EG Courses

Object Oriented Programming in LabVIEW

WORKBOOK

Revision 2.1

Object-Oriented
Programming
in LabVIEW

Table of Contents

General Information

This document provides step-by-step instructions for all Object Oriented Programming (OOP) course exercises and projects. OOP is part of a series of specialized training courses collectively known as EG Courses.

EG Courses are a great opportunity to fully learn the principles of LabVIEW programming. Classes, conducted by certified LabVIEW architects, provide extensive knowledge and develop practical skills in the field of real-time systems, automation test applications, measurement monitoring, broadly understood data acquisition and many other topics related to the construction and maintenance of control systems. The modular structure of the courses allows to adjust their content to the expectations of students.

The OOP course presents the principles and techniques of object-oriented programming in the LabVIEW environment.

All the materials needed to complete the exercises as well as the proposed solutions are available in the GitHub repository: www.github.com/EGCourses/OOP.

ABBREVIATIONS

BD	Block Diagram
CP	Connector Pane
FP	Front Panel
LMB	Left Mouse Button
RMB	Right Mouse Button

Words like FOR, WHILE, CASE etc. refer to the corresponding programming object (for example, CASE stands for *Case Structure*, WHILE stands for *While Loop*, and so on).

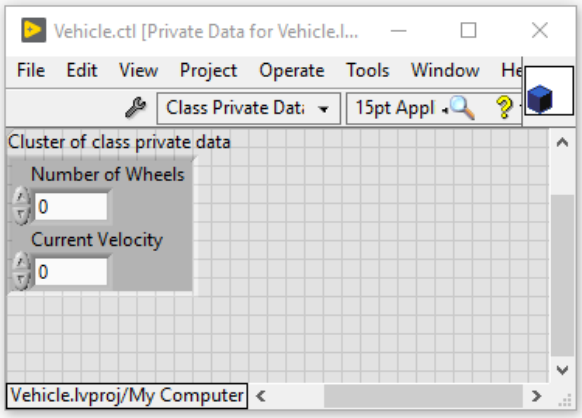
EXERCISE 1 Class Attributes

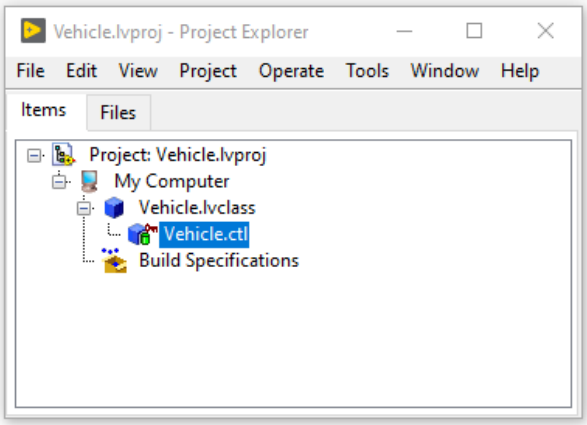
GOAL

Create new blank project named *Vehicle* and new *Vehicle* class to it. Add two class fields for *Number of Wheels* and *Current Velocity* values.

IMPLEMENTATION

1. Create class.
   - Create new project and save it on disk.
   - Under My Computer target, add new class *Vehicle*. Save it on disk under subdirectory *Vehicle*.
   - Open *Vehicle.ctl* file and add two numeric controls into *Cluster of class private data*: *Number of Wheels* and *Current Velocity*. Set appropriate representation (number of wheels value is always an integer).
   - Save the class and the project.





END OF EXERCISE

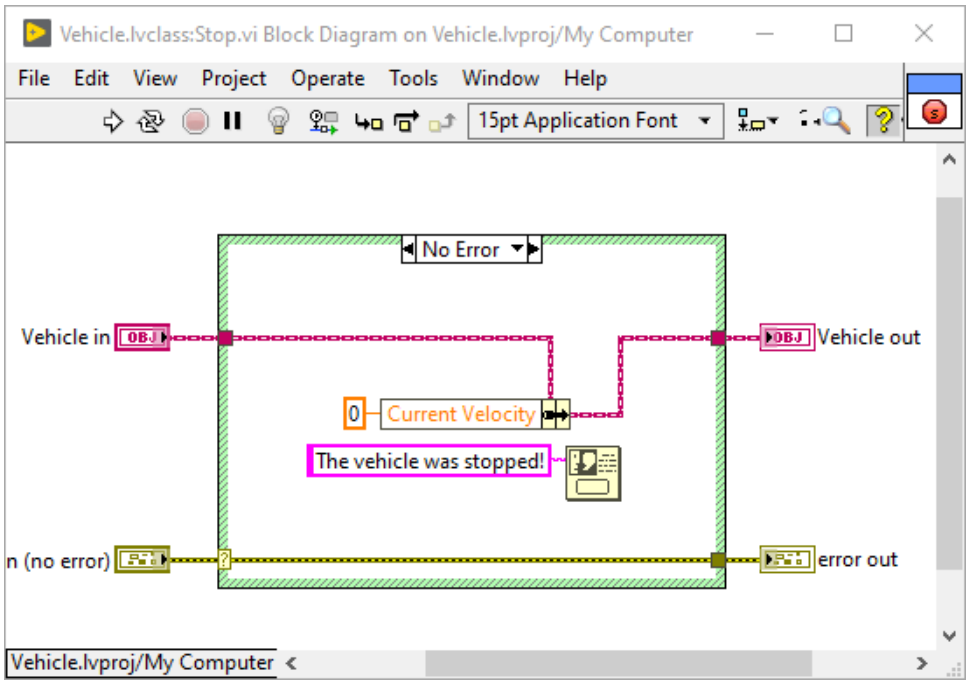
EXERCISE 2 Static Methods

GOAL

Add *Stop* method to the class created in the previous step. Method should set *Current Velocity* to 0 and display information about stopping the vehicle.

IMPLEMENTATION

1. Create static *Stop* method.
   - Open *Vehicle* project from previous step.
   - From context menu select *New →VI from Static Dispatch Template*, name it *Stop* and save.
   - On the BD of this method, use *Bundle by Name* function in order to set *Current Velocity* to 0.
   - Use *One Button Dialog* function for displaying message about stopping the vehicle.



END OF EXERCISE

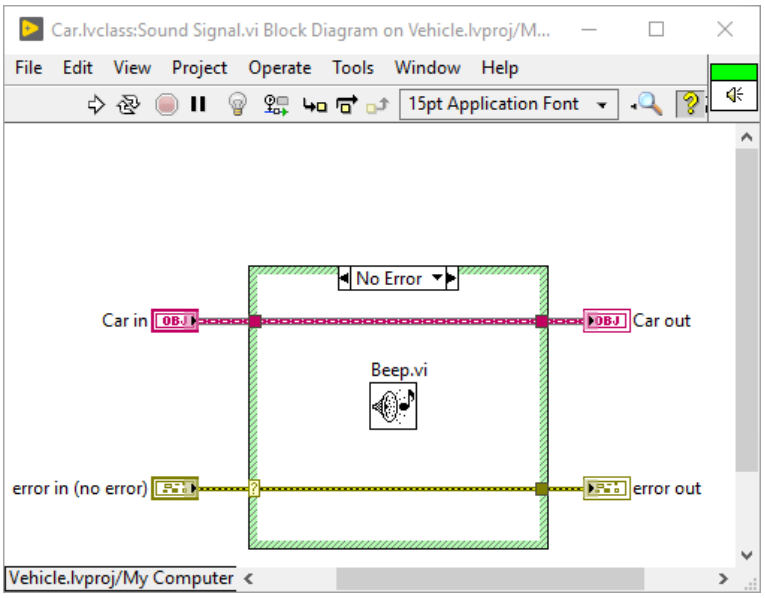
EXERCISE 3 Inheritance

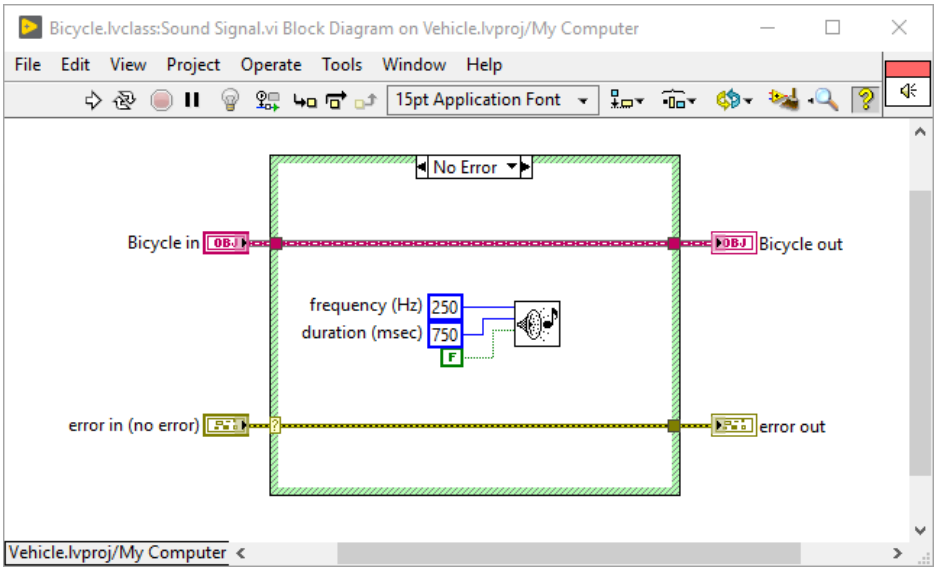
GOAL

Add *Car* and *Bicycle* classes that inherit from *Vehicle* to the previously created project. Add dynamic *Sound Signal* method that will play different sound for *Car* and for *Bicycle*.

IMPLEMENTATION

1. Create child classes.
   - Open *Vehicle* project from previous steps.
   - Under *My Computer*, add new class named *Car*.
   - From context menu, open *Properties* of the newly created class. Go to the *Inheritance* tab and select *Change Inheritance*.. option. Select *Vehicle* as the parent class.
   - Repeat two previous steps for *Bicycle* class.
2. Add dynamic method.
   - From the context menu of parent class, select *New → VI from Dynamic Dispatch Template*. Name it *Sound Signal* and save it.
   - From the context menu of *Car* class select option *New →VI for Override…* and select created earlier dynamic method.
   - Switch to the BD and remove calling parent method.
   - Use *Beep* function, available on functions palette, in order to play the sound. Clear the VI and save it.
   - Override the same method for the *Bicycle* class. Use the same function to play the sound, but configure it differently so the car and the bicycle sounds are not the same.



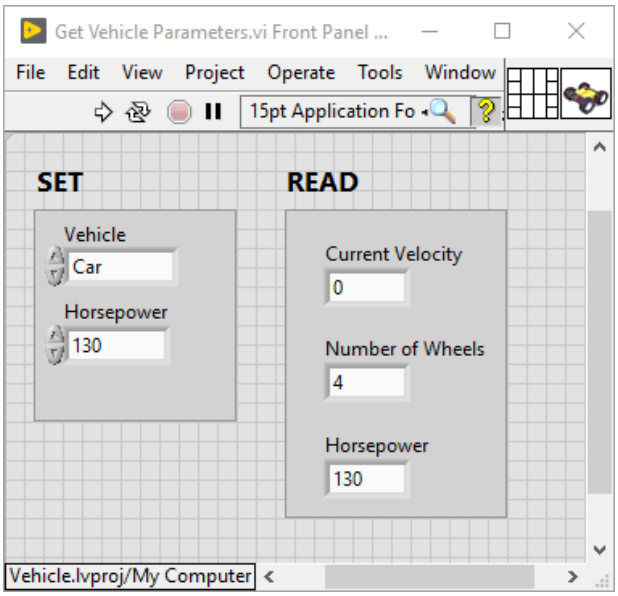


END OF EXERCISE

EXECRISE 4 Class Casting

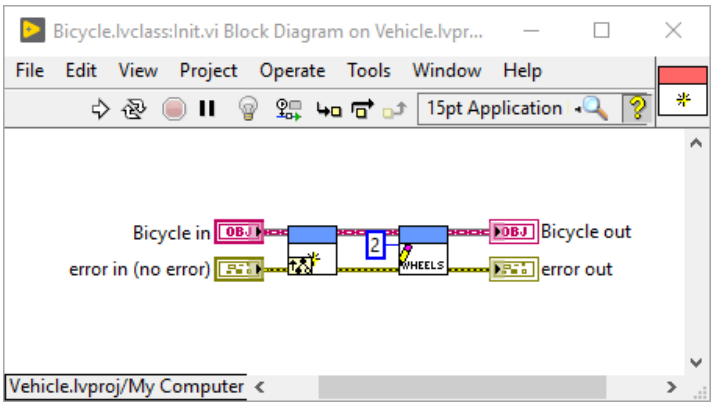
GOAL

Add *Horsepower* parameter to the *Car* class. Prepare a VI that will allow to generate one of the vehicle types created in previous steps. The VI should also display all parameters of created vehicle.

