FSDMsgX

What is it?

Its utility library to enable you to parse grammar usually used by text that are field separator delimited, fixed length, branching based on data parsed, looking ahead in the stream for a specific byte and base future parsing decisions.

Why do I need it, JPOS already has the schema based FSD?

The standard FSD one in JPOS does not support a more flexible branching operations ,lookaheads,optionalfields, out of the box PCI compliance (do not need to worry where you data gets dumped, if compliance is set it will always dump it in a compliant format, no need for a ProtectedDebugListener which helps only when you log to Q2).
This removes the jdom library dependence.

Scope

How to use the available library.

Classes

1. FSDMsgX
   Is the main object that you would create. It a container that allows you to add the various packager that have been defined above. This class itself is a packager, hence it can be added to any of the packagers.
2. FixedFieldPackager
   If you have a fixed field that needs to be parsed used this.
   e.g.

```
//public FixedFieldPackager(String name,int size,Interpreter interpretter)


FixedFieldPackager p = new FixedFieldPackager("F1", 2,
                                                AsciiInterpreter.INSTANCE);
            FSDMsgX msg = new FSDMsgX("Test");
            msg.add(p);
            String s = "12ABCDEFH";
            int offset = msg.unpack(s.getBytes());
            System.out.println(msg.get("F1"));
```

What this snippet does is it will extract 2 bytes from the text input. The constructor takes a name, size and how to interpret the data. The interpreter objects used are from the jpos library.
The sysout shows you that the field F1 will have “12”.
This just shows one parsing rule as we “add” the rule to the container msg. You can write all your rules and add them in order and the unpack call will assign values to your fields based on the rules.
Pack is the reverse process, where you set the fields and get a formatted byte array.

3. **VariableFieldPackager**

If you have a variable field that terminates with a delimiter use this.

```
VariableFieldPackager p = new VariableFieldPackager("F1", 20, new Byte((byte)
                                                0x1c), AsciiInterpreter.INSTANCE);
FSDMsgX msg = new FSDMsgX("Test");
msg.add("F1", p);
String inStream = "123456" + (char) 0x1c;
msg.unpack(inStream.getBytes());
```

Here the rule is the data can be a max of 20 and is terminated by a 0x1c (FS). At unpacking time it implements the rule and if data vailable is less than equal to 20 bytes and terminated by a FS its valid and data is assigned to the field F1 for later reuse. The delimiter FS is not unpacked into the field.
Similarly at packing time, the delimiter is added to the output stream of bytes.

4. **BranchFieldPackager**

   If you need to parse future text based on a value in a previous field use this.

```
AFSDFieldPackager f1 = new FixedFieldPackager("F1", 5,
                                                AsciiInterpreter.INSTANCE);
AFSDFieldPackager f2 = new FixedFieldPackager("F2", 2,
                                                 AsciiInterpreter.INSTANCE);
AFSDFieldPackager case01 = new FixedFieldPackager("F4", 3,
                                                   AsciiInterpreter.INSTANCE);
AFSDFieldPackager case02 = new FixedFieldPackager("F5", 4,
                                                   AsciiInterpreter.INSTANCE);

Map<String, AFSDFieldPackager> caseMap = new HashMap<String,AFSDFieldPackager>();
             caseMap.put("01", case01);
             caseMap.put("02", case02);
             AFSDFieldPackager f3 = new BranchFieldPackager("F3", "F2", caseMap,
null);

             FSDMsgX msg = new FSDMsgX("Test");
             msg.add(f1);
             msg.add(f2);
             msg.add(f3);

             msg.set("F1", "ABCDE");
             msg.set("F2", "02");
             msg.set("F4", "333");
             msg.set("F5", "4444");
byte[] outStream = msg.pack();
```

_ Here is a slightly more complex parsing rule, where we read the first 5 bytes into Field F1, read the next 2 bytes into F2.
Now based on the value in F2 the field following it will be 3 wide if the value in F2 is 01, and it will be 4 wide if the value in F2 is 02.

The BranchFieldPackagers' constructor takes a name of the field that will be used for the value test. A Map is provided with the key and will contain values of F2, the maps value will be a packager to use for that key.
In the example you will notice we have set F4 and F5 but if you check the output byte array from packing, it will contain the data 4444 as F2 is set to 02 and the value in F5 will be used.

There is a very useful **getParserTree** method on the packagers which describes your grammar.

