Software Engineering Project

Part 2 - Domain Analysis

Use the description of the ISWOSS software system you received in Part 1 of this project.

In Part 1 you have completed the ***Requirement Analysis.***

The objective of this part is to start performing the ***Domain Analysis*** of the Ice Skating Winter Olympics Scoring System. Use the description of the ISWOSS software system you received in Part 1.

The Domain Analysis is a large phase which can be divided in the following parts:

1) Defining Classes
    - Identify the major type of domain objects and define them
2) Defining Relationships
    - Describe the major associations between those objects
3) Defining Operations
    - Identify the major operations required to support class structure and system functions
4) Defining Attributes and Inheritance
    - Determine the properties that describe the classes
5) Performing Validation and Iteration.
    - Review, test and repair the model you are creating during the process.

The deliverable of the Domain Analysis include:

- ***Class diagrams*** which identify the key classes, or types of the domain
- ***Class specifications***, which contain all the semantics definitions of the classes, their relationships, their attributes, and their key operations
- ***Object-scenario diagrams***, which will illustrate how the objects will interact to carry out key system functions (i.e. a trace of the system in execution).
- ***Data Dictionary*** which lists all the domain entities including classes, relationships and attributes.

In this phase of the project, you will focus your attention to the points 1) and 2) of the above list i.e.

1) Defining classes
2) Defining Relationships

Step 5) must be and will be performed continuously. Use the description of the ISWOSS software system you received in Part 1.

I. Defining Classes (50%)

Here are the steps you have to perform.

1) ***Identify the candidate*** classes of the ISWOSS.
   - Stay at the logical level and remain within the scope of the system**.**

A few words of warning:

- The problem statement often includes implementation characteristics: you are interested only in logical classes at this point.
- The problem statement may include contextual information irrelevant to the system's responsibilities, and you may not care about those classes.
- Natural language is inherently ambiguous, there may be aliases or one term may apply to different things.
- A concept can be described using either a noun or a verb phrase (i.e. *gardner* or *someone gardens).*
- The nouns can be classes (i.e. team) but may also be objects (i.e. figure skating event), relationship, or attributes of the class.

So…

Discovering the Key Classes

- Examine tangible things and the role they play in the system;
- Outline the steps necessary to complete the use case listed in the system function statement produced in Part I and identify the objects that participate in the scenario;
- Identify the responsibility of each class, the knowledge the class maintains and the action it provides. List the classes that collaborate with it to support these responsibilities.
- When you identified a *candidate* class for your system, choose meaningful names for it.
- Remain within the system's scope. Concentrate only on the classes that are needed to carry out the responsibilities listed in the system charter.

2) While you identify the ***Key Abstractions*** of the ISWOSS (i.e. the classes now, and later, the relationships and the attributes) ***start building a Data Dictionary***. [Note: The word ***Key***, here means "major"; i.e. those items that reveal the most about the domain].
   - The ***Data Dictionary*** is the central repository for the abstractions that are relevant to the domain and lists all the domain entities including classes, relationships and attributes.
   - Choose a meaningful name for each abstraction that you identify and insert in the data dictionary.

- The Data Dictionary grows and, if it becomes too large, it may be necessary to use a simple database to manage the information.
- As the analysis proceeds, some abstractions in the data dictionary will turn out to be classes, some relationships, and other simply attributes.

3) When you identify a class, in StarUML or Visio create a class icon and label it with its name. **Create a class diagram:**

   - Place a class icon for each class, labeled with its name.
   - At the moment the classes in the diagram will be standing alone since you have not identified the relationships between classes yet.

4) Define the Class Specifications as indicated in the example below. Insert the specifications in Visio or StarUML where possible.

   Example: *Sample Class Specifications:*

Class name: Session; *Documentation*: *Definition*: A session is an event of a conference in which two or more presenters will give a talk. *Constraints*: A session must be part of a conference. There cannot be more than two sessions of the same type at a time. *Class name*: Club; *Documentation*: *Definition*: A club is a collection of people dedicated to a particular interest or activity. *Constraints*: A session must be part of a conference. There cannot be more than two sessions of the same type at a time.

In conclusion, start by discovering the Key Classes, then filter your list, add the discovered classes to the Data Dictionary, and finally create a Class diagrams containing the identified classes.

PROGRESS SO FAR: After completing this part you have:

- Discovered the major domain abstractions of the system
- Named the abstractions carefully

- Noted any rule or constraints about each abstraction
- Built the Data Dictionary by describing all the abstractions discovered so far.

DELIVERABLES – **Release #2a)**

- A class diagram containing all the identified classes (no relationships yet).
- A class specification for each class.
- A data dictionary that lists all the entities that have been discovered so far.

