CIS 3309

COMPONENT-BASED SOFTWARE DESIGN

BAPTISM BY FIRE – AN INTRODUCTION TO COURSE ESSENTIALS

SUGGESTIONS ON THE DESIGN OF YOUR GAME OF NIM PROJECT

FALL SEMESTER 2017 (VER 1.0 JULY 4, 2017)

Work Requirements:

You are to design and then code a simulator for the game of Nim. You will first want to play the game yourselves to understand exactly how it works and what the rules are. Further information may be found on the internet, or you can check out this document (which also came from the Internet). Your simulation should work for just one play of the game; you need not handle replays or count the number of wins for each player. Note that we are not asking you to master the strategies (or the mathematics behind the strategies) for playing the game. This is not necessary for you to simulate the game, unless you want to give the players hints as to what they should do next (NOT for this project, please).

Introduction:

This is a somewhat easier project compared to those that follow. But it does carry with it several challenges not unlike those you will experience with the next three projects. Many of these challenges are listed below. The bolded items in this list represent terminology with which we need to become familiar over the next couple of weeks.

1. You are learning a new language, **C# .NET**.
2. You will gain some mastery in the use of an extensive **Framework Class Library**, or FCL, an integral part of the Visual Studio software.
3. You are learning the ropes of a new **Integrated Development Environment** (IDE).
4. You will learn how to design a **form**.
5. You will gain some degree of mastery in the practice of **OOD** and **OOP**, most of the material in the first 6 chapters of your book, and the first two sections of Chapter 12 on designing and implementing with classes.
6. You will begin to acquire some knowledge of **MVC (Model-View-Controller)** software design paradigm.
7. You will learn how to write **event driven programs**, something you probably have not yet done.
8. You will begin to gain some mastery over the **art and science of debugging** (the first two sections of Chapter 11). This will require using a **scientific method** approach, where you will form hypotheses as to what is causing program error, and then experiment with your code to test out your hypotheses.

PLEASE NOTE: There are many good approaches to this project. At best, what is presented here might just be one of them. COME TO CLASS WITH QUESTIONS!! Whatever you do -- there should be NO PROGRAMMING in the first week. This is a design time and without a good design, most of you will not get this project done. For clarity's sake: IF YOU DO NOT USE THE CLASSES DESCRIBED BELOW, OR SOMETHING VERY SIMILAR, YOUR CREDIT FOR THIS PROJECT WILL BE ZERO.

We UNDERSTAND YOU ARE STARTING COLD TURKEY. And so, THIS WILL BE HARD, but if we work together we can do this. However … you have to be a full participant, asking and helping to answer questions. We will help you; even give you some code. BUT – ***you have to ask questions!!***

Project Goals:

The goals for this project are summarized next. They are directly related to the challenges listed earlier.

- Learning the basics of object-oriented design and programming
- Understanding how to analyze a problem statement (a specification) and
  - Map out the behavior of a software system that meets the needs specified system using a **UML**-like behavior diagram
    http://www.ibm.com/developerworks/rational/library/769.html
    http://www.smartdraw.com/resources/tutorials/uml-diagrams
  - Provide a detailed description of the classes you will use as the basic building blocks of your software system, including descriptions of
    a) the responsibility or purpose of each class,
    b) the attributes (or data stores) required for each class,
    c) the methods required for a class, including the name and purpose of each method, the return values (for functions), and the arguments needed, and
    d) the exceptions that must be handled by the class in order to keep your system from crashing (Crashing systems cost money -- in your case -- points)
- Practicing incremental design and implementation, starting with the core part of your project and expanding outward, adding and testing new features in an incremental and patient fashion.
- Learning to program in C# .NET
- Learning a lot of new terminology
- Learning how to design windows forms that meet the needs of and make sense to a user
- Learning the fundamentals of more complex algorithm design as well as event-driven programming
- Learning to navigate the Visual Studio IDE
- Learning about software testing. There is a lot of software testing to be done. Although this is not a primary focus of the course, it deserves considerable attention. You will want to test out each version of your program, as it is developed
- Learning to work effectively in groups
- Learning the basic ideas behind MVC
- Learning how to find errors in your code and use the Visual Studio debugger to assist in this task

These goals are very similar to those of the next three projects. The intent here is to get you started learning how to carry out the design and implementation tasks listed above using the new tools also described above.

BORING, But IMPORTANT: More so than other first projects I have used, the Game of Nim Project requires a significant amount of attention to detail, especially as related to the controls on your form. We need to work hard to protect the players from doing things out of turn, or making mistakes that will leave our simulation in an unusable state. Such protections from "web-site" errors and abuse are not unusual in windows or web site development. This game simply provides more challenges than the other introductory projects we have used in the past.