A sample output for the above example. It can be called before or after unpacking/packing. If you run it without setting fields you will just see the gramar without data. Here we see the data.
It gets the first 5 bytes, then the next 2. Then it does a switch on the F2 and executes the packager assigned for that value. We haven't set default field packager in the constructor, typically used if the value in F2 is not mapped to a packager, it will choose the default packager.

```
[Test]
 Field [F1] : Fixed [5] : ABCDE
 Field [F2] : Fixed [2] : 02
 Field [F3] : [Branch]
      switch (F2)
           01:
                Field [F4] : Fixed [3]
           02:
                Field [F5] : Fixed [4] : 4444
           default:
                [Not Set]
```

LookAheadPackager

If you need to base parsing decision based on a byte of data found at an offset from the current parsing position use this.

```
                String s = "ABCD1234;XYZ";
                FSDMsgX ifSet = new FSDMsgX("ifSet");
                AFSDFieldPackager f0_a = new FixedFieldPackager("F0", new
        String(new byte[] { (byte) ';' }),
                                AsciiInterpreter.INSTANCE);
                AFSDFieldPackager f1_a = new FixedFieldPackager("F1", 4,
        AsciiInterpreter.INSTANCE);
                AFSDFieldPackager f2_a = new FixedFieldPackager("F2", 4,
        AsciiInterpreter.INSTANCE);

                ifSet.add("F1", f1_a);
                ifSet.add("F2", f2_a);
                ifSet.add("F0", f0_a);

                FSDMsgX ifNotSet = new FSDMsgX("ifnotSet");
                AFSDFieldPackager f1_b = new FixedFieldPackager("F1", 8,
        AsciiInterpreter.INSTANCE);
                ifNotSet.add("F1", f1_b);

                AFSDFieldPackager f3 = new FixedFieldPackager("F3", 3,
        AsciiInterpreter.INSTANCE);

                FSDMsgX main = new FSDMsgX("main");
                String[] setFields = { "F0" };
                AFSDFieldPackager look = new LookAheadPackager("LA", 8,
        Byte.valueOf((byte) ';'), ifSet, ifNotSet, setFields,
                                null);
                main.add("LA", look);
                main.add("F3", f3);
                System.out.println(main.getParserTree(""));

                main.unpack(s.getBytes());
```

Here the rule is as follows,
If the 0 based offset , offset 8 is a semi colon, there are 2 fields F1 and F2 before it and one 3 wide field after it. If semicolon is not present then there is a 8 wide field followed by a 3 wide field.
So if it finds the semi colon, use the ifset packager in the constructor else use the ifnotset paramater.

If your use the packagers **parsetree** method, this is what you will see when a semicolon is present. It tells you the various paths it can take and since it was called ater the pack, it shows you the parsed data.

```
[main]

Field [LA] : [LookAhead]
                    offset[8] find[0x3B]
                          [if found]
                                [ifSet]
                                Field [F1] : Fixed [4] : ABCD
                                Field [F2] : Fixed [4] : 1234
                                Field [F0] : Fixed [1] : ;
                          [if not found]
                                [ifnotSet]
                                Field [F1] : Fixed [8]
                    Check Field[F0,]
Field [F3] : Fixed [3] : XYZ
```

If you remove the semicolon this is what you will see. You may notice F0 has a semicolon, but thats just because its a constant field in the packager defined.

```
[main]

Field [LA] : [LookAhead]
                    offset[8] find[0x3B]
                          [if found]
                                [ifSet]
                                Field [F1] : Fixed [4]
                                Field [F2] : Fixed [4]
                                Field [F0] : Fixed [1] : ;
                          [if not found]
                                [ifnotSet]
                                Field [F1] : Fixed [8] : ABCD1234
                    Check Field[F0,]
Field [F3] : Fixed [3] : XYZ
```

OptionalPackager

This is a utility packager that is usually usefule when there are optional fields at the end of the stream of data. Basically there I mandatory data in the message, then at the end there are optional fields present.

Pci compliance

Each packager has a constructor that can take an APCICompliance object, 2 implementations are provide, TrackDataCompliance for tack data handling and WipeCompliance for masking the full field. If you set these, the dump and hexdump methods that prints a pretty xml and raw hex data will protect the

data based on the compliance object set. Use the jpos ISOUtil method for track data compliance.

e.g.

