IBIS Simulation

DEVELOPER GUIDE

Developed by:

AHMAD HATZIQ B MOHAMAD

CHONG YONG XIANG

LING ZHI WEI

NG JING HUI DARRELL

Table of Contents

This technical document is intended for developers who wish to extend on the IBIS Simulation program application. This is written for **v1.0.0** of the application.

Supported OS: Windows

Programming Language: Java 11

User Interface (UI) Framework: JavaFX

Build Automation Tool: Gradle

IDE (v1.0.0): IntelliJ IDEA Community Edition

Java Development Kit (JDK) is **not** required to be installed to run the application.

Setting Up & Launching

Set up the source code project in a Gradle-supported IDE, preferably IntelliJ.

Build the project using the **`build.gradle`** file.

Launch the application through the IDE, by running **`Launcher.java`**.

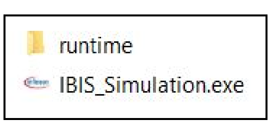
Packaging / Exporting as Executable File

Ensure that you have the **runtime** folder, which includes all the JDK dependencies!

Run the Gradle task: `launch4j` > **`createExe`**

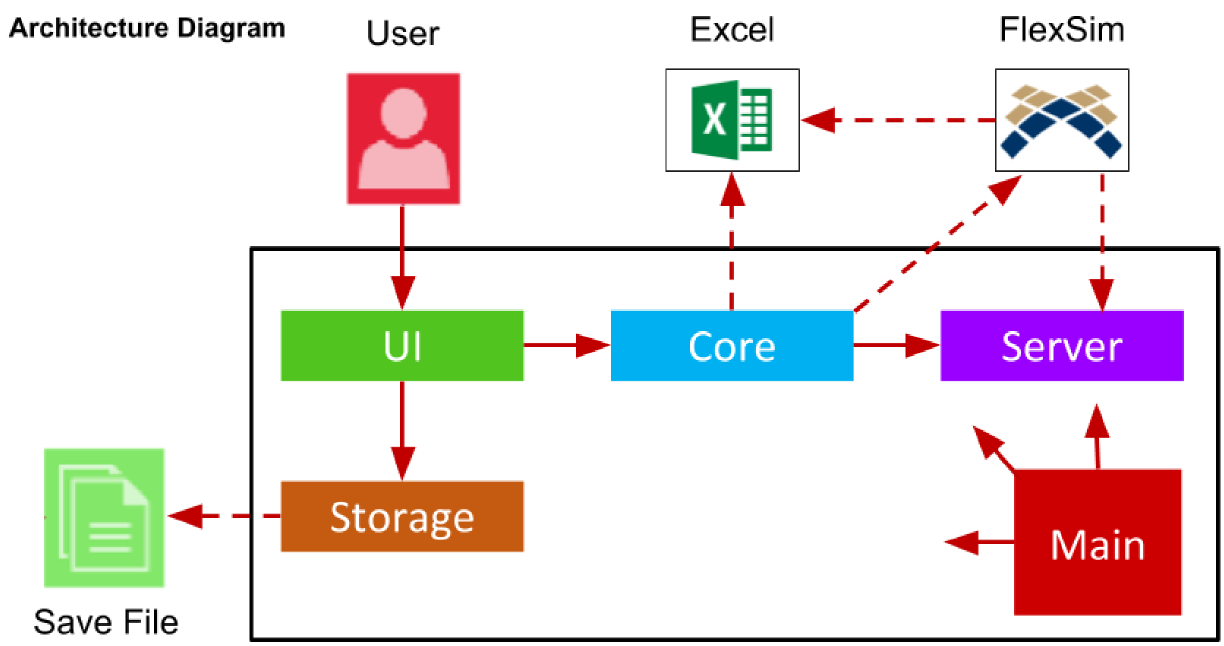
This will generate an executable file, **`IBIS_Simulation.exe`**, which can be found in the `\build\launch4j` folder:

Copy this to the same directory as the runtime folder:



The program can now be launched by double-clicking `IBIS_Simulation.exe`.

Design Architecture



The Architecture Diagram given above explains the high-level design of the program application. The program adopts a multi-layered architecture that provides a user interface (UI) around the **Core** component that processes Excel files and interacts with FlexSim software to facilitate the simulation runs. A simple **Storage** component has been implemented to save and load the input fields provided by the user.

Given below is a quick overview of each component.

Main: Initialises the Core and UI components, in the correct sequence, while preloading the Storage data, and connects them up with each other.

UI: Displays the Graphical User Interface (GUI) for the user input.

Core: Executes the entire simulation through its sub-components.

Server: Interacts with the FlexSim simulation software for running of simulation runs.

Storage: Reads data from, and writes data to, a text (.txt) save file.

