2017

An Object-Oriented Simulator for Flash Processes in Chemical Engineering

GROUP 1

JESSE WARD-BOND, XINLONG CHEN, JOSEPH NAHAS, JEFFREY BYLES

EXECUTIVE SUMMARY

Flash separation is an essential unit operation in chemical engineering. Flashing normally occurs when a stream undergoes a large pressure drop and part of the stream flashes from liquid to vapour phase. Therefore, this flashing separation can be described as a specific process or termed as an object-oriented process in terms of Java language. Java has been well known as an object-oriented computer program that can be used to solve complex problems such as the flash separation process. This project involved the development of a Java-based program simulator to investigate flashing processes.

The simulator was designed to solve three flash systems depending on information given. In each case, the feed composition and operating pressure were known. The isothermal temperature, the adiabatic feed temperature, or the adiabatic flash temperature were known. The simulator was tasked with determining outlet compositions and flow rates, as well as either the heat added/removed from the system, the adiabatic flash temperature, or the adiabatic feed temperature.

The developed simulator was designed to be able to handle both ideal and non-ideal cases to calculate the vapour-liquid equilibrium data (*K* values). For non-ideal cases, Peng-Robinson and Soave-Redlich-Kwong thermodynamic models were both used. Bubble and dew point calculations were performed to confirm that flash separation was possible for each simulation. The computation used numerous root-finding techniques such as Ridder's method. The simulator allowed the input of information from both the keyboard or a text file under a specific layout (CSV). This simulator program also printed the results to the user's interactions pane as well as a text file output.

Validation of the simulator was performed using a spreadsheet created in Microsoft Excel, as well as commercial engineering software (UniSim). The program's robustness was also tested through the input of many dubious values. In addition, an anonymous flash tank simulator was also assessed and validated under the same conditions.

The main limitations introduced by programming errors are in the bounds of the temperature iteration process. The assumption that nitrogen and ethanol were both incompressible fluids for all cases is not a fully realistic assumption. There are a few hard coding errors of boundary conditions and a few arithmetic errors (coding wise). Due to these errors, the simulator was not always able to correctly predict flash separation of certain operating conditions when compared to external sources. However, the simulator was much more robust and accurate than the anonymous simulator. For future improvement, proper use of exception handling would greatly increase the robustness of the code. In addition, the use of GUIs could improve user friendliness. A windowed interface with textboxes and buttons would provide a much better experience than keyboard console entry.

1 TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

2 INTRODUCTION

Java is a computer programming language originally developed in 1991 by James Gosling of Sun Microsystems[1]. Unlike procedural programming languages such as C and Pascal, Java is termed an object-oriented programming language[2]. This means that programs can be broken down into objects, which act in a similar manner to real-life objects. In Java, objects have their own methods, which are used to perform certain tasks. Just like a reactor is an object, a Java object called `reactor` could have the same characteristics as a reactor. Objects themselves are instances of a class, which is similar to a blueprint; classes provide the instructions in making an object and it defines what constitutes an object. This allows the creation of multiple, distinct objects that are instances of the same class. For example, in a chemical system there may be more than one reactor, and while both would be reactors, each one is a unique entity.

Chemical processes are often composed of many complex interactions. For example, they consist of chemical species being processed, streams containing their own mixtures of species, reactors converting certain species to other species, separation vessels or systems acting to purify mixtures, heat exchangers working to preserve a constant operating temperature, and many other possible components. Such complex systems require intricate relationships to accurately model processes being studied. Trying to solve a chemical system simultaneously in one-step can be both foolish and impossible. Often such chemical systems must be broken down into several components in order to be properly assessed and solved. The advantage of object-oriented design allows one to break down an initially complex system into much smaller, more manageable parts. With the chemical system as an example, each stream can be broken down into each chemical species. Each species can then be defined by its thermodynamic and physical properties, and be characterized by its enthalpy and heat capacity. This allows each species object to be compartmentalized for easy access once its information is needed for a particular process or calculation.

A proper chemical system simulation should be robust and generic. With such a simulator, different simulations can be easily analyzed and compared. A simple simulator that did not employ object-oriented design could easily solve a single system, but rarely is one chemical system the only one considered. With the use of object-oriented design, each object can be constructed as necessary, allowing the creation of multiple chemical systems from one simulator. Since classes are designed to be generic, this also allows another user to create their own objects that may be specific to their system. Such robust systems simply are not easily possible without the use of object-oriented design.

The proceeding simulation employed the use of object-oriented design using Java. The system to be solved is a simple flash tank operation consisting of a single inlet stream and two outlet streams, one vapour and one liquid stream. The simulator will look at three possible cases and solve the system with respect to the outlet streams. The simulator is designed to be both generic and robust, and will deconstruct a complicated chemical system into smaller, simpler parts.

3 BACKGROUND

Flash separation is a common unit operation in chemical processes. Flashing occurs when a stream undergoes a pressure drop and part of the stream "flashes" into the vapour phase.[3] This may be contrasted to boiling, or distillation where the temperature is increased and a part of the stream evaporates into the vapour phase.

Flashing can be used for species separation; pressure reduction, boiler blowdown heat recovery, and venture condensate return steam traps. Understanding the results of a flash process (or dictating a result from a variable input) can be a complex matter but is key for designing systems that rely on the flashing phenomena. At low temperatures and pressures with ideal species, the procedure is simpler than when critical behaviours, multispecies interactions and azeotrope behaviours must be considered.

When considering a flash system there may be many questions regarding the system, including the following:

1. How does the feed temperature affect flashing process and resulting separation? This may be useful to know when a stream temperature is determined upstream by another unit operation and the effects on the resulting liquid and vapour streams are of interest. An example of this situation would be a continuous blow vapour stream being used to pre-heat a glycol circuit, where the boiler has a varying temperature and pressure feed. It would be of interest to know if the variation would ever cause inadequate heating and require a backup steam heat exchanger. In addition, if so the design and usage frequency to reach set points.

2. If tank is flashing at a given temperature and pressure how much power will be required to operate the tank? This may be asked when the products for each phase are more of a concern. This information can be used to do a separation economic analysis, and to design resulting heat exchangers. The resulting heat exchangers may be used inside the tank or before the feed stream is fed into the reactor. For example, if a separation process is designed to concentrate product in one of the phases – assuming VLE data is present – the operating conditions inside the flash separation can be controlled by either heating feed so tank conditions are met, or heating the inside of the tank so they are met.

There are obviously several other questions that could be asked that would result in a system with an appropriate amount of degrees of freedom to solve. The value in understanding these questions is so that an engineer designing a simulation can understand the degree of robustness it requires.

This report presents a flash separation simulator designed in java to robustly solve three different types of flash separation problems: an isothermal flash separation (Case One), an adiabatic flash at an unknown temperature (Case Two), and an adiabatic flash with an unknown feed temperature (Case Three). In all cases, the flash and feed pressures are known or user-input, and the component flows in the feed stream are defined.

The solution algorithm for each of the three cases in this report are similar. A mass balance is performed at a known or guessed temperature, and then an energy balance is performed to determine the

unknown temperatures/heat flows. The differences between each case arise from the extent to which the mass and energy balances must be solved simultaneously. In Case One and Case Three, the mass and energy balances can be performed independently – solving for the heat flow into the reactor or the temperature of the feed stream respectively. In Case Two, the mass balance must be solved as iterations through temperature are also performed. For the simulator presented in this report, a flash separation problem is considered "solved" when the liquid and vapour flows of each component, the temperature and pressure of every stream, and the heat flow into the reactor are known.

The mass balance is generally solved using a root-finding numerical method on the objective Rashford-Rice equation (**Equation 1**). Combination of the V/F terms (where V is the vapour flow rate and F is the feed stream flow rate) into the single ψ term increases the stability of the iterations. Remaining mass balance parameters can be calculated easily from component mass balances (**Equation 2**).

$$\sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} \frac{z_i(K_i - 1)}{1 + \psi(K_i - 1)} = 0 \quad \textbf{(1)}$$

$$z_i = x_i \mathcal{L} + y_i \mathcal{V} \quad \textbf{(2)}$$

In the above equation, K_i represents the vapour-liquid partitioning of species *i*. Determination of these values are a necessary first step to solving the Rashford-Rice equation. The exact calculation process depends on the ideality of the system. For instance, in an ideal solution with no mixing enthalpies, the relationship is easily described by Raoult's law (**Equation 3a**). This ideal assumption is not valid in most cases, which calls for the use of non-ideal equations of state (EOS), like Peng-Robinson or Soave-Redlich-Kwong. Broadly, non-ideal K_i values are functions of temperature, pressure, and whatever parameters the VLE model in use requires (**Equation 3b**). In this report we use the Peng-Robinson equation of state to predict fugacity coefficients which can be used to calculate non-ideal K_i values (**Equation 3c**). The intermediate steps are quite lengthy, and are shown in the appendix.

$$K = \frac{y}{x} = \frac{P_i^{sat}}{P} \quad \textbf{(3a)}$$

$$K_i = f(VLE\ Model, T, P) \quad \textbf{(3b)}$$

$$K_i = \frac{\phi_i^L}{\phi_i^V} \quad \textbf{(3c)}$$

In Case One, when the flash is isothermal, the energy balance is used to solve for the heat flow in to the reactor (**Equation 4**). In the remaining cases, adiabatic (Q=0) temperatures are solved from the enthalpy balance. Stream enthalpies are calculated by integrating through heat capacity correlations for the liquid and gas phases.[3,4] These correlations are complex polynomial functions of temperature (Equation 5a,b), so it is easiest to use numerical methods to solve for the unknown temperature. The calculation of stream enthalpies depends on the phase and reference state of each component: In the feed stream F, all components

are assumed to be a liquid unless specified as non-condensable. In the feed stream V, all components are vapour etc.

$$Q = \Delta H_L + \Delta H_V - \Delta H_F \tag{5}$$

$$\int_{T_1}^{T_2} c_{p,liq} = C_1\left(T - T_{ref}\right) + C_2\frac{\left(T^2 - T_{ref}^2\right)}{2} + C_3\frac{\left(T^3 - T_{ref}^3\right)}{3} + C_4\frac{\left(T^4 - T_{ref}^4\right)}{4}C_5\frac{\left(T^5 - T_{ref}^5\right)}{5} \tag{6a}$$

$$\int_{T_1}^{T_2} c_{p,gas} = C_1(T_2 - T_1) + \frac{C_2(T_2^2 - T_1^2)}{2} + \frac{C_3(T_2^3 - T_1^3)}{3} - C_4\left(\frac{1}{T_2} - \frac{1}{T_1}\right) \tag{6b}$$

3.1 RIDDER'S METHOD

In this report, Ridder's method is used as the numerical root-finding method. This method calculates a root (x_R) between upper(x_U) and lower bounds (x_L) (**Equation 7**). The next upper and lower bounds picked based off the new location of the root. Other root-finding methods were tested, but ultimately Ridder's method was chosen for reasons outlined in (**Section 4** – Design and Discussion).

$$x_R = x_M + (x_M - x_L)\frac{sgn[f(x_L) - f(x_U)]f(x_M)}{\sqrt{f^2(x_M) - f(x_L)f(x_U)}} \tag{7}$$

4 DESIGN AND DISCUSSION

The design will be presented in the order in which the classes/objects are implemented. This approach follows the UML mostly from the bottom, up (**Figure 1**). The object-oriented paradigm was followed rigorously, with the code being separated into the most logical physical components of a flash tank. The flash tank is an object containing three stream objects (one inlet two outlets), and each stream is composed of an array of species objects. The calculations performed on the flash tank are separated into case one, two and three classes, and the VLE models used for each series of calculations can be objects of either Ideal, Sloave-Reidlich-Kwong or Peng-Robinson classes.

4.1 IMPORTING PHYSICAL DATA

The simulator is run and controlled by the static main method in the `Group1FlashTankSimulator` class. This class also manages all user-input and file IO operations. The first step in the main method is to import the necessary physical property data for every species to perform all subsequent calculations. The data is stored in a .CSV file and is imported by creating a `ChemicalPropertiesTable` object, which is a child of the DataTableIO class. The DataTableIO class is simply a generic table manipulating class: It can import any CSV as a 2-dimensional table of strings. Upon construction with the empty constructor, it defaults to importing a table called "DataSIUnits.csv". If the "DataSIUnits.csv" table can't be found, the `FileNotFoundException` error is caught and a dialogue box is opened, prompting the user to select the .CSV file they want to import. `Scanner` objects scan the table to first determine its maximum dimensions, and then to import the table. Because the position of `Scanner` within a file can't be reset, it is necessary to use two separate `Scanner` objects. The imported .CSV is stored as a `String [] []` instance variable.

Being a generic table-importer, `DataTableIO` has methods to perform various table operations like printing the table to the console (`printable ()`), appending the table with a new row of the same length (`appendTable()`), and taking single rows and columns from the table (`extractRow()`, `extractColumn()`). The subclass `ChemicalPropertiesTable` adds two methods to DataTableIO: One that extracts the two topmost rows of the table to get the column names and units (`extractHeadings()`) and one that extracts the two leftmost columns of the table to get the ID numbers and names of every chemical species in the table (`extractIdentities()`). To make the outputs more readable and easier to display on the console, they are returned as Nx2 vertical arrays.

4.2 THE SPECIES

`Group1FlashTankSimulator` creates the relevant species based on user-input and stores them in a `Species[]` arrays. `Species` is an abstract class with two child classes representing the different sates the species can be in for our simulation: Vapour`Species` and `LiquidSpecies`. A `String[]` corresponding to a row from the table stored in `ChemicalPropertiesTable` is required in order for objects of these types to be constructed. Ultimately there are 26 relevant physical parameters – all of them

instance variables of a `Species` object – that are parsed from the this `String[]` parameter in the `Species` super-class. `Species` has three methods, one of which is abstract public methods that are consistent for all species, regardless of whether they are a `VapourSpecies` or a `LiquidSpecies`. The first concrete method is `calcVapourPressure()`, which takes in a temperature and uses the Antoine's constants for a given species to calculate and return the vapour pressure of that species, this is mainly used for the ideal vapour-liquid equilibrium calculations. The second concrete method `integrateLiqHeatCapacity()`, takes in a start and end temperature and calculates the integral of the liquid heat capacity between those temperatures (see Appendix, **sec. 10.1**). Eventual liquid and gas-phase enthalpy calculations both involve this integral (albeit with different bounds) so it is necessary that both `VapourSpecies` and `LiquidSpecies` inherit this method.

The abstract method is `calcEnthalpy()`. This method is the only functional difference of `VapourSpecies` and `LiquidSpecies`. In both cases a Δh (J/mol) is calculated between that species and the reference state of liquid at 273.15 K. In a `LiquidSpecies` object this calculation simply returns the result of `integrateLiqHeatCapacity` or 0 for non-condensable species. In `VapourSpecies` this method must take into account the latent heat of vapourization (λ). Because this property is only known at the boiling temperature, it is necessary to alter the path of the enthalpy integral, which won't affect the final results because enthalpy is a state function. Non-condensable `VapourSpecies` use gas at 273.15 K as their reference state. Changing reference states between species won't affect the final enthalpy balance as long as the individual reference states remain the same throughout the calculations.

4.3 STREAMS

In our simulator, species are grouped together into streams through a `Stream` class with a `Species[]` instance variable. This array of an abstract type means that the streams can be composed of any of the child classes: `LiquidSpecies`, `VapourSpecies`, or a mixture of both types of `Species`. In our case, we know that the flash tank feed stream is 100% liquid except when there are non-condensables, so it is useful to be able to create streams that can store both kinds of `Species`. `Stream` objects are also defined by arrays of molar flows and mole fractions, a total molar flow, a temperature, and a pressure. The only unique method in a `Stream` object is the `totalEnthalpy()`. This method calculates the total enthalpy flow in the stream as a mole-fraction weighted sum and then multiplies it by the total molar flow rate of the stream to get an energy flow in J/s (see Appendix, **sec. 10.1**). The getters and setters in the `Stream` class are arranged such that inter-dependant properties (i.e. total molar flow and mole fractions on molar flows) are recalculated any time that one of the instance variables is changed.

4.4 THE FLASH TANK

Central to our object-oriented simulator is the `FlashTank` class. Objects of this type are defined by a `HeatExchanger` object, three `Stream` objects (feed, vapour outlet, liquid outlet), an operating pressure, and an operating temperature. The vapour outlet is a `Stream` whose `Species[]` is entirely made up of `VapourSpecies` in the same order as the similar array in the feed stream. Likewise, the liquid outlet stream is entirely `LiquidSpecies`. The feed stream is generally composed of `LiquidSpecies`, with the

exception being the presence of non-condensable components. The exact number and order of the species are conserved throughout the streams. Even non-condensables will be present in the liquid outlet `Species[]` array, albeit with negligible mole fractions and molar flows. The `HeatExchanger` class is very simple, and doesn't warrant its own section: `HeatExchanger` objects simply have an instance variable Q in J/s that represents the heat flowing into or out of the `FlashTank` object.

A `FlashTank` object does not have any unique methods besides the typical getters and setters. However, because a `FlashTank` contains `Stream` objects which contain `Species` objects and because deep copying is always performed when passing objects, it was necessary to implement "deep setters", mutator methods in the `FlashTank` class. These are mutators that can set any relevant values of in the deeper classes. For instance `setFeedStreamOutletTemperature()` calls the feed stream method `setTemperature()`. The methods `setFlashTemp()` in `FlashTank` sets the operating temperature of the flash tank. During this operation it is also necessary that the temperatures of the outlet streams be changed to match the operating temperature. This could have been done with `setPressure()` as well, but this method was never needed, as in every case that the simulator solves, the operating pressure and thus the outlet pressures are known and constant.

4.5 SOLVING THE SYSTEM

There are three different frameworks in which the unknown parameters of a `FlashTank` object need to be solved. In *case one* the operating and feed conditions are completely specified, and the simulator only needs to determine the outlet compositions and the heat flow into the flash tank. In *case two* the feed is still completely defined, but only the operating pressure is known. In this case the flash tank is adiabatic, and the flash temperature as well as outlet compositions must be determined. *Case three* is also adiabatic, but in this case the operating conditions are completely defined while the feed stream temperature is unknown. The complete outlet compositions must be solved as well as the feed temperature. `CaseOne`, `CaseTwo`, and `CaseThree` are all subclasses of the `FlashModel` class, and share the same method `solveSystem()` (abstract in `FlashModel`) which solves the unknown parameters of the system according to the case, and returns a completely specified `FlashTank` object (or an error message). Constructing these objects requires a `FlashTank` object and a `VLEModel` object. Logical operations determine which case object to construct based on user input. For instance: the user is prompted to answer whether the system is adiabatic (case two or case three) and then prompted to enter whether they have the feed or flash temperatures. It isn't generally recommended to use class-structures for logic, we felt that this was the best way to keep the code readable and functional.

All `FlashModel` classes have a third instance variable, a `double[] kiArray`. This is because the numerical methods used in the cases are implemented through anonymous classes, which can only access instance variables of the enclosing class. The use of anonymous classes becomes particularly useful in cases two and three, which require multiple implementations of the same interface with difference definitions.

Case One

All `solveSystem()` methods begin by creating default vapour and liquid outlet streams of the flash tank parameter. After the default vapour and liquid `Stream` objects are created for the `FlashTank` object within `CaseOne`, the system is checked to confirm whether or not it is flashable using the method `CaseOne` method `checkFlashable()`. A local copy of the instance variable `FlashTank` object is created, and k_i values are approximated using the Wilson-corrected Soave-Reidlich-Kwong equation (see Appendix, **sec. 10.2.2**). These k_i values are used to calculate the mole fractions in the vapour (bubble point calculations) or in the liquid (dewpoint calculations) under the assumption that the mole fractions in the opposite outlet stream are equal to the mole fractions in the feed stream, excluding non-condensable components. The outlet compositions are subbed back into an equation for calculating k_i values (generally Raoult's law or the Peng-Robinson equation of state), and these new k_i values are used to calculate the outlet compositions once again. When the k_i values no longer change by more than 0.0001 between iterations, the iterative loops are stopped and the appropriate sum is calculated: $\sum x_i k_i$ for dewpoint, or $\sum y_i/k_i$ for bubble point. If either of these sums is strictly less than 1, then the system is not flashable under the conditions the conditions it is presently at and a `NotFlashable` exception is thrown. The direction that the system deviates from the two phase region (bubble pressure or dew pressure side) can be determined simply by looking at which sum is <1. This method works for both ideal and non-ideal cases. Using Raoult's law to calculate ideal VLE leads to k_i-values that converge after one iteration, as the k_i values are not composition-dependant. Generally, if the outlet can't exist in two phases under the operating flash conditions then the Peng-Robinson cubic equation of state will only have one real root, and a NotFlashable exception is thrown from inside the k_i value calculation, which still works in the `checkFlashable()` framework.

When the `checkFlashable()` method is completed without throwing any exceptions, the system can be solved. The first step is to guess the k_i values as mentioned previously. The Rashford-Rice objective function is solved using ridders method to find ratio of vapour molar flow to feed molar flow ratio. Initially the vapour flow rate was found using the Rashford-Rice process to find $\sum y_i$, but convergence of this function within more complicated iterative schemes was messy. The V/F ratio is used to determine all the outlet compositions through various mass balances which are divided into separate private methods to help improve code readability. No parameters are passed into these methods as they all act on the `FlashTank` instance variable. With the outlet compositions known, k_i values are recalculated using the appropriate VLE model and this process is repeated until the k_i values differ by less than 0.0001 between iterations. A simple enthalpy balance sets the heat flow for the `HeatExchanger` object within the `FlashTank` object. The `solveSystem()` method returns the fully solved `FlashTank` to the main method.

Case Two

In case two, the operating temperature of the flash tank is unknown and the system is adiabatic. In order to successfully solve for the operating temperature, Ridder's method must be used to find the temperature that sets the enthalpy balance to 0. Doing this requires that VLE be solved using Ridder's method (as in `CaseOne`) at every temperature that the exterior Ridder's method iterates through. The two

distinct uses of Ridder's method requires an interface be implemented twice in `CaseTwo`, which is not possible in Java. This problem is solved through anonymous classes that implement the necessary interface.

In order to successfully iterate through temperature, Ridder's method must be bounded by temperatures within which the system can be successfully solved. Before the main iterations in `solveSystem()` are performed, a method called `findBounds()` iterates a local `FlashTank` object cloned from the instance variable from 100 to 2000 K by increments of 1 K, and tries `checkFlashable()` at every temperature to approximate the range spanned by the dew and bubble temperatures of the mixture. Once these bounds are determined, the main iterative methods can be performed. Obviously a trial-and-error method to find the temperature bounds is less than ideal, but dew and bubble temperature calculations couldn't be worked into the framework of our code.

In some mixtures, no adiabatic flash temperature can be found. In these instances, although the interior Ridder's method can find a solution to the VLE, the exterior Ridder's method won't be able to find a solution in the interval bound by the bubble and dew temperatures. Without a solution, the exterior Ridder's method throws a `BadFunction` exception, which is caught within `CaseTwo` and turned into a `NotFlashable` exception to caught within the main method. Otherwise the calculation is completed and `solveSystem()` returns (a clone of) the fully solved `FlashTank` object.

Case Three

Case three works almost the same as case one, as the operating temperature and pressures are both known. `checkFlashable()` is called, then the VLE of system is solved with Ridder's method. After that, Ridder's method is used once again to solve for the feed temperature which would make the system adiabatic. The temperature bounds on the enthalpy balance Ridder's method were accidently left hardcoded, but ideally would be found the same way as they are within the `solveSystem()` in `CaseTwo`.

4.6 SOLVING VLE

The `VLEModel` class is an abstract class containing a single, abstract method, `calculateKi()`. This method is the only public method in all sub-classes, and is solely responsible for receiving a `FlashTank` object and calculating and returning the k_i values of each species in the outlet of the `FlashTank`. All of the previously mentioned cases are constructed with a `VLEModel` subclass and a `FlashTank` object. The type of `VLEModel` subclass determines the method used so solve the VLE of the system: `IdealModel` calculates k_i values from Antoine's equation, `PRModel` uses the Peng-Robinson EOS to predict k_i values, and `WilsonSRKModel` uses the Wilson correlation for the Sloave-Reidlich-Kwong EOS to predict k_i values. The Peng-Robinson method requires numerous intermediate calculations, which are divided into private methods within the `PRModel` class. To avoid repetitive passing of the `FlashTank` object throughout the various methods, `PRModel` has a `FlashTank` instance variable. Upon being passed a `FlashTank` object, the `calculateKi()` model immediately copies it to the instance variable.

4.7 NUMERICAL METHODS

All numerical methods are contained in the `numericMethods` package. The `RootFinder` class contains static methods for finding roots to any object implementing the `HasRoot` interface, which contains a single method definition for `findYGivenX()`. Static methods for root-finding methods using Bisection, Ridder's and Newton-Raphson were built. The bisection and Ridder's methods both take upper and lower bounds as arguments, and all three methods have arguments for tolerance, target value, and maximum number of iterations. Ridder's method was the only method implemented in the code, as it does not require any derivatives, and converges faster than bisection. The `ridderRoot()` method throws a `BadFunction` exception if a root is not found in the specified interval. The class `CubicEquationSolver` does what its name suggests, and is used to solve the Peng-Robinson cubic EOS for the compressibility factor. If there is only one solution to the cubic equation, `CubicEquationSolver` throws a `NotFlashable` exception. Ideally this exception should be thrown from `PRModel` class, and not from the `CubicEquationSolver` class. As it currently is, `CubicEquationSolver` is not robust, and only works in the context of a cubic EOS. Three root-finding methods were tested, but ultimately Ridder's method was implemented. Ridder's method is guaranteed to converge faster than bisection if a root is bounded by upper and lower bounds, and doesn't rely on an initial guess or being able to compute the derivative of a function like the Newton-Raphson method.

The `HasRoot` interface is always implemented as an anonymous class within the parameter list of the root-finding methods as they are called in the `FlashModel` cases. As stated previously, this allows for multiple implementations of the `HasRoot` interface within the `FlashModel` cases.