The the interpretter for the fixed field was a BCDInterpretter.

```
<fsdmsgX name="Test">

      <field id="F1" value="123456______3456"/>
</fsdmgX>

0000  F1 F2 F3 F4 F5 F6 6D 6D  6D 6D 6D 6D F3 F4 F5 F6  ......mmmmmm....
```

More complicated Example

Sample working example, the values used for setting are random. Here the Thales A0,A1,A6 and FA are implemented , though there is no gaurantee the implementation is correct as I need this to setup an example. The behavior is correct to best of my knoledge.
The Thales FA command message has a complicated structure with optional fields. If you have a Thales manual you should look it up.
See the sample output of the hexdump , FA message in xml and the parsetree to see the logic of the parser.

```
package com.ols.thales;


import java.util.HashMap;
import java.util.Map;

import org.jpos.fsdpackager.AFSDFieldPackager;
import org.jpos.fsdpackager.BranchFieldPackager;
import org.jpos.fsdpackager.FSDMsgX;
import org.jpos.fsdpackager.FixedFieldPackager;
import org.jpos.fsdpackager.LookAheadPackager;
import org.jpos.fsdpackager.OptionalPackager;
import org.jpos.fsdpackager.compliance.TrackDataCompliance;
import org.jpos.iso.AsciiInterpreter;
import org.jpos.iso.ISOException;
import org.jpos.iso.ISOUtil;

public class ThalesMessage {

	public FSDMsgX getFSDMessage() {

		FSDMsgX message = new FSDMsgX("Thales");
		FixedFieldPackager stan = new FixedFieldPackager("stan", 4,
AsciiInterpreter.INSTANCE);
		message.add(stan);

		FixedFieldPackager command = new FixedFieldPackager("command", 2,
AsciiInterpreter.INSTANCE);
		message.add(command);

		Map<String, AFSDFieldPackager> commandCases = new HashMap<String,
AFSDFieldPackager>();
		commandCases.put("A0", getA0Packager());
		commandCases.put("A1", getA1Packager());
		commandCases.put("A6", getA6Packager());
		commandCases.put("FA", getFAPackager());

		BranchFieldPackager branchCommand = new
BranchFieldPackager("commandBrancher", "command", commandCases, null);

		message.add(branchCommand);
		return message;
```

```
	}

	private static AFSDFieldPackager getFAPackager() {
		FixedFieldPackager f1 = new FixedFieldPackager("zmk-scheme", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f2 = new FixedFieldPackager("zmk", 32,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f3 = new FixedFieldPackager("import-key-scheme", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f4 = new FixedFieldPackager("key-to-import-under-zmk", 32,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f5 = new FixedFieldPackager("mode", 1,
AsciiInterpreter.INSTANCE);
		OptionalPackager f6 = new OptionalPackager("optiona-variant",
				new FixedFieldPackager("attalla-variant", 1,
AsciiInterpreter.INSTANCE));

		FSDMsgX optionalGrp1 = new FSDMsgX("OptionalGroup1");
		AFSDFieldPackager semicolon = new
FixedFieldPackager("delimiter",";",AsciiInterpreter.INSTANCE);
		AFSDFieldPackager reserved1 = new FixedFieldPackager("Reserved-
1","O",AsciiInterpreter.INSTANCE);
		AFSDFieldPackager keySchemeLMK = new FixedFieldPackager("key-scheme-
lmk",1,AsciiInterpreter.INSTANCE);
		AFSDFieldPackager keyCheckValue = new FixedFieldPackager("key-check-value-
len",1,AsciiInterpreter.INSTANCE);
		optionalGrp1.add(semicolon);
		optionalGrp1.add(reserved1);
		optionalGrp1.add(keySchemeLMK);
		optionalGrp1.add(keyCheckValue);
		LookAheadPackager determineGrp1 = new LookAheadPackager("LAGRP1", 0, new
Byte((byte)';') ,optionalGrp1, null, new String[]{"key-scheme-lmk","key-check-value-len"},
null);

		FSDMsgX optionalGrp2 = new FSDMsgX("OptionalGroup2");

		AFSDFieldPackager lmkid = new FixedFieldPackager("LMK-
ID",2,AsciiInterpreter.INSTANCE);
		AFSDFieldPackager percent = new FixedFieldPackager("delimiter-
2","%",AsciiInterpreter.INSTANCE);
		optionalGrp2.add(percent);
		optionalGrp2.add(lmkid);
		LookAheadPackager determineGrp2 = new LookAheadPackager("LAGRP2", 0, new
Byte((byte)'%') ,optionalGrp2, null, new String[]{"LMK-ID"}, null);


		FSDMsgX container = new FSDMsgX("TranslateZPKReq-FA");
		container.add(f1);
		container.add(f2);
		container.add(f3);
		container.add(f4);
		container.add(f5);
