**Rapid-I Vision Measuring System Project**

**Rapid Wrapper – MainDll.cpp file**

**Class Structure:**

**Class Name: AddDimAction**

**Member Variables**:

* **dim**: A pointer to the DimBase object, representing a dimension to be added or manipulated.
* **PPStatusAction**: A boolean flag indicating whether the action has a particular status (used for toggling).
* **CurrentActionType**: This variable stores the current action type, which is set to ADDDIMACTION in the constructor.
* **ActionStatus**: A method that determines whether an action is currently active or not (not explicitly defined in the provided code but used as a state handler).

**Constructor and Destructor:**

* **AddDimAction()**: Initializes the action by setting the action type to ADDDIMACTION and setting the PPStatusAction to false.
* **~AddDimAction()**: Destructor that cleans up by setting the CurrentDimAddAction pointer in dim to NULL, indicating that no action is associated with that dimension anymore.
* **Class Declaration:**
  + AddImageAction is a subclass of RAction (presumably a base class for actions in this system).
  + It defines specific fields like ImageCount, PausableAction, and baseFG.
  + **Constructor (AddImageAction):** Initializes the action type (CurrentActionType) to IMAGE\_ACTION and sets PausableAction to false. It also initializes ImageCount to zero.
  + **Destructor (~AddImageAction):** Deletes the baseFG object (if allocated), which represents a frame grabber or image capture source, and handles any potential errors.
* **Methods:**
  + **execute:** Sets the action's status to true (indicating that the action is in progress or has been executed). Returns true upon successful execution.
  + **redo:** Increments the ImageAction\_Count (likely a global counter maintained in MAINDllOBJECT to track the number of image actions) and sets the action's status to true. It signifies the "redo" behavior of the action.
  + **undo:** Decrements the ImageAction\_Count and sets the action's status to false, indicating that the action has been undone.
  + **AddImage\_DroValue:** Creates a new AddImageAction and sets its action type and frame grabber (baseFG). It then adds the new action to the MAINDllOBJECT action stack.
  + **getFramegrab:** Returns the baseFG, which is an object representing the image capture mechanism or device.
* **Operator Overloading:**
  + **operator<< (Output Stream):** Serializes the AddImageAction instance into a human-readable format, including details like CurrentActionType, ImageCount, and properties of FramegrabBase (presumably representing the image object). The operator is designed to print information about the action.
  + **operator>> (Input Stream):** Deserializes an AddImageAction from a stream, interpreting each tag in the input and populating the class's properties accordingly. This is typically used for loading the action's state from a saved file or data stream.
* **Error Handling:**
  + The class contains try-catch blocks to handle exceptions and ensure that errors in the action handling (such as during memory allocation or stream reading) are caught and logged using the MAINDllOBJECT->SetAndRaiseErrorMessage method. This error handling mechanism prevents the application from crashing and provides meaningful error messages.
* **Class Declaration:**
  + **Inherits from RAction:** The class inherits from RAction, which seems to be a generic base class for various action types.
  + **Attributes:**
    - **CurrentActionType**: Identifies the type of action. For AddPointAction, this is set to RapidEnums::ACTIONTYPE::ADDPOINTACTION.
    - **CriticalAction**: A flag indicating whether the action is critical. It’s initialized to false.
    - **PausableAction**: A flag indicating whether the action can be paused. It’s initialized to false.
    - **AddToShape**: A flag to indicate whether the action involves adding the point to a shape or not. It’s initialized to true.
    - **shape**: A pointer to the ShapeWithList object representing the shape to which points are added.
    - **baseFG**: A pointer to the FramegrabBase object, which likely represents a frame grabber or an image capturing device.
* **Constructors and Destructor:**
  + **Constructor (Default)**: Initializes an AddPointAction object with default values (critical, pausable, and add-to-shape flags).
  + **Constructor (Parameterized)**: Takes a ShapeWithList object and a FramegrabBase object, initializing the object with these values and setting other flags.
  + **Destructor**: Deletes the baseFG object, ensuring proper memory management and handling exceptions gracefully.
* **Methods:**
  + **execute()**: Executes the action by adding points to the shape or deleting points if AddToShape is false. Sets the action’s status to true if successful.
  + **redo()**: Redoes the action by pushing the action onto the PointAtionList and modifying the points in the shape. Sets the action’s status to true.
  + **undo()**: Undoes the action by removing it from the PointAtionList and modifying the points to remove the added point.
  + **pointAction()**: This function adds a point action to a shape. It creates a new AddPointAction object and configures it based on the passed parameters, adding it to the shape’s PointAtionList. It also handles specific conditions like enabling digital zoom or two-step homing.
  + **setParams()**: Sets the parameters for the shape and baseFG objects.
  + **getFramegrab()**: Returns the baseFG (framegrabber object).
  + **getShape()**: Returns the shape object (the shape to which points are added).
  + **Serialization (Stream Operators)**:
    - **operator<< (Output Stream)**: Serializes the AddPointAction object into a stream, including its attributes and nested object values like the FramegrabBase.
    - **operator>> (Input Stream)**: Deserializes the AddPointAction object from a stream, extracting the object’s state from the input data and reconstructing its properties.
* **Error Handling:**
  + The class uses try-catch blocks throughout its methods to catch exceptions and log error messages using MAINDllOBJECT->SetAndRaiseErrorMessage. This ensures that any errors during action execution, redo, undo, or serialization are handled gracefully, and relevant error codes are logged.
* **Old Format Handling (ReadOldPP):**
  + There’s a method (ReadOldPP) to handle deserialization of older data formats (possibly for backward compatibility with legacy systems or data files). This method reads legacy data from a stream and maps it to the current AddPointAction structure.

