HW1 – Strategizing About Person Matching

Estimated time: 8-12 hours

Objectives

- Become familiar with using UML class diagrams to describe not trivial object structures.
- Become familiar with the Strategy pattern
- Become familiar with basic unit testing techniques

Overview

In this assignment, you will look for opportunities to apply the strategy pattern in the construction of a program that reads *Person* objects from an input file, computes which pairs of objects represent the same real person, and then summarizes the results.

If you program satisfies the requirements but you do not apply the strategy pattern, you will receive a failing grade. Also, you will probably end up taking several times longer to do this assignment if you do not make good use of the strategy pattern. This assignment is not about getting a piece of code to "work"; rather, it is about applying principles, patterns, and best practices to come up with a good design that is easy to implement and test.

Instructions

To complete this assignment, you do the following:

1. Clone the course's shared Git repository, locate the hw1 folder, and examine its contents.
2. Study the strategy pattern.
3. Study the functional and non-functional requirements listed below.
4. Design a program that satisfies the following requirements by applying the strategy design pattern effectively. Capture the structure of that design in one or more UML Class Diagrams.
5. Implement that program. Note: Update you design diagram if your design changes.
6. Test your program using the provided test files, and well as any others you decide to create.
7. Write up a short report that includes your UML Class Diagrams and insights uncovered during the design, implementation, or testing. Save your report as PDF file called HW1-Report.pdf.

Requirements

For the purposes of expressing the requirements, the requirements will refer to the program you must build as *PersonMatcher*.

Functional Requirements

1. Command-line Interface Features

1.1. A user should be able to run *PersonMatcher* from the command line with 2 or 3 parameters.

1.1.1. The first parameter identifies a matching algorithm (see Req. Def. 3). It can be a number or name that uniquely identifies the algorithm.

1.1.2. The second parameter is the name of an input file. Initially, PersonMatcher must accept either JSON or XML files. The JSON file will have a .json and the XML files will have .xml extension.

1.1.3. The third parameter is an optional output file name

Examples:

```
PersonMatcher 1 JSON_PersonTestSet_3.json Results.txt
PersonMatcher 2 XML_PersonTestSet_4.xml Results.txt
```

1.2. When a user run *PersonMatcher* with invalid parameters, it must detect and report this fact, along with a help message about correct usage.

2. When a user runs *PersonMatcher* with valid parameters, it must do the following:

2.1. Load the input file input into a set of *Person* objects, based on whether the input file is a JSON or XML file. Table 1 list all the properties for *Person* object. Sample data files are available in the Data subdirectory of the shared Git repository.

2.2. Compare every unique pair of Person objects, using the specific matching algorithm (Req. Def 3) and then save every matching pair in a set of matching pairs.

2.3. Summarize the result, i.e., the matching-pair set, using one of the following methods:

2.3.1. If no output file is specified on the command line, then *PersonMatcher* should print the matching pairs to the console as illustrated below:

```
Match:
    Id=15, Name=Charles Frank Chatterton, BirthDate = 10/11/1981
    Id=1015, Name=Charles Frank Chatterton, BirthDate = 10/11/1981
Match:
    Id=17, Name=Edward Steven Oscarson, BirthDate = 7/23/1965
    Id=23, Name=Edward Steven Oscarson, BirthDate = 7/23/1965
Match:
    Id=18, Name=James Nathan Newton, BirthDate = 12/2/2000,
    Id=24, Name=James Nathan Newton, BirthDate = 12/2/2000
```

2.3.2. If an output file is specified on the command line, then *PersonMatcher* should list the two object ids of each matching pair to a line of the output file, e.g.

```
15,1015
```

```
17,23
18,24
20,32
20,1332
24,26
30,1230
32,1332
```

3. A matching algorithm is a Boolean function that compares a subset properties for *Person* objects in a *MatchPair*, with aim of correctly algorithm identify those that represent the same person.
   3.1. A *MatchPair* is an object that contains two different Person objects.
   3.2. *PersonMatch* must support at least three different matching algorithms. Below are ideas for several possible algorithms, but feel free to implement for own algorithms.

        1 – Match the person objects based primarily on the person's names
        2 – Match the person objects based primarily on mother and birth information
        3 – Match the person objects based primarily on identifiers

   3.3. A match algorithm says that a *MatchPair* is a true match, then that Match Pair should be saved so it can be included in the summary.
4. Person objects
   4.1.1. Any property of a *Person* object, except *Object Id*, may contain a null or bad data. A null for any property value means that the property is not known for the person. A "" or "0" are valid values and are different than a null. For example, if a person is believed not to have middle name, the *MiddleName* property will be a "", not a null. A null would mean that is no information on the middle name.
   4.1.2. The JSON and XML tags for a field corresponds to the property name. See sample data.

Non-Functional Requirements

1. *PersonMatch's* design must be extensible for new types of input files.
2. *PersonMatch's* design must be extensible for new match algorithms.
3. *PersonMatch's* design must be extensible for ways of summarizing the results (true *MatchPairs*).
4. *PersonMatch's* implementation must follow the design, be well organized, and readable.

Table 1 – Person Properties

Property Name	Data Type	Description
ObjectId	Integer	An unique identifier for the object, not the person
StateFileNumber	String	A state file number -- highly unique, but only available to people born in the Utah and some bad values may exit
SocialSecurityNumber	String	A federal tax identifier – highly unique, but

		is only occasionally available and prone to error
FirstName	String	The person’s first name – newborns may have empty or bad first names
MiddleName	String	The person’s middle name – often null
LastName	String	The person’s last name
BirthYear	Integer	The year of the person’s birth date
BirthMonth	Integer	The month of the person’s birth date
BirthDay	Integer	The day of the person’s birth date
Gender	String	The person’s gender – “F” or “M”, blank, or bad data
NewbornScreeningNumber	String	A number assigned to babies born in Utah – highly unique, but only available for children born in Utah medical facilities
IsPartOfMultipleBirth	String	The person was born with a twin, triplet, etc. – “T”, “F”, or blank
BirthOrder	Integer	If the person was part of a multiple birth, then this is person’s birth order relative to the others. For example, The first of a set of twins born would have a value of 1 and the second would have a value of 2
BirthCounty	String	The county in which the person was born
MotherFirstName	String	The person’s mother’s first name
MotherMiddleName	String	The person’s mother’s middle name
MotherLastName	String	The person’s mother’s last name
Phone1	String	A phone number for the person
Phone2	String	A phone number for the person

Submission Instructions

Zip up your entire solution, including test cases and sample input files, in a zip archive file called CS5700_hw1_<*fullname*>.zip, where fullname is your first and last names. Then, submit the zip file to the Canvas system.

Grading Criteria

Criteria	Max Points
A clear and concise structural model using UML Class Diagrams in the HW1-Report.pdf file.	20
Good, clear use of the Strategy pattern, evident in the design and code.	20
The design and implementation meet all functional and non-functional requirements	10
Implementation follows the principles classification, encapsulation, localization of design decisions, abstraction, and inheritance	20

The implementation is readable and follows good practices	10
The implementation matches the design	10
Meaning insights documented in HW1-Report.pdf	10