative that the EventType enum is properly updated with the required types for class instantiation, else a run time exception will be thrown. The factory method works as follows:

1) Parse through input parameters defined by Object… args. Ensure all parameters required for event instantiation are present.

2) Ensure all input data types implement Payload

3) Proceed with event creation, delegated to template methods that implement the concrete switch statement

[1] Since this breaks the single responsibility principle, we will likely refactor this so that the EventDispatcher is implemented by a separate class, rather than the factory. This will have no impact on use.

```java
/**
 * Creates an event of type PropertyChangeEvent based on the
 * EventType enum value being passed.
 *
 * @param event
 *            Type of an event as specified from the ENUM.
 * @param source
 *            Source that fired the update.
 * @param args
 *            Vararg of Object type.
 * @throws IllegalArgumentException If input types do not match enum
 * @return An event of type that was specified.
 */
public PropertyChangeEvent createEvent(EventType event, Object source,
    Object... args) {

    /*
    Before an event can be instantiated, the input parameters into the
    factory must be validated. This is done using the reflection API
     */
    Object[] eventArgumentList = event.getArgumentList();
    for (int i = 0; i < eventArgumentList.length; i++) {
        Class<?> clazz = args[i].getClass();
        if (clazz != eventArgumentList[i]) {
            throw new IllegalArgumentException(
                "Argument data types do not match enum"
            );
        }
        /*
        Require that all parameters are marked as a Payload
         */
        if (!Arrays.asList(clazz.getInterfaces()).contains(Payload.class)) {
            System.out.println("Warning: Using non payload, argument type " +
               clazz);
        }
    }
    /*
     * Now that input parameters have been validated, proceed with
     * event creation, delegated to templates defined by each module
     * These templates implement the switch statements for event
     * instantiation.
     */
        PropertyChangeEvent temp;
        for (FactoryTemplate subclass : this.templates) {
            temp = subclass.createEvent(event, source, args);
            if (temp != null) {
                return temp;
            }

        }
       throw new IllegalArgumentException("Event Not Found");
```

Figure) Single factory method for event creation. Note that console warnings will be replaced with exceptions in the actual project.

1.3.2: Factory Template

We realized that having a single factory method, although a clever solution to our problem, would cause some issues. Specifically, all 18 team members would update a single file and add a new method every time a new event was written. To circumvent the limitless merge conflicts that would arise from this, we incorporated the template method into the design. A portion of our factory algorithm (the switch statement that checks the event type and instantiates the proper event) is delegated to a separate class, of which there are four – One for each module. Therefore, each developer only needs to update their own team's factory template whenever a new event is added, rather than touching the common file.

```
/**
 * EventFactory private constructor.
 */
private EventFactory() {
    this.eventManager = EventManager.getInstance();
    this.templates = Arrays.asList(
        new LogicFactoryTemplate(), new GamificationFactoryTemplate(),
        new FrontEndFactoryTemplate(), new PersistanceFactoryTemplate());

}
```

Figure) The EventFactory singleton constructor takes care of instantiating each module's template. There is nothing stopping teams from adding more if extra separation of event creation is desired.

Each factory template extends the abstract *FactoryTemplate* class, which enforces implementation of an eventCreation() method (Which just serves as a hook for the EventFactory to call instead of defining all events inside of a single file).

```java
/**
 * OverWritten eventCreation method from FactoryTemplate. This overwritten
 * method will only manage events created by front end.
 */
@Override
public PropertyChangeEvent eventCreation(EventType event, Object source,
    Object... args) {

    switch (event) {
        case USER_LOGIN:
            LOG.trace("Returning: " + event.getName());
            return new UserLoginEvent(source, args);
        case USER_DELETE_ACCOUNT:
            LOG.trace("Returning: " + event.getName());
            return new UserDeleteAccountEvent(source, args);
        case USER_CREATE_ACCOUNT:
            LOG.trace("Returning: " + event.getName());
            return new UserCreateAccountEvent(source, args);
        case USER_CHANGE_PASSWORD:
            LOG.trace("Returning: " + event.getName());
            return new UserChangePasswordEvent(source, args);
        default:
            LOG.warn("Event passed was not one of Front End events");
            return null;
    }
}
```