**Main Functionalities**

**setDim(DimBase\* dim):**

* **Purpose**: Sets the dim pointer to the given DimBase object. This allows the action to reference and manipulate a specific dimension.
* **How it works**: Assigns the passed DimBase pointer to the dim member variable.

**execute():**

* **Purpose**: Executes the action to add a dimension (if valid).
* **How it works**:
  + The function starts by checking if the action is valid via ActionStatus().
  + Depending on the measurement type of the dim object, the function manipulates points and vectors. For different types of measurements (e.g., distance between points, distance to a cone, etc.), it updates the dimension properties (ParentPoint1, ParentPoint2).
  + The method toggles PPStatusAction to indicate if the action was completed or is in progress.
  + It updates the main dimension list (MAINDllOBJECT->getDimList().addItem()), and if the dimension is selected, it adds it to the selected dimension list (MAINDllOBJECT->getSelectedDimList()).

**redo():**

* **Purpose**: Reapplies the action (typically after an undo).
* **How it works**:
  + Similar to execute(), it checks for action validity and processes the dimension properties based on measurement type.
  + If the dimension involves GD&T (Geometric Dimensioning and Tolerancing), it updates reference datums.
  + The dimension is added back to the dimension list and selected dimensions list.
  + For DIM\_POINTCOORDINATE types, child dimensions are also added recursively to the list.
  + The method then sets the action status to true (indicating the action was successfully reapplied).

**undo():**

* **Purpose**: Reverts the action (i.e., removes the added dimension).
* **How it works**:
  + If the dimension is a GD&T measurement, it decreases the reference datum values for the parent shapes.
  + The dimension is removed from the dimension list (MAINDllOBJECT->getDimList().removeItem()), and if it's a DIM\_POINTCOORDINATE, its child measurements are also removed.
  + Finally, the action status is set to false, indicating that the action was undone.

**addDim(DimBase\* dim, bool DisableAction):**

* **Purpose**: Adds a dimension to the action queue and disables/enables the action as needed.
* **How it works**:
  + Creates a new AddDimAction object.
  + Sets the dimension for this action.
  + If DisableAction is true, it toggles the action status.
  + Adds the action to the main action queue (MAINDllOBJECT->addAction()), and ensures that the dimension is not selected initially (MAINDllOBJECT->selectMeasurement(dim->getId(), false)).

**getDim():**

* **Purpose**: Returns the DimBase object associated with the current action.
* **How it works**: Simply returns the dim pointer.

**3. Serialization (Input/Output)**

The class also includes operator overloads for **streaming input and output**:

**operator<< (output stream):**

* **Purpose**: Allows the AddDimAction object to be output to a stream (e.g., to a file or console).
* **How it works**:
  + It outputs various details about the current action, such as the action type, ID, original name, and dimension ID.
  + Uses getId() and getOriginalName() methods (presumably from the parent class or DimBase).

**operator>> (input stream):**

* **Purpose**: Allows the AddDimAction object to be read from a stream (e.g., for loading previously saved states).
* **How it works**:
  + Reads from the stream to extract the action’s state, including the dimension ID, action name, and other attributes.
  + It checks whether the format is old (MAINDllOBJECT->IsOldPPFormat()), and if so, it uses a custom method ReadOldPP() to handle the old format.
* **Action Handling:**
  + The core functionality of the AddImageAction class revolves around managing the state of an image action, which could include operations like adding an image, counting image-related actions, and allowing redo/undo capabilities.
* **State Persistence and Restoration:**
  + The operator>> and operator<< allow the class to serialize and deserialize its state. This is essential for saving the current state of the application and restoring it at a later time (e.g., after a system restart or for undo/redo operations).
* **Frame Grabber Management:**
  + The baseFG field is of type FramegrabBase, which represents an image capture device or source. This field is managed throughout the lifecycle of the AddImageAction and is included in the serialized data for persistent storage.
* **Counting and Managing Image Actions:**
  + ImageAction\_Count is used to track the total number of image-related actions that have occurred in the system. The execute, redo, and undo methods manipulate this count to reflect the current number of image actions in the system.
* **Action History and Undo/Redo:**
  + The AddImageAction class is part of a larger action management system. It supports undo and redo functionality, allowing the user to reverse or reapply image-related actions as necessary. This is typical of software with complex workflows, such as image processing tools or graphic editors.
* **Pausable Action:**
  + The PausableAction flag allows the action to be paused, though it is not currently being used in this code. This could be useful for scenarios where image actions may need to be temporarily halted or resumed, depending on the context of the application.