Project Fundamentals:

Three aspects of this project should be developed in parallel, and by now, in great detail.

A. A detailed map of the behavior of your ATM system (a **Behavior Diagram** – also called an **Activity Diagram)**) is needed.
B. As the Behavior Diagram is being developed, we also need to construct a layout of the forms required for the game (it is best to stick with one form for this game). What you will see by the time you are done, is that the form object (an instance of the Forms class of the .NET FCL) you design and the ***code behind the form*** will serve as the driver for your game of Nim. (This time, we will do the form FIRST.)
C. A model (**class**) is needed for every entity that requires processing in the development of the behavior diagram. Each **attribute** (or data store; the book calls them fields) of the class should be described, along with each **method** (and its return values and arguments). The **exceptions** to be handled should also be documented. I have provided some hints as to how I would like to see this done.

If not for your own benefit, just for mine (I expect to read what you have done), your documentation should be clearly organized with the attributes, properties, methods, and exceptions described in separate sections for each class. **IT SHOULD ALSO BE NEAT and UNCLUTTERED**! The method descriptions, return values, and arguments should also be described in separate sections for each method. You will need two classes for this project: a Player class, and an Internal Board class. These two classes will define two user-defined data types. You will build your game (implement your driver) "on-top" of these new data types, using the methods provided to get the required work done. Your game will also use a number of data types in the Visual Studio Framework Class Library (FCL), mostly (for now) data types (classes) modeling the controls you place on your form.

Building a data model (designing a class) in programming is very much like building an entity model (a table) in a relational database. It is all about modeling and ensuring that the required data stores are included and that all the methods (operations) that your system needs to perform on these data stores are included.

The most difficult job you have facing you is 1) to figure out the responsibilities of these classes (a summary description of what they are supposed to model) and then 2) the details of how to build the model.

In designing these classes there are two important principles to keep in mind.

1. Your classes should maximize cohesiveness, that is software elements that logically belong together should be placed together in a class. (Put another way, you should put in a class only those data stores and methods that are functionally related – related to the data stores protected by the class.)
2. Your classes should be minimally or loosely coupled to each other. That is, the connections among classes should normally be limited to passing messages back and forth through the methods of the classes. Data in one class should almost never be transmitted to another except through the methods in each class.

These concepts have been around since the 1960s (which proves there is nothing new under the sun – except that there is a tremendous amount of new technology new under the sun – and there is more every fraction of every second). Of course, figuring out what this all means takes practice and more practice, as well as an examination of case studies through examples, which we will do as often as we can, starting now.

Once we have completed our design tasks, your classes will provide us with a set of user-defined data types, similar in almost every way to the built-in data types, such as *int* and *float* provided by C#.NET . These classes, those in the FCL, and the built-in types, will form the foundation of your project. You will build your game (implement your project driver) "on-top" of these new data types, using the methods provided to get the required work done. We sometimes say that you will program against the methods in these classes. Your game will also use a number of data types in the Visual Studio Framework Class Library (FCL), mostly data types (classes) modeling the controls you place on your form.

Building a data model (designing a class) in programming is very much like building an entity model (a table) in a relational database. It is all about modeling (the Modeling part of MVC), and ensuring that the required data stores are included and that all the methods (operations) that your system needs to perform on these stores are included. Each of your models might look something like this:

Class x – statement of responsibility or purpose

Members

Attributes – list of data stores (simple variables and structures) and descriptions of each

Methods –

Method1 – description (purpose or responsibility)

Return Value – (description)