Folder Structure

```
src → main:
      ↳ java
        ↳ com
          ↳ nusinfineon
            ↳ Launcher.java
            ↳ Main.java
            ↳ core
              ↳ Core.java
              ↳ InputCore.java
              ↳ OutputCore.java
              ↳ RunCore.java
              ↳ input
                ↳ LotEntry
                  ↳ GenericLotEntry.java
                  ↳ LotEntry.java
                  ↳ MJLotEntry.java
                  ↳ SPTLotEntry.java
              ↳ output
                ↳ OutputAnalysisCalculation.java
                ↳ OutputAnalysisDriver.java
                ↳ OutputAnalysisUtil.java
              ↳ util
                ↳ FlexScriptDefaultCodes.java
                ↳ ScriptGenerator.java
                ↳ Server.java
            ↳ exceptions
              ↳ CustomException.java
            ↳ storage
              ↳ JsonParser.java
            ↳ ui
              ↳ MainGui.java
              ↳ Ui.java
              ↳ UiManager.java
              ↳ UiPart.java
            ↳ util
              ↳ Directories.java
              ↳ LotSequencingRule.java
              ↳ Messages.java
      ↳ resources
        ↳ images
          ↳ icon.ico
          ↳ icon_large.png
        ↳ output
          ↳ IBIS_Simulation_Output_Visualisation.twb
          ↳ product_key_cost.xlsx
        ↳ view
          ↳ MainGui.fxml
```

Important Files

Class / File	Description
Main	
`Launcher.java`	Application launcher. **Application should be launched by running this class.**
`Main.java`	Main component that initialises the application.
`Directories.java`	Defines URLs, directories, folder/file names etc. to be used globally. **Add / edit paths here.**
UI	
`UiManager.java`	Manager that operates the whole UI.
`MainGui.java`	Front-end model and logic of the whole user interface window. **Edit UI model and logic here.**
`MainGui.fxml`	Defines the layout design of the GUI window. (Built on SceneBuilder) **Edit layout here.**
`Messages.java`	Defines messages to be displayed. **Add / edit messages here.**
Core	
`Core.java`	Back-end model of the user-defined input. Main logic of executing the entire simulation. **Edit main execution flow here.**
InputCore	
`InputCore.java`	Enumerates all combinations of lot sequencing rule(s), minimum batch size(s) and additional settings from user-defined input, and creates a new Excel file for each scenario. **Refer to the InputCore section for more details.**
`LotEntry.java`	Abstract class for a lot entry in Actual Lot Info sheet for sorting according to different lot sequencing rules. To be inherited for each rule, as each rule has a different comparable for sorting by Java Collections. **Create new types of lot entry by extending this abstract class.**
`GenericLotEntry.java`	Generic lot entry. Used for Randomise rule.
`MJLotEntry.java`	Lot entry for sorting according to Most Jobs rule. Takes `lotSize` as comparable.
`SPTLotEntry.java`	Lot entry for sorting according to Shortest Processing Time rule. Appends `processTime` which is taken as comparable.

`LotSequencingRule.java`	Enum of lot sequencing rules for use as keys in hashmaps. **Add new rules as keys here.**
RunCore	
`RunCore.java`	Interfaces with FlexSim to run all the simulation runs with the use of FlexScript and a server. **Refer to the RunCore section for more details.**
`Server.java`	Listens to FlexSim to start off the next simulation run (if any).
`ScriptGenerator.java`	Creates FlexScript for each simulation run.
`FlexScriptDefaultCodes.java`	Default FlexScript codes.
OutputCore	
`OutputCore.java`	Processes all output Excel files and generates an Excel file for Tableau data visualisation. **Refer to the OutputCore section for more details.**
`OutputAnalysisCalculation.java`	Processes a single output Excel file and generates relevant summary statistics. **Edit calculations to output Excel file (from FlexSim output) here.**
`OutputAnalysisUtil.java`	Provides utility functions for OutputCore and OutputAnalysisCalculation.
`OutputAnalysisDriver.java`	**Run this class to generate `tableau-excel-file.xlsx` from a folder of output Excel files.**
`product_key_cost.xlsx`	Lookup table for per unit cost of each product key. **Update this Excel workbook for new products and updated costs.**
`IBIS_Simulation_Output_Visualisation.twb`	Master copy of Tableau workbook for output data visualisation. Workbook generated in the output folder takes a copy of this workbook. **Update this Tableau workbook to edit data visualisation.**
Storage	
`JsonParser.java`	Reads and writes the user input data on a text (.txt) file in JSON format.
Images	
`icon.ico`	`IBIS_Simulation.exe` icon (Only required in `build.gradle`).
`icon_large.png`	Icon for UI window.

