By Stephen Huang, Eric Tsai, and Jason Ya

We all worked equally on this project. Eric worked on the android programming and helped with documentation. Jason worked on documentation and helped with the android programming. Stephen worked on WebService, CakePHP and helped with android programming and documentation.

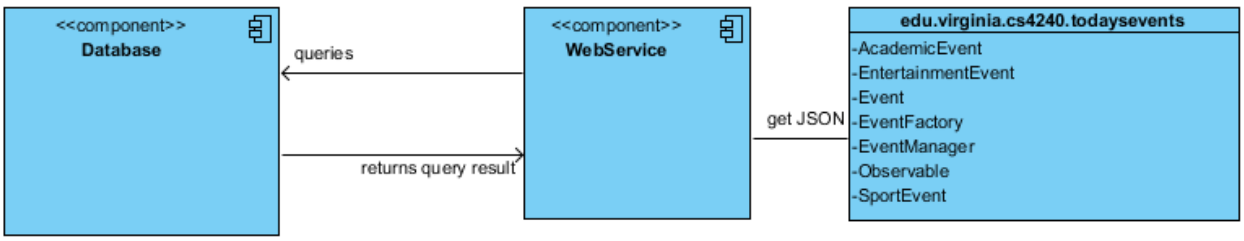
Introduction

The system that we have designed for this project is an events framework for an Android application that allows users to stay up-to-date with events on their campus or other location. The framework is designed to be flexible and customizable such that designers who wish to use it on their campus can deliver their specific list of events to end users.

