Writing MetaModel Constraints in Java Streams

Work in groups of 2 in this assignment. So find a long-term partner.

A fundamental tenet of future software design is the elevation of programming beyond standard languages like C, C++, C#, and Java. In Model Driven Engineering, the **Object Constraint Language (OCL)** is used to define and evaluate metamodel constraints. There are other languages besides OCL. In this course, we use Java Streams to express constraints. Java streams exhibit a flavor similar to OCL, but I trust the Java language designers more than I do the OCL designers.

In this assignment, you will write metamodel constraints as Java Streams. You will need to read a bit about Java Streams to become familar with it.

Preface

1. Read about Java Streams

I gave a short tutorial in class on Java Streams and provided some in-class examples for you to try. But you still need to brush-up on your knowledge. Listed below are some tutorials that you can consult, (try #1 first). Frankly, I don't think any of them are particularly good.

1. https://www.tutorialspoint.com/java8/java8_streams.htm
2. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
3. http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

2 Install MDELite7 Software

MDELite7 is a fifth-generation of experimental tools for writing MDE applications. Install MDELite7. (You should have done this in the previous programming assignment). Follow its installation instructions and read its (short) main document.

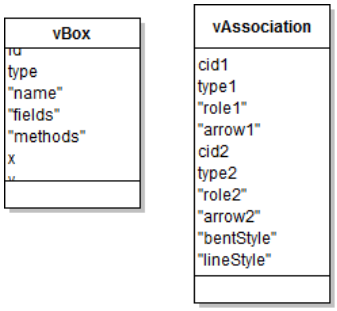
Violet

USE THE VERSION OF VIOLET THAT COMES WITH MDELite7 -- the version that is downloadable from the violet web site is not usable for this assignment. Violet is a Java-based tool that allows you to draw a number of different UML diagrams, including class diagrams, state diagrams, and so on. You can run Violet from MDELite7 by the invocation:

```
> java MDL.Violet
```

Like many drawing tools, Violet is not very smart. Look at the figure below. Violet allows you to customize relationships between pairs of classes by drawing lines between boxes. Once a line is drawn, you can say 'put a diamond at this end of the line', 'put an inheritance arrow at this other end', and so on, yielding a diagram that is UML-like, but nonsensical. The figure below has several errors: (1) inheritance links never have diamonds on them, (2) inheritance links don't have labels, and (3) inheritance links between classes are not dashed and are always solid.

Violet allows you to draw any nonsensical set of beautiful boxes and links, and to output them as an XML documents with a **.violet** extension. If you click on the figure above, you will see the XML file that is produced. I ask you -- who in God's name can read XML?? Continuing, you will also see that Violet is not very good at defining methods and variables -- it only provides a big fat text string in which you can scribble almost anything. If you click on the figure below, you will see the MDELite Schema for Violet class diagrams and the tuples that define the above class diagram. Notice that the content of the XML file is minimal in database form.



MetaModel Constraints

In the first part of this assignment, you will write Java Stream expressions to evaluate **vpl** database constraints. Fortunately, a violet-to-database translator has been written for you. It is available in the MDELite7 toolset and can be invoked by:

```
> java Violett.ClassParser X.class.violet X.vpl.pl
```

and yes, **Violett** with 2 t's is correct (ask me why in class), where **X** is the name that you gave to the violet file (its full name is X.class.violet). **Violett.ClassParser** is hardcoded to output tuples that conform to the **vpl** schema (vpl.schema.pl).

> **Note**: what this means is that if the schema is changed, then **VioletClassParser** must be manually changed. There is no reason for you to change **vpl.schema.pl** or **Violett.ClassParser** in this assignment.

The naming scheme is important: "**X.vpl.pl**" -- **X** is as before, **vpl** is the name of database schema, and "**pl**" simply designates a fact file. To test conformance, you will write the following Java program:

```
> java VPL.Conform X.vpl.pl
```

which evaluates the database constraints that you apply to validate all **vpl** databases. The way you get started is:

- understand the **vpl** schema
- look at how **Violet** and **Violett.ClassParser** encodes tuples in a **vpl** database
- draw some beautiful, simple, and nonsensical Violet Class diagrams to realize what constraints to write and how to write these constraints as Java Streams.

So again, you will (a) translate each of your **.class.violet** files to a **vpl** database using the provided MDELite7 tool and (b) write a set of English constraints for **vpl** databases, (c) translate these constraints into Java Stream expressions (this is your **VPL.Conform** program), and (d) check whether the **.violet** diagram conforms to your metamodel constraints. The next two sections provide background that you need to understand before you can do anything.

