EQUILIBRIUM SIMULATOR 2015 MAINTENANCE GUIDE

JAMES CURRAN

CONTENTS

1. System Overview 2
   1. Introduction 2
   2. Menu Hierarchy 3
   3. User-Defined Classes 3
      1. Inheritance Overview 3
      2. In-Depth Class Descriptions 5
2. Software Analysis 22
3. Algorithm Design 23
4. Full Program Listings 37

1: System Overview

1a) Introduction

Equilibrium Simulator 2015 was designed to serve as a teaching aid and revision tool, to help students understand the concept of reversible chemical reactions and the effects of changes to the conditions of a reaction. The program’s major features are:

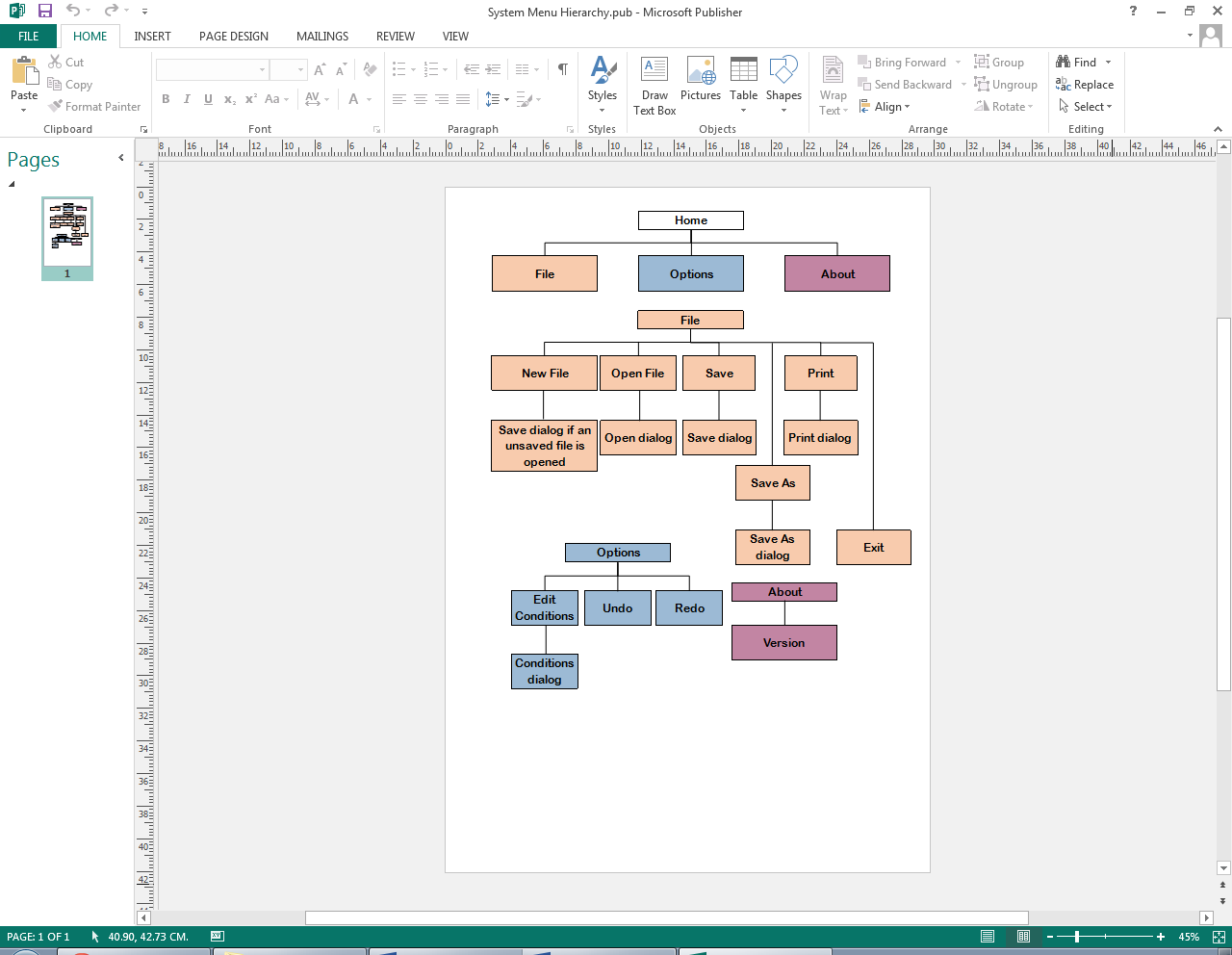
* Worked calculation of equilibrium constant of given reaction
* Simplified animation representing a reversible reaction, adjustable by changing conditions
* Graphs of reactant/product concentration or forward/backward reaction rate against time, adjustable by changing conditions
* Side-by-side comparison of reaction data
* Automatic organisation of reaction data for printing
* Simple explanation of how Kc is affected by changes in conditions
* Save and load files exclusive to the program, consisting of up to 5 reactions per file

Additional features include:

* Being able to change the colours representing the reactants and products
* Ability to show/hide certain information via a check box
* Functions for students attempting to balance equations

The program was produced in Python 3.2 with PyQt4.

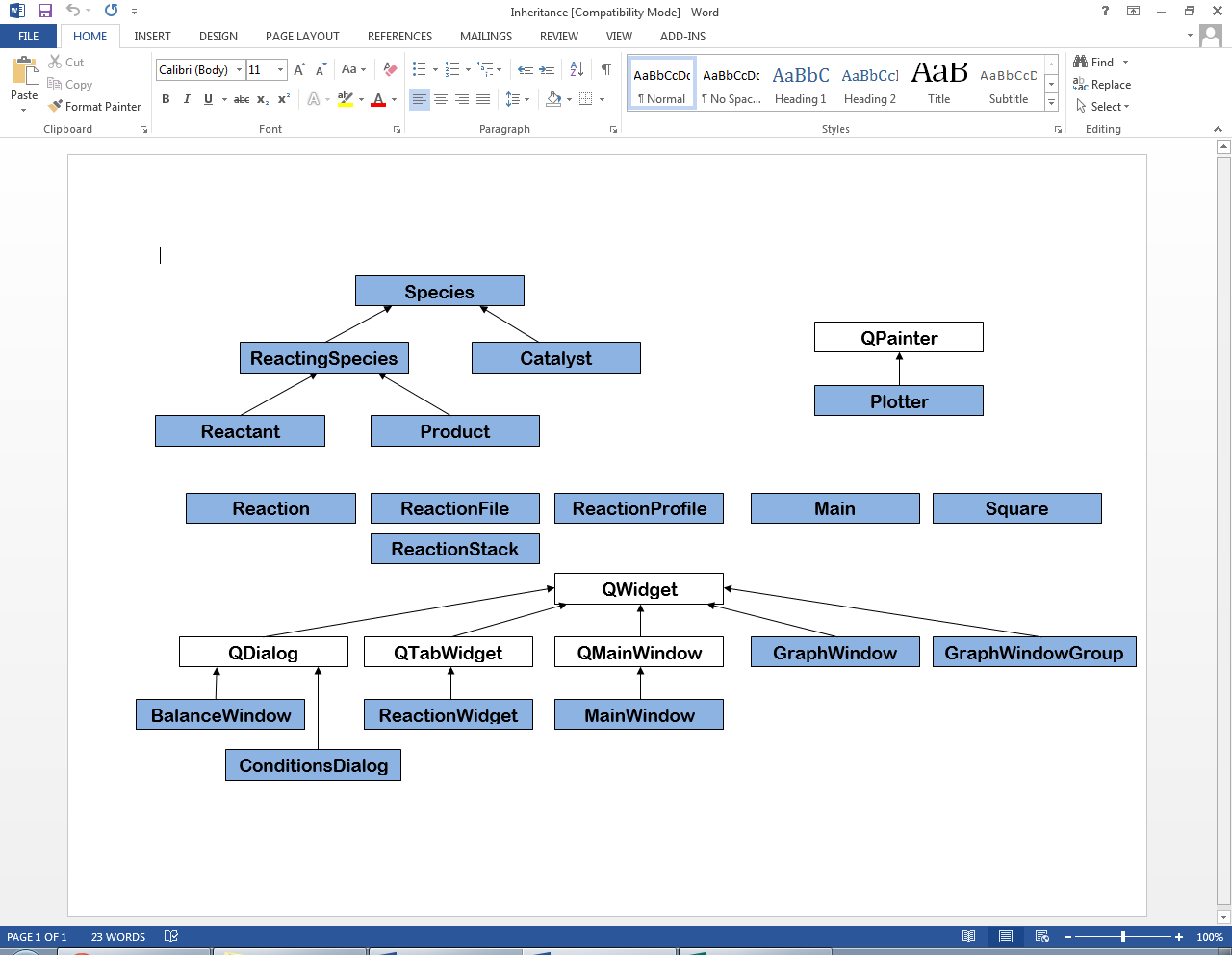
b) Menu Hierarchy



c) User-Defined Classes

i: Inheritance Overview

Classes coloured white are built in to PyQt and extended. Classes in blue are user-defined.



Basic class descriptions

* Species: any chemical which is somehow involved in a reaction; this includes reactants, products and catalysts.
* ReactingSpecies: any chemical which reacts in the reaction; this includes reactants and products, but not the catalyst.
* Reactant: in an ordinary reaction these are the chemicals present at the start, although in a reversible reaction these are not completely removed. These act as the products of the opposite (“backwards”) reaction, which takes place at the same time.
* Product: in an ordinary reaction these are the chemicals present at the end, but in reversible reactions these do not completely replace the reactants. These act as the reactants of the opposite (“backwards”) reaction, which takes place at the same time.
* Catalyst: A chemical, the presence of which alters the rate of both the reactions which take place. The catalyst is not used up or converted at any point; it can be thought of as a surface on which the reaction takes place.
* Plotter: A painting device specifically for plotting graphs based on the reaction.
* Reaction: Represents the chemical reaction.
* ReactionFile: Each file saved by the program is a binary dump of a ReactionFile object. ReactionFile objects consist of a list of up to 5 Reaction objects, and an integer which is the index of the last Reaction object to be viewed in the list.
* ReactionProfile: The main screen, which shows the equation of the reaction, animation and buttons which show other information
* Main: This class handles program startup.
* Square: Each object is one of the coloured squares used in the animation.
* ReactionStack: Holds alternate versions of a reaction when the Undo or Redo functions are performed on it.
* GraphWindow: Displays a graph of either concentration or reaction rate during the reaction.
* GraphWindowGroup: A window containing one GraphWindow for each ReactionProfile on-screen.
* BalanceWindow: The window which students use to attempt to balance the reaction’s equation.
* ConditionsDialog: Where all main factors (conditions) related to the reaction are edited.
* ReactionWidget: A tabbed section representing the list of reactions in ReactionFile. Clicking tabs allows users to switch between reactions.
* MainWindow: Contains the menu bar, a ReactionWidget object on the left and a separate ReactionProfile object on the right, for comparison with the current ReactionProfile object on the left. Only the one on the left can be edited.

ii: In-Depth Class Descriptions

Note: Because of Python’s typing system the types specified for parameters are not guaranteed to be that type at runtime, but the program was designed assuming that each parameter will have that type.

Public methods are defined as:

* <identifier>(self, <param data type> <param name>, ... ) : <data type returned>

Catalyst (chem/Catalyst.py)

Parent class: Species  
No child classes

Public methods:

* \_\_init\_\_(self)
* GetEfficacyAsString(self) : str

Returns a verbal description of catalyst strength

* GetEfficacy(self) : float

Returns catalyst strength, represented by a float value

* SetEfficacy(self, int e)

No new public fields

New private fields:

* Efficacy : float – the strength of the catalyst

Product (chem/Product.py)

Parent class: ReactingSpecies  
No child classes

Public methods:

* \_\_init\_\_(self)

No new public fields

No new private fields

Reactant (chem/Reactant.py)

Parent class: ReactingSpecies  
No child classes

Public methods:

* \_\_init\_\_(self)

No new public fields

No new private fields

ReactingSpecies (chem/ReactingSpecies.py)

Parent class: Species  
Child classes: Reactant; Product

Public methods:

* \_\_init\_\_(self)
* GetSRatio(self) : int

Returns the species’ stoichiometric ratio.

No new public fields

No new private fields

Reaction (chem/Reaction.py)

No parent classes  
No child classes

Public methods:

* \_\_init\_\_(self)
* GetKc(self) : str

Calculates the value and units of Kc.

The value (x) is found by calculating the numerator and the denominator, then dividing the numerator by the denominator. The numerator is the product of the concentration of each product to the power of the product’s stoichiometric ratio. The denominator is the product of the concentration of each reactant to the power of the product’s stoichiometric ratio. The overall units are found by calculating a mole power (y): this is the sum of the stoichiometric ratios of the reactants, subtracted from the sum of the stoichiometric ratios of the products. The volume power (z) is easily calculated by multiplying y by 3. The function returns a string in the form “Kc = x mol^y dm^z”.