Figure) The front-end team's actual implementation of our template method

1.3.3: Event Manager

Once an event has been instantiated, an operation must handle broadcasting and notification to subscribers that an event has been fired. The EventManager singleton class has this responsibility. Using the PropertyChange library built into Java, the EventManager registers listener classes, and notifies them when an event they are subscribed to is dispatched.

```java
/**
 * Registers a new property change listener, mapped to a specific event
 * type.
 *
 * @param eventType
 *            From the EventType enum.
 * @param pcl
 *            A new instance of an event listener, which implements
 *            PropertyChangeListener
 */
public void addPropertyChangeListener(PropertyChangeListener pcl,
    EventType... eventType) {
    for (EventType event : eventType) {
        this.support.addPropertyChangeListener(event.toString(), pcl);
    }
}

/**
 * Unregisters a property change listener.
 *
 * @param pcl
 *            The event listener (implementing PropertyChangeListener) to
 *            unregister
 */
public void removePropertyChangeListener(PropertyChangeListener pcl) {
    this.support.removePropertyChangeListener(pcl);
}
```

Figure) Listeners can be registered and unregistered from the EventManager.

Since addPropertyChangeListener() accepts a vararg of EventTypes, a listener can subscribe to any number of events at once. The ability to unsubscribe from a subset of this list and subscribe to more events after the initial method call will soon be supported. For now, listeners are registered as soon as the program starts, inside the main method. (See Section 1.4 for more information about listeners)

```java
// Inside entry point method of project, after EventManager instance has been
created
eventManager.addPropertyChangeListener(
    new AccountListener(),
    EventType.USER_LOGIN,
    EventType.USER_CREATE_ACCOUNT,
    EventType.USER_CHANGE_PASSWORD,
    EventType.USER_CHANGE_PASSWORD,
    EventType.USER_DELETE_ACCOUNT);
```

Figure) Initial subscription should take place in the main method. Modifying subscription during run-time will soon be supported.

Using the PropertyChange library, Java handles most of the heavy lifting for us. The mapping between EventType and listener is delegated to the PropertyChangeSupport class, which makes notification of listeners of an incoming event very simple:

```
/**
 * Fires the input event, for all registered listeners to handle.
 *
 * @param evt
 *            Event to fire - Any subtype of CribbageEvent (or
 *            PropertyChangeEvent)
 */
public void notify(PropertyChangeEvent evt) {
    this.support.firePropertyChange(evt);
}
```

Figure) Notify alerts all listeners of an incoming event.

1.3.4: Event Dispatcher

We've now seen the two main components of event management – Creation (EventFactory) and notification (EventManager). The final component is the EventDispatcher, which only has the job of notifying the event manager that an event is to be fired, so that classes that are using the event system don't have to refer to the EventManager singleton instance directly. In the current version of architecture, the EventFactory also acts as the EventDispatcher, responsible for sending an event to the EventManager to be fired. In the future a separate class implementing EventDispatcher functionality may be written.

```
UserLoginEvent ule =
    (UserLoginEvent) eventFactory.createEvent(
        EventType.USER_LOGIN,
        this,
        fakeUser
    );
eventFactory.fireEvent(ule);
```

Figure) The EventFactory creates an event, and can fire said event

1.4: Listening for Events

Now that we have a way to send events across the entire project, we want to be able to use the data stored in them. Listeners are classes that implement the java.beans.PropertyChangeListener interface, and make up the other part of each team's interaction with the architecture. Each team will write a collection of events that manage outgoing data to other tiers, along with a suite of listeners to handle incoming data.

The implementation of listeners is completely up to each team. The architecture does not care who receives events, so while a standardized way of writing listeners is good for code readability and design, it is not necessarily required for the system to work. This means each team has some freedom in how to implement listeners. A team may choose to write a separate listener for every individual event, or a listener that can handle multiple, related events (like a router or controller in web development)[2]. For the sake of example, I will use an example of the latter implementation; a single AccountListener to handle the various account management events.

```
public class AccountListener implements PropertyChangeListener {
public AccountListener() {
    this.eventFactory = EventFactory.getInstance();
}
@Override
public void propertyChange(PropertyChangeEvent evt) {
/* Toy functionality that is shared among all events
* handled by this controller. The benefits of a class
* hierarchy manifests itself here
*/
User associatedUser = ((AccountEvent) evt).getUser();
hashPassword(associatedUser);
boolean userIsValid = validateUser(associatedUser);
}
```

Figure) A toy example of a listener that handles a family of events. Recall that this listener was instantiated in the main method, along with the list of EventType enums to listen for (Section 1.3.3)

[2] Another interesting implementation, one could use the chain of command pattern to handle and transform events before their destination, sort of like how middleware is used in web development.