VPL Schemas and Databases

Before continuing, you should understand the set of tables that are generated. The **vpl** schema consists of 2 tables, namely **vBox** and **vAssociation**. In class, I explained the basic syntax of database schemas and their tuples. Here's the schema:

```
% vpl class database

dbase(vpl,[vBox,vAssociation]).

% type = c(class),i(nterface),n(ote). packages are skipped
table(vBox,[id,type,"name","fields","methods",x,y]).

% type1,type2 = c(lass) or i(nterface
% lineStyle = ""(solid) or "DOTTED"
% arrow1,2 = V, TRIANGLE, DIAMOND, BLACK_DIAMOND
% bentStyle = "", HV, VH, HVH, VHV
table(vAssociation,[id,cid1,type1,"role1","arrow1",cid2,type2,"role2","arrow2","bentStyle","lineStyle","middleLabel"]).
```

Here's what the above means:

- A vpl database consists of two tables, **vBox** and **vAssociation**.
- Table **vBox** has 7 attributes:
  - **id** -- tuple identifier
  - **type** -- is 'c' for class, 'i' for interface, and 'n' for note
  - **name** -- name of the box given in the diagram
  - **fields** -- a Java string of multiline field declarations given in the diagram, newlines are indicated by character '%'
  - **methods** - a Java string of multiline method declarations given in the diagram, newlines are indicated by character '%'
  - **x,y** -- the X,Y position the box has in the violet diagram
- Table **vAssociation** has 12 attributes
  - **id** -- tuple identifier (always distinct from the identifiers of vBox tuples)
  - **cid1** -- the vBox identifier of one end of an association
  - **type1** -- 'c' for class or 'i' for interface
  - **role1** -- a Java string indicating the name of the role of cid1
    I assume the format of a role can be any of "rolename" OR "cardinality" OR "cardinality rolename", where cardinality is any one of the strings "1", "0..1", "n", "m", "0..*", "1..*", "*". This is all you need for this assignment.
  - **arrow1** -- V for pointer, TRIANGLE for inheritance, DIAMOND for 0..1, BLACK_DIAMOND for exactly 1, or "" (meaning NONE)
  - **cid2, type2, role2, arrow2** -- for the other side of the association
  - **bentStyle** -- the legal bent styles, "" (for straight), HV, VH, HVH, and VHV
  - **lineStyle** -- the legal line styles: "" (for SOLID) or DOTTED
  - **middlelabel** -- The translation of violet to vpl assumes that all middle labels of associations are empty. For each non-empty middle label, a tuple appears in this table, along with the cids of the classes/interfaces that are connected. Basically this table should never have rows. If it has tuples, flag an error!

Your Java program, **VPL.Conform**, evaluates metamodel constraints using Java Streams that will check the correctness of **.violet** models of class diagrams. There are all sorts of constraints that you could write on checking the validity of fields and methods. But because Violet simply mashes field and method declarations into a big fat text string, this makes it almost impossible to do fancy checking. So to keep things simple -- **and you should thank me for this** -- do NOT write checks about method and field declarations! **Limit yourself to checking the following constraints:**

- **Middle Labels Constraint** -- no middle labels on associations is permitted.
- **All vBox Identifiers are unique**
- **All vAssociation identifiers are unique**
- **No illegal references: each cid1 and cid2 field must reference a legal vBox identifier**
- **Unique Name Constraint** -- classes and interfaces must have unique names, no class can have the same name of an interface, and vice versa.
- **Null Names Constraint** -- classes and interfaces cannot have null names.
- **Triangle Constraint** -- inheritance is a line with one arrow ending in a TRIANGLE. Its other arrow must be NONE.
- **No Labels in Inheritance Constraint** -- inheritance cannot have role labels at either end.

- **Solid Association Constraint** -- non-implements, non-extends associations must be SOLID.
- **Extends Constraint** -- class-extends relationships must be SOLID.
- **Implements Constraint1** -- class implements relationships must be DOTTED.
- **Implements Constraint2** -- only classes can implement interfaces.
- **All interfaces have no fields** -- the field attribute is empty ("")
- **All notes should have no methods and no fields** -- the field and method attributes are empty ("")

Please check MDELite html documentation for constraints. There is a class, **PrologDB.Constraints**, that has a set of methods that can be used to express each of the above constraints in one method call.

In the process of developing **`VPL.Conform`**, do the following:

- Create a violet class diagram X using **MDL.Violet** and translate it into **X.vpl.pl** using **MDL.VioletClassParser** and then evaluate it using **`vpl.Conform`**, as shown in the following sequence of program invocations:

`> java MDL.Violet`	`create class diagram X.class.violet`
`> java Violett.ClassParser X.class.violet X.vpl.pl`	`transform X.class.violet to X.vpl.pl`
`> java VPL.Conform X.vpl.pl`	`see if X.vpl.pl conforms to constraints`

The following zip file contains a set of 8 Violet designs that you are to validate. For each file, present the list of error messages (if any) that your tool produces. I presume that you will test your work on other Violet designs. You should include them in your documentation too.

Start with this "shell" of a NetBeans Project.

What to Submit to Canvas

All of the below in a zip file (including your Netbeans or Eclipse Project). The zip file must unzip into **<yourName>/<YourFilesAndDirectories>**.

1. Your program needs to run correctly on Linux machines, even though you may have developed them on Macs and Windoze. The TA will grade your program running on Linux.
2. Your Java **`vplConform`** source.
3. An easy to read english description of your program -- document anything that you have added beyond what was given in the shell.
4. For each Violet example, the output of your conformance test and maybe the violet diagram for that example.
5. A PDF file (in the required format) that the Grader should read to provide any information that is not obvious. The contents of the PDF file can be minimal.

You should expect other violet files, of the Grader's choosing, will be used to evaluate your rules/constraints.