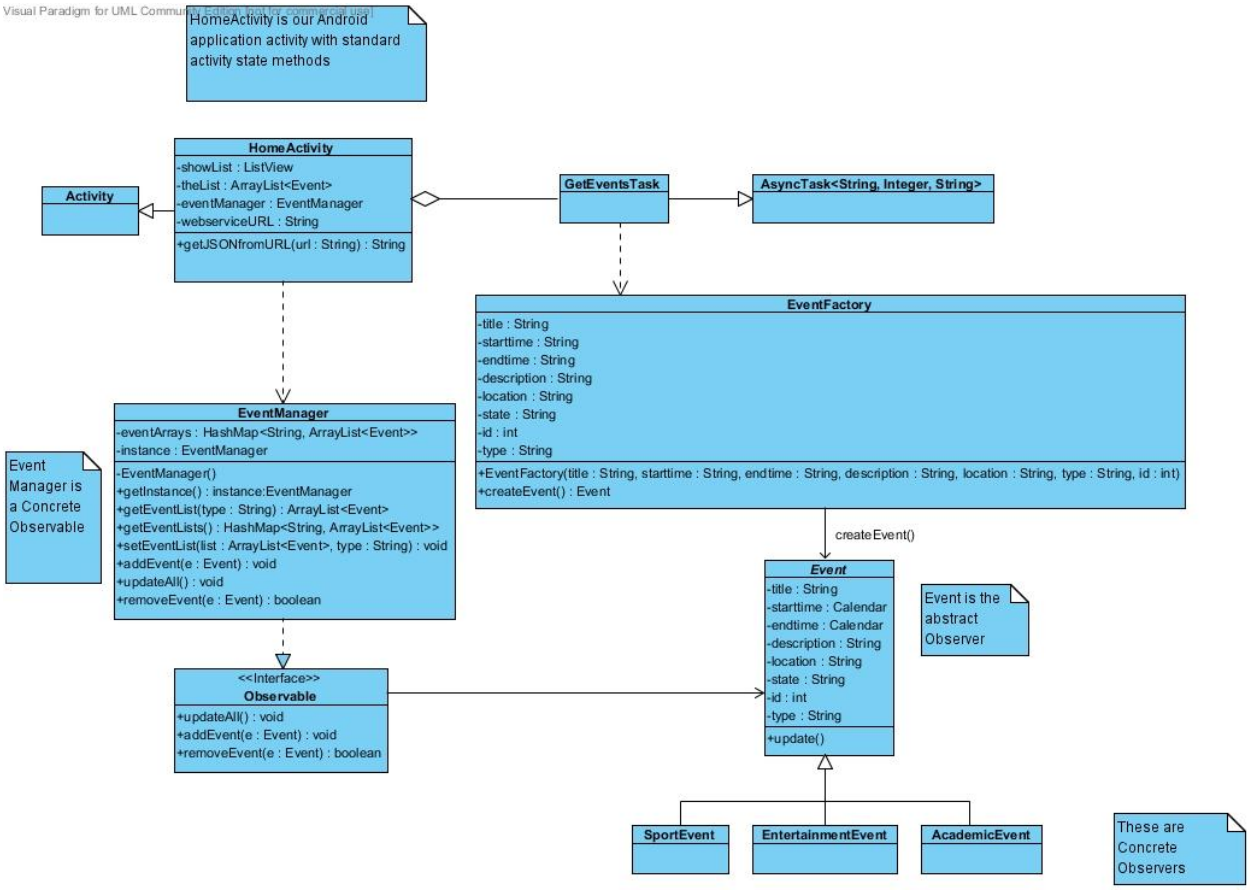
High-level Design/Architecture

Our system is composed of a single stand-alone package that acts as a general framework for users to customize based on their needs. On an architecture level, our design is composed of three components that serve as the framework of the program. The first of these components is the data source, which in our specific implementation of the framework, is a MySQL database containing information on recent events on the UVA campus. The second component is the WebService component, which queries the database to request updated information. Upon receipt of this information, the WebService component converts the returned query data into JSON format for output. Depending on the data source the user has available, the database can be replaced with another type of information source, for example, a live updating feed, as long as it can properly interface with the WebService component. The framework created for this project is largely contained within the third component, which is a package that parses the JSON data from the WebService to create a displayable list of events.

In the high-level design of this system, we focused on modularity of components and a design that would be flexible for various sources of input. To achieve this, we included the WebService component to act as a sort of filter between the input data and the processing package. By doing this, we ensured that the package could focus solely on event managing, knowing that its input data would consistently be in JSON format for simple handling. Likewise, the event handling is restricted to the package whereas the WebService simply acts as a data filter. In this way, each component has strong cohesion and is loosely coupled to the other packages.



Additional Design Documentation



Events Database

-Brief Justification

The database used for our android application is a MySQL relational database that we used to contain dummy events. The dummy events are prototypes for what later could be replaced by methods of information gathering like user-generated input, screen scraping and data mining from social media, relevant websites, etc. With the time constraints of the project and a lack of available sources for information gathering, the dummy events are considered the most suitable for the project.

-Database Specifications

The key-value pairs for our Events database include title (string), starttime (date-time), endtime (date-time), description (string), location (string), type (string), id (int). Likewise, in our android application, an object that contains these fields is thus considered an Event. The keys are self-explanatory and are justified by the necessities of what information is sufficient for clients when they view our Events application. Note that due to the properties and rules that dictate relational databases, no two entries are permitted to have the same id.

Example of Database Implementation:

title	starttime	endtime	description	location	type	id
Basketball game	2012-12-08 16:00:00	2012-12-08 18:30:00	UVa vs MVS	JPA	Sports	1
Chamber Singers	2012-12-08 20:00:00	2012-12-08 23:00:00	Free concert, featuring Three Notch'd Road	Old Cabell Hall	Entertainment	2
…	…	…	…	…	…	…

Events Webservice

http://plato.cs.virginia.edu/~sh4fd/cake/events/main

-Brief Justification and Specification

The Webservice provides means of obtaining and providing information between user input through the web browser and our MySQL database. The Webservice is written in the CakePHP Framework, which uses the Model-View-Controller architecture for web services. Use of REST principles allow users to interact with information through the URLs of the web service, which is relevant to how our android application functions in obtaining data from our Events database.

-Webservice Function

The primary URL used for our android application provides information in the form of JSON formatted data of all events that are currently stored in our Events database. The JSON format essentially treats an object as a formatted array of the corresponding key and value pairs for that object entry in our database. The decision to use the JSON format lies in the ease of creating the format structure as well as parsing the format in our android application, and remains robust because JSON is interoperable, extensible and an overall better data exchange format. XML was an alternative choice, the consequence being the need to write a set of rules in structuring the XML and parsing it that will only grow larger and larger if more keys/fields were to be included in the future.

Java Events Creation

In our android application, we provide the URL that returns all events in JSON formatted text, and then using the Gson library, the events are then parsed and fed into the factory to create the proper Event objects based on the type field. These events objects are then added to the event list in our Event Manager for centralized management. Because there is no direct equivalent of Date-time in Java, we decided to read in the starttime and endtime of each event as strings, and then using SimpleDateFormat converted them into Calendar objects and created Java Event Objects with Calendar fields starttime and endtime.