Implementation

This is a brief overview of the entire program flow:

1. After the user launches the application, the Main component initialises the application and displays the GUI.

2. The user fills in the required input and clicks “Run Simulation” when ready.

3. If all the inputs are valid, a confirmation message will be displayed.
Else, an error message will be displayed.

4. Once the user confirms the execution, a wait message will be displayed.

The following steps 5 to 9 are illustrated in the sequence diagram below.

5. The MainGui component will pass all the input fields to the Core component and calls the `execute()` function to kick off the operations.

6. All the core subcomponents, InputCore, RunCore and OutputCore, are initialised with the user-defined input parameters and the input Excel file extracted from the database.

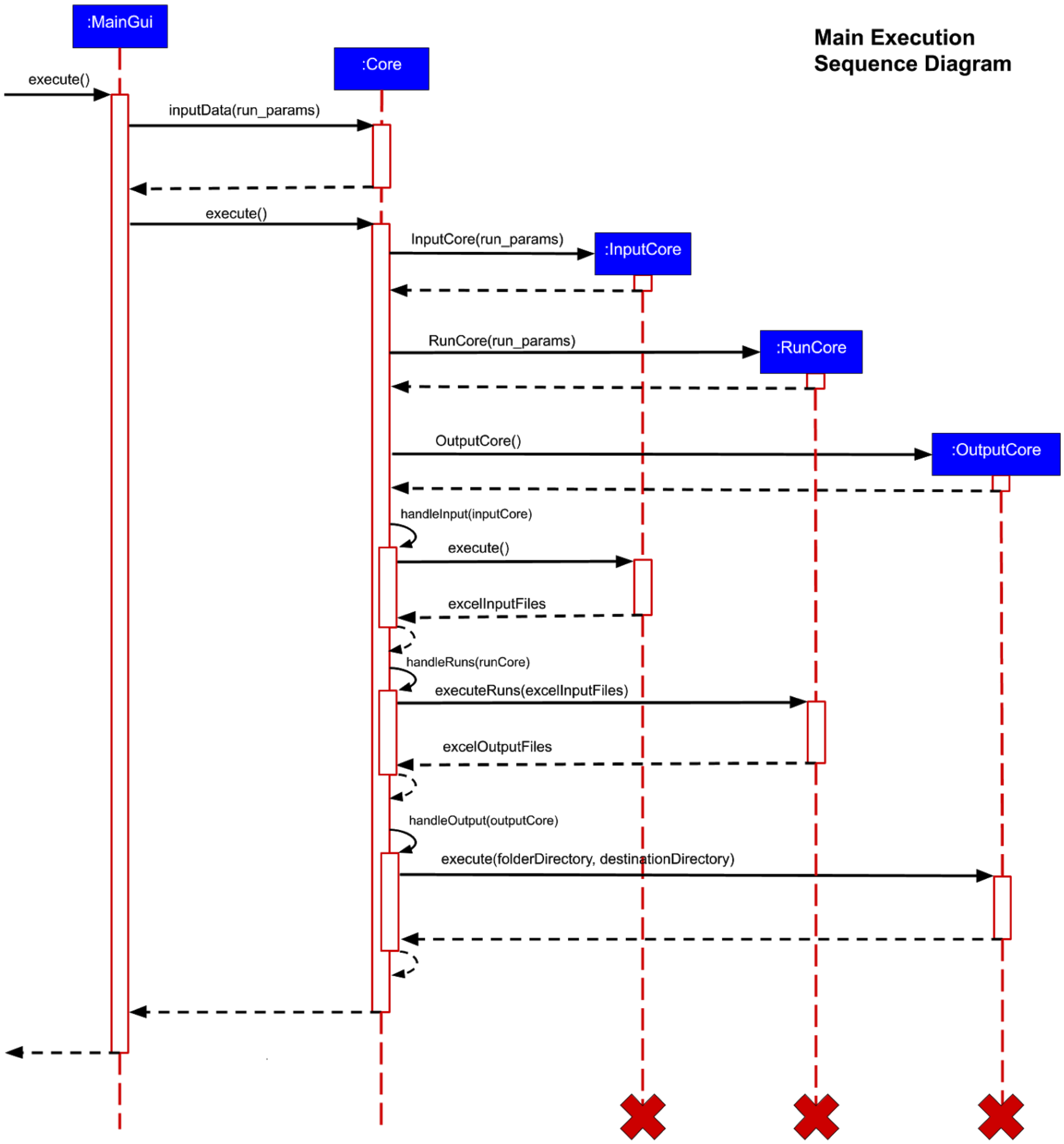
7. InputCore will generate the input Excel file required for each run and pass all of them to RunCore.

8. RunCore will execute the FlexSim simulation for each run, which builds the model based on the given input Excel file, then runs the model and generates the output Excel report.

9. OutputCore will process all the output files to generate an Excel file required for data visualisation on the designed Tableau dashboard.

