

Project 1

CMSC 433
Programming Language Technologies and Paradigms
Fall 2003

Due September 17, 2003 at 6pm

Updates to this writeup are listed in Section 5.

In this project, you will create a simple infrastructure for processing events of interest in your programs, purposes of debugging or analysis. All classes must be placed in a Java package called `cmsc433`.

1 Events

We are interested in collecting and processing notable events in a program, like method entry and exit, thread creation, variable value changes, etc. To do this, we have to define a representation for events. We do this with two classes: events are instances of the `Event` class, which consists of one or more instances of the `IR` class.

1.1 The IR class

An IR is a information record. Actual events are composed of one or more IRs. Each IR contains two pieces of information: the `eventName` and the `eventValue`. For example, if you wanted to create an event to indicate that the `String.valueOf()` method was called, you could create a IR with an `eventName` of “MethodEntered” and an `eventValue` of “String.toString()”.

The public methods of this class are shown below:

```
public class IR implements java.io.Serializable {
    public IR (String eventName, String eventValue);
    public String getName();
    public String getValue();
    public String toString();
}
```

`IR.toString` returns the string representation of an IR, which is defined as the string representation of the `eventName` followed by a “:” character, followed by the string representation of the `eventValue`.

The `eventName` and the `eventValue` strings may not contain the “+” or “:” characters. Any attempt to create an invalid IR should result in a `MalformedIRException` being thrown from the constructor. This is an unchecked exception, since it extends the `RuntimeException` class:

```
public class MalformedIRException extends RuntimeException {
    public MalformedIRException (String errorMsg);
}
```

The message passed to the constructor should indicate the problem that occurred.

Note that this class implements the interface `java.io.Serializable`; doing so does not add any further programming obligation on your part. Objects that implement this interface can be serialized automatically and copied across the network; we will exploit this functionality later on.

1.2 The Event class

An event, which is an instance of the `Event` class, consists of one or more IR instances. For example, if you wanted to log the time when a program entered the `String.valueOf()` method, you can create two IRs. The first one could have `eventName` “MethodEntered” and `eventValue` “String.toString()”, while the second one could have `eventName` ”TimeStamp” and `eventValue` = *time*. (Assume that *time* is the text representation of result of calling `getTime()` on an instance of `java.util.Date()`).

The public methods of this class are shown below:

```
public class Event implements java.io.Serializable {
    public Event(IR ir);
    public Event(Event rec, IR ir)
    public java.util.Iterator iterator();
    public String toString();
}
```

Events are created with either of two constructors. The first creates a fresh event having a single IR. The second creates a new event from the IRs of an existing event, and one additional one. Your implementation should ensure that the creation of the new event does not alter the definition of the existing one. In particular, if you have

```
Event e = new Event(new IR("greeting","hello"));
String sbefore = e.toString();
Event e2 = new Event(e,new IR("time","1130"));
String safter = e.toString();
```

The two strings `sbefore` and `safter` should be the same.

The string representation of an `Event`, which you must implement in `Event.toString()` method, will be the string representation of each of its internal IRs separated

by a “+” character. *The order of the IRs in this string should be the order in which they were added to create the Event object.* `Event.iterator()` returns an object that implements the `java.util.Iterator` interface and that iterates over the IRs contained in the Event (again, in the order they were added). You can assume that the `Iterator.remove()` method will not be called, and therefore do not have to implement it; that is, simply have it throw `UnsupportedOperationException` (see the Java API for `java.util.Iterator` for more on this exception). Like IRs, Events can be serialized.

2 Processing Events

Events, by themselves, don’t do very much. We need other objects that will process the Events in interesting ways.