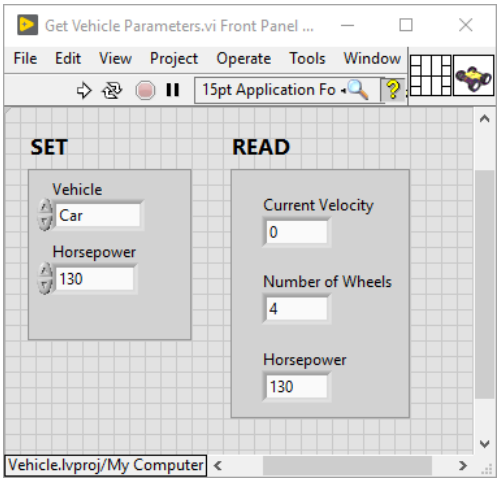


IMPLEMENTATION

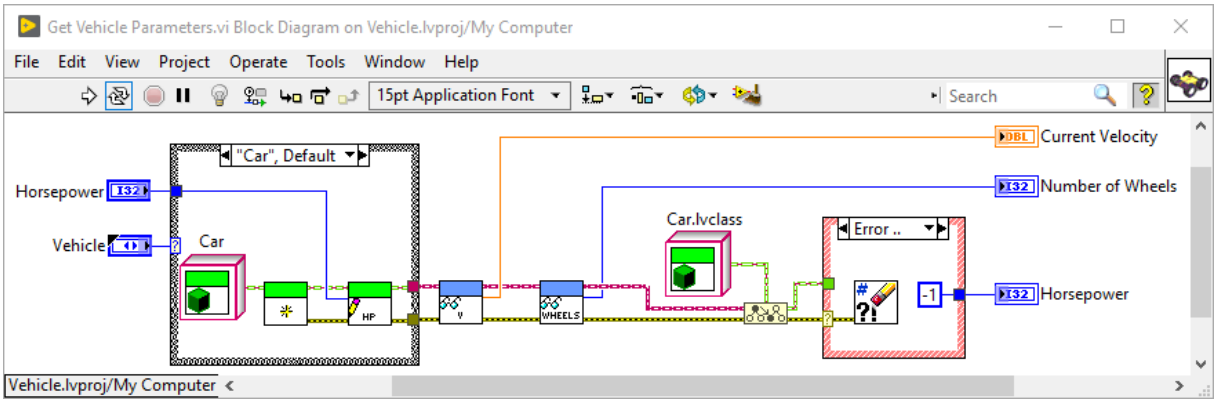
1. Add methods and accessors to the classes.
   - Open *Vehicle* project from previous exercises.
   - Add numeric parameter named *Horsepower* to the private data cluster of the *Car* class.
   - Create accessors (read and write) for both fields of the *Vehicle* class (inside the cluster of class private data open context menu and select *Create Accessor…*).
   - Similarly, create accessor for *Horsepower* value.
   - Create dynamic *Init* method for the *Vehicle* class. Inside this method, *Current Velocity* should be set to 0.
   - Override *Init* method for *Car* class. Leave calling parent method on the BD (this will execute *Init* method of *Vehicle* class). Set *Number of Wheels* to 4 using appropriate accessor.
   - Similarly, add *Init* method to the Bicycle class and set *Number of Wheels* to 2.



2. Prepare VI for generating specific vehicle.
   - Under *My Computer*, create new blank VI. Name it *Get Vehicle Parameters*.
   - Create new enum that includes available types of vehicles: *Car* and *Bicycle* and save it as typedef.
   - Prepare GUI of the VI so it includes two sections: first for configuring vehicle and second for displaying all vehicle parameters.



   - Switch to the BD of the VI. Draw *Case Structure* and connect *Vehicle* enum to its selector.
   - Configure two cases for both vehicle types. Inside each case, place constant of specific class by dragging appropriate item from project tree. In each case, call *Init* method.
   - Inside the case for *Car*, place additional accessor for writing horsepower. Configure it with value set by the user.
   - Outside the CASE, place appropriate accessors for reading *Current Velocity* and *Number of Wheels*.
   - Use *To More Specific Class* in order to check whether created class is a *Car*. Place another *Case Structure* and configure it with error wire. If function did not return error – read *Horsepower* using appropriate accessor. Otherwise, set horsepower to -1 and clear the error.



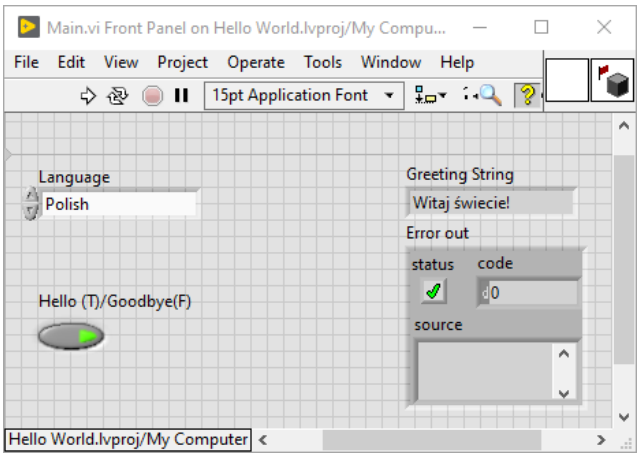
3. Test the VI and fix all detected issues.

END OF EXECRISE

EXERCISE 5 Hello, World!

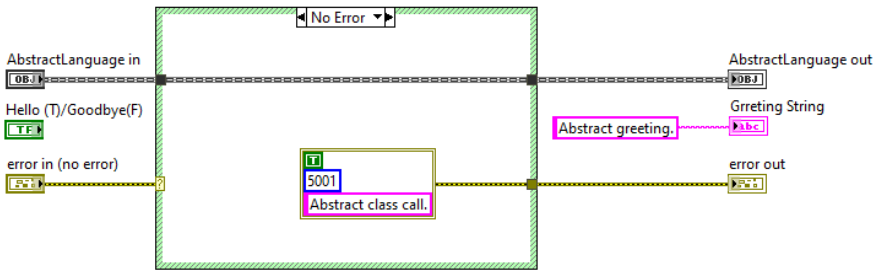
GOAL

Create simple application that displays greeting in language selected by the user. Use object oriented approach and support both hello and goodbye strings.

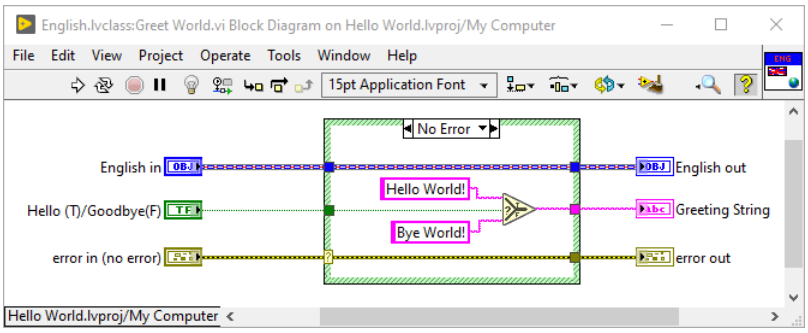


IMPLEMENTATION

1. Abstract language class.
   - Create new empty project.
   - Create new class and name it *AbstractLanguage*.
   - Create new dynamic method for this class and name it *Greet World*.
   - On the FP of the VI, prepare control for boolean *Hello (T)/Goodbye (F).* Prepare indicator for *Greeting String*. Connect input and outputs to the connector pane.
   - Switch to the BD of the VI and inside *No error*, generate custom error indicating that the abstract method was executed.

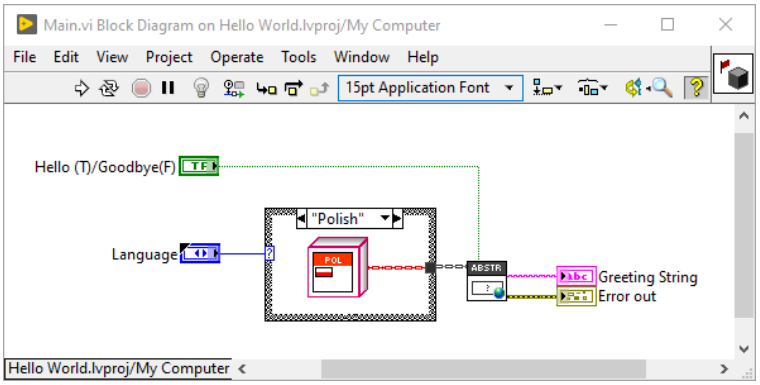


2. English and polish greetings.
   - Create new class for *English* and change its inheritance to *AbstractLanguage*.
   - Override method *Greet World* and remove parent method call.
   - Based on the boolean value, pass appropriate greeting string to the method output.



- Similarly, add class for Polish and prepare *Greet World* method.

3. Main VI.
   - Prepare enum containing available languages and save it as typedef.
   - Create new VI and prepare GUI so it allows to choose the language and hello/goodbye string. Prepare indicator for greeting string and for error out.
   - Switch to the BD of the VI and draw *Case Structure*. Configure it with language enum control.
   - Inside the CASE, place appropriate constant of class. After the CASE, place Greet World method from the abstract class.



- Test Main VI and fix all detected issues.

END OF EXERCISE

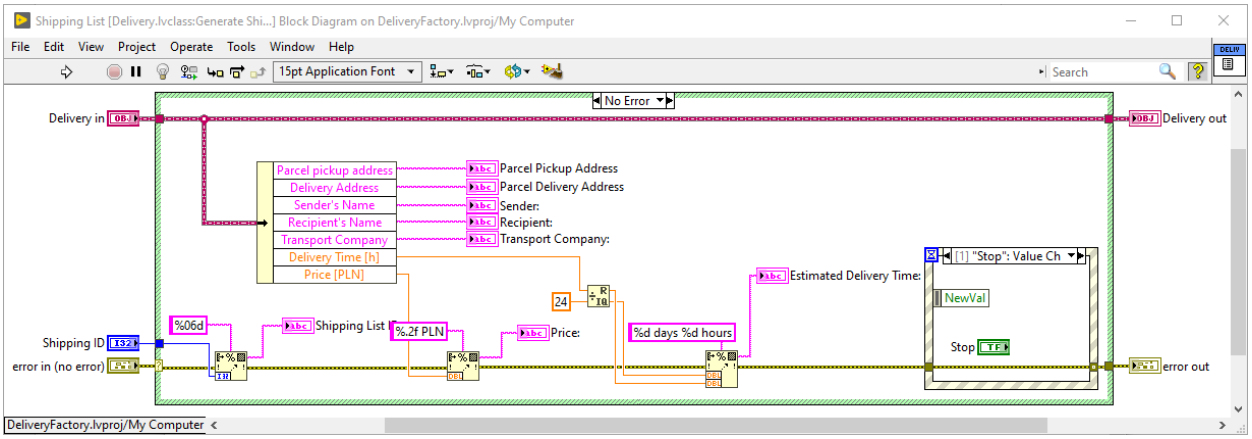
DELIVERY SERVICE 1 Delivery Class

GOAL

Implement parent *Delivery* class needed for *Delivery Service* project. Use provided VIs and controls.

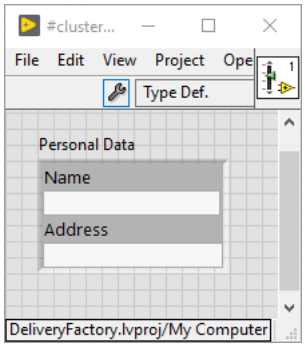
IMPLEMENTATION

1. Familiarize yourself with the exercise starting point.
   - Go to ...*Exercises**Project 1 – Delivery Service**Delivery Service 1*) and copy the project to your working directory.
   - Open the project and analyze provided components:
     i. *Main.vi* with prepared GUI,
     ii. *Delivery* class with fields, accessors and *Generate Shipping List* method,
     iii. *Configuration Data* and *Conrols*.
2. Finish implementation of *Delivery::Generate Shipping List* method.
   - Use *Unbundle by Name* function in order to read class parameters and fill the following indicators: *Parcel Pickup Address, Parcel Delivery Address, Sender, Recipient, Transport Company*.
   - Extend unbundle function in order to read *Delivery Time [h]* and *Price [PLN]*.
   - Format price value into string and fill *Price* indicator - suggested format: 12.50 PLN.
   - Similarly, format delivery time value into string and fill *Estimated Delivery Time* indicator - suggested format: %d days %d hours (useful function: *Quotient & Remainder*).
   - Fill *Shipping List ID* indicator in a way that every ID has 6 digits (e.g. *00001* instead of *1*).
   - After initialization of indicators, draw *Event Structure*. Add event case for *Stop* button (Value Change).

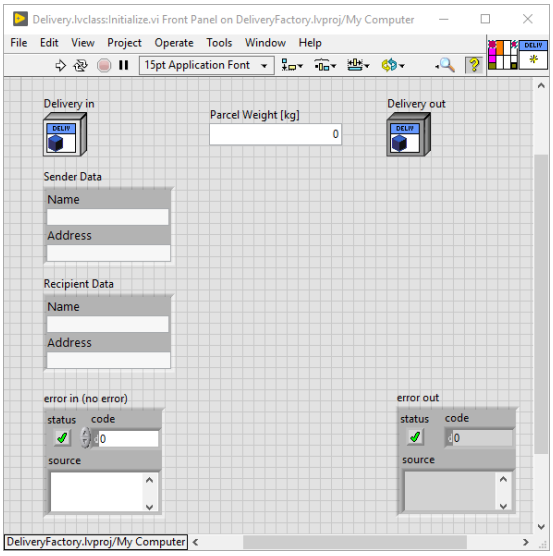


3. Implement *Initialize* method.

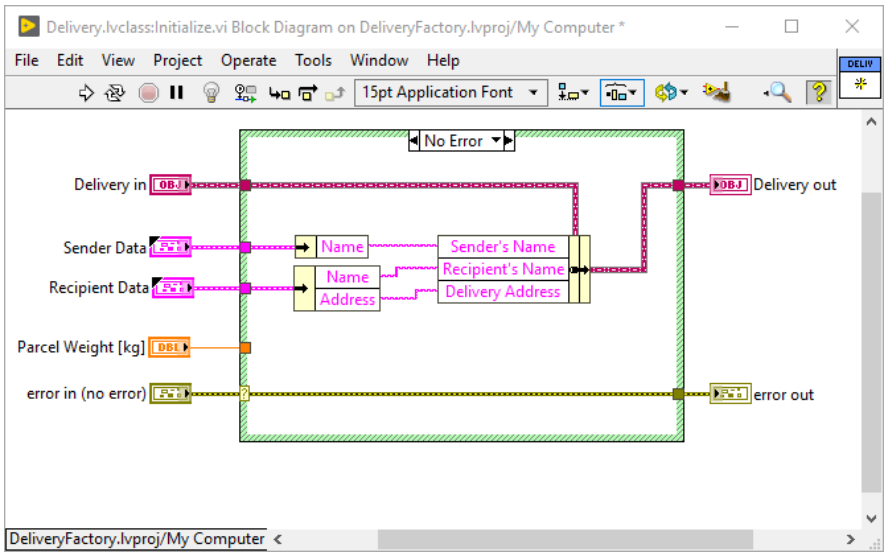
- Create new type definition under the *Controls* virtual folder. Add cluster and fill it with two string controls: *Name*, *Address*. Name cluster *Personal Data* and save typedef as *#cluster_PersonalData.ctl*.