10. Once completed, a completed message will be displayed.

Main Execution Sequence Diagram



MainGui

The MainGui component configures the interactions with the user through the UI and with the Core component when starting the simulation runs.

Important Methods:

`configureUi()`:

- Configures the UI from the back-end / saved input field data

`handleModelExecution()`:

- Validates input
- Confirms run with user
- Shows wait alert box
- Calls `execute()`
- Closes wait alert box when `execute()` ends

`execute()`:

- Passes input field data to Core
- Saves input field data into save file
- Calls `execute()` in Core
- Shows completion box when `execute()` ends

Core

The Core component provides a back-end model and manages the main overall flow of execution.

Important Methods:

`inputData(**input_params):`

- Assigns input field values from MainGui to Core local attributes

`execute():`

- Initialises core subcomponents with required input field data (if any)
- Handles input
- Handles runs
- Handles output
- Cleans up

`handleInput(InputCore):`

- Executes processing and generation of input Excel files
- Creates new Input folder (Empties it if exists) and moves generated input Excel files in

`handleRuns(RunCore):`

- Executes all runs

`handleOutput(OutputCore):`

- Creates new Output and Raw Output folder (Empties it if exists) and moves generated output Excel files into the latter
- Executes processing of output summaries on output Excel files in Raw Output folder
- Generates `tableau-excel-file.xlsx` in Output folder
- Copies master Tableau workbook from resources to Output folder
- Opens Tableau Server / Tableau workbook (if Tableau Desktop installed)

InputCore

The InputCore component creates copies of the original input Excel file, one copy for each run, and modifies the required columns of the parameters that the user varies in the GUI.

In the **`execute()`** method of InputCore, all the lot sequencing rules and minimum batch sizes defined by the user are enumerated. A single random seed is generated and used throughout each execution (for the randomise lot sequencing rule)!

In each iteration, the Input Excel file is duplicated and modified based on the lot sequencing rule and minimum batch size for that particular run.

Important Methods:

`editMinBatchSize(...)`:

- Takes the copy of the Excel workbook
- Gets the **`Product Info and Eqpt Matrix`** sheet
- Gets the **`BIB Slot Utilization Min`** column
- Sets all rows to that particular run's minimum batch size

`processLotSequencing(...)`:

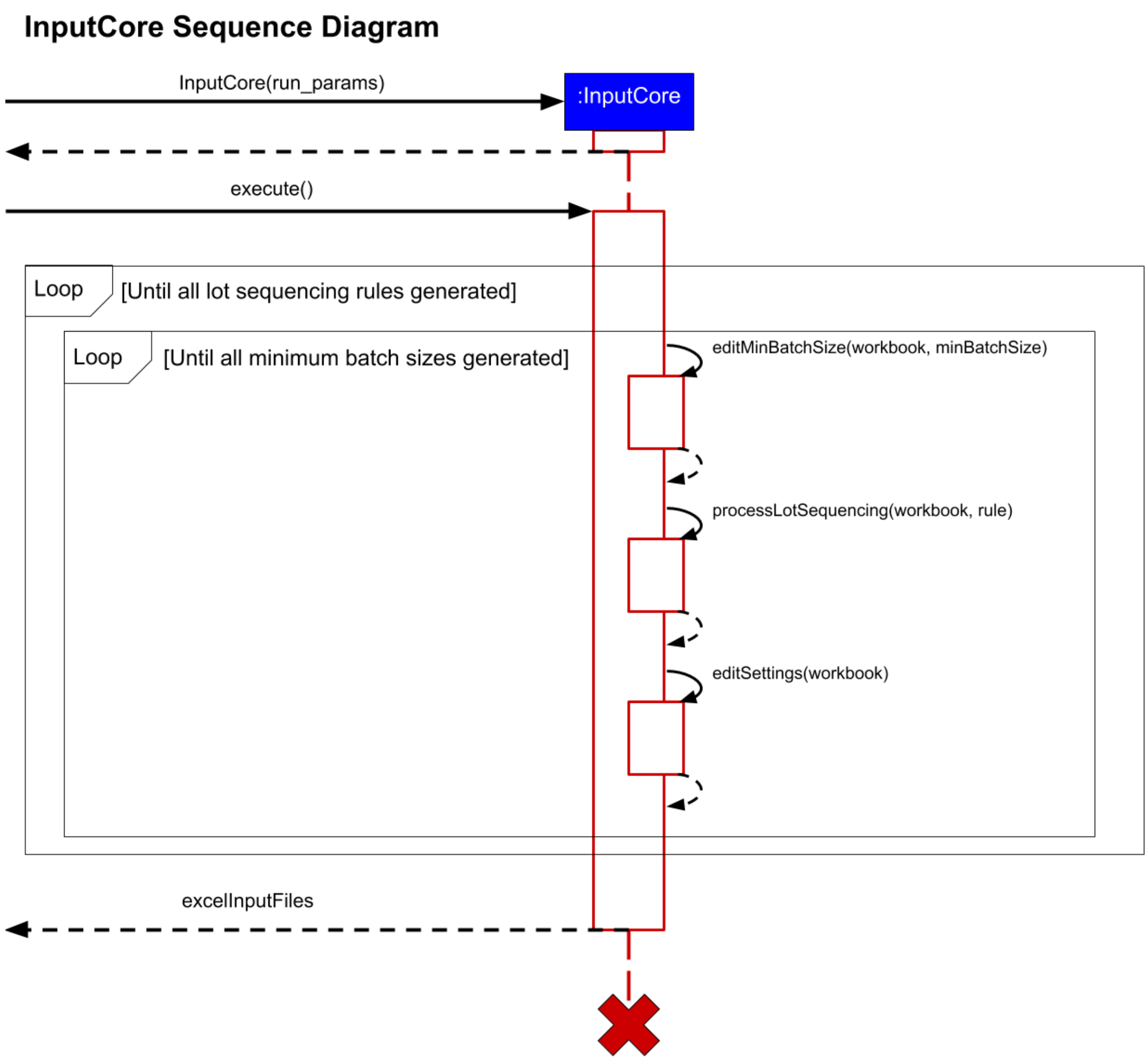
- Takes the copy of the Excel workbook
- Gets the **`Actual Lot Info`** sheet
- Reorders the row (lot) entries based on that particular run's lot sequencing rule:
    - **Shortest Processing Time:**
        - Gets the **`Process Time`** sheet
        - Copies corresponding process time from `Process Time (Hr)` column in **`Process Time`** sheet to a new column in **`Actual Lot Info`**
        - For each period, sort in ascending order of new `Process Time` column and add to temporary lot list
    - **Most Jobs:**
        - For each period, sort in descending order of **`Lotsize`** column and add to temporary lot list
    - **Random Sequence:**
        - For each period, randomise lot sequence using random seed and add to temporary lot list
    - **First-Come-First-Served:**
        - Does nothing
- Clears **`Actual Lot Info`** sheet
- Populates **`Actual Lot Info`** sheet with temporary lot list