		container.add(f6);
		container.add(determineGrp1);
		container.add(determineGrp2);

		return container;
```

```
	}

	private static FSDMsgX getA0Packager() {

		FixedFieldPackager f1 = new FixedFieldPackager("mode", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f2 = new FixedFieldPackager("key-type", 3,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f3 = new FixedFieldPackager("key-scheme-key-under-lmk", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f4 = new FixedFieldPackager("scheme-zmk-or-tmk", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f5 = new FixedFieldPackager("key-zmk-or-tmk", 32,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f6 = new FixedFieldPackager("key-scheme-key-under-zmk-or-
tmk", 1, AsciiInterpreter.INSTANCE);

		FSDMsgX container = new FSDMsgX("GenerateKeyReq-A0");
		container.add(f1);
		container.add(f2);
		container.add(f3);
		container.add(f4);
		container.add(f5);
		container.add(f6);
		return container;

	}

	private static AFSDFieldPackager getA6Packager() {
		FixedFieldPackager f1 = new FixedFieldPackager("key-type", 3,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f2 = new FixedFieldPackager("zmk-scheme", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f3 = new FixedFieldPackager("zmk", 32,
AsciiInterpreter.INSTANCE, new TrackDataCompliance());
		FixedFieldPackager f4 = new FixedFieldPackager("import-key-scheme", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f5 = new FixedFieldPackager("key-to-import-under-zmk", 32,
AsciiInterpreter.INSTANCE,
				new TrackDataCompliance());
		FixedFieldPackager f6 = new FixedFieldPackager("scheme-key-encrpt-under-lmk",
1, AsciiInterpreter.INSTANCE);
		OptionalPackager f7 = new OptionalPackager("optiona-variant",
				new FixedFieldPackager("attalla-variant", 1,
AsciiInterpreter.INSTANCE));

		FSDMsgX container = new FSDMsgX("ImportKeyReq-A6");
		container.add(f1);
		container.add(f2);
		container.add(f3);
		container.add(f4);
		container.add(f5);
		container.add(f6);
		container.add(f7);
		return container;

	}
```

```
	private static AFSDFieldPackager getA1Packager() {

		FixedFieldPackager f0 = new FixedFieldPackager("error", 2,
AsciiInterpreter.INSTANCE);

		FixedFieldPackager f1 = new FixedFieldPackager("scheme-key-under-lmk", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f2 = new FixedFieldPackager("key-under-lmk", 32,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f3 = new FixedFieldPackager("scheme-key-under-zmk", 1,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f4 = new FixedFieldPackager("key-under-zmk", 32,
AsciiInterpreter.INSTANCE);
		FixedFieldPackager f5 = new FixedFieldPackager("check-value", 6,
AsciiInterpreter.INSTANCE);

		FSDMsgX container = new FSDMsgX("GenerateKeyRsp-A1");
		container.add(f0);
		container.add(f1);
		container.add(f2);
		container.add(f3);
		container.add(f4);
		container.add(f5);
		return container;
	}

	public static void main(String[] args) throws ISOException {
		ThalesMessage m = new ThalesMessage();


		FSDMsgX fsd2 = m.getFSDMessage();
		fsd2.set("stan", "1234");
		fsd2.set("command", "FA");
		fsd2.set("zmk-scheme", "A");
		fsd2.set("zmk", ISOUtil.padleft("", 32, 'A'));
		fsd2.set("import-key-scheme", "B");
		fsd2.set("key-to-import-under-zmk",ISOUtil.padleft("", 32, 'A'));
		fsd2.set("mode", "1");
//you can comment out optional atalla-variant+ grp1 + grp2, and run the test. Comment out
grp1 and  grp2 and run the test. Lastly comment out grp2 and run the test.
		fsd2.set("attalla-variant", "V");

		//optional group fields , you can comment out grp1 and grp2,
		fsd2.set("key-scheme-lmk", "U");
		fsd2.set("key-check-value-len", "0");

		//optional grp2 field
		fsd2.set("LMK-ID", "00");



		byte[] outStream2 = fsd2.pack();
		System.out.println(ISOUtil.hexdump(outStream2));
		System.out.println(fsd2.dump(""));
		System.out.println(fsd2.getParserTree(""));


//		FSDMsgX mess = m.getFSDMessage();
```

```
//            mess.unpack(outStream2);
//            System.out.println(mess.dump(""));
//            System.out.println(mess.getParserTree(""));


      }

}
```

