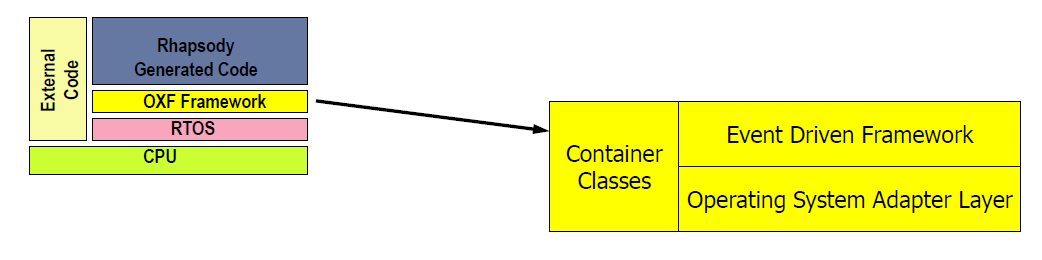
**Part 1. OXF = Object eXecution Framework**

It serves as a layer between the application and the RTOS.



The OXF layer consists of a set of classes which implement the event driven framework.

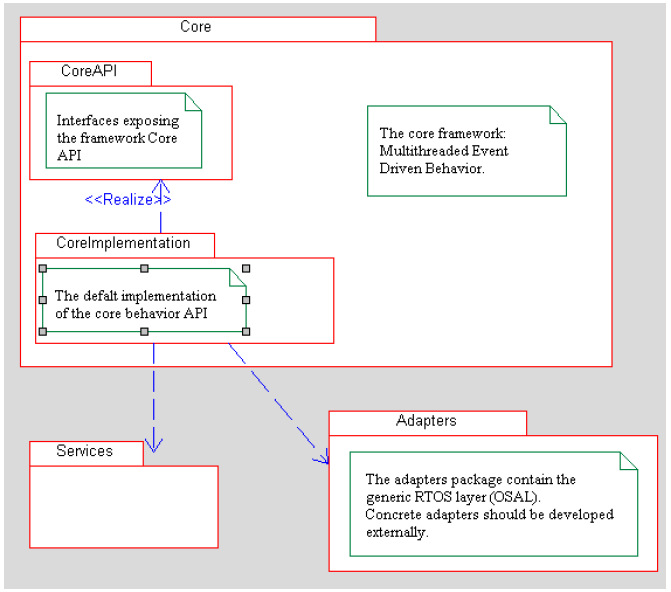
There are 3 main parts

1. Event Driven Framework (Core API and Core Implementation package)
2. Operating System Adapter Layer (OSAL in Adapter package)
3. Services (Services package includes e.g. Container class)

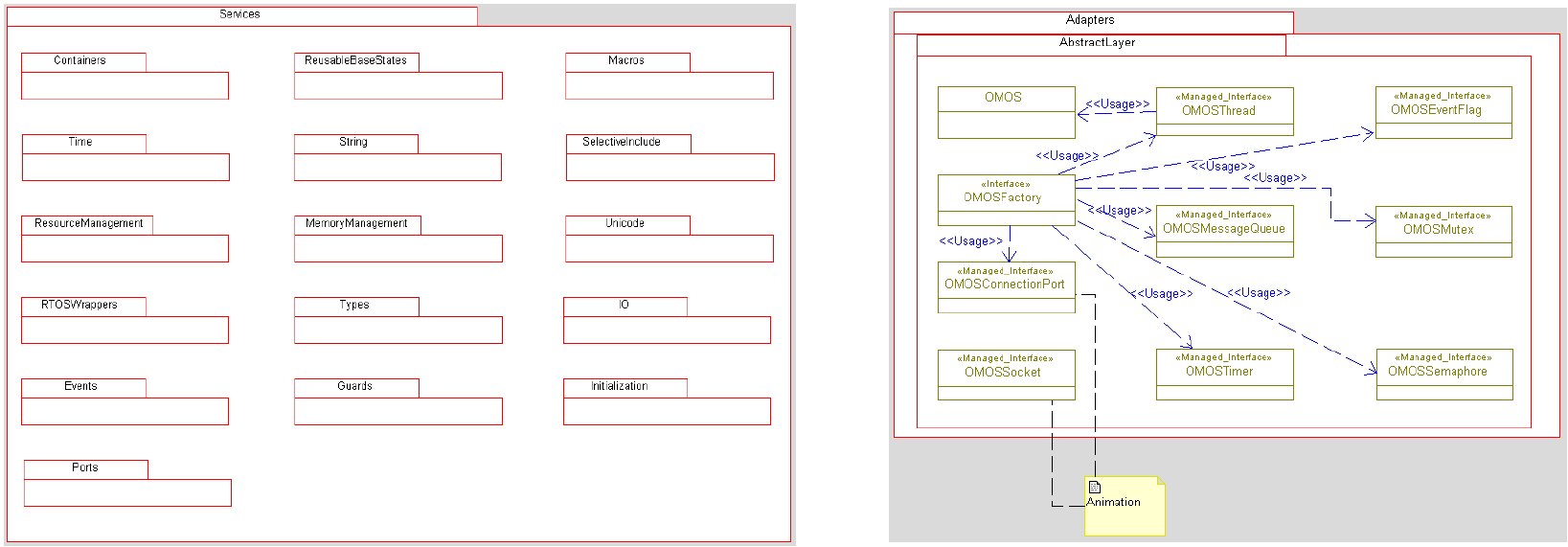
OXF design package

+ Consist of many different nested packages.

