# Productizing of the Human Tracking functionality

# Motivation:

A framework was conceived and implemented to productize the face/body detection algorithm implementation with the following specialties. This serves as the first draft

1. The product should be OS independent/agnostic
2. The productization should use modular blocks so that it guarantees full extensibility, by which it means that the components that that were developed to productize this algorithm implementation should be usable for a similar implementation of a different algorithm, in a much broader sense to Strategy pattern and at the architecture level.
3. The product should be available in more than one platform, such as Windows and Linux.
4. Implementation was done using WINFORMS UI (C#) and QT in Windows while using QT in Linux (UBUNTU).
5. Product architecture was developed/evolved by me after studying the best of all the CAD/CAM products without the feature being available at the time of architecture development.

The above expectations make the product abstract in definition in developing, testing, evaluating and packaging the product.

The need for a modularized, customizable, extensible, re-usable framework that would integrate the UI, the algorithm implementation and the platform specifics, while keeping the framework cross-platform was successfully attempted.

Architecture (parameterized commands, events and notifiers), Platform (Windows, Linux), Technology (OpenVINO for face recognition) and Feature (Skills that are to be executed) are all kept modular as much as possible so that these corners could be independent in development.

**The architecture is a loosely coupled, highly cohesive, adhering to SOLID/DRY design principles, fully extensible, less-constrained, type-safe, cross-platform, modular (componentizable/easily to be integrated)** and **parameterizable** design implemented evolving from a POC to a production ready version (first draft) **even without the feature code or “meat” that executes the skills.**

**The first draft of the architecture design concentrates on Commands and event observers and notifiers as they form the heart of the requirement of a typical engineering software solution.**

The basic idea to implement the framework is gained from the similar experience of mine working with the architecture of various CAD/CAM products such as Delmia-V5, AutoCAD, Visual Die Maker, Siemens NX Part Modeling, TAAM(Avionics)

**COMMANDS ( User directive from UI to the core executions)**

1. Each command is implemented as a templated Interface that needs to know the input and output for the command aka Skill to execute
2. A bridge pattern is used between the templated interface and its Curiously Recurring Template Pattern type implementation. This gives the strength to use Interface object without knowing the underlying implementation while also not necessitating the actual implementor agree to the Interface’s contracts.

The command is parametrized with the input data, output data and the implementor which actually executes the command

template<typename IData, typename IData, typename SkillExecutor>

static ISkill< InputDataToSkillExecuteType, OutputDataFromSkillExecuteType>\*

CreateSkill(SkillExecutor& executor )

{

auto skill = GenericSkillImplWrapper<InputDataToSkillExecuteType, OutputDataFromSkillExecuteType, SkillExecutor>::GenericSkillImplementer::GetInstance(executor);

return skill;

}

where GenericSkillImplWrapper is a specific variety of a command which does not do Undo/Redo. Similar commands could be created for specialized commands which do Undo/Redo, Serialization etc.,

template <typename InputDataToSkillExecuteType, typename OutputDataFromSkillExecuteType, typename ExecutorType>

class GenericSkillImplWrapper

{

}

In the above ExecutorType is any class that the feature developer implements which should have a method bool RunExecute() implemented.

1. The commands manager is capable of storing all the commands (currently it has only one) and “Execute” call from the manager, calls the factory method to create a single instance of a specific command using scott meyer’s singleton pattern.
2. The adapter layer for C# to C++, namely C++/CLI layer and for that matter all the class are implemented using private implementation pattern to shield off from the CPP variables which either are not needed by the UI or which creates dependency problems for the UI, which is unnecessary. Classname\_Impl is the name used for such classes.
3. The ::Initializer() method of the C++/CLI adapter layer is called from C# by the call to m\_AbstractSkill = new MngdFaceDetectionCommand(); in C#.

Fully extensible: The user can give further down new implementations to the command concrete types to implement proprietary executions.

Less-Constrained: Unlike the traditional command patterns, which require the implementation to oblige to the contracts of the Command Receiver by implementing the base class functions, this new design just needs the input data type, output data type, the class that just has a method that serves as the execute call. The strength of this design is that the programmer can design his own class hierarchy that does not need him/her to even know about the architecture of the command.

Type-safe (at the compile time OR compiler error): A generic command just says that it can execute a set of calls but at the time of the implementation the command gets concrete. The command has an input data, returns output data and at times, continuously streams output data(think of simulations). This requirement either gives rise to RTTI in the design that needs (Dynamic\_cast) (breaking of the laws of OOP, which makes the caller know what is inside the command, which necessitates that the caller, which would otherwise do not know the implementation (independent of implementation, became independent of the concrete implementation)) OR reinterpret\_cast or direct\_cast from void\*, which is dangerous for a potential crash.