- Create new method of *Delivery* class from dynamic dispatch template (*RMB → New → VI from Dynamic Dispatch Template*). Name it *Initialize*.
- Prepare three inputs (controls) for this method: *Parcel Weight [kg]* (DBL type), *Sender Data* (PersonalData cluster) and *Recipient Data* (PersonalData cluster). Configure Connector Pane of the VI.



- Using *Bundle by Name* function, prepare inputs for three class fields: *Sender's Name*, *Recipient's Name*, *Delivery Address*. Fill them using information from appropriate controls.



END OF EXERCISE

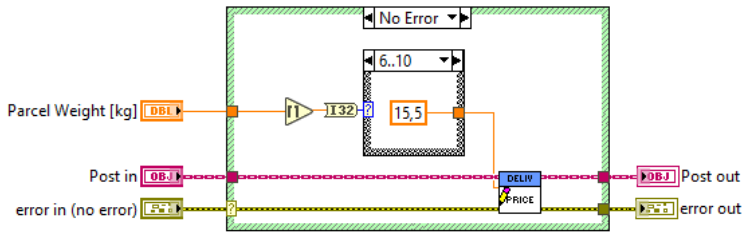
DELIVERY SERVICE 2 Specific Delivery Types

GOAL

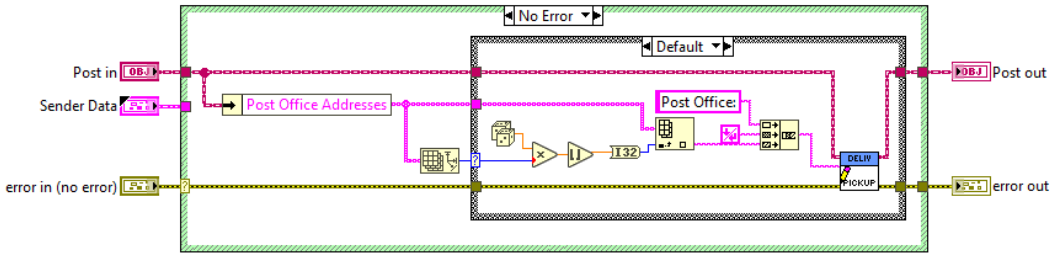
Create and implement classes inheriting from *Delivery* for three different delivery types: *Post*, *Courier*, *Pigeon*.

IMPLEMENTATION

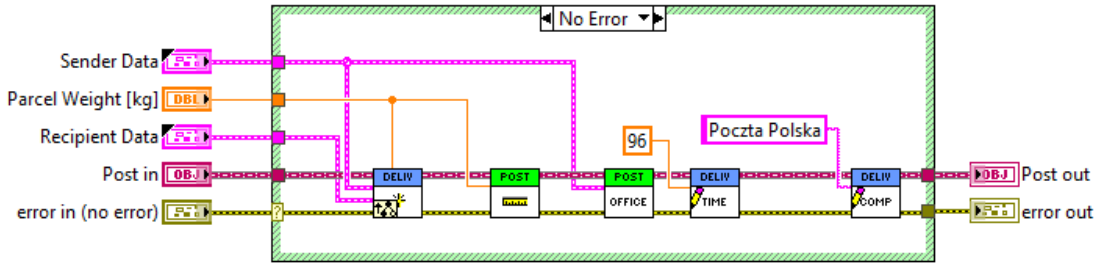
1. Implement *Post* class.
   - Create new class for *Post* and change its inheritance to *Delivery*. Add new field to the *Post* class private data cluster for *Post Office Addresses* (1D Strings Array).
   - Add *SetPrice* method to the class. This method should have input for parcel weight and should write appropriate price to the class based on this weight. The heavier the parcel, the higher price. You can decide about specific prices.



   - Add *ChoosePickupAddress* method. This method should use post office addresses list from the class data and choose the one that is most suitable for the parcel sender. The chosen post office will be parcel pickup address. Update appropriate filed in the class. As algorithm for choosing suitable office you can simply choose randomly one of the offices from the list. If list of offices is empty, generate custom error.

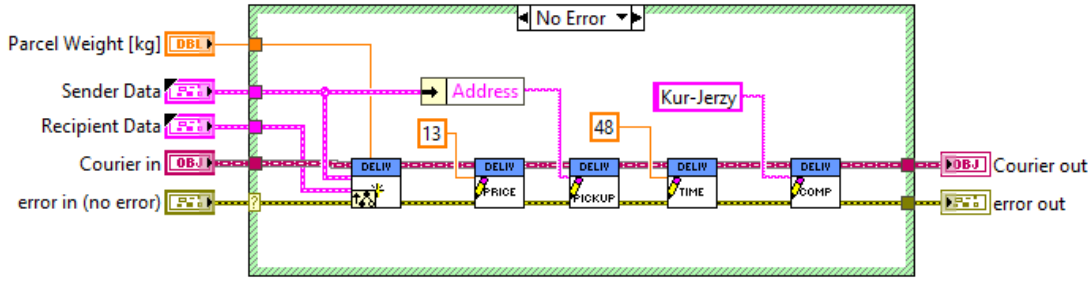


   - Override *Initialize* method for *Post* class, do not remove calling parent method. After that, following actions should be performed (using appropriate supporting methods and accessors): setting price, choosing pickup address, setting delivery time (can be hardcoded), setting transport company name (e.g. *Poczta Polska*).

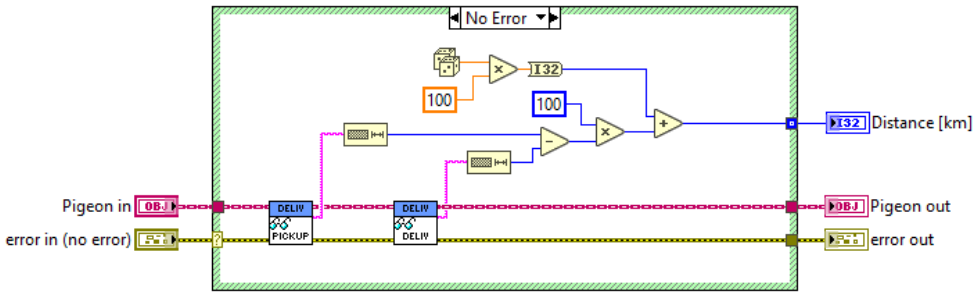


- Create accessors for *Post Office Addresses* filed. This accessor will be used in a future by the factory class.

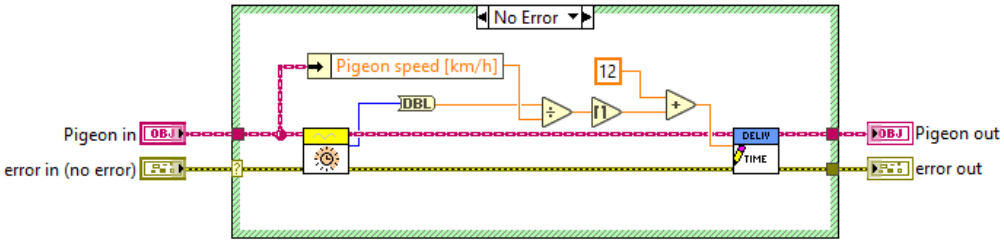
2. Implement *Courier* class.
   - Create new class for *Courier* and change its inheritance to *Delivery*.
   - Override *Initialize* method for *Courier* class, do not remove calling parent method. This method should perform following actions: setting price (can be hardcoded), setting pickup address (equal to the sender's address), setting delivery time (can be hardcoded) and setting transport company name.



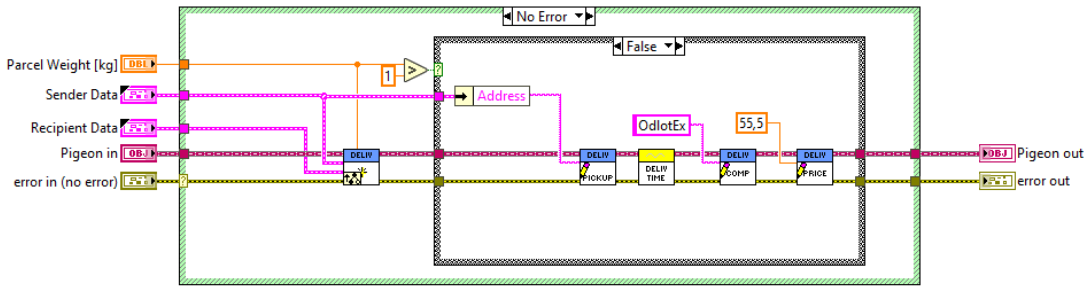
3. Implement *Pigeon* class.
   - Create new class for *Pigeon* and change its inheritance to *Delivery*. Add new field to the *Pigeon* class private data cluster: *Pigeon speed [km/h]* (DBL).
   - Create accessors for new filed. They will be used in a future.
   - Add static *Calculate Distance* method. This method should calculate distance in kilometers between pickup and delivery addresses (read them from class). Simulate real calculation with any algorithm – e.g. distance can be random number from some range or difference in strings length.



- Add static *Calculate Delivery Time* method. This method should calculate delivery time based on distance and pigeon speed. Read necessary parameters from the class and use them for calculation. Resulted time write to the class using accessor.



- Override *Initialize* method for *Pigeon* class, do not remove calling parent method. First, this method should check parcel weight and generate custom error if it is higher than 1 kg. Otherwise, method should initialize class by setting pickup address, calculating delivery time, setting transport company name and setting the price (can be hardcoded).

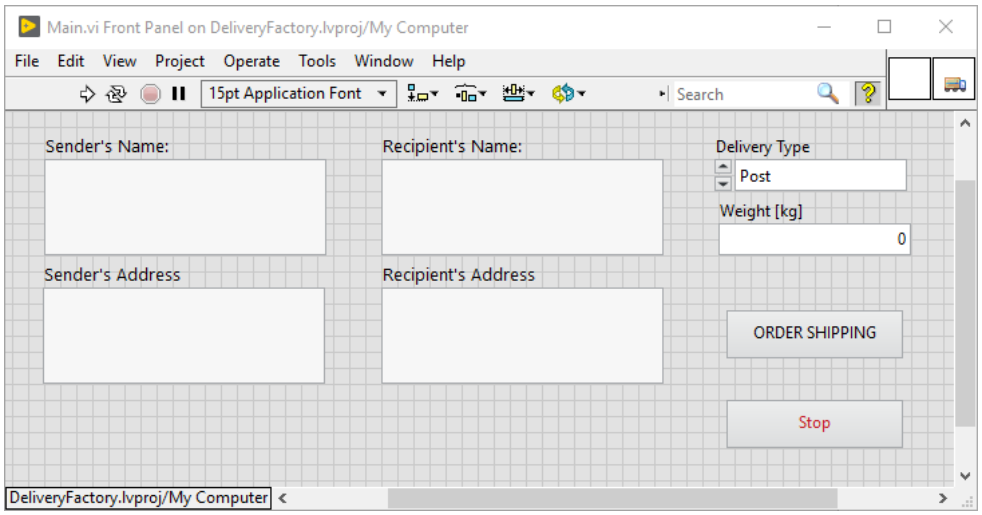


END OF EXERCISE

DELIVERY SERVICE 3 Factory and Main Window

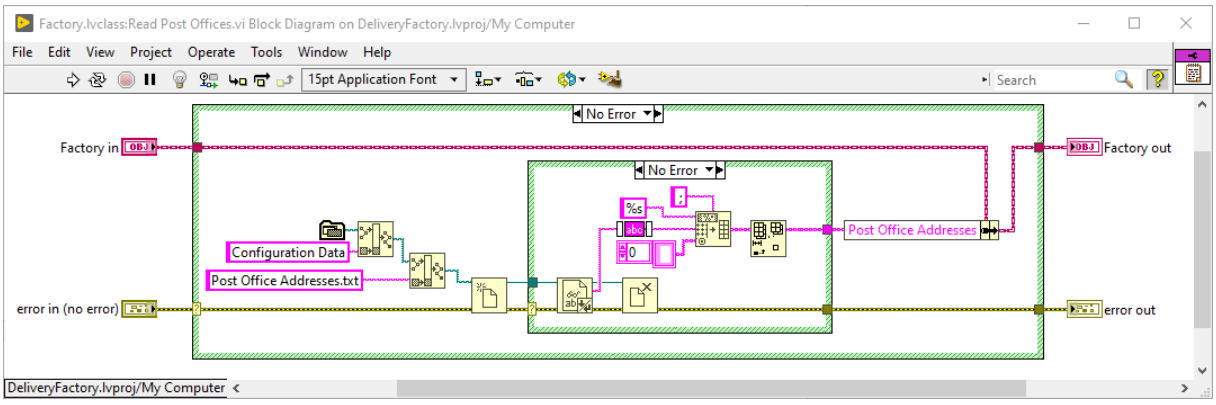
GOAL

Create *Factory* class that will produce specific delivery services. Finish implementation of Main.vi so it allows to configure and generate shipping list.