The output looks like

```
0000  31 32 33 34 46 41 41 41  41 41 41 41 41 41 41 41  1234FAAAAAAAAAAA

0010  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0020  41 41 41 41 41 41 41 42  41 41 41 41 41 41 41 41  AAAAAAABAAAAAAAA
0030  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0040  41 41 41 41 41 41 41 41  31 56 3B 4F 55 30 25 30  AAAAAAAA1V;OU0%0
0050  30                                                0

<fsdmsgX name="Thales">
       <field id="stan" value="1234"/>
       <field id="command" value="FA"/>
       <fsdmsgX name="TranslateZPKReq-FA">
              <field id="zmk-scheme" value="A"/>
              <field id="zmk" value="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"/>
              <field id="import-key-scheme" value="B"/>
              <field id="key-to-import-under-zmk"
value="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"/>
              <field id="mode" value="1"/>
              <field id="attalla-variant" value="V"/>
              <fsdmsgX name="OptionalGroup1">
                     <field id="delimiter" value=";"/>
                     <field id="Reserved-1" value="O"/>
                     <field id="key-scheme-lmk" value="U"/>
                     <field id="key-check-value-len" value="0"/>
              </fsdmgX>
              <fsdmsgX name="OptionalGroup2">
                     <field id="delimiter-2" value="%"/>
                     <field id="LMK-ID" value="00"/>
              </fsdmgX>
       </fsdmgX>
</fsdmgX>

[Thales]
Field [stan] : Fixed [4] : 1234
Field [command] : Fixed [2] : FA
Field [commandBrancher] : [Branch]
       switch (command)
              A1:
                     [GenerateKeyRsp-A1]
                     Field [error] : Fixed [2]
                     Field [scheme-key-under-lmk] : Fixed [1]
```

```
                Field [key-under-lmk] : Fixed [32]
                Field [scheme-key-under-zmk] : Fixed [1]
                Field [key-under-zmk] : Fixed [32]
                Field [check-value] : Fixed [6]
        A6:
                [ImportKeyReq-A6]
                Field [key-type] : Fixed [3]
                Field [zmk-scheme] : Fixed [1]
                Field [zmk] : Fixed [32]
                Field [import-key-scheme] : Fixed [1]
                Field [key-to-import-under-zmk] : Fixed [32]
                Field [scheme-key-encrpt-under-lmk] : Fixed [1]
                Field [optiona-variant] : [OPTIONAL]
                      Field [attalla-variant] : Fixed [1]
        FA:
                [TranslateZPKReq-FA]
                Field [zmk-scheme] : Fixed [1] : A
                Field [zmk] : Fixed [32] : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                Field [import-key-scheme] : Fixed [1] : B
                Field [key-to-import-under-zmk] : Fixed [32] :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                Field [mode] : Fixed [1] : 1
                Field [optiona-variant] : [OPTIONAL]
                      Field [attalla-variant] : Fixed [1] : V
                Field [LAGRP1] : [LookAhead]
                offset[0] find[0x3B]
                      [if found]
                            [OptionalGroup1]
                            Field [delimiter] : Fixed [1] : ;
                            Field [Reserved-1] : Fixed [1] : O
                            Field [key-scheme-lmk] : Fixed [1] : U
                            Field [key-check-value-len] : Fixed [1] : 0
                      [if not found]
                            [Not Set]
                Check Field[key-scheme-lmk,key-check-value-len,]
                Field [LAGRP2] : [LookAhead]
                offset[0] find[0x25]
                      [if found]
                            [OptionalGroup2]
                            Field [delimiter-2] : Fixed [1] : %
                            Field [LMK-ID] : Fixed [2] : 00
                      [if not found]
                            [Not Set]
                Check Field[LMK-ID,]
        A0:
                [GenerateKeyReq-A0]
                Field [mode] : Fixed [1]
                Field [key-type] : Fixed [3]
                Field [key-scheme-key-under-lmk] : Fixed [1]
                Field [scheme-zmk-or-tmk] : Fixed [1]
                Field [key-zmk-or-tmk] : Fixed [32]
                Field [key-scheme-key-under-zmk-or-tmk] : Fixed [1]
        default:
                [Not Set]
```

Limitations:

All field names have to be unique.

Can be time consuming to wire the objects.