1. **Action Handling:**
   * **Add Points to Shape:** The main function of the AddPointAction is to manage the addition of points to a shape. When the action is executed, it adds the points from baseFG (frame grabber) to the shape if AddToShape is true. If AddToShape is false, the points are deleted instead.
2. **Undo/Redo Functionality:**
   * The class supports undo and redo operations. Redo adds the points back to the shape, while undo removes them. This allows the action to be reversed or reapplied as needed, supporting complex workflows in applications that require state manipulation.
3. **Serialization and Persistence:**
   * The AddPointAction class can be serialized to a stream for persistence, allowing it to be saved and loaded. This is handled by the operator<< and operator>>, making it easy to store action history or state and reload it when necessary (for undo/redo or recovery purposes).
4. **Memory Management:**
   * The class dynamically allocates the baseFG object and ensures its proper deletion in the destructor. This prevents memory leaks by handling the cleanup of allocated resources.
5. **Shape Manipulation:**
   * The class is designed to interact with a ShapeWithList object, which is likely a shape or geometric object that stores a collection of points. The points are manipulated through methods like ManagePoint and Modify\_Points, with different flags (ADD\_POINT, REMOVE\_POINT) determining whether points are added or removed.
6. **Integration with Other Features:**
   * The class is integrated with other system components, such as handling digital zoom (DigitalZoomIsOn) and two-step homing (ActionForTwoStepHoming), enabling additional functionality depending on the system's state or settings.

**Workflow**

1. **Action Creation**:
   * The user initiates an action (e.g., adding a dimension) by calling addDim().
   * A new AddDimAction object is created, and the specified DimBase object is linked to it.
2. **Executing the Action**:
   * The execute() function is called, which sets up the necessary dimension properties based on the measurement type.
   * The dimension is added to the dimension list and selected dimensions list if applicable.
3. **Redoing the Action**:
   * The redo() function is called if the action is undone and needs to be reapplied. This function restores the dimension and related properties.
4. **Undoing the Action**:
   * The undo() function is called if the action needs to be reverted. The dimension is removed from the dimension list, and any associated child measurements are also removed if applicable.
5. **Serialization**:
   * The operator<< and operator>> provide ways to save the state of the AddDimAction object and load it back from a file or stream.
   * This is important for saving and restoring actions within a session or when working with persistent data.
6. **Creating and Adding an Image Action:**
   * The AddImage\_DroValue method is responsible for creating a new AddImageAction and setting its properties, such as the baseFG (image frame grabber). The action is then added to the action stack using MAINDllOBJECT->addAction(action). This signifies that an image has been added or an image-related action has occurred.
7. **Executing an Image Action:**
   * When the action is executed (via execute), the action's status is set to true. This method might be triggered when the user initiates an image action, such as capturing or manipulating an image.
8. **Redoing an Image Action:**
   * If the user wants to redo an image action that was previously undone, the redo method is called. It increments the ImageAction\_Count and sets the action status to true to reflect that the action has been reapplied.
9. **Undoing an Image Action:**
   * The undo method is called to reverse an image-related action. It decrements the ImageAction\_Count and sets the action status to false. This could be triggered by a user request to undo the last image manipulation or action.
10. **Serializing and Deserializing the Action:**
    * The action can be serialized to a stream (e.g., for saving or logging) via the operator<<. Similarly, the action's state can be deserialized from a stream (e.g., loading from a saved file or history) via the operator>>.
11. **Error Handling:**
    * Throughout the lifecycle of the action (from creation to execution and serialization), error handling ensures that any issues that arise during memory allocation, stream reading/writing, or other unexpected conditions are caught and logged, preventing crashes and providing error messages to the user or system logs.
12. **Creating and Adding a Point:**
    * **pointAction()** is the entry point for adding a point to a shape. This method creates a new AddPointAction instance, sets its parameters (shape and frame grabber), and adds it to the shape’s PointAtionList.
    * Depending on system settings like digital zoom or two-step homing, it adjusts the behavior of the action (e.g., enabling digital zoom).
    * The action is then added to the MAINDllOBJECT's action stack for further management (such as undo/redo).
13. **Executing the Action:**
    * When the action is executed via the **execute()** method, it checks whether the shape is valid and whether there are points available in the baseFG. If so, it either adds the points to the shape or deletes them, depending on the AddToShape flag.
    * The action status is set to true if the execution is successful.
14. **Redoing and Undoing the Action:**
    * When the action is redone (via **redo()**), the points are added back to the shape, and the action is pushed to the PointAtionList.
    * When undone (via **undo()**), the points are removed from the shape, and the action is removed from the PointAtionList.
15. **Serialization/Deserialization:**
    * **operator<<** and **operator>>** allow the AddPointAction object to be serialized and deserialized from streams, facilitating saving and loading the action history or state. The ReadOldPP method handles backward compatibility with older formats.
16. **Error Handling and Logging:**
    * Throughout the process, the class uses try-catch blocks to ensure that any errors during execution, serialization, or deserialization are caught and logged, preventing crashes and providing informative error messages for debugging.