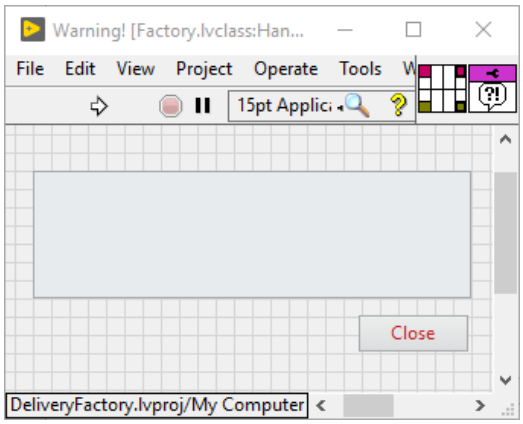


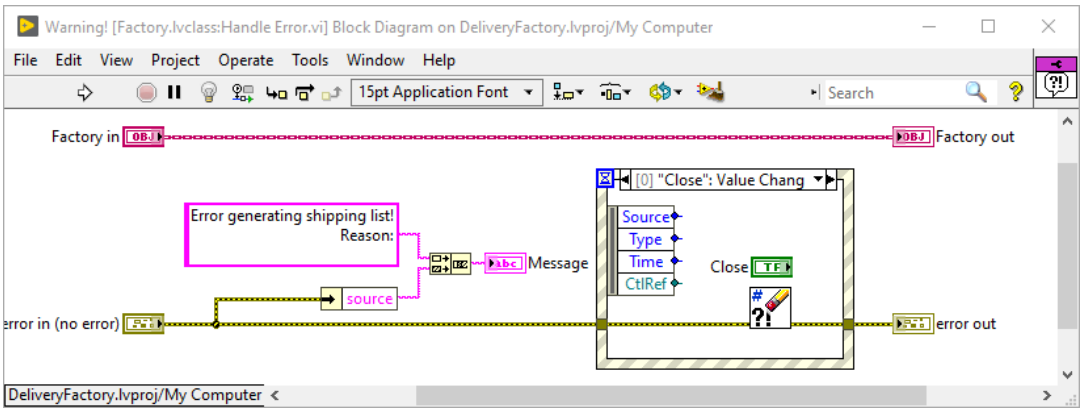
IMPLEMENTATION

1. Create *Factory* class and prepare supporting VIs.
   - Add new class to the project under virtual folder *Factory*.
   - Add class fields to the private data cluster:
     i. *Next List ID* (I32),
     ii. *Delivery History* (1D array of *Delivery* objects),
     iii. *Post Office Addresses* (1D array of strings).
   - Add new static method *Read Post Offices*. This method should read *Post Office Addresses.txt* file that is located under *Configuration Data* subdirectory. File is formatted with one office address per line. Update appropriate field of the *Factory* class. If error occurs during opening file, set empty array as a list of offices and clear the error.

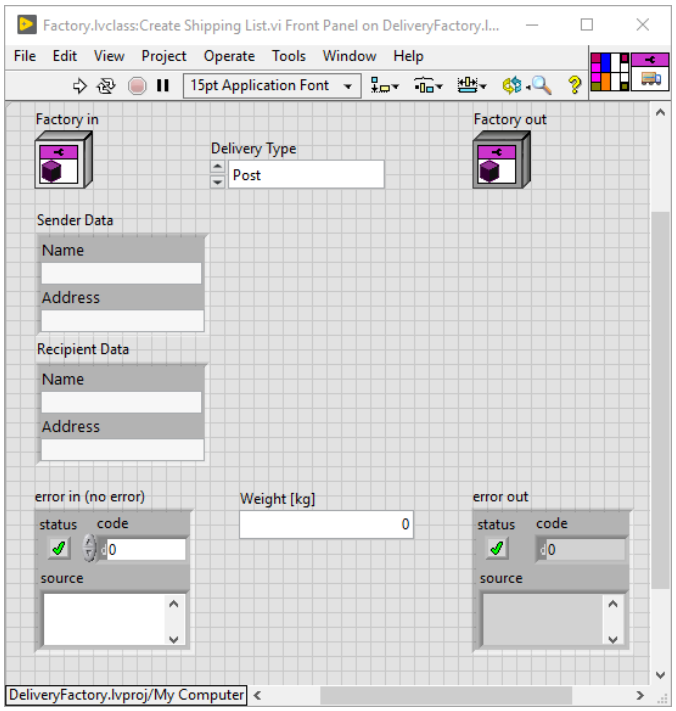


- Add new static method *Handle Error*. This method should display information about error during generating shipping list. Window should be closed after clicking *Close* button.

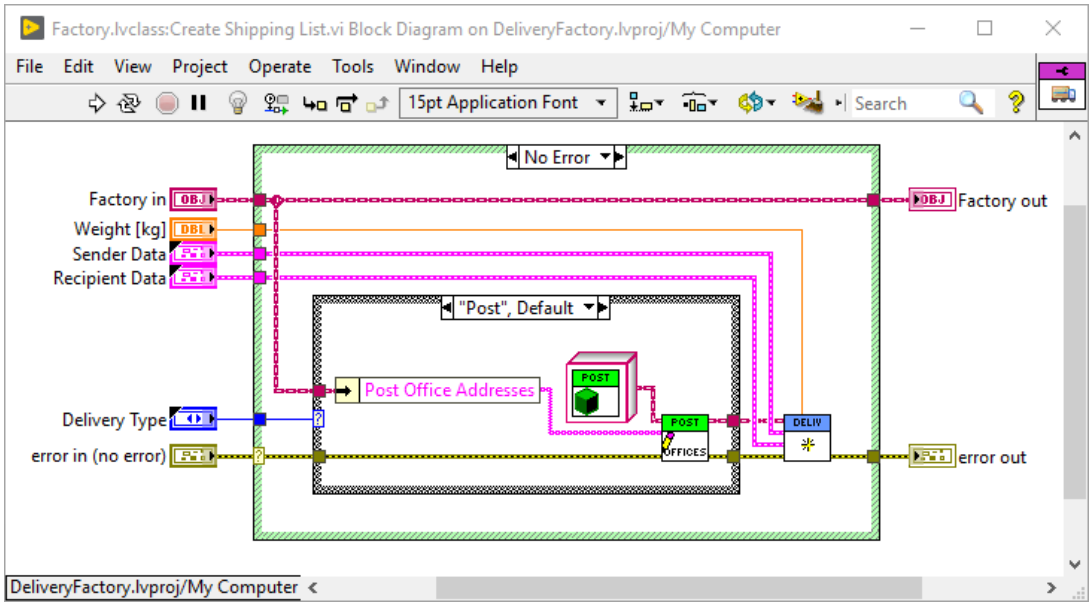




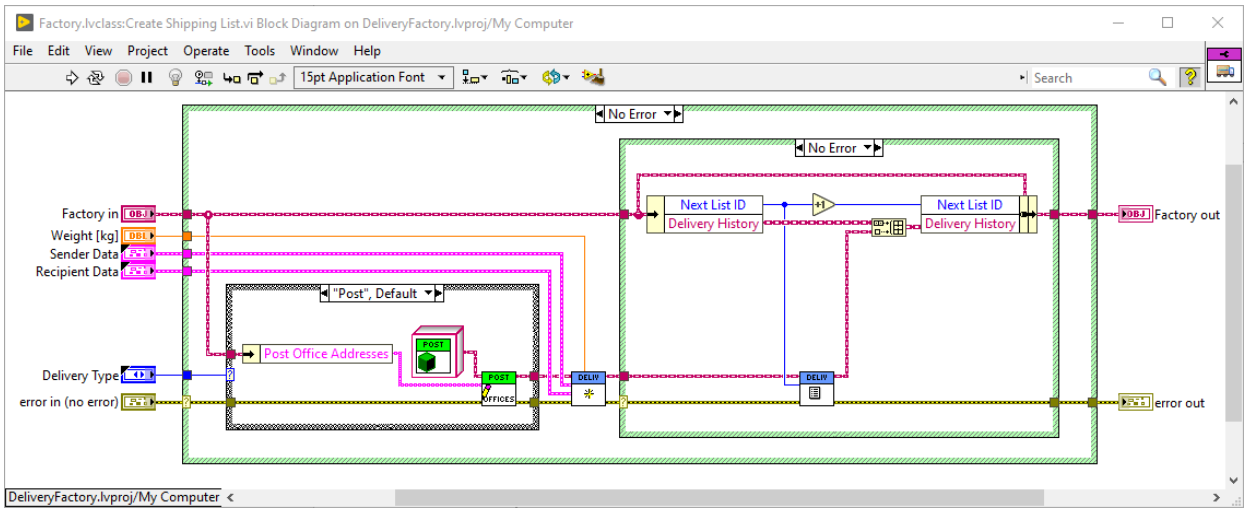
2. Implement *Factory::Create Shipping List*.
   - Add new static method called *Create Shipping List* to the Factory class.
   - Prepare controls for this method: *Sender Data* (*PersonalData* cluster), *Recipient Data* (*PersonalData* cluster), *Delivery Type* (*DeliveryType* enum, available in *Controls* folder), *Weight [kg]* (DBL).



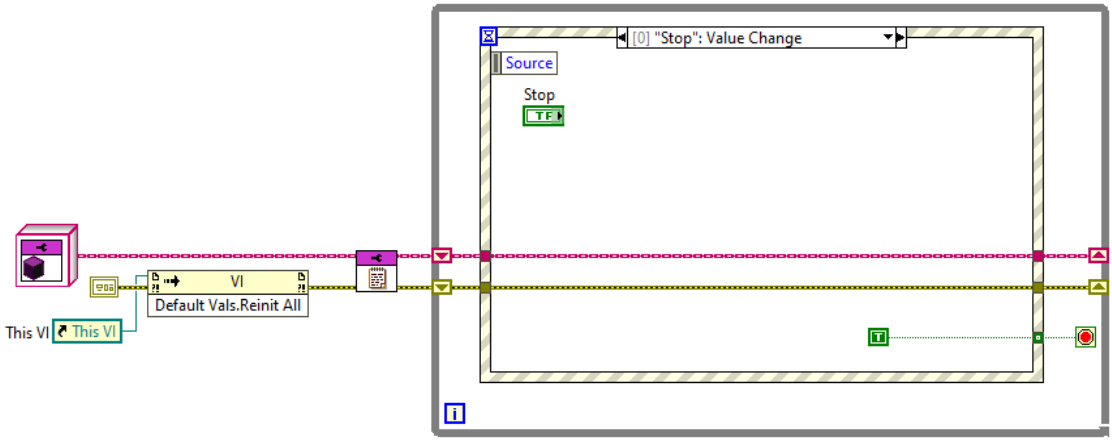
- Go to the BD and implement initialization of specific *Delivery* class based on the *Delivery Type* value. For *Pigeon* class, set pigeon speed before *Initialize* method. For *Post*, set list of office addresses before *Initialize* method.



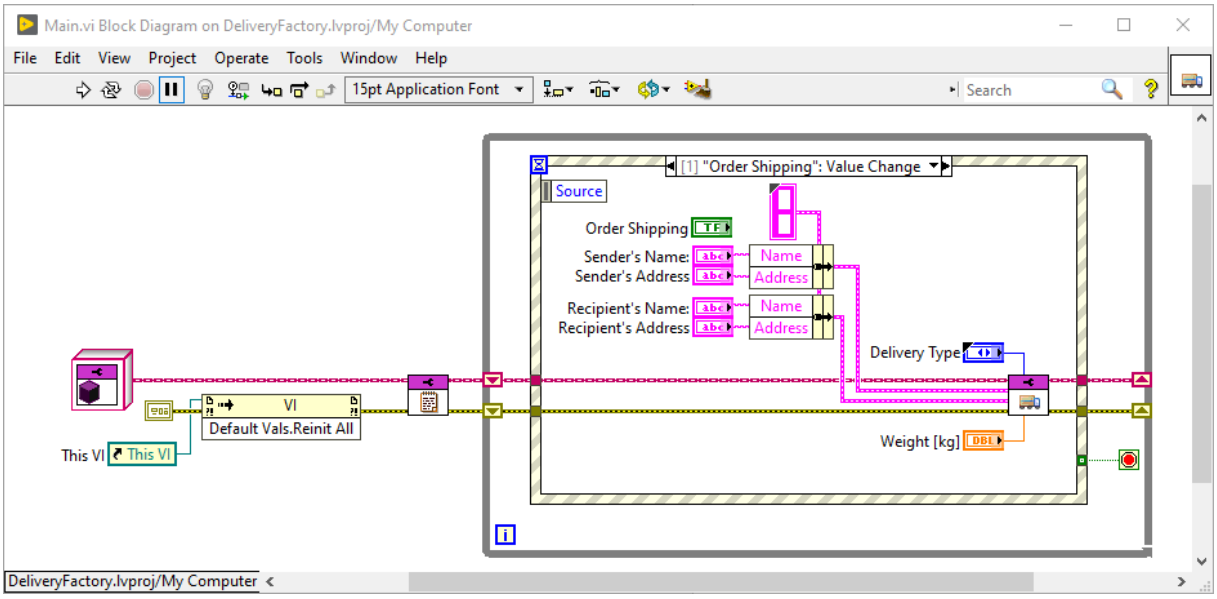
- After class initialization, draw *Case Structure* and connect error wire to it. If error occurred, execute *Factory::Handle Error* method. If no error occurred, execute *Delivery::Generate Shipping List* method and increment *Next List ID* value. Add generated class to the *Delivery History* array.



3. Implement *Main.vi*.
   - Familiarize with FP of *Main.vi*.
   - Go to the BD and implement VI initialization before the WHILE. Use *Invoke Node* with *This VI* reference connected in order to execute *Default Values* → *Reinitialize All To Default* method.
   - During initialization process, add initialization of *Factory* class. Place class constant and execute *Read Post Offices* method. Create shift register for this object inside the WHILE.



   - Add event case for *Order Shipping* button (Value Change). In this case, execute *Factory::Create Shipping List* method. Build all necessary inputs using controls prepared on *Main.vi*. *Factory* object outputted from this method pass to the shift register.