+ There is a set of interfaces in CoreAPI and a default implementation in CoreImplementation for framework behavior. This allows users to provide their own implementation.



OXF Services package and Adapters package (OSAL)

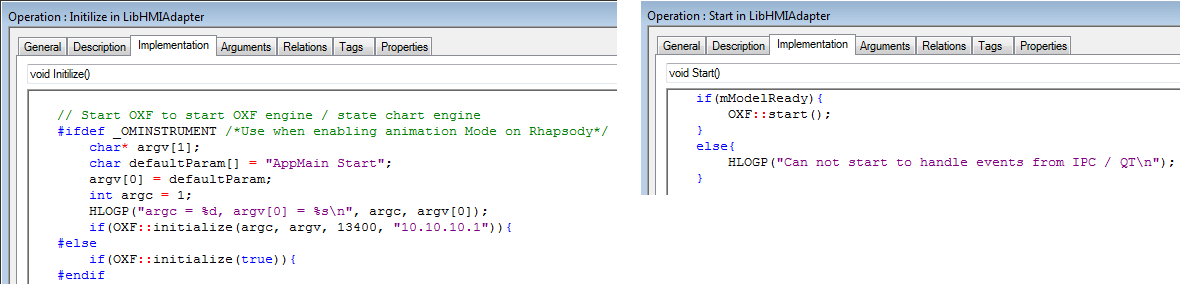


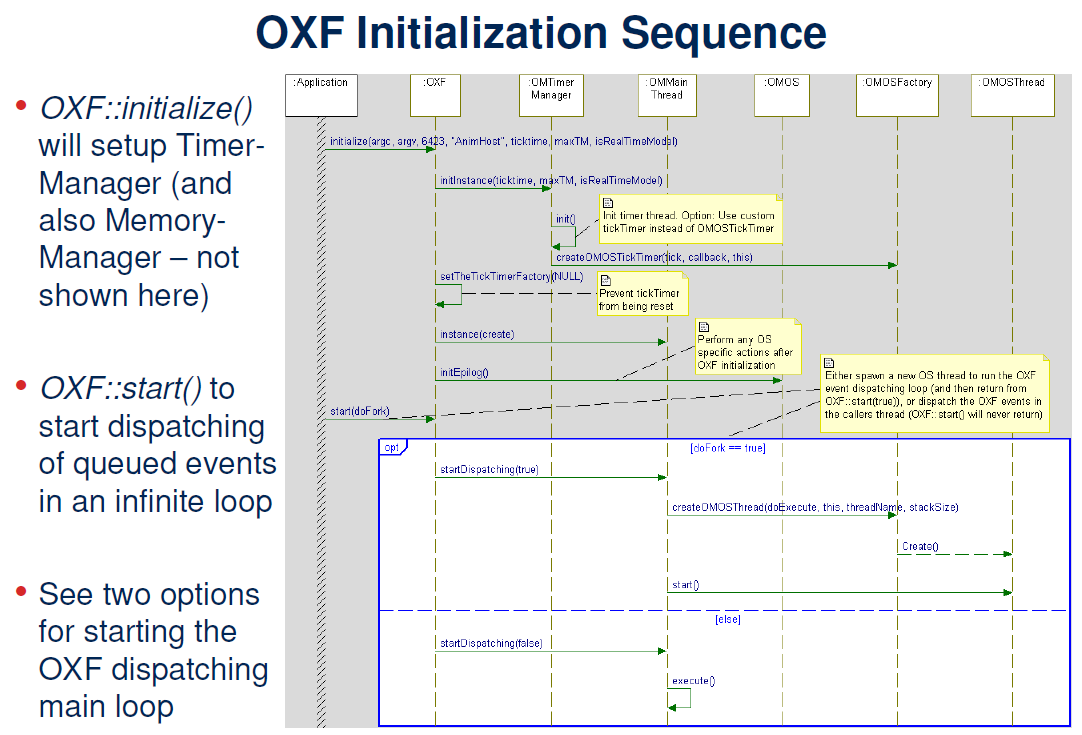
OXF Initialization

* OXF::initialize() – to create the timer thread, and set the default active class
* OXF::start() – to create an OMOS wrapper around the calling thread so that OXF run as the mainThread
* OXF::start(true) – to spawn a new thread, in which the OXF dispatching loop will run and not return.
* OXF::end() – must be called on the same thread, which runs the OXF dispatching loop

A typical main routine generated in Rhapsody







**\*\*\* Client and Server Example Using OXF code**

+ Create new Rhapsody projects

+ Create new Object Model Diagrams (OMD)