Cross-platform : The entire architecture is written in C++11/14 cross-platform C++ and can be used across all the platforms.

Parameterizable: This is mainly in the perspective of the type of the command that needs to be used. A command could be

1. Redo/Undo-able
2. Non redo/undo-able
3. Serializable/Non-serializable OR
4. Implement its own policies.

All of these commands can now be extended from the very generic base class.

**EVENT OBSERVATIONS AND NOTIFICATIONS (Return of data from core to the platform specific UI)**

The design is also provided with a parameterized event observation and notification mechanism which would continuously return/steam the output data to the platform specific UI. Please also note that this framework can be used for any event observation.

1. Instantiation of the command’s executor is made when the initialization of the client is made. This registers the call back function for a specific event.

void FaceDetectionCommandExecutor::RegisterHostUIUpdateEventHandler()

{

// Initialize all the objects for event handlers

AdaptiveHumanTracker::Event eventForSubscription =

AdaptiveHumanTracker::Event::FACE\_DETECTION;

AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::Register<FaceDetectionCommandExecutor>(eventForSubscription, \*this);

}

A method to unregister is also provided

void FaceDetectionCommandExecutor::UnRegisterHostUIUpdateEventHandler()

{

// Un-Initialize all the objects for event handlers

AdaptiveHumanTracker::Event eventForSubscription =

AdaptiveHumanTracker::Event::FACE\_DETECTION;

AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::UnRegister<FaceDetectionCommandExecutor>(

eventForSubscription, \*this);

}

When the **event** is observed, the event notifier will call the **OnEventNotified**(data) of the above **this** object, which would call the set of functions till the UI.

The C++/CLI adapter object instantiates the Command Executor. This registers the

# Introduction to the new framework to support execute multiple skills

# Overall structure of the components

QT (UI)

(Windows/Linux)

OpenVINO code to do face detection.

Skill Executors

**Cross-platform** Framework implemented in C++14/17

Skills Manager

WinForms

(C#-UI)

Windows

Event handler

Diagram-1: The unfilled arrow blocks facing right side are all direct calls from left to right.

The grey-filled arrow blocks from right to left are all Function, Lambda or pointer to function call backs using Event Handling Mechanism of the framework.

## A Brief overview of the components

A Few nomenclature definitions:

1. **Host**: It is the invoker of the entire process. This could be the User Interface specific to a platform like WINFORMS, QT, MFC etc. or a console application requiring a command to be executed with command line arguments. The invoker is a user.
2. **Client**: In the context of the framework, the client is some component/object that registers to the framework and caters to the framework’s expectation to execute a task.
3. **Skill Executors**: It is a type of a **client** that caters to the execution of the Skills. These are included in the **CommandExecutors** project. A compile-time contract that an executor should abide by is to have a method **RunExecute().** The skill executors are just the same as that of Command Receivers in the typical Command Pattern design.
4. **Event Observers:** It is a type of a **client** that an appropriate action when a specific event is notified and observed. The files are included in the project titled **EventObserversNotifiers.** Any client that need to act as an observer for a specific event, should implement OnEventNotified() method. The event observers are the same as that of the Observer design pattern.
5. **Framework:** This is the entire set of combination of creational, structural and behavioral design patterns that holds the host, the algorithm implementation and the platform specific dependencies together making it possible for the various users, such as UI developers, algorithm implementers, and framework feature developers to work independently to evolve the product.

Also the framework does not constrain the clients to be of a specific hierarchy or to implement any interface. A client can act as a client under various heads such as the same client could well be an executor or be an event observer or both.

The cross-platform framework, which is implemented with advanced C++14/17 standards provide the following utilities and also acts as the indirection between the platform specific Host UI applications with the executor of the skills, thereby abstracting the entire skill execution.

The cross platform framework contains the major features such as

1. Command Manager Framework
   1. This is implemented with a further down low level design to have
      1. Interface implementation mechanism with the a templated abstract ISkill interface
      2. Curiously recurring template pattern
      3. Bridge
      4. Factory to create each ISkill command
   2. The ISkill currently supports Execute() and Terminate() methods to execute and abort the running of the functionality respectively.
   3. In order to have a future proofing, the same ISKill interface is capable of supporting the following
      1. Undo()
      2. Redo()
      3. ReExecute()
      4. Journalize()
   4. The ISKill(Command) is capable of getting exposed through any other scripting language such as Python, Ruby, VB etc.
   5. The current implementation of the ISkill(command) assumes that the command takes an input data, runs the skill, modified some other data or creates some new data or both and then return the data.
   6. Additional feature of the ISkill (Command) is to integrate the **Event Handler** to continually update the Host (UI) on the running status/output (in this case, the image or the text)
2. Event Handler Framework:
   1. Interface implementation mechanism with the a templated abstract IObserver interface exposing UpdateAction() method.
   2. The framework is the classic observer pattern in action while also implementing
      1. Curiously Recurring Template Pattern
      2. Observer allowing clients, who listens/follows the events to take action and also allows them to unsubscribe from the events.
   3. It should be kept in mind that the **Event Handler** is solely responsible to reverse communicate across language boundaries, across technologies to communicate the Host UI from the core C++ cross-platform source.