4.8 USER I/O

`Group1FlashTankSimulator` prompts users to add the simulation parameters to the text file supplied with the source code, or to input the same parameters directly into the console. If the user chooses to input the data through the text file, then they must do so in format outlined in the Read Me file. The main method checks that most inputs are valid before proceeding. The exception to this are non-negative temperature checks, species double-entry checks, and pressure drop tests. In the latter, the check to determine if $P_{feed} > P_{flash}$ is not performed because the P_{feed} does not play a role in any calculations; the enthalpy change due to pressure was assumed to be negligible. After performing the simulation, the system prints the results to the console and prompts the user to save a file. If the user chooses to save the outputs of the simulation, they can name a new output text file or use the default one (overriding any results already stored there).

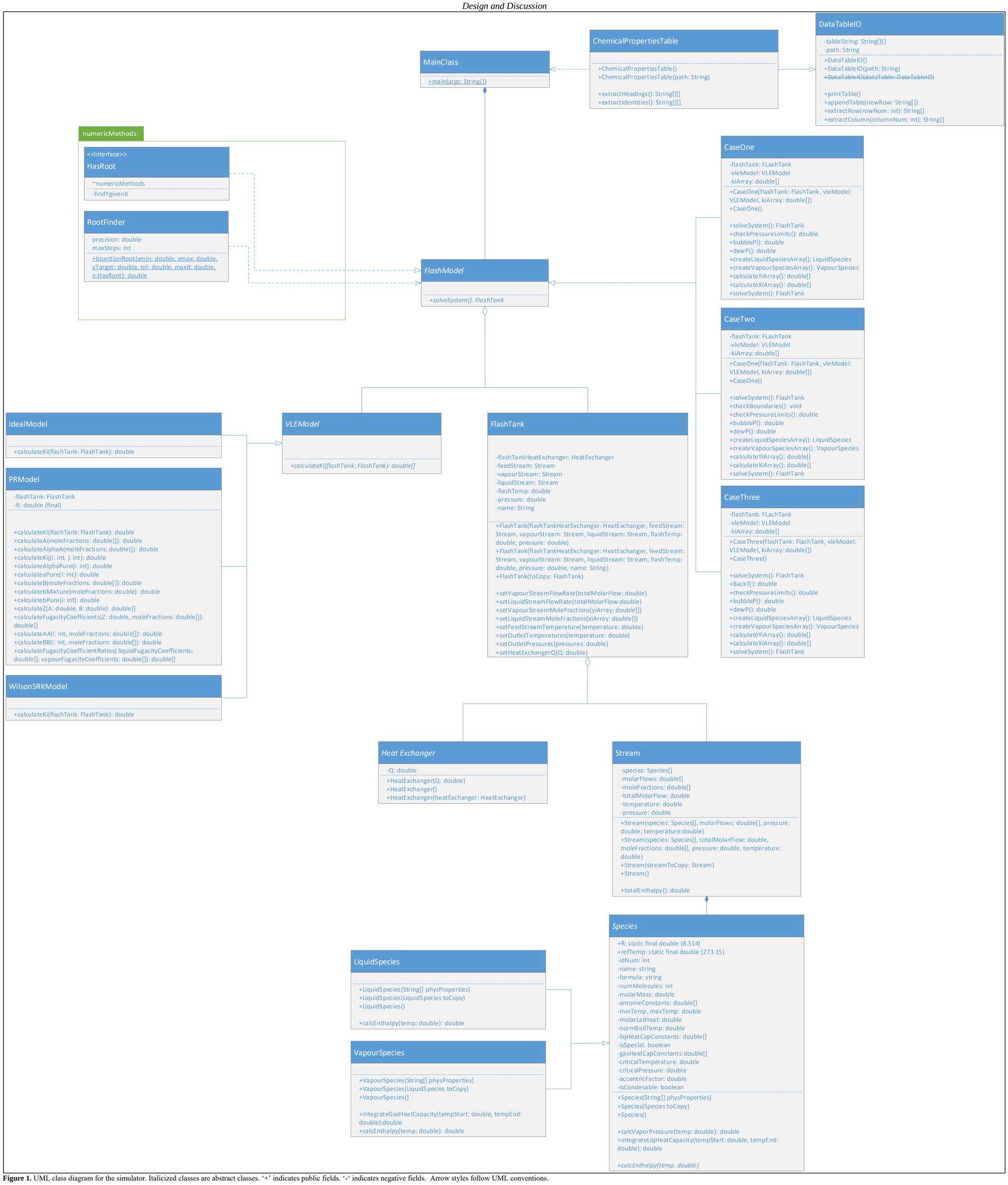

Figure 1. UML class diagram for the simulator. Italicized classes are abstract classes. '+' indicates public fields. '-' indicates negative fields. Arrow styles follow UML conventions.

5 VALIDATION AND SIMULATION RESULTS

5.1 NUMERICAL METHOD VALIDATION

Ridder's method is used as the root-finding algorithm in this simulator. The programmed Ridder's method was validated against similar calculations performed in an excel sheet. The function $y = x^2 - 5$ was used for validation, as this function has a simple, irrational solution. To validate the programmed Ridder's method, code was temporarily introduced into the `RootFinder` class to print the state of the program at various iterations. This code was removed immediately after use. The programmed Ridder's method successfully calculated the value of *x* as 2.23607 (**Figure 2**, **Figure 3**).

		Equation being solved for	
Y Target	0	**f(x)=X^2-5**	
		Actual Solution	2.236068
	2		

Itteration	xL	xU	xM	f_xL	f_xU	f_xM	xR	f_xR	**xR*xM**	f_xL*f_xR	f_xR*f_xM	f_xM*f_xU	f_xL*f_xM	f_xR*f_xU
1	0	4	2	-5	11	-1	2.267261	0.140474	4.534522	-0.70237	-0.140473539	-11	5	1.545209
2	2	2.267261	2.13363062	-1	0.140474	-0.44762	2.236088	8.84E-05	4.770985	-8.8E-05	-3.95527E-05	-0.062879	0.44762	1.24E-05
3	2.133631	2.236088	2.18485918	-0.44762	8.84E-05	-0.22639	2.236068	1.19E-08	4.885494	-5.3E-09	-2.68453E-09	-2E-05	0.101337	1.05E-12
4	2.184859	2.236068	2.21046358	-0.22639	1.19E-08	-0.11385	2.236068	3.93E-13	4.942747	-8.9E-14	-4.47961E-14	-1.35E-09	0.025775	4.67E-21
5	2.210464	2.236068	2.22326578	-0.11385	3.93E-13	-0.05709	2.236068	8.88E-16	4.971373	-1E-16	-5.07055E-17	-2.25E-14	0.0065	3.49E-28

Figure 2. Intermediate steps using Ridder's method on excel to find the root of $f(x) = x^2-5$.

```
Welcome to DrJava.  Working directory is C:\Users\Jeff\Desktop\Computer Aided Design\Flash Tank Simulator\.vscode
> run RootFinderValidation
xL: 0.0
xM: 2.0
xU: 4.0
xR: 2.2672612419124243
f_xL: -5.0
f_xM: -1.0
f_xU: 11.0
f_xR: 0.14047353907826832
xL: 2.0
xM: 2.133630620956212
xU: 2.2672612419124243
xR: 2.2360877357711173
f_xL: -1.0
f_xM: -0.44762037331800997
f_xU: 0.14047353907826832
f_xR: 8.836206600193464E-5
xL: 2.133630620956212
xM: 2.1848591783636646
xU: 2.2360877357711173
xR: 2.2360679801513115
f_xL: -0.44762037331800997
f_xM: -0.2263903707200523
f_xU: 8.836206600193464E-5
f_xR: 1.1857966164541267E-8
... solved after 3 iterations...
>
```

Figure 3. Intermediate steps using `Rootfinder.ridderRoot()` to find the solution to $f(x) = x^2-5$.

5.2 INTERMEDIATE VALUE VALIDATION

Determination of the K_i values is the most calculation-intensive process in the simulator due to the various intermediate calculations needed to solve the Peng-Robinson EOS. Ideal K_i values were compared against those generated by an ideal-case spreadsheet ("validation.xslx", provided with this report) and were found to agree closely. A summary can be seen in **Table 1**. Non-ideal cases were validated on Unisim R430 software, but this software doesn't provide intermediate calculations for the VLE models. For this reason, non-ideal K_i values were validated by comparing final simulation results to Unisim results. A short validation for pentane and water is presented in "validation.xslx", but these calculations were primarily used for debugging. Several online spreadsheets are available for calculation of Peng-Robinson parameters of mixtures of species (www.cheguide.com, www.learncheme.com). As the veracity of these spreadsheets is unproven, the results aren't included. However, it is worth noting that the intermediate Peng-Robinson parameters appeared to agree with those calculated in our simulator.

Table 1. Ideal K_i values calculated using the simulator and the "validation.xlsx" spreadsheet. Only condensable species are presented, as non-condensable species have infinitely large K_i values.

Species	Simulator K_i	Spreadsheet K_i
cyclohexane	0.901427972	0.9014272
hexane	1.29972539	1.29972539
n-pentane	3.397871369	3.397871368
water	0.417216947	0.417216947

5.3 ENTHALPY VALIDATION

Enthalpy calculations were validated two ways. First, the latent heat values were back calculated, and heat capacities were checked against values found by quick literature searches (data not shown). These steps served for preliminary validation. Final validation was performed by comparing simulation results to values calculated in "validation.xlsx". For the 6-main species tested in this simulator, all values were in perfect agreement (up to 10^{-6}, data not shown). These results are available in "validation.xlsx".

5.4 VALIDATION OF RESULTS

The results from the simulation presented in this report were validated alongside those from a simulator created by another group. These results were validated against the "validation.xlsx" spreadsheet for the ideal case, and the Unisim R430 software using the Peng-Robinson fluid package for the non-ideal. Full detailed results can be seen in **Figure 4**-**Figure 7**. Workable copies of these tables are available in validation.xlsx. Non-ideal and ideal flashes were tested for each case.

	Group Code			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 1								
Case Number	Case 1								
Validation Source	Excel Validation Sheet								
Feed Pressure (Pa)	200000								
Tank Pressure (Pa)	100000								
Fluid Model	Ideal			Ideal			Ideal		
Molar Flow (Mol/s)	100	30.8	69.2	100	30.4	69.6	100	30.8	69.2
Temperature (K)	338.15	338.15	338.15	338.15	338.15	338.15	338.15	338.15	338.15
Q To Tank (J/s)	-	11,600,000		-	901,600			1,020,000	
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.7	0.49	0.79	0.7	0.49	0.79	0.7	0.49	0.79
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane									
Methanol									
Nitrogen									
n-Pentane	0.3	0.51	0.21	0.3	0.51	0.21	0.3	0.51	0.21
Water									
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	70.0	15.1	54.7	70.0	14.9	55.0	70.0	15.1	54.7
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane									
Methanol									
Nitrogen									
n-Pentane	30.0	15.7	14.5	30.0	15.5	14.6	30.0	15.7	14.5
Water									

	Java Simulation			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 2								
Case Number	Case 1								
Validation Source	Unisim								
Feed Pressure (Pa)	200000								
Tank Pressure (Pa)	100000								
Fluid Model	Peng Robinson			"Non Ideal"			Peng Robinson		
Molar Flow (Mol/s)	100	37.6	62.4	100	36.3	63.7	100	37.1	62.9
Temperature (K)	338.15	338.15	338.15	338.15	338.15	338.15	338.15	338.15	338.15
Q To Tank (J/s)	-10,200,000			-1,077,460			1,020,000		
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.7	0.516	0.811	0.7	0.51	0.81	0.7	0.512	0.811
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane									
Methanol									
Nitrogen									
n-Pentane	0.3	0.484	0.189	0.3	0.49	0.19	0.3	0.488	0.189
Water									
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	70.0	19.4	50.6	70.0	18.5	51.6	70.0	19.0	51.0
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane									
Methanol									
Nitrogen									
n-Pentane	30.0	18.2	11.8	30.0	17.8	12.1	30.0	18.1	11.9
Water									

Figure 4. Validation of Case One performance for both the simulator and anonymous simulator. A mixture of 70% cyclohexane and 30% n-pentane flashed at 338.15K and 100000 Pa.

	Java Simulation			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 7								
Case Number	Case 1								
Validation Source	Excel Validation Sheet								
Feed Pressure (Pa)	250000								
Tank Pressure (Pa)	150000								
Fluid Model	Ideal			Ideal			Ideal		
Molar Flow (Mol/s)	300	269.15	30.85	300	185.3	114.7	300	268.6	31.4
Temperature (K)	350	350	350	350	350	350			
Q To Tank (J/s)	1,230,000,000			-2,747,620			5,115,536		
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.1667	0.156	0.26	0.1667	0.01	0.27	0.1667	0.156	0.26
Ethane	0.1667	0.186	0	0.1667	0.27	0.01	0.1667	0.186	0
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	0.1667	0.164	0.19	0.1667	0.12	0.24	0.1667	0.164	0.19
Methanol									
Nitrogen	0.1667	0.186	0	0.1667	0.27	0	0.1667	0.186	0
n-Pentane	0.1667	0.177	0.08	0.1667	0.18	0.13	0.1667	0.177	0.08
Water	0.1667	0.131	0.47	0.1667	0.06	0.34	0.1667	0.131	0.47
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	50.0	42.0	8.0	50.0	1.9	31.0	50.0	41.9	8.2
Ethane	50.0	50.1	0.0	50.0	50.0	1.1	50.0	50.0	0.0
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	50.0	44.1	5.9	50.0	22.2	27.5	50.0	44.1	6.0
Methanol									
Nitrogen	50.0	50.1	0.0	50.0	50.0	0.0	50.0	50.0	0.0
n-Pentane	50.0	47.6	2.5	50.0	33.4	14.9	50.0	47.5	2.5
Water	50.0	35.3	14.5	50.0	11.1	39.0	50.0	35.2	14.8

	Java Simulation			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 8								
Case Number	Case 1								
Validation Source	Unisim								
Feed Pressure (Pa)	250000								
Tank Pressure (Pa)	150000								
Fluid Model	Peng Robinson			"Non Ideal"			Peng Robinson		
Molar Flow (Mol/s)	300	NS	NS	300	299.85	0.15	300	300	0
Temperature (K)	350	NS	NS	350	350	350	350	350	350
Q To Tank (J/s)	-						144,720,000,000		
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.1667			0.1667	0.1666	0.42	0.1667	0.1667	0
Ethane	0.1667			0.1667	0.1668	0.07	0.1667	0.1667	0
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	0.1667			0.1667	0.1666	0.3	0.1667	0.1667	0
Methanol									
Nitrogen	0.1667			0.1667	0.1668	0.001	0.1667	0.1667	0
n-Pentane	0.1667			0.1667	0.1667	0.12	0.1667	0.1667	0
Water	0.1667			0.1667	0.1665	0.075	0.1667	0.1667	0
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	50.0			50.0	49.9	0.1	50.0	50.0	0.0
Ethane	50.0			50.0	50.0	0.0	50.0	50.0	0.0
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	50.0			50.0	50.0	0.0	50.0	50.0	0.0
Methanol									
Nitrogen	50.0			50.0	50.0	0.0	50.0	50.0	0.0
n-Pentane	50.0			50.0	50.0	0.0	50.0	50.0	0.0
Water	50.0			50.0	49.9	0.0	50.0	50.0	0.0

Figure 5. Validation of Case One performance for both the simulator and anonymous simulator. A mixture of equal parts of all critical components at 150000 Pa and 350 K.

	Java Simulation			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 9								
Case Number	Case 2								
Validation Source	Excel Validation Sheet								
Feed Pressure (Pa)	300000								
Tank Pressure (Pa)	100000								
Fluid Model	Ideal			Ideal			Ideal		
Molar Flow (Mol/s)	150	15.9	134.1	150	16	134	150	16	134
Temperature (K)	350	331.7	331.7	350	331.7	331.7	350	331.7	331.7
Q To Tank (J/s)	0								
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.2	0.1	0.21	0.2	0.1	0.21	0.2	0.1	0.21
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	0.13	0.1	0.14	0.13	0.1	0.13	0.13	0.1	0.13
Methanol									
Nitrogen									
n-Pentane	0.4	0.74	0.36	0.4	0.74	0.36	0.4	0.74	0.36
Water	0.27	0.06	0.29	0.27	0.06	0.3	0.27	0.06	0.3
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	30.0	1.6	28.2	30.0	1.6	28.1	30.0	1.6	28.1
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	19.5	1.6	18.8	19.5	1.6	17.4	19.5	1.6	17.4
Methanol									
Nitrogen									
n-Pentane	60.0	11.8	48.3	60.0	11.8	48.2	60.0	11.8	48.2
Water	40.5	1.0	38.9	40.5	1.0	40.2	40.5	1.0	40.2

	Java Simulation			Anonymous Code			Source		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 10								
Case Number	Case 2								
Validation Source	Unisim								
Feed Pressure (Pa)									
Tank Pressure (Pa)									
Fluid Model	Peng Robinson			"Non Ideal"			Peng Robinson		
Molar Flow (Mol/s)	150	NS	NS	150	0.07324	149.9268	150	142.4	7.6
Temperature (K)	350	NS	NS	350	350	350	350	336.65	336.65
Q To Tank (J/s)	NS			-3212.24			0		
Species Molar Fraction	-								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	0.2	NS	NS	0.2	0.2	0.14	0.2	0.21	0
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	0.13	NS	NS	0.13	0.13	0.16	0.13	0.14	0
Methanol									
Nitrogen									
n-Pentane	0.4	NS	NS	0.4	0.4	0.14	0.4	0.42	0
Water	0.27	NS	NS	0.27	0.27	0.017	0.27	0.23	1
Species Molar Flows	Mols/s								
Benzene									
iso-Butane									
n-Butane									
Cyclohexane	30.0	NS	NS	30.0	0.0	21.0	30.0	29.9	0.0
Ethane									
Ethanol									
Ethylbenzene									
n-Heptane									
n-Hexane	19.5	NS	NS	19.5	0.0	24.0	19.5	19.9	0.0
Methanol									
Nitrogen									
n-Pentane	60.0	NS	NS	60.0	0.0	21.0	60.0	59.8	0.0
Water	40.5	NS	NS	40.5	0.0	2.5	40.5	32.8	7.6

Figure 6. Validation of Case Two performance for both the simulator and anonymous simulator. A mixture of %20 cyclohexane, 13% n-hexane, 40% n-pentane, %27 water at 1000000 Pa and 350 K.

	Java Simulation			Source			Anonymous Code			Java Simulation			Source			Anonymous Code		
	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid	Feed	Vapour	Liquid
Validation Number	Validation 5									Validation 6								
Case Number	Case 3									Case 3								
Validation Source	Excel Validation Sheet									Unisim								
Feed Pressure (Pa)	200000									200000								
Tank Pressure (Pa)	100000									100000								
Fluid Model	Ideal			Ideal			-			Peng Robinson			Peng Robinson			"Non-Ideal"		
Molar Flow (Mol/s)	100	30.79	69.21	100	30.8	69.2	100	NS	NS	100	37.6	62.4	100	37.24	62.76	100	NS	NS
Temperature (K)	327.16	338.15	338.15	384.2	338.15	338.15	NS	338.15	338.15	332.2	338.15	338.15	359.93	338.15	338.15	NS	338.15	338.15
Q To Tank (J/s)	0			0			0			0			0			0		
Species Molar Fractior	6 Species									6 Species								
Benzene																		
iso-Butane																		
n-Butane																		
Cyclohexane	0.7	0.489	0.794	0.7	0.49	0.794	NS	NS	NS	0.7	0.516	0.811	0.7	0.513	0.811	0.7	NS	NS
Ethane																		
Ethanol																		
Ethylbenzene																		
n-Heptane																		
n-Hexane																		
Methanol																		
Nitrogen																		
n-Pentane	0.3	0.511	0.206	0.3	0.51	0.206	NS	NS	NS	0.3	0.484	0.189	0.3	0.487	0.189	0.3	NS	NS
Water																		
Species Molar Flows	Mols/s									Mols/s								
Benzene																		
iso-Butane																		
n-Butane																		
Cyclohexane	70.0	15.1	55.0	70.0	15.1	54.9	NS	NS	NS	70.0	19.4	50.6	70.0	19.1	50.9	70.0	NS	NS
Ethane																		
Ethanol																		
Ethylbenzene																		
n-Heptane																		
n-Hexane																		
Methanol																		
Nitrogen																		
n-Pentane	30.0	15.7	14.3	30.0	15.7	14.3	NS	NS	NS	30.0	18.2	11.8	30.0	18.1	11.9	30.0	NS	NS
Water																		

Figure 7. Validation of Case Three performance for both the simulator and anonymous simulator. A mixture of 70% cyclohexane, 30% n-pentane at 1000000 Pa and 338.15 K.

5.5 TESTING FOR DUBIOUS INPUTS

The robustness of the simulator was assessed by entering in many dubious inputs and judging the output of the simulation. This section presents multiple dubious tests performed and the results of such tests. Note that the same tests performed here were performed for the anonymous simulator (**Section 7.3**). The dubious tests are summarized in **Table 2**.

Table 2. Dubious case tests. All cases were tested under the following operating conditions: case 1, T = 350 K, P = 101325 Pa, 100 mol/s of cyclohexane (50%) and n-pentane (50%). For each case where one of the operating conditions was changed, the corresponding change is listed. "Pass" indicates the simulator performed the action that should be performed to handle the dubious input, while "fail" indicates the simulator was unable to perform the correct action.

Test	Special Operating Conditions	Pass/Fail	Description
Two of the same species	Water (0.5) and water (0.5)	Fail	Says that operating pressure is above the bubble pressure. Ends program.
Negative temperature	T = -350K (case 1/2/3)	Fail	Case 1 and 3: Says that operating pressure is below dew pressure. Case 2: Says that adiabatic flash is not possible [333-346 K]
Negative mole fraction	(-0.7) and (0.3)	Pass/fail	Pass: when entering mole fractions through the interactions pane. Fail: when entering mole fractions through Inputs.txt.
Negative flow rate	-100 mol/s	Pass/fail	Pass: when entering flow rate through the interactions pane. Fail: when entering mole fractions through Inputs.txt.
Negative pressure	-101325 Pa	Fail	Reports that operating pressure is above bubble pressure for the system.
Addition of species	Copy/pasted data for n-pentane in new row and renamed "TEST"	Pass	Correctly recognizes the new species, and gives proper separation with what would be n-pentane.
File in wrong directory	Remove DataSIUnits.csv or remove Inputs.tct	Pass	

Of the seven tests performed, the simulator could handle two of them properly, while an additional two functioned properly depending on whether the data was input through the interactions pane or the input file. All the failed tests could be properly accounted for through the use of exception handling in the constructor of the objects that use the inputs in question.

6 EXTENSIONS AND IMPROVEMENTS

While our simulation was designed with the intention of being robust and accurate, there are certain limitations that exist in the final design. Some limitations are due to programming errors discovered during the validation, and others are due to assumptions made in the design. The main limitations introduced by programming errors are in the bounds of the temperature iteration process in `CaseThree`: this process was left hardcoded to search for an adiabatic feed temperature between 300 and 400 K. In following versions, this should be changed to use bounds calculated in the exact same way as in `CaseTwo`. The enthalpy balances in all `FlashModel` subclasses need to be modified to remove the incorrect double-multiplication of each stream's total molar flow rate. Further debugging is necessary.

Future versions can improve the design of the program to make it more robust, to increase the accuracy of the simulations, and to improve user experience. For example, the non-ideal case was solved using the Peng-Robinson equations of state. Use of this cubic equation of state requires an intermediate calculation involving a binary interaction parameter k_{ij} (**Equation 8**).

$$(\alpha a)_{ij} = \sqrt{(\alpha a)_i (\alpha a)_j}(1 - k_{ij}) \quad \textbf{(8)}$$

This interaction parameter is an empirical parameter dependent on two different chemicals. While the binary interaction parameter is available in literature, the use of the parameter would require a specific value for each pair of interacting species. With six species being studied, this would mean that 15 different k_{ij} values would be necessary to account for each binary interaction. For simplicity, the value of k_{ij} was assumed to be ideal (zero) for all cases. Proper implementation of the binary interaction parameters would increase the accuracy of the Peng-Robinson equations of state, further improving the estimation of the VLE in the system.

In our simulations we assumed nitrogen and ethanol to be non-condensable in all cases. While this assumption is valid for the pure substances (as their normal boiling temperatures are 77.35 K and 184.1 K, respectively)[3], in mixtures the two species may behave differently; this may result in trace amounts being condensed into the liquid phase.

The user experience of our can be drastically improved. The simulator can be aesthetically improved by creating a user interface for simulation input and output, and using dialog boxes for file management. Additionally, adding options for the user to choose the type of numerical methods used in the simulator would help increase usability and would allow the user to increase/decrease tolerance and CPU load. Finally, the simulator should be adjusted to automatically determine which case needs to be used based on user input, and not through console ask/answer blocks.

7 TESTING OF ANONYMOUS CODE

This section will discuss and analyze another flash tank simulator. The structure and the design of the simulator will be explored, the accuracy of the simulator will be validated, and the robustness of the simulator will be tested through the use of dubious input values. The validations and dubious input values tested for the anonymous code were also tested for our own simulator.

7.1 ASSESSMENT OF CODE STRUCTURE

This section aims to judge the structure of an anonymous group's flash tank simulator. It will assess their use of object-oriented design, employment of inheritance and polymorphism, and adherence to proper Java conventions and styles. This will be performed by assessing each class, starting with the main method.

7.1.1 SysEnterence

The main method is a short method; its purpose is to ask the user to confirm inputs from the attached CSV data file, "expData.csv". The main method then stores the selection as an `int` and constructs a `FlashUnit` object by calling the `FileManager` class using the input as an argument. `FileManager` acts as the file I/O program for the simulator but it seemingly is also used to construct the `FlashUnit` object.

