```csharp
// Written by Camille Rasmussen
// UID: u0717763
// CS 3500 Fall 2014

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using SpreadsheetUtilities;
using System.Text.RegularExpressions;
using System.Xml;

namespace SS
{
    /// <summary>
    /// An AbstractSpreadsheet object represents the state of a simple spreadsheet.  A
    /// spreadsheet consists of an infinite number of named cells.
    ///
    /// A string is a cell name if and only if it consists of one or more letters,
    /// followed by one or more digits AND it satisfies the predicate IsValid.
    /// For example, "A15", "a15", "XY032", and "BC7" are cell names so long as they
    /// satisfy IsValid.  On the other hand, "Z", "X_", and "hello" are not cell names,
    /// regardless of IsValid.
    ///
    /// Any valid incoming cell name, whether passed as a parameter or embedded in a formula,
    /// must be normalized with the Normalize method before it is used by or saved in
    /// this spreadsheet.  For example, if Normalize is s => s.ToUpper(), then
    /// the Formula "x3+a5" should be converted to "X3+A5" before use.
    ///
    /// A spreadsheet contains a cell corresponding to every possible cell name.
    /// In addition to a name, each cell has a contents and a value.  The distinction is
    /// important.
    ///
    /// The contents of a cell can be (1) a string, (2) a double, or (3) a Formula.  If the
    /// contents is an empty string, we say that the cell is empty.  (By analogy, the contents
    /// of a cell in Excel is what is displayed on the editing line when the cell is selected.)
    ///
    /// In a new spreadsheet, the contents of every cell is the empty string.
    ///
    /// The value of a cell can be (1) a string, (2) a double, or (3) a FormulaError.
    /// (By analogy, the value of an Excel cell is what is displayed in that cell's position
    /// in the grid.)
    ///
    /// If a cell's contents is a string, its value is that string.
    ///
    /// If a cell's contents is a double, its value is that double.
    ///
    /// If a cell's contents is a Formula, its value is either a double or a FormulaError,
    /// as reported by the Evaluate method of the Formula class.  The value of a Formula,
    /// of course, can depend on the values of variables.  The value of a variable is the
    /// value of the spreadsheet cell it names (if that cell's value is a double) or
    /// is undefined (otherwise).
    ///
    /// Spreadsheets are never allowed to contain a combination of Formulas that establish
    /// a circular dependency.  A circular dependency exists when a cell depends on itself.
    /// For example, suppose that A1 contains B1*2, B1 contains C1*2, and C1 contains A1*2.
    /// A1 depends on B1, which depends on C1, which depends on A1.  That's a circular
    /// dependency.
    /// </summary>
    85 references
    public class Spreadsheet : AbstractSpreadsheet
    {
        // This list holds the current referencedCells (there may be empty cells here, but
        // they are ignored when this is iterated through)
        private List<Cell> referencedCells;
        // This DependencyGraph holds all the dependency relationships of the cells.
        private DependencyGraph cellDependencyGraph;

        // ADDED FOR PS5
        /// <summary>
        /// True if this spreadsheet has been modified since it was created or saved
        /// (whichever happened most recently); false otherwise.
        /// </summary>
        31 references | 0/9 passing
        public override bool Changed { get; protected set; }

        /// <summary>
        /// The non-parameterized constructor -
        /// Creates an empty spreadsheet that imposes no extra validity conditions,
        /// normalizes every cell name to itself, and has the version "default".
        /// This constructor also creates the empty
        /// list of referenced cells and the empty cell dependency graph.
        /// </summary>
        61 references | 0/59 passing
        public Spreadsheet()
            : base(x => true, x => x, "default")
        {
            referencedCells = new List<Cell>();
            cellDependencyGraph = new DependencyGraph();
            Changed = false;
        }

        /// <summary>
        /// The 3-parameterized constructor -
        /// Constructs a spreadsheet by recording its variable validity test,
        /// its normalization method, and its version information.  The variable validity
        /// test is used throughout to determine whether a string that consists of one or
        /// more letters followed by one or more digits is a valid cell name.  The variable
        /// equality test should be used thoughout to determine whether two variables are
        /// equal.
        /// This constructor also creates the empty
        /// list of referenced cells and the empty cell dependency graph.