By utilizing upcasting and downcasting, listeners can be very flexible, and written to an individual team's needs.

```
// Inside AccountListener class
AccountEvent accountEvent = (AccountEvent) evt;
switch (accountEvent.getEventType()) {
    case USER_CHANGE_PASSWORD:
        // Do something
    case USER_CREATE_ACCOUNT:
        // Do something else
    case USER_LOGIN:
        // Fire an event back even
        this.eventFactory.createEvent(EventType.USER_LOGIN_RESPONSE);
    default:
        // Something must have gone wrong, plan for the "impossible"
}
```

Figure) Routing related events based on their event type. Allows events to be sorted without using the instanceof keyword

Section 2: Usage

This section essentially serves as a checklist for the steps a team should take when trying to communicate with other modules.

2.1: Checklist for writing a new Event

2.1.1: Design

Before writing any code, the individual or team should design the sequence that their event will be a part of. There are a few questions that need to be answered in the design:

When will the event be fired?

In the case of the front-end team, all our events will be fired from callback functions from GUI components. For example, if a user enters a username and password, the LoginPage class will detect that and create + fire the appropriate event.

Will I need to wait for the response to this event? Is this event part of a request/response cycle or just fired without caring about a response?

Once again, in the case of the front-end team, once we fire a UserLoginEvent, we are expecting a response to come back to us at some point. What to display on the screen during that waiting period, what to do if a response times out, is cancelled, etc. are all follow-up questions that need to be answered. If a request/response cycle is in order, one should probably consult the team members of the module to communicate with. A listener/event combination may be required in another module for the entire process to work properly (i.e: logging in, which may even require requests/responses between three tiers!). I suspect many events will require this sort of two-way communication.

What 'family' of related events will my event be a part of? What data/functionality is shared among all those classes?

Just like how every *AccountEvent* has a user, or every *LobbyEvent* has a lobby reference, a class hierarchy may need to be designed before implementation of the event. If a concrete event extends PropertyChangeEvent directly, there should be a very good reason for doing so. This removes all the advantages of upcasting that were present in the AccountListener in Section 1.4, along with potentially restricting future events that may end up being related.

Once the design has been established, including either a sequence diagram and/or UML, code may be written.

2.1.2: Write or extend abstract class

From the design, it should be very clear how this new event falls into an existing (or new) hierarchy of events. Extending an existing abstract class is as easy as a super() call (inside the concrete subclass), but writing a new one may take some time. Refer to section 1.2.1 for an example of an abstract *AccountEvent* class.

2.1.2.1: To null or not to null

Inside of the example in section 1.2.1, you may notice that in *AccountEvent's* super call to PropertyChangeEvent, two fields are null:

```
protected AccountEvent(Object source, EventType eventType, User user) {
    super(source, eventType.toString(), null, null);
    this.user = user;
}
```

These fields represent "Object oldValue" and "Object newValue" present inside the superclass PropertyChangeEvent. Since this is part of the java.beans package, we can't modify this class, so these two fields are technically required. I've been setting them to null inside of all my subclasses, which technically breaks Liskov's substitution principle (one of the core SOLID design principles), which is a design flaw. However, I just can't seem to find a use for oldValue and newValue. They are present because PropertyChangeEvents are supposed to be used differently than we are using them (once again, a design flaw on my part). I would recommend continuing to null these parameters and not calling getOldValue() or getNewValue() anywhere, unless there is a good use for it.

2.1.3: Determine parameters for the event

Simply write down (or refer to your UML, where this question should be answered) the required parameters for this new event. For example, a UserChangePasswordEvent requires a User input and a Password input. **Note: Inputs cannot be primitive types and must be tagged with the Payload interface. That's why we use a Password object and not a String for UserChangePasswordEvent**

Refer to section 1.1 for more information about the Payload interface.

2.1.4: Add the new event to the EventType enum

This step is interchangeable with 2.1.4. In fact, you should probably do these steps at the same time. As convention, the enum name of your event should be camel case, and identical to the event class name (without the word 'event' at the end). Example: UserCreateAccountEvent -> USER_CREATE_ACCOUNT

Note that to add the event to the enum, you must be aware of what parameters your event will take. Add each parameter to the parentheses after the enum name with the .class of the type the event will take. **Note: Order matters, and you need to know what order you placed the types in**

```
USER_CHANGE_PASSWORD(User.class, Password.class),
```

Now when I create this event, I know that args[0] will be a User object, and args[1] will be a password.