`editSettings(...)`:

- Takes the copy of the Excel workbook
- Gets the **`Settings`** sheet
- For each required setting parameter:
    - Gets the setting under **`Parameter`** column
    - Modifies the **`Value`** column to the user-defined value

Once all the required input Excel files are generated, they are passed back to Core.

The following sequence diagram illustrates the execution flow of InputCore.



RunCore

Executing a single run of the simulation in the RunCore component can be broken down into three parts; model startup, setting model parameters and communication between FlexSim and our program.

Model Startup

For model startup, a customised command line string was executed through the Java Runtime class, which in turn, interfaces with the environment it is running on. Although there are other options for starting a program in Java, such as through the Desktop class which opens programs by their default applications, this method provides the most flexibility. Through the command line, a program can be executed along with special flags that start the program in certain modes or with specific instructions. In our program, we utilised this flexibility to allow users to run the simulation in the background, and allow the modification of model parameters externally through FlexScript. The latter point is key for the next part as the express license cannot access a majority of the model settings and code.

Setting Model Parameters

In order to change model parameters externally, the FlexSim model must be run with a special script file. This script contains code that allows developers to modify many of the model parameters before running it. A few examples of the possible modifications include runtime, run speed, altering model objects and controlling data input and output. By generating a FlexScript based on the user-defined options, we are able to control a majority of the FlexSim functions without the user ever needing to open the application. Additionally, since FlexScript was made for developers, it bypasses many of the license restrictions that a normal user faces if FlexSim was used directly. Hence, our program can provide all users with the same modeling flexibility.

Establishing Communication

After starting and modifying the model, the final step is to establish communication between FlexSim and our program. This will allow our program to monitor the simulation and detect when the current simulation is completed, before starting the next run. While there are multiple alternatives to carry this out, our program uses *Windows Sockets API (WSA)* to establish communication. Firstly, our program starts a socket server on the host computer and then listens on a user-defined port for a client to connect. In the FlexScript file described earlier, FlexSim is instructed to start a client socket and then forms a local connection using the host IP address and the desired port. The server picks up the communication request and accepts. This establishes a communication channel between FlexSim and our program. By combining this with model startup and FlexScript, our program can then interface with the FlexSim model and automate all the runs entirely without user input. This significantly reduces the learning curve and attention needed to run the simulations.

