Using Message Metadata

The QuickFIX/J `DataDictionary` contains information about supported FIX messages and fields. This is sometimes called message *metadata*. QuickFIX/J uses this information for parsing, formatting, and validating messages. However it has other potential uses as well. For example, you could write a simple message formatter that uses `DataDictionary`information to print human-friendly representations of FIX messages. In this article we'll develop a message formatter as an example of using QuickFIX/J metadata.

The basic idea is simple, but there are a few complications to consider. Let's say we have a parsed FIX message and want to format it in way that that's easier to read than tag/value pairs. We might start by simply iterating over the fields and printing their values.

```
public class MessagePrinter {

    public void print(PrintStream out, DataDictionary dd, Message message)
throws FieldNotFound {
        Iterator fieldIterator = message.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            String value = message.getString(field.getTag());
            out.println(dd.getFieldName(field.getTag()) + ": " + value);
        }
    }
}
```

To test our printer, we can use the following program that parses a message and the prints it.

```
public class MetadataExample {
    public static void main(String[] args) throws Exception {
        DataDictionary dd = new DataDictionary("FIX44.xml");

        Message m = new Message("8=FIX.4.4\0019=247\00135=s\00134=5\001"
                + "49=sender\00152=20060319-09:08:20.881\001"
                +
"56=target\00122=8\00140=2\00144=9\00148=ABC\00155=ABC\001"
                + "60=20060319-09:08:19\001548=184214\001549=2\001"
                + "550=0\001552=2\00154=1\001453=2\001448=8\001447=D\001"
                +
"452=4\001448=AAA35777\001447=D\001452=3\00138=9\00154=2\001"
                +
"453=2\001448=8\001447=D\001452=4\001448=aaa\001447=D\001"
                + "452=3\00138=9\00110=056\001", dd);

        MessagePrinter printer = new MessagePrinter();
        printer.print(System.out, dd, m);
    }
}
```

The output looks like...

```
SecurityIDSource: 8
OrdType: 2
Price: 9
SecurityID: ABC
Symbol: ABC
TransactTime: 20060319-09:08:19
CrossID: 184214
CrossType: 2
CrossPrioritization: 0
NoSides: 2
```

That's not a bad start, but it's missing quite a few fields. For example, there are no header or trailer fields. In QuickFIX messages, the body of the message is represented as a `FieldMap`. The header and trailer are each in their own `FieldMap`, separate from the body. We need to iterate over the fields in all three `FieldMaps` associated with the message. To do this, we'll first refactor the code by extracting the existing field iteration code into a `printFieldMap` method. Then, we'll use that method for the header, body, and trailer sections of the message.

```
public class MessagePrinter {

    public void print(PrintStream out, DataDictionary dd, Message message)
throws FieldNotFound {
          printFieldMap(out, dd, message.getHeader());
          printFieldMap(out, dd, message);
          printFieldMap(out, dd, message.getTrailer());
    }

    private void printFieldMap(PrintStream out, DataDictionary dd, FieldMap
fieldMap)
            throws FieldNotFound {
        Iterator fieldIterator = fieldMap.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            String value = fieldMap.getString(field.getTag());
            out.println(dd.getFieldName(field.getTag()) + ": " + value);
        }
    }
}
```

And the output is shown below.

```
BeginString: FIX.4.4
BodyLength: 247
MsgSeqNum: 5
MsgType: s
SenderCompID: sender
SendingTime: 20060319-09:08:20.881
TargetCompID: target
SecurityIDSource: 8
OrdType: 2
Price: 9
SecurityID: ABC
Symbol: ABC
TransactTime: 20060319-09:08:19
CrossID: 184214
CrossType: 2
CrossPrioritization: 0
NoSides: 2
CheckSum: 056
```

That's better, but we're still missing fields. Why is that? The fields are missing because we aren't processing the repeated groups within the message. To print the groups we must iterate over the group keys in the `FieldMap` and then retrieve the repeating sets of fields for each group. Before printing the repeating fields, we'll print the group count.