2.1.5: Write a concrete class

Once it is clear how an event falls into a hierarchy, its concrete implementation can be written. Refer to Section 1.2.2 for an example of a concrete event class.

Be mindful of the order of parameters in the enum. This is where it matters. Casting to the correct type may be required in super() calls and while setting instance variables

All events must have the same constructor signature:

```
public UserChangePasswordEvent(Object source, Object... args) {
    super(source, eventType, (User) args[0]);
    this.newPassword = (Password) args[1];
}
```

2.1.6: Fire the event

This step is completely dependent on the context of the event. This question should have been answered in the design, ("when will the event be fired"). The final step is to use the event factory to create and fire the event, passing in whatever parameters are related to the event context.

```
// Tracer bullet. Should succeed
User fakeUser = new User(
    "acarney@skidmore.edu",
    "ACarney",
    "password",
    false);

UserLoginEvent ule =
    (UserLoginEvent) eventFactory.createEvent(
        EventType.USER_LOGIN,
        this,
        fakeUser
    );
eventFactory.fireEvent(ule);
```

For a front-end example, the input username and password are packaged into a User object, which is then passed to the EventFactory for instantiation. Passing the newly created event into fireEvent() is the last step you need to take.

Note: The 'source' parameter in an event should almost always be 'this', which just means that the receiver of the event can determine where the event was fired from, if necessary.

Be careful when calling the createEvent() method. Your IDE will not provide introspection into the required parameters for creating an event, nor will it warn you if the parameters are incorrect or out of order. This makes unit testing an *absolute priority*.

No operation involving events works properly until it is unit tested.

2.1.7: Handling a response

In the design, you should have answered the question of whether your class requires a response from the event you sent out. If no response is necessary, then ignore this section. Otherwise, another design challenge is ahead of you (although most of the work should be already done if it was in the design). The notion of 'callback functions' may be of use here, and most likely a separate listener waiting for a 'response event' may be required. See section 2.3 for more information about handling request/response pairs, which relies on the process of writing listeners from section 2.2.

2.2: Checklist for writing a new listener

2.2.1: Design

Before writing any code, a sound design is required. Just like with events, there is a set of questions that should be answered and planned out in UML or sequence diagrams before continuing.

What will this listener listen for?

In other words, what is the purpose of this listener. There is a large difference in the implementation of listeners that handle a single event versus multiple, so ensure to plan for what events (or at least, what family of events, defined by a proper class hierarchy) will be handled by this listener.

How will the listener respond?

This is where UML is extremely important. Assuming the events come in from some disconnected source (the UML should not care about where the events are coming from), how does this listener play into the larger design of your team's module?

For example, an AccountListener should connect to the DatabaseFacade, and have some way of sending events back to be listened to (we discuss request/response cycles in 2.3)

(If listener handles a family of events) How will shared functionality be implemented?

This was the entire point of allowing listeners to subscribe to multiple events, and setting up the class hierarchy of related events. The benefits of all events related to account management subclass *AccountEvent* is that a listener can upcast an incoming event to *AccountEvent*, do something (like get the User, which is shared among all subclasses), then potentially downcast the event and route it, based on EventType (See Section 1.4 for more detail)

What operations need to be performed on *all* the incoming events, versus what operations need to be specific to the concrete implementation of the incoming classes. The goal is to try and reduce repeated code as possible, so try to think DRY.

2.2.2: Implementation

Refer to Section 1.4 for an example of a concrete AccountListener class. The single absolute requirement of a listener class is that it *implements* PropertyChangeListener. This will ensure that the concrete class implements the properyChange() method. Inside the propertyChange() method is where the code to handle the incoming events is written.

Note: The signature of propertyChange() cannot be changed. This means that the method will always be passed an event object of type PropertyChangeEvent. Since all events are subclasses of this, downcasting will almost always be required to achieve desired results.

If you need to route different events based on their type, sort incoming events by their EventType, not using instanceof.

2.2.3: Register your listener

Before your listener can work, it must be registered. Inside the main method of the project (or wherever classes are instantiated for the first time, if it is no longer in the main class) make sure to add the following code:

```
eventManager.addPropertyChangeListener(new YourListener(),
EventType.EVENT_ONE, EventType.EVENT_TWO, ...);
```

This instantiates the listener and subscribes to a variable number of defined event types.