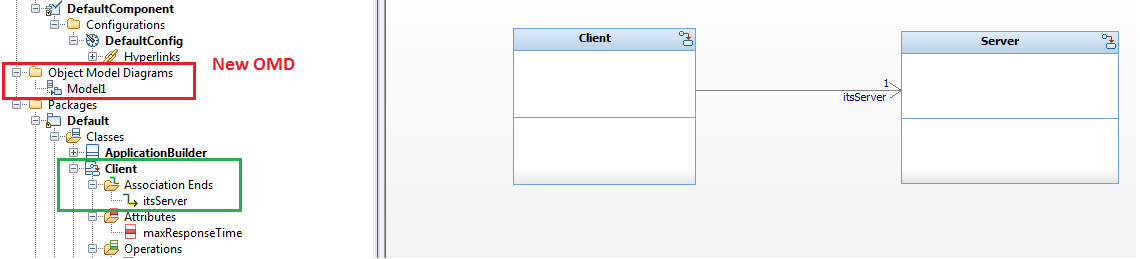
+ Create 2 new classes named “Client” and “Server”

+ **Add new Statechart** to both classes => in generated code, we detect several OXF includes (omreactive.h, state.h, event.h) and inheritance from OMReactive inside the header file.

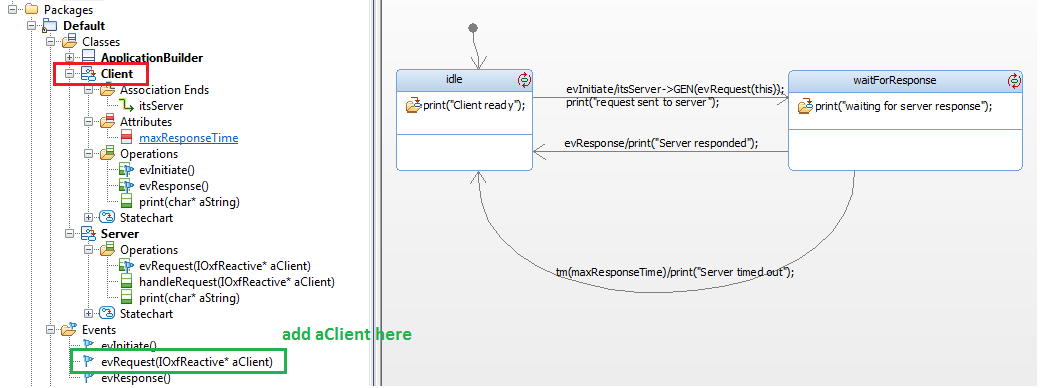
+ Draw state-chart for Client and Server