2.1 The EventProcessor interface

Because many different kinds of application classes may want to process Events, we will use an interface (rather than a class) to define objects that process Events. The EventProcessor interface is defined as follows:

```
public interface EventProcessor {
    public void process (Event rec) throws ProcessException;
}
```

The idea is that each class that implements EventProcessor will do something different in its `process` method, as part of handling events. The `process` method might throw an exception (for example, RemoteClient and RemoteServer, defined below, must communicate across the network, so it is possible that they will encounter problems and throw exceptions). To accurately handle the variety of exceptions that could be thrown, we will use the Java 1.4 *chained exceptions* mechanism. In particular, we will create our own exception class, `ProcessException`, with a two-argument constructor `ProcessException (String message, Throwable cause)`. Any exceptions that cannot be handled locally in the `process` method should be wrapped in a `ProcessException` and rethrown. The original exception is provided as the second argument of the constructor. Any code which catches this exception can get the original cause using the `getCause()` method, defined in `Throwable`. The on-line Java API reference for `Throwable` provides a good explanation of this technique.

2.2 Concrete EventProcessors

As part of this assignment you will write several classes that implement EventProcessor. If done properly, anyone (including professors and TAs) should be able to write concrete EventProcessors and integrate them immediately into your system.

Printer The Printer event processor writes the Event's string representation to the `java.io.PrintWriter` that was passed into its constructor, calling its `println` method.

```
public class Printer implements EventProcessor {
    public Printer(java.io.PrintWriter o);
    ....
}
```

Dispatcher A dispatcher's processing responsibility is to forward incoming Events to any other EventProcessors. Any EventProcessor that wants to be notified of incoming Events must call a dispatcher's `attach` method, passing itself as a parameter. When the dispatcher's `process` method is called with some Event *e*, it will call the `process` method of all EventProcessors that attached to it, passing them *e*. This should happen in the order that the EventProcessors were attached.

```
public class Dispatcher implements EventProcessor {
    public Dispatcher();
    public void attach(EventProcessor ep);
    ....
}
```

Filter A filter compares the string representation of the events it processes against some regular expression. Those events that match this expression either partially or completely are then forwarded on to other EventProcessors that are attached to the filter; otherwise the events are dropped. For example, an event with string representation "food:burger" will match the regular expressions "urg", ".*urg.*", and "urg??" (among others). A filter's forwarding capability can be inherited by having `Filter` extend the `Dispatcher` class, defined above. The constructor will take a `String` as a parameter, which represents a single regular expression. Use the `java.util.Regex` package to define regular expressions and perform pattern matching.

```
public class Filter extends Dispatcher {
    public Filter (String regex);
    ....
}
```

TimeStamper A timestamper will create a new Event that adds an IR containing the current timestamp to the given Event. The `eventName` of the added IR should be "Timestamp", and the `eventValue` should be the string representation of a call to `java.util.Date.getTime()`.

```
public class Timestamper extends Dispatcher {
    public Timestamper();
}
```

```
    ....  
}
```

RemoteClient and RemoteServer This pair of EventProcessors is used to transport events across a network for remote processing.

Let's start with RemoteClient. The constructor for this class will take a hostname and port number. RemoteClient's processing responsibility will be to (1) open a socket connection to a server running on the host and listening on the port passed into the constructor; (2) write the Event to this socket.

Since Events implement `java.io.Serializable`, you should send and receive Events over a `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` objects, respectively.

```
public class RemoteClient implements EventProcessor {  
    public RemoteClient (InetAddress addr, int port);  
    ....  
}
```

The server that listens for Events from a RemoteClient will be an instance of the RemoteServer class. This class also implements the EventProcessor interface.

```
public class RemoteServer extends Dispatcher {  
    RemoteServer (int port) throws java.io.IOException;  
    public void go ();  
    ....  
}
```

The constructor for RemoteServer takes as its argument a port on which to listen for TCP connections. When the constructor is called, it opens a listen socket (*aka* a server socket) on localhost at port number port. The process of opening a socket might fail with a `java.io.IOException`; the constructor should *reflect* that exception to the caller, and thus declares that it throws a `java.io.IOException`.

RemoteServer instances are single threaded, completely processing one Event at a time. This is done in the go method as follows:

1. Accept a connection from a RemoteClient using the listen socket.
2. Read the Event from the connection.
3. Close the connection.
4. Process the Event.
5. Repeat.