Local variables:

* reactants : list(Reactant) – a list of reactants which were selected to be used
* products : list(Product) – a list of products which were selected to be used
* x : int – stepper for iterating through the above lists
* numerator : float – the numerator explained above
* denominator : float – the denominator explained above
* value : float – the actual value of Kc (numerator divided by denominator)
* rpower : int – the total of the stoichiometric values of each reactant
* ppower : int – the total of the stoichiometric values of each product
* power : int – ppower subtract rpower; used to calculate the overall units of Kc
* GetKcChange(self, int temperaturechange) : str

If the two reactions are both endothermic or both not, this function returns “increased”, “decreased” or “not changed” depending on a) how the temperature of the left reaction compares with that of the right reaction; b) whether the reactions are endothermic or not.

* GetTemperature(self) : int

Returns the current reaction’s temperature.

* SetTemperature(self, int newtemp)
* GetReactants(self) : list(Reactant)

Returns a list of all the current reaction’s reactants, used or not.

* SetReactants(self, list(Reactant) newr)
* GetProducts(self) : list(Product)
* Returns a list of all the current reaction’s products, used or not.
* SetProducts(self, list(Product) newp)
* GetCatalyst(self) : Catalyst

Returns the current reaction’s catalyst, used or not.

* SetCatalyst(self, Catalyst newc)
* GetVolume(self) : float

Returns the volume of the vessel in the current reaction.

* SetVolume(self, float newv)
* GetEndothermic(self) : boolean

Returns whether the current reaction is endothermic (takes in heat when products are produced) or not.

* SetEndothermic(self, boolean newe)
* GetReactantColour(self) : QColor

Returns the colour that represents the reactants.

* SetReactantColour(self, QColor colour)
* GetProductColour(self) : QColor

Returns the colour that represents the products.

* SetProductColour(self, QColor colour)

Public fields:

* REACTING\_SPECIES\_LIMIT: constant describing the maximum number of reacting species each side of the equation (default value 3)

Private fields:

* Temperature : int – the reaction’s temperature
* Reactants : list(Reactant) – all the reactants in the reaction, used or not
* Products : list(Product) – all the products in the reaction, used or not
* Catalyst : Catalyst – the reaction’s catalyst, used or not
* Volume : float – the volume of the vessel the reaction would take place in
* Endothermic : boolean – whether the reaction takes in or gives out heat overall, when reactants are converted into products
* ReactantColour : QColor – the colour which represents the reactants in the reaction
* ProductColour : QColor – the colour which represents the products in the reaction

Species (chem/Species.py)

No parent classes  
Child classes: ReactingSpecies; Catalyst

Public methods:

* \_\_init\_\_(self)
* GetConcentration(self, float volume) : float

Returns the species’ concentration (number of moles divided by volume).

* GetFormulaForLabels(self) : str

Replaces instances of “\_x” in its formula with “<sub>x</sub>” and “^x” with “<sup>x</sup>”, so that it can return an HTML-compatible string so that superscript and subscript numbers can be displayed in a more user-friendly way.

Local variables:

* + x : int – stepper for iterating through the formula
  + newformula : string – starts empty and is added to depending on the current character in the formula
* GetMoles(self) : float

Returns the number of moles of this species in the current reaction.

* SetMoles(self, float newvolume)
* GetFormula(self) : str

Returns the species’ formula. Different to GetFormulaForLabels because it still uses \_ and ^. This is the formula as it would be entered by the user.

* SetFormula(self, str newformula)
* GetUsed(self) : boolean

Returns whether the user has marked the species to be used in the reaction.

* SetUsed(self, boolean u)

No public fields

Private fields:

* Formula : string – the species’ chemical formula
* Moles : float – the number of moles of this species in the reaction
* Used : boolean – whether or not the species should be displayed to the user and factored in to calculations

BalanceWindow (gui/BalanceWindow.py)

Parent class: QDialog  
No child classes

Public methods:

* \_\_init\_\_(self, Reaction reaction, MainWindow window)

Local variables:

* + x : int – stepper for iterating through the reactant and product lists of reaction, and this window’s reactant and product lists (each pair of lists is different because the lists of reaction include reacting species which are not being used)
  + species : ReactingSpecies – the next reacting species to be added to its corresponding list
  + box : QSpinBox – the next box to be added to the window’s list of answer boxes
  + formula : string – the formula of the current reacting species, with the stoichiometric value removed
  + arrows : QLabel – a label containing an image of rotated double arrows
* paintEvent(self, QPaintEvent QPaintEvent)

Draws the window’s layout.

Local variables:

* x : int – stepper for iterating through this window’s reactant and product lists
* close(self)

Reveals the windows that it hid when it opened, then closes itself.

No new public fields

New private fields:

* grid : QGridLayout – the grid-based layout of the widgets in the window
* reactants : list(Reactant) – all the reactants which have been selected to be used
* products : list(Product) – all the products which have been selected to be used
* boxes : list(QSpinBox) – a list of drop-down boxes supporting the values 1 to 9, for users to select their answer
* formulae : list(string) – a list of the formulae of the reacting species used in the reaction
* symbols : list(QPixmap) – a list of images. These will be a tick or a cross, depending on whether or not the students’ guessed stoichiometric value matches the correct value
* themainwindow : MainWindow – the main program window
* okbtn : QPushButton – a simple button which says “OK”, and closes the window when pressed

ConditionsDialog (gui/ConditionsDialog.py)

Parent class: QDialog  
No child classes

Public methods:

* \_\_init\_\_(self, Reaction reaction)

Local variables:

* + efficacy : int – the strength of the catalyst in the reaction
* ApplyChanges(self)

Runs the validation code on all data that will be used in the reaction. If any data is invalid an error message appears and the user cannot continue; if all data is valid the reaction data is edited accordingly and the user is allowed to continue.

Local variables:

* errorlist : list(string) – the text which is displayed in the error message which appears if there is any invalid data. A string is added for each item of invalid data which generally corresponds to the error made
* formularegex : string – a regular expression string for validating formulae. Current value is: "[2-9]?(\(([A-Z][a-z]?(\_[0-9])\*)+\)(\_[0-9])\*|[A-Z][a-z]?(\_[0-9])\*)+((\^[0-9])\*\^(\+|-))?"
* molesorvolumeregex : string – a regular expression string for validating volume or number of moles values. Current value is: "[0-9]?[0-9]\.[0-9][0-9]?"
* x :int – stepper for iterating through all the lists
* formula : string – the formula currently being validated
* moles : string – the number of moles currently being validated (user cannot input moles as float; only string)
* volume : string - the volume string to be validated
* texttoadd : string – contains all the strings taken from errorlist, separated by “\n” for a new line. This string has more memory than normal reserved for it (declared using “””””” rather than “”) because if all input data is invalid, the string will be very long
* GetReaction(self) : Reaction

Returns this window’s current reaction.

* GetCanShowFormulae(self) : boolean

Returns whether or not the MainWindow that opened this one can display the current reaction’s equation.

No new public fields

New private fields:

* CanReturnReaction : boolean – whether or not the window should return the edited reaction to the window that opened it
* reactantbox : QGroupBox – a group of the widgets that relate to reactant data
* reactantform : QFormLayout – manages the layout of reactantbox
* reactantinfoboxes : list(QLineEdit) – a list of all the text boxes for entering reactant data
* productbox : QGroupBox – a group of the widgets that relate to product data
* productform : QFormLayout – manages the layout of productbox
* productinfoboxes : list(QLineEdit) – a list of all the text boxes for entering product data
* catalystbox : QGroupBox – a group of the widgets that relate to catalyst data
* catalystform : QFormLayout – manages the layout of catalystbox
* catalystinfoboxes : list(QLineEdit) – a list of all the text boxes for entering catalyst data
* checkbuttons : list(QCheckBox) – a list of all the check boxes in the window
* good : QRadioButton – if selected when “OK” is pressed, the catalyst will be of good strength
* poor : QRadioButton – if selected when “OK” is pressed, the catalyst will be of poor strength
* inhibitor : QRadioButton – if selected when “OK” is pressed, the catalyst will be an inhibitor
* cataefficacygroup : QVBoxLayout – a group of the above QRadioButtons
* endo : QRadioButton – if selected when “OK” is pressed, the reaction will be endothermic
* exo : QRadioButton – if selected when “OK” is pressed, the reaction will not be endothermic
* heatbox : QGroupBox – a group of the widgets that relate to forward heat transfer data
* heatform : QFormLayout – manages the layout of heatbox
* otherbox : QGroupBox – a group of the widgets that relate to other reaction data (still mandatory)
* otherform : QFormLayout – manages the layout of otherbox
* scriptinfo1 : QLabel – contains a string explaining how to enter subscript data into the text boxes
* scriptinfo2 : QLabel – contains a string explaining how to enter superscript data into the text boxes
* try : QCheckBox – if selected, a BalanceWindow will be displayed so that students can attempt to balance the reaction equation
* okbtn : QPushButton – an “OK” button. If pressed, the window attempts to validate the data in each text box. An error message is displayed if any data is invalid. If all data is valid the window simply closes and the changes are applied
* cancelbtn : QPushButton – a “Cancel” button. If pressed, the window closes without any changes being applied
* rightbox : QGroupBox – groups together the data on the right (catalyst, forward heat transfer, other conditions)
* rightform : QVBoxLayout – manages the layout of rightbox
* reaction : Reaction – the reaction passed in, unless a Reaction object fails to be passed in, in which case a new Reaction object is instantiated
* errorbox : QMessageBox – the window which appears if any data is invalid
* showformulae : boolean – whether or not the formulae should be shown on the window which opened this window.

GraphWindow (gui/GraphWindow.py)

Parent class: QWidget  
No child classes

Public methods:

* \_\_init\_\_(self, Reaction reaction, str graphof, QWidget parentwidget, boolean forprinting)
* paintEvent(self, QPaintEvent e)

Draws the window’s layout.

Local variables:

* change : float – represents the overall effect of the catalyst in the reaction. Starts as 1 in case no catalyst is used, but if one is used, change is set to the catalyst’s efficacy
* eqmpoint : float – transformation of change according to the size of the graph and the number of moles of catalyst
* pen : QPen – the pen object which draws the graph
* reactantpath : QPainterPath – the path that the pen should take to draw the curve for the reactants
* productpath : QPainterPath – the path that the pen should take to draw the curve for the products
* white : QColor – the colour of the graph background

No new public fields

New private fields:

* graphof : string – either “Concentration” or “Rate”
* graph : QWidget – the widget the graph should be drawn on
* plotter : Plotter – the object which draws the graph on graph
* reaction : Reaction - the reaction passed in
* parentwidget : QWidget – the widget that this window is contained in
* forprinting : boolean – whether or not the widget has been instantiated in order to be printed (graph not initially obscured)

GraphWindowGroup (gui/GraphWindowGroup.py)

Parent class: QWidget  
No child classes

Public methods:

* \_\_init\_\_(self, GraphWindow leftwindow, GraphWindow rightwindow)
* updateReactions(self)

Calls the update() method of its left, then right, GraphWindow objects.

No new public fields

New private fields:

* Left : GraphWindow – the graph on the left
* Right : GraphWindow – the graph on the right
* closebtn : QPushButton – a simple “Close” button.
* grid : QGridLayout – manages the layout of the other widgets

Main (gui/main.py)

No parent class  
No child classes

Public methods:

* \_\_init\_\_(self)

No public fields

Private fields:

* Application : QApplication – an application object which forms the base of the program
* CurrentFile : ReactionFile – the file currently being manipulated in the program
* Window : MainWindow – the program’s main window

MainWindow (gui/MainWindow.py)

Parent class: QMainWindow  
No child classes

Public methods:

* \_\_init\_\_(self)

Local variables:

* newfile : QAction – the “New” button in the File menu
* openfile : QAction – the “Open” button in the File menu
* savefile : QAction – the “Save” button in the File menu
* saveasfile : QAction – the “Save as” button in the File menu
* printfile : QAction – the “Print” button in the File menu
* exitaction : QAction – the “Exit” button in the File menu
* editconds : QAction – the “Edit Conditions” button in the “Options” menu
* version : QAction – the “Version” button in the “About” menu
* editcd(self)

Shows the “Edit Conditions” dialog.

* showopen(self)

Shows the “Open...” dialog.

Local variables:

* + fileaddress : string – the address of the file selected in the “Open...” dialog
* open(self, str fileaddress)

Loads the file at fileaddress, writes fileaddress in the file last.ptr, and sets the current ReactionFile to the loaded file. Also adds fileaddress to the window title.

Local variables:

* + openfrom : file – the file at fileaddress
  + loadedfile : ReactionFile – the ReactionFile object read from openfrom
  + file : file – the file “last.ptr”
* save(self)

Calls ReactionWidget.save().

* showsaveas(self)

Shows the “Save As...” dialog.

* shownew(self)

If the current file hasn’t been saved yet, a message box opens asking the user if they want to save.