7.1.2 FileManager

This class performs file I/O for the program. It consists of nine private static instance variables, as they will not be changed during the simulation. This explains the lack of accessor and mutator methods in the class. The first instance variable, `dbaseSize`, is a final integer value that is declared outside the constructor. While instance variables are usually only declared outside the constructor in this situation, `dbaseSize` should not be initialized and defined outside the constructor. Since its size is hard-coded to be six, the program will always only be able to read a data sheet that contains six components. There is a comment in the code that explains that the user can simply change the value directly in the class. However, the user should not be changing the code directly, nor should be allowed to. Therefore, the database size should be automatically determined by the file manager directly.

The most glaring error in FileManager is the lack of a proper constructor. Instead of a constructor, there is a method named `loadComponentsDB` that returns a `FlashUnit` object based on the user's number of components entered in the main method. This method, which is 200 lines long, performs the actions of what a constructor should, as well as performs the actions of what the method's name implies. In the beginning of the method instance variables are initialized, as should be done in a constructor. The method then uses the received information from the file to create `GasComp` and `LiquidComp` objects, as well as a `FlashUnit` object, which is the object that is returned. These object constructions should arguably not be done in a file manager; in following object-oriented design, it would make sense that the file manager simply imports the data from the file. Ideally the data should be passed from the file manager to an object designed for building the system given the component and operating conditions.

The class also contains several peculiarities that should be mentioned. For example, the code contains many `try/catch` blocks for detecting errors, such as if the file is not found or if there is a

mismatch in the number of components. While error handling is important and appreciated, these tasks were not implemented properly. Custom exceptions should exist in their own class, and in this case the method should simply be throwing the errors as they arise instead of throwing and catching in the same class. Even more peculiar is how the error messages are implemented. For example, at lines 34-37, there is an `if` statement to detect whether the DBase.csv file is in the correct folder. If this statement is `false`, the program prints the error and, using a `while` loop, counts down from 20000 to 0 before ending the program through `System.exit(0)`. This was seemingly used to delay the system exit, giving the user a small amount of time to read the error message before the program was ended. This is an unnecessary solution to an unnecessary problem; proper use of exception handling would omit the need to use `System.exit(0)` altogether.

7.1.3 Component, GasComp, and LiquidComp

`Component` is an abstract class with the intention of represent one of the components being flashed. Therefore, it has two child classes, `GasComp` and `LiquidComp`. While the idea of making an abstract component and concrete gaseous components and liquid components is sound, the execution was poor. Firstly, all instance variables in Component are protected so that they can be used in the child classes without using to using accessors and mutators. The use of child classes should not result in changing instance variables from private to protected; this can result in security breaches as the instance variables become easily accessible. Secondly, the instance variables that are arrays were all initialized outside of the constructor, against proper Java conventions. (In fact, as with `FileManager`, there was no constructor.) Finally, the class has an abstract accessor to be overwritten in the child classes; this should not be performed.

`Component` and its two child classes do not employ proper techniques for inherited classes. `GasComp` and `LiquidComp` have identical methods except for one method, `PsatCr`, that is found in `GasComp`. All shared methods should be found in the parent class as they would be inherited from the parent class. The shared methods are abstract in the Component class instead of being defined. Since there are no two methods that have different definitions in `GasComp` and `LiqComp`, there should not be any abstract methods in Component except for the clone method. GasComp and LiqComp also have methods for calculating parameters for the Peng-Robinson EOS. The non-ideal VLE (or any VLE) models should not be in these classes as a chemical component is not defined by these parameters.

7.1.4 FlashUnit

`FlashUnit` is responsible for solving the system. Given the size of the project. the class is large: It is almost 1000 lines, indicating that many of its functions should be performed by separate classes, or that there are redundancies that are taking up space. There are 17 instance variables, many of which, such as `n` and `counter`, should not be instance variables given the way they are used. Many of the names of the instance variables were capitalized and therefore did not follow proper conventions in naming variables. The instance variables are also all declared to be static; there are some comments aiming to explain why this is the case, but they are unclear.

The constructor receives parameters sent from the `FileManager` class that were in turn received from the data file. While the constructor in this case uses the parameter list to initialize the instance variables,

it does so incorrectly. Most importantly, when initializing its own component instance variables, it does not use the clone method for the component objects (lines 49-50). Even though proper deep copying is employed, since the clone method is not invoked the reference location of the object is still copied, resulting in an important privacy leak. Importantly, the copy constructor also contains this privacy leak.

Unlike `FileManager`, `FlashUnit` does have proper exception handling to a certain extent. While the custom exception `NoSeparationException` is properly defined in its own class, this exception is used to define every type of reason that the flash may not be possible. While this is not incorrect, ideally there should be a different exception for each reason of an inseparable system. In addition, there is still a `try/catch` block instead of simply throwing the exception and making another class handle the error.

The program solves the system in an interesting way. Case one, case two, and case three are each defined twice: once for and ideal system and once for a non-ideal system. Since the VLE is not inputted into the solutions generically, this means that other methods of VLE (such as Wilson SRK, UNIFAC) cannot easily be implemented in the code. Since the three cases are also directly coded into `FlashUnit`, it becomes difficult to add other cases to be solved from the simulator. Both limit the robustness of the program. Many variables are also initialized within each case, and such variables are also common between cases. This demonstrates great redundancy and inefficiency in the code. Since so much of the simulation is performed in this class, it highlights the lack of polymorphic and object-oriented principles in the design.

7.1.5 Function

`Function` is an interface acting as the basis for the functions that are used in the root finder methods. However, it contains multiple method names for the same calculation of root finding. A single method should be defined to represent a function being used in root finding. This would ensure a generic implementation of the function so it could be used in different root finders.

7.1.6 RootFinder

`RootFinder` is used to solve systems with more that one unknown. It was designed to work using bisection method, a primitive root finding method that can take long to converge. The root finder is also defined to only handle the cases for the simulation, and is controlled by a `switch` method. This limits the functionality of the root finder and restricts it to the specific cases being studied.

7.2 VALIDATION OF RESULTS

The anonymous group's code was validated under the same conditions as the code presented in this report. These results are presented in detail alongside the results in Section **5.3** (**Figure 4**-**Figure 7**). Overall the simulation worked well for ideal case trials, but would often not solve non-ideal cases. Finding solutions for Case One isothermal problems was easier than finding solutions for the adiabatic cases. The enthalpy calculations tended to be negative and far from the Q predicted in "validation.xlsx" (ideal) and Unisim (non-ideal).

7.3 TESTING FOR DUBIOUS INPUTS

The anonymous simulator was tested for robustness by entering in multiple dubious values. This section will present multiple dubious tests performed and the results of such tests. Note that the same tests performed here were performed for our own simulator (**Section 5.1**). The dubious tests are summarized in **Table 3**.

Table 3: Dubious case tests for anonymous code. All cases were tested under the following operating conditions: case 1, T = 350 K, P = 101325 Pa, 100 mol/s of cyclohexane (50%) and n-pentane (50%). For each case where one of the operating conditions was changed, the corresponding change is listed. "Pass" indicates the simulator performed the action that should be performed to handle the dubious input, while "fail" indicates the simulator was unable to perform the correct action.

Test	Special Operating Conditions	Pass/Fail	Description
Two of the same species	Water (0.5) and water (0.5)	Fail	Says that no flash is possible, but does not state why. Ends program.
Negative temperature	T = -350K (case 1/2/3)	Fail	No separation possible, but not because of the negative input: reports that T_{bubble} = T_{dew} = 373.0 K.
Negative mole fraction	(-0.7) and (0.3)	Pass	Says that mole fractions do not add to 1. Ends program.
Negative flow rate	-100 mol/s	Fail	Solves system with flow rates as if they were positive (correct magnitudes but negative signs for V and L).
Negative pressure	-101325 Pa	Fail	Says that pressure is "Beyond Iteration Limit Calculating T bubble." Ends program.
Addition of species	Copy/pasted data for n-pentane in new row and renamed "TEST"	Fail	System.exit(0). Reports "One of the components is null." Note: works if a component is replaced with another name/component.

File in wrong directory	Remove expData.csv or DBase.csv	Pass	Reports that the file is not found.

Of the seven tests performed on the anonymous code, the simulator could handle two of them properly. For the case of negative pressure, while the simulator did not give a solution, as expected, it was not able to conclude that this was due to the dubious pressure entered. For the addition of species test, the reason for this error is reported in **Section 7.1.2**. Proper use of exception handling would greatly increase the robustness of the simulator.

8 CONCLUSIONS

The flash simulator was designed to be robust, generic, and accurate of a simple flash process that could solve one of three cases. In order to confirm the accuracy of the simulator, numerical validation was performed using external sources: an Excel spreadsheet was used to confirm the ideal cases while Unisim was used to confirm the non-ideal cases. To test the robustness of the simulator, numerous dubious input values were inputted into the simulation. Validation was also performed on an anonymous simulator.

Intermediate calculations were tested under ideal conditions. First, the validity of the coded Ridder's method, part of the `RootFinder` class, was assessed. A trivial function was tested using the coded Ridder's method and using an Excel sheet. The coded Ridder's method successfully solved the function in only 3 iterations. Next, the ideal K_i values for each of the condensable species were also tested against an Excel validation. Under the same conditions, the K values calculated from the code were the exact same as those calculated from Excel.

Each case was tested under different systems to test the accuracy of the simulation. Four tests were performed for case 1, which consisted of isothermal flash and determination of heat added or cooled from the flash separator. Of these four tests, two were the same test but with ideal or non-ideal solutions. Under ideal conditions in a two-component system, the simulator predicted identical separation as the Excel spreadsheet. The anonymous simulator also correctly predicted the separation. Using non-ideal solutions with the same system, the simulator also correctly predicted the separation when compared to Unisim; the two solutions were almost identical. The anonymous simulation also solved this problem with ease; it was also within 10% for the calculation of the heat removed from the system. However, it should be noted that the heat calculation was grossly over-estimated for the simulator. As previously mentioned, this was due to multiplying by enthalpy in twice. When testing a system with 6 components that did not flash, the simulator correctly did not find a simulation.

Two tests were performed for case 2, which was defined by solving for the unknown adiabatic flash temperature at a given feed temperature. A four-component mixture was. While both the simulator and the anonymous simulator predicted the correct solution under ideal conditions, neither could predict the correct separation under non-ideal solution. For our simulator, the solution simply could not be converged. For the anonymous simulator, an grossly incorrect solution was achieved, and the system was not adiabatic.

Only two tests were performed for case 3, which is defined by solving for the unknown adiabatic feed temperature given the flash temperature. A two component mixture was tested under ideal and non-ideal conditions. The anonymous simulator could not determine a solution in both cases. Our simulator was able to correctly determine the compositions of the outlet streams in both cases, but it was unable to correctly predict the feed temperature. In both ideal and non-ideal situations, the simulator predicted a lower feed temperature than flash temperature, while the two sources of validation each predicted a larger feed temperature than flash temperature.

Dubious input values were inputted into each simulator to test their robustness. Of the seven tests performed, the simulator passed two of them completely, passed two under certain circumstances, and failed three completely. This ultimately depended on whether data was inputted directly into the console or through the input text file. Proper exception handling in the constructors of the classes that use the desired information would properly detect these dubious inputs; this would also render it able to catch errors regardless of whether the information was entered through the console or the input text file. The anonymous code passed two out of the seven tests performed. The simulator did not employ exception handling in a meaningful way, indicating that the proper use of exception handling could greatly help its robustness.

Overall, the designed simulator could correctly predict certain flash cases. It was also much more accurate and robust than the anonymous code, and better employed an object-oriented design. However, neither was properly able to handle all studied validations. More importantly, due to time constraints, full validation was not performed. Ideally, more tests would be performed to test the two simulators. For example, for case 2, tests would be performed using the non-condensable species nitrogen and ethane. More tests would also be performed assessing the validity of case 2 and case 3. In addition, validation of more intermediate calculations would be performed. Doing this could help in determining discrepancies with the simulators and the validation sources.

9 REFERENCES

[1] W. Savitch, *Absolute Java*, Pearson Education, Essex, **2016**.

[2] A. Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, *The Java Language Specification*, Oracle America, Redwood City, **2015**.

[3] J. M. Smith, H. C. Van Ness, M. M. Abbott, *Introduction to Chemical Engineering Introduction to Chemical Engineering*, MCGraw-Hill, **2003**.

[4] R. H. Perry, D. W. Green, *Perry's Chemical Engineers' Handbook*, McGraw-Hill, New York, **2008**.

10 APPENDIX – SAMPLE CALCULATIONS

10.1 ENTHALPIES

10.1.1 Liquid Enthalpies

For all species, liquid enthalpies are calculated using the correlation provided in Perry's Handbook where C_1 through C_5 are constants.[4]

$$\Delta h_i^{liq} = C_1(T - T_{ref}) + C_2\frac{(T^2 - T_{ref}^2)}{2} + C_3\frac{(T^3 - T_{ref}^3)}{3} + C_4\frac{(T^4 - T_{ref}^4)}{4}C_5\frac{(T^5 - T_{ref}^5)}{5} \quad \textbf{(A1)}$$

10.1.2 Gas Enthalpies

In general, gas enthalpies are calculated using correlations provided in Smith and Van Ness, where C_1 through C_4 are constants (not the same as the liquid enthalpy constants).[3]

$$\int_{T_1}^{T_2} c_{p,gas} = C_1(T_2 - T_1) + \frac{C_2(T_2^2 - T_1^2)}{2} + \frac{C_3(T_2^3 - T_1^3)}{3} - C_4\left(\frac{1}{T_2} - \frac{1}{T_1}\right) \quad \textbf{(A2)}$$

For condensable species the latent heat must be included. Since the latent heat is only known at the normal boiling temperature, we integrate through the liquid heat capacity to get to that temperature, add the latent heat (λ), and then integrate through the gas heat capacity to get to the final temperature.

$$\Delta h_i^{gas} = \int_{T_{ref}}^{T_b} c_{p,liq}dT + \lambda + \int_{T_b}^{T} c_{p,gas}dT \quad \textbf{(A3)}$$

For non-condensable species:

$$\Delta h_i^{gas} = \int_{T_{ref}}^{T} c_{p,gas}dT \quad \textbf{(A4)}$$

The total feed stream enthalpy is calculated as a mole fraction weighted sum of individual species enthalpies

$$\Delta H = \textbf{Molar Flow} * (\sum z_i \Delta h_i) \quad \textbf{(A5)}$$

The general Enthalpy balance

$$\boldsymbol{Q + \Delta H_F = \Delta H_V + \Delta H_L} \quad \textbf{(A6)}$$

10.2 *K* VALUES

10.2.1 Ideal *K* Values

Ideal *K* values (repesenting the VLE of each species) are calculated using Raoult's law.

$$\boldsymbol{K_i = \frac{P_i^{sat}}{P} = \frac{y_i}{x_i}} \quad \textbf{(A7)}$$

10.2.2 Non-Ideal k Values (Wilson Soave-Reidlich-Kwong)

K-values are also calculated using the Wilson SRK correlation. These *k* values are primarily used as guesses for the non-ideal case. This equation uses the critical pressure (P_{ci}), the critical temperature (T_{ci}) and the accentric factor (ω_i) of each species.

$$\boldsymbol{K_i = \frac{y_i}{x_i} = \frac{P_{ci}}{P} exp\left(5.37(1+\omega_i)\left(1-\frac{T_{ci}}{T}\right)\right)} \quad \textbf{(A8)}$$

10.2.3 Non-Ideal k Values (Peng Robinson)

None ideal models are created to compensate for phenomena at high pressures and to account for intra-species interactions. These non-ideal models allow for a different method of calculating K_i than the proposed earlier.

$$\boldsymbol{K_i = \frac{y_i}{x_i} = \frac{\phi_i^L}{\phi_i^V}} \quad \textbf{(A9)}$$

Where ϕ_L is the fugacity coefficient of the liquid and ϕ_V is the fugacity coefficient for the vapour. These can be calculated from the Peng-Robinson equation of state as demonstrated in the following equations.

The Peng-Robinson equation is:

$$\boldsymbol{P = \frac{RT}{\tilde{v} - b} - \frac{\alpha a}{\tilde{v}^2 + 2b\tilde{v} - b^2}} \quad \textbf{(A10)}$$

$$\boldsymbol{\alpha = \left[1 + \left(0.37464 + 1.5422\omega - 0.2699\omega^2\right)\left(1 - \sqrt{T_r}\right)\right]^2} \quad \textbf{(A11)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c} \quad \textbf{(A12)}$$

$$b = 0.07780\frac{RT_c}{P_c} \quad \textbf{(A13)}$$

The first step to finding the fugacity coefficients is to find the roots to the cubic EOS (the bulk liquid and gas phase compressabilities). If the equation has one root, the mixture is either a super cooled liquid or a super heated vapour. Otherwise the smaller solution represents the liquid phase and the larger the vapour phase.

$$Z^3 - (1 - B)Z^2 + \left(A - 2B - 3B^2\right)Z - \left(AB - B^2 - B^3\right) = 0 \quad \textbf{(A14)}$$

With the variables:

$$A = \frac{\alpha aP}{R^2T^2} \quad \textbf{(A15)}$$

$$B = \frac{bP}{RT} \quad \textbf{(A16)}$$

For mixtures, the above terms have to be calculated using the Peng-Robinson mixing rules.

$$(\alpha a)_m = \sum\sum y_i y_j (\alpha a)_{ij} \quad \textbf{(A17)}$$

$$(\alpha a)_{ij} = \sqrt{(\alpha a)_i (\alpha a)_j}\left(1 - k_{ij}\right) \quad \textbf{(A18)}$$

$$b_m = \sum y_i b_i \quad \textbf{(A19)}$$

Once all the above parameters are calculated, the fugacity coefficients can be calculated for each phase using the respective compressibility factors. Allowing for calculation of the K_i values for each species according to **Equation A8**

$$\ln(\boldsymbol{\phi_i}) = (\boldsymbol{BB})_i\left(\boldsymbol{Z_p} - \mathbf{1}\right) - \boldsymbol{ln}\left(\boldsymbol{Z_p} - \boldsymbol{B}\right) - \frac{\boldsymbol{A}}{\mathbf{2}\sqrt{\mathbf{2}}\boldsymbol{B}}\left((\boldsymbol{AA})_i - (\boldsymbol{BB})_i\right)\ln\left[+\frac{\boldsymbol{Z_p} + \left(\sqrt{\mathbf{2}} + \mathbf{1}\right)\boldsymbol{B}}{\boldsymbol{Z_p} - \left(\sqrt{\mathbf{2}} - \mathbf{1}\right)\boldsymbol{B}}\right] \quad \textbf{(A20)}$$

With:

$$(\boldsymbol{AA})_i = \frac{\mathbf{2}}{(\boldsymbol{a\alpha})_m}\left[\sum_j^{n_c}(\boldsymbol{a\alpha})_{ij}\right] \quad \textbf{(A21)}$$

$$(\boldsymbol{BB})_i = \frac{\boldsymbol{b_i}}{\boldsymbol{b_m}} \quad \textbf{(A22)}$$

10.3 System Parameters

With K_i values known, the parameters of the flash tank can be found by solving from the objective Rashford-Rice function for ψ (**Equation A22**). This property essentially defines flow rate of the vapour stream (**Equation A23**), from which every remaining property can be calculated.

$$\sum_{i=1}^{n}\frac{\boldsymbol{z_i}(\boldsymbol{K_i} - \mathbf{1})}{\mathbf{1} + \boldsymbol{\psi}(\boldsymbol{K_i} - \mathbf{1})} = \mathbf{0} \quad \textbf{(A23)}$$

$$\boldsymbol{V} = \boldsymbol{F} * \boldsymbol{\psi} \quad \textbf{(A24)}$$

$$L=F-V \quad \textbf{(A25)}$$

$$y_i = \frac{Fz_iK_i}{F + V * (K_i - 1)} \qquad (condensable) \quad \textbf{(A26a)}$$

$$y_i = \frac{Fz_i}{V} \qquad (non - condensable) \quad \textbf{(A26b)}$$

$$x_i = \frac{Fz_i - Vy_i}{L} \quad \textbf{(A27)}$$

10.4 Bubble and Dew Point Checks

If the system is between the bubble and dew pressures, then the following inequalities for bubble and dew points hold true (**Equations A28** and **A29** respectively)

$$\sum_{i=1}^{n} x_i ki > 1 \quad \textbf{(A28)}$$

$$\sum_{i=1}^{n} \frac{y_i}{K_i} > 1? \quad \textbf{(A29)}$$

10.5 Numerical Methods

10.6 Numerical Methods

10.6.1 Bisection method

1. Starting with two points, x_L and x_U, which has the root bounded ($f(x_L) \cdot f(x_U) < 0$)
2. Find the midpoint: $x_R = (x_L + x_U)/2$
3. If $f(x_L) \cdot f(x_R) < 0$ then set $x_U = x_R$, else set $x_L = x_R$
4. Repeat Steps 2 to 3 until $\varepsilon_A < \varepsilon_s$, where

$$\varepsilon_A = \left| \frac{X_R^{new} - X_R^{old}}{X_R^{new}} \right| \times 100\% \quad \textbf{(A30)}$$

5. The root is set to the midpoint of x_L and x_U with an error of $(x_u - x_l)/2$
6. Improve the convergence by systematically splitting the interval in half

10.6.2 Ridder's method

$$x_R = x_M + (x_M - x_L) \frac{sgn[f(x_L) - f(x_U)] f(x_M)}{\sqrt{f^2(x_M) - f(x_L) f(x_U)}} \quad \textbf{(A31)}$$

1. Find x_M from the bound roots with $x_M = (x_L + x_U)/2$.
2. Calculate x_R.
3. Find which interval has the root bound.
4. Set X_L and X_U based on the new bounds.
5. Repeat Steps 1 to 4 until $\varepsilon_A < \varepsilon_b$ or the maximum number of iterations is reached
6. Set the root to equal the last x_R.