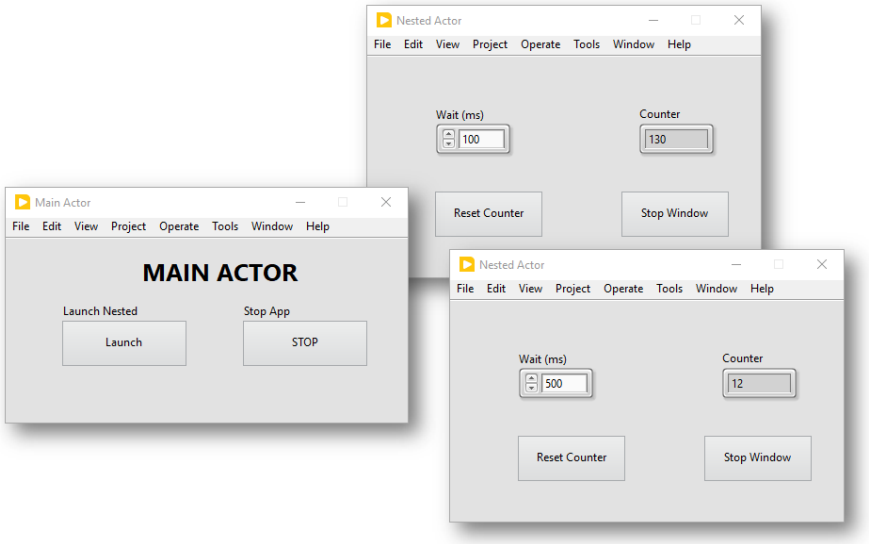
4. Test the application and fix all detected issues.
   - Try to generate various shipping lists. Does the application increment list IDs?
   - Can you force errors with *Post* (empty list of offices) and *Pigeon* (too heavy parcel)?
   - Clean-up your code and organize your project tree. Do your virtual folders and VI locations are reflected on disk?

END OF EXERCISE

EXERCISE 6 My First Actor

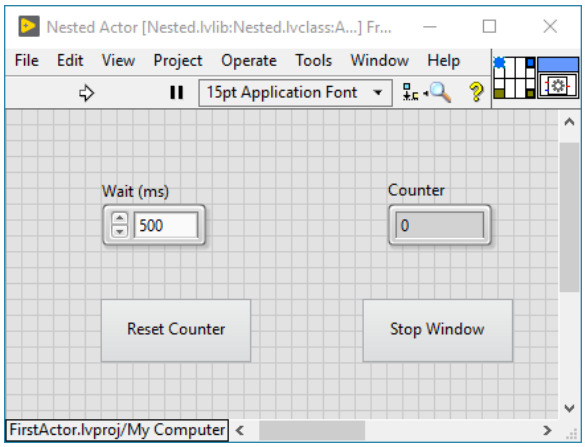
GOAL

Create application that launches one root actor. This actor should allow to launch any number of nested actors. It should be allowed to close single nested actor by clicking *Stop Window* button. After closing main (root) actor, all nested actors should be closed. Nested actors should display loop count and users should be able to change loop execution time and reset the counter.

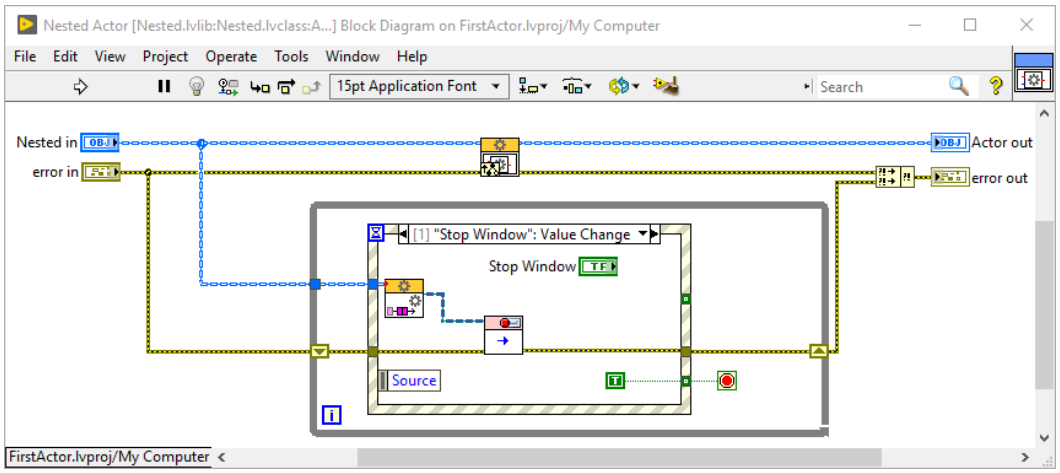


IMPLEMENTATION

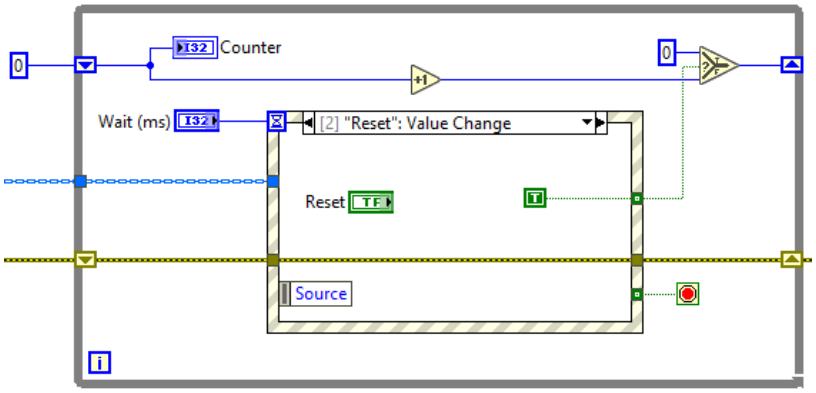
1. Implement class for nested actors.
   - Create new blank project and save it.
   - Under *My Computer*, create new actor (use option from context menu). Name the actor *Nested*.
   - Override *Actor Core* method. Open the VI and go to the *VI Properties* (RMB on VI icon). Go to *Window Appearance* and select *Top-level application window*). Set *Window title* to *Nested Actor*.
   - On the FP of *Actor Core*, prepare necessary controls and indicators: *Wait (ms)* (I32 control), *Counter* (I32 indicator), *Reset Counter* and *Stop Window* buttons.



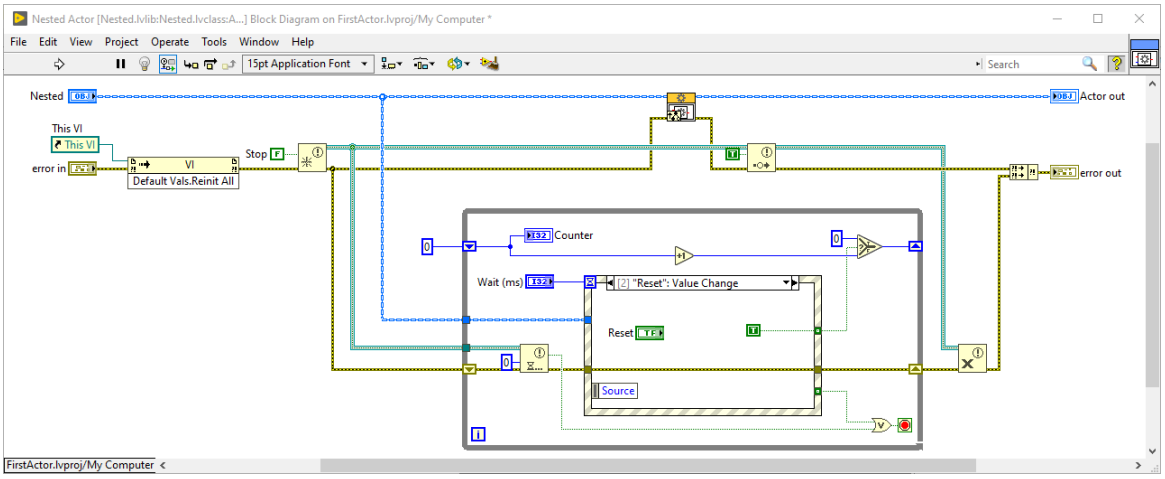
- Go to the BD and draw *While Loop* for GUI support. Add *Event Structure* inside it and configure event case for *Stop Window* button. Inside this case, send normal stop message to self (use *Read Self Enqueuer* and *Send Normal Stop* functions). Stop the GUI loop as well.



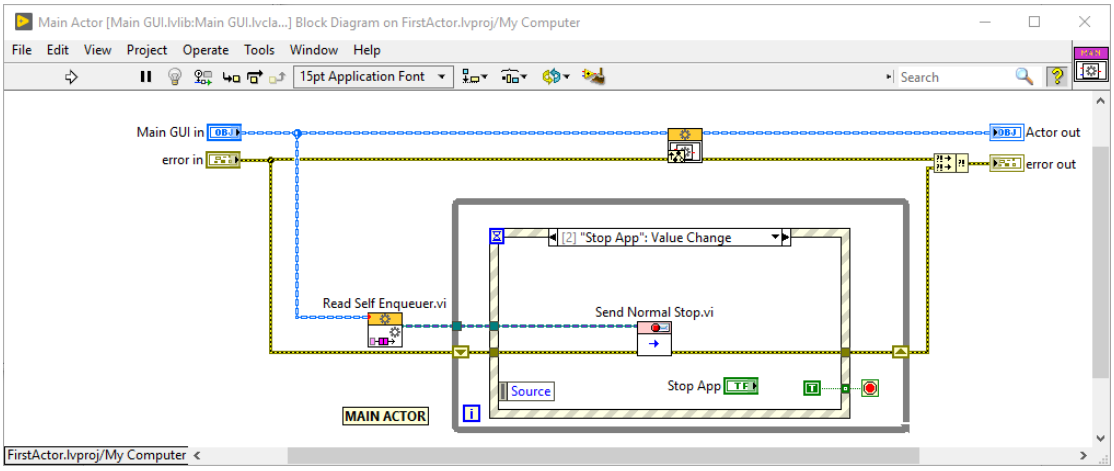
- Connect *Wait (ms)* control to the *Event Timeout* terminal (upper left corner) and leave Timeout case empty. Set default value for *Wait (ms)* control to 500 (on the FP select *Data Operations → Make Current Value Default* for this control.
- Prepare shift register for I32 value and initialize it with 0. In every loop iteration, display its value using *Counter* indicator and increment it.
- Add event case for *Reset* button. Return *True* value from it (create tunnel on event structure border). Outside the event structure, place *Select* function. Configure it by the value returned from the structure. Each time the *Reset* button is clicked, counter value should be set to 0. Otherwise, value should be passed unchanged.



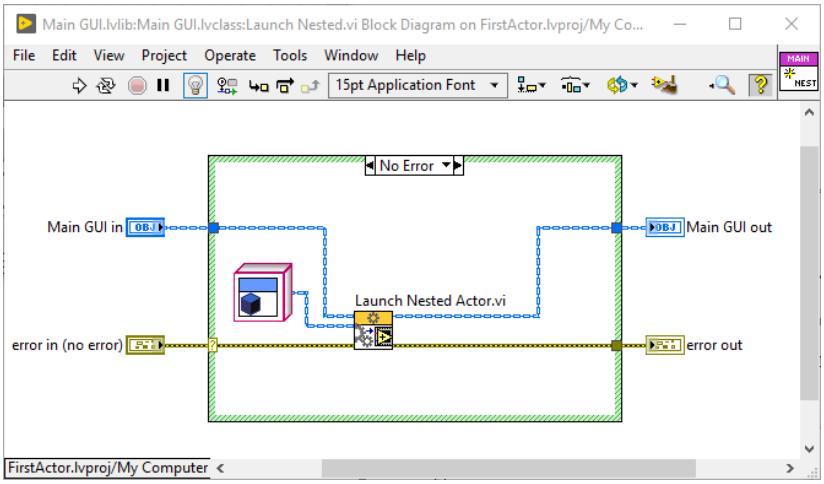
- Add initialization logic before the both threads. Initialize FP to default values (use *Invoke Node* with *This VI* reference). Place *Obtain Notifier* function and configure it with boolean type.
- After *Actor Core* parent method (thread responsible for processing messages), send *True* notification using created notifier.
- Inside the GUI loop, wait for notification (set timeout to 0). If *True* notification arrived, stop the loop.
- Right after GUI loop, place *Release Notifier* function and release created notifier.



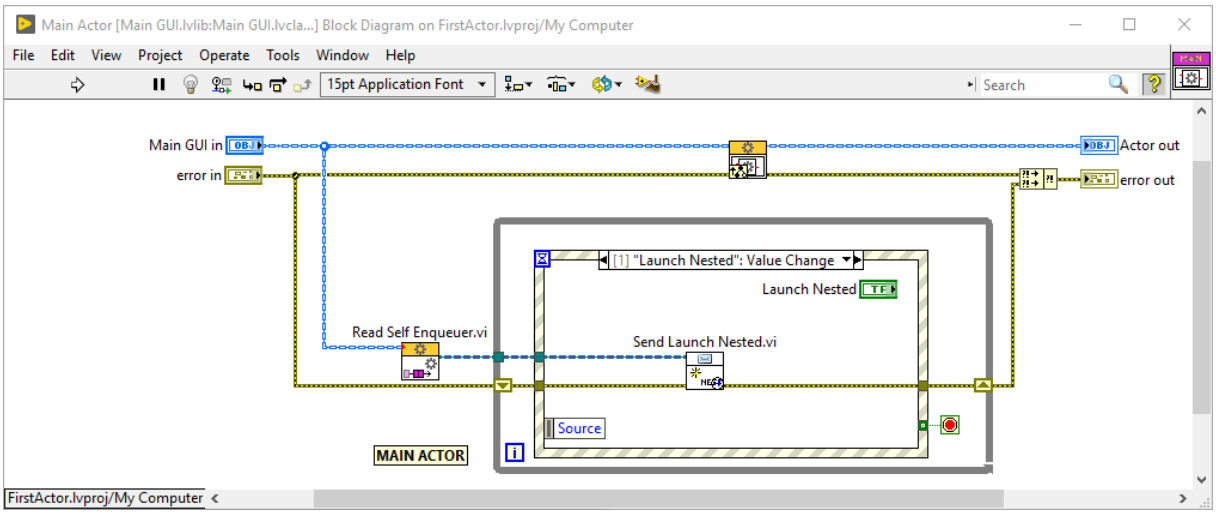
2. Create main actor.
   - Create new actor and name it *Main GUI*. Override *Actor Core* method. Similarly as for previous actor, open *VI Properties* for *Actor Core* and set it as *Top-level application window*. Set appropriate title to the VI.
   - Prepare two button on *Actor Core* FP: *Launch Nested* and *Stop App*.
   - Draw *While Loop* for GUI support. Place *Event Structure* inside it. Add event case for *Stop App* button. Inside this case, stop the loop and send normal stop message to self enqueuer.