Local variables:

* file : file – the file “last.ptr”
* showprint(self)

Shows the “Print” dialog.

* printout(self)

Organises the file into an A4 layout and sends the layout to the printer chosen in the “Print” dialog.

Local variables:

* pw : float – the width of the page to print on
* ph : float – the height of the page to print on
* ww : float – the width of a reaction widget
* painter : QPainter – the device which draws images of widgets onto the page
* scale : float – the factor to scale each widget image by
* GetWindowTitle(self) : str

Returns the string displayed at the very top of the main window – generally “Equilibrium Simulator 2015” but if a file was opened recently its filepath will be included in the string.

* SetComparingReaction(self, Reaction reaction)
* GetComparingReaction(self) : Reaction

Returns the reaction on the right-hand side of the main-window.

New public fields:

* MenuBar – the bar across the top of the window
* filemenu – the File menu
* optionsmenu – the Options menu
* aboutmenu – the About menu

New private fields:

* TITLE : string – the string displayed at the very top of the window
* showorhide : string – describes whether the graph buttons should say “Show graph” or “Hide graph”
* printbox : QPrintDialog – the “Print” dialog window
* printer : QPrinter – the printer that the final document will be sent to
* Squares : list(Square) – a list of all the square objects in the animation
* ReactionsWindow : ReactionWidget – the widget on the left; a tabbed group of up to 5 reaction screens
* ComparingProfile : ReactionProfile – the reaction screen on the right
* newfilebox : QMessageBox – a window asking the user if they want to save the current file when opening a new one

Plotter (gui/Plotter.py)

Parent class: QPainter  
No child classes

Public methods:

* \_\_init\_\_(self, QWidget area, str graphof)
* GetFinalY(self) : int

Returns the distance from the middle of the graph where the graph curves end. This is in the middle on a rate graph, but a higher value on a concentration graph.

No new public fields

New private fields:

* GraphArea : QWidget – the area the Plotter will operate on
* ShowLabels : boolean – whether or not labels will be shown
* finalY : int – the distance from the middle of the graph where the curves will stop

ReactionProfile (gui/ReactionProfile.py)

Parent class: QWidget  
No child classes

Public methods:

* \_\_init\_\_(self, Reaction reaction, boolean readonly)

Local variables:

* + formulafont : QFont – the font used to draw the equation
* paintEvent(self, QPaintEvent e)

Draws the window’s layout.

Local variables:

* reactantside : string – the reactant side of the equation
* productside : string – the product side of the equation
* target : QRectF – the rectangle to draw arrows in
* arrows : QPixmap – the double arrow image
* portion : QRectF – the area of arrows to draw in target
* SetReaction(self, Reaction reaction)
* GetReaction(self) : Reaction

Returns the reaction being displayed.

* MakeConcGraphs(self)

Displays a GraphWindowGroup with the concentration GraphWindows for the current (left) and compared (right) reactions.

Local variables:

* gwg : GraphWindowGroup – a side-by-side graph comparison to be displayed
* MakeRateGraphs(self)

Displays a GraphWindowGroup with the rate GraphWindows for the current (left) and compared (right) reactions.

Local variables:

* gwg : GraphWindowGroup – a side-by-side graph comparison to be displayed
* SetReactantColour(self)

Shows the “Select Color” dialog for reactants.

Local variables:

* picker : QColorDialog – a colour palette window
* SetProductColour(self)

Shows the “Select Color” dialog for products.

Local variables:

* picker : QColorDialog – a colour palette window
* GetKc(self)

Doesn’t calculate Kc, but actually reformats the string returned by Reaction.GetKc() to an HTML-compatible string for GUI display. It then updates the screen to display the string.

Local variables:

* kc : string – the original Kc string
* newkc : string – a copy of kc, reformatted to include HTML tags
* x : int – a stepper used to iterate on kc
* ShowWorking(self)

Displays a window of plain text which provides a step by step calculation of the value and units of Kc.

Local variables:

* working : QPlainTextEdit – the window which will contain the worked solution in plain text
* reactants : list(Reactant) – a list of all reactants that are selected to be used
* products : list(Product) – a list of all products that are selected to be used
* volume : float – the reaction volume
* pvalues : list(float) – a list of product concentrations and stoichiometric values
* rvalues : list(float) – a list of reactant concentrations and stoichiometric values
* kcvalue : string – contains the final value and units of Kc
* x : int – stepper used for iterating on pvalues, rvalues, products and reactants
* pproductsum : int – the total of the stoichiometric values of the products
* rproductsum : int – the total of the stoichiometric values of the reactants
* EnableAnimation(self)

If time is -1 (animation should not play), sets it to 0 (first frame of animation).

* SetUpAnimation(self, QPainter painter)

Clears the window’s array of Square objects and draws a new 10x10 grid of Square objects (adding each one to the array).

* StartOrContinueAnimation(self, QPainter painter)

If time is -1 (animation should not play), resets all the Squares. If more than that, the colour of each square is changed based on a random chance, and the time is incremented by 1. The random chance is decided by the time and the catalyst strength and number of moles. If the time reaches 1500, it is set to -1 again.

Local variables:

* eqmpoint : float – the number of screen updates before the reaction is supposed to reach equilibrium
* efficacy : float – the strength of the catalyst
* chance : int – the odds of a square changing colour
* GetTime(self) : int

Returns the time value used in the animation.

* RemoveMe(self, QWidget qwidg)

Removes the given widget from the ExtraWindows list. All widgets in this list are made to update when this widget updates.

* Clone(self)

Sets the reaction being compared to a copy of the current reaction.

* SetKcDifference(self, str string)
* PrintGraph(self, QPainter painter, str graphof)

Renders a snapshot of a GraphWindow, which is sent to the printer.

No public fields

ReactionWidget (gui/ReactionWidget.py)

Parent class: QTabWidget  
No child classes

Public methods:

* \_\_init\_\_(self)
* newtab(self)

Opens a new tab with a blank reaction.

Local variables:

* length : int – the number of reactions in the current file
* save(self)

Overwrites the current file with the reactions in their current state, unless the file is new, in which case showsaveas() is called.

Local variables:

* saveto : file – the file last.ptr. If fileaddress is valid, saveto becomes the file to write the current ReactionFile object into
* fileaddress : string – the address stored in last.ptr
* showsaveas(self)

Shows the “Save As...” dialog.

Local variables:

* fileaddress : string – the address selected in the “Save As...” dialog to save the file to
* saveas(self, str fileaddress)

Saves the current reaction file at fileaddress.

Local variables:

* saveto : file – the file in which to write the current ReactionFile object. After the file is written, saveto is used to represent the last.ptr file
* EditReaction(self)

Opens the Conditions Dialog and applies changes if OK was pressed and the data was valid. Also responsible for displaying the difference in Kc between the left and right reactions.

Local variables:

* currentreaction : Reaction – the reaction currently being displayed on the left-hand side
* dialog : ConditionsDialog – an instance of the “Edit Conditions” window
* reaction : Reaction – the reaction being edited in dialog
* copiedreaction : Reaction – the reaction currently being displayed on the right-hand side
* currentprofile : ReactionProfile – the screen displaying currentreaction
* oldt : int – the temperature of copiedreaction
* newt : int – the temperature of currentreaction (comparison is done by copying a reaction then changing a condition on the original, so the temperature of the copied reaction is seen as the “old” temperature)
* RemoveReaction(self, int index)

Removes the reaction at index in the current file’s list, and closes the corresponding tab.

* SetCurrentReaction(self)

Updates the current reaction, and sets the index of the last tab that was open.

* GetCurrentReaction(self) : Reaction

Returns the reaction currently being displayed.

* SetOpenReactionFile(self, ReactionFile file)

Refreshes the window using file.

Local variables:

* x : int – stepper used to iterate over the list of reactions in the file
* index : int – the index of the last viewed reaction in the list

No new public fields

New private fields:

* CurrentFile : ReactionFile – the current ReactionFile object being manipulated
* CurrentReaction : Reaction – the current Reaction object being manipulated
* plusbtn : QPushButton – the “+” button on the right of the tab bar
* errorbox : QMessageBox – an error window

Square (gui/Square.py)

No parent class  
No child classes

Public methods:

* \_\_init\_\_(self, int x, int y, Reaction reaction)
* flip(self)

Changes from reactant colour to product colour or vice versa.

* Draw(self, QPainter painter, int chance)

Draws itself on the screen.

* Reset(self)

Changes to its reactant colour.

No public fields

Private fields:

* x : int – the x-coordinate of the square’s top-left corner
* y : int – the y-coordinate of the square’s top-left corner
* currentcolour : QColor – the current colour of the square; only equal to reactantcolour or productcolour
* reactantcolour : QColor – the colour representing the reactants
* productcolour : QColor – the colour representing the products

ReactionFile (ReactionFile.py)

No parent class  
No child classes

Public methods:

* \_\_init\_\_(self)
* AddReaction(self, Reaction r)

Adds a reaction to the list, if the list is less than 5 items in length.

* DeleteReaction(self, int index)

Removes a reaction from the list. If the list is 1 item long, the item is replaced with a blank reaction instead.

* GetReactions(self) : list(Reaction)

Returns the current list of reactions.

* GetLastTabOpen(self) : int

Returns the index of the last tab that was opened (the last reaction that was viewed).

* SetLastTabOpen(self, int i)

No public fields

Private fields:

* Reactions : list(Reaction) – a list of the reactions stored in the file
* LastTabIndex : int – the index of the last-viewed reaction object in Reactions

2: Software Analysis

The program was written in Python due to the relatively short time frame of four months, for one person. It was necessary to develop in a language where algorithms and an object-oriented system can be implemented quickly without having to spend time on specifics such as garbage collection. Furthermore, Python’s range of standard libraries means that a developer avoids having to download additional modules for the program. Python is quite slow compared to other programming languages but this shortcoming is not significant in a school environment and is far outweighed by the benefits.

PyQt was chosen to be used with Python because it is a popular choice for developers who want to create GUI-based applications. This was considered important as the end user suggested, and survey results showed, that there is a wide range of IT ability in the Chemistry department and it would be useful for the program to resemble the software they already use, so that teachers (and students) are comfortable using the program. PyQt accomplishes this very well – it provides a large range of controls, window styles and GUI layouts which don’t look out of the ordinary, and that the department is very familiar with using. It is also very quick and easy to program the GUI. PyQt’s animation facilities are limited but the end user did not demand complicated animation, so PyQt was sufficient.

It was necessary to extend some classes to provide a connection between the PyQt interface and the chemical concepts of the project. This involved the creation of some classes which inherited from PyQt classes (such as ConditionsDialog from QDialog) and contained extra properties and methods for handling the logic and calculation involved in the chemical concepts. This was generally done by including a Reaction object as one of the parameters of these classes’ constructors.