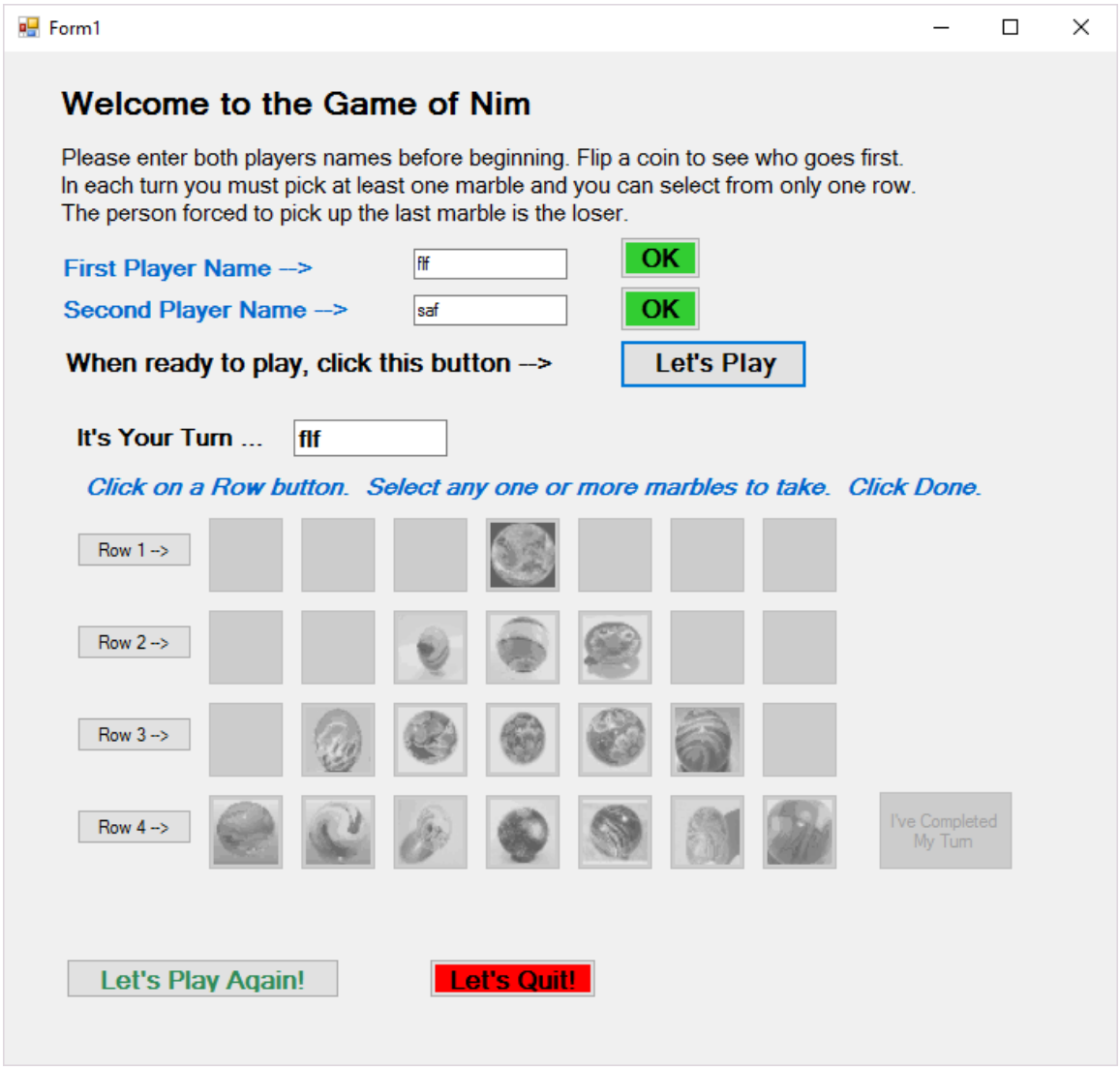
Arguments – (list and description of each)

Method2 – etc.

Exceptions to be Handled – (list and description of each exception you handled – if any)

We are going to ask you to design your project so that the code behind the form serves as the "Controller" for the game, calling the methods in your classes (the Player class, the Internal Board class, and FCL classes) to get the work done. In other words, the Controller queries and modifies the state of your internal data models and modifies the windows form representation of your user-interface model.

In designing your form, be sure we can see the name and token for each Player and be sure we can see a two-dimensional model of the Nim Board. As needed, we should also be able to see information indicating whose turn it is, whether or not we have a winner (and who it is), and describing what has taken place with each Player move. and You can use a ***message box*** to do this or you can use a label and textbox that become visible as needed (try one of each) . A sample form is shown below. Your first task will be to figure out what each control represents and hen name every control on the form.



What kind of forms controls will you use to represent each of the 4x7 items serving the user's view of the board? Once you have gone through the pain of "statically" dragging the 28 (or perhaps just 16, once you understand the game) controls onto your form, we can talk about "dynamically" drawing these controls and compare the pros and cons of using one technique or the other.

We will work on some of this together in class. The work has to start TODAY.

A few additional notes on MVC (from Wikipedia) --

In addition to dividing the application into three kinds of components, the MVC design defines the interactions between them.

- A **controller** can send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document). It can also send commands to the model to update the model's state (e.g., editing a document). For our purposes, these commands are transmitted via function calls in C#. An example of such a controller is your Form class.
- A **model** notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. We will use two models: one for a player, and one for the internal representation of the board. The latter will look a good bit different from the user view of the board. We will explain why as we move along in designing our game.
- A **view** requests from the model the information that it needs to generate an output representation to the user.