+ Create a new ApplicationBuilder class: Client Server instantiation

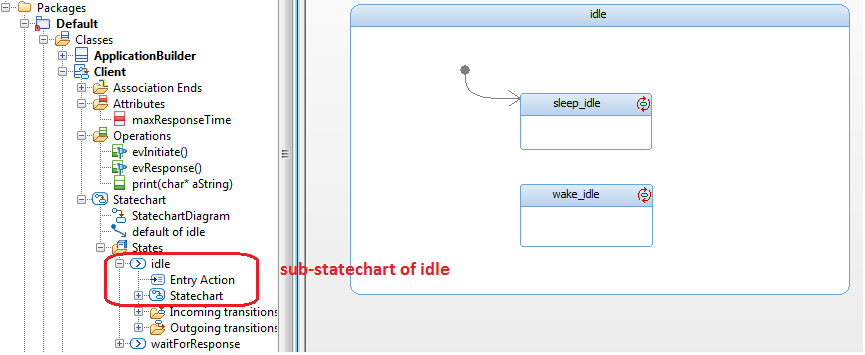
1 – OMD: Client and Server



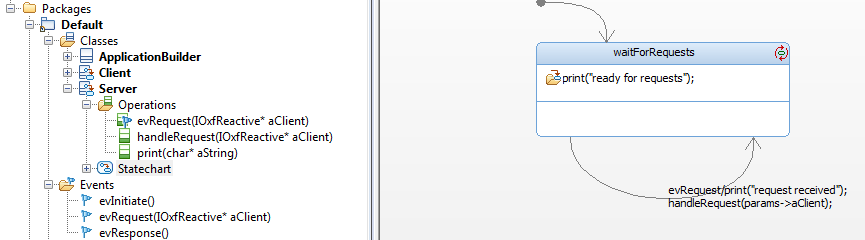
2 – class Client



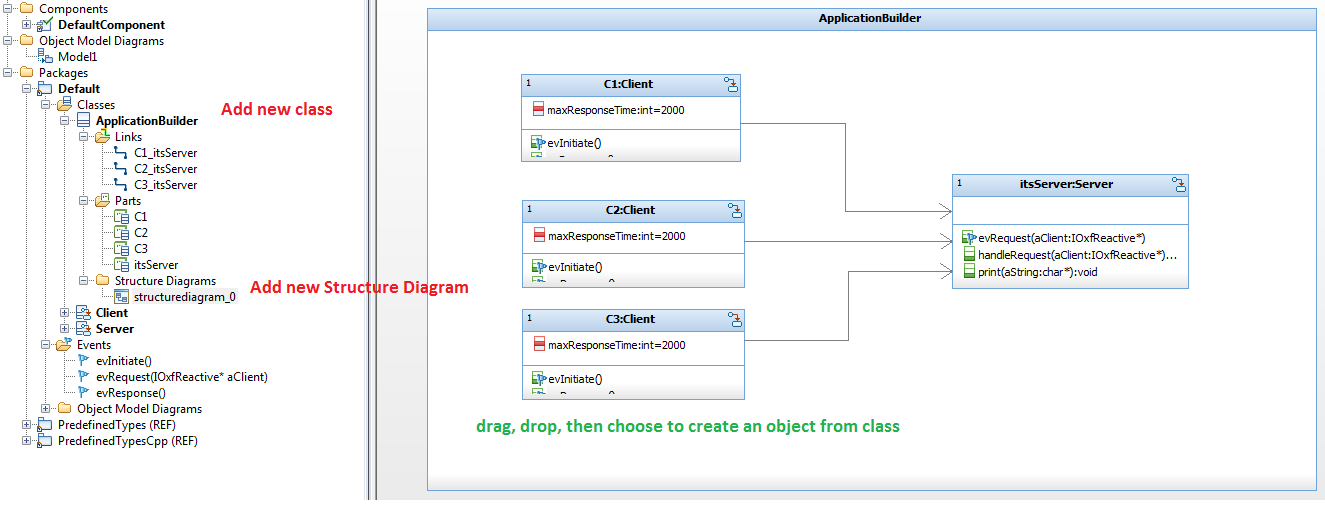
Add some sub-state of idle



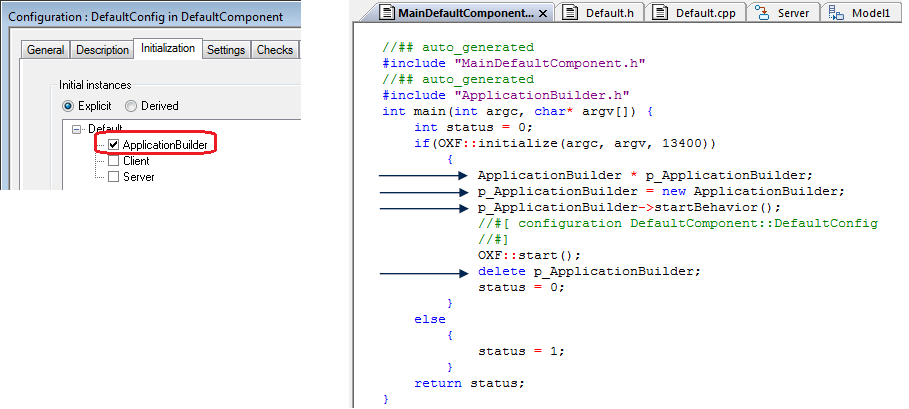
3 – class Server



4 – class ApplicationBuilder



5 – Application initialization

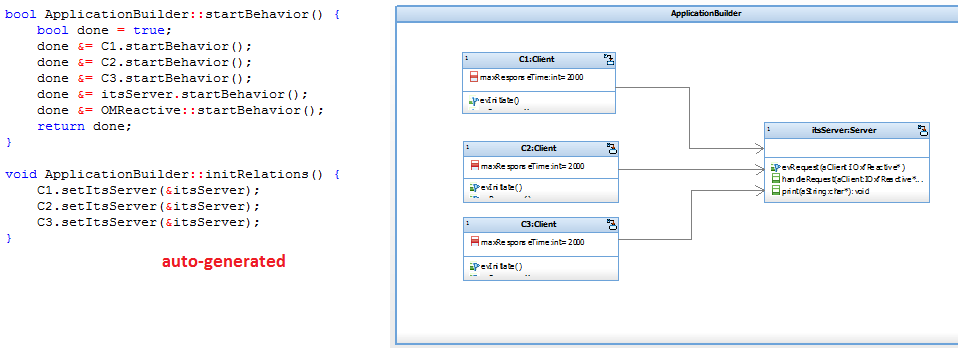


**Part 2. Inspect generated code**

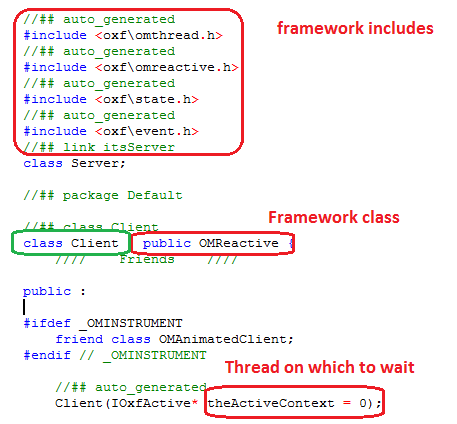
\*\*\* Class ApplicationBuilder

+ init relationship between objects

+ start behavior each object



Typical generated header of a class which consists of state-chart

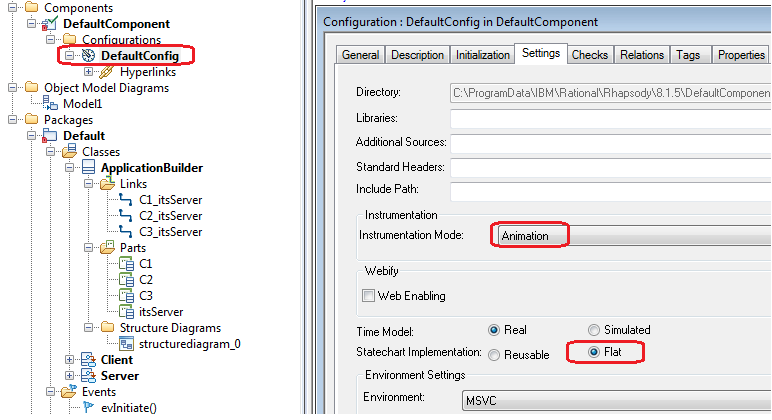


\*\*\* OMReactive

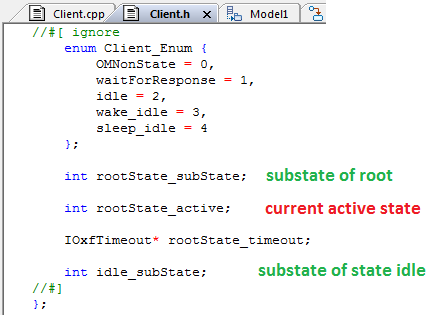