## Procedure to set-up skill to be run from framework

## RunExecute() and Terminate() obligation:

* Create any stand-alone class, name it with a suffix <Executor>.cpp/.h and keep it in CommandExecutors project. In Visual Studio add the header and CPP file while for Linux, modify the CMAKE file to include this file while building.
* The file **FaceIdentificationCommandExecutor.h/.CPP**  shall be a good example to start with
* The framework is less-constraining on the programmers and so any stand-alone class that implements the Skill will do. Implementing the skill means, the executor class should implement RunExecute() and Terminate() method among others. The feature developer does not need to inherit from any specific interface. He/she shall use such a mechanism for their own benefit but the frame work does not constrain.
* The only obligation is that the stand-alone class has to define member methods namely **RunExecute(const DataType&) and Terminate()**

For example **FaceIdentificationCommandExecutor.cpp** implements a method **RunExecute()** as defined below

std::unique\_ptr<NullData> FaceIdentificationCommandExecutor::RunExecute(const FaceIdentificationCommandExecutorInputData& inputData)

### Points to be noted:

* RunExecute() method returns the unique\_pointer to the output data from the Skill/Command Execute. If the command does not have to return any data, it should return **NullData,** which is a framework default equivalent to no data.
* The RunExecute() method takes **const InputDataType&** as the signature.
* It is a good programming practice to implement **Private Implementation** to implement RunExecute. The class **FaceIdentificationCommandExecutor** has a private implementer namely FaceIdentificationCommandExecutor\_impl. This private implementer executes the skill/command and returns bool to acknowledge if the running was a success or a failure. The private implementation also contains the output data from the execution stored, which shall be returned when needed.
* This class **FaceIdentificationCommandExecutor** becomes the client to call the set of methods to implement the skill.

### A few words regarding Terminate() method

* Terminate() call that the **ISkill<InputData,OutputData>** interface declares is implemented by **GenericSkill** Bridge class. (Please refer to the class diagram below).
* In order to complete the implementation, the Skill executor, (in this case **FaceIdentificationCommandExecutor** has to implement a method **void** **Terminate()**
* The idea behind Terminate() call is that the Executor is the only class that is capable of running the Skill execute and so it is the only class that knows how to Terminate() the currently running execute thread/process.
* In the example **FaceIdentificationCommandExecutor**, the Terminate call has the following body

void FaceIdentificationCommandExecutor::Terminate()

{

if (m\_PrivateImplementor)

{

m\_PrivateImplementor->Terminate();

}

}

And the **private implementor** which actually runs the execute command has the following body to terminate the execute call.

void Terminate()

{

if (stopIdentification\_ == true)

return; // Already stopped execution

std::mutex this\_mutex;

this\_mutex.lock();

{

stopIdentification\_ = true;

}

this\_mutex.unlock();

{

std::unique\_lock<std::mutex> wait\_for\_variable\_lock(runExecuteCompleteMutex\_);

runExecuteCompleteCV\_.wait\_for(wait\_for\_variable\_lock, chrono::seconds(5), [this] {return faceIdentificationRunStatus\_; });

}

}

stopIdentification\_ is the state variable to break the execution of the skill in this case. This is merely set to true so that the thread which runs the skill execute is broken.

The RunExecute() method of FaceIdentificationCommandExecutor\_Impl(private implementation) has the following while loop

while (!stopIdentification\_) {

capture.read(image);

cv::Mat image\_for\_drawn;

…

…

}

By setting stopIdentification\_ to true, the while loop will be broken and the thread returns. This is how the Skills Execution is aborted if the user intends to.