        /// </summary>
        /// <param name="isValid"></param>
        /// <param name="normalize"></param>
        /// <param name="version"></param>
        11 references | 0/11 passing
        public Spreadsheet(Func<string, bool> isValid, Func<string, string> normalize, string version)
            : base(isValid, normalize, version)
        {
            referencedCells = new List<Cell>();
            cellDependencyGraph = new DependencyGraph();
            Changed = false;
        }

        /// <summary>
        /// The 4-parameterized constructor -
        /// This constructor has the same parameters as the 3-parameter, with
        /// the addition a string specifying a path to a file. A saved spreadsheet
        /// is read from a file (see the Save method) and is used to construct a
        /// new spreadsheet. The new spreadsheet uses the provided validity delegate,
        /// normalization delegate, and version. This constructor also creates the
        /// empty list of referenced cells and the empty cell dependency graph.
        /// </summary>
        /// <param name="filePath"></param>
        /// <param name="isValid"></param>
        /// <param name="normalize"></param>
        /// <param name="version"></param>
        9 references | 0/8 passing
        public Spreadsheet(string filePath, Func<string, bool> isValid, Func<string, string> normalize, string version)
            : base(isValid, normalize, version)
        {
            referencedCells = new List<Cell>();
            cellDependencyGraph = new DependencyGraph();

            string newName = "";

            try
            {
                using (XmlReader reader = XmlReader.Create(filePath))
                {
                    try
                    {
                        while (reader.Read())
                        {
                            if (reader.IsStartElement())
                            {
                                switch (reader.Name)
                                {
                                    case "spreadsheet":
                                        if (this.Version != reader["version"])
                                            throw new SpreadsheetReadWriteException("Mismatched version info!");
                                        break;
                                    case "cell":
                                        break;
                                    case "name":
                                        reader.Read();
                                        newName = reader.Value;
                                        break;
                                    case "contents":
                                        reader.Read();
                                        this.SetContentsOfCell(newName, reader.Value);
                                        break;
                                }
                            }
                        }
                    }

                    catch (Exception e)
                    {
                        throw new SpreadsheetReadWriteException(e.Message);
                    }
                    finally
                    { reader.Close(); }
                }
            }
            catch(Exception e)
            {
                throw new SpreadsheetReadWriteException(e.Message);
            }
            Changed = false;
        }

        // ADDED FOR PS5
        /// <summary>
        /// Returns the version information of the spreadsheet saved in the named file.
        /// If there are any problems opening, reading, or closing the file, the method
        /// should throw a SpreadsheetReadWriteException with an explanatory message.
        /// </summary>
        /// <param name="filename"></param>
        /// <returns></returns>
        6 references | 0/4 passing
        public override string GetSavedVersion(string filename)
        {
            try
            {
                using (XmlReader reader = XmlReader.Create(filename))
                {
                    try
                    {
                        while (reader.Read())
                        {
                            if (reader.IsStartElement())
                            {
                                switch (reader.Name)
                                {
                                    case "spreadsheet":
                                        return reader["version"];
                                    default:
                                        throw new SpreadsheetReadWriteException("Version information was not located correctly.");
                                }
                            }
                            throw new SpreadsheetReadWriteException("Version information not found.");
                        }
                        throw new SpreadsheetReadWriteException("No data found in file.");
                    }

                    catch (Exception e)
                    {
                        throw new SpreadsheetReadWriteException(e.Message);
                    }
                    finally
                    { reader.Close(); }
                }
            }
            catch(System.IO.FileNotFoundException e)
            {
                throw new SpreadsheetReadWriteException(e.Message);
            }
        }

        // ADDED FOR PS5
        /// <summary>
        /// Writes the contents of this spreadsheet to the named file using an XML format.
        /// The XML elements should be structured as follows:
        ///
        /// <spreadsheet version="version information goes here">
        ///
        /// <cell>
        /// <name>
        /// cell name goes here
        /// </name>
        /// <contents>
        /// cell contents goes here
        /// </contents>
        /// </cell>
        ///
        /// </spreadsheet>
        ///
        /// There should be one cell element for each non-empty cell in the spreadsheet.
        /// If the cell contains a string, it should be written as the contents.
        /// If the cell contains a double d, d.ToString() should be written as the contents.
        /// If the cell contains a Formula f, f.ToString() with "=" prepended should be written as the contents.
        ///
        /// If there are any problems opening, writing, or closing the file, the method should throw a
        /// SpreadsheetReadWriteException with an explanatory message.