Important Methods:

`executeRuns(excelInputFiles)`:

- Iterates through each of the various combinations of parameters generated until all are completed:
    - Starts a new FlexSim process for the input Excel file
    - Waits for a connection from the server which indicates FlexSim is done
    - Appends the generated output Excel file to a list of output Excel files
- Returns the list of all the output Excel files

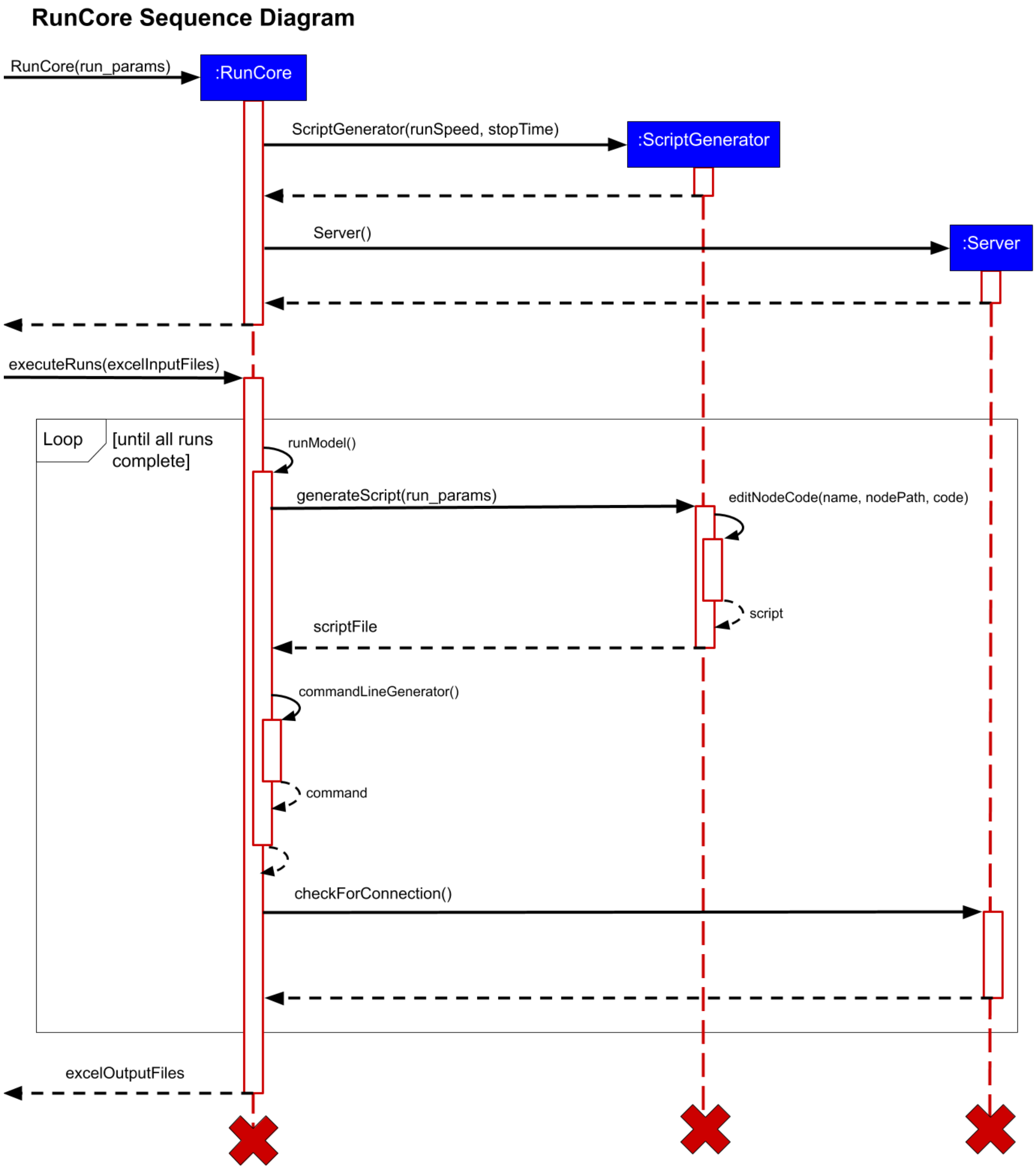
`runModel()`:

- Uses script generator to generate a FlexScript file to control the FlexSim model
- Uses command line generator to generate the appropriate command line string to be passed to Java runtime

`commandLineGenerator(isModelShown)`:

- Generates a command line with the appropriate execution flags

The following sequence diagram illustrates the execution flow of RunCore.



OutputCore

After all the simulation runs have finished executing, the individual simulation output files have to be pre-processed for analysis. The pre-processing occurs in two stages through the OutputCore component: pre-processing a single file and pre-processing all the files.

Pre-processing a Single File

For each output Excel file, the sheets are analysed to generate the KPIs for a single simulation run. These KPIs summarise the performance of each simulation run configuration. The KPIs are then appended to the single Excel output file, which helps in pre-processing in the next section.