3: Algorithm Design

Calculating Kc

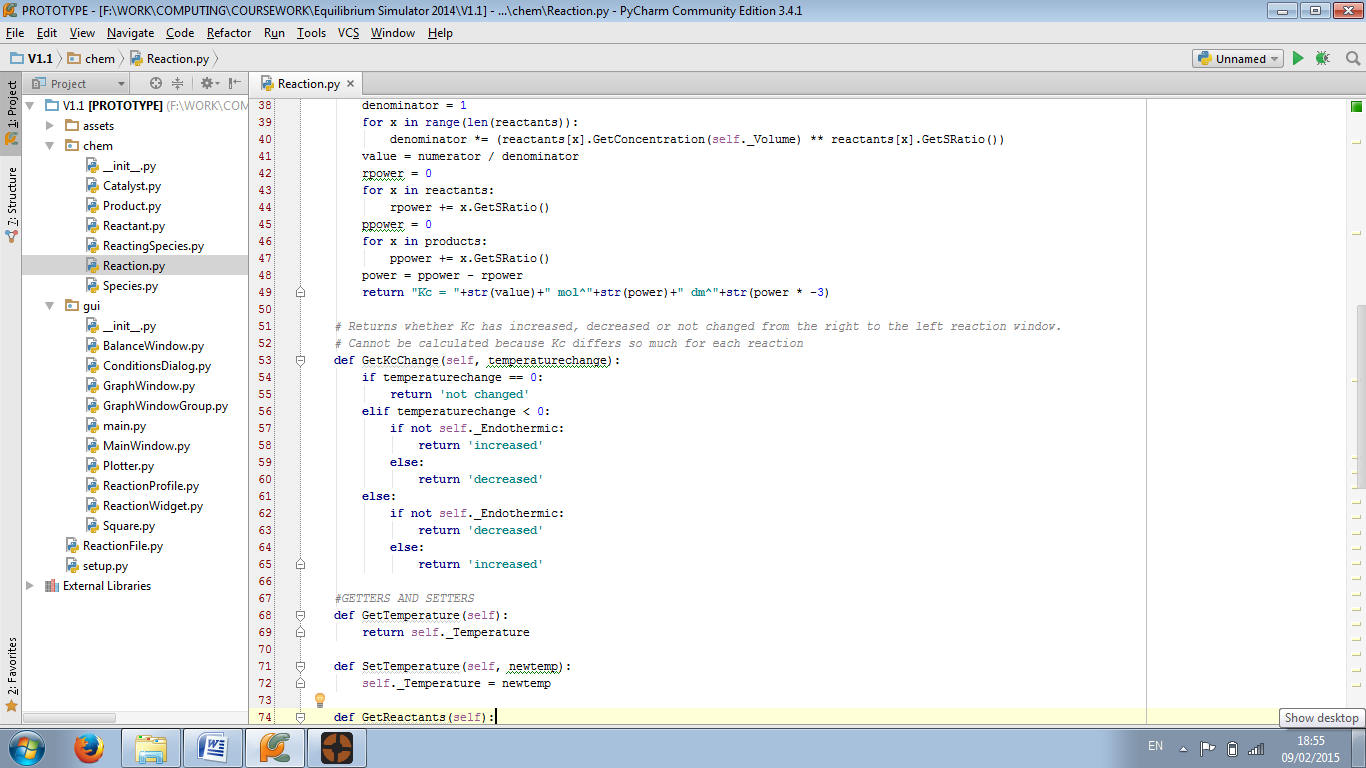
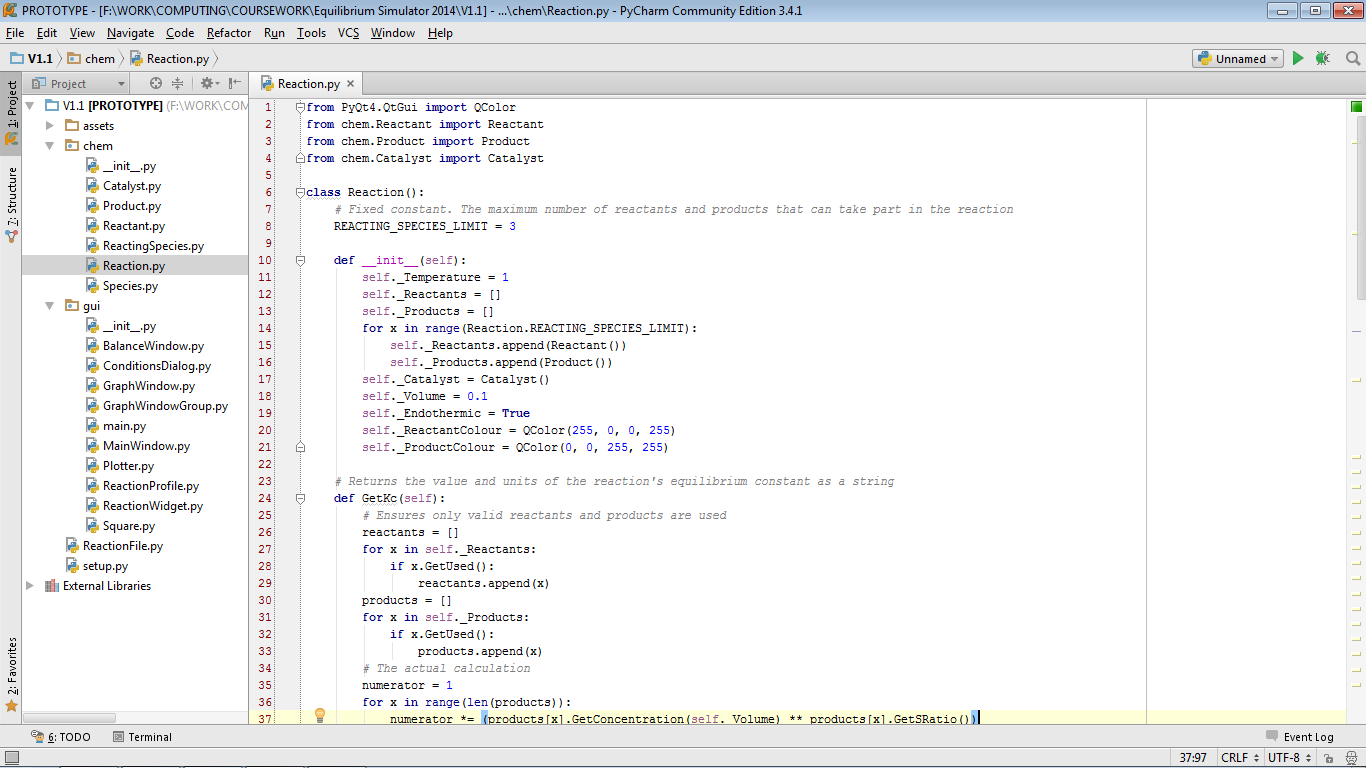
The GetKc(self) function is in the Reaction class and uses its properties. It returns the value and units of the reaction’s equilibrium constant. The process is as follows:

1. The reactants and products lists are initialized as empty.
2. For each reactant and product in the reaction, the algorithm checks if they are used. If so they are appended to the corresponding list. From here on “reactants” and “products” only refers to the ones in these lists.
3. The numerator is initialized as 1.
4. For each product in turn, numerator is multiplied by its concentration to the power of its stoichiometric ratio.
5. For each reactant in turn, denominator is multiplied by its concentration to the power of its stoichiometric ratio.
6. value is initialized as numerator divided by denominator.
7. The total of the stoichiometric ratios for each product is calculated.
8. The total of the stoichiometric ratios for each reactant is calculated.
9. Concentration is measured in mol dm-3. The overall units are found by dividing the product of the units of concentration of the products by the product of the units of concentration of the reactants. In the algorithm, the power of the moles is simply the total of the ratios of the products minus that of the reactants. The power of the volume is this multiplied by -3.
10. A string containing value and the two powers is returned.

Pseudocode

FUNCTION GetKc(self)  
 reactants 🡨 []  
 FOR x 🡨 0 TO self.\_Reactants.length – 1 DO  
 IF self.\_Reactants[x].GetUsed() == TRUE THEN  
 reactants.append(self.\_Reactants[x])  
 ENDIF  
 ENDFOR  
 products 🡨 []  
 FOR x 🡨 0 TO self.\_Products.length – 1 DO  
 IF self.\_Products[x].GetUsed() == TRUE THEN  
 products.append(self.\_Products[x])  
 ENDIF  
 ENDFOR  
 numerator 🡨 1  
 FOR x 🡨 0 TO products.length – 1 DO  
 numerator 🡨 numerator \* (products[x].GetConcentration(self.\_Volume) \*\* products[x].GetSRatio())  
 ENDFOR  
 denominator 🡨 1  
 FOR x 🡨 0 TO reactants.length – 1 DO  
 denominator 🡨 denominator \* (reactants[x].GetConcentration(self.\_Volume) \*\* reactants[x].GetSRatio())  
 ENDFOR  
 value 🡨 numerator / denominator  
 rpower 🡨 0  
 FOR x 🡨 0 TO reactants.length - 1 DO  
 rpower 🡨 rpower + reactants[x].GetSRatio()  
 ENDFOR  
 ppower 🡨 0  
 FOR x 🡨 0 TO products.length – 1 DO  
 ppower 🡨 ppower + reactants[x].GetSRatio()  
 ENDFOR  
 power 🡨 ppower – rpower  
 RETURN “Kc = “+str(value)+” mol^”+str(power)+” dm^”+str(power \* -3)  
END

Program Code



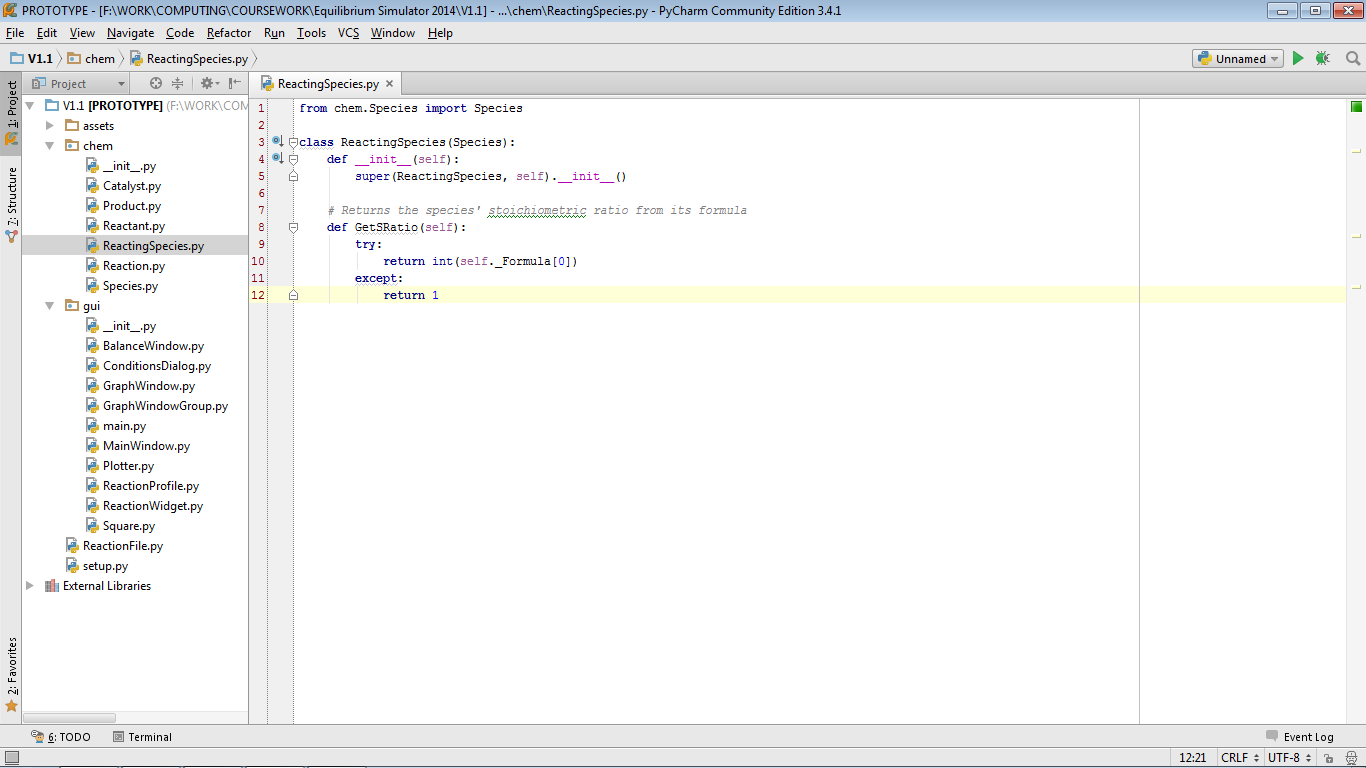
Obtaining Stoichiometric Ratios

The GetSRatio(self) function is in the ReactingSpecies class and uses its properties. It returns the reacting species’ stoichiometric ratio based on its formula. In parallel with convention, a formula which does not begin with a number is implied to have a ratio of 1. The validation regex means that currently, ratios may not be more than 9.

Pseudocode

FUNCTION GetSRatio(self)  
 TRY  
 RETURN int(self.\_Formula[0])  
 EXCEPT  
 RETURN 1  
 ENDTRY  
END

Program code



Making strings HTML-compatible

The GetFormulaForLabels function is in the Species class. It returns the species’ formula (which was entered by the user as plain text) in a HTML-based format so that it can be displayed to the end user according to convention. It does this by making a copy of the formula, and replacing every instance of “\_x” with “<sub>x</sub> and every instance of “^x” with “<sup>x</sup>” where x is an integer. This results in the formula having superscript and subscript characters in the right places.