        /// </summary>
        /// <param name="filename"></param>
        13 references | 0/9 passing
        public override void Save(string filename)
        {
            try
            {
                using (XmlWriter writer = XmlWriter.Create(filename))
                {
                    try
                    {
                        string content = "";

                        writer.WriteStartDocument();
                        writer.WriteStartElement("spreadsheet");

                        writer.WriteAttributeString("version", this.Version);

                        foreach (Cell cell in referencedCells)
                        {
                            if (cell.GetContents() != "")
                            {
                                writer.WriteStartElement("cell");

                                writer.WriteElementString("name", cell.GetName());

                                if (cell.GetContents() is double || cell.GetContents() is string)
                                    content = cell.GetContents().ToString();
                                if (cell.GetContents() is Formula)
                                    content = "=" + cell.GetContents().ToString();

                                writer.WriteElementString("contents", content);

                                writer.WriteEndElement();
                            }
                        }

                        writer.WriteEndElement();
                        writer.WriteEndDocument();
                    }
                    catch (Exception e)
                    {

                        throw new SpreadsheetReadWriteException(e.Message);
                    }
                    finally { writer.Close(); }
                }
            }
            catch(Exception e)
            {
                throw new SpreadsheetReadWriteException(e.Message);
            }
            Changed = false;
        }

        // ADDED FOR PS5
        /// <summary>
        /// If name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, returns the value (as opposed to the contents) of the named cell.  The return
        /// value should be either a string, a double, or a SpreadsheetUtilities.FormulaError.
        /// </summary>
        /// <param name="name"></param>
        /// <returns></returns>
        35 references | 0/19 passing
        public override object GetCellValue(string name)
        {
            if (name == null)
                throw new InvalidNameException();
            name = Normalize(name);
            if (!IsValid(name) || !ValidateVariableName(name))
                throw new InvalidNameException();

            foreach (Cell cell in referencedCells)
            {
                if (name == cell.GetName())
                {
                    return cell.GetValue();
                }
            }
            // cell has not been referenced yet
            return "";
        }

        /// <summary>
        /// Enumerates the names of all the non-empty cells in the spreadsheet.
        /// </summary>
        17 references | 0/5 passing
        public override IEnumerable<String> GetNamesOfAllNonemptyCells()
        {
            foreach (Cell cell in referencedCells)
            {
                // Do not return empty cells that have already been referenced
                if (!cell.GetContents().Equals(""))
                    yield return cell.GetName();
            }
        }

        /// <summary>
        /// If name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, returns the contents (as opposed to the value) of the named cell.  The return
        /// value should be either a string, a double, or a Formula.
        /// </summary>
        21 references | 0/11 passing
        public override object GetCellContents(string name)
        {
            name = this.Normalize(name);

            if (name == null)
                throw new InvalidNameException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();

            foreach (Cell cell in referencedCells)
                if (name == cell.GetName())
                    return cell.GetContents();
            return "";
        }

        // ADDED FOR PS5
        /// <summary>
        /// If content is null, throws an ArgumentNullException.
        ///
        /// Otherwise, if name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, if content parses as a double, the contents of the named
        /// cell becomes that double.
        ///
        /// Otherwise, if content begins with the character '=', an attempt is made
        /// to parse the remainder of content into a Formula f using the Formula
        /// constructor.  There are then three possibilities:
        ///
        ///   (1) If the remainder of content cannot be parsed into a Formula, a
        ///       SpreadsheetUtilities.FormulaFormatException is thrown.
        ///
        ///   (2) Otherwise, if changing the contents of the named cell to be f
        ///       would cause a circular dependency, a CircularException is thrown.
        ///
        ///   (3) Otherwise, the contents of the named cell becomes f.
        ///
        /// Otherwise, the contents of the named cell becomes content.
        ///
        /// If an exception is not thrown, the method returns a set consisting of
        /// name plus the names of all other cells whose value depends, directly
        /// or indirectly, on the named cell.
        ///
        /// For example, if name is A1, B1 contains A1*2, and C1 contains B1+A1, the
        /// set {A1, B1, C1} is returned.