## The flow of call while executing a skill

1. The invoker/host is platform specific and to execute the skill calls the framework API. Please refer to the MngdFaceIdentificationCommand.cpp/.h, which is called by the WINFORMS host. This class has a private implementation. The call is as follows

FaceIdentificationCommandExecutor\* **m\_executor** = new FaceIdentificationCommandExecutor(fptr, useOpenCVHighGUI);

bool Execute(FaceIdentificationCommandExecutorInputData& inputData)

{

std::unique\_ptr<NullData> otNullData =

**AdaptiveHumanTracker::SkillsManager<FaceIdentificationCommandExecutorInputData, NullData>::GetInstance()->Execute< FaceIdentificationCommandExecutor>(inputData, \*m\_executor);**

if (otNullData)

return true;

return false;

}

* The input to the skill to be executed is defined by the structure FaceIdentificationCommandExecutorInputData.
* Theoretically, a skill execution shall return another data. In this case, nothing is returned and so it uses NullData type.
* The yellow block above is the framework API call that executes the skill. It calls the execute method on the single instance of the SkillsManager with templated types of Input data type and Output data type, while the execute call is a templated type of a specific command executor.
* A compile time error is thrown if the executor does not have RunExecute() method implemented by the client.
* Ideally, the call by the host shall call the skill execute as follows without having to call the Execute at the SkillsManager. It is for time being implemented to keep things simple. The call sequence by the Host shall be the following

std::unique\_ptr<SkillOutputDataType> nullOutput;

// The templated interface ISKill<T1,T2> below is an abstraction of the any // type of skill that takes a specific Input and gives an output. Any host //or client shall use this without having to know what the underlying type //is

ISkill< SkillInputDataType, SkillOutputDataType>\* skill = SkillsFactory::CreateSkill<SkillInputDataType, SkillOutputDataType, SkillExecutor>(executor); // Command Factory Call

// The following Execute API of the framework calls the executor’s //RunExecute method

if (m\_Skill->Execute(inputData)) // Skill execute call returns true/false

{

// Output data has to be queried

auto outputDataFromSkill = m\_Skill->GetSkillOutputData();

return outputDataFromSkill;

}

return nullOutput;

## ISkill Input and Output Data:

It is the feature developer’s responsibility to define and use the input and output data types for the command/Skill. If multiple data is needed, it is advised to create a class or structure to contain all of them in one single data type that shall be used as InputData.

IN the face identification skill, the input data structure is defined as follows

struct FaceIdentificationCommandExecutorInputData

{

std::string modelDirectoryPath;

std::string dataDirectoryPath;

};

The skill does not return any data and so the framework provided NullData shall be used in the scope.

## Client for calling the actual skill implementation:

1. The executors which are contained by **CommandExecutors** project act as the clients that call the actual Skill execution methods
2. These command executors are the legendary **Command Receivers**, by definition of the classic **Command Pattern**.
3. Further, it is advised to use **Private Implementation design paradigm** to implement the executors. Please have a look at the **CommandExecutors\FaceIdentificationCommandExecutor.cpp** and its corresponding header to understand the design. FaceIdentificationCommandExecutor\_impl is the private implementer for **FaceIdentificationCommandExecutor in CPP file.**

## Construction of the Executors:

1. The skill executors are also the observers to the events registered. When the command/Skill is executed, when the skill is still running, such as from a live video output, the image detected and recognized need to be posted to the Host UI where it should be displayed. In order to achieve this, the executors are also observers for various events.
2. The constructors of executors have three main inputs that need to be initialized. They are
   1. bool m\_OpenCVHighGUIDisplay;
      * This variable if set to true, will display the output with cv:ImageShow APIs and it does not need any Host UI framework. By default it is true. If you construct the executor from Host UI such as QT, this variable should be set to false explicitly.
   2. **std::function<void(cv::Mat\*)> m\_UIUpdateHandlerSTDFunction**;
      * m\_UIUpdateHandlerSTDFunction is of type std::function<void(cv::Mat\*)>, which will be called when the OnEventNotified method is called. This std::function is a function which is a lambda capturing \*this\* of the class which updates the Host/UI.
   3. UIUpdateCallbackFuncPtr m\_UIUpdateHandler = nullptr;

* m\_UIUpdateHandler of the type UIUpdateCallbackFuncPtr which is a type def typedef void(\*UIUpdateCallbackFuncPtr)(cv::Mat\*);, is a pointer to the function.
* This member variable is the pointer to the function that needs to be called by the executor when an event is notified, such as when an image is analyzed and need to be displayed by the Host UI.

(Note: This part will be covered in the Event Handler Section below)

**Important to note:**

* UIUpdateHandlerSTDFunction is by default set to Stub method and UIUpdateHandler is set to nullptr
* Only one of the UpdateHandler variable shall be set. This is the reverse communication mechanism across platform from the C++/Core to the host UI.
* In case of QT, UIUpdateHandlerSTDFunction is set while UIUpdateHandler is set when the host is Windows Winforms (C#)
* Please have a look at the various constructors of the FaceIdentificationCommandExecutor class.