Pseudocode

FUNCTION GetFormulaForLabels

x 🡨 0

newformula 🡨 ""

WHILE x < self.\_Formula.length DO

IF self.\_Formula[x] == "\_" THEN

newformula 🡨 newformula + "<sub>"

x 🡨 x + 1

newformula 🡨 newformula + self.\_Formula[x]

newformula 🡨 newformula + "</sub>"

ELSE

IF self.\_Formula[x] == "^" THEN

newformula 🡨 newformula + "<sup>"

x 🡨 x + 1

newformula 🡨 newformula + self.\_Formula[x]

newformula 🡨 newformula + "</sup>"

ELSE

newformula 🡨 newformula + self.\_Formula[x]

ENDIF

ENDIF

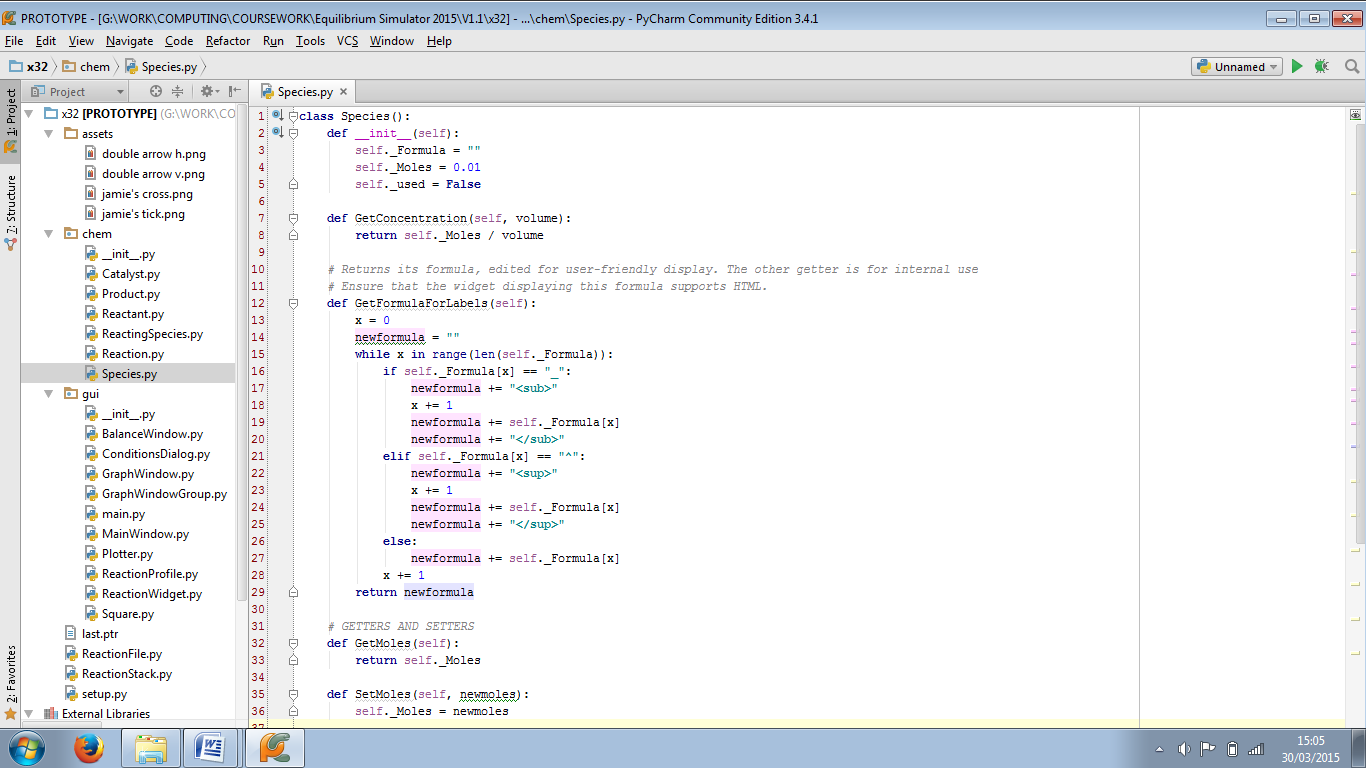
x 🡨 x + 1

ENDWHILE

RETURN newformula

END

Program code



Input validation

The ApplyChanges(self) function is in the ConditionsDialog class and uses its properties. It validates all data input in this window. Sometimes this includes using regular expressions. Whenever there is invalid data, a message is added to a long output string displayed in an error box.

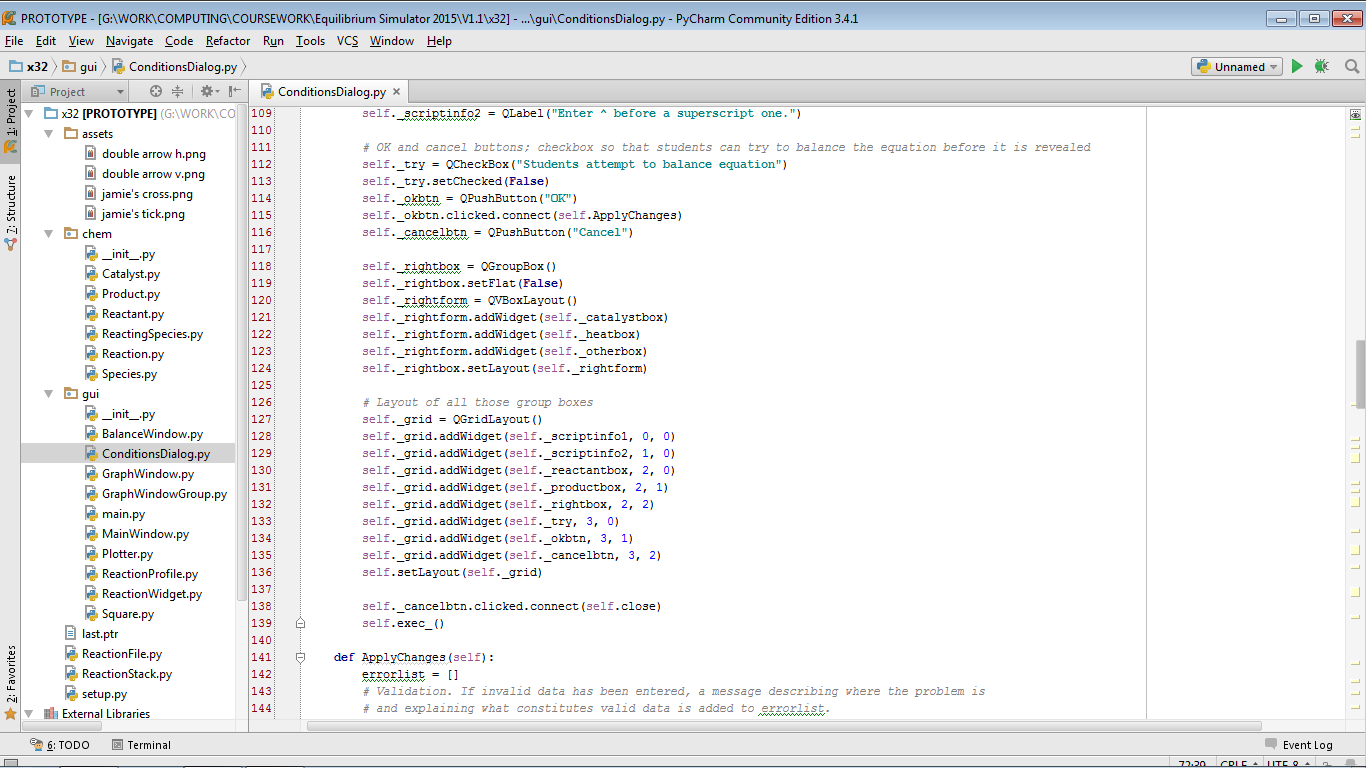
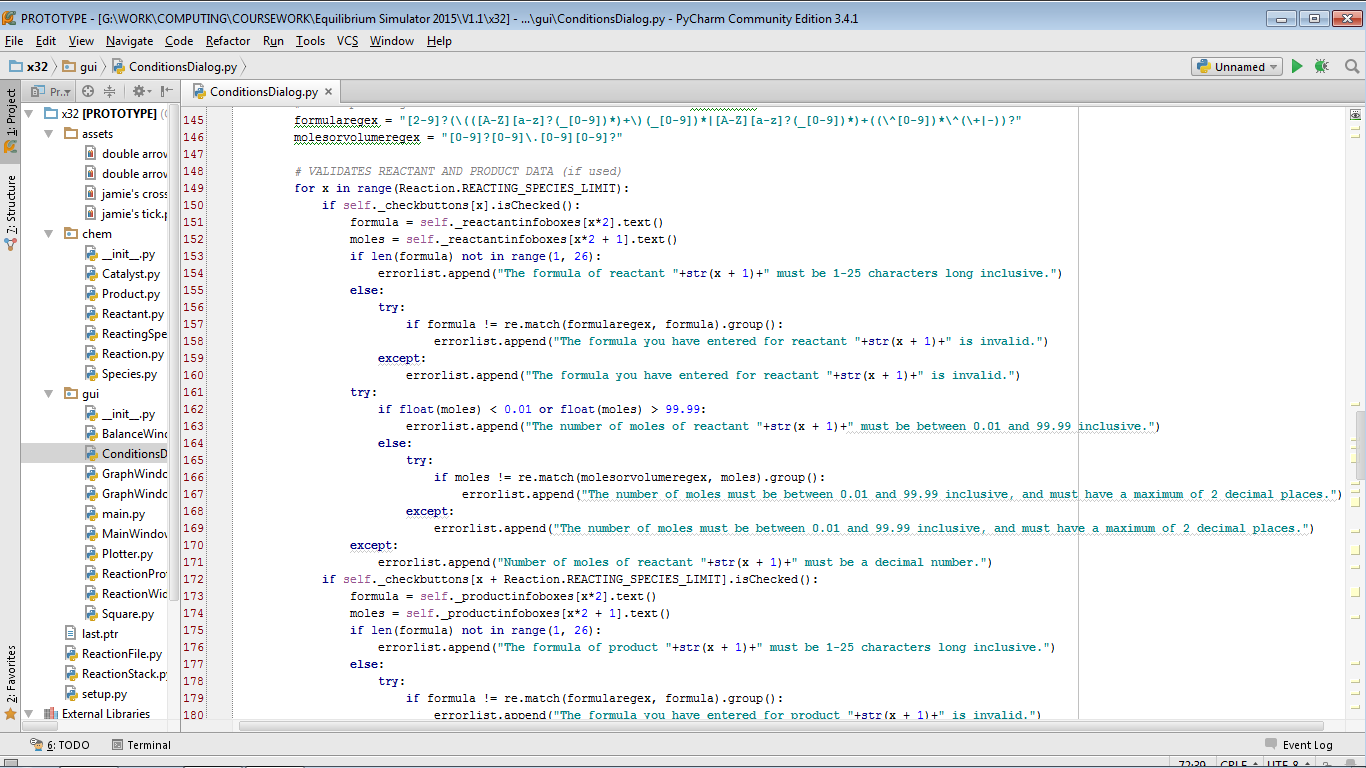
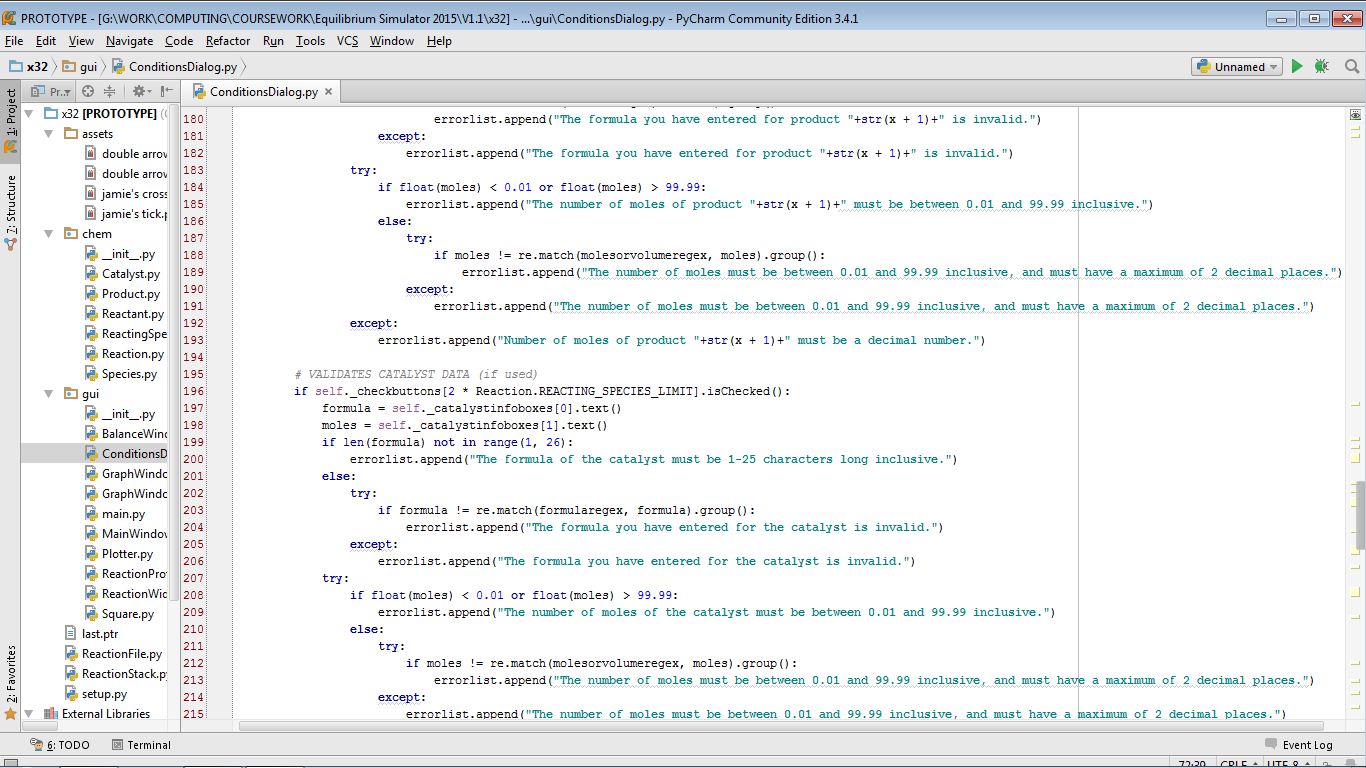
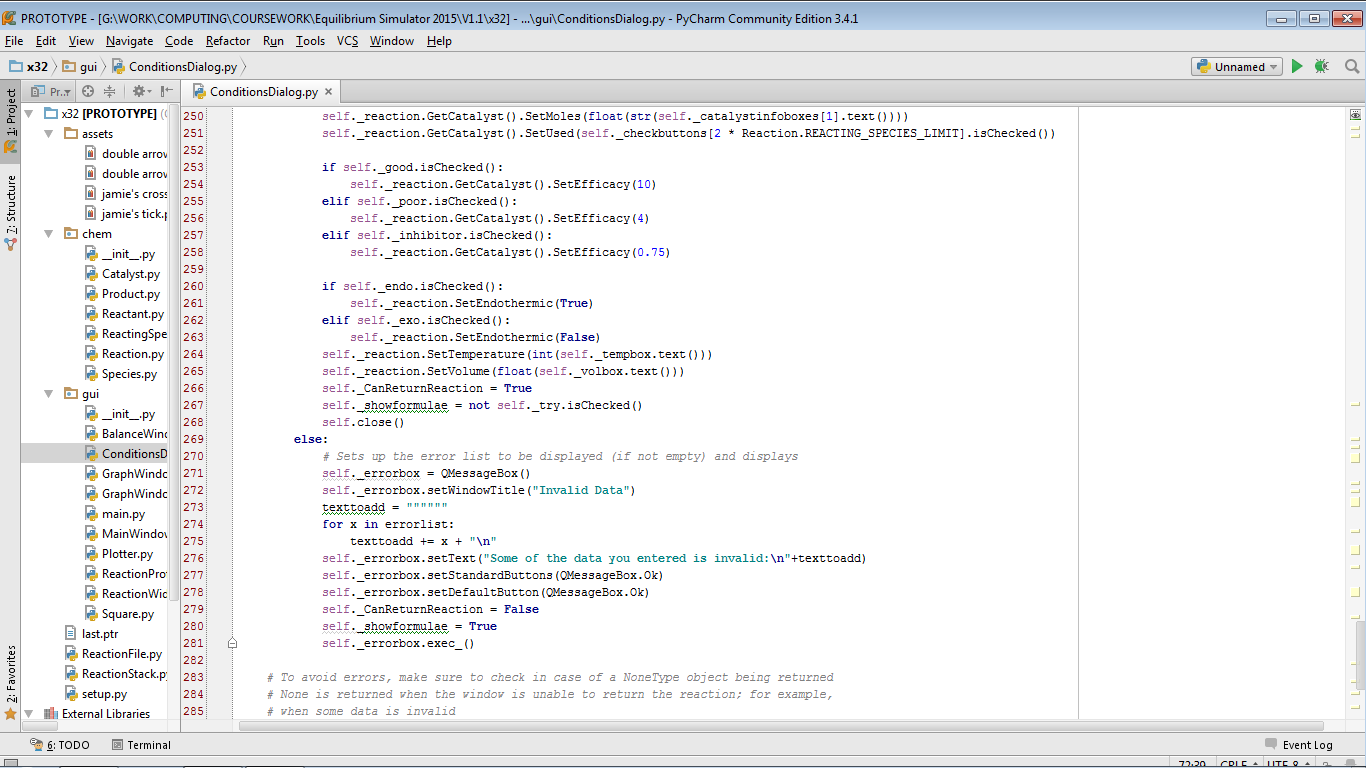
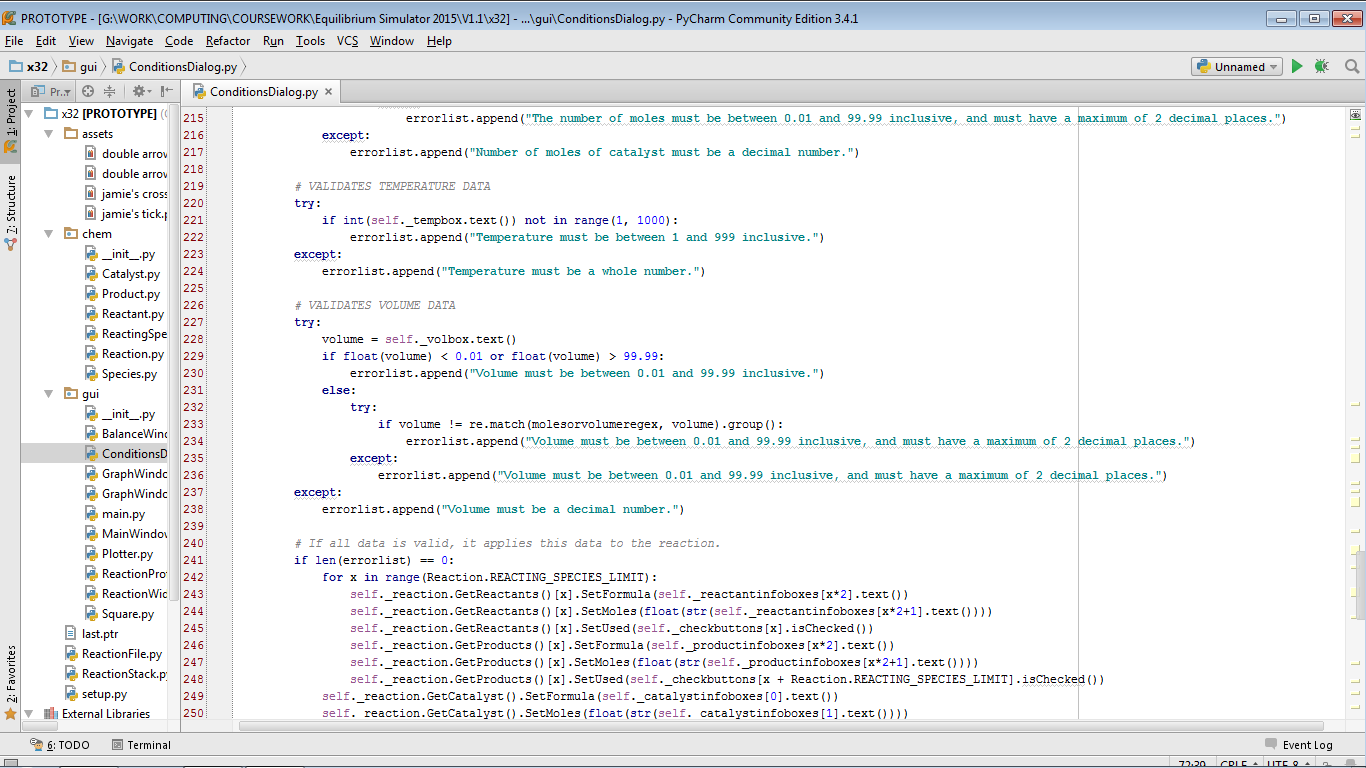
Pseudocode