OMReactive is one of Rhapsody’s framework base classes. This is a class that simply waits for timeouts and events. When receives a timeout or an event, it calls the rootState\_processEvent operation (if Flat).

The Rhapsody framework allows two ways of implementing statecharts:

* ***Reusable***: based on the state design pattern where each state is an object. Results in faster execution and if a lot of statecharts are inherited can result in smaller code.
* ***Flat***: uses a switch statement. Results in less code that is easier to ready, but is slower.

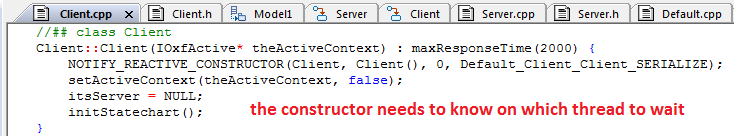


Each state is assigned to a unique number

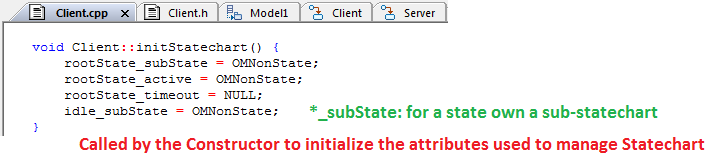


\*\*\* Some need-to-know functions in Flat mode

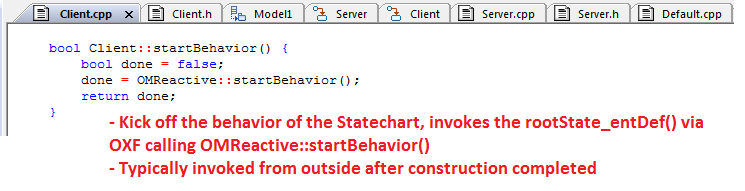
1 - Constructor



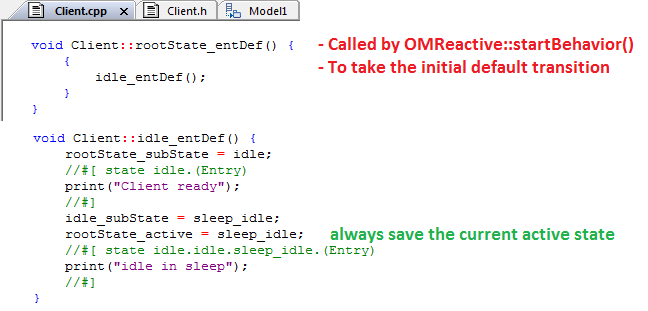
2 - Client::initStatechart()



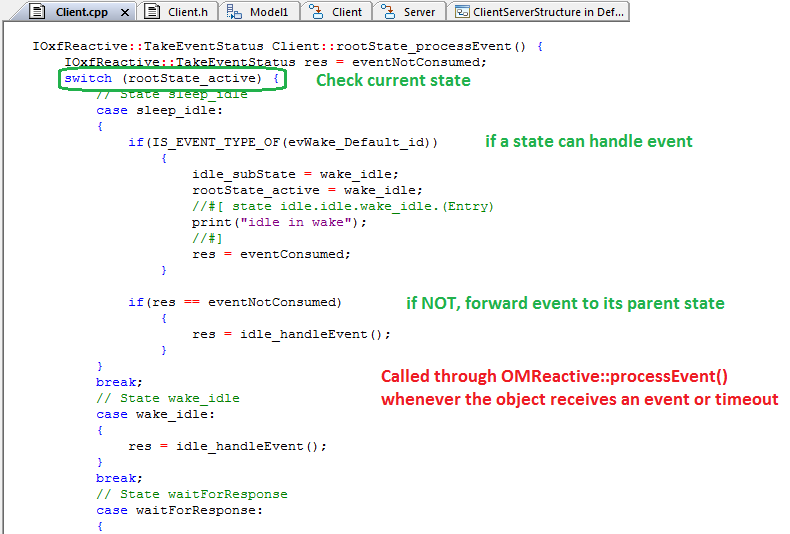
3 – Client::startBehavior()