II. Defining Relationships (50%)

Classes do not exist in isolation but they are related in a variety of ways that form the class structure of the system.

Relationships help further define the classes by exposing their content or dependency on the content of others.

a) Identify the relationships

There are 3 key kinds of class relationships: association, aggregation, and inheritance.

Association denotes some semantic dependency among classes; ***Aggregation*** denotes "part of" relationship.

An association is a bidirectional relationship. Here is an example of association.

Example of Association
Consider a Scoring System of Competition Events. Among the others, the class Judge, and the class Competition_Event have been identified as key abstraction of this system. In such a system "A Judge scores a Competition_Event and a Competition_Event is scored by a Judge". Therefore the classes "Judge" and "Event" have a link of association (a relationship) since the objects of the class Judge must be aware of which event they judge, and the objects of the class Event must be aware of which judges they are judged by.

An aggregation represents a whole/part relationship and occurs when an object is physically constructed from other objects (for example an engine contains a cylinder), or when an object logically contains another object (for example a shareholder owns a share).

Example of Aggregation
- an Engine contains a Cylinder (physically)
- a Shareholder owns a Share (logically)

Finally, inheritance is used to express generalization/specialization relationships.

Example of Inheritance

- a Triangle is a Figure (Figure is a generalization of a Triangle)

b) Name the relationships

Name the relationships by giving them concise meaningful names. A relationship name often represents the role of the target class to the source. It is a concise name that provides significant semantic information.

c) Add the cardinality

The cardinality is used to indicate the quantity. To determine the cardinality, think always of any instance of the source class and then define whether or not this instance participates in the relationship. If it must, the lower bound will be 1, otherwise the lower bound will be 0. The maximum number of instances of the target at any given time gives you the upper bound of the cardinality. For example, in the example of association, a Judge can score multiple Competion_Events. In that case a cardinality ***1 .. n*** should be used to decorate the relationship.

d) Annotate the class diagram

Fully annotate the initial class diagram with the relationships you have discovered.

e) Polish your work

Spend time on issues such relationship names, class names.

PROGRESS SO FAR: After completing this part you have:

- Discovered the major relationships between key abstractions.
- Found additional abstractions.
- Defined all the relationships, including all the cardinality.

DELIVERABLES – **Release #2b)**

- A class diagram fully annotated with the relationships and the new classes discovered in the meanwhile. (50%)

III. Create class templates for prototype

Since we are working in Agile, (while you should not think about implementation yet!), in this last step you need to catch some time by preparing for the prototype by preparing the proper files and by learning how to use Doxigen to create the automatic documentation.
Observe that you have identified the classes and their relationships but **you have NOT found out the operations of each class yet**. Nevertheless you can start preparing the .h and .cpp files (if you are working in C++) for each class. The file will contain for now just the name of the class as well as the proper specifications you have identified in this part of the project. The proper specification should be written in form of comments.

In order to easily generate automatic documentation of the prototype that you will be building in the next part of the project, use from now on Doxygen comments in your code.

Doxigen is a documentation system for C++, C, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP, and C#. Both a version for Linux and Windows is available.
Doxygen is already installed on neptune.cs.kent.edu and on poseidon.cs.kent.edu. If you are planning to work under another operating system you must install your version (you can download Doxigen here.)

To use Doxigen on neptune or Poseidon connect to the machine and move into the folder that contains all your classes you created and type the command

```
$ doxygen -g
```

A file called *Doxygen* will be generated. Then type the command

```
$ doxygen Doxygen
```

The command generates documentation both in html and in latex.

You will submit the html folder you will generate together with the other deliverables of this project.

An example of the Doxygen style of tags has been used in the file *Thing.h* and *Thing.cpp* attached to this project. More information can be found in the doxygen manual as well as online if necessary.

While you do not have to return anything for this part yet, you will start learning Doxigen comments that will be required in the next part of the project.

- Prepare all the deliverables as indicated in part I and II.

  The data produced in this part of the project must be inserted in the appropriate part of the report template provided in part 1 and must be printed and collected in the 3 binder folder. Each printed page must contain the date and the name of the author in the Header of the page. Pages must NOT be numerated. Separate main sections in the binder with a separator.

Deadline: 11:59 pm, March 13, 2018

Important: The releases must zipped in a folder and dropped in the appropriate dropbox under Learn by the deadline. Since delay in the project cost lot of money to the company, **heavy late penalties** WILL BE APPLIED if this Release is late. The penalties will apply to the whole group.

G**rading scale:**

- (20%) A class diagram that contains ONLY (no relationships) classes.
- (20%) A class specification for each class.
- (10%) A data dictionary that lists all the entities that have been discovered so far.
- (50%) A class diagram fully annotated with the relationships and the new classes discovered in the meanwhile.