PROCEDURE ApplyChanges(self):  
 formularegex 🡨 "[2-9]?(\(([A-Z][a-z]?(\_[0-9])\*)+\)(\_[0-9])\*|[A-Z][a-z]?(\_[0-9])\*)+((\^[0-9])\*\^(\+|-))?"  
 molesorvolumeregex 🡨 "[0-9]?[0-9]\.[0-9][0-9]?"  
 FOR x 🡨 0 TO Reaction.REACTING\_SPECIES\_LIMIT DO  
 IF self.\_checkbuttons[x].isChecked() THEN  
 formula 🡨 self.\_reactantinfoboxes[x\*2].text()  
 moles 🡨 self.\_reactantinfoboxes[x\*2 + 1].text()  
 IF formula.length < 1 OR formula.length > 25 THEN

OUTPUT “The formula of reactant "+str(x + 1)+" must be 1-25 characters long inclusive."  
 ELSE  
 TRY  
 IF formula DOES NOT MATCH formularegex THEN  
 OUTPUT “The formula you have entered for reactant "+str(x + 1)+" is invalid."  
 EXCEPT  
 OUTPUT “The formula you have entered for reactant "+str(x + 1)+" is invalid."  
 ENDTRY  
 ENDIF  
 TRY  
 IF float(moles) < 0.01 OR float(moles) > 99.99 THEN  
 OUTPUT "The number of moles of reactant "+str(x + 1)+" must be between 0.01 and 99.99 inclusive.”  
 ELSE  
 TRY  
 IF moles DOES NOT MATCH molesorvolumeregex THEN  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDIF  
 EXCEPT  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places.”  
 ENDTRY  
 ENDIF  
 EXCEPT  
 OUTPUT "Number of moles of reactant "+str(x + 1)+" must be a decimal number."  
 ENDTRY  
 ENDIF  
 IF self.\_checkbuttons[x + Reaction.REACTING\_SPECIES\_LIMIT].isChecked() THEN  
 formula 🡨 self.\_productinfoboxes[x\*2].text()  
 moles 🡨 self.\_productinfoboxes[x\*2 + 1].text()  
 IF formula.length < 1 OR formula.length > 25 THEN  
 OUTPUT "The formula of product "+str(x + 1)+" must be 1-25 characters long inclusive.”  
 ELSE  
 TRY  
 IF formula DOES NOT MATCH formularegex THEN  
 OUTPUT "The formula you have entered for product "+str(x + 1)+" is invalid."  
 EXCEPT  
 OUTPUT "The formula you have entered for product "+str(x + 1)+" is invalid.")  
 ENDTRY  
 TRY  
 IF float(moles) < 0.01 OR float(moles) > 99.99 THEN  
 OUTPUT "The number of moles of product "+str(x + 1)+" must be between 0.01 and 99.99 inclusive.”  
 ELSE  
 TRY  
 IF moles DOES NOT MATCH molesorvolumeregex THEN  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 EXCEPT  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDTRY  
 ENDIF  
 EXCEPT  
 OUTPUT "Number of moles of product "+str(x + 1)+" must be a decimal number."  
 ENDTRY  
 ENDIF  
 ENDFOR   
 IF self.\_checkbuttons[2 \* Reaction.REACTING\_SPECIES\_LIMIT].isChecked() THEN  
 formula 🡨 self.\_catalystinfoboxes[0].text()  
 moles 🡨 self.\_catalystinfoboxes[1].text()  
 IF formula.length < 1 OR formula.length > 25 THEN  
 OUTPUT "The formula of the catalyst must be 1-25 characters long inclusive."  
 ELSE  
 TRY  
 IF formula DOES NOT MATCH formularegex THEN  
 OUTPUT "The formula you have entered for the catalyst is invalid."  
 ENDIF  
 EXCEPT  
 OUTPUT "The formula you have entered for the catalyst is invalid."  
 ENDTRY  
 ENDIF  
 TRY  
 IF float(moles) < 0.01 OR float(moles) > 99.99 THEN  
 OUTPUT "The number of moles of the catalyst must be between 0.01 and 99.99 inclusive."  
 ELSE  
 TRY  
 IF moles DOES NOT MATCH molesorvolumeregex THEN  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDIF  
 EXCEPT  
 OUTPUT "The number of moles must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDTRY  
 ENDIF  
 EXCEPT  
 OUTPUT "Number of moles of catalyst must be a decimal number."  
 ENDTRY  
 ENDIF  
 TRY  
 IF int(self.\_tempbox.text()) < 1 OR int(self.\_tempbox.text()) > 999 THEN  
 OUTPUT "Temperature must be between 1 and 999 inclusive."  
 ENDIF  
 EXCEPT  
 OUTPUT "Temperature must be a whole number."  
 ENDTRY  
 TRY  
 volume 🡨 self.\_volbox.text()  
 IF float(volume) < 0.01 OR float(volume) > 99.99 THEN  
 OUTPUT "Volume must be between 0.01 and 99.99 inclusive."  
 ELSE  
 TRY  
 IF volume DOES NOT MATCH molesorvolumeregex THEN  
 OUTPUT "Volume must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDIF  
 EXCEPT  
 OUTPUT “Volume must be between 0.01 and 99.99 inclusive, and must have a maximum of 2 decimal places."  
 ENDTRY  
 ENDIF  
 EXCEPT  
 OUTPUT "Volume must be a decimal number."  
 ENDTRY  
END

If all data is valid it then sets all fields in the Reaction object to the values entered.

Program code

Animation

The StartOrContinueAnimation function is in the ReactionProfile class and uses its properties. Depending on the ReactionProfile object’s time value this algorithm does three different things. If time is -1, the grid of squares and the eqmpoint value are reset. If time is 0, the value of eqmpoint (the time value when the chance of a square changing colour is equal for either colour) is calculated. If time is more than -1, until time reaches 1500 (at which point it resets to -1), the Draw function of each square is called each frame, which may trigger a colour change depending on a chance value (also calculated each frame). The StartOrContinueAnimation, Reset, Draw and flip methods are all shown here (the last three methods are from the Square class).