-Event Class Specification

Our Event Class is an abstract class with several fields already defined and with the method update(), since it is our abstract Observable. The Concrete Observables, and thus the

event objects our Event Factory creates, set their own types to be the corresponding string. The intent is that for further development down the line, each event is responsible for certain interfaces specific to those kinds of events. As mentioned before, the Events class contains Calendar type starttime and endtime, mainly for the purpose of being able to compare the current time to the set time range of the events so the Observables are able to update their state, which is returned as a string with choices being: upcoming, ongoing and ended. The EventManager is our Observer, and updates the list view in our android application with the corresponding status on the events list. Once again, the intent is that for further development, we might perhaps introduce something like a State design pattern where with changes in state, the event object will provide the user with different options and function differently. For example, sport events won't offer the option to view scores if the event hasn't started, nor would an entertainment event continue to sell tickets for the event once the event is over.

Reflection and Design Highlights

The purpose of the events application is to provide users a simple way to find events and provide details such as location, date and event status. In order to manage and handle requests to search for events through our events Webservice, the Singleton design pattern was a logical choice to be our Event Manager. We needed an Event Manager that contains all event information and displays them to the client, as well as updates event information. By providing only a single instance of the Event Manager, we avoid conflicting interests that might occur if we had multiple Event Managers, such as one manager trying to edit an event that another manager had removed.

Originally, Event Manager had hardcoded linked lists for each type of Event but we realized this was bad design because Event type could vary. Therefore, in order to encapsulate this variation, we used a HashMap<String, LinkedList<Event>> instead of hard coding linked lists for each type. The hash map uses Event type as its key. This way, Event Manager can add any type of Event without code modification.

Event information is obtained from the Webservice, but once it is parsed for use in the android application, it must be molded into an Event object, and to go further, into a type of an Event. To manage the events creation from populating the attributes of the object to creating the correct type of Event, the Factory design pattern is used to construct the Event Factory. The factory is an object that instantiates/creates the proper type of event object and hides the instantiation from the client. Then the Event Manager stores the object into a data structure that contains Events (the abstract data type), thereby encapsulating the type of the event.

Another design pattern that we used was the Observer design pattern, for status updates of our events. Every event has listed start and end times and the accompanying string status that indicated whether the event is upcoming, ongoing, or has ended. The status is determined by comparing the current time to the start and end times, so once a certain condition is met, the change in status must be updated in the Event and in the Event Manager. Consequently, our event manager will need to handle many events and handle the status changes; so the observer pattern is a good fit for such a task. The Observer design pattern solves the potential issue of

having several different update calls for each of the Event types (Observers). By implementing the Observer design pattern, we create a situation where updates only require a single call from EventManager, the class where the change occurs. This single update call is then propagated to each of the Observer classes, avoiding a situation where each of the Observers must continue querying the Observable class to find out when a change has occurred.

An issue we ran into while implementing the design of our system was the importance of creating a modular design. While retrieving information from the database for conversion into the JSON format, we found that the JSON format needed a class that directly matched the database type (DATETIME) in order to properly store the retrieved data in JSON format. Because the Java Calendar class could not properly match the DATETIME format, we had to utilize a different container to hold the information. Our solution was to use the String class to hold the information. However, this created another problem as comparisons between different dates/times did not work properly when the comparisons were done between two Strings. To solve this, we used the Factory to create the events with this information in Calendar type instead of String type. Therefore, by using the Factory design pattern, we were able to hide the details of the implementations related to create any type of event.

In terms of high level design patterns, a key feature of our design is a framework that can be redesigned by users to suit their purposes. As a result, it was important to make our design modular and easily adaptable to a number of situations. Our high-level approach to achieving this was to separate the responsibilities of the system into three distinct categories similar to the Model-View-Controller (MVC) paradigm. Instead though, we used a Model-View-Adapter (MVA) paradigm.

The Model is essentially the database or other data input source, which contains the latest state of events (not necessarily in the proper formatting). The Adapter is the WebService component, which processes the information from the database or another input source and sends that information as a command to the View to update the list of events. The View is composed of elements/classes from the TodaysEvents package, namely the classes that requests data from the WebService and the Android display elements that allow the application to create a proper output view on the application screen. The High-level diagram outlines the MVA relationship.

The reason why we use MVA is because it decouples the relationship between model and view. In MVA, the Adapter (WebService) is responsible for all interactions between Model and View. This way, a user who wants to use our framework can use a different database or re-organize data without adversely affecting the View. Also, if the user wants to add to the view, or has a view/gui of his own, he can easily do so since Model and View are loosely coupled.

By modeling our system after the MVA pattern, our system can be separated into components that have distinct responsibilities. As a result, if a user wants to use our framework to implement his Events application, the framework is easier to understand because it is based on the established MVA model.