1. Event handling mechanism is described in detail below.

## Setting up Event Handlers with command executors

1. The flow of call is back and forth in nature. In order to use the Event handling mechanism provided in the framework, the following settings have to be made.
   1. **Forward call settings**:
      * The client which needs to be used as an observer for an event should implement OnEventNotified() method. For example, the FaceIdentificationCommandExecutor class implements it as follows

void FaceIdentificationCommandExecutor::OnEventNotified(cv::Mat& openCVImage)

{

//if (m\_BitmapFromCVMatFunctionPtr )

// m\_BitmapFromCVMatFunctionPtr(&openCVImage);

if (m\_UIUpdateHandler)

m\_UIUpdateHandler(&openCVImage);

else

m\_UIUpdateHandlerSTDFunction(&openCVImage);

}

* The observer, which here is the FaceIdentificationCommandExecutor, should then subscribe to the FACE\_IDENTIFICATION event by registering itself at the time of construction as follows

// Initialize all the objects for event handlers

AdaptiveHumanTracker::Event eventForSubscription =

AdaptiveHumanTracker::Event::FACE\_IDENTIFICATION;

AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::Register<FaceIdentificationCommandExecutor>(eventForSubscription, \*this);

Please note that singleton EventHandler with templates for types **Event** and **Data Type**, which in this case is cv::Mat is used to invoke the **Register** call with template type for the ObserverType, with the event and the observer objects as the method arguments.

* 1. **Call-back call settings**:
     + There are two types of the call back provisions. One is of a simple Pointer to Function and the other is a std::Function<> constructed by Lambda input.
     + The pointer to function mechanism (**m\_UIUpdateHandler**) is used for WINFORMS using C#/C++-CLI and the std::function<> (**m\_UIUpdateHandlerSTDFunction**) constructed by C++ lambda is used by QT to call their respective platform specific methods from cross-platform framework. Please refer to the FaceIdentificationCommandExecutor.cpp
     + The framework takes the pointer to function as first priority and if it does not exist, it calls the std::function<>, the second one.
     + In order to set the above variables, these are set as initializer list to the constructor of FaceIdentificationCommandExecutor object.
  2. **Notify when identifying the event**
* It is the client’s responsibility to notify all the subscribers/observers to take appropriate action. The RunExecute call is the first one to recognize if the **FACE\_IDENTIFICATION** event has just happened. So the following code notifies all the subscribers/observers to take appropriate action.

if (m\_OpenCVHIGHGUIDisplay)

{

cv::namedWindow("Face identification", cv::WINDOW\_NORMAL);

cv::imshow("Face identification", face\_id\_image);

}

else

{

AdaptiveHumanTracker::Event eventForSubscription = AdaptiveHumanTracker::Event::FACE\_IDENTIFICATION;

// Framework API to notify on FACE\_IDENTIFICATION event AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::Notify(eventForSubscription, face\_id\_image);

}

IN the above code snippet from FaceIdentificationCommandExecutor.cpp, there are two blocks for IF and ELSE. The IF block displays the image using OpenCV frame and is suitable for console output clients.

The ELSE block is the one that the framework notifies. On notification, all the observers that are subscribed to the event gets called in the same thread.

1. The file Events.h defines many events that the feature programmer shall be interested in. They are as follows

namespace AdaptiveHumanTracker {

enum class EVENTSOBSERVERSNOTIFIERS\_API Event

{

FACE\_REGISTRATION,

FACE\_DETECTION,

FACE\_IDENTIFICATION,

HUMAN\_DETECTION,

BODY\_IDENTIFICATION,

ADAPTIVE\_PERSON\_IDENTIFICATION,

WORK\_STARTED,

WORK\_RUNNING,

WORK\_ABORTED,

WORK\_SUCCEEDED,

WORK\_FAILED

};

1. Based on the nature of the Host/UI, the method which actually updates the image of the Windows Form or the slot in QT shall be used as the callback method while constructing the Client/CommandExecutor.
2. In QT, the file identify.cpp, shall be an example for Face Identification skill. The method which is responsible for updating the image is

void identify::OnImageNotified(cv::Mat\* notifiedImage)

{

QImage matQimage = this->getQImage(\*notifiedImage);

QPixmap pixmap(QPixmap::fromImage(matQimage));

this->MainWindow\_obs->pixmapImage = pixmap;

emit imageReceived();

cv::waitKey(1);

}

1. However this method cannot be directly passed onto the Command executor to be called when an image is emitted with the identification by the core engine. In order to package it and call this method, a lambda of this method is returned, which is then set to the command executor. Please have a look at the code below.