10.6.3 Newton's method

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad \textbf{(A32)}$$

Algorithm

1. With an initial guess, x_i, calculate $f(x_i)$ and $f'(x_i)$

2. Calculate x_{i+1}

3. Repeat Steps 1 and 2 until the exact root is found, or the desired error is reached.

$$\varepsilon_A = \left|\frac{x_{i+1} - x_i}{x_{i+1}}\right| \times 100\%$$

4. Check to make sure that $f(x_{i+1}) \approx 0$ and set the root to the last x_{i+1}.

11 APPENDIX – SOURCE CODE

11.1 CODING

11.1.1 CaseOne

```
import numericMethods.*;
import customExceptions.NotFlashable;
import customExceptions.BadFunction;

/**
 * A FlashModel (FlashClient) class that represents system described in "Case One" in the project requirment
document.
 * Requires Validation and Exception Handling Analysis.
 */
public class CaseOne extends FlashModel
{


     private FlashTank flashTank;
     private VLEModel vleModel;
     private double[] kiArray;



//-------------------------------------------------CONSTRUCTORS-----------------------------------------
-------------


     /**
      * Takes in all requred instance variables for Case One (A tank, pressure and temperature).
      * @param flashTank must be a properly consturcted FlashTank Object
      * @param vleModel The model that is used to determine the Ki values.
      */
     public CaseOne(FlashTank flashTank, VLEModel vleModel){

          this.flashTank = flashTank.clone();
          this.vleModel = vleModel; //No params, nothing to clone
          this.kiArray = new double[flashTank.getFeedStream().getSpecies().length];

     }

     /**
      * Default constructor
      * For constructing a blank object in the main method for the purpose of being overwritten
      * */
     public CaseOne()
     {
          this.flashTank = null;
          this.vleModel = null;
          this.kiArray = null;
     }

//-----------------------------------------------------METHODS----------------------------------------
-------------


     /**
      * This FlashClients required solveSystem method definition.
      * @throws NotFlashable if the bubble and dewpoint checks fail.
      * @return a FlashTank object with all the parameters calculated according to case1
      */
     public FlashTank solveSystem() throws NotFlashable{
```

```
//Create default liquid and vapour outlet streams @the flash temp and pressure
LiquidSpecies[] liquidStreamSpecies = createLiquidSpeciesArray();
VapourSpecies[] vapourStreamSpecies = createVapourSpeciesArray();

double outletTemperature = this.flashTank.getFlashTemp();
double outletPressure = this.flashTank.getPressure();
double[] defaultMoleFractions = this.flashTank.getFeedStream().getMoleFractions();

this.flashTank.setLiquidStream(new Stream(liquidStreamSpecies, 0.0, defaultMoleFractions,
                                          outletPressure, outletTemperature));
this.flashTank.setVapourStream(new Stream(vapourStreamSpecies, 0.0, defaultMoleFractions,
                                          outletPressure, outletTemperature));

//Check if it is flashable
checkFlashable();

//Guess ki values
this.kiArray = (new WilsonSRKModel()).calculateKi(this.flashTank);

double tolerance = 0.0001;
double maxDifference = 0.0;
int counter = 1;

try{
     do {
          counter++;
          //Calculate V, uses Ki values w. Ridder's Method
          //Implements the interface in an anonymous class
          double VFRatio = RootFinder.ridderRoot(0,1,0,0.0001,2000, new HasRoot() {
               public double findYGivenX(double psi) {

                    double summation = 0.0;

                    //Extract feed mole fractions
                    double[] moleFractionsFeed = flashTank.getFeedStream().getMoleFractions();

                    //Calculate the sum of yi
                    for (int i=0; i<flashTank.getFeedStream().getSpecies().length;i++)
                    {
                         summation += moleFractionsFeed[i] * (kiArray[i] - 1)
                              / (1 + psi * (kiArray[i]-1));
                    }

                    return summation;
               }
          } ); //returns VFRatio that satisfies mass balance

          double vapourFlow = this.flashTank.getFeedStream().getTotalMolarFlow() * VFRatio;

          this.flashTank.setVapourStreamFlowRate(vapourFlow);

          //Calculate L
          double liquidFlow = flashTank.getFeedStream().getTotalMolarFlow() - vapourFlow;
          this.flashTank.setLiquidStreamFlowRate(liquidFlow);

          //Calculate yi[]
          double[] yiArray = calculateYiArray();
          this.flashTank.setVapourStreamMoleFractions(yiArray);

          //Calculate xi[] using mass balance (not k values)
          double[] xiArray = calculateXiArray();
          this.flashTank.setLiquidStreamMoleFractions(xiArray);

          //Recalculate the K values and compare
          double[] kiOld = this.kiArray;
          this.kiArray = vleModel.calculateKi(this.flashTank.clone());

          maxDifference = 0.0;
          for(int i=0; i<this.kiArray.length; i++) {
```

```
                    double difference = Math.abs(kiOld[i]-kiArray[i]);
                    if(difference >= maxDifference)
                        maxDifference = difference;
                }

            } while((maxDifference>tolerance) && (counter<1000));
        }
        catch(BadFunction e) {
            System.out.println("Something broke in case one. You should never get this error here");
            System.exit(0);
        }

        //Calculate Q, the temperature of the flash tank is known at this point
        //Q = V*Hv + L*Hl - F*Hf
        double Q = this.flashTank.getVapourStream().getTotalMolarFlow() *
this.flashTank.getVapourStream().totalEnthalpy()
            + this.flashTank.getLiquidStream().getTotalMolarFlow() *
this.flashTank.getLiquidStream().totalEnthalpy()
            - this.flashTank.getFeedStream().getTotalMolarFlow() *
this.flashTank.getFeedStream().totalEnthalpy();
        this.flashTank.setHeatExchangerQ(Q);


        return this.flashTank.clone();
    }


    private void checkFlashable() throws NotFlashable{
        //DEW POINT CHECK
        FlashTank localTank = this.flashTank.clone();
        //guess Ki
        double[] localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
        double[] localKiArrayOld = null;

        //yi = zi
        localTank.setVapourStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

        //calculate Ki and Xi
        double[] xiArray = new double[localTank.getFeedStream().getSpecies().length];
        double tolerance = 0.0001;
        double maxDifference = 0.0;
        int counter = 0;
        do{
            counter++;

            //Calculate xiArray
            for(int i=0; i<xiArray.length; i++) {
                if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                    xiArray[i] = localTank.getFeedStream().getMoleFractions()[i]/localKiArray[i];
                else
                    xiArray[i] = 0;
            }
            localTank.setLiquidStreamMoleFractions(xiArray);

            //calculate new ki vals
            localKiArrayOld = localKiArray;
            localKiArray = this.vleModel.calculateKi(localTank);

            maxDifference = 0.0;
            for(int i=0; i<localKiArray.length; i++) {
                double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                if(difference >= maxDifference)
                    maxDifference = difference;
            }
        }while((maxDifference>tolerance) && (counter<1000));

        //calculate the dewsum
        double dewSum = 0.0;
        for(int i=0; i<xiArray.length; i++) {
```

```
                dewSum += xiArray[i]; //y/k
            }

            //BUBBLE POINT CHECK
            localTank = this.flashTank.clone();
            //guess Ki
            localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
            localKiArrayOld = null;

            //xi=zi
            localTank.setLiquidStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

            //calculate Ki and yi
            double[] yiArray = new double[localTank.getFeedStream().getSpecies().length];
            tolerance = 0.0001;
            maxDifference = 0.0;
            counter = 0;
            do{
                counter++;

                //Calculate yiArray
                for(int i=0; i<yiArray.length; i++) {
                    if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                        yiArray[i] = localTank.getFeedStream().getMoleFractions()[i] * localKiArray[i];
                    else
                        yiArray[i] = localTank.getFeedStream().getMoleFractions()[i];
                }
                localTank.setVapourStreamMoleFractions(yiArray);

                //calculate new ki vals
                localKiArrayOld = localKiArray;
                localKiArray = this.vleModel.calculateKi(localTank);

                maxDifference = 0.0;
                for(int i=0; i<localKiArray.length; i++) {
                    double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                    if(difference >= maxDifference)
                        maxDifference = difference;
                }
            }while((maxDifference>tolerance) && (counter<1000));

            //calculate the bubbleSum
            double bubbleSum = 0.0;
            for(int i=0; i<yiArray.length; i++) {
                bubbleSum += yiArray[i]; //x*k
            }

            if(bubbleSum<1)
                throw new NotFlashable("System not solved...\nThe operating pressure is above the bubble
pressure for this system.");
            else if(dewSum<1)
                throw new NotFlashable("System not solved...\nThe operating pressure is below the dew pressure
for this system.");
        }


        /**
         * Creates an array of liquid species matching the species in the feed stream.
         * @return an array of LiquidSpecies matching the feed stream species.
         */
        private LiquidSpecies[] createLiquidSpeciesArray() {

            int length = this.flashTank.getFeedStream().getSpecies().length;
            LiquidSpecies[] toReturn = new LiquidSpecies[length];

            for(int i=0; i<length; i++)
            {
                toReturn[i] = new LiquidSpecies(flashTank.getFeedStream().getSpecies()[i]); //The way
LiquidSpecies is made, this will work even if the feed has different phases
```

```
          }

          return toReturn;
     }


     /**
      * Creates an array of vapour species matching the species in the feed stream
      * @return an array of vapour species matching the feed stream species
      */
     private VapourSpecies[] createVapourSpeciesArray() {

          int length = flashTank.getFeedStream().getSpecies().length;
          VapourSpecies[] toReturn = new VapourSpecies[length];

          for(int i=0; i<length; i++)
          {
               toReturn[i]  =  new  VapourSpecies(flashTank.getFeedStream().getSpecies()[i]);  //The  way
LiquidSpecies is made, this will work even if the feed has different phases
          }

          return toReturn;
     }


     /**
      * Only called after the flash tank V has been calculated
      * @return The array of yi values
      */
     private double[] calculateYiArray() {

          double[] yiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

          double[] moleFractionsFeed = this.flashTank.getFeedStream().getMoleFractions();
          double V = this.flashTank.getVapourStream().getTotalMolarFlow();

          //Calculate yis
          for (int i=0; i<this.flashTank.getFeedStream().getSpecies().length;i++)
          {
               if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable()) { //non-condensable
                    yiArray[i] = flashTank.getFeedStream().getTotalMolarFlow() * moleFractionsFeed[i] / V;
               }
               else {//condensable
                    yiArray[i]  =  flashTank.getFeedStream().getTotalMolarFlow()  *  moleFractionsFeed[i]*
kiArray[i]
                         / (flashTank.getFeedStream().getTotalMolarFlow() + V*(this.kiArray[i]-1));
               }
          }

          return yiArray;
     }

      /**
      * Only called after the flash tank V has been calculated
      * @return The array of xi values
      */
     private double[] calculateXiArray() {

          double[] xiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

          double[] yiArray = this.flashTank.getVapourStream().getMoleFractions();

          for(int i=0; i<xiArray.length; i++) {
               xiArray[i]         =         (this.flashTank.getFeedStream().getTotalMolarFlow()         *
this.flashTank.getFeedStream().getMoleFractions()[i]
                                   -            this.flashTank.getVapourStream().getTotalMolarFlow()      *
this.flashTank.getVapourStream().getMoleFractions()[i])
                    / this.flashTank.getLiquidStream().getTotalMolarFlow();
          }
```

```
          return xiArray;
     }
}//End of class
```

11.1.2 CaseTwo

```
import numericMethods.*;
import customExceptions.NotFlashable;
import customExceptions.BadFunction;
import java.util.Arrays;

/**
 * A FlashModel (FlashClient) class that represents system described in "Case One" in the project requirment
document
 * Requires Validation and Exception Handling Analysis.
 * @version 11.0 - 2017/11/21
 */
public class CaseTwo extends FlashModel
{


     private FlashTank flashTank;
     private VLEModel vleModel;
     private double[] kiArray;



//-------------------------------------------------CONSTRUCTORS-----------------------------------------
-------------


     /**
      * Takes in all requred instance variables for Case One (A tank, pressure and temperature).
      * @param flashTank must be a properly consturcted FlashTank Object
      * @param vleModel The model used in determining the K values
      */
     public CaseTwo(FlashTank flashTank, VLEModel vleModel){

          this.flashTank = flashTank.clone();
          this.vleModel = vleModel; //No params, nothing to clone
          this.kiArray = new double[flashTank.getFeedStream().getSpecies().length];

     }

     /**
      * Default constructor
      * For constructing a blank object in the main method for the purpose of being overwritten
      * */
     public CaseTwo()
     {
          this.flashTank = null;
          this.vleModel = null;
          this.kiArray = null;
     }

//-----------------------------------------------------METHODS----------------------------------------
-------------


     /**
      * This FlashClients required solveSystem method definition.
      * @throws NotFlashable if the bubble and dewpoint checks fail.
      * @return a FlashTank object with all the parameters calculated according to case1
      */
     public FlashTank solveSystem() throws NotFlashable{

          //Create default liquid and vapour outlet streams @the flash temp and pressure
```

```
          LiquidSpecies[] liquidStreamSpecies = createLiquidSpeciesArray();
          VapourSpecies[] vapourStreamSpecies = createVapourSpeciesArray();

          double outletTemperature = 0; //default flashTemp
          double outletPressure = this.flashTank.getPressure();
          double[] defaultMoleFractions = this.flashTank.getFeedStream().getMoleFractions();

          this.flashTank.setLiquidStream(new Stream(liquidStreamSpecies, 0.0, defaultMoleFractions,
                                                    outletPressure, outletTemperature));
          this.flashTank.setVapourStream(new Stream(vapourStreamSpecies, 0.0, defaultMoleFractions,
                                                    outletPressure, outletTemperature));

          //get Temperature bounds from bubble to dew


          double[] bounds = findBounds();

          System.out.println(Arrays.toString(bounds));
          //Ridder's method through temp to find an adiabatic temp
          try{
               double adiabaticTemp = RootFinder.ridderRoot(bounds[0], bounds[1], 0, 0.1, 2000, new HasRoot()
{
                    public double findYGivenX(double T) throws BadFunction {
                         flashTank.setFlashTemp(T);

                         //Guess ki values
                         kiArray = (new WilsonSRKModel()).calculateKi(flashTank);

                         double tolerance = 0.0001;
                         double maxDifference = 0.0;
                         int counter = 1;

                         do {//solve the system at the given temp
                              counter++;
                              //Calculate V/F uses Ki values w. Ridder's method
                              //Implements the interface in an anonymous class
                              double VFRatio = RootFinder.ridderRoot(0,1,0,0.0001,2000, new HasRoot() {
                                   public double findYGivenX(double psi) {

                                        double summation = 0.0;

                                        //Extract feed mole fractions
                                        double[]                    moleFractionsFeed                    =
flashTank.getFeedStream().getMoleFractions();

                                        //Calculate the sum of yi
                                        for (int i=0; i<flashTank.getFeedStream().getSpecies().length;i++)
                                        {
                                             summation += moleFractionsFeed[i] * (kiArray[i] - 1)
                                                  / (1 + psi * (kiArray[i]-1));
                                        }
                                        //System.out.println("Summation is: " + summation + " psi is: " +
psi);

                                        return summation;
                                   }
                              } ); //returns VFRatio that satisfies mass balance

                              double vapourFlow = flashTank.getFeedStream().getTotalMolarFlow() * VFRatio;

                              flashTank.setVapourStreamFlowRate(vapourFlow);

                              //Calculate L
                              double   liquidFlow   =   flashTank.getFeedStream().getTotalMolarFlow()   -
vapourFlow;
                              flashTank.setLiquidStreamFlowRate(liquidFlow);

                              //Calculate yi[]
                              double[] yiArray = calculateYiArray();
```

```
                              flashTank.setVapourStreamMoleFractions(yiArray);

                              //Calculate xi[] using mass balance (not k values)
                              double[] xiArray = calculateXiArray();
                              flashTank.setLiquidStreamMoleFractions(xiArray);

                              //Recalculate the K values and compare
                              double[] kiOld = kiArray;
                              try {
                                   kiArray = vleModel.calculateKi(flashTank.clone());
                              }
                              catch (NotFlashable e) {
                                   throw new BadFunction("The origin of this is a NotFlashable error in case
2."
                                                                  + "It likely means that no T could be found
in the flashable interval such that"
                                                                  + "Q = 0... Returning the lowest Q found");
                              }

                              maxDifference = 0.0;
                              for(int i=0; i<kiArray.length; i++) {
                                   double difference = Math.abs(kiOld[i]-kiArray[i]);
                                   if(difference >= maxDifference)
                                        maxDifference = difference;
                              }

                         } while((maxDifference>tolerance) && (counter<1000));

                         //Calculate Q, the temperature of the flash tank is known at this point
                         //Q = V*Hv + L*Hl - F*Hf
                         double      Q      =      flashTank.getVapourStream().getTotalMolarFlow()      *
flashTank.getVapourStream().totalEnthalpy()
                              +              flashTank.getLiquidStream().getTotalMolarFlow()              *
flashTank.getLiquidStream().totalEnthalpy()
                              -               flashTank.getFeedStream().getTotalMolarFlow()               *
flashTank.getFeedStream().totalEnthalpy();
                         flashTank.setHeatExchangerQ(Q);

                         System.out.println(T+"-"+Q);
                         return Q;
                    }
               });
          }
          catch(BadFunction e)
          {
               throw new NotFlashable("An adiabatic flash between [" + bounds[0] + "," + bounds[1] + "] is
not possible");
          }



          return this.flashTank.clone();
     }

     /**
      * Determines if the mixture is flashable at a given pressure and temperature by calculating the
sum(xi*ki) and sum(yi/ki).
      * If either of these are below 1, then the mixture is not flashable under the given conditions.
      * @throws NotFlashable if the mixture exceeds the bubble pressure or falls short of the dew pressure
      */
     private void checkFlashable() throws NotFlashable{
     //DEW POINT CHECK
          FlashTank localTank = this.flashTank.clone();
          //guess Ki
          double[] localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
          double[] localKiArrayOld = null;

          //yi = zi
          localTank.setVapourStreamMoleFractions(localTank.getFeedStream().getMoleFractions());
```

```
        //calculate Ki and Xi
        double[] xiArray = new double[localTank.getFeedStream().getSpecies().length];
        double tolerance = 0.0001;
        double maxDifference = 0.0;
        int counter = 0;
        do{
            counter++;

            //Calculate xiArray
            for(int i=0; i<xiArray.length; i++) {
                if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                    xiArray[i] = localTank.getFeedStream().getMoleFractions()[i]/localKiArray[i];
                else
                    xiArray[i] = 0;
            }
            localTank.setLiquidStreamMoleFractions(xiArray);

            //calculate new ki vals
            localKiArrayOld = localKiArray;
            localKiArray = this.vleModel.calculateKi(localTank);

            maxDifference = 0.0;
            for(int i=0; i<localKiArray.length; i++) {
                double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                if(difference >= maxDifference)
                    maxDifference = difference;
            }
        }while((maxDifference>tolerance) && (counter<1000));

        //calculate the dewsum
        double dewSum = 0.0;
        for(int i=0; i<xiArray.length; i++) {
            dewSum += xiArray[i]; //y/k
        }

    //BUBBLE POINT CHECK
        localTank = this.flashTank.clone();
        //guess Ki
        localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
        localKiArrayOld = null;

        //xi=zi
        localTank.setLiquidStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

        //calculate Ki and yi
        double[] yiArray = new double[localTank.getFeedStream().getSpecies().length];
        tolerance = 0.0001;
        maxDifference = 0.0;
        counter = 0;
        do{
            counter++;

            //Calculate yiArray
            for(int i=0; i<yiArray.length; i++) {
                if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                    yiArray[i] = localTank.getFeedStream().getMoleFractions()[i] * localKiArray[i];
                else
                    yiArray[i] = localTank.getFeedStream().getMoleFractions()[i];
            }
            localTank.setVapourStreamMoleFractions(yiArray);

            //calculate new ki vals
            localKiArrayOld = localKiArray;
            localKiArray = this.vleModel.calculateKi(localTank);

            maxDifference = 0.0;
            for(int i=0; i<localKiArray.length; i++) {
                double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
```

```
                    if(difference >= maxDifference)
                         maxDifference = difference;
               }
          }while((maxDifference>tolerance) && (counter<1000));

          //calculate the bubbleSum
          double bubbleSum = 0.0;
          for(int i=0; i<yiArray.length; i++) {
               bubbleSum += yiArray[i]; //x*k
          }

          if(bubbleSum<1)
               throw new NotFlashable("System not solved...\nThe operating pressure is above the bubble
pressure for this system.");
          else if(dewSum<1)
               throw new NotFlashable("System not solved...\nThe operating pressure is below the dew pressure
for this system.");
     }

     /**
      * Determines an approximate dew and bubble temperature for the mixture by scanning from 100 to 2000K
for the (first) range
      * at which the mixture is flashable.
      * @return an array with [bubble Temp, dew Temp]
      * @throws NotFlashable if no flashable range is found
      */
     private double[] findBounds() throws NotFlashable {
          FlashTank flashTankHolder = this.flashTank.clone(); //gotta hold dat. I need the original back

          double T = 100; //gotta start somewhere
          double[] toReturn = new double[2];
          int counter = 0;

          //Get the lower flashable temperature. loops while it isnt flashable
          boolean isFlashable = false;
          while(!isFlashable&&(counter<=2000)) {
               try {
                    checkFlashable();
                    toReturn[0] = T;//lowerbound
                    isFlashable = true;//end the loop when its true
               }
               catch(NotFlashable e) {//if it isnt flashable, try the next temp
                    T++;
                    this.flashTank.setFlashTemp(T);
                    counter++;
               }
          }

          if(isFlashable == false) //if it hasn't reached flashable by 2000 iterations, its probably not
flashable
               throw new NotFlashable();

          //Get the upper flashable bound. Loops while it is flashable.
          while(isFlashable&&(counter<=2000)) {
               this.flashTank.setFlashTemp(T);
               try {//if it is flashable, try the next temperature
                    checkFlashable();
                    T++;
                    this.flashTank.setFlashTemp(T);
                    counter++;
               }
               catch(NotFlashable e) {
                    toReturn[1] = T;//upper bound
                    isFlashable = false;//end the loop when its false
               }

          }

          this.flashTank = flashTankHolder.clone();
```

```
        return toReturn;
    }

    /**
     * Caculates the dew and bubble temperatures
     * @return an array with [bubble temperature, dew temperature]
     */
    private double[] calculateTemperatureBounds() throws NotFlashable{
        FlashTank localTank = this.flashTank.clone();
        double[] toReturn = new double[2];


        //Estimate T
        double guessTemperature = 0.0;
        for(int i=0; i<localTank.getFeedStream().getSpecies().length; i++) {
            double Tc = localTank.getFeedStream().getSpecies()[i].getCriticalTemperature();
            double Pc = localTank.getFeedStream().getSpecies()[i].getCriticalPressure();
            double w = localTank.getFeedStream().getSpecies()[i].getAccentricFactor();
            double Ti_sat = Tc/(1 - 3 * Math.log(localTank.getPressure()/Pc) /
                                (Math.log(10) * (7+7*w)));

            if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                guessTemperature += Ti_sat * localTank.getFeedStream().getMoleFractions()[i];
        }

        localTank = this.flashTank.clone();

     //BUBBLE TEMPERATURE
        //guess Ki
        double[] localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
        double[] localKiArrayOld = null;

        //xi=zi
        localTank.setLiquidStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

        double bubbleTemperature = guessTemperature;
        double bubbleSum = 0.0;
        int counterOuter = 0;
        boolean hasBeenPassed = false;
        do {
            counterOuter++;

            //calculate Ki and yi
            double[] yiArray = new double[localTank.getFeedStream().getSpecies().length];
            double tolerance = 0.0001;
            double maxDifference = 0.0;
            int counterInner = 0;
            do{
                counterInner++;

                //Calculate yiArray
                for(int i=0; i<yiArray.length; i++) {
                    if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                        yiArray[i]      =      localTank.getFeedStream().getMoleFractions()[i]      *
localKiArray[i];
                    else
                        yiArray[i] = localTank.getFeedStream().getMoleFractions()[i];
                }
                localTank.setVapourStreamMoleFractions(yiArray);

                //calculate new ki vals


                localKiArrayOld = localKiArray;
                localKiArray = this.vleModel.calculateKi(localTank);

                maxDifference = 0.0;
                for(int i=0; i<localKiArray.length; i++) {
                    double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
```

```
                    if(difference >= maxDifference)
                        maxDifference = difference;
                }
            }while((maxDifference>tolerance) && (counterInner<1000));

            //calculate the bubbleSum
            double bubbleSumOld = bubbleSum;
            bubbleSum = 0.0;
            for(int i=0; i<yiArray.length; i++) {
                bubbleSum += yiArray[i]; //x*k
            }

            if(((bubbleSumOld<1)&&(bubbleSum>1))||((bubbleSumOld>1)&&(bubbleSum<1)))//if  the  step  has
passed the bubble point
                hasBeenPassed = true;
            else if (bubbleSum>1) {//the temp is too high
                bubbleTemperature--;
                localTank.setFlashTemp(bubbleTemperature);
            }
            else if(bubbleSum<1) {//the temp is too low
                bubbleTemperature++;
                localTank.setFlashTemp(bubbleTemperature);
            }

        }while((!hasBeenPassed)&&(counterOuter<1000));
        toReturn[0] = bubbleTemperature;

    //DEW TEMPERATURE
        localTank = this.flashTank.clone();
       //guess Ki
        localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
        localKiArrayOld = null;

        //yi = zi
        localTank.setVapourStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

        double dewTemperature = guessTemperature;
        double dewSum = 0;
        counterOuter = 0;
        hasBeenPassed = false;
        do{
            counterOuter++;

            //calculate Ki and Xi
            double[] xiArray = new double[localTank.getFeedStream().getSpecies().length];
            double tolerance = 0.0001;
            double maxDifference = 0.0;
            int counterInner = 0;
            do{
                counterInner++;

                //Calculate xiArray
                for(int i=0; i<xiArray.length; i++) {
                    if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                        xiArray[i] = localTank.getFeedStream().getMoleFractions()[i]/localKiArray[i];
                    else
                        xiArray[i] = 0;
                }
                localTank.setLiquidStreamMoleFractions(xiArray);

                //calculate new ki vals
                localKiArrayOld = localKiArray;
                localKiArray = this.vleModel.calculateKi(localTank);

                maxDifference = 0.0;
                for(int i=0; i<localKiArray.length; i++) {
                    double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                    if(difference >= maxDifference)
                        maxDifference = difference;
```

```
                    }
               }while((maxDifference>tolerance) && (counterInner<1000));

               //calculate the dewSum
               double dewSumOld = dewSum;
               dewSum = 0.0;
               for(int i=0; i<xiArray.length; i++) {
                    dewSum += xiArray[i]; //y/k
               }

               //figure out which way to move the dewSum
               if(((dewSumOld<1)&&(dewSum>1))||((dewSumOld>1)&&(dewSum<1)))//if the step has passed the
bubble point
                    hasBeenPassed = true;
               else if (dewSum>1) {//the temp is too high
                    dewTemperature--;
                    localTank.setFlashTemp(dewTemperature);
               }
               else if(dewSum<1) {//the temp is too low
                    dewTemperature++;
                    localTank.setFlashTemp(dewTemperature);
               }

          }while((!hasBeenPassed)&&(counterOuter<1000));
          toReturn[1] = dewTemperature;

          return toReturn;
     }


     /**
      * Creates an array of liquid species matching the species in the feed stream.
      * @return an array of LiquidSpecies matching the feed stream species.
      */
     private LiquidSpecies[] createLiquidSpeciesArray() {

          int length = this.flashTank.getFeedStream().getSpecies().length;
          LiquidSpecies[] toReturn = new LiquidSpecies[length];

          for(int i=0; i<length; i++)
          {
               toReturn[i] = new LiquidSpecies(flashTank.getFeedStream().getSpecies()[i]); //The way
LiquidSpecies is made, this will work even if the feed has different phases
          }

          return toReturn;
     }

     /**
      * Creates an array of vapour species matching the species in the feed stream
      * @return an array of vapour species matching the feed stream species
      */
     private VapourSpecies[] createVapourSpeciesArray() {

          int length = flashTank.getFeedStream().getSpecies().length;
          VapourSpecies[] toReturn = new VapourSpecies[length];

          for(int i=0; i<length; i++)
          {
               toReturn[i] = new VapourSpecies(flashTank.getFeedStream().getSpecies()[i]); //The way
LiquidSpecies is made, this will work even if the feed has different phases
          }

          return toReturn;
     }


     /**
      * Only called after the flash tank V has been calculated
```

```
     * P
     */
    private double[] calculateYiArray() {

        double[] yiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

        double[] moleFractionsFeed = this.flashTank.getFeedStream().getMoleFractions();
        double V = this.flashTank.getVapourStream().getTotalMolarFlow();

        //Calculate yis
        for (int i=0; i<this.flashTank.getFeedStream().getSpecies().length;i++)
        {
            if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable()) { //non-condensable
                yiArray[i] = flashTank.getFeedStream().getTotalMolarFlow() * moleFractionsFeed[i] / V;
            }
            else {//condensable
                yiArray[i]  =  flashTank.getFeedStream().getTotalMolarFlow()  *  moleFractionsFeed[i]*
kiArray[i]
                    / (flashTank.getFeedStream().getTotalMolarFlow() + V*(this.kiArray[i]-1));
            }
        }

        return yiArray;
    }


    private double[] calculateXiArray() {

        double[] xiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

        double[] yiArray = this.flashTank.getVapourStream().getMoleFractions();

        for(int i=0; i<xiArray.length; i++) {
            xiArray[i]         =         (this.flashTank.getFeedStream().getTotalMolarFlow()         *
this.flashTank.getFeedStream().getMoleFractions()[i]
                            -            this.flashTank.getVapourStream().getTotalMolarFlow()      *
this.flashTank.getVapourStream().getMoleFractions()[i])
                / this.flashTank.getLiquidStream().getTotalMolarFlow();

        }

        return xiArray;
    }
}//End of Class
```