4 – Client::rootState\_entDef()

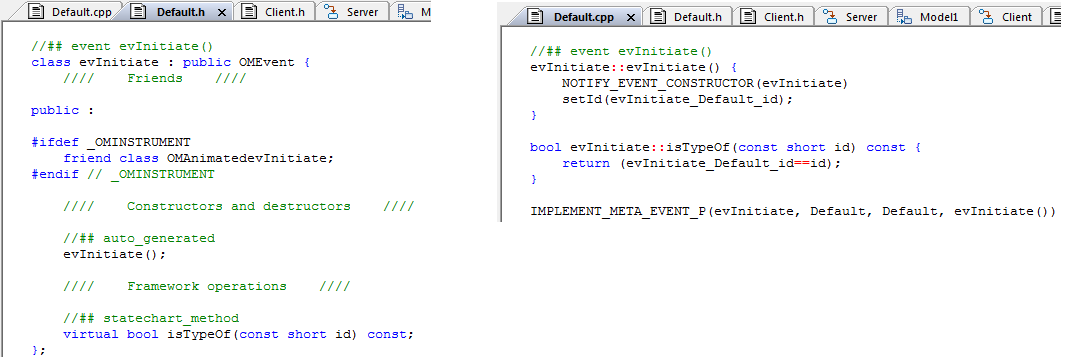


5 – Client::rootState\_processEvent()

