Loading and Saving Spreadsheet Data
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW9" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Tag the version that you would want us to correct with the assignment number. For example, "HW9". **IMPORTANT REMINDER: If a HW does not have a tag by the due date, it will NOT be graded and it will receive automatically 0. If you think you will be late with a HW do the following:**

  1) send an email to all TAs and cc the instructor BEFORE THE DUE DATE and tell us that you will be late
  2) send us another email when you finish and tag the assignment so that we can grade

Assignment Instructions:

Read each step's instructions *carefully* before you write any code.

In this assignment, you will add loading and saving capabilities to your spreadsheet application. If you created a Workbook class for the previous assignment (or would like to create one for this assignment) then add Load and Save methods to it. Otherwise you can add these methods to your Spreadsheet class.

Design an XML format for your spreadsheet data. At a high level, it will probably have a structure somewhat like:

```
<spreadsheet>
 <cell name="B1">
  <bgcolor>FF8000FF</bgcolor>
  <text>=A1+6</text>
 </cell>
</spreadsheet>
```

You'll obviously have more than one cell in most cases. Make sure you do the following:

- Provide saving and loading functions that take a stream as the parameter.
- Add menu options in the UI for saving and loading.
- Make sure the saving and loading code is in the logic engine.
- Use existing XML classes from the .NET framework.

- When saving, only write data from cells that have one or more non-default properties. This means that if a cell hasn't been changed in any way then you don't need to write data for it to the file.
- Clear all spreadsheet data before loading file data. The load-from-file action is NOT a merge with existing content.
- Clear the undo/redo stacks after loading a file.
- Make sure formulas are properly evaluated after loading.
- You may assume only valid XML files will be loaded, but make sure loading is resilient to XML that has different ordering from what your saving code produces as well as extra tags. As a simple example, if you're always writing the <bgcolor> tag first for each cell followed by the <text> tag, then your loader must still support files that have these two in the opposite order. Also, if you didn't write more than these two tags within the <cell> content, your loader should just ignore extra tags when loading. See the example below.
- Use XML reading/writing capabilities from .NET. Do not write your own XML parsers that do things manually down at the string level. We discussed several options in class, such as XDocument, XmlDocument, XmlReader, and XmlWriter.

If you saved:

```
<spreadsheet>
 <cell name="B1">
  <bgcolor>FF8000</bgcolor>
  <text>=A1+6</text>
 </cell>
</spreadsheet>
```

then you must be able to load:

```
<spreadsheet>
 <cell unusedattr="abc" name="B1"> <text>=A1+6</text>
  <some_tag_you_didnt_write>blah</some_tag_you_didnt_write>
  <bgcolor>FF8000</bgcolor> <another_unused_tag>data</another_unused_tag>
 </cell>
</spreadsheet>
```

Point breakdown (the assignment is worth 10 points):

- 5 points for implementing the correct functionality

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should

concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.

- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include "TDD" in all commit messages with tests that are written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of	● Every method, attribute, type, and test case has a

Comments	comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*