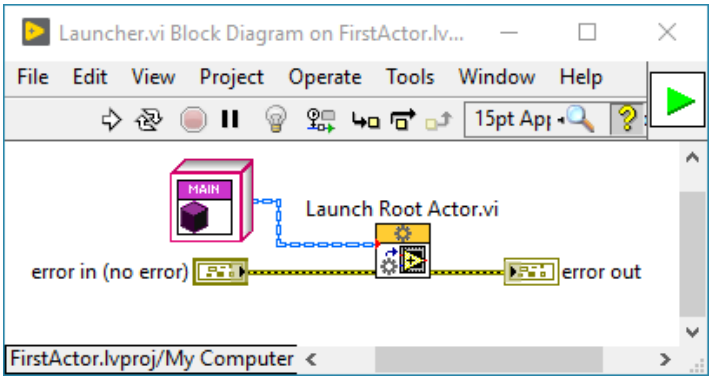
- Add new static method to the *Main GUI* class. Inside this method, implement launching nested actors (use *Launch Nested Actor* function).



- Create message for this method by opening its context menu in the project tree and selecting *Actor Framework → Create Message*.
- Go back to *Actor Core* method and add event case for *Launch Nested* button. Inside this case, send message for launching the nested actor. Message should be sent using self enqueuer (use appropriate message class method generated in a previous step).



3. Create Launcher.
   - Add new empty VI to the project and name it *Launcher*.
   - Inside this VI, use Launch Root Actor function in order to launch *Main GUI* actor.



   - Run *Launcher* and test the application. Fix all detected issues.

END OF EXERCISE

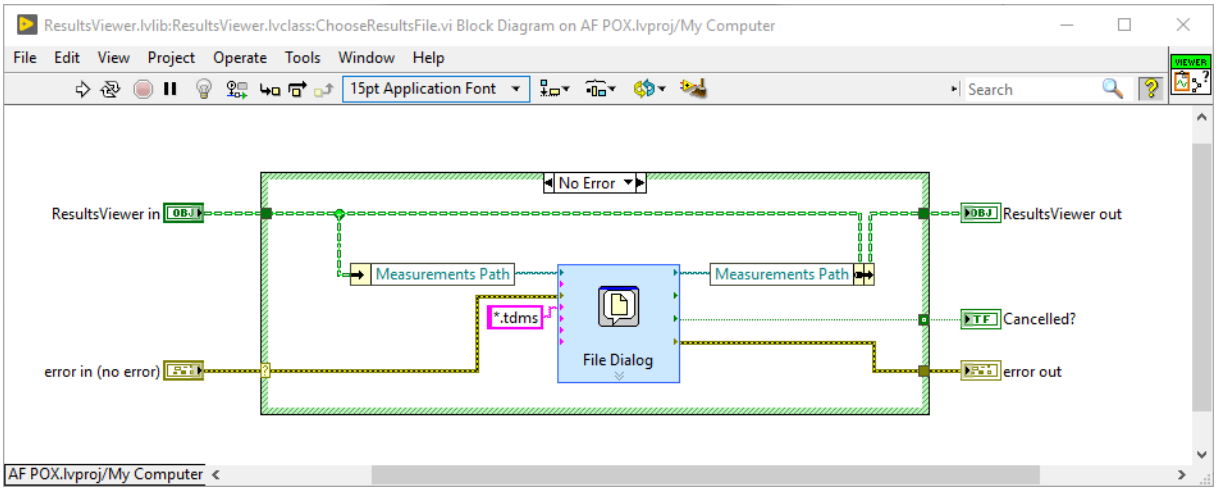
PULSE OXIMETER AF 1 Result Viewer

GOAL

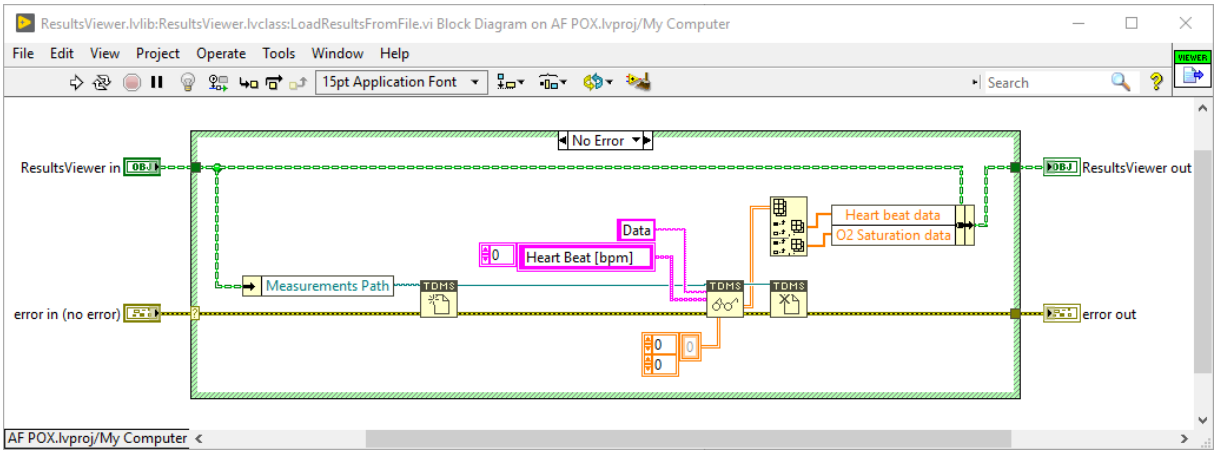
Modify Pulse Oximeter application created as a final project of GPL2 course. Prepare *Result Viewer* actor that will replace current asynchronous VI. Actor should handle opening and displaying measurement *.tdms files.

IMPLEMENTATION

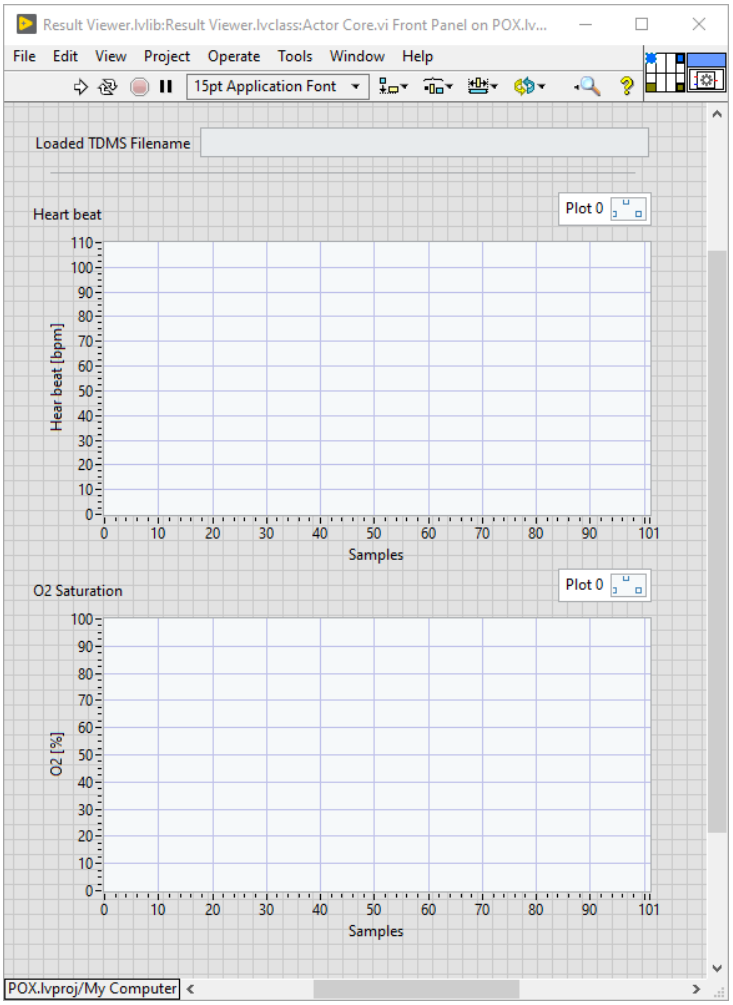
1. Familiarize yourself with the exercise starting point.
   - Go to *...\Exercises\Project 2 – AF POX\AF POX 1* and copy the project to your working directory.
   - Analyze current application status.
   - Test only *Result Viewer* functionality – there is no need for connecting hardware for this exercise. You can find examples of *.tdms files under *Measurements* folder in the project location.
2. Add *Result Viewer* actor class.
   - Create new actor and call it *Result Viewer*.
   - Open private data cluster of this class and add following controls: *Measurements Path* (path), *Heart beat data* (1D array of DBL), *O2 Saturation data* (1D array of DBL).
   - Create accessors for all class fields.
3. Implement *Choose Results File* method.
   - Add new static method and save it.
   - Implement method logic: open file dialog window (*File Dialog* function) and use selected path to update *Measurement Path* field of the class. Accept only *.tdms extensions. Method should also return information whether user canceled choosing the path.
   - Update connector pane of the VI so the method has output for cancelation flag.



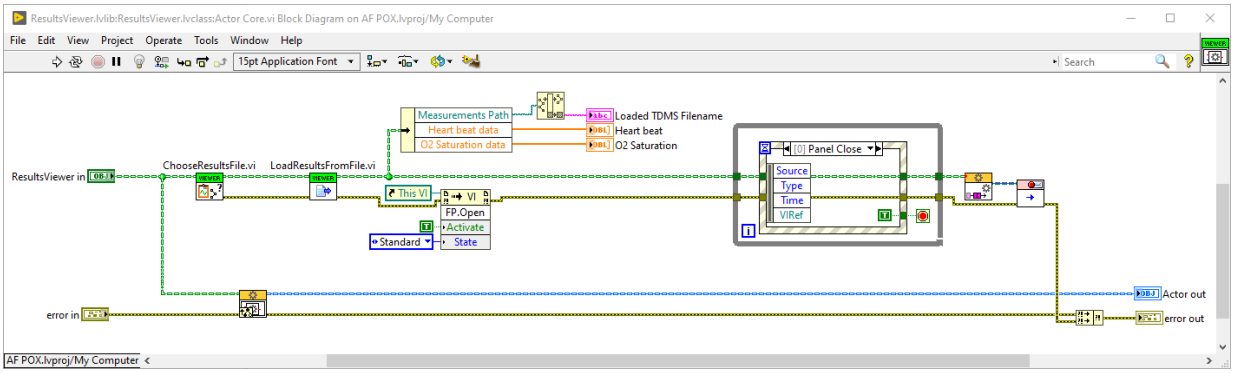
4. Implement *Load Results From File* method.
   - Add new static method and save it.
   - Implement method logic: using path saved inside the class, open TDMS file and read measurement data. Update appropriate class fields with this data.



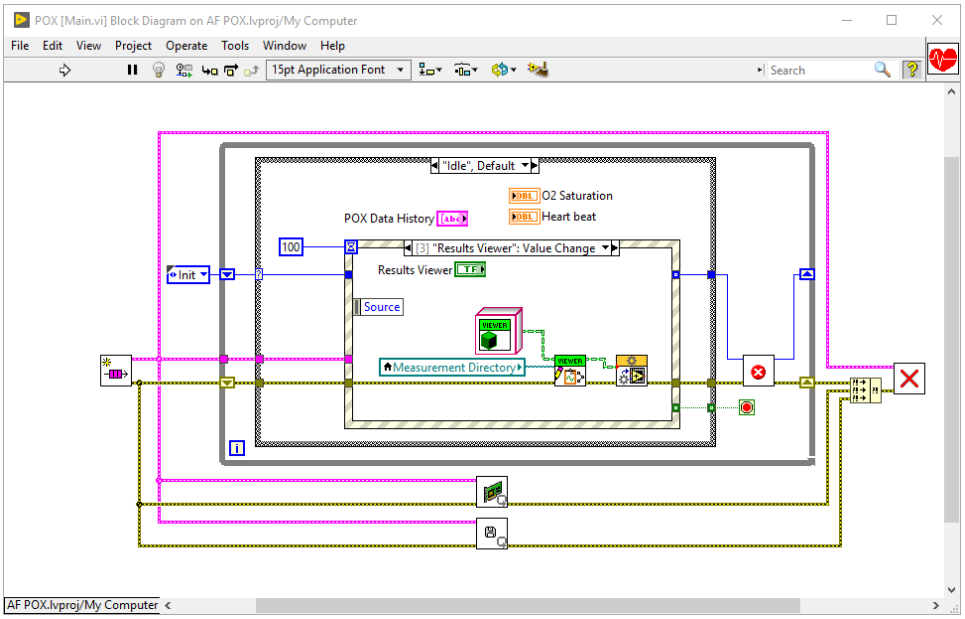
5. Modify *Actor Core* method.
   - Override *Actor Core* method. Prepare result viewer GUI by copying FP elements from previous version of the result viewer window.