Pseudocode

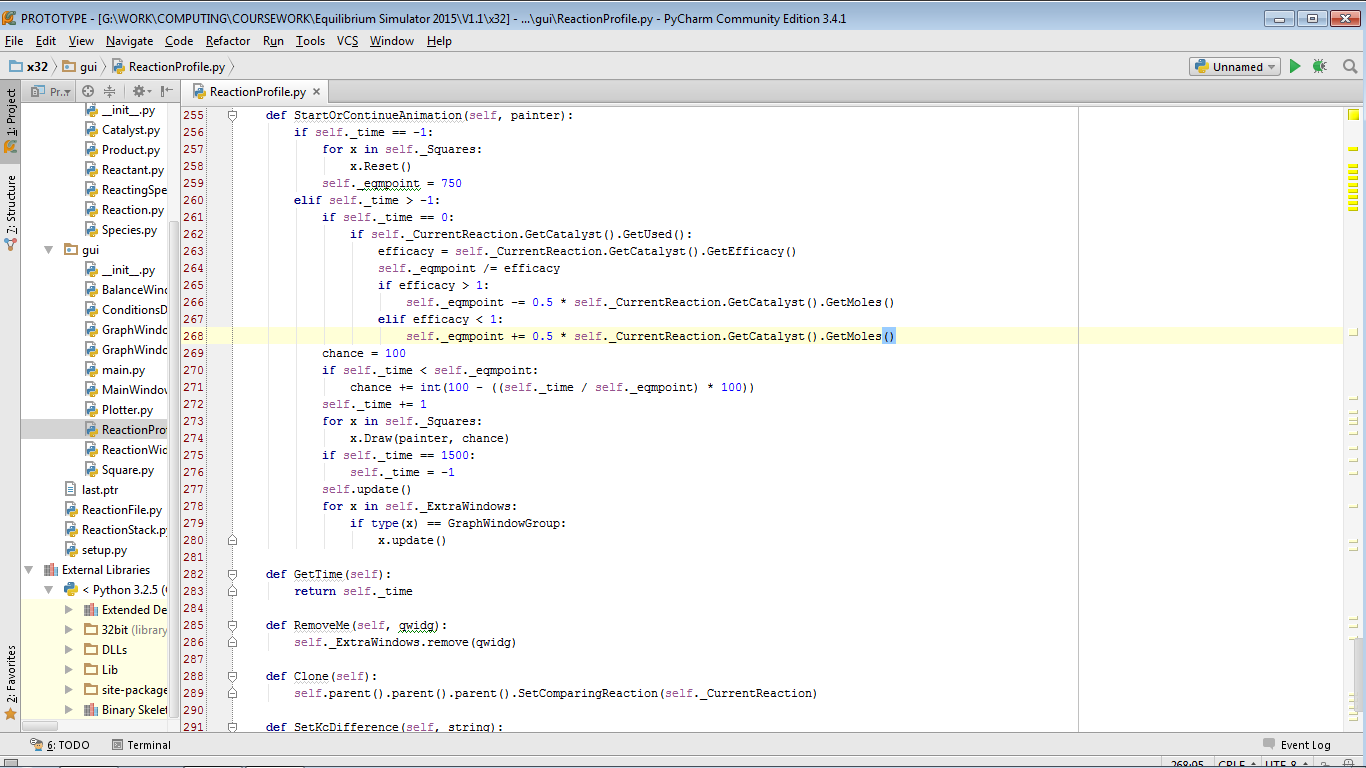
PROCEDURE StartOrContinueAnimation(self, painter : Plotter)  
 IF self.\_time == 1 THEN  
 FOR x 🡨 0 TO 99 DO  
 self.\_Squares[x].Reset()  
 ENDFOR  
 self.\_eqmpoint 🡨 750  
 ELSE  
 IF self.\_time > -1 THEN  
 IF self.\_time == 0 THEN  
 IF self.\_CurrentReaction.GetCatalyst().GetUsed() THEN  
 efficacy 🡨 self.\_CurrentReaction.GetCatalyst().GetEfficacy()  
 self.\_eqmpoint 🡨 self.\_eqmpoint / efficacy  
 IF efficacy > 1 THEN  
 self.\_eqmpoint 🡨 self.\_eqmpoint – 0.5 \* self.\_CurrentReaction.GetCatalyst().GetMoles()  
 ELSE  
 IF efficacy < 1 THEN  
 self.\_eqmpoint 🡨 self.\_eqmpoint + 0.5 \* self.\_CurrentReaction.GetCatalyst().GetMoles()  
 ENDIF  
 ENDIF  
 ENDIF  
 ENDIF  
 chance 🡨 100  
 IF self.\_time < self.\_eqmpoint THEN  
 chance 🡨 chance + int(100-((self.\_time / self.\_eqmpoint) \* 100))  
 ENDIF  
 self.\_time 🡨 self.\_time + 1  
 FOR x 🡨 0 TO 99 DO  
 self.\_Squares[x].Draw(painter, chance)  
 ENDFOR  
 IF self.\_time == 1500 THEN  
 self.\_time 🡨 -1  
 ENDIF  
 self.update()  
 FOR x 🡨 0 TO self.\_ExtraWindows.length - 1 DO  
 IF self.\_ExtraWindows[x] IS OF TYPE GraphWindowGroup THEN  
 self.\_ExtraWindows[x].update()  
 ENDFOR  
 ENDIF  
 ENDIF  
END

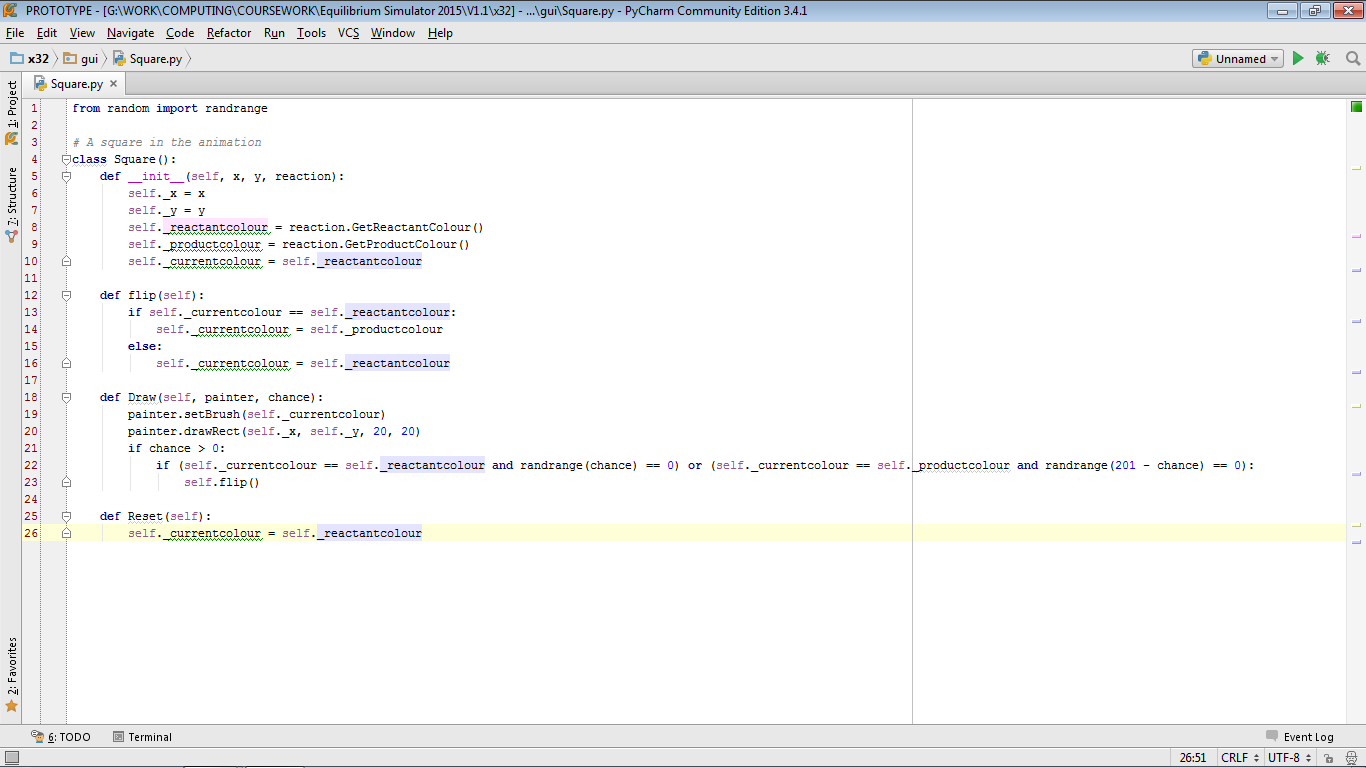
PROCEDURE Draw(self, painter : Plotter, chance : int)  
 painter.setBrush(self.\_currentcolour)  
 painter.drawRect(self.\_x, self.\_y, 20, 20)  
 IF chance > 0 THEN  
 IF (self.CurrentColour == self.ReactantColour AND randrange(chance) == 0) OR (self.CurrentColour == self.ProductColour AND randrange(201 - chance) == 0) THEN  
 self.flip()  
 ENDIF  
 ENDIF  
END

PROCEDURE flip(self)  
 IF self.\_currentcolour == self.\_reactantcolour THEN  
 self.\_currentcolour 🡨 self.\_productcolour  
 ELSE  
 self.\_currentcolour 🡨 self.\_reactantcolour  
 ENDIF  
END

PROCEDURE Reset(self)  
 self.\_currentcolour 🡨 self.\_reactantcolour

Program code





Using Undo and Redo

When “Undo” is pressed the reaction is reverted to before it was last changed. The “Redo” button cancels the last undo. Editing a reaction will clear the stack used for redo operations and changing reactions or files will clear both stacks. The following methods show how the stacks are used and implemented in the program: EditReaction in ReactionWidget, undo, redo, clearstacks and pushtoundo in MainWindow, and push and pop in ReactionStack.

Pseudocode

PROCEDURE EditReaction(self)  
 currentreaction 🡨 self.CurrentProfile.GetReaction()  
 dialog 🡨 ConditionsDialog(currentreaction)  
 dialog.show()  
 reaction 🡨 dialog.GetReaction()  
 IF reaction NOT NULL THEN  
 self.parent.pushtoundo(currentreaction)  
 self.CurrentProfile.SetReaction(reaction)  
 self.parent.clearstacks(2)  
 IF NOT dialog.GetCanShowFormulae() THEN  
 dialog.hide()  
 self.parent.hide()  
 BalanceWindow(reaction, self.parent()).exec\_()  
 ENDIF  
 ENDIF  
END

PROCEDURE undo(self)  
 IF self.\_undostack.length > 0 THEN  
 self.\_redostack.push(self.\_ReactionsWindow. GetCurrentReaction())  
 self.\_ReactionsWindow.SetCurrentReaction( self.\_undostack.pop())  
 ENDIF  
END