        /// </summary>
        96 references | 0/42 passing
        public override ISet<string> SetContentsOfCell(string name, string content)
        {
            name = this.Normalize(name);

            if (content == null)
                throw new ArgumentNullException();
            if (name == null)
                throw new InvalidNameException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();
            if (!IsValid(name))
                throw new InvalidNameException();
            try
            {
                Double.Parse(content);
                // gotten past the parser, this is a double and will be added
                Changed = true;
                ISet<string> recalculators = SetCellContents(name, Double.Parse(content));
                RecalculateCells(recalculators);
                return recalculators;
            }
            catch (FormatException e)
            {
                if (content.Equals(""))
                {
                    Changed = true;
                    ISet<string> recalculators = SetCellContents(name, content);
                    RecalculateCells(recalculators);
                    return recalculators;
                }

                if (content[0].Equals('='))
                {
                    // this is for sure a formula, an attempt is made
                    Changed = true;
                    // make sure to send formula without = sign
                    ISet<string> recalculators = SetCellContents(name, new Formula(content.Remove(0, 1), this.Normalize, this.IsValid));
                    RecalculateCells(recalculators);
                    return recalculators;
                }
                else
                {
                    // this is just a plain string
                    Changed = true;
                    ISet<string> recalculators = SetCellContents(name, content);
                    RecalculateCells(recalculators);
                    return recalculators;
                }
            }
        }

        /// <summary>
        /// If name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, the contents of the named cell becomes number.  The method returns a
        /// set consisting of name plus the names of all other cells whose value depends,
        /// directly or indirectly, on the named cell.
        ///
        /// For example, if name is A1, B1 contains A1*2, and C1 contains B1+A1, the
        /// set {A1, B1, C1} is returned.
        /// </summary>
        2 references
        protected override ISet<string> SetCellContents(string name, double number)
        {
            bool referenced = false;
            Cell newReferenced;
            ISet<String> cellsToRecalculate = new HashSet<String>();

            if (name == null)
                throw new InvalidNameException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();

            // this method will throw a CircularException and no change made to spreadsheet
            foreach (String cellName in GetCellsToRecalculate(name))
                cellsToRecalculate.Add(cellName);

            // has this cell already been referenced?
            foreach (Cell cell in referencedCells)
            {
                if (name == cell.GetName())
                {
                    // modify contents
                    cell.SetDoubleCellContents(number);
                    referenced = true;
                }
            }
            // hasn't been referenced, add it to list of referenced
            if (!referenced)
            {
                newReferenced = new Cell(name, number);
                referencedCells.Add(newReferenced);
            }

            // return dependents - cells that depend on it
            return cellsToRecalculate;
        }

        /// <summary>
        /// If text is null, throws an ArgumentNullException.
        ///
        /// Otherwise, if name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, the contents of the named cell becomes text.  The method returns a
        /// set consisting of name plus the names of all other cells whose value depends,
        /// directly or indirectly, on the named cell.
        ///
        /// For example, if name is A1, B1 contains A1*2, and C1 contains B1+A1, the
        /// set {A1, B1, C1} is returned.
        /// </summary>
        4 references | 0/1 passing
        protected override ISet<string> SetCellContents(string name, string text)
        {
            bool referenced = false;
            Cell newReferenced;
            ISet<String> cellsToRecalculate = new HashSet<String>();

            if (name == null || text == null)
                throw new InvalidNameException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();

            // this method will throw a CircularException and no change made to spreadsheet
            foreach (String cellName in GetCellsToRecalculate(name))
                cellsToRecalculate.Add(cellName);

            // has this cell already been referenced?
            foreach (Cell cell in referencedCells)
            {
                if (name == cell.GetName())
                {
                    // modify contents
                    cell.SetTextCellContents(text);
                    referenced = true;
                }
            }
            // hasn't been referenced, add it to list of referenced
            if (!referenced)
            {
                newReferenced = new Cell(name, text);
                referencedCells.Add(newReferenced);
            }

            return cellsToRecalculate;
        }

        /// <summary>
        /// If the formula parameter is null, throws an ArgumentNullException.
        ///
        /// Otherwise, if name is null or invalid, throws an InvalidNameException.
        ///
        /// Otherwise, if changing the contents of the named cell to be the formula would cause a
        /// circular dependency, throws a CircularException.  (No change is made to the spreadsheet.)
        ///
        /// Otherwise, the contents of the named cell becomes formula.  The method returns a
        /// Set consisting of name plus the names of all other cells whose value depends,
        /// directly or indirectly, on the named cell.
        ///
        /// For example, if name is A1, B1 contains A1*2, and C1 contains B1+A1, the
        /// set {A1, B1, C1} is returned.