11.1.3 CaseThree

```
import numericMethods.*;
import customExceptions.NotFlashable;
import customExceptions.BadFunction;

/**
 * A FlashModel (FlashClient) class that represents system described in "Case Three" in the project
requirement document
 * Requires Validation and Exception Handling Analysis.
 */
public class CaseThree extends FlashModel
{


    private FlashTank flashTank;
    private VLEModel vleModel;
    private double[] kiArray;



//-----------------------------------------------CONSTRUCTORS-----------------------------------------
-------------
```

```
     /**
      * Takes in all requred instance variables for Case One (A tank, pressure and temperature).
      * @param flashTank must be a properly consturcted FlashTank Object.
      * @param vleModel The model used in determining the K values.
      */
     public CaseThree(FlashTank flashTank, VLEModel vleModel){

          this.flashTank = flashTank.clone();
          this.vleModel = vleModel; //No params, nothing to clone
          this.kiArray = new double[flashTank.getFeedStream().getSpecies().length];

     }

     /**
      * Default constructor
      * For constructing a blank object in the main method for the purpose of being overwritten
      * */
     public CaseThree()
     {
          this.flashTank = null;
          this.vleModel = null;
          this.kiArray = null;
     }

//-----------------------------------------------------METHODS-----------------------------------------
-------------


     /**
      * This FlashClients required solveSystem method definition. Should have a "Logger" to inform user
status of system solution (completed succesfully, completed with errors etc)
      * @throws NotFlashable if the bubble and dewpoint checks fail.
      * @return a FlashTank object with all the parameters calculated according to case1
      */
     public FlashTank solveSystem() throws NotFlashable{

          //Create default liquid and vapour outlet streams @the flash temp and pressure
          LiquidSpecies[] liquidStreamSpecies = createLiquidSpeciesArray();
          VapourSpecies[] vapourStreamSpecies = createVapourSpeciesArray();

          double outletTemperature = this.flashTank.getFlashTemp();
          double outletPressure = this.flashTank.getPressure();
          double[] defaultMoleFractions = this.flashTank.getFeedStream().getMoleFractions();

          this.flashTank.setLiquidStream(new Stream(liquidStreamSpecies, 0.0, defaultMoleFractions,
                                                  outletPressure, outletTemperature));
          this.flashTank.setVapourStream(new Stream(vapourStreamSpecies, 0.0, defaultMoleFractions,
                                                  outletPressure, outletTemperature));

          //Check if it is flashable
          checkFlashable();

          //Guess ki values
          this.kiArray = (new WilsonSRKModel()).calculateKi(this.flashTank);

          double tolerance = 0.0001;
          double maxDifference = 0.0;
          int counter = 1;

          try{
               do {
                    counter++;
                    //Calculate V, uses Ki values w. Ridder's method
                    //Implements the interface in an anonymous class
                    double VFRatio = RootFinder.ridderRoot(0,1,0,0.0001,2000, new HasRoot() {
                         public double findYGivenX(double psi) {

                              double summation = 0.0;
```

```
                    //Extract feed mole fractions
                    double[] moleFractionsFeed = flashTank.getFeedStream().getMoleFractions();

                    //Calculate the sum of yi
                    for (int i=0; i<flashTank.getFeedStream().getSpecies().length;i++)
                    {
                         summation += moleFractionsFeed[i] * (kiArray[i] - 1)
                              / (1 + psi * (kiArray[i]-1));
                    }

                    return summation;
               }
          } ); //returns VFRatio that satisfies mass balance

          double vapourFlow = this.flashTank.getFeedStream().getTotalMolarFlow() * VFRatio;

          this.flashTank.setVapourStreamFlowRate(vapourFlow);

          //Calculate L
          double liquidFlow = flashTank.getFeedStream().getTotalMolarFlow() - vapourFlow;
          this.flashTank.setLiquidStreamFlowRate(liquidFlow);

          //Calculate yi[]
          double[] yiArray = calculateYiArray();
          this.flashTank.setVapourStreamMoleFractions(yiArray);

          //Calculate xi[] using mass balance (not k values)
          double[] xiArray = calculateXiArray();
          this.flashTank.setLiquidStreamMoleFractions(xiArray);

          //Recalculate the K values and compare
          double[] kiOld = this.kiArray;
          this.kiArray = vleModel.calculateKi(this.flashTank.clone());

          maxDifference = 0.0;
          for(int i=0; i<this.kiArray.length; i++) {
               double difference = Math.abs(kiOld[i]-kiArray[i]);
               if(difference >= maxDifference)
                    maxDifference = difference;
          }

     } while((maxDifference>tolerance) && (counter<1000));


     for(int T = 300; T<=400; T++) {
          FlashTank localFT = flashTank.clone();

          localFT.setFeedStreamTemperature(T);

          double     toReturn     =          localFT.getVapourStream().getTotalMolarFlow()     *
localFT.getVapourStream().totalEnthalpy()
               +                 localFT.getLiquidStream().getTotalMolarFlow()                *
localFT.getLiquidStream().totalEnthalpy()
               -                  localFT.getFeedStream().getTotalMolarFlow()                 *
localFT.getFeedStream().totalEnthalpy();


          System.out.println("T is: " + T + " and Q is " + toReturn);
     }

          //Calculate T of the feed from 0=VHv+LHl-FHf
          double backT = RootFinder.ridderRoot(1,2000.0,0,0.01,1000, new HasRoot() {
          public double findYGivenX(double T) {

               FlashTank localFT = flashTank.clone();

               localFT.setFeedStreamTemperature(T);
```

```
                        double    toReturn    =        localFT.getVapourStream().getTotalMolarFlow()    *
localFT.getVapourStream().totalEnthalpy()
                                +               localFT.getLiquidStream().getTotalMolarFlow()              *
localFT.getLiquidStream().totalEnthalpy()
                                -                localFT.getFeedStream().getTotalMolarFlow()               *
localFT.getFeedStream().totalEnthalpy();


                        System.out.println("T is: " + T + " and is it 0? " + toReturn);

                        return toReturn;
                    }
               }); //returns T that satisfies mass balance
               this.flashTank.setFeedStreamTemperature(backT);

          }
          catch(BadFunction e) {
               System.out.println("Something broke in case three. You should never get this error here");
               System.exit(0);
          }


          return this.flashTank.clone();
     }


     private void checkFlashable() throws NotFlashable{
          //DEW POINT CHECK
          FlashTank localTank = this.flashTank.clone();
          //guess Ki
          double[] localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
          double[] localKiArrayOld = null;

          //yi = zi
          localTank.setVapourStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

          //calculate Ki and Xi
          double[] xiArray = new double[localTank.getFeedStream().getSpecies().length];
          double tolerance = 0.0001;
          double maxDifference = 0.0;
          int counter = 0;
          do{
               counter++;

               //Calculate xiArray
               for(int i=0; i<xiArray.length; i++) {
                    if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                         xiArray[i] = localTank.getFeedStream().getMoleFractions()[i]/localKiArray[i];
                    else
                         xiArray[i] = 0;
               }
               localTank.setLiquidStreamMoleFractions(xiArray);

               //calculate new ki vals
               localKiArrayOld = localKiArray;
               localKiArray = this.vleModel.calculateKi(localTank);

               maxDifference = 0.0;
               for(int i=0; i<localKiArray.length; i++) {
                    double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                    if(difference >= maxDifference)
                         maxDifference = difference;
               }
          }while((maxDifference>tolerance) && (counter<1000));

          //calculate the dewsum
          double dewSum = 0.0;
          for(int i=0; i<xiArray.length; i++) {
               dewSum += xiArray[i]; //y/k
```

```
          }

          //BUBBLE POINT CHECK
          localTank = this.flashTank.clone();
          //guess Ki
          localKiArray = (new WilsonSRKModel()).calculateKi(localTank);
          localKiArrayOld = null;

          //xi=zi
          localTank.setLiquidStreamMoleFractions(localTank.getFeedStream().getMoleFractions());

          //calculate Ki and yi
          double[] yiArray = new double[localTank.getFeedStream().getSpecies().length];
          tolerance = 0.0001;
          maxDifference = 0.0;
          counter = 0;
          do{
               counter++;

               //Calculate yiArray
               for(int i=0; i<yiArray.length; i++) {
                    if(localTank.getFeedStream().getSpecies()[i].getIsCondensable())
                         yiArray[i] = localTank.getFeedStream().getMoleFractions()[i] * localKiArray[i];
                    else
                         yiArray[i] = localTank.getFeedStream().getMoleFractions()[i];
               }
               localTank.setVapourStreamMoleFractions(yiArray);

               //calculate new ki vals
               localKiArrayOld = localKiArray;
               localKiArray = this.vleModel.calculateKi(localTank);

               maxDifference = 0.0;
               for(int i=0; i<localKiArray.length; i++) {
                    double difference = Math.abs(localKiArrayOld[i]-localKiArray[i]);
                    if(difference >= maxDifference)
                         maxDifference = difference;
               }
          }while((maxDifference>tolerance) && (counter<1000));

          //calculate the bubbleSum
          double bubbleSum = 0.0;
          for(int i=0; i<yiArray.length; i++) {
               bubbleSum += yiArray[i]; //x*k
          }

          if(bubbleSum<1)
               throw new NotFlashable("System not solved...\nThe operating pressure is above the bubble
pressure for this system.");
          else if(dewSum<1)
               throw new NotFlashable("System not solved...\nThe operating pressure is below the dew pressure
for this system.");
     }


     /**
      * Creates an array of liquid species matching the species in the feed stream.
      * @return an array of LiquidSpecies matching the feed stream species.
      */
     private LiquidSpecies[] createLiquidSpeciesArray() {

          int length = this.flashTank.getFeedStream().getSpecies().length;
          LiquidSpecies[] toReturn = new LiquidSpecies[length];

          for(int i=0; i<length; i++)
          {
               toReturn[i]  =  new  LiquidSpecies(flashTank.getFeedStream().getSpecies()[i]);  //The  way
LiquidSpecies is made, this will work even if the feed has different phases
          }
```

```
        return toReturn;
    }

    /**
     * Creates an array of vapour species matching the species in the feed stream
     * @return an array of vapour species matching the feed stream species
     */
    private VapourSpecies[] createVapourSpeciesArray() {

        int length = flashTank.getFeedStream().getSpecies().length;
        VapourSpecies[] toReturn = new VapourSpecies[length];

        for(int i=0; i<length; i++)
        {
            toReturn[i] = new VapourSpecies(flashTank.getFeedStream().getSpecies()[i]); //The way
LiquidSpecies is made, this will work even if the feed has different phases
        }

        return toReturn;
    }


    /**
     * Only called after the flash tank V has been calculated
     * P
     */
    private double[] calculateYiArray() {

        double[] yiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

        double[] moleFractionsFeed = this.flashTank.getFeedStream().getMoleFractions();
        double V = this.flashTank.getVapourStream().getTotalMolarFlow();

        //Calculate yis
        for (int i=0; i<this.flashTank.getFeedStream().getSpecies().length;i++)
        {
            if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable()) { //non-condensable
                yiArray[i] = flashTank.getFeedStream().getTotalMolarFlow() * moleFractionsFeed[i] / V;
            }
            else {//condensable
                yiArray[i] = flashTank.getFeedStream().getTotalMolarFlow() * moleFractionsFeed[i]*
kiArray[i]
                    / (flashTank.getFeedStream().getTotalMolarFlow() + V*(this.kiArray[i]-1));
            }

        }

        return yiArray;
    }


    private double[] calculateXiArray() {

        double[] xiArray = new double[this.flashTank.getFeedStream().getSpecies().length];

        double[] yiArray = this.flashTank.getVapourStream().getMoleFractions();

        for(int i=0; i<xiArray.length; i++) {
            xiArray[i]         =         (this.flashTank.getFeedStream().getTotalMolarFlow()         *
this.flashTank.getFeedStream().getMoleFractions()[i]
                                -            this.flashTank.getVapourStream().getTotalMolarFlow()      *
this.flashTank.getVapourStream().getMoleFractions()[i])
                / this.flashTank.getLiquidStream().getTotalMolarFlow();

        }

        return xiArray;
    }
```

```
}//End of class
```

11.1.4 ChemicalPropertiesTable

```
/**
 * A table for chemical properties with unique methods for getting units and species names.
 */
public class ChemicalPropertiesTable extends DataTableIO
{

//-------------------------------------------------CONSTRUCTORS-----------------------------------------
------------


     /**
      * Prompts the user to select a CSV file containing the data table.
      */
     public ChemicalPropertiesTable()
     {
          super();
     }


     /**
      * Reads in the table at the path provided.
      * @param path a String containing the absolute path of the data file
      */
     public ChemicalPropertiesTable(String path)
     {
          super(path);
     }
//-------------------------------------------------METHODS-----------------------------------------------
-----------


     /**
      * Gets the first two rows of the table (Headings and units) and returns them as columns.
      * It's better to return a vertical array as it prints to the console much nicer.
      * @return a 2D array containing the headings and their associated units as columns
      */
     public String[][] extractHeadings()
     {
          String[][] headingsRow = new String[2][extractRow(0).length];

          //Get the first two rows of the table
          headingsRow[0] = extractRow(0); //the headings
          headingsRow[1] = extractRow(1); //the units

          //Transpose the array
          String[][] transposed = new String[headingsRow[0].length][headingsRow.length]; //makes number of
rows in new array = no columns in the old

          for(int i=0; i<headingsRow[0].length; i++)
          {
               for(int j=0; j<2; j++)
               {
                    transposed[i][j] = headingsRow[j][i];
               }
          }

          return transposed;
     }


     /**
      * Gets the species name and ID number columns and returns them as columns.
      * It's better to return a vertical array as it prints to the console much nicer.
      * @return a 2D array containing the species name and ID number
      */
```

```
     public String[][] extractIdentities()
     {
          //Get the two leftmost columns from the data table as arrays
          String[][] columnsAsRows = new String[2][extractColumn(0).length];
          columnsAsRows[0] = extractColumn(0);
          columnsAsRows[1] = extractColumn(1);

          //Transpose the array
          String[][] transposed = new String[columnsAsRows[0].length][columnsAsRows.length]; //makes number
of rows in new array = no columns in the old

          for(int i=0; i<columnsAsRows[0].length; i++)
          {
               for(int j=0; j<2; j++)
               {
                    transposed[i][j] = columnsAsRows[j][i];
               }
          }

          return transposed;
     }


//-------------------------------------------------TEST-------------------------------------------------
-----------
//     public static void main(String[] args)
//     {
//          ChemicalPropertiesTable chemTable = new ChemicalPropertiesTable();
//
//          String[][] toPrint = chemTable.extractIdentities();
//          for(String[] row : toPrint)
//          {
//               for(String element : row)
//               {
//                    System.out.printf("%12.12s\t", element);
//               }
//               System.out.println("");
//          }
//     }


}//End of ChemicalPropertiesTable class
```

11.1.5 DataTableIO

```
import java.io.PrintWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;

import javax.swing.JFileChooser;
import javax.swing.filechooser.FileNameExtensionFilter;
import java.awt.Component;

import customExceptions.*;

/**
 * Imports a comma-delimited txt file to read data and to add new data.
 * The values in the table cannot be changed but new rows can be added, in which case the original file is
also appended.
 * Note: Exception is commented out
 * @version 1.2 2017/10/31
 */
public class DataTableIO
{


     private String[][] tableString;
     private String path;
```

```
//-------------------------------------CONSTRUCTORS-----------------------------------------------------------
---


     /**
      * Prompts the user to select a .txt file containig a comma-delimited table.
      * Counts the number of rows and headings, and turns the table into a 2D array of strings.
      * */
     public DataTableIO()
     {
          System.out.println("Reading file...");

          //Must be initialized outside a codeblock
          tableString = null;
          Scanner inputScanner = null;
          Scanner scanNumOfLines = null;


          try { //reading in the file automatically
               inputScanner = new Scanner(new FileInputStream("DataSIUnits.csv"));
               scanNumOfLines = new Scanner(new FileInputStream("DataSIUnits.csv"));
               this.path = "DataSIUnits.csv";
          }
          catch(FileNotFoundException e) {
               System.out.println("DataSIUnits.csv not found in default directory. Please navigate to it");
               try                {//opening                a                dialogue                box
//shouldn't throw this exception but it needs to be handled to compile
                    //Build dialogue box
                    JFileChooser chooser = new JFileChooser();
                    FileNameExtensionFilter filter = new FileNameExtensionFilter(".csv or .txt files", "txt",
"csv");      //will only display txt and csv files
                    chooser.setFileFilter(filter);
                    //Display DB
                    int isApproved = chooser.showOpenDialog(null);
                    //Init scanner
                    inputScanner = new Scanner(new FileInputStream(chooser.getSelectedFile()));
                    scanNumOfLines = new Scanner(new FileInputStream(chooser.getSelectedFile()));
                    this.path = chooser.getSelectedFile().getAbsolutePath();
               }
               catch(FileNotFoundException f) {
                    System.out.println("The file you selected wasn't found.\nHonestly congrats.\nI don't
how this happened");
                    System.exit(0);
               }
          }



          /*
           * For the following, the scanNumOfLines is used.
           * Scanner can't reset it's position...
           * We need the other scanner object to actually read the table.
           */
          String[] headingStringArray = scanNumOfLines.nextLine().split(",");    //Split the first row into
headings
          int numRows, numColumns;

          numColumns = headingStringArray.length;
          numRows = 1;                                                    //The heading row has already
been used
          while(scanNumOfLines.hasNextLine())
          {
               numRows++;
               scanNumOfLines.nextLine();
          }
          scanNumOfLines.close();
```

```
          /*
           * Now the other scanner object can be used.
           * We can create the data table from a tab or comma delimited file.
           */
          this.tableString = new String[numRows][numColumns];

          inputScanner.useDelimiter(",|\\n");
          for(int i=0; i<this.tableString.length; i++)
          {
               for(int j=0; j<this.tableString[i].length; j++)
               {
                    this.tableString[i][j] = inputScanner.next();
               }
          }
          inputScanner.close();

     }//End of constructor


     /**
      * Takes in the path name for the CSV file and creates a string array containing the data
      * @param path a string for the absolute path to the data file
      *
      */
     public DataTableIO(String path)
     {
          System.out.println("Reading file...");

          this.path = path;

          //Must be initialized outside a codeblock
          tableString = null;
          Scanner inputScanner = null;
          Scanner scanNumOfLines = null;

          //Designate the input stream and catch errors
          try
          {
               inputScanner = new Scanner(new FileInputStream(path));
               scanNumOfLines = new Scanner(new FileInputStream(path));
          }
          catch(FileNotFoundException e)
          {
               System.out.println("No accessible file @" + path + " found.");
               System.exit(0);
          }

          /*
           * For the following, the scanNumOfLines is used.
           * Scanner can't reset it's position...
           * We need the other scanner object to actually read the table.
           */
          String[] headingStringArray = scanNumOfLines.nextLine().split(",");    //Split the first row into
headings
          int numRows, numColumns;

          numColumns = headingStringArray.length;
          numRows = 1;                                                    //The heading row has already
been used
          while(scanNumOfLines.hasNextLine())
          {
               numRows++;
               scanNumOfLines.nextLine();
          }
          scanNumOfLines.close();

          /*
           * Now the other scanner object can be used.
           * We can create the data table from a tab or comma delimited file.
```

```
          */
         this.tableString = new String[numRows][numColumns];

         inputScanner.useDelimiter(",|\\n");
         for(int i=0; i<this.tableString.length; i++)
         {
              for(int j=0; j<this.tableString[i].length; j++)
              {
                   this.tableString[i][j] = inputScanner.next();
              }
         }
         inputScanner.close();

    }//End of constructor


//-----------------------------------------------------METHODS-----------------------------------------
-------------


     /**
      * Prints the table of strings stored in DataTableIO to the console
      */
     public void printTable()
     {
          for(int i=0; i<this.tableString.length; i++)
          {
               for(int j=0; j<this.tableString[i].length; j++)
               {
                    System.out.printf("%-5.5s\t", this.tableString[i][j]);
               }
               System.out.println();
          }
     }//End of printTable method


     /**
      * Adds a new row to the bottom of an existing 2-dimensional table of Strings.
      * @param newRow an array of strings to be added to the bottom of the table
      * @throws IncorrectArraySize if the number of columns in the new row doesn't match the number of
columns in the old table.
      */
     public void appendTable(String[] newRow) throws IncorrectArraySize
     {
         //Throw error if the array doesn't have the right amount of columns
          if(newRow.length != this.tableString[0].length)
               throw new IncorrectArraySize("Check the number of columns in the array you entered." +
                                            "\n It should match the number of columns in the table");

          //Deep copy the old table to the placeholder (copying a smaller table into a bigger one)
          String[][] newTableString = new String[this.tableString.length + 1][this.tableString[0].length];
          for(int i=0; i<this.tableString.length; i++)
          {
               //System.out.println(tableString.length + " " + tableString[1].length +"\t" + this.numRows +
" " + this.numColumns);
               for(int j=0; j<this.tableString[i].length; j++)
               {
                    newTableString[i][j] = this.tableString[i][j];
               }
          }

          //Add the new row as the bottom row of the format is table[finalRow].length
          for(int j=0; j<newTableString[newTableString.length - 1].length; j++)
               newTableString[newTableString.length - 1][j] = newRow[j];

          //Re-assign the instance variables. It has already been deep copied, there will be privacy leaks.
          this.tableString = newTableString;

          //Open the table csv @path
```

```
          PrintWriter tableWriter = null;
          try
          {
               System.out.println("Searching for file...");
               tableWriter = new PrintWriter(this.path);   //Should overwrite the existing file
          }
          catch(FileNotFoundException e)
          {
               System.out.println("No file found @" + this.path);
               System.exit(0);
          }

          //Print the table in csv
          System.out.println("Writing new table to " + this.path);
          for(int x=0; x<tableString.length; x++)
          {
               for(int y=0; y<tableString[x].length; y++)
               {
                    tableWriter.print(tableString[x][y]);
                    if(y == (tableString[x].length -1))      // If it is the last element, print a new line
                         tableWriter.println();
                    else
                         tableWriter.print(",");              //Otherwise print a comma
               }
          }
          tableWriter.close();

     }//End of appendTable


     /**
      * Gets a row from the table corresponding to a row index number.
      * @param rowNum the row number
      * @return a string array corresponding to the desired table row
      */
     public String[] extractRow(int rowNum)
     {
       String[] row = new String[this.tableString[0].length];
       for (int i=0;i<this.tableString[0].length;i++)
       {
         row[i] = this.tableString[rowNum][i];
       }
         return row;
     }


     /**
      * Gets a column from the table corresponding to a column index number.
      * @param columnNum the column number (starting at 1)
      * @return a String array containing the contents of that column
      */
     public String[] extractColumn(int columnNum)
     {

          String[] column = new String[this.tableString.length];

          for(int i=0; i<this.tableString.length; i++)
               column[i] = this.tableString[i][columnNum];

          return column;
     }


//-----------------------------------------------HOUSEKEEPING METHODS-----------------------------------
-------------


     /**
      * Returns a the held table in a 2D array of Strings.
```

```
     * @return A 2D array of Strings containing the data in the imported table.
     */
    public String[][] getTableString()
    {

         String[][] toReturn = new String[this.tableString.length][this.tableString[0].length];

         for(int i=0; i<this.tableString.length; i++)
         {
              for(int j=0; j<this.tableString[i].length; j++)
              {
                   toReturn[i][j] = this.tableString[i][j];
              }
         }

    return toReturn;
    }//End of getTableString


    /**
     * Returns the absolute path of the imported table.
     * @return A string representing the absolute path of the imported table
     */
    public String getPath()
    {
         return this.path;
    }


}//End of DataTableIO
```

11.1.6 FlashModel

```
import numericMethods.*;
import customExceptions.NotFlashable;

/**
 * Parent class for cases, just for the sake of compiling
 * Child classes consist of CaseOne, CaseTwo, and CaseThree
 */
public abstract class FlashModel
{
    public abstract FlashTank solveSystem() throws NotFlashable;


     }
```