Pre-processing All Files

After pre-processing, all the individual output files will have nicely formatted KPI sheets. This component will then compile all these KPIs into a single Excel file. Through this single Excel file, Tableau can then access the run data and generate the relevant visualisations, such as Daily Throughput Rate and Cycle Time.

Important Methods:

`execute(folderDirectory, destinationDirectory)`:

- Executes the main operations for OutputCore
- **`appendSummaryStatisticsOfFolderOfExcelFiles(folderDirectory)`:**
    - Processes all Excel output files in the specified `folderDirectory`
    - For each output Excel file:
        - **`appendSummaryStatisticsOfSingleOutputExcelFile(...)`**
- **`generateTableauExcelFile(...)`:**
    - Processes all Excel files into a single Excel file, used for Tableau visualisation

`appendSummaryStatisticsOfSingleOutputExcelFile(outputExcelFile)`:

- Generates 6 new sheets aggregating the KPIs of each output Excel file
    - `RUN_TYPE_AND_IBIS_UTILIZATION`
    - `PRODUCT_STAY_TIME`
    - `PRODUCT_THROUGHPUT`
    - `DAILY_OUTPUT`
    - `PRODUCT_TIME_IN_SYSTEM`
    - `PRODUCT_OUTPUT_WORTH`
- The method of aggregating each of the KPIs is as follows:

Sheet Name	Calculation Explanation
`RUN_TYPE_AND_IBIS_UTILIZATION`	Uses the `Util Res Rep` sheet. Calls `calculateAverageIbisOvenUtilRate(...) in OutputAnalysisCalculation.java`. Above method calculates the average of all IBIS rows i.e. rows with `IBIS` under the `Platform` column. Calls `fileStringToFileName(...)` in `OutputAnalysisUtil.java`. Above method extracts the filename of the Excel file and saves it as runtype metadata.
`PRODUCT_STAY_TIME`	Uses the `Throughput Product Rep` sheet. Calls `calculateProductCycleTimeFromThroughputProduct(...)` in `OutputAnalysisCalculation.java`. Above method gets the average product cycle time, derived from the `StayTime Average (hr)` column. The average is obtained for each unique product ID. Row entries with 0 period are ignored.
`PRODUCT_THROUGHPUT`	Uses the `Daily Throughput Product Rep` sheet. Calls `calculateProductThroughput(...)` in `OutputAnalysisCalculation.java`. Above method gets the average product throughput. Then, for each valid row entry, the throughput for each row is first calculated ie `QTY_OUT / TIME_IN_SYSTEM`. With these row throughputs, the average for each product ID is obtained.
`DAILY_OUTPUT`	Uses the `Daily Throughput Product Rep` sheet. Calls `calculateDailyThroughput(...)` in `OutputAnalysisCalculation.java`. Above method sums up all the `QTY_OUT` for each day.
`PRODUCT_TIME_IN_SYSTEM`	Uses the `Daily Throughput Product Rep` sheet. Calls `calculateProductCycleTimeFromDailyThroughput(...)` in `OutputAnalysisCalculation.java`. Above method gets the average `TIME_IN_SYSTEM` for each

	unique product ID.
PRODUCT_OUTPUT_WORTH	Uses the Daily Throughput Product Rep sheet and the Product Cost Excel file, obtained externally. The Product Cost Excel file is hard-coded into the application currently. It can be updated by replacing the file inside the resources folder. Calls calculateTotalProductWorth(...) in OutputAnalysisCalculation.java. Above method calculates the product worth for each product ID. Product worth is the total product output (QTY_OUT) multiplied with the associated cost within the Product Cost Excel file.