Note that the go method does not return. Checked exceptions thrown within the go should be *masked*. That is, catch the exception within go, print some message if you wish, and then continue processing as before.

2.2.1 Extra Credit

The `TimeStamper` class creates a new event by chaining an IR representing the time onto each event it receives. We could create a class called `Attacher` that generalizes this behavior to permit arbitrary IRs to be attached to incoming events. For extra credit, implement the `Attacher` class, having the following signature:

```
public class Attacher extends Dispatcher {
    public Attacher(IRGenerator gen);
    ...
}
```

The constructor takes a object having type `IRGenerator`, which is an interface defined as follows:

```
public interface IRGenerator {
    public IR genIR();
}
```

Now write a class `TimeStamper2` that implements the same functionality as the `TimeStamper` class above, but uses `Attacher` and an appropriate implementation of `IRGenerator` for its implementation.

3 Putting it all together

Ultimately, we'll want to build an application that makes use of the various event processors we have defined. We have provided a sample application that will test all of the `EventProcessors` you have built, called `SimpleTest`. Within a single application, it starts a `RemoteServer` in a separate thread, which adds timestamps to the events it receives and prints them out. It then creates a dispatcher that routes events to a printer and to a filter, and the filter forwards successfully filtered events to a remote client that sends them to the above-mentioned server. A picture of this can be seen in the comments for the code itself.

When you run this program, it will output something like the following every second:

```
Food:Hamburger+Timestamp:1062775969409
Food:Hamburger
Cheese:Limburger+Timestamp:1062775969416
Cheese:Limburger
Drink:Beer
Beer:Bitburger+Timestamp:1062775969423
Beer:Bitburger
```

That is, four events from the local processor, and three events from the remote processor (the `Drink:Beer` event is filtered out), which have timestamps added

to them. Note that because of nondeterminism in thread scheduling, the order the events are printed and timestamps will vary. You want to make sure that you always get events of this format, and that on average you are getting these seven events per second.

For testing purposes, so everyone uses different ports, test your code so that RemoteServers listen using port *x3yyy*, where *x* is your section number (either 1 or 2) and *yyy* is the last three digits of your account number. You can do this with SimpleTest by running at as follows:

```
java -Dport=x3yyy cmsc433.SimpleTest
```

4 Final Notes

For all of the classes that you write, you may assume that all constructor arguments will be non-null. If you wish, you can write constructors to check for null arguments, and upon finding them, throw a `NullPointerException`.

4.1 Reference Material

You may find it useful to look at Eckel's *Thinking in Java*, Chapter 12. This has information on input and output streams, serialization, and regular expressions. Chapter 11 of this same book has information on Java Collections, which you may also find useful. Finally, there is a nice tutorial on client/server programming in Java at <http://java.sun.com/docs/books/tutorial/networking/sockets/index.html>.

5 Updates to this assignment

Summary of changes:

(9/15/2003)

- The constructors for `Filter`, `Dispatcher`, etc. had spurious `void` return types; these were removed as constructors should not declare a return type.
- The `SimpleTest.java` sample file included calls to a constructor for events that took two string arguments, as in `new Event ("Food", "Hamburger");` these should have been `new Event (new IR("Food", "Hamburger"))`.
- Clarified that a `Printer` should call the given `java.io.PrintWriter`'s `println` method.

(9/13/2003)

- The description of `RemoteServer` clarifies how to handle exceptions thrown within the constructor, and within the `go` method.

(9/8/2003)

- Section 4 clarifies what to do when constructor arguments are `null`.
- The description of `Filter` clarifies the regular expression matching semantics.
- The description of `Event` specifies that the `remove` method need not be implemented (and if it is, you can assume it won't be called).

(9/5/2003)

- The definition of `Event` has changed: the method `addIR` was removed in favor of a constructor that creates a new `Event` in terms of an existing one. This allows events to be immutable, which makes understanding event processing simpler. This change resulted in a slight change to the description of `TimeStamper`.
- An extra credit problem was added.
- A sample test driver was added. This driver uses a threaded server to test your event processors. You don't need to understand the server to use it (in the case you don't currently understand threads).