To retrieve the sets fields in a repeating group, we create an empty `Group` object and then call `FieldMap.getGroup` to populate the empty object with fields. This `Group` object is also a `FieldMap` so we can call our existing `printFieldMap`method recursively. A benefit of the recursive call is that repeating groups nested within repeating groups will also be printed.

```
public class MessagePrinter {

    public void print(PrintStream out, DataDictionary dd, Message message)
throws FieldNotFound {
        printFieldMap(out, dd, message.getHeader());
        printFieldMap(out, dd, message);
        printFieldMap(out, dd, message.getTrailer());
    }

    private void printFieldMap(PrintStream out, DataDictionary dd, FieldMap
fieldMap)
            throws FieldNotFound {
        Iterator fieldIterator = fieldMap.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            String value = fieldMap.getString(field.getTag());
            out.println(dd.getFieldName(field.getTag()) + ": " + value);
        }

        Iterator groupsKeys = fieldMap.groupKeyIterator();
        while (groupsKeys.hasNext()) {
            int groupCountTag = ((Integer) groupsKeys.next()).intValue();

            out.println(dd.getFieldName(groupCountTag) + ": count = "
                    + fieldMap.getInt(groupCountTag));

            Group group = new Group(groupCountTag, 0);
            int i = 1;
            while (fieldMap.hasGroup(i, groupCountTag)) {
                fieldMap.getGroup(i, group);
                printFieldMap(out, dd, group);
                i++;
            }
        }
    }
```

And the new result is...

```
BeginString: FIX.4.4
BodyLength: 247
MsgSeqNum: 5
MsgType: s
SenderCompID: sender
SendingTime: 20060319-09:08:20.881
TargetCompID: target
SecurityIDSource: 8
OrdType: 2
Price: 9
SecurityID: ABC
Symbol: ABC
TransactTime: 20060319-09:08:19
CrossID: 184214
CrossType: 2
CrossPrioritization: 0
NoSides: 2
NoSides: count = 2
OrderQty: 9
Side: 1
NoPartyIDs: 2
NoPartyIDs: count = 2
PartyIDSource: D
PartyID: 8
PartyRole: 4
PartyIDSource: D
PartyID: AAA35777
PartyRole: 3
OrderQty: 9
Side: 2
NoPartyIDs: 2
NoPartyIDs: count = 2
PartyIDSource: D
PartyID: 8
PartyRole: 4
PartyIDSource: D
PartyID: aaa
PartyRole: 3
CheckSum: 056
```

All the fields are there now, but there are still a few problems. The group count tag is shown twice because it's also a top level field. We'll add a few lines of code to avoid printing the group count field as a normal field. To do this we can ask the data dictionary for the type of a field. In this case, we are checking whether the field is a `NumInGroup` type.

```
private void printFieldMap(String prefix, DataDictionary dd, String
msgType, FieldMap fieldMap)
            throws FieldNotFound {

        Iterator fieldIterator = fieldMap.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            if (!isGroupCountField(dd, field)) {
                printField(prefix, dd, fieldMap, field);
            }
        }

        // . . .
    }

    private boolean isGroupCountField(DataDictionary dd, Field field) {
        return dd.getFieldTypeEnum(field.getTag()) == FieldType.NumInGroup;
    }
```

We'll also add some code to nicely print the field value translation for fields with enumerated values. We'll use the data dictionary to get the "value name" for fields that have enumerated values.

```
private void printFieldMap(String prefix, DataDictionary dd, String
msgType, FieldMap fieldMap)
            throws FieldNotFound {

        Iterator fieldIterator = fieldMap.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            if (!isGroupCountField(dd, field)) {
                String value = fieldMap.getString(field.getTag());
                if (dd.hasFieldValue(field.getTag())) {
                    value = dd.getValueName(field.getTag(),
                              fieldMap.getString(field.getTag())) + " ("
+ value + ")";
                }
                System.out.println(prefix + dd.getFieldName(field.getTag())
+ ": " + value);
            }
        }

        // . . .
```

This is becoming almost useful, but it could definitely be a little more user friendly. For example, it's difficult to see the grouping structure. We'll add a little code for cosmetic purposes to indent the groups and print delimeters after each repeating set of fields.