11.1.7 FlashTank

```
/**
 * This class represents a flash distillation vessel and contains three stream objects for the inflow and
outflows.
 * Each stream is composed of an array of species. The outlet streams should usually be the same temperature
as the flash tank.
 * Notes: When flash tank temp is set, outlet stream temps are also set
  */
public class FlashTank implements Cloneable
{
    //  INSTANCE VARIABLES

    private HeatExchanger flashTankHeatExchanger;
    private Stream feedStream;
    private Stream vapourStream;
    private Stream liquidStream;

    private double flashTemp;
    private double pressure;
```

```
    private String name;


//-----------------------------------------------CONSTRUCTORS-----------------------------------------
-------------

    /**
      * Builds a flash tank out of its necessary components.
      * @param flashTankHeatExchanger The simple heat exchanger. Used to store the value of Q.
      * @param feedStream The inlet of the flash tank. Contains known species and compositions.
      * @param vapourStream The outlet vapour stream. For now it will be empty, but it will be solved in
downstream methods.
      * @param liquidStream The outlet liquid stream. For now it will be empty, but it will be solved in
downstream methods.
      * @param flashTemp The temperature of the flash tank. For case 1 and 3 it is known; for case 2 it will
be solved for.
      * @param pressure The operating pressure of the flash tank
      */
    public FlashTank(HeatExchanger flashTankHeatExchanger, Stream feedStream, Stream vapourStream, Stream
liquidStream, double flashTemp, double pressure)
    {

      this.flashTankHeatExchanger = flashTankHeatExchanger.clone();

      this.feedStream = feedStream.clone();
      this.vapourStream = vapourStream.clone();
      this.liquidStream = liquidStream.clone();

      this.flashTemp = flashTemp;
      this.pressure = pressure;

      this.name = "NoName";

    }


    /**
      * Same constructor as above but also has a name of the flash tank.
      * @param flashTankHeatExchanger The simple heat exchanger. Used to store the value of Q.
      * @param feedStream The inlet of the flash tank. Contains known species and compositions.
      * @param vapourStream The outlet vapour stream. For now it will be empty, but it will be solved in
downstream methods.
      * @param liquidStream The outlet liquid stream. For now it will be empty, but it will be solved in
downstream methods.
      * @param flashTemp The temperature of the flash tank. For case 1 and 3 it is known; for case 2 it will
be solved for.
      * @param pressure The operating pressure of the flash tank.
      * @param name The name of the flash tank.
      */
        public FlashTank(HeatExchanger flashTankHeatExchanger, Stream feedStream, Stream vapourStream,
Stream liquidStream, double flashTemp, double pressure, String name)
    {

      this.flashTankHeatExchanger = flashTankHeatExchanger.clone();

      this.feedStream = feedStream;
      this.vapourStream = vapourStream;
      this.liquidStream = liquidStream;

      this.flashTemp = flashTemp;
      this.pressure = pressure;

      this.name = name;

    }


    /**
     * The copy constructor.
```

```
     * @param toCopy The object that is copied.
     */
    public FlashTank(FlashTank toCopy)
    {

      this.flashTankHeatExchanger = toCopy.flashTankHeatExchanger.clone();

      this.feedStream = toCopy.feedStream.clone();
      this.vapourStream = toCopy.vapourStream.clone();
      this.liquidStream = toCopy.liquidStream.clone();

      this.flashTemp = toCopy.flashTemp;
      this.pressure = toCopy.pressure;

      this.name = toCopy.name;
    }//end of copy constructor


//-------------------------------------------------HOUSEKEEPING METHODS---------------------------------
-------------


    public Stream getVapourStream() {
         return this.vapourStream.clone();
    }
    public Stream getLiquidStream() {
        return this.liquidStream.clone();
    }
    public Stream getFeedStream() {
         return this.feedStream.clone();
    }
    public String getName() {
        return this.name;
    }
    public double getFlashTemp() {
         return this.flashTemp;
    }
    public double getPressure() {
        return this.pressure;
    }
    public HeatExchanger getFlashTankHeatExchanger() {
         return this.flashTankHeatExchanger.clone();
    }

    public void setFeedStream(Stream feedStream) {
        this.feedStream = feedStream.clone();
    }

    public void setVapourStream(Stream vapourStream) {
         this.vapourStream = vapourStream.clone();
    }

    public void setLiquidStream(Stream liquidStream) {
         this.liquidStream = liquidStream.clone();
    }

    public void setFlashTankHeatExchanger(HeatExchanger flashTankHeatExchanger) {
        this.flashTankHeatExchanger = flashTankHeatExchanger.clone();
    }

    public void setName(String name) {
        this.name = name;
    }

    /**
     * Also changes the temperatures of the outlet streams
     * @param flashTemp The updated flash temperature
     */
    public void setFlashTemp(double flashTemp) {
```

```
        this.flashTemp = flashTemp;

        //needs to set the outlet streams to the flash temp as well
        this.vapourStream.setTemperature(flashTemp);
        this.liquidStream.setTemperature(flashTemp);
    }

    public void setPressure(double pressure) {
        this.pressure = pressure;
    }

    public FlashTank clone()
    {
        return new FlashTank(this);
    }

    public String toString() {
        return ("Q= "+this.flashTankHeatExchanger.getQ() +
                "\n\nINLET\t" + this.feedStream.toString() +
                "\n\nVAPOUR OUTLET\t" + this.vapourStream.toString() +
                "\n\nLIQUID OUTLET\t" + this.liquidStream.toString());
    }


//---------------------------------------------DEEP HOUSEKEEPING METHODS---------------------------------
-----------


    /**
     * Sets the molar flow rate of the vapour stream outlet.
     * @param totalMolarFlow the molar flow rate of the vapour outlet in mol/s
     */
    public void setVapourStreamFlowRate(double totalMolarFlow){
        this.vapourStream.setTotalMolarFlow(totalMolarFlow);
    }

    /**
     * Sets the molar flow rate of the liquid outlet stream.
     * @param totalMolarFlow the molar flow rate of the liquid outlet in mol/s
     */
    public void setLiquidStreamFlowRate(double totalMolarFlow){
        this.liquidStream.setTotalMolarFlow(totalMolarFlow);
    }

    /**
     * Sets the mole fractions in the vapour outlet.
     * The deep copy is done in the Stream class.
     * @param yiArray an array of yi values
     */
    public void setVapourStreamMoleFractions(double[] yiArray) {
        this.vapourStream.setMoleFractions(yiArray);
    }

    /**
     * Sets the mole fractions in the liquid outlet.
     * The deep copy is done in the Stream class.
     * @param xiArray an array of xi values
     */
    public void setLiquidStreamMoleFractions(double[] xiArray) {
        this.liquidStream.setMoleFractions(xiArray);
    }

    /**
     * Sets the feed stream temperature, used only at the end of case 3
     * @param temperature temperature in K
     */
    public void setFeedStreamTemperature(double temperature) {
        this.feedStream.setTemperature(temperature);
    }
```

```
    /**
     * Sets the temperatures of both outlet streams
     * @param temperature temperature in K
     */
    public void setOutletTemperatures(double temperature) {
         this.vapourStream.setTemperature(temperature);
         this.liquidStream.setTemperature(temperature);
    }

    /**
     * Sets the pressure of both outlet streams
     * @param pressures Pressure in Pa
     */
    public void setOutletPressures(double pressures) {
         this.vapourStream.setPressure(pressure);
         this.vapourStream.setPressure(pressure);
    }

    /**
     * Sets the Q value for the heat exchanger.
     * @param Q the Q value in J/s
     */
    public void setHeatExchangerQ(double Q) {
         this.flashTankHeatExchanger.setQ(Q);
    }
}
```

11.1.8 Group1FlashTankSimulator

```
import java.util.Scanner;
import customExceptions.NotFlashable;
import java.io.*;
import java.text.DecimalFormat;


/**
 * The main method
 * */
public class Group1FlashTankSimulator {
     public static void main(String[] args) throws IOException {
          //For Scanner, IO
          Scanner inputs = new Scanner(System.in);
          DecimalFormat numbers = new DecimalFormat("####0.000");
          ChemicalPropertiesTable givens = new ChemicalPropertiesTable();
          String[][] listOfComponents;
          PrintWriter outputs = null;

          //For species properties
          Species[] species;
          int numberOfSpecies = 0;
          String[] speciesNames;
          int idNumber=0;

          //For stream properties
          Stream inletStream = new Stream();
          double[] moleFractions;
          double totalFlow=0;
          double[] componentFlows;

          //For operating conditions
          double feedTemperature=0;
          double flashTemperature=0;
          double flashPressure;
          double feedPressure;
          int idealOrNot = 2; //user will set it to 1 or 0
          HeatExchanger heater = new HeatExchanger();
          VLEModel vle= new IdealModel();
```

```
          FlashModel workingCase = new CaseOne(); // Use no-argument constructor as this will be overwritten
depending on the selected case

          //Welcome!
          System.out.println("Welcome to Group 1's flash tank simulator.");
          System.out.println("Please consult the attached CSV file regarding the species for this
simulator.");
          System.out.println("If you have your own species that aren't included please add them at the
bottom of the table.");
          System.out.println("Would you like to enter your data through the console? Answer 'yes' or 'no'.");

          //User is given the option to enter data through the console or through the .txt file
          boolean rightData = false;
          while (rightData == false)
          {
            String dataEntry = inputs.next();

            if(dataEntry.equals("no")) {
              System.out.println("Please enter your information in the 'ReadMe.txt' file. Save the file and
close it once completed.");
              System.out.println("Instructions for order of information can be found in the associated
'ReadMe.txt' file.");
              //Essentially pauses the program so the user can enter their data if they haven't already.
              System.out.println("Once your data is entered, type 'yes' in the interactions pane.");
              boolean letsGo = false;
              while (letsGo == false) {
                String yes = inputs.next();
                if (yes.equals("yes")){
                System.out.println("Thank you.");
                letsGo=true;
                }
                else System.out.println("Incorrect input. Type 'yes' once data is properly entered.");
              }

              //Open up the file. If it's not there for some reason, the IOException will be thrown
              Scanner txtInputs = new Scanner(new File("Inputs.txt"));

              //Need to determine which temperature is being provided.
              System.out.println("Are you trying to solve case 1, case 2, or case 3?");
              System.out.println("Answer with '1', 2', or '3'.");
              boolean rightCase = false;
              while (rightCase == false)
              {
                int caseNumber = inputs.nextInt();
                if ((caseNumber != 1) && (caseNumber != 2) && (caseNumber != 3))
                  System.out.println("Incorrect number entered. Try again.");
                else if (caseNumber == 1) {
                  feedTemperature = txtInputs.nextDouble();
                  System.out.println("Feed Temperature: " + feedTemperature);
                  System.out.println("Flash Temperature: " + feedTemperature); //Isothermal case for case 1
                  flashTemperature = feedTemperature;
                  rightCase = true;
                }
                else if (caseNumber == 2) {
                  feedTemperature = txtInputs.nextDouble();
                  System.out.println("Temperature: " + feedTemperature); //Only feed temperature is given
for case 2
                  rightCase = true;
                }
                else if (caseNumber == 3) {
                  flashTemperature = txtInputs.nextDouble();
                  System.out.println("Temperature: " + flashTemperature); //Only flash temperature is given
for case 3
                  rightCase = true;
                }
              }
              //Fill in more operating conditions
              feedPressure = txtInputs.nextDouble();
              System.out.println("feedPressure: " + feedPressure);
```

```
            flashPressure = txtInputs.nextDouble();
            System.out.println("flashPressure: " + flashPressure);
            numberOfSpecies = txtInputs.nextInt();
            System.out.println("Number of species: " + numberOfSpecies);
            species = new Species[numberOfSpecies];

            //Get species and flowrate data
            for (int i=0;i<numberOfSpecies;i++)
            {
              idNumber = txtInputs.nextInt();
              System.out.println("ID #" + (i+1) + ": " + idNumber);
              if ((idNumber == 5) || (idNumber == 11))
                species[i] = new VapourSpecies(givens.extractRow(idNumber+1));
              else species[i] = new LiquidSpecies(givens.extractRow(idNumber+1));
            }
            componentFlows = new double[numberOfSpecies];
            for (int i=0;i<numberOfSpecies;i++)
            {
              componentFlows[i]=txtInputs.nextDouble();
              System.out.println("Flow " + (i+1) + ": " + componentFlows[i]);
            }

            //Construct all 3 streams. Outlets are constructed as empty for the purpose of being solved
            inletStream = new Stream(species, componentFlows, feedPressure, feedTemperature);
            Stream vapourStream = new Stream();
            Stream liquidStream = new Stream();

            //Construct flash tank
            FlashTank flashTank = new FlashTank(heater, inletStream, vapourStream, liquidStream,
flashTemperature, flashPressure);

            //Figure out ideal or non-ideal.
            //Ideal uses Raoult's Law while non-ideal uses Peng-Robinson EOS
            idealOrNot = txtInputs.nextInt();
            if (idealOrNot == 1)
              vle = new IdealModel();
            else vle = new PRModel();

            //Determine which case is being solved
            //In line with what user specified
            if (feedTemperature == flashTemperature) {
              workingCase = new CaseOne(flashTank, vle);
            }
            else if (flashTemperature == 0) {
              workingCase = new CaseTwo(flashTank, vle);
            }
            else if (feedTemperature == 0) {
              workingCase = new CaseThree(flashTank, vle);
            }
            rightData = true;
            txtInputs.close();
          } //End of file input
          else if (dataEntry.equals("yes")) //user enters data through the interactions pane
          {
            //First: Determine which case is being solved
            boolean correctSelection = false;
            while (correctSelection == false) {
              System.out.println("Our simulator will help you solve one of three cases.");
              System.out.println("Is your system adiabatic? Type 'yes' or 'no'.");
              String adiabaticOrNot = inputs.next();

              if (adiabaticOrNot.equals("yes")) {
                boolean correctChoice = false;
                while (correctChoice == false) {
                  System.out.println("Do you have the feed or flash temperature? Type 'feed' or 'flash'");
                  String feedOrFlash = inputs.next();
                  if (feedOrFlash.equals("feed")) {
                    //This corresponds to Case 2.
                    System.out.println("Please enter the feed temperature in K.");
```

```
                feedTemperature = inputs.nextDouble();
                correctChoice = true;
              }
              else if (feedOrFlash.equals("flash")) {
                //This corresponds to Case 3.
                System.out.println("Please enter the flash temperature in K.");
                flashTemperature = inputs.nextDouble();
                correctChoice = true;
              }
            }

            correctSelection = true;
          }

          else if (adiabaticOrNot.equals("no")) {
            //This corresponds to Case 1.
            System.out.println("Enter the constant operating temperature in K.");

            double isoTemp = inputs.nextDouble();
            feedTemperature = isoTemp;
            flashTemperature = isoTemp;
            correctSelection = true;
          }
        }

        //Pressure is the only operating variable that is always defined
        System.out.println("Please enter the feed pressure in Pa.");
        feedPressure = inputs.nextDouble();
        System.out.println("Please enter the flash tank pressure in Pa. Note that it must be less than
the feed pressure.");
        flashPressure = inputs.nextDouble();

        //Giving the user the ID of each species based on the CSV file
        listOfComponents = givens.extractIdentities();
        System.out.println("Species ID numbers and names: ");
        for (int i=0;i<listOfComponents.length;i++) {
          System.out.println(listOfComponents[i][0] + "\t" + listOfComponents[i][1]);
        }

        //Determine the number of species in the system
        boolean correctNumberOfSpecies = false;
        while (correctNumberOfSpecies == false) {
          System.out.println("Please enter the number of species");
          numberOfSpecies = inputs.nextInt();
          if ((numberOfSpecies > listOfComponents.length) || (numberOfSpecies <= 0))
            System.out.println("Incorrect number of species entered.");
          else correctNumberOfSpecies = true;
        }

        //components defined based on number of species to be entered
        moleFractions = new double[numberOfSpecies];
        speciesNames = new String[numberOfSpecies];
        species = new Species[numberOfSpecies];

        //Determine which species are being studied
        for (int i=0;i<numberOfSpecies;i++) {
          boolean correctID = false;
          while (correctID == false) {
            System.out.println("From the list of species above, please enter the ID of species " +
(i+1));
            idNumber = inputs.nextInt();
            if ((idNumber <= 0) || (idNumber > listOfComponents.length))
              System.out.println("Incorrect species ID entered.");
            else correctID = true;
          }

          //Ethane (ID 5) and nitrogen (ID 11) are considered non-condensible vapours in our simulation.
          //All other compounds will enter the feed as liquids
          if ((idNumber == 5) || (idNumber == 11))
```

```
                species[i] = new VapourSpecies(givens.extractRow(idNumber+1));
              else species[i] = new LiquidSpecies(givens.extractRow(idNumber+1));

              speciesNames[i]=listOfComponents[idNumber+1][1]; //Get the name of the species from the list
of components
            }

            //User can input total flows or component flows
            //If the total flow is given, then the mole fractions must also be given
            //Individual flow rates are then determined in the Stream's constructor
            int typeOfFlow=2;
            boolean properFlow = false;
            while (properFlow == false) {
              System.out.println("Do you have the total flow rate or the flow rate of each component?");
              System.out.println("Type '0' for total or '1' for components");
              typeOfFlow = inputs.nextInt();
              if (typeOfFlow == 0) {
                double sum = 1;
                boolean correctTotalFlow = false;
                while (correctTotalFlow == false) {
                  System.out.println("Enter the total flow rate in mol/s:");
                  totalFlow=inputs.nextDouble();
                  if (totalFlow <= 0)
                    System.out.println("Invalid flowrate entered.");
                  else correctTotalFlow = true;
                }

                for (int i=0;i<numberOfSpecies-1;i++) {
                  boolean correctMoleFraction = false;
                  while (correctMoleFraction == false) {
                    System.out.println("Enter the mole fraction of " + speciesNames[i]);
                    moleFractions[i]=inputs.nextDouble();
                    if ((moleFractions[i] > 1) || (moleFractions[i] <= 0))
                      System.out.println("Incorrect mole fraction inputted.");
                    else correctMoleFraction = true;
                  }
                  sum -= moleFractions[i]; //final mole fraction is determined from the others
                }
                moleFractions[numberOfSpecies-1] = sum;
                System.out.println("Final mole fraction: " + sum);
                inletStream = new Stream(species, totalFlow, moleFractions, feedPressure, feedTemperature);
                properFlow=true;
              }
              //Individual flows are given. Total flow and mole fractions are calculated in the Stream's
constructor
              else if (typeOfFlow == 1) {
                componentFlows = new double[numberOfSpecies];
                for (int i=0;i<numberOfSpecies;i++) {
                  boolean correctFlow = false;
                  while (correctFlow == false) {
                    System.out.println("Enter the molar flow in mol/s of " + speciesNames[i]);
                    componentFlows[i]=inputs.nextDouble();
                    if (componentFlows[i] <=0) System.out.println("Invalid flowrate entered.");
                    else correctFlow = true;
                  }
                }
                inletStream = new Stream(species, componentFlows, feedPressure, feedTemperature);
                properFlow=true;
              }
            }

            //Create empty outlet streams to be solved
            Stream vapourStream = new Stream();
            Stream liquidStream = new Stream();

            //With all the givens, we can construct the reactor
            FlashTank flashTank = new FlashTank(heater, inletStream, vapourStream, liquidStream,
flashTemperature, flashPressure);
```

```
    //Need to determine if we have an ideal case or non-ideal case
    //Raoult's Law for ideal case, Peng-Robinson EOS for non-ideal case
    boolean rightInput = false;
    while (rightInput == false) {
      System.out.println("Are we working with ideal solutions? Type 'yes' or 'no'");
      String vlePicker = inputs.next();
      if (vlePicker.equals("yes")) {
        vle = new IdealModel();
        idealOrNot = 1;
        rightInput = true;
      }
      else if (vlePicker.equals("no")) {
        vle = new PRModel();
        idealOrNot = 0;
        rightInput = true;
      }
    }

    //Now we must determine which case we are working with
    if (feedTemperature == flashTemperature) {
      workingCase = new CaseOne(flashTank, vle);
    }
    else if (flashTemperature == 0) {
      workingCase = new CaseTwo(flashTank, vle);
    }
    else if (feedTemperature == 0) {
      workingCase = new CaseThree(flashTank, vle);
    }

    rightData = true;
  }

}//End of user input

//Solve the system
//Prints to the interactions pane and to the .txt file
//If the system can't be solved, neither action is performed
  try
{
  outputs = new PrintWriter(new FileOutputStream("SolvedSystem.txt"));
}
catch (FileNotFoundException e)
{
  System.out.println("Error opening output file.");
}


FlashTank solved = null;
String fileName = null;
//Ask the user if they want to name the results file
//If not, a generic name will be given
try {
  solved = workingCase.solveSystem();
  System.out.println("Would you like to name the file of your simulation?");
  System.out.println("Type 'yes' or 'no'");
  boolean yesOrNo = false;
  while (yesOrNo == false){
    String option = inputs.next();
    if (option.equals("yes")){
      System.out.println("Enter the file name with .txt at the end.");
      fileName = inputs.next();
      yesOrNo=true;
    }
    else if (option.equals("no")) {
      fileName = "SolvedSystem.txt";
      yesOrNo=true;
    }
    else System.out.println("Invalid answer. Try again.");
  }
```

```
            System.out.println(solved);
             try {
                  outputs = new PrintWriter(new FileOutputStream(fileName));
                }
                catch (FileNotFoundException e)
                {
                  System.out.println("Error opening output file.");
                }

            if (idealOrNot == 1)
              outputs.println("Solved using ideal solutions");
            else if (idealOrNot == 0)
              outputs.println("solved using non-ideal solutions");
            outputs.println("");
            outputs.println("Q = " + numbers.format(solved.getFlashTankHeatExchanger().getQ()) + " J/s");
            outputs.println("");
            outputs.println("INLET");
            outputs.println("");
            outputs.println("Total               Molar               Flow:               "               +
numbers.format(solved.getFeedStream().getTotalMolarFlow()) + " mol/s");
            outputs.println("Temperature: " + numbers.format(solved.getFeedStream().getTemperature()) + "
K");
            outputs.println("Pressure: " + solved.getFeedStream().getPressure() + " Pa");
            for (int i=0;i<solved.getFeedStream().getSpecies().length;i++)
            {
              outputs.println("Species " + (i+1) + " = " + solved.getFeedStream().getSpecies()[i].getName());
              outputs.println("\tMolar Flow = " + numbers.format(solved.getFeedStream().getMolarFlows()[i])
+ " mol/s");
              outputs.println("\tMole               Fraction               =               "               +
numbers.format(solved.getFeedStream().getMoleFractions()[i]));
            }
            outputs.println("");
            outputs.println("VAPOUR OUTLET");
            outputs.println("");
            outputs.println("Total               Molar               Flow:               "               +
numbers.format(solved.getVapourStream().getTotalMolarFlow()) + " mol/s");
            outputs.println("Temperature: " + numbers.format(solved.getVapourStream().getTemperature()) + "
K");
            outputs.println("Pressure: " + solved.getVapourStream().getPressure() + " Pa");
            for (int i=0;i<solved.getFeedStream().getSpecies().length;i++)
            {
              outputs.println("Species       "       +       (i+1)       +       "       =       "       +
solved.getVapourStream().getSpecies()[i].getName());
              outputs.println("\tMolar               Flow               =               "               +
numbers.format(solved.getVapourStream().getMolarFlows()[i]) + " mol/s");
              outputs.println("\tMole               Fraction               =               "               +
numbers.format(solved.getVapourStream().getMoleFractions()[i]));
            }
            outputs.println("");
            outputs.println("LIQUID OUTLET");
            outputs.println("");
            outputs.println("Total               Molar               Flow:               "               +
numbers.format(solved.getLiquidStream().getTotalMolarFlow()) + " mol/s");
            outputs.println("Temperature: " + numbers.format(solved.getLiquidStream().getTemperature()) + "
K");
            outputs.println("Pressure: " + numbers.format(solved.getLiquidStream().getPressure()) + " Pa");
            for (int i=0;i<solved.getFeedStream().getSpecies().length;i++)
            {
              outputs.println("Species       "       +       (i+1)       +       "       =       "       +
solved.getLiquidStream().getSpecies()[i].getName());
              outputs.println("\tMolar               Flow               =               "               +
numbers.format(solved.getLiquidStream().getMolarFlows()[i]) + " mol/s");
              outputs.println("\tMole               Fraction               =               "               +
numbers.format(solved.getLiquidStream().getMoleFractions()[i]));
            }
            System.out.println("The solved system has been saved to the file " + "'" + fileName + "'.");
            System.out.println("Thank you!");

          }
```

```
          catch (NotFlashable e) {
            System.out.println(e.getMessage());
          }
          finally {
            inputs.close();
            outputs.close();
          }

     }//End of main

}//End of class
```

11.1.9 HeatExchanger

```
/**
 * Very simple heat exchanger with a Q value in J/s.
 */
public class HeatExchanger implements Cloneable
{


     private double Q;


//---------------------------------------CONSTRUCTORS-----------------------------------------------------
------------

     /**
      * Simple Heat Exchanger.
      * No Q is set as it will be calculated later.
      */
     public HeatExchanger() {
          this.Q = 0;
     }


     /**
      * Simple Heat Exchanger.
      * @param Q the heat supplied/removed
      */
     public HeatExchanger(double Q) {
          this.Q = Q;
     }


     /**
      * The copy constructor
      * @param heatExchanger The object to be copied.
      */
     public HeatExchanger(HeatExchanger heatExchanger) {
          this.Q = heatExchanger.Q;
     }


//---------------------------------------HOUSEKEEPING METHODS-------------------------------------------
------------


     public double getQ() {
          return this.Q;
     }
     public void setQ(double Q) {
          this.Q = Q;
     }
     public HeatExchanger clone() {
          return new HeatExchanger(this);
     }
}
```

11.1.10 IdealModel

```
import customExceptions.NotFlashable;

/**
 * Calculates the K values for each species with Psat/P (Raoult's Law)
 */
public class IdealModel extends VLEModel
{

     /**
      * Calculates the K values for each species with Psat/P (Raoult's Law).
      * @param flashTank The flash tank being solved.
      * @return The ideal K values for each species.
      */
     public double[] calculateKi(FlashTank flashTank)
     {

          double[] kiArray = new double[flashTank.getFeedStream().getSpecies().length];

          for (int i=0; i<flashTank.getFeedStream().getSpecies().length; i++)
          {
               //calculate vapour pressure at flash tank temperature.
               kiArray[i]                                                                                    =
flashTank.getFeedStream().getSpecies()[i].calcVapourPressure(flashTank.getFlashTemp())/flashTank.getPressu
re();
          }

          for (int i=0; i<flashTank.getFeedStream().getSpecies().length; i++) {
               if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable()) //T>Tc
                    kiArray[i] = 1e30; //Just a really high number
          }

          return kiArray;
     }
}
```