2.2.4: Test the listener functionality

The only requirement for a class to be a listener is that it implements the PropertyChangeListener interface. Therefore, it is easy to create a MOCK listener class to unit test. In fact, this step could even be completed before writing the concrete listener class implementation.

Inside the @Before section (the setup method) of your test, get the instance of the event factory, setup whatever payload you need to test your listener, and instantiate a mock listener instance.

```
@Before
public void setUp() throws Exception {
    eventFactory = EventFactory.getInstance();
    eventManager = EventManager.getInstance();

    //Create listener instance + register it
    eventManager.addPropertyChangeListener(new LogInListenerMOCK(),
        EventType.USER_LOGIN);
}
```

Inside the @Test portions of your unit test, ensure that

1) The listener was properly instantiated
2) Neither instance of Factory/Manager is null

To test an event/listener combination, create a mock payload and a mock event, fire it, and write a test to detect if the listener caught the event. The implementation of asserting that the event is caught is up to the programmer

```java
@Test
public void loginEventIsCaught() {
    // Mock payload
    User mockUser = new User("acarney@skidmore.edu", "acarney", "password",
        UserRole.UNAUTHORIZED);
    // Mock event
    UserLoginEvent event =
        (UserLoginEvent) eventFactory
            .createEvent(EventType.USER_LOGIN, this, mockUser);
    // Fire it
    eventFactory.fireEvent(event);
}
```

Remember, code does not work until it is tested.

2.3: Considerations for Request/Response configurations

Sometimes, we don't just want to fire an event and not care about what's done with it. When a user logs in and a UserLoginEvent is fired, the front end wants to know what happens to that event. The true power of the event system shines when it is used to handle asynchronous requests between different tiers of the application.

If a login event is fired, the front-end can continue to process operations from the user without locking up. Only when the "response" is handled by whoever is listening does the front-end page respond to the user's login request. This ensures that the client is never locked out of interacting with the program (at least not when making simple requests).

2.3.1: Design Questions

Request/response configurations require two-way communication between different modules. Communication between developers on separate tiers is pivotal to get this to work.

UML representing cross-communication:

The full UML that represents the event management system incorporates a two-way communication protocol between the front-end and logic layer. Each tier is perfectly symmetric, with a family of events on both sides, along with listeners that are waiting for the other side's events.

Although the use of the words "request/response" in the events were discouraged by the professor, it is a good starting point for how to think about what each event represents.

For example, a UserLoginEvent from front-end should have a counterpart response, such as UserLoginSuccessEvent from the logic layer. Likewise, the Logic layer would write a full listener to subscribe to all *AccountEvents* from front-end, while the front-end would in turn write a listener to subscribe to all *AccountSuccess* (or *AccountResponse*) events from the logic tier.

When planning a system like this, try to incorporate *symmetry* into the design. Refer to the full UML of the event architecture system to see this in action

What will the requestor do during the downtime?

The purpose of utilizing an event-based system instead of synchronous calls is to stop the requestor from being locked while the request is handled. Although a login event would in theory be nearly instant, we don't want to lock up the entire application while the request is being processed, especially if something goes wrong for some reason.

Therefore, it is important to consider what will happen to the application while the request is being handled. For example, the front-end could close the dialog box for logging in while the request is being handled, so that the user can click around and do other things until the request is processed. Once again, these cycles should be nearly instant, but it's still beneficial to plan for the worst.

This leads directly to the next question:

What will the requestor do with the response?

In event driven architecture there is a notion of **callback functions,** methods that are "called back" once an asynchronous process has been resolved. This is an important question for the design of the team sending the request – How will the class that sent the event know when it has been fulfilled? What will be done with the resulting data?

Most likely, I suspect that classes that fire events will contain a callback function that is called from the *listener* that subscribes to the response event tied to the request.

For example, a LoginPage that fires a UserLoginEvent should have a loginCallback() method that is **called from the AccountResponseListener**, which handles the incoming response from the logic tier accordingly (tell the user that their operation failed/succeeded, change the screen to show they are logged in, etc.)

Once again, UML is extremely important to answer these types of questions.

Acknowledgements

I'd like to specifically thank the entire front end team for feedback and support while coding this system. I'd especially like to mention Sten Leinasaar for playing a pivotal role in the system's design and architecture. Additionally, none of this would have been possible without Professor Read's help, especially with how to integrate standard patterns into the design.