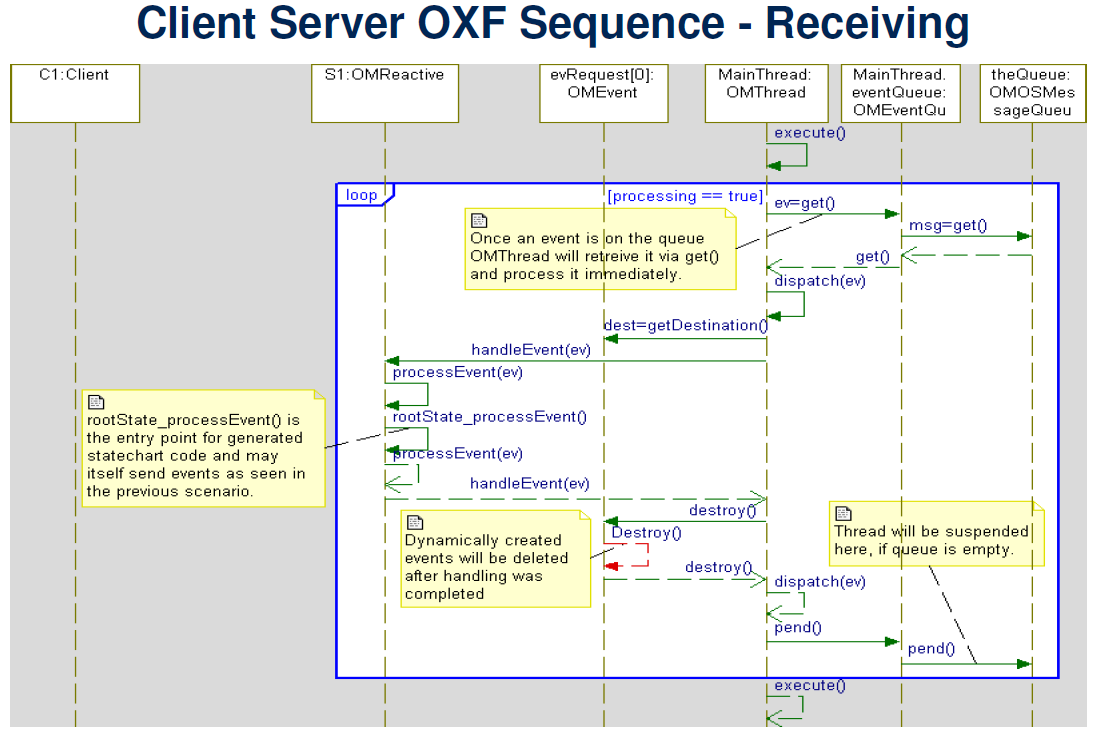


\*\*\* Event specification based on OXF

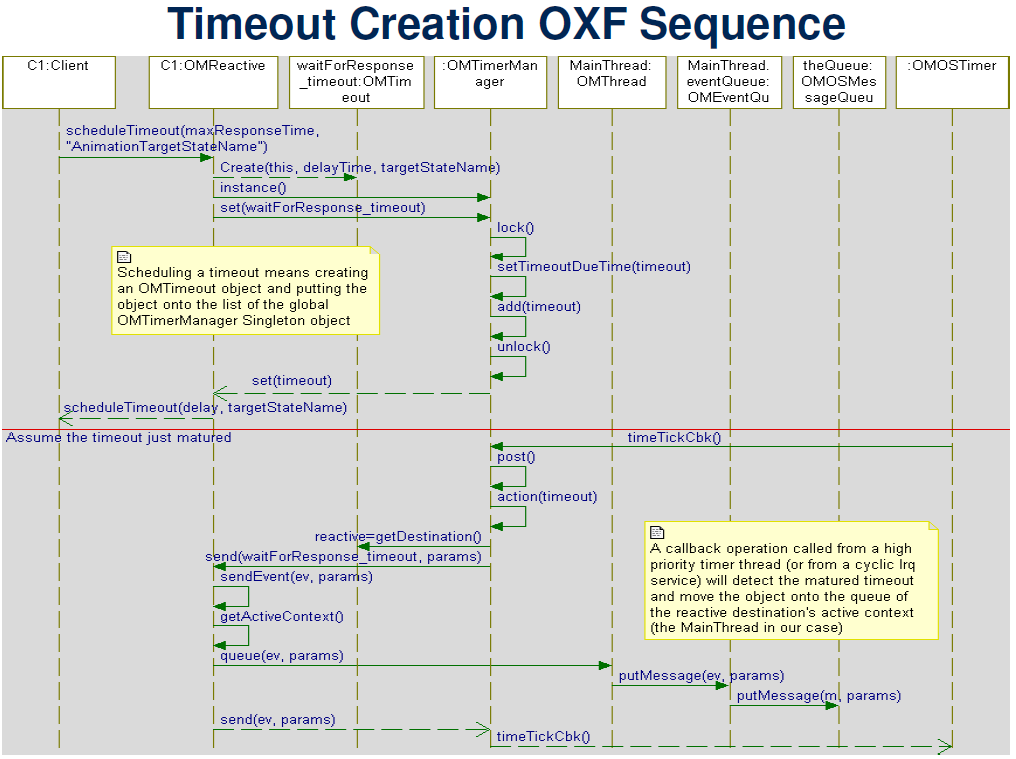
Any event is defined as a class, which inherits from OXF class OMEvent.