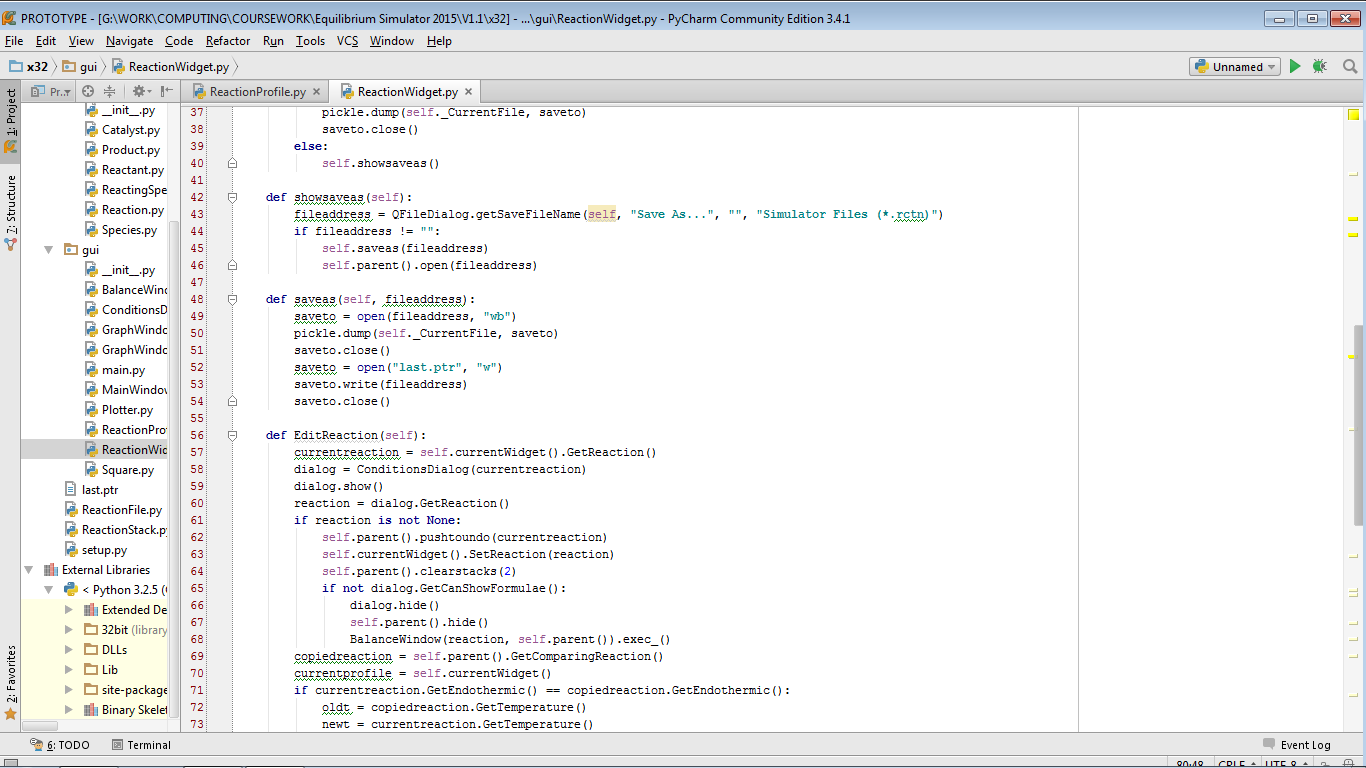
PROCEDURE redo(self)  
 IF self.\_redostack.length > 0 THEN  
 self.\_undostack.push(self.\_ReactionsWindow. GetCurrentReaction())  
 self.\_ReactionsWindow.SetCurrentReaction( self.\_redostack.pop())  
 ENDIF  
END  
  
PROCEDURE clearstacks(self, stack : int)  
 IF stack == 1 THEN  
 self.\_undostack.clear()  
 ELSE  
 IF stack == 2 THEN  
 self.\_redostack.clear()  
 ELSE  
 IF stack == 3 THEN  
 self.\_undostack.clear()  
 self.\_redostack.clear()  
 ENDIF  
 ENDIF  
 ENDIF  
END

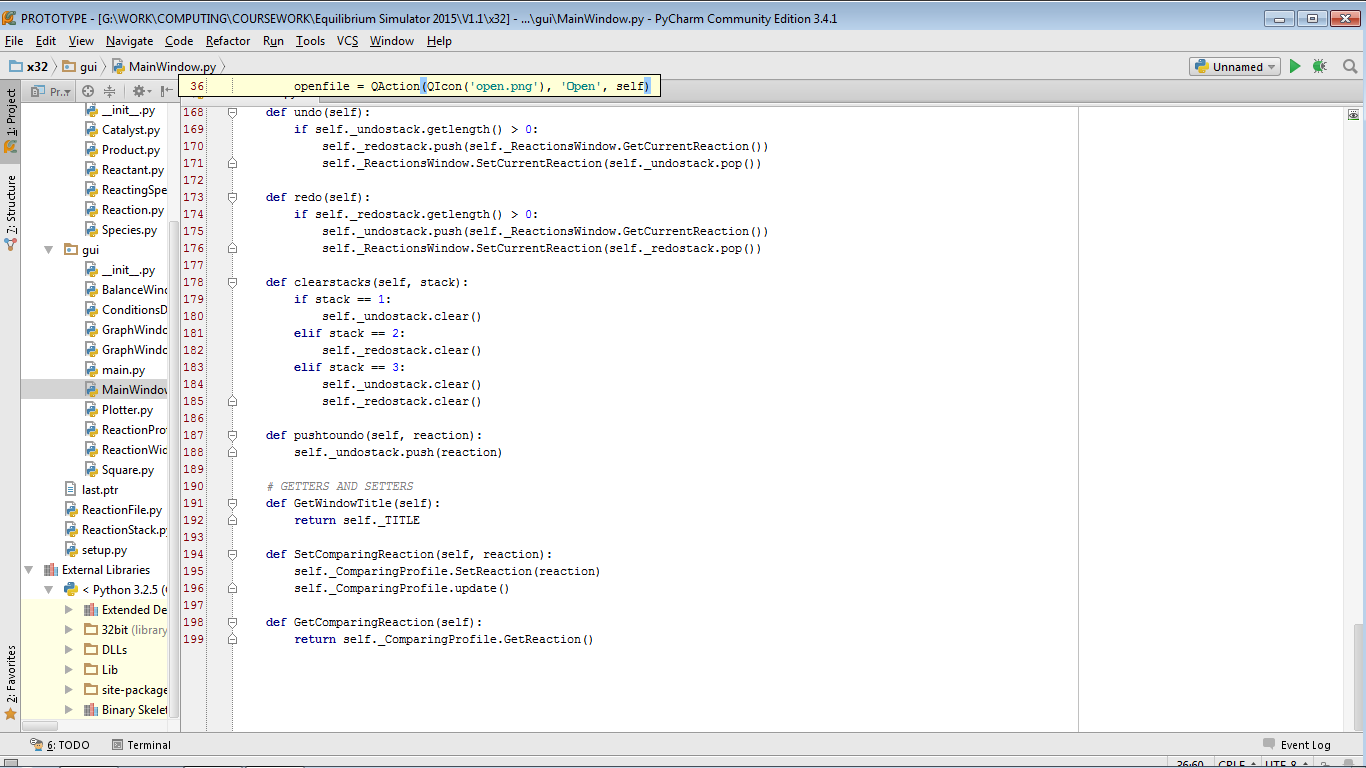
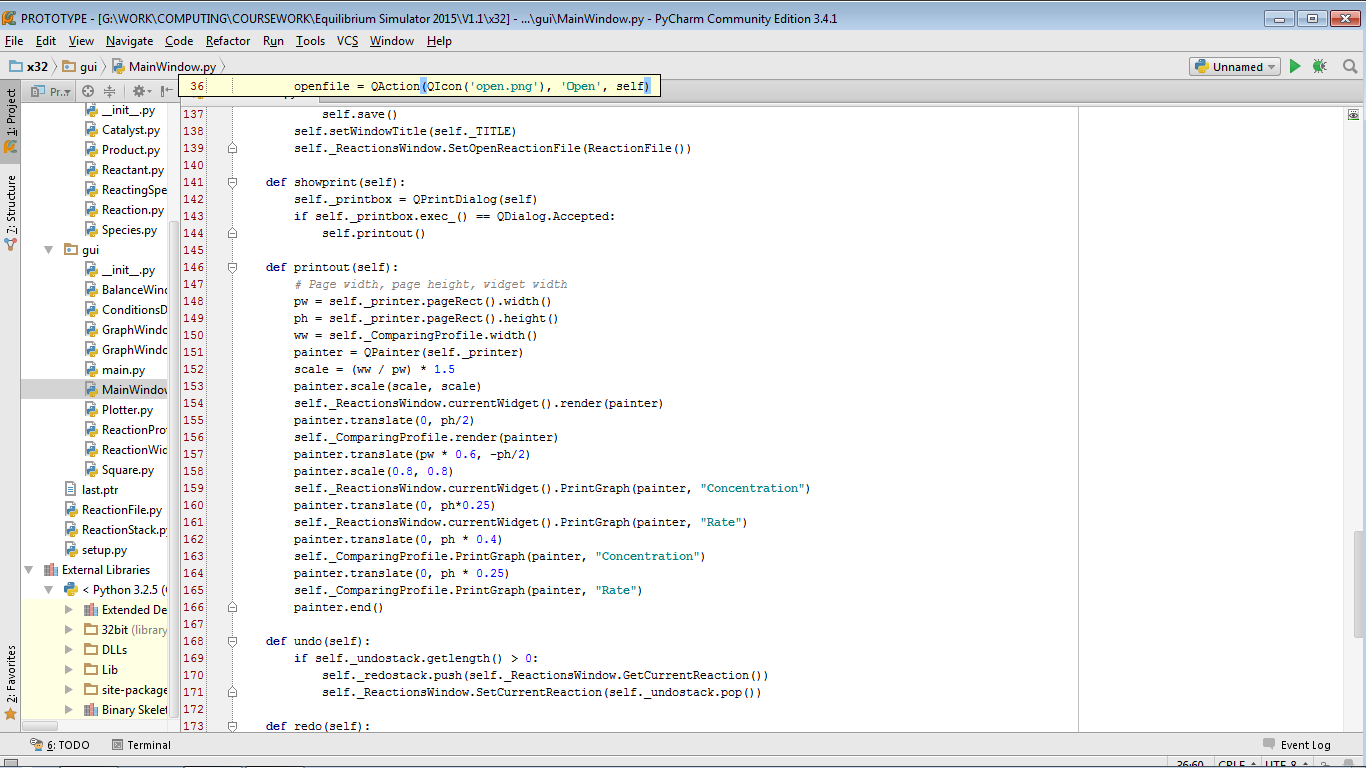
PROCEDURE pushtoundo(self, reaction : Reaction)  
 self.\_undostack.push(reaction)  
END

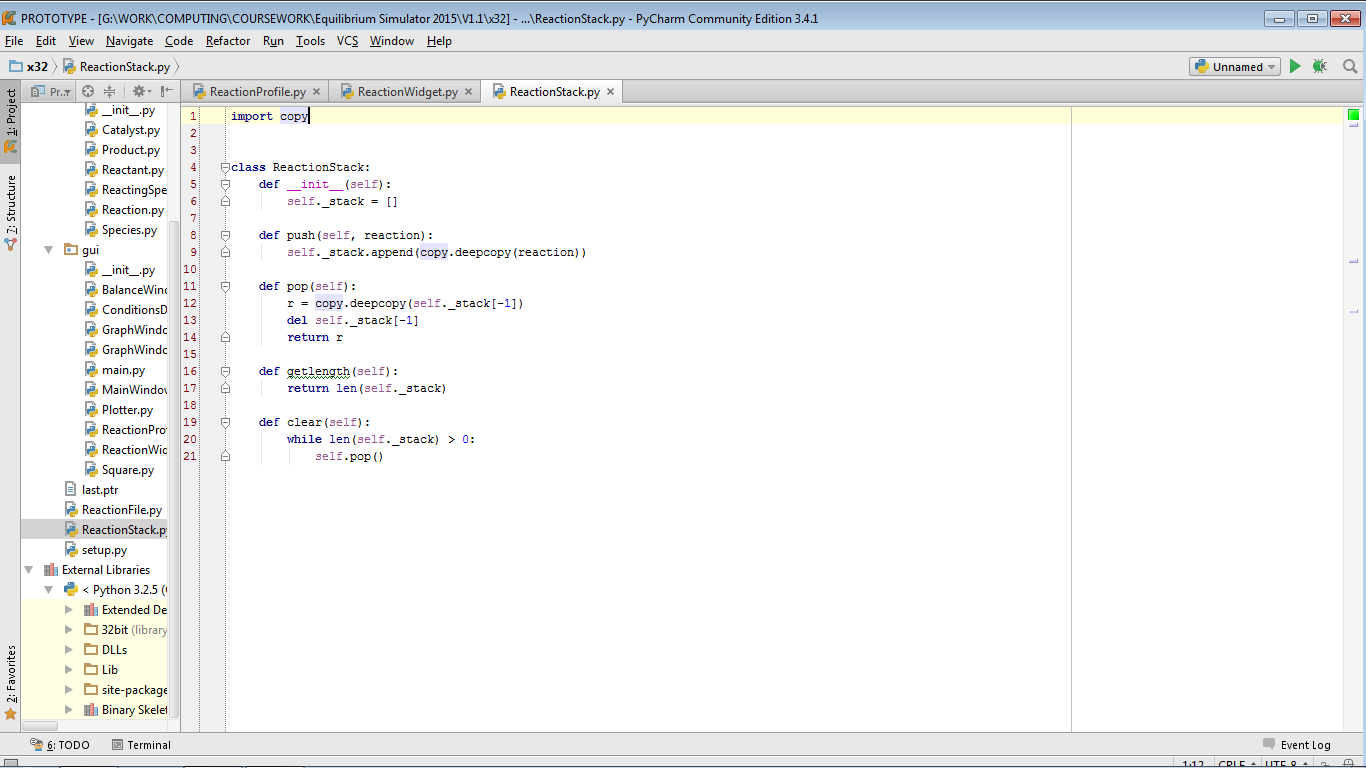
PROCEDURE push(self, reaction : Reaction)  
 self.\_stack.append(copy(reaction))  
END

FUNCTION pop(self)  
 r 🡨 copy(self.\_stack[self.\_stack.length – 1]  
 DELETE self.\_stack[self.\_stack.length – 1]  
 RETURN r  
END

Program code







4: Full Program Listings

Listings are available in Item 11 of the Appendix.