11.1.11 LiquidSpecies

```
import java.util.Arrays;
import customExceptions.*; //To get the innappropriate boundaries exception


/**
 * Class representing a species in the liquid phase.
 * This class uses a liquid-only method of calculating species enthalpy.
 */
public class LiquidSpecies extends Species
{
//-----------------------------------------------CONSTRUCTORS-----------------------------------------
----------

     /**
      * The constructor. Calls the super constructor as there are no additional instance variables
      * @param physProperties an array of physical properties. Must be in the following order:
idNum,Name,Formula,numMolecules,Molar Mass,antoineA,antoineB,antoineC,Tmin,Tmax,Latent Heat(molar),Boiling
Temperature, liqheatC1, liqheaC2, liqheatC3, liqHeatC4, liqheatEquation, criticalTemp, criticalPress
      * */
     public LiquidSpecies(String[] physProperties)
     {
          super(physProperties);
     }//End of constructor


     /**
      * The copy constructor.
      * There are no new instance variables here, so it should be able to take a gas species and make it
into a liquid object
      * @param toCopy The object to be changed.
```

```
     */
     public LiquidSpecies(Species toCopy)
     {
          super(toCopy);
     }//End of Copy Constructor


     /**
      * The default constructor.
      */
     public LiquidSpecies()
     {
          super();
     }//End of default constructor


//-------------------------------------------------METHODS-------------------------------------------------
------------


     /**
      * Calculates the enthalpy of the liquid species at a given temperature using the reference temp.
      * Will return 0 if the species is not condensable.
      * * @param temp the temperature in K
      * @return the enthalpy in J/mol
      */
     public double calcEnthalpy(double temp)
     {
          double enthalpy = 0.0;

          if(super.getIsCondensable() == false)
               enthalpy = 0.0;
          else
               enthalpy = integrateLiqHeatCapacity(super.refTemp, temp);

          return enthalpy; //J/mol
     }


     /**
      * @return a new LiquidSpecies object
      */
     public LiquidSpecies clone()
     {
          return new LiquidSpecies(this);
     }
}//End of LiquidSpecies class
```

11.1.12 PRModel

```
import numericMethods.CubicEquationSolver;
import customExceptions.NotFlashable;
/**
 * Calculates Ki Values using the Peng-Robinson Equations of State
 * Requires cubic equations to be solved
 */
public class PRModel extends VLEModel
{
     private FlashTank flashTank;
     public static final double R = 8.314;

     /**
      * The method for determining Ki values.
      * Calculates Ki values based off of fugacity coefficients.
      * The only method that should be public. Other methods are intermediate steps in determining Ki values.
      * @param flashTank Is the flash tank to be solved.
      * @return The ki values for each species that is used to determine non-ideal VLE.
      * */
     public double[] calculateKi(FlashTank flashTank) throws NotFlashable {
```

```
          this.flashTank = flashTank;//it has to be shared between a ton of methods, but we don't want to
use constructor

          double A_vapour = calculateA(this.flashTank.getVapourStream().getMoleFractions());
          double B_vapour = calculateB(this.flashTank.getVapourStream().getMoleFractions());
          double A_liquid = calculateA(this.flashTank.getLiquidStream().getMoleFractions());
          double B_liquid = calculateB(this.flashTank.getLiquidStream().getMoleFractions());
          double[] Z_vapour = calculateZ(A_vapour,B_vapour); //For the liquid, the value of Z will be the
smallest root from the cubic equation
          double[] Z_liquid = calculateZ(A_liquid,B_liquid); //For the vapour, the value of Z will be the
largest root from the cubic equation

          double[]      liquidFugacityCoefficients      =      calculateFugacityCoefficients(Z_liquid[0],
this.flashTank.getLiquidStream().getMoleFractions());
          double[]      vapourFugacityCoefficients      =      calculateFugacityCoefficients(Z_vapour[1],
this.flashTank.getVapourStream().getMoleFractions());

          double[]                                              ki                                          =
calculateFugacityCoefficientRatios(liquidFugacityCoefficients,vapourFugacityCoefficients);

           for (int i=0; i<this.flashTank.getFeedStream().getSpecies().length; i++) {
               if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable())
                    ki[i] = 1e30; //Just a really high number
          }

          return ki;
     }


//-----------------------------------------Calculating A-----------------------------------------

     /**
      * Need to calculate an 'A' term for both liquid and vapour phases.
      * @param moleFractions The species moleFractions will be from either the liquid phase (xi) or the
vapour phase (yi).
      * @return The value of A for either the liquid or vapour phase. Dependent on all species in question.
      * */
     private double calculateA(double[] moleFractions) {
          double alphaA = calculateAlphaA(moleFractions);
          double pressure = flashTank.getPressure();
          double temperature = flashTank.getFlashTemp();

          return alphaA*pressure/(Math.pow(this.R, 2)*Math.pow(temperature, 2));
     }

      /**
      * Need to calculate an 'alphaA' term for both liquid and vapour phases.
      * @param moleFractions The species moleFractions will be from either the liquid phase (xi) or the
vapour phase (yi).
      * Individual values of alphaA are combined to determine the overall alphaA parameter.
      * @return The value of alphaA for either the liquid or vapour phase. Dependent on all species in
question.
      * */
     private double calculateAlphaA(double[] moleFractions) {
          //Equation 11.4a
          //Does summation include when i = j?
          double alphaA = 0;
          for (int i =0; i<moleFractions.length;i++) {
               for (int j = 0; j<this.flashTank.getFeedStream().getSpecies().length;j++) {

                 double alphaAi = calculateAlphaPure(i)*calculateaPure(i);
                 double alphaAj = calculateAlphaPure(j)*calculateaPure(j);

                 double kij = calculateKij(i,j);

                 double alphaAij = Math.sqrt(alphaAi*alphaAj)*(1-kij);

                 alphaA += moleFractions[i]*moleFractions[j]*alphaAij;
               }
```

```
          }

          return alphaA;
     }

     /**
      * For the purpose of our simulator, this will always return 0.
      * @param i The index number for the first species in the for loop from calculateKi.
      * @param j The index number for the second species in the for loops from calculateKi.
      * @return The binary interaction parameter, which will simply be set to 0.
      **/
     private double calculateKij(int i, int j) {

       return 0;
     }

     /**
      * The alpha value is dependent on the species' accentric factor and critical temperature
      * @param i The index number for the first species in the for loop from calculateKi.
      * @return The alpha value, which is calculated for an individual species.
      **/
     private double calculateAlphaPure(int i) {
          double w = flashTank.getFeedStream().getSpecies()[i].getAccentricFactor();
          double                                    criticalTemperature                                        =
this.flashTank.getFeedStream().getSpecies()[i].getCriticalTemperature();

          return Math.pow(1+(0.37464 + 1.54226*w-0.26992*Math.pow(w, 2))
                               *(1-Math.sqrt(this.flashTank.getFlashTemp()/criticalTemperature)),2);
     }

      /**
      * The a value is dependent on the species' critical pressure and critical temperature.
      * @param i The index number for the first species in the for loop from calculateKi.
      * @return The a value, which is calculated for an individual species.
      **/
     private double calculateaPure(int i) {

       double criticalPressure = flashTank.getFeedStream().getSpecies()[i].getCriticalPressure();
       double criticalTemperature = flashTank.getFeedStream().getSpecies()[i].getCriticalTemperature();

       return 0.45724*Math.pow(PRModel.R,2)*Math.pow(criticalTemperature, 2)/criticalPressure;
     }


//-----------------------------------------------------Calculating B-------------------------------------


      /**
      * Need to calculate an 'A' term for both liquid and vapour phases.
      * @param moleFractions The species moleFractions will be from either the liquid phase (xi) or the
vapour phase (yi).
      * @return The value of B for either the liquid or vapour phase. Dependent on all species in question.
      * */
     private double calculateB(double[] moleFractions) {

       double pressure = flashTank.getPressure();
       double b = calculatebMixture(moleFractions);
       double temperature = flashTank.getFlashTemp();

       return b*pressure/(PRModel.R*temperature);
     }

     /**
      * The 'B' term is dependent on the 'b' term, which is based on the mixture of species.
      * @param moleFractions The species moleFractions will be from either the liquid phase (xi) or the
vapour phase (yi).
      * @return The value of b for either the liquid or vapour phase. Dependent on all species in question.
      * */
     private double calculatebMixture(double[] moleFractions) {
```

```
          double bMix=0;
          for (int i=0;i<flashTank.getFeedStream().getSpecies().length;i++) {
               double zi = moleFractions[i]; //Here zi is not the overall mole fraction, but that of the
phase (xi or yi)
               double bi = calculatebPure(i);

               bMix += zi*bi;
          }

          return bMix;
     }

     /**
      * The 'b' term for the mixture is dependent on the 'b' term of each individual species.
      * @param i The ID number of the species being calculated.
      * @return The value of b for either the specific species.
      * */
     private double calculatebPure(int i) {
          double                                    criticalTemperature                                    =
this.flashTank.getFeedStream().getSpecies()[i].getCriticalTemperature();
          double criticalPressure = this.flashTank.getFeedStream().getSpecies()[i].getCriticalPressure();


          return 0.07780*PRModel.R*criticalTemperature/criticalPressure;
     }


//-----------------------------------------------------Calculating Z---------------------------------------
--

     /**
      * Solves the cubic equation of state for the compressibility factor.
      * @param A The A value for either the liquid or vapour phase.
      * @param B The B value for either the liquid or vapour phase.
      * @return a sorted array of Z values.
      */
     private double[] calculateZ(double A, double B) throws NotFlashable {

       double a = -(1-B);
       double b = (A-2*B-3*B*B);
       double c = -(A*B-B*B-Math.pow(B,3));
       double[] constants = {a,b,c};

       return (new CubicEquationSolver()).solveCubicEquation(constants);
     }


//----------------------------------Calculating Fugacity Coefficients--------------------------------


     /**
      * Determines the fugacity coefficients for either the liquid or vapour phase.
      * @param Z The compressibility factor for either the liquid or vapour phase.
      * @param moleFractions the xi or yi values.
      * @return a sorted array of Z values.
      */
     private double[] calculateFugacityCoefficients(double Z, double[] moleFractions) {

       double[] fugacityCoefficients = new double[this.flashTank.getFeedStream().getSpecies().length];

       for (int i =0; i<fugacityCoefficients.length; i++) {

         double A = calculateA(moleFractions);
         double AA = calculateAA(i, moleFractions);
         double B = calculateB(moleFractions);
         double BB = calculateBB(i, moleFractions);

         double lnFugacity = BB*(Z-1)-Math.log(Z-B)-A/(2*Math.sqrt(2)*B)*(AA-BB)
```

```
           *Math.log((Z+(Math.sqrt(2)+1)*B)/(Z-((Math.sqrt(2)-1)*B)));

         fugacityCoefficients[i] = Math.exp(lnFugacity);
       }

       return fugacityCoefficients;
     }

      /**
      * Needed for determining fugacity coefficients.
      * Calculated for each species and is dependent on the mixture alphaA value and the alphaA value of
the species with every other species.
      * @param i The ID number of the species whose AA value is being calculated.
      * @param moleFractions the xi or yi values.
      * @return The AA value for the specific species in the liquid or vapour phase.
      */
     private double calculateAA(int i, double[] moleFractions) {

          double alphaAijSum = 0;
          double alphaAi = calculateAlphaPure(i)*calculateaPure(i);

          for (int j = 0; j<flashTank.getFeedStream().getSpecies().length; j++) {

            double alphaAj = calculateAlphaPure(j)*calculateaPure(j);

            double kij = calculateKij(i,j);

            double alphaAij = Math.sqrt(alphaAi*alphaAj)*(1-kij);
            alphaAijSum += moleFractions[j]*alphaAij;
          }

          double alphaAMixture = calculateAlphaA(moleFractions);

          return (2/alphaAMixture)*alphaAijSum;
     }

       /**
      * Needed for determining fugacity coefficients.
      * Calculated for each species and is dependent on the mixture b value and the b value of the species
with every other species.
      * @param i The ID number of the species whose AA value is being calculated.
      * @param moleFractions the xi or yi values.
      * @return The BB value for the specific species in the liquid or vapour phase.
      */
     private double calculateBB(int i, double[] moleFractions) {

       double bi = calculatebPure(i);
       double bm = calculatebMixture(moleFractions);

       return bi/bm;
     }


//-----------------------------------------Calculating Coeff ratios-------------------------------


     /**
      * The actual Ki calculation!
      * Calculated for each species based on the liquid and vapour fugacity coefficients.
      * @param liquidFugacityCoefficients The fugacity coefficients of each species in the liquid phase.
      * @param vapourFugacityCoefficients The fugacity coefficients of each species in the vapour phase.
      * @return The Ki values for each species in an array.
      */
     private  double[]  calculateFugacityCoefficientRatios(double[]  liquidFugacityCoefficients,  double[]
vapourFugacityCoefficients) {

          double[] ki = new double[liquidFugacityCoefficients.length];

          for (int i = 0; i<liquidFugacityCoefficients.length; i++)
```

```
            ki[i] = liquidFugacityCoefficients[i]/vapourFugacityCoefficients[i];


          return ki;
     }
}
```

11.1.13 Species

```
import java.util.Arrays;
import customExceptions.*; //To get the innappropriate boundaries exception

/**
 * A molecular species with some of its associated properties and methods to calculate enthalpies, vapour
pressures etc
 * Everything is done in SI units.
 * Abstract class with child classes being LiquidSpecies and VapourSpecies
 */
public abstract class Species implements Cloneable
{

     public static final double R = 8.314; //the ideal gas constant J/mol*K
     public static final double refTemp = 273.15;

     //Basic properties
     private int idNum;
     private String name;
     private String formula;
     private int numMolecules;
     private double molarMass;  //kg/mol

     //Antoine's eq properties
     private double[] antoineConstants;
     private double minTemp, maxTemp;  //the min and max temps for antoines equation in K

     //Latent heat properties
     private double molarLatHeat;  //J/mol @ normal boiling temp
     private double normBoilTemp;  //K

     //to calculate liquid heat capacity at a given temp
     private double[] liqHeatCapConstants;
     private boolean isSpecial; //True if the special equation should be used

     //to calculate gas heat capacity at a given temp
     private double[] gasHeatCapConstants;

     //To for non-ideal calculations
     private double criticalTemperature;  //K
     private double criticalPressure;  //pa
     private double accentricFactor;  //unitless

     //
     private boolean isCondensable;


//----------------------------------------------CONSTRUCTORS-----------------------------------------------
-------------


     /**
      * Takes in an array of strings from the table as parameters.
      * @param physProperties an array of physical properties. Must be in the following order:
idNum,Name,Formula,numMolecules,Molar Mass,antoineA,antoineB,antoineC,Tmin,Tmax,Latent Heat(molar),Boiling
Temperature, liqheatC1, liqheaC2, liqheatC3, liqHeatC4, liqheatEquation, criticalTemp, criticalPress
      */
     public Species(String[] physProperties)
     {

         this.idNum = Integer.parseInt(physProperties[0]);
```

```
        this.name = physProperties[1];
        this.formula = physProperties[2];
        this.numMolecules = Integer.parseInt(physProperties[3]);
        this.molarMass = Double.parseDouble(physProperties[4]);

        this.isCondensable = Boolean.parseBoolean(physProperties[5]);

        this.antoineConstants = new double[3];
        for(int i=0; i<3; i++)
             this.antoineConstants[i] = Double.parseDouble(physProperties[i+6]);
        this.minTemp = Double.parseDouble(physProperties[9]);
        this.maxTemp = Double.parseDouble(physProperties[10]);

        this.molarLatHeat = Double.parseDouble(physProperties[11]);
        this.normBoilTemp = Double.parseDouble(physProperties[12]);

        this.liqHeatCapConstants = new double[5];
        for(int i=0; i<5; i++)
             this.liqHeatCapConstants[i] = Double.parseDouble(physProperties[i+13]);
        this.isSpecial = Boolean.parseBoolean(physProperties[18]);

        this.gasHeatCapConstants = new double[4];
        for (int i=0; i<4; i++)
             this.gasHeatCapConstants[i] = Double.parseDouble(physProperties[i+19]);

        this.criticalTemperature = Double.parseDouble(physProperties[23]);
        this.criticalPressure = Double.parseDouble(physProperties[24]);
        this.accentricFactor = Double.parseDouble(physProperties[25]);

    }//End of constructor


    /**
     * The copy constructor
     * @param toCopy the Species object to be copied
     */
    public Species(Species toCopy)
    {

         this.idNum = toCopy.idNum;
         this.name = toCopy.name;
         this.formula = toCopy.formula;
         this.numMolecules = toCopy.numMolecules;
         this.molarMass = toCopy.molarMass;

         this.antoineConstants = Arrays.copyOf(toCopy.antoineConstants, toCopy.antoineConstants.length);
         this.minTemp = toCopy.minTemp;
         this.maxTemp = toCopy.maxTemp;

         this.molarLatHeat = toCopy.molarLatHeat;
         this.normBoilTemp = toCopy.normBoilTemp;

         this.liqHeatCapConstants              =              Arrays.copyOf(toCopy.liqHeatCapConstants,
toCopy.liqHeatCapConstants.length);
         this.isSpecial = toCopy.isSpecial;

         this.gasHeatCapConstants              =              Arrays.copyOf(toCopy.gasHeatCapConstants,
toCopy.gasHeatCapConstants.length);

         this.criticalTemperature = toCopy.criticalTemperature;
         this.criticalPressure = toCopy.criticalPressure;
         this.accentricFactor = toCopy.accentricFactor;

         this.isCondensable = toCopy.isCondensable;

    }//End of copy constructor


    /**
```

```
     * The default constructor
     */
    public Species()
    {

        this.idNum = 0;
        this.name = "No name yet";
        this.formula = "No formula yet";
        this.numMolecules = 0;
        this.molarMass = 0.0;

        this.antoineConstants = null;
        this.minTemp = 0.0;
        this.maxTemp = 0.0;

        this.molarLatHeat = 0.0;
        this.normBoilTemp = 0.0;

        this.liqHeatCapConstants = null;
        this.isSpecial = false;

        this.gasHeatCapConstants = null;

        this.criticalTemperature = 0.0;
        this.criticalPressure = 0.0;
        this.accentricFactor = 0.0;

        this.isCondensable = true;

    }//End of default constructor


//-------------------------------------------------METHODS-------------------------------------------------
-------------


    /**
     * Returns the vapour pressure given a temperature using the Antoine Equation
     * @param temp the temperature in Kelvin
     * @return the saturation pressure in Pa
     */
    public double calcVapourPressure(double temp)
    {

        double pSat;
        double tempInC = temp - 273.15; //convert to C

        pSat =   (Math.pow(Math.E, antoineConstants[0] - (antoineConstants[1] / (antoineConstants[2] +
tempInC))));

        return pSat * 1000; //convert from kPa to Pa
    }


    /**
     * Returns the heat capacity ovintegrated over the given temperature interval using equations from
Perry's handbook.
     * @param tempStart the starting temperature used to calculate the integrated heat capacity in K
     * @param tempEnd the ending temperature used to calculate the integrated heat capacity in K
     * @return the integrated heat capacity in J/mol*K
     */
    protected double integrateLiqHeatCapacity(double tempStart, double tempEnd)
    {

        double iHeatCapacity = 0.0;

        //Calculate the heat capacity according to the necessary equations from Perry's handbook
        if(this.isSpecial)
        {
```

```
                System.out.println("This species (" + this.name + ") is special");

                double tStart = 1 - tempStart / this.criticalTemperature;
                double tEnd = 1 - tempEnd / this.criticalTemperature;
                iHeatCapacity   =   (Math.pow(this.liqHeatCapConstants[0],2)   *   (Math.log(Math.abs(tEnd))-
Math.log(Math.abs(tStart)))
                     + this.liqHeatCapConstants[1] * (tEnd - tStart)
                     -  this.liqHeatCapConstants[0]  *  this.liqHeatCapConstants[2]  *  (Math.pow(tEnd,2)  -
Math.pow(tStart,2))
                     -  this.liqHeatCapConstants[0]  *  this.liqHeatCapConstants[3]  *  (Math.pow(tEnd,3)  -
Math.pow(tStart,3)) / 3
                     - Math.pow(this.liqHeatCapConstants[2],2) * (Math.pow(tEnd,4) - Math.pow(tStart,4)) /
12
                     -  this.liqHeatCapConstants[2]  *  this.liqHeatCapConstants[3]  *  (Math.pow(tEnd,5)  -
Math.pow(tStart,5)) / 10
                     - Math.pow(this.liqHeatCapConstants[3],2) * (Math.pow(tEnd,6) - Math.pow(tStart,6)) /
30)
                     / (-1.0 / this.criticalTemperature); //this term comes from the substitution of t for T
           }
           else
           {
                iHeatCapacity = this.liqHeatCapConstants[0] * (tempEnd - tempStart)
                     + this.liqHeatCapConstants[1] * (Math.pow(tempEnd,2) - Math.pow(tempStart,2)) / 2
                     + this.liqHeatCapConstants[2] * (Math.pow(tempEnd,3) - Math.pow(tempStart,3)) / 3
                     + this.liqHeatCapConstants[3] * (Math.pow(tempEnd,4) - Math.pow(tempStart,4)) / 4
                     + this.liqHeatCapConstants[4] * (Math.pow(tempEnd,5) - Math.pow(tempStart,5)) / 5;
           }

           return iHeatCapacity / 1000; //need to convert to J/mol*K
      }


      /**
       * Calculates the enthalpy of the species at a given temperature compared to the reference enthalpy of
liquid at 273.15
       * gaseous species are ideal gasses.
       * @param temp the temperature in K
       * @return the enthalpy in J/mol
       */
      public abstract double calcEnthalpy(double temp);



//-----------------------------------------HOUSEKEEPING METHODS-----------------------------------------
--------------


      public int getIdNum() {
           return this.idNum;
      }

      public String getName() {
           return this.name;
      }

      public String getFormula() {
           return this.formula;
      }

      public int getNumMolecules() {
           return this.numMolecules;
      }

      public double getmolarMass() {
           return this.molarMass;
      }

      public double[] getAntoineConstants() {
           double[] constants = Arrays.copyOf(this.antoineConstants, this.antoineConstants.length);
```

```
     return constants;
}

public double getMinTemp()
{
     return this.minTemp;
}

public double getMaxTemp() {
     return this.maxTemp;
}

public double getMolarLatHeat() {
     return this.molarLatHeat;
}

public double getNormBoilTemp() {
     return this.normBoilTemp;
}

public double[] getLiqHeatCapConstants() {
     double[] constants = Arrays.copyOf(this.liqHeatCapConstants, this.liqHeatCapConstants.length);
     return constants;
}

public boolean getIsSpecial() {
     return this.isSpecial;
}

public double[] getGasHeatCapConstants() {
     return Arrays.copyOf(this.gasHeatCapConstants, this.gasHeatCapConstants.length);
}

public double getCriticalTemperature() {
     return this.criticalTemperature;
}

public double getCriticalPressure() {
     return this.criticalPressure;
}

public double getAccentricFactor() {
     return this.accentricFactor;
}

public boolean getIsCondensable() {
     return this.isCondensable;
}


/**
 * Changes the species to a different species.
 * You can't change only one parameter in a species.
 * @param physProperties a new species from the species table
 */
public void setSpecies(String[] physProperties) {

    this.idNum = Integer.parseInt(physProperties[0]);
    this.name = physProperties[1];
    this.formula = physProperties[2];
    this.numMolecules = Integer.parseInt(physProperties[3]);
    this.molarMass = Double.parseDouble(physProperties[4]);

    this.isCondensable = Boolean.parseBoolean(physProperties[5]);

    this.antoineConstants = new double[3];
    for(int i=0; i<3; i++)
         this.antoineConstants[i] = Double.parseDouble(physProperties[i+6]);
    this.minTemp = Double.parseDouble(physProperties[9]);
```

```
    this.maxTemp = Double.parseDouble(physProperties[10]);

    this.molarLatHeat = Double.parseDouble(physProperties[11]);
    this.normBoilTemp = Double.parseDouble(physProperties[12]);

    this.liqHeatCapConstants = new double[5];
    for(int i=0; i<5; i++)
         this.liqHeatCapConstants[i] = Double.parseDouble(physProperties[i+13]);
    this.isSpecial = Boolean.parseBoolean(physProperties[18]);

    this.gasHeatCapConstants = new double[4];
    for (int i=0; i<4; i++)
         this.gasHeatCapConstants[i] = Double.parseDouble(physProperties[i+19]);

    this.criticalTemperature = Double.parseDouble(physProperties[23]);
    this.criticalPressure = Double.parseDouble(physProperties[24]);
    this.accentricFactor = Double.parseDouble(physProperties[25]);

}


/**
 * Compares the instance variables of two species objects and returns TRUE if all equal.
 * @param toCompare an object of Species type
 * @return true if equals, false if not
 */
public boolean equals(Species toCompare)
{
     if(this.idNum == toCompare.idNum
             && this.name == toCompare.name
             && this.formula == toCompare.formula
             && this.numMolecules == toCompare.numMolecules
             && this.molarMass == toCompare.molarMass
             && Arrays.equals(this.antoineConstants, toCompare.antoineConstants)
             && this.minTemp == toCompare.minTemp
             && this.maxTemp == toCompare.maxTemp
             && this.molarLatHeat == toCompare.molarLatHeat
             && this.normBoilTemp == toCompare.normBoilTemp
             && Arrays.equals(this.liqHeatCapConstants, toCompare.liqHeatCapConstants)
             && this.isSpecial == toCompare.isSpecial
             && Arrays.equals(this.gasHeatCapConstants, toCompare.gasHeatCapConstants)
             && this.criticalTemperature == toCompare.criticalTemperature
             && this.criticalPressure == toCompare.criticalPressure
             && this.accentricFactor == toCompare.accentricFactor
             && this.isCondensable == toCompare.isCondensable)

          return true;
     else
          return false;
}//End of equals


/**
 * Prints all the instance variables.
 * @return a string of all the species info
 */
public String toString()
{
     return "Species idNum=" + idNum + "\nname=" + name + "\nformula="
          + formula + "\nnumMolecules=" + numMolecules + "\nmolarMass="
          + molarMass + "\nantoineConstants="
          + "\nisCondensable=" + isCondensable + "\n"
          + Arrays.toString(antoineConstants) + "\nminTemp=" + minTemp
          + "\nmaxTemp=" + maxTemp + "\nmolarLatHeat=" + molarLatHeat
          + "\nnormBoilTemp=" + normBoilTemp + "\nliqHeatCapConstants="
          + Arrays.toString(liqHeatCapConstants) + "\nisSpecial="
          + isSpecial + "\ngasHeatCapConstants=" + Arrays.toString(gasHeatCapConstants)
          + "\ncriticalTemperature=" + criticalTemperature
          + "\ncriticalPressure=" + criticalPressure
```

```
                + "\naccentricFactor=" + accentricFactor;
     }//End of tostring


     /**
      * @return new Species object
      */
     public abstract Species clone();
}
```