std::function <void (cv::Mat\* mat)> identify::GetImageNotifiedFunction()

{

return [this](cv::Mat\* mat) mutable { return this->OnImageNotified(mat);};

}

1. The command executor is constructed from the Host/UI executor button press event call back as follows
2. Construction of Command executor with event handlers

The events mentioned in Point-1 are/have to be unique. The current example, Face Identification uses **FACE\_IDENTIFICATION type.**

The event handling has four steps

1. **Register the client:**

* The client in this case is the command executor itself. So, as part of the agreement, the command executor has to implement void OnEventNotified(ObservableDataType& data).
* Face identification implements the above method as follows

void FaceIdentificationCommandExecutor::OnEventNotified(cv::Mat& openCVImage)

{

if (m\_UIUpdateHandler)

m\_UIUpdateHandler(&openCVImage); // Winforms (C#)

else

m\_UIUpdateHandlerSTDFunction(&openCVImage); (QT)

}

* The observable data type here is cv::Mat. Either the m\_UIUpdateHandler or the m\_UIUpdateHandlerSTDFunction will be called.
* Registration of the client (observers) have to be made while at the construction time of the Command Executors.
* The method RegisterHostUIUpdateEventHandler in the class FaceIdentificationCommandExecutor is defined and called in the FaceIdentificationCommandExecutor constructor just before returning.
* Notable points in the register method are

void FaceIdentificationCommandExecutor::RegisterHostUIUpdateEventHandler()

{

**// Pick the observable event : FACE\_IDENTIFICATION**

AdaptiveHumanTracker::Event eventForSubscription =

AdaptiveHumanTracker::Event::FACE\_IDENTIFICATION;

**// Call the Event Handler method as follows**

AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::**Register**<FaceIdentificationCommandExecutor>(eventForSubscription, \*this);

}

* Registering the command executor as an observer subscribes it to FACE\_IDENTIFICATION event triggered anywhere.

1. **Notify when a face is identified:**
   * Because the image has to be displayed onto the Host UI(QT or windows), the event FACE\_IDENTIFICATION is notified. This is made as follows in the RunExecute() code of FaceIdentificationCommandExecutor’s private implementation.

for (size\_t j = 0; j < humandetect\_msg.persons\_size(); j++)

{

humandetect\_msg.mutable\_persons(j)->set\_source(NEED\_RECOG);

if (m\_OpenCVHIGHGUIDisplay)

{

cv::namedWindow("Face identification", cv::WINDOW\_NORMAL);

cv::imshow("Face identification", face\_id\_image);

}

else

{

**// Pick the FACE\_IDENTIFICATION event as we know that //the face is identified**

AdaptiveHumanTracker::Event eventForSubscription = AdaptiveHumanTracker::Event::FACE\_IDENTIFICATION;

**// Notify to all the observers/clients subscribed to the //FACE\_IDENTIFICATION event. This call will call the //OnEventNotified() on the observer/Client, in our case //FaceIdentificationCommandExecutor::OnEventNotified**

**AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::Notify(eventForSubscription, face\_id\_image);**

}

}

1. **Take appropriate action when OnEventNotified method is called on the client/FaceIdentificationCommandExecutor**

void FaceIdentificationCommandExecutor::OnEventNotified(cv::Mat& openCVImage)

{

**// If Windows C# winforms UI handler is set**

if (m\_UIUpdateHandler)

m\_UIUpdateHandler(&openCVImage); // Winforms (C#)

else **// If QT UI handler is set**

m\_UIUpdateHandlerSTDFunction(&openCVImage); (QT)

}

1. **Unregister when done with UI handling and in destructor of the command executor**

The method UnregisterHostUUpdateEventHandler() is defined in FaceIdentificationCommandExecutor whose body is as follows

void FaceIdentificationCommandExecutor::UnRegisterHostUIUpdateEventHandler()

{

**// Pick the appropriate event to unsubscribe**

AdaptiveHumanTracker::Event eventForSubscription =

AdaptiveHumanTracker::Event::FACE\_IDENTIFICATION;

**// Unsubscribe the observer from the occurrence of the event**

AdaptiveHumanTracker::EventHandler<AdaptiveHumanTracker::Event, cv::Mat>::**UnRegister**<FaceIdentificationCommandExecutor>(

eventForSubscription, \*this);

}

The above method is called in the destructor of the FaceDetectionCommandExecutor as follows

FaceIdentificationCommandExecutor::~FaceIdentificationCommandExecutor()

{

UnRegisterHostUIUpdateEventHandler();

}

## Executing any Skill from any Host/UI/Console

The following example shows how to execute the Skill

Prerequisites:

1. The Skill that needs to be executed should be created as a command executor and constructed with setting up of appropriate event handlers if any needed.
2. The Host/UI will know only the command executor and so unnecessary headers should not be exposed in the header file of the command executor
3. The command / Skill Execute framework needs the command executor to implement RunExecute() as mentioned above. Care should be taken to adhere to the return type and the signature.
4. If any event has to be handled, then the event handlers should also be set as described in the example FaceIdentificationCommandExecutor.
5. The below example shows how to execute the command. This one is picked from the QT host.