- Go to the BD and add *While Loop* and *Event Structure* for GUI support.
- Add event case for *Panel Close* (choose *Panel Close* event, not *Panel Close?* since the default behavior for this event should be preserved).
- Inside this event, stop the while loop. Right after the loop, send normal stop message to self enqueuer in order to stop the other thread as well.
- Before the WHILE, add initialization code. Execute *Choose Results File* and *Load Results From File* methods (for now ignore scenario when user canceled choosing the file). Fill all the indicators with appropriate data read from the class.
- As the last step of the initialization process, place *Invoke Node* and connect to it *This VI* reference. Execute *Front Panel.Open* method. Set *Activate* to *True* and *State* to *Standard*.



6. Launching *Result Viewer* actor.
   - Open *Main.vi* and switch to BD. Go to *Idle* state and find event case for *Result Viewer* button.
   - Remove currently executed VI, but leave local variable for *Measurement Directory*.
   - Place constant of *Result Viewer* class and set directory using appropriate accessor. Launch this actor as root.
   - Run *Main.vi* and test *Result Viewer*. Fix all detected issues.



END OF EXERCISE

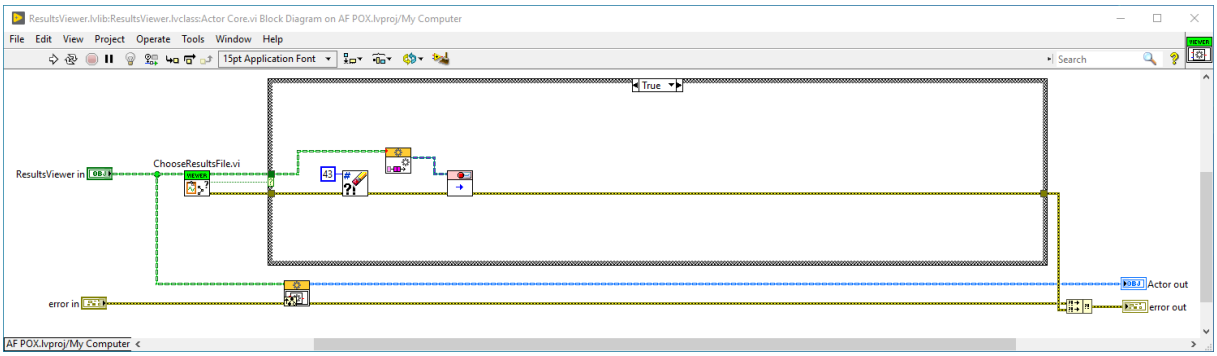
PULSE OXIMETER AF 2 Stopping Actors

GOAL

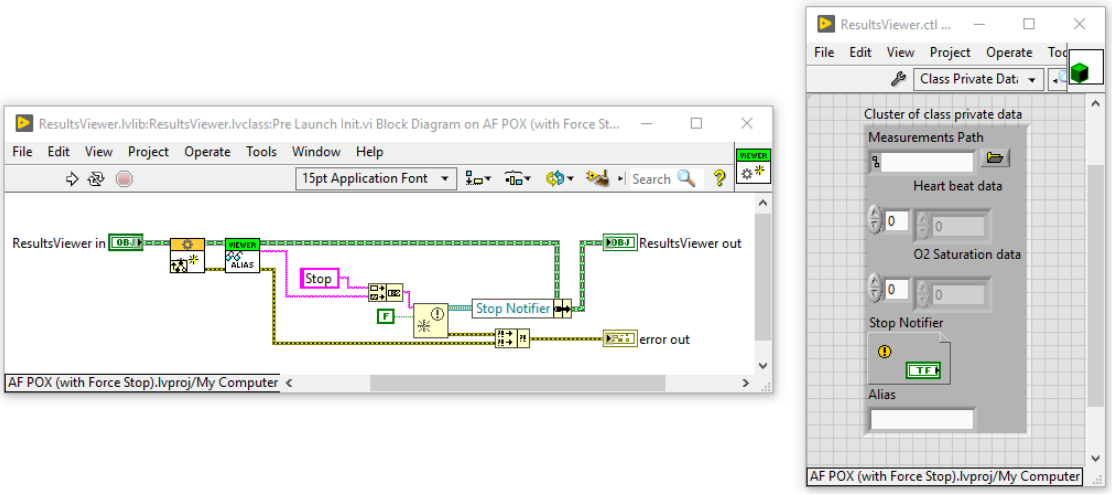
Modify the application so that all the *Result Viewer* windows are closed within the main application window.

IMPLEMENTATION

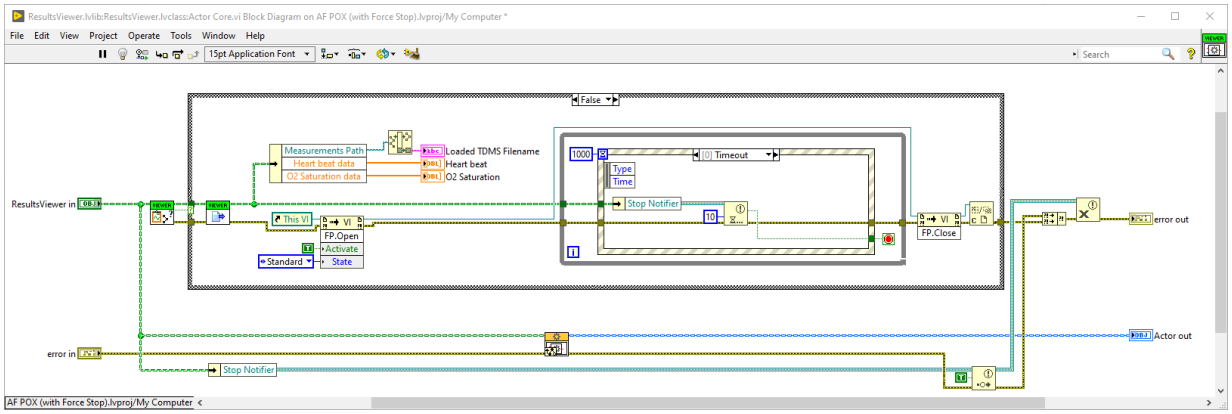
1. Support cancelation of choosing file for *Result Viewer*.
   - Go to the *Actor Core* of *Result Viewer*. Draw *Case Structure* around the WHILE and the initialization code. Leave only first method (*Choose Results File*) outside the CASE.
   - If user canceled choosing the path, clear error with 43 code (using *Clear Error* function) and send normal stop message to self enqueuer.



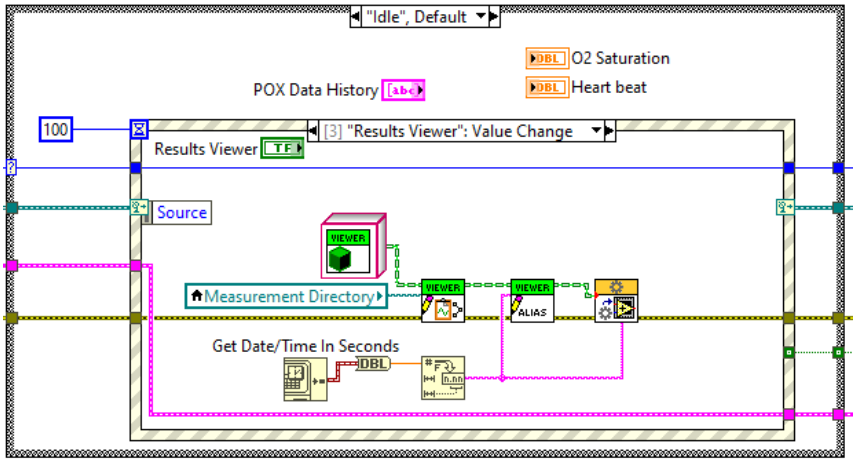
2. Implement stopping all result viewers from *Main.vi*.
   - Currently, *Result Viewer* actor supports closing itself only by closing the window (supported in the GUI loop of *Actor Core*). In order to support closing this actor from outside (so by receiving stop message), the notifier must be used so the GUI thread closes after the stop message is received.
   - Override *Pre Launch Init* method and initialize notifier there (*Obtain Notifier* function). Add necessary field to the class so it keeps the reference to the notifier.
   - Each instance of *Result Viewer* has to create its own notifier – in order to guarantee this, each notifier has to have unique name. Add another field to the class private data cluster: string for *Alias*. Create accessors for this field.
   - Use *Alias* string to set name to the notifier created in *Pre Launch Init* method.



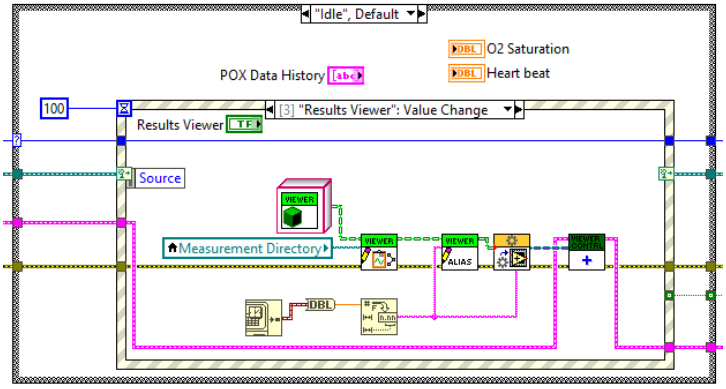
- Go back to the *Actor Core* method. Implement sending the notification right after parent *Actor Core* method stopped (so the stop message was received). Inside the GUI thread, implement waiting on notification and stopping the thread if it arrived (use event case for *Timeout*). Release the notifier after both threads are stopped.
- Using *Invoke Node* and *This VI* reference, execute *Front Panel.Close* method right after GUI thread is stopped.
- Move the code responsible for sending stop message to the event case for *Panel Close* (currently this code is executed after the WHILE).

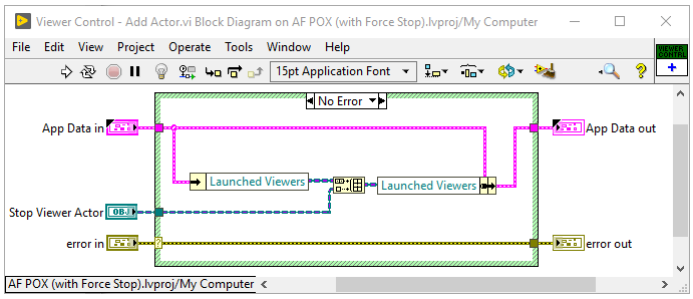


- Go to the *Main.vi* and the event case for *Result Viewer* button. Before launching the actor, use accessor in order to set unique *Alias* string to the class (algorithm for generating unique string is not imposed – you can use f. ex. timestamp which will be different for each new window opened).

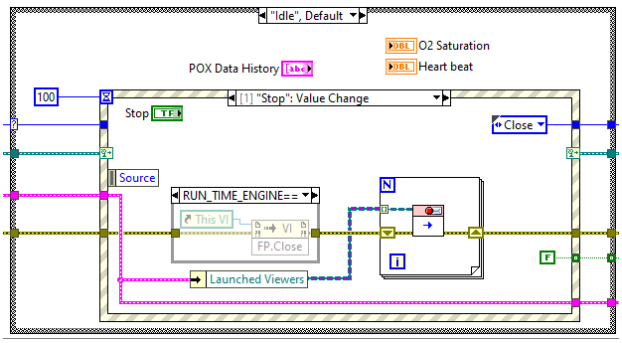


- Open typedef for cluster of application data (*#cluster_ApplicationData.ctl*). Add new element to it: 1D array of actor enqueuers (*Launch Root Actor* function returns enqueuer, so you can use this terminal to create object of appropriate type). Name the array *Launched Viewers*.
- Find the tunnel for *Application Data* cluster located on a border of main loop. Change it to shift register and wire it through all the cases.
- Go back to the event case for *Result Viewer* button. After the actor is launched, add newly created enqueuer to the array of *Launched Viewers*. You can create additional VI that will perform this operation.



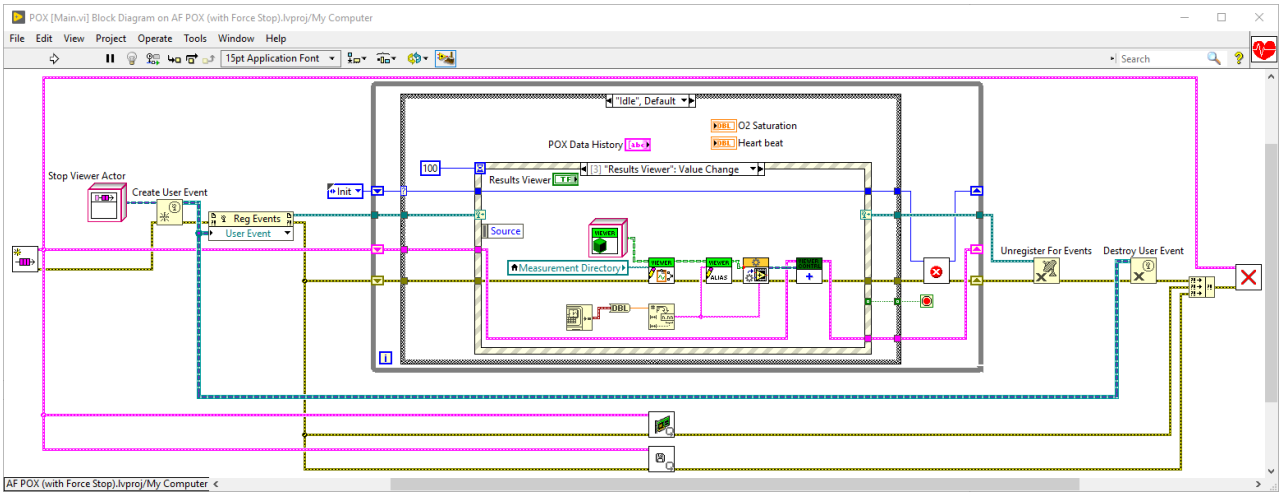


- Implement sending stop messages to all *Launched Viewers*. You can do it inside the event case for *Stop* button.