        /// </summary>
        2 references
        protected override ISet<string> SetCellContents(string name, Formula formula)
        {
            bool referenced = false;
            Cell newReferenced;
            ISet<String> cellsToRecalculate = new HashSet<String>();


            if (object.Equals(formula, null))
                throw new ArgumentNullException();
            if (name == null)
                throw new InvalidNameException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();

            // add dependents to see if it will form a CircularException
            foreach (string formula1 in formula.GetVariables())
                cellDependencyGraph.AddDependency(formula1, name);

            try
            {
                // this method will throw a CircularException and no change made to spreadsheet
                foreach (String cellName in GetCellsToRecalculate(name))
                    cellsToRecalculate.Add(cellName);
            }
            catch (CircularException)
            {
                // delete dependents!! This cell is not being modified!!
                foreach (string formula1 in formula.GetVariables())
                    cellDependencyGraph.RemoveDependency(formula1, name);
                // rethrow the exception to notify program
                throw new CircularException();
            }

            // has this cell already been referenced?
            foreach (Cell cell in referencedCells)
            {
                if (name == cell.GetName())
                {
                    // modify contents
                    cell.SetFormulaCellContents(formula, LookUpCellValue);
                    referenced = true;
                }
            }
            // hasn't been referenced, add it to list of referenced
            if (!referenced)
            {
                newReferenced = new Cell(name, formula, LookUpCellValue);
                referencedCells.Add(newReferenced);

            }

            return cellsToRecalculate;
        }

        /// <summary>
        /// If name is null, throws an ArgumentNullException.
        ///
        /// Otherwise, if name isn't a valid cell name, throws an InvalidNameException.
        ///
        /// Otherwise, returns an enumeration, without duplicates, of the names of all cells whose
        /// values depend directly on the value of the named cell.  In other words, returns
        /// an enumeration, without duplicates, of the names of all cells that contain
        /// formulas containing name.
        ///
        /// For example, suppose that
        /// A1 contains 3
        /// B1 contains the formula A1 * A1
        /// C1 contains the formula B1 + A1
        /// D1 contains the formula B1 - C1
        /// The direct dependents of A1 are B1 and C1
        /// </summary>
        15 references | 0/5 passing
        protected override IEnumerable<string> GetDirectDependents(string name)
        {
            if (name == null)
                throw new ArgumentNullException();
            if (!ValidateVariableName(name))
                throw new InvalidNameException();

            return cellDependencyGraph.GetDependents(name);
        }

        /// <summary>
        /// THE LOOKUP FUNCTION FOR DELEGATE
        /// Given a variable symbol as its parameter, lookup returns the variable's value
        /// (if it has one) or throws an ArgumentException (otherwise).
        /// </summary>
        /// <param name="cellName"></param>
        /// <returns></returns>
        3 references
        private double LookUpCellValue(string cellName)
        {
            foreach (Cell cell in referencedCells)
            {
                if (cell.GetName() == cellName)
                {
                    try
                    {
                        return double.Parse(cell.GetValue().ToString());
                    }
                    // value is not a double...
                    catch(FormatException e)
                    {
                        // not a double
                        // remember, these are caught and cause Formula Error to be returned by Evaluate
                        throw new ArgumentException();
                    }
                }
            }
            // contains empty string (not in list)
            throw new ArgumentException();
        }

        /// <summary>
        /// Another helper method. Validates the variable's syntax (consist of one or more letters
        /// followed by one or more digits)
        /// If it is invalid, false is returned
        /// </summary>
        /// <param name="name"></param>
        /// <returns></returns>
        7 references
        private bool ValidateVariableName(string name)
        {
            Regex pattern = new Regex("^[A-Za-z]+[0-9]+$");

            if (pattern.IsMatch(name))
                return true;
            else
                return false;
        }

        /// <summary>
        /// Goes through the ISet returned from SetCellContents and recalculates
        /// each cell.
        /// </summary>
        /// <param name="cellsToRecalculate"></param>
        4 references
        private void RecalculateCells(ISet<string> cellsToRecalculate)
        {
            // go through each cell that needs to be recalculated
            for (int j = 0; j < cellsToRecalculate.Count; j++)
            {
                // find it in the list of referencedCells
                for (int i = 0; i < referencedCells.Count; i++)
                {
                    // find the cell
                    if (referencedCells[i].GetName() == cellsToRecalculate.ElementAt(j))
                    {
                        // make sure its contents is a Formula (has to be, otherwise it wouldn't be a dependent)
                        if (referencedCells[i].GetContents() is Formula)
                        {
                            // Recalculate this cell's value by passing in a new Formula in order to invoke
                            // the Evaluate method again.
                            referencedCells[i].RecalculateFormulaCell(
                                new Formula(referencedCells[i].GetContents().ToString(), Normalize, IsValid), LookUpCellValue);
                        }
                    }
                }
            }
        }
```