11.1.14 Stream

```
import java.util.Arrays;

/**
 * A class representing a stream.
 * Contains a set of species along with flow rates, temperature, and pressure.
 */
public class Stream implements Cloneable
{
     //Physical properties of each species
     private Species[] species;

     //Component breakdown
     private double[] molarFlows;
     private double[] moleFractions;
     private double totalMolarFlow;

     //Properties of stream
     private double temperature;
     private double pressure;


     //-------------------------------------------------CONSTRUCTORS-----------------------------------
---------------

     /**
      * Receies an array of @param Species objects, each with physical and thermodynamic properties.
      * Defined by amount of each component, temperature and pressure of system.
      * @param species Each species in the stream
      * @param molarFlows The molar flow of each species in the stream
      * @param pressure The pressure of the stream
      * @param temperature The temperature of the stream
      */
     public Stream(Species[] species, double[] molarFlows, double pressure, double temperature) {

          if (species.length != molarFlows.length) {
               System.out.println("Error: Array lengths don't match");
               System.exit(0);
          }
          this.species = new Species[species.length];
          this.molarFlows = new double[molarFlows.length];
          this.moleFractions = new double[molarFlows.length];
          for (int i=0;i<species.length;i++) {
               this.species[i] = species[i];
               this.molarFlows[i] = molarFlows[i];
          }

          this.temperature = temperature;
          this.pressure = pressure;
          this.totalMolarFlow = 0; //calculated once molar flows have been assigned


          //Total molar flow rate calculated from individual flow rates
          for (int i=0; i<molarFlows.length;i++) {
               this.totalMolarFlow = this.totalMolarFlow + molarFlows[i];
          }

          //Mole Fractions determined based on molar flow rates and total molar flow rate
```

```
          for (int i=0;i<molarFlows.length;i++) {
            this.moleFractions[i]=this.molarFlows[i]/this.totalMolarFlow;
          }

     } //End of constructor

     //If the user provides the total flow instead of component flows
      public Stream(Species[] species, double totalMolarFlow, double[] moleFractions, double pressure,
double temperature) {

          if(species.length != moleFractions.length) {
               System.out.println("Error: Array lengths don't match");
               System.exit(0);
          }
          this.species = new Species[species.length];

          this.moleFractions = new double[moleFractions.length];
          for (int i=0;i<species.length;i++) {
               this.species[i] = species[i];
               this.moleFractions[i] = moleFractions[i];
          }

          this.temperature = temperature;
          this.pressure = pressure;
          this.totalMolarFlow = totalMolarFlow;


          //Component molar flow rates calculated from total flow rate and component fractions
          this.molarFlows = new double[species.length];
          for (int i=0; i<this.molarFlows.length;i++) {
               this.molarFlows[i] = this.totalMolarFlow*this.moleFractions[i];
          }

     } //End of constructor

      /**
       * The default constructor
       * For creating empty streams (outlets) that will be solved
       * */
      public Stream() {
        this.species = new LiquidSpecies[0];

        //Component breakdown
        this.molarFlows = new double[0];
        this.moleFractions = new double[0];
        totalMolarFlow=0;

        //Properties of stream
        this.temperature=0;
        this.pressure=0;
      }

      /**
      * The copy constructor
      * @param streamToCopy an object of type Stream
      */
     public Stream(Stream streamToCopy) {

          this.temperature=streamToCopy.temperature;
          this.pressure=streamToCopy.pressure;
          this.totalMolarFlow=streamToCopy.totalMolarFlow;

          this.species = new Species[streamToCopy.species.length];
          this.molarFlows = new double[streamToCopy.molarFlows.length];
          this.moleFractions = new double[streamToCopy.moleFractions.length];

          for (int i=0;i<this.species.length;i++)
          {
               this.species[i] = streamToCopy.species[i].clone(); //will make new species objects
```

```
            this.molarFlows[i] = streamToCopy.molarFlows[i];
            this.moleFractions[i] = streamToCopy.moleFractions[i];
        }

    } //End of copy constructor


    //-------------------------------------------------METHODS-----------------------------------------
--------------


    /*
     * calculates the total enthalpy in J/mol of the stream using the streams temperature
     * @Return enthalpy of the feed stream in J/s
     */
    public double totalEnthalpy() {
        double totalEnthalpy;
        double totalMolarEnthalpy = 0;

        for (int i=0;i<this.species.length;i++) {
            totalMolarEnthalpy                =                totalMolarEnthalpy                +
this.moleFractions[i]*this.species[i].calcEnthalpy(this.temperature);
        }

        totalEnthalpy = totalMolarEnthalpy*this.totalMolarFlow;
        return totalEnthalpy;//J
    }


//-------------------------------------------------HOUSEKEEPING METHODS----------------------------------
--------------


    public Species[] getSpecies() {
        Species[] returnSpecies = new Species[this.species.length];

        for (int i = 0; i< species.length; i++) {
            returnSpecies[i] = this.species[i].clone();
        }

        return returnSpecies;
    }
    public double[] getMolarFlows() {
        return (Arrays.copyOf(this.molarFlows, this.molarFlows.length));
    }
    public double[] getMoleFractions() {
        return (Arrays.copyOf(this.moleFractions, this.moleFractions.length));
    }
    public double getTotalMolarFlow() {
        return this.totalMolarFlow;
    }
    public double getTemperature() {
        return this.temperature;
    }
    public double getPressure() {
        return this.pressure;
    }


    public void setSpecies(Species[] species) {
        for (int i=0; i<species.length; i++) {
            this.species[i] = species[i].clone();
        }
    }
    /**
     * Sets the molar flow rates of each species, and recalculates the corresponding total molar flow and
mole fractions
     * @param molarFlows The incoming array of molar flows
     */
```

```
    public void setMolarFlow(double[] molarFlows) {
        this.totalMolarFlow = 0;
        for(int i=0;i<molarFlows.length;i++) {
            this.molarFlows[i]=molarFlows[i];
            this.totalMolarFlow = this.totalMolarFlow + molarFlows[i];
        }
        for(int i=0; i<molarFlows.length;i++) {
            this.moleFractions[i] = this.molarFlows[i]/this.totalMolarFlow;
        }
    }

    /**
     * Sets the mole fraction of each species and recalculates the molar flows.
     * @param moleFractions The incomming array of mole fractions
     */
    public void setMoleFractions(double[] moleFractions) {
        this.moleFractions = Arrays.copyOf(moleFractions, moleFractions.length);
        for(int i=0; i<this.species.length; i++)
            this.molarFlows[i] = this.totalMolarFlow * this.moleFractions[i];
    }
    /**
     * Sets the total molar flow rate, and recalculates the corresponding individual molar flow rates.
     * @param totalMolarFlow The incoming total molar flow rate.
     */
    public void setTotalMolarFlow(double totalMolarFlow) {
        this.totalMolarFlow = totalMolarFlow;

        //Calc new molar flows
        for(int i=0; i<this.species.length; i++)
            this.molarFlows[i] = totalMolarFlow * this.moleFractions[i];
    }
    public void setTemperature(double temperature) {
        this.temperature=temperature;
    }
    public void setPressure(double pressure) {
        this.pressure = pressure;
    }

    /**
     * Overriden equals method.
     * Invokes the equals method from the species class to compare each variable within each species
     * returns false if any variable is different in value
     * @param stream the stream that is compared with for equals
     * @return true if equals, false if not
     */
    public boolean equals(Stream stream)
    {
        if                        ((this.species.length!=stream.species.length)                    ||
(this.molarFlows.length!=stream.molarFlows.length)                                                 ||
(this.moleFractions.length!=stream.moleFractions.length))
        {
            return false;
        }

        for (int i=0;i<this.species.length;i++)
        {
            boolean isTrue = (this.species[i].equals(stream.species[i]));
            if (isTrue==false)
                return false;
        }
        for (int i=0;i<this.molarFlows.length;i++)
        {
            if   ((this.molarFlows[i]   !=   stream.molarFlows[i])   ||   (this.moleFractions[i]   !=
stream.moleFractions[i]))
            {
                return false;
            }
        }
```

```
          return    ((this.temperature==stream.temperature)    &&    (this.pressure==stream.pressure)    &&
(this.totalMolarFlow==stream.totalMolarFlow));
     }


     /**
      * Overriden toString Method
      * For the species array, only the name is returned, using getName()
      * @return sends a string containing desired information
      * Puts molar flow and fraction of each species together
      */
     public String toString()
     {
          String str;
          str = "\nTotal Molar Flow= " + this.totalMolarFlow + "\nTemperature= " + this.temperature +
"\nPressure= " + this.pressure;
          for (int i=0;i<this.species.length;i++)
          {
               str = str + "\nSpecies " + (i+1) + "=" + this.species[i].getName();
               str = str + "\n\tMolar Flow=" + this.molarFlows[i];
               str = str + "\n\tMol Fraction=" + this.moleFractions[i];

          }


          return str;
     }


     public Stream clone()
     {
          return new Stream(this);
     }
} //End of class
```

11.1.15 VapourSpecies

```
import java.util.Arrays;
import customExceptions.*; //To get the innappropriate boundaries exception

/**
 * A class representing a species in the vapour phase.
 * This class also needs the liquid heat capacity, that is why liquid heat capacity is still in the species
class.
 * This class adds methods to calculate gas heat capacities assuming they behave as ideal gases, and
 */
public class VapourSpecies extends Species
{
//-------------------------------------------------CONSTRUCTORS-----------------------------------------
------------

     /**
      * The constructor. Calls the super constructor as there are no additional instance variables
      * @param physProperties an array of physical properties. Must be in the following order:
idNum,Name,Formula,numMolecules,Molar Mass,antoineA,antoineB,antoineC,Tmin,Tmax,Latent Heat(molar),Boiling
Temperature, liqheatC1, liqheaC2, liqheatC3, liqHeatC4, liqheatEquation, criticalTemp, criticalPress
      * */
     public VapourSpecies(String[] physProperties)
     {
          super(physProperties);
     }//End of constructor


     /**
      * The copy constructor.
      * There are no new instance variables here so it should be able to take a liquid species and make it
into a gas species.
      * @param toCopy The object to be copied.
      */
```

```
     public VapourSpecies(Species toCopy)
     {
          super(toCopy);
     }//End of copy constructor

     /**
      * Default constructor
      * */
     public VapourSpecies()
     {
          super();
     }


//-----------------------------------------------METHODS-----------------------------------------------
-------------


     /**
      * Calculates the gas enthalpy assuming ideal gas and using the integrated equation from Van Ness
      * @param tempStart the starting temperature in K
      * @param tempEnd the final temperature in K
      * @return The ideal gas heat capacity in J/mol*K
      */
     private double integrateGasHeatCapacity(double tempStart, double tempEnd)
     {

               double gasEnthalpy = (super.getGasHeatCapConstants()[0] * (tempEnd - tempStart)
                                        +  super.getGasHeatCapConstants()[1]  *  (Math.pow(tempEnd,  2)  -
Math.pow(tempStart, 2)) / 2
                                        +  super.getGasHeatCapConstants()[2]  *  (Math.pow(tempEnd,  3)  -
Math.pow(tempStart, 3)) / 3
                                        - super.getGasHeatCapConstants()[3] * (1/tempEnd - 1/tempStart))
* super.R;

               return gasEnthalpy;
     }


     /**
      * Calculates the difference in enthalpy of a gas when temperature changes from the reference temp to
the temp of interest.
      * Since the latent heat is only known at one temperature, we have to exploit the path-independant
properties of enthalpy.
      *
      * Calculates the enthalpy change to go from ref temp to boiling point as a liquid, and then from
boiling point to actual temp as a gas.
      *
      * Ignores pressure effects.
      * gaseous species are ideal gasses.
      * @param temp the temperature in K
      * @return the enthalpy in J/mol
      */
     public double calcEnthalpy(double temp)
     {

          double heatCapacity = 0.0;
          double enthalpy = 0.0;
          boolean isZero = true;

          //Check if all heat capacity constants are 0
          for(int i=0; isZero && i<super.getGasHeatCapConstants().length; i++)
               isZero = (super.getGasHeatCapConstants()[i] == 0.0);

          if(isZero) { //approximate heat capacity with R
               System.out.println("The gas heat capacity array is filled with zeros... \nApproximating the
gas heat capacity"
                                       + " with the ideal gas constant");
               if(super.getNumMolecules() == 1)
```

```
                    heatCapacity = 3/2 * R;
               else if(super.getNumMolecules() == 2)
                    heatCapacity = 5/2 * R;
               else if(super.getNumMolecules() > 2)
                    heatCapacity = 3 * R;
               else {
                    System.out.println("The number of molecules is not a positive integer. Try re-entering
the species");
                    System.exit(0);
               }

               //Calculate enthalpy with assumed heat cap
               if(super.getIsCondensable() == false)
                    enthalpy = heatCapacity * (temp - super.refTemp);
               else
                    enthalpy  =  super.integrateLiqHeatCapacity(super.refTemp,  super.getNormBoilTemp())  +
super.getMolarLatHeat() + heatCapacity *(temp - super.getNormBoilTemp());
          }

          //if heat cap constants are known
          else if(super.getIsCondensable() == false)
               enthalpy = this.integrateGasHeatCapacity(super.refTemp, temp);
          else
               enthalpy   =   super.integrateLiqHeatCapacity(super.refTemp,   super.getNormBoilTemp())   +
super.getMolarLatHeat() + this.integrateGasHeatCapacity(super.getNormBoilTemp(), temp);

          return enthalpy; //J/mol
     }


//-----------------------------------------------HOUSEKEEPING METHODS---------------------------------


     /**
      * @return a new VapourSpecies object
      */
     public VapourSpecies clone()
     {
          return new VapourSpecies(this);
     }
}
```

11.1.16 VLEModel

```
import customExceptions.NotFlashable;
/**
 * Model for calculating the K values.
 * Child classes: IdealModel, PRModel, and WilsonSRKModel.
 */
public abstract class VLEModel
{
    public abstract double[] calculateKi(FlashTank flashTank) throws NotFlashable;
}
```

11.1.17 WilsonSRKModel

```
/**
 * Calculates the K values for each species based on critical pressure and critical temperature
 */
public class WilsonSRKModel extends VLEModel
{
  /**
 * The K values here usually serve as initial guesses for the other models.
 * @param flashTank The flash tank being solved.
 * @return The initial K values of the species.
 */
  public double[] calculateKi(FlashTank flashTank) {
    int n = flashTank.getFeedStream().getSpecies().length;
    double[] initialKi = new double[n];
```

```
    for (int i=0;i<n;i++) {
      double critP = flashTank.getFeedStream().getSpecies()[i].getCriticalPressure();
      double P = flashTank.getPressure();
      double w = flashTank.getFeedStream().getSpecies()[i].getAccentricFactor();
      double critT = flashTank.getFeedStream().getSpecies()[i].getCriticalTemperature();
      double T = flashTank.getFlashTemp();

      initialKi[i] = critP/P * Math.exp(5.37 * (1+w) * (1-critT/T));
    }

     for (int i=0; i<flashTank.getFeedStream().getSpecies().length; i++) {
               if(!flashTank.getFeedStream().getSpecies()[i].getIsCondensable()) //T>Tc
                    initialKi[i] = 1e30; //Just a really high number
          }

    return initialKi;
  }
}
```

11.2 NUMERICAL METHODS CODING

11.2.1 CubicEquationSolver

```
package numericMethods;
import customExceptions.NotFlashable;

/**
   * Solves a cubic equation of form:
   * x^3 + a*x^2 + b*x + c = 0
   * Used for the PR EOS to find compressibility factor Z
   * */
public class CubicEquationSolver
{


  /**
   * Solves a cubic equation of form:
   * x^3 + a*x^2 + b*x + c = 0
   * @return the minimum root and the maximum root (of 3 roots).
   * */
  public double[] solveCubicEquation(double[] constants) throws NotFlashable
  {
    if (constants.length != 3) {
      System.out.println("Wrong number of constants.");
      System.exit(0);
    }
    double[] roots;
    double a = constants[0];
    double b = constants[1];
    double c = constants[2];


    double Q = (Math.pow(a,2)-3*b)/9;
    double R = (2*Math.pow(a,3)-9*a*b+27*c)/54;

    double M = Math.pow(R, 2) - Math.pow(Q, 3);

    if(M<0) {
      roots = new double[2];
      double theta = Math.acos(R/Math.sqrt(Math.pow(Q, 3)));

      double x1 = -(2*Math.sqrt(Q)*Math.cos(theta/3))-a/3;
      double x2 = -(2*Math.sqrt(Q)*Math.cos((theta+2*Math.PI)/3))-a/3;
      double x3 = -(2*Math.sqrt(Q)*Math.cos((theta-2*Math.PI)/3))-a/3;
      double minX = Math.min(x1,x2); //Only the smallest and largest roots are needed to find Z.
      roots[0] = Math.min(minX,x3);
      double maxX = Math.max(x1,x2);
      roots[1] = Math.max(maxX,x3);

      return roots;

    }
    else
    {
         throw new NotFlashable("System not solved...\nOnly one root found for the cubic EoS.");
    }
  }
}
```

11.2.2 HasRoot

```
//this is used for bisection and Ridder's method
package numericMethods;
import customExceptions.BadFunction;
```

```
/**
 * Interface for equations that can be solved.
 * Note: got rid of the y-target. It will just be a parameter of Bisection method in RootFinder
 */
public interface HasRoot
{
     public double findYGivenX(double x) throws BadFunction;

}//end of interface
```

11.2.3 Main

```
//this is used for testing numerical methods
package numericMethods;

public class main
{
  public static void main(String[] args)
  {
    double y=0;

    test sender=new test();

    double root=sender.calcRoot(y);

    System.out.println("root=" +root);
  }//end of main method
}//end of main class
```

11.2.4 NonlinearEquation

```
//this is used for Newton’s method
package numericMethods;

public interface NonlinearEquation
{
  public double returnValue(double x);
  public double returnDerivative(double x);
}
```

11.2.5 RootFinder

```
package numericMethods;
import customExceptions.NoRootFound;
import customExceptions.BadFunction;

/**
 * Finds the solution to any object implementing HasRoot using: Bisection or Ridder's method
 * */
public class RootFinder
{
     /**
      * Solves the roots of findYGivenX that sets Y to Ytarget.
      */
     public static double bisectionRoot(double xmin , double xmax, double yTarget, double tol, long maxIt,
HasRoot e) throws BadFunction
     {
          double a, b, m, y_m, y_a,y_b,xDiff;

          a=Math.min(xmin,xmax);
          b=Math.max(xmin,xmax);
          m=0.0;//just to initialize

          xDiff=b-a;
          long iter=0;
          while ( xDiff > tol&& iter<maxIt)
          {
```

```
            m = (a+b)/2;           // Mid point

            y_a = e.findYGivenX(a)-yTarget;       // y_a = f(a)
            y_m = e.findYGivenX(m)-yTarget;       // y_m = f(m)
            y_b = e.findYGivenX(b)-yTarget;       // y_b = f(b)

            if ( (y_m > 0 && y_a < 0) || (y_m < 0 && y_a > 0) )
            {
                 b = m;
                 // f(a) and f(m) have different signs: move b
            }

            else
            {
                 if((y_m > 0 && y_b > 0) || (y_m < 0 && y_b < 0))
                 {
                      System.out.println("ya, ym, yb all have the same sign. Returning 0");
                      return 0;
                 }

                 // f(a) and f(m) have same signs: move a

                 a = m;

            }
            xDiff=Math.abs(b-a);
            iter++;
        }
    return m;
    }//End of bisection method

    public static double ridderRoot(double xMin , double xMax, double yTarget, double tolerance, int maxIt,
HasRoot e) throws BadFunction {
          double xL=Math.min(xMin,xMax);
          double xU=Math.max(xMin,xMax);
          double xM=(xL+xU)/2;
          int sgn=0;
          double xR=0;

          double f_xL;
          double f_xU;
          double f_xM;
          double f_xR;

          double epsilon_a=xU-xL;
          int iter=0;

          double[] solution=new double[maxIt];

          while(epsilon_a>tolerance && iter<maxIt)
          {

               f_xL=e.findYGivenX(xL)-yTarget;
               f_xU=e.findYGivenX(xU)-yTarget;
               f_xM=e.findYGivenX(xM)-yTarget;

               //find xR
               if((Math.pow(f_xM, 2)-f_xL*f_xU<0))
                    throw new BadFunction("There are no real roots");

               xR = xM + (xM-xL) * (Math.signum(f_xL-f_xU)*f_xM / Math.sqrt(Math.pow(f_xM, 2)-f_xL*f_xU));
               f_xR=e.findYGivenX(xR)-yTarget;
               solution[iter]=xR;

               if(xR<xM) {
                    if((f_xL * f_xR)<0) {
                         xU = xR;
                    }
                    else if((f_xR * f_xM)<0) {
```

```
                    xL = xR;
                    xU = xM;
               }
               else if((f_xM * f_xU)<0) {
                    xL = xM;
               }
               else {

               }
          }

          if(xR>xM) {
               if((f_xL * f_xM)<0) {
                    xU = xM;
               }
               else if((f_xM * f_xR)<0) {
                    xL = xM;
                    xU = xR;
               }
               else if((f_xR * f_xU)<0) {
                    xL = xR;
               }
               else {
                    System.out.println("Wow ridders nailed the solution");
                    return xR;
               }
          }

          xM=(xL+xU)/2;

          if(iter>0) {//calculate the error
               epsilon_a=Math.abs((solution[iter]-solution[iter-1])/solution[iter]);
          }

          iter ++;
        }

        System.out.println("... solved after "+iter+" iterations...");

        if(epsilon_a>tolerance)
             System.out.println("Maximum number of iterations reached before convergence");

        return xR;
    }//end of Ridder's method



    public static double Newton(double x_0, double tol, long maxIt, NonlinearEquation e) throws BadFunction
    {
      double x_old=x_0;
      double x_new;
      double x_diff;
      long iter=0;

      do
      {
        x_new=x_old-e.returnValue(x_old)/e.returnDerivative(x_old);
        x_diff=Math.abs(x_old-x_new);
        x_old=x_new;

        iter++;
      } while(x_diff>tol && iter<maxIt);

      if(x_diff>tol) System.out.println("Maximum number of iterations reached before convergence");

      return x_new;
    }
}//end of bisection class
```

11.2.6 Test

```
//this class has the actual equation used for testing numerical methods
package numericMethods;

import java.lang.Math;

public class test implements HasRoot
{
 public double function(double x)
  {
    double y=5*(0.07-Math.exp(-0.04*x))+2*Math.exp(-0.04*x);

    return y;
  }//end of efficiency calculation

  public double calcRoot(double y)
  {
    double xMin=53;
    double xMax=54;
    double tolerance=0.0001;
    int maxIt=1000;

    //already works
    //return RootFinder.bisectionRoot(this, y, xMin, xMax);

    //testing with Ridder's method
    return RootFinder.ridderRoot(xMin, xMax, y, tolerance, maxIt, this);

    //has to match the bisection method in RootFinder
  }//end of root finder method

  public double findYGivenX(double x)
  {
    double root=this.function(x);

    return root;
  }//end of root finder method
}//end of class
```

11.3 CUSTOMEXCEPTIONS

11.3.1 BadFunction

```
package customExceptions;

public class BadFunction extends Exception
{

     private String msg;

     public BadFunction (String msg)
     {
          this.msg = msg;
     }

     public BadFunction()
     {
          this.msg = "The function you are using for rootfinder isn't going to work!";
     }

     public String getMsg()
     {
          return this.msg;
     }

     public void setMsg(String msg)
     {
          this.msg = msg;
     }
}
InappropriateBoundaries
package customExceptions;

/**
 * Used when the boundaries of an interval end on  values that they shouldn't
 */
public class InappropriateBoundaries extends Exception
{


     private String message;


//-----------------------------------------------CONSTRUCTORS-----------------------------------------
-----------


     /**
      * Creates an InappropriateBoundaries object with the default error message.
      * @param boundaryOne One of the illegal boundaries
      * @param boundaryTwo The other boundary
      */
     public InappropriateBoundaries(double boundaryOne, double boundaryTwo)
     {
          this.message = "The boundaries " + boundaryOne + " and " + boundaryTwo + " cannot be used for
these calculations./n"
               + "Please input new boundaries.";
     }


     /**
      * Creates an InappropriateBoundaries object with a custom error message.
      * @param message The custom error message.
      */
     public InappropriateBoundaries(String message)
     {
          super(message);
```

```
     }//End of custom message constructor
}//End of InappropriateBoundaries class
```

11.3.2 IncorrectArraySize

```
package customExceptions;

/**
 * This is an exception that will be thrown when an array is the wrong size
 */
public class IncorrectArraySize extends Exception
{


     private String message;


//-------------------------------------------CONSTRUCTORS-----------------------------------------------
---


     /**
      * Creates a custom error message.
      * @param errorMessage A string containing a custom error message
      */
     public IncorrectArraySize(String message)
     {
          super(message);
     }//End of Constructor


     /**
      * Creates an IncorrectArraySize object with the following error message
      *
      * "The array you have entered is not the proper size for this method."
      * "Double check the number of elements you are supposed to have."
      */
     public IncorrectArraySize()
     {
          this.message = "The array you have entered is not the proper size for this method." +
               "\nDouble check the number of elements you are supposed to have.";
     }
}
```

11.3.3 NoRootFound

```
package customExceptions;

public class NoRootFound extends Exception{

     private String msg;

     public NoRootFound(String msg) {
          this.msg = msg;
     }

     public NoRootFound() {
          this.msg = "No Root Found";
     }

     public String getMsg() {
          return this.msg;
     }

     public void setMsg(String msg) {
          this.msg = msg;
     }
}
```

11.3.4 NotFlashable

```
package customExceptions;

public class NotFlashable extends Exception
{

     public NotFlashable(String message)
     {
          super(message);
     }

     public NotFlashable()
     {
          super("The current mixture is not flashable under these operating conditions.");
     }
}
```