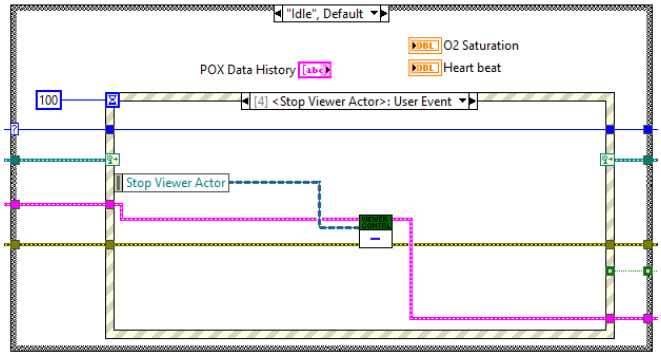


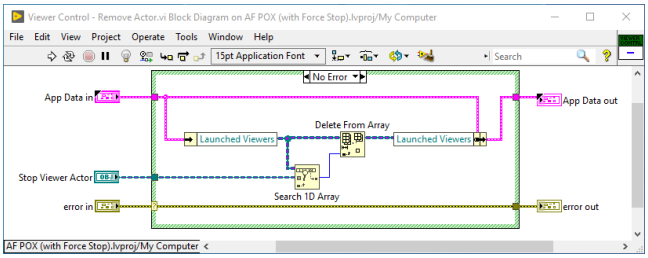
- Run the *Main.vi* and open multiple result viewer windows. Click *Stop* button on main window and test whether all viewers were closed. Fix all detected issues.

3. Handle closing the single actor from *Result Viewer* window.
   - Run *Main.vi* and open multiple result viewer windows. Close one of the viewers by closing its window. After that, stop the main application window. This should result with an error – one of the actors was already closed, but the main window tried to send to it stop message anyway (*Launched Viewers* array had invalid reference to the already closed actor).
   - In order to fix this error, *Result Viewer* has to notify the main application window that it closes. This way, main loop can remove invalid reference from the array.
   - Go to the *Main.vi*. Use user event for supporting described scenario. As a user event data type, set actor enqueuer object (name the constant connected to *Create User Event* function). Register this event in event structure of *Idle* state. After the main thread is stopped, unregister and destroy user event.

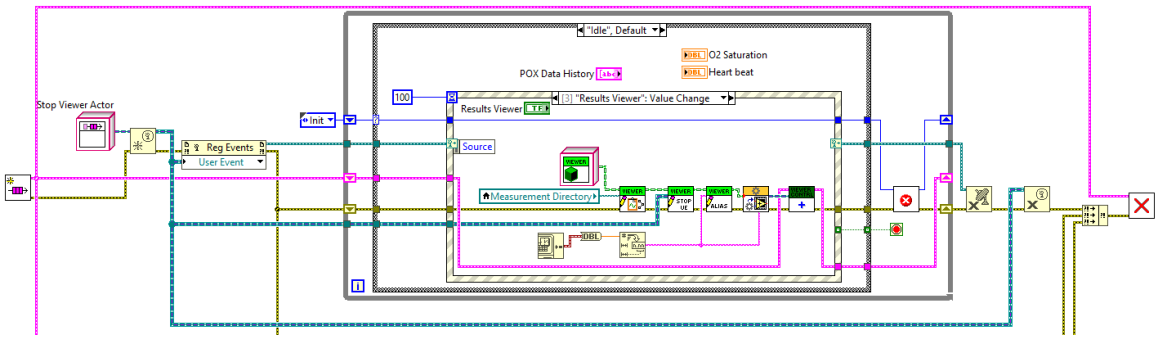


- Add event case for this user event. Inside this case, search *Launched Viewers* array in order to find the actor that has been closed (enqueuer of closed actor will arrive within this event). After finding the index of the actor, remove it from the array.

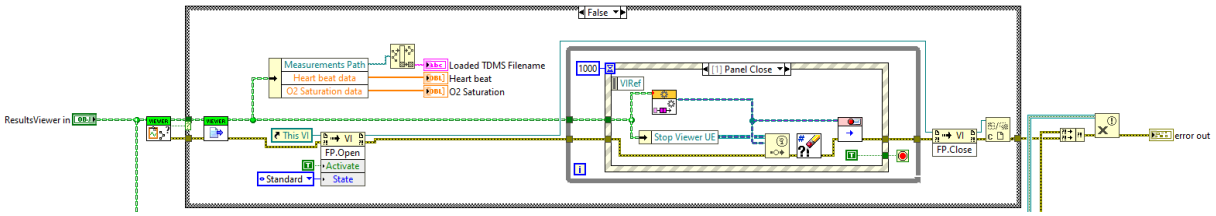




- Go to the *Result Viewer* actor. Add new field to its private data cluster: reference to the user event created in a previous step. Create accessors for this item.
- Go back to the *Main.vi* and event case for *Result Viewer* button. Before launching the actor, use appropriate accessor in order to set user event reference.



- Open *Actor Core* of *Result Viewer*. Inside the event case for *Panel Close*, generate user event and send self enqueuer within it.



- Run *Main.vi* and test the latest modification. Fix all detected issues.

END OF EXERCISE

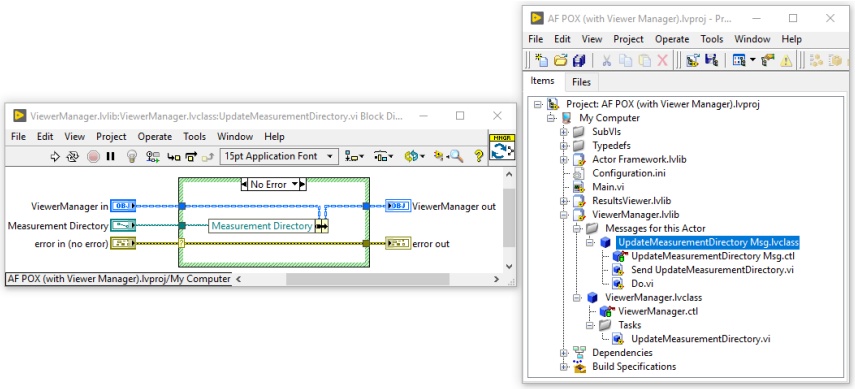
PULSE OXIMETER AF 3 Viewer Manager

GOAL

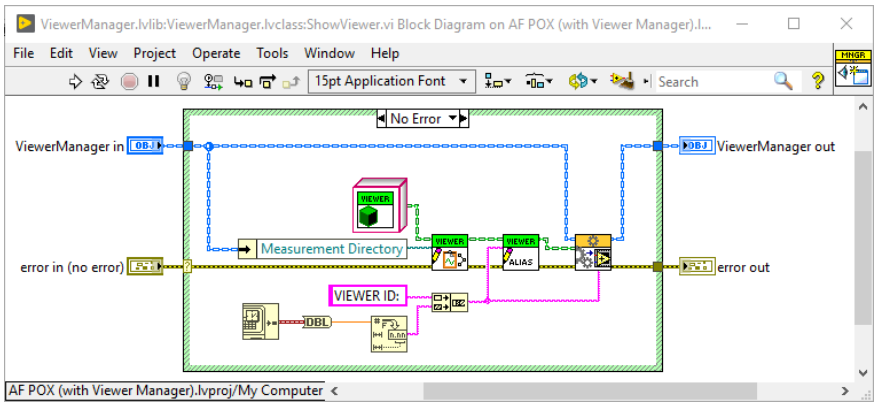
Modify the architecture of the application so that the main thread does not have to keep and update references to *Result Viewer* actors. Create one root actor named *Viewer Manager*. This actor should launch *Result Viewer* windows as nested actors. This way, closing *Viewer Manager* actor will also close all the viewers.

IMPLEMENTATION

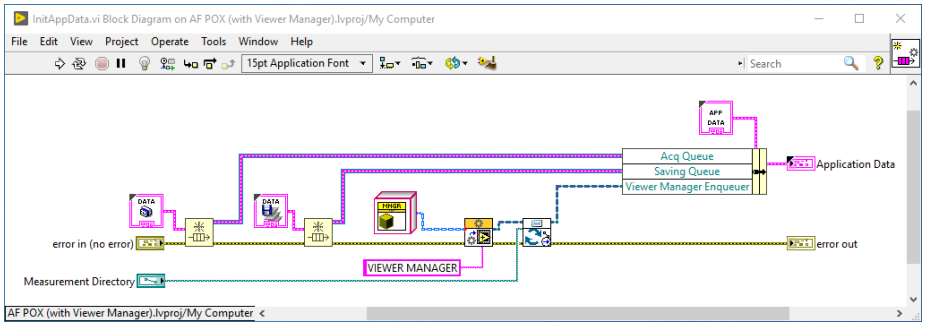
1. Familiarize yourself with the exercise starting point.
   - Go to ...*Exercises\Project 2 – AF POX\AF POX 3* and copy the project to your working directory.
   - Analyze current application status. Functionality related to user event and sending stop messages to all opened viewers was deleted. The project is ready for new implementation of this feature.
2. Create *Viewer Manager*.
   - Create new actor and name it *Viewer Manager*.
   - Add *Measurement Directory* field to the private data cluster.
   - Add static method *Update Measurement Directory*. This method should have control for measurement directory and should update class field using this control. Remember to update connector pane of this method so there is input for measurement directory control. Create message for this method.



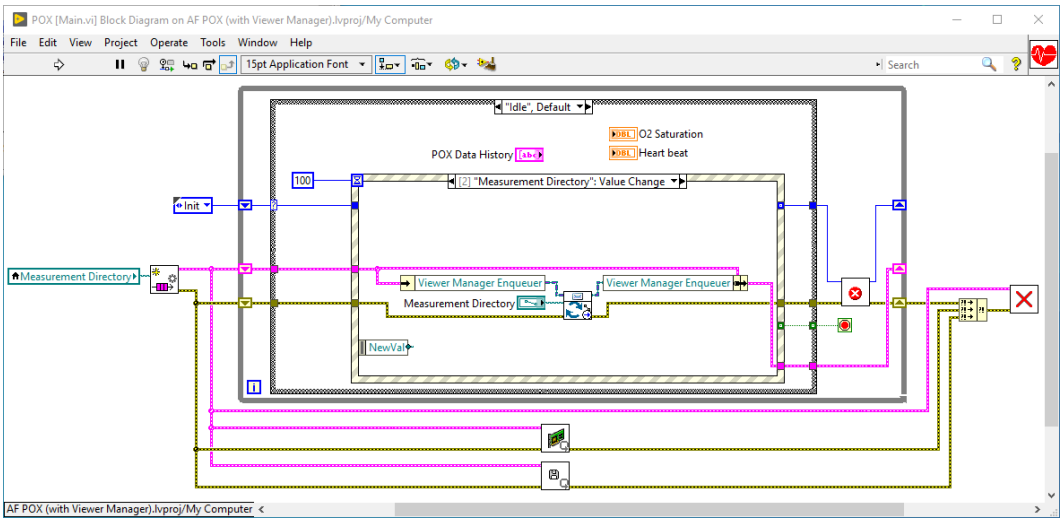
   - Add another static method *Show Viewer*. Inside this method, implement launching new *Result Viewer* as nested actor. Remember about setting the alias and measurement directory. Create message for this method.



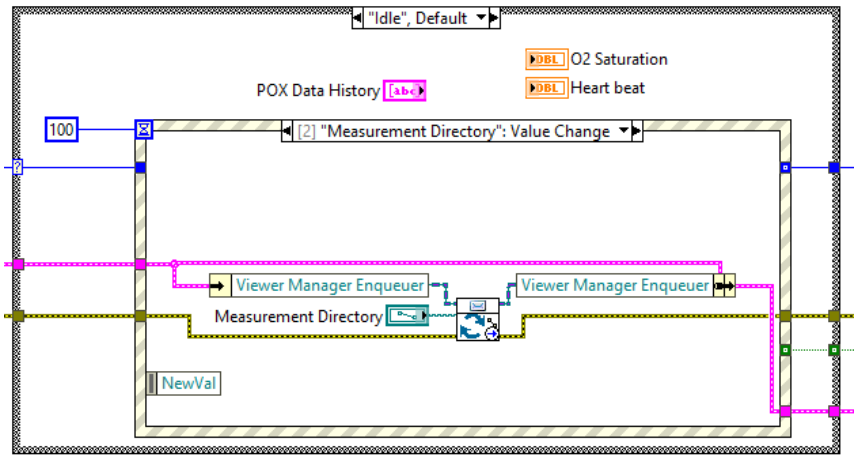
3. Modify *Main.vi*.
   - Open *Application Data* cluster (*#cluster_ApplicationData.ctl*). Add new item to it: actor enqueuer for *Viewer Manager* actor.
   - Go to *Main.vi* and open *InitAppData* function. Implement launching *Viewer Manager* actor as a root. Right after launching the actor, send message to it for updating measurement directory. Create control for the path input. Update connector pane of *InitAppData* function by adding tunnel for measurement directory path.



   - Go back to *Main.vi*. Create local variable for *Measurement Directory* control. Use it to configure *InitAppData.vi*.
   - Go to the event structure inside the *Idle* case. Add event case for *Measurement Directory: Value Change*. Inside this case, send message for *Viewer Manager* actor so it updates the directory.



- Go to the event case for *Result Viewer* button. Replace code inside this case for sending appropriate message to the *Viewer Manager*.



4. Test the application.
   - Run Main.vi and test the application. Check various combinations of launching and closing Result Viewer windows. Fix all detected issues.
   - Compare this step of the project with previous one (without Viewer Manager). Which approach is better? Which architecture generated less work and is more resistible for the bugs and errors?

END OF EXERCISE