\*\*\* Event Sending and Receiving

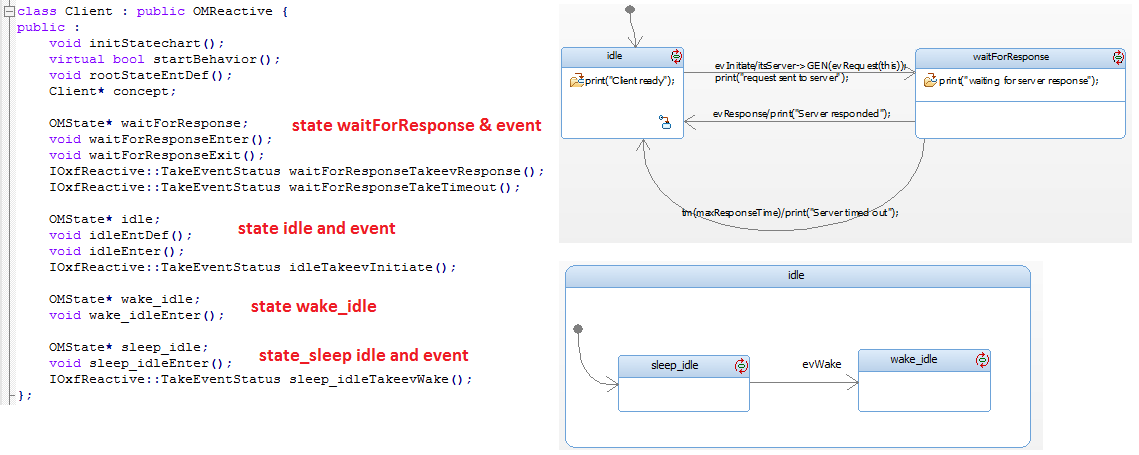


\*\*\* Timeout handling



\*\*\* State-chart Implementation: Reusable

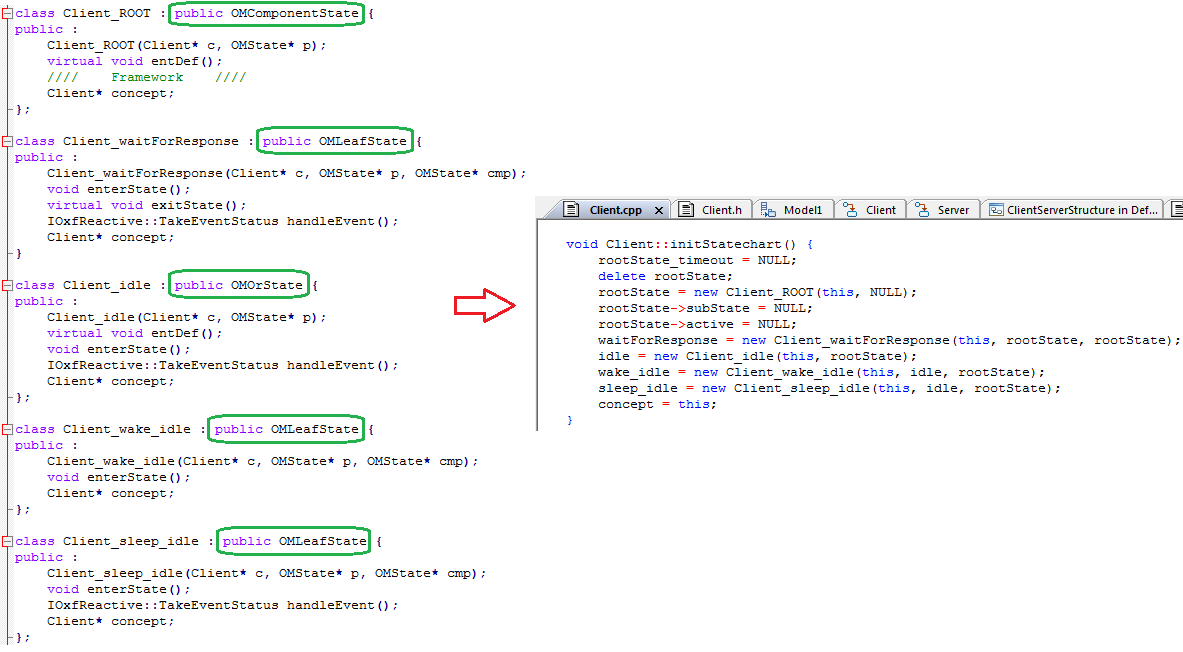
1 - Each state is an OMState object.



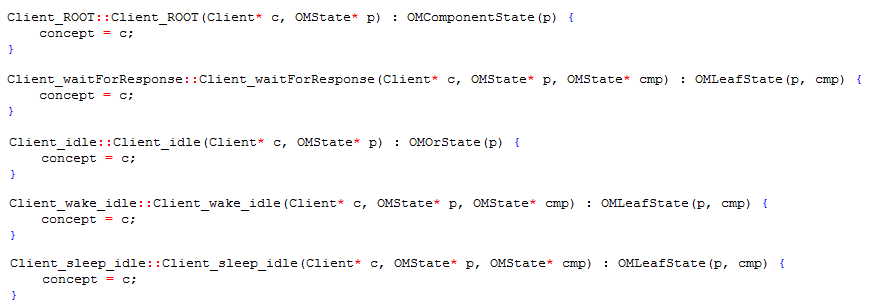
initStateChart()

+ create state objects: rootState, waitForResponse, idle, wake\_idle, sleep\_idle

+ make rootState->subState, rootState->active = NULL (inherited from class OMComponentState)



Each class maintains a pointer to class Client (concept)



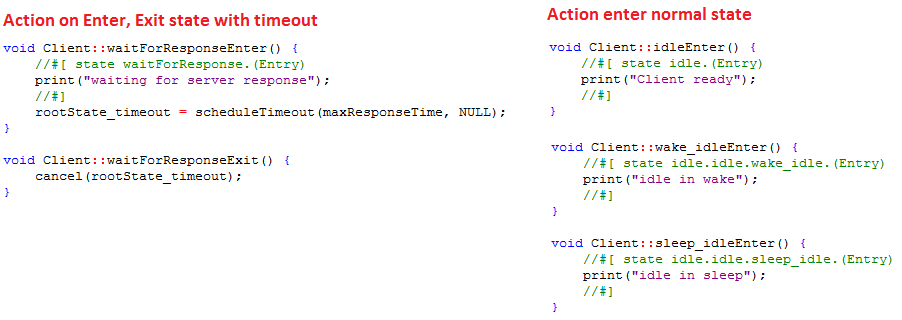
2 – Default transition

Enter sub state from Root state.

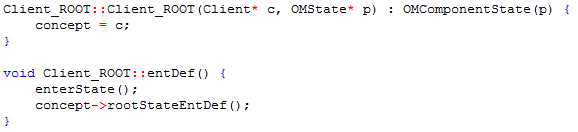


3- Enter and exit state

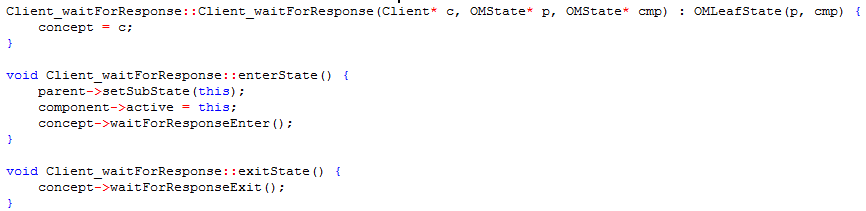
Class Client



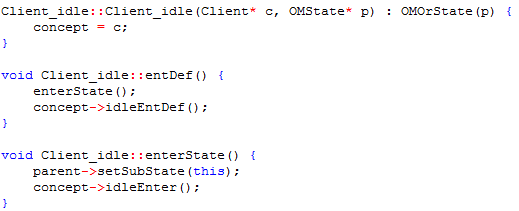
Class Client\_ROOT