In Identify.cpp in the project AdaptivePersonIdentification, the following lines of code are executed when the Skill Execute button for **Face Identification** is pressed.

if (this->m\_SkillType == SkillType::FACE\_IDENTIFICATION)

{

**// Populate the data for command input**

FaceIdentificationCommandExecutorInputData FaceIdentificationCmdInputData;

FaceIdentificationCmdInputData.modelDirectoryPath = ui->modelDir\_LE->text().toStdString();

FaceIdentificationCmdInputData.dataDirectoryPath = ui->dataDir\_LE->text().toStdString();

**// Get the captured lambda of the member method of the Host/UI which updates the image**

auto onNotifyCb = GetImageNotifiedFunction();

**// Construction of the Face Identification Command Executor**

m\_FaceIdentificationExeutor = std::unique\_ptr<FaceIdentificationCommandExecutor>(new FaceIdentificationCommandExecutor(**onNotifyCb**, **false**));

**// In the above line, OnNotifyCB is the std::function OF the captured lambda OF the UpdateImage method of the Host UI.**

**// Execute the Skill as follows.**

std::unique\_ptr<NullData> otNullData = AdaptiveHumanTracker::SkillsManager<FaceIdentificationCommandExecutorInputData, NullData>::GetInstance()->Execute<FaceIdentificationCommandExecutor>(FaceIdentificationCmdInputData, \*m\_FaceIdentificationExeutor);

}

## Command Manager and wrapper methods

ISkill<INputData,OutputData> \*m\_Skill

template<typename SkillExecutor>

std::unique\_ptr<SkillOutputDataType> **Execute**(SkillInputDataType& inputData, SkillExecutor& executor)

**Template**<InputDataType, OutputDatatype>

Tempate<InputData,OutputData>

Bool Execute(InputDataType& data)

Void Terminate()

**ISkill (Interface)**

Skill Manager

**SkillsFactory**

Template<InputData,OutputData,Executor>

Static ISkill< InputData , OutputData > CreateSkill(executor)

GenericSkillImplWrapper<InputData, OutoutData, SkillExecutor>::GenericSkillImplementer::GetInstance(executor);

Return instance

m\_Skill->Execute(inputData)

QT (Host/UI) -Invoker

WinForms(C#) -Invoker

Diagram-2

**Skill(Command) classes hierarchy: Command pattern with Bridge and CRTP Implementation. Diagram-3**

**Template**<InputDataType, OutputDatatype>

Bool Execute(InputDataType& data)

Void Terminate()

**ISkill (Interface)**

**Template**<GenericSkillImplWrapper, InputDataType, OutputDatatype>

Bool Execute(InputDataType& data) override

Void Terminate() override

**GenericSkill (Bridge)**

static\_cast<GenericSkillImplementor\*>(this)->ActuallTerminate(data);

static\_cast<GenericSkillImplementor\*>(this)->ActuallyExecute(data);

**Template**<InputDataType, OutputDatatype, ExecutorType>

**GenericSkillImplWrapper (Concrete Implementation Wrapper)**

**GenericSkillImplementer(Concrete Implementation Wrapper)**

Reference to AnyCommandExecutor

Singleton to GenericSkillImplementer

SkillExecutor->RunExecute(inputData);

GetInstance()

AnySkillExecutor->Terminate();

ActuallyExecute(InputDataType& data)

ActuallyTerminate(InputDataType& data)

**Event Handler classes hierarchy: Observer pattern with Bridge and CRTP Implementation.**

**Diagram-4**

**Template**<ObservableDataType>

void UpdateAction(ObservableData& data)

**IObserver (Interface)**

**Template**<GenericObserverImplWrapper, ObservableDataType>

Bool UpdateAction (ObservableDataType& data) override

**GenericObserver(Bridge)**

static\_cast<GenericObserverImplementer\*>(this)->ActuallyUpdateAction(static\_cast<observer\_Data\_type&>(data));

**Template**<, ObservableDataType, ClientType>

GenericObserverImplWrapper **(Concrete Implementation Wrapper)**

**GenericObserverImplementer(Concrete Implementation Wrapper)**

AnyClient ->OnEventNotified

Reference to AnyClient

ActuallyUpdateAction(ObservableDataType & data)

## Salient features of the framework:

### Advantages

1. **The framework encourages loose coupling between components.** The framework does not even dictates the skill executor to implement an interface full of methods. It just needs one method to be implemented, namely RunExecute(). In otherwords, the command executor shall choose to follow a completely different hierarchy while allowing itself to be the command executor also, while having multiple other roles.