The final code looks like the following.

```
public class MessagePrinter {

    public void print(DataDictionary dd, Message message) throws
FieldNotFound {
        String msgType = message.getHeader().getString(MsgType.FIELD);
        printFieldMap("", dd, msgType, message.getHeader());
        printFieldMap("", dd, msgType, message);
        printFieldMap("", dd, msgType, message.getTrailer());
    }

    private void printFieldMap(String prefix, DataDictionary dd, String
msgType, FieldMap fieldMap)
            throws FieldNotFound {

        Iterator fieldIterator = fieldMap.iterator();
        while (fieldIterator.hasNext()) {
            Field field = (Field) fieldIterator.next();
            if (!isGroupCountField(dd, field)) {
                String value = fieldMap.getString(field.getTag());
                if (dd.hasFieldValue(field.getTag())) {
                    value = dd.getValueName(field.getTag(),
fieldMap.getString(field.getTag())) + " (" + value + ")";
                }
                System.out.println(prefix + dd.getFieldName(field.getTag())
+ ": " + value);
            }
        }

        Iterator groupsKeys = fieldMap.groupKeyIterator();
        while (groupsKeys.hasNext()) {
            int groupCountTag = ((Integer) groupsKeys.next()).intValue();
            System.out.println(prefix + dd.getFieldName(groupCountTag) + ":
count = "
                    + fieldMap.getInt(groupCountTag));
            Group g = new Group(groupCountTag, 0);
            int i = 1;
            while (fieldMap.hasGroup(i, groupCountTag)) {
                if (i > 1) {
                    System.out.println(prefix + "  ----");
                }
                fieldMap.getGroup(i, g);
                printFieldMap(prefix + "  ", dd, msgType, g);
                i++;
            }
        }
    }

    private boolean isGroupCountField(DataDictionary dd, Field field) {
        return dd.getFieldTypeEnum(field.getTag()) == FieldType.NumInGroup;
    }

}
```

And the final output looks like...

```
BeginString: FIX.4.4
BodyLength: 247
MsgSeqNum: 5
MsgType: NewOrderCross (s)
SenderCompID: sender
SendingTime: 20060319-09:08:20.881
TargetCompID: target
SecurityIDSource: EXCHANGE_SYMBOL (8)
OrdType: LIMIT (2)
Price: 9
SecurityID: ABC
Symbol: ABC
TransactTime: 20060319-09:08:19
CrossID: 184214
CrossType:
CROSS_TRADE_WHICH_IS_EXECUTED_PARTIALLY_AND_THE_REST_IS_CANCELLED (2)
CrossPrioritization: NONE (0)
NoSides: count = 2
  OrderQty: 9
  Side: BUY (1)
  NoPartyIDs: count = 2
    PartyIDSource: PROPRIETARY_CUSTOM_CODE (D)
    PartyID: 8
    PartyRole: CLEARING_FIRM (4)
    ----
    PartyIDSource: PROPRIETARY_CUSTOM_CODE (D)
    PartyID: AAA35777
    PartyRole: CLIENT_ID (3)
  ----
  OrderQty: 9
  Side: SELL (2)
  NoPartyIDs: count = 2
    PartyIDSource: PROPRIETARY_CUSTOM_CODE (D)
    PartyID: 8
    PartyRole: CLEARING_FIRM (4)
    ----
    PartyIDSource: PROPRIETARY_CUSTOM_CODE (D)
    PartyID: aaa
    PartyRole: CLIENT_ID (3)
CheckSum: 056
```

I hope this gives you some ideas about how to use the metadata API in QuickFIX/J. Please leave a comment if you have questions or suggestions.

Steve Bate

File	Modified
MessagePrinter.java	May 26, 2006 by Steve Bate
MetadataExample.java	May 26, 2006 by Steve Bate

Download All