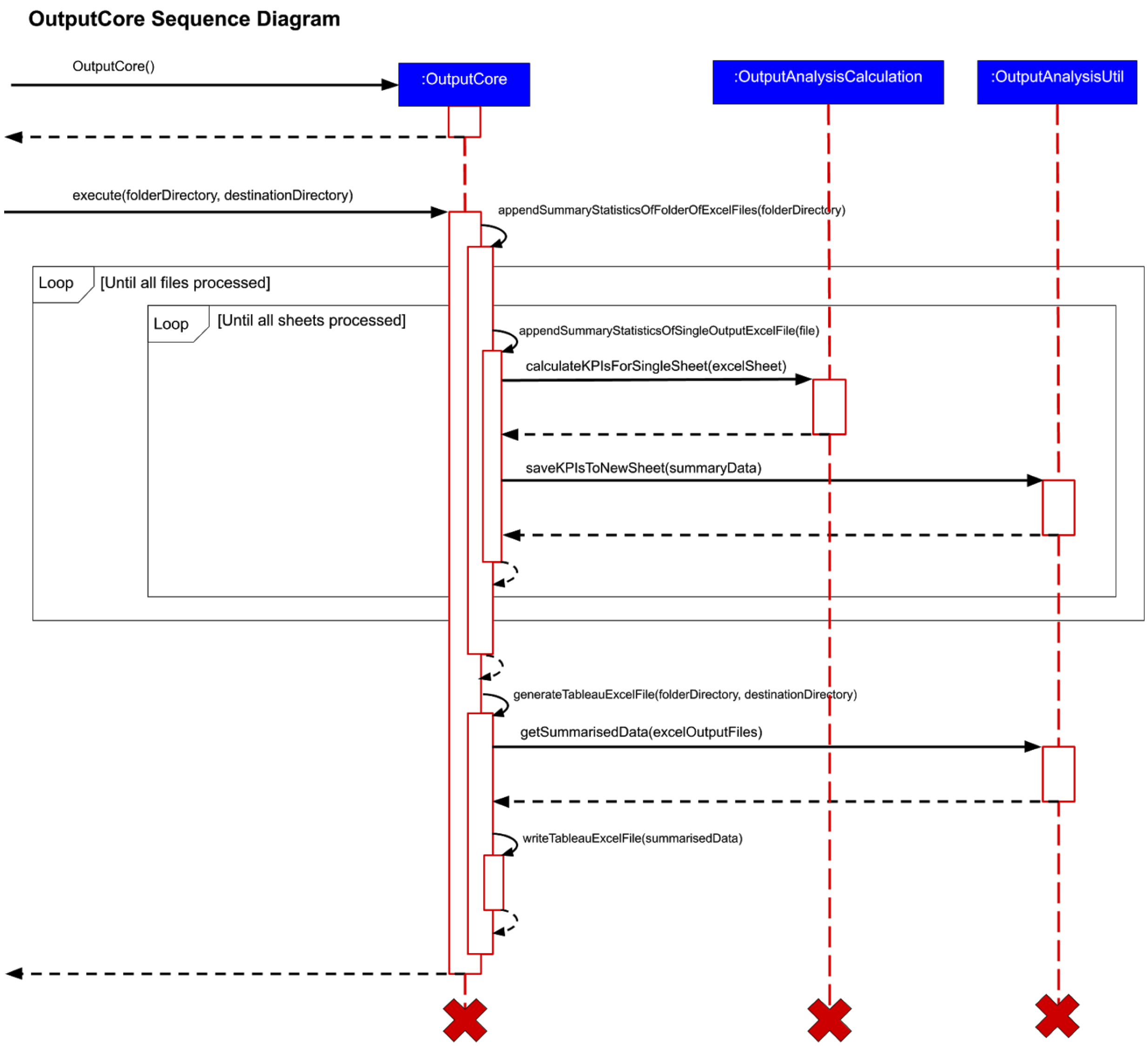
generateTableauExcelFile(folderOfExcelFiles, destinationDirectory):

- Generates a new Excel file, which contains 6 sheets, for Tableau data visualisation:
    - IBIS_UTILIZATION
    - STAY_TIME
    - TIME_IN_SYSTEM
    - THROUGHPUT
    - THROUGHPUT_DAILY
    - PRODUCT_OUTPUT_WORTH
- Each sheet summarises the results for all the simulation runs
- The method of aggregating each of the KPIs is as follows:

Sheet Name	Calculation Explanation
IBIS_UTILIZATION	Uses the RUN_TYPE_AND_IBIS_UTILIZATION sheet. Appends the entry of each simulation run as a new row.
STAY_TIME	Uses the PRODUCT_STAY_TIME sheet. Appends the stay time for each product ID from each simulation run.
TIME_IN_SYSTEM	Uses the PRODUCT_TIME_IN_SYSTEM sheet. Appends the time spent in the system for each product ID from each simulation run.
THROUGHPUT	Uses the PRODUCT_THROUGHPUT sheet. Appends the throughput for each product ID from each simulation run.

`THROUGHPUT_DAILY`	Uses the `DAILY_OUTPUT` sheet. Appends the daily output for each simulation run in a column wise manner.
`PRODUCT_OUTPUT_WORTH`	Uses the `PRODUCT_OUTPUT_WORTH` sheet. Appends the output worth for each product ID from each simulation run.

The following sequence diagram illustrates the execution flow of OutputCore.



Appendix A: Use Cases

Use Case: Run Simulation

Main Success Scenario:

1. User enters the input fields and requests to run the simulation.
2. Application requests for confirmation.
3. User confirms.
4. Application runs all the requested simulation runs and displays a “Simulation Completed” message after all runs are completed.

Use case ends.

Extension(s):

1a. Application detects an error in the entered input field.

1a1. Application requests for the correct input.

1a2. User enters new input and requests to run the simulation.

Steps 1a1 - 1a2 are repeated until the input entered are all correct.

Use case resumes from step 2.