Also, this loose coupling provides a lot of space for the algorithms developer to continue with experimenting with various algorithms without having to adhere to design constraints

While a client needs to observe an event, it just needs OnEventNotified to be implemented.

The Command Pattern together with the Bridge and Curiously Recurring Template Pattern does not force the programmer to always inherit from ISKill<> interface, rather provide a class that has the implementation for a specific Skill.

1. **The framework is highly cohesive adhering to SOLID design principles.**

Each component in this framework is designed to carry out a very specific task.

* 1. The interface is meant only to have a base class pointer
  2. The Bridge and CRTP are meant only to introduce type safety and decoupling the command executor from the dictates of ISKill<> base class
  3. The private implementations are meant to reduce coupling among methods inside the parent class and the private implementer is responsible for running the command/Skill.

1. **Extensibility/Interchangeability:**
   1. This framework has been designed to accommodate execution of any chunk of tasks and does not differentiate between the application domains. This sae framework shall be used in applications that are used for various other domains such as Avionics, CAD, CAM etc.,
   2. The framework alone can be used for other applications which do not have one. The setting up of the framework is very easy. The driver is the command executor which many application would already have. The UI/Host is very easy to be integrated with the framework. Especially for the M.L applications, this is quite suitable because it does not have any dependency on platform, Host-UI, or the class hierarchies of the implementing objects, that run the complicated algorithms.
   3. Most of the framework has been implemented using templates and so inclusion of the header will just be needed.
2. **Type-Safe:**

The unique selling point for this framework is the type-safe mechanism that has been implemented underneath. No Skill/Command call shall even compile if there is a mismatch between the various types of data if encountered.

Legacy software solutions have a lot compulsions when implementing such frameworks and they use void\* (WINWORD) kind of data while casting back to the known formats. These are completely avoided.

Though the UI will know only ISkill<InputDataTYpe,OutoutDataTYpe> type pointers, the underlying implementation is completely encapsulated, more so with private implementations. SO even the headers that contain the algorithm implementations are not exposed.

1. **Customizable**:

The framework defines Skills that defines **Execute** and **Terminate calls.** The Skills can have multiple methods such as **Undo(), Redo(), Journalize()** and shall also be implemented for UI less, batch mode executions, which case **Journalize** () call will be utilized.

The Same Skills shall be wrapped for a Python, JAVA or VB callers as a script to be used by a different application.

A new type of a command (as in command pattern) shall be defined and implemented. Currently, the command takes in a data (of a type) returns a data of a type, while event handlers/observers update the HOST with the data/images.

1. **Modular design:**

The framework shall be integrated by any application since it does not depend on the OS.

### Limitations

1. The framework is still evolving and contains some POC designs as well. These shall be made very robust only by evolving the design based on various implementers who use this framework from their feedback.
2. The current framework does not use a single type of compiler in all the platforms. In Linux, it GCC is used while, CL is used in Windows. This might give rise to incompatibilities between the low level design of the framework and the compilation. Since various compilers do not implement policy classes so effectively (generics), the compilers might not succeed in some platform.
3. The release and packaging framework is very new. The entire product depends on various third party libraries and their C++ include files cause problems when they do not adhere to the multi-threading or C++17 specifications. Some of the warnings on those lines are silenced while some of the third party headers are modified to un-expose the headers from product build.
4. The algorithm implementation uses a lot of shared pointers and has included many includes that are visible even for the framework. These unnecessary includes should be avoided by designing a private implementation for all the .CC files. This way, the framework will not know of the implementation details of the algorithms.
5. The entire framework is split in to multiple DLL binaries. Though this ensures that the immediate dependencies to be intact, the intactness of the farther dependencies are not checked. One such example is the CPU\_EXTENSIONS.DLL, which needs to be built practically for each hardware, not just for every OS.

## A Few more words

Right from the start, an honest attempt has been made to make this framework truly desirable on the parameters mentioned as its **Advantages**. A very recent addition to the list is to explore calling Python from C++ from the framework side’s executor’s end.

The framework has been designed from the past experience of using them in various engineering software solutions taking the best of all of them. We try to follow the golden words of the DESIGN, which is **what is designed should not be seen and what is seen should not be designed.**

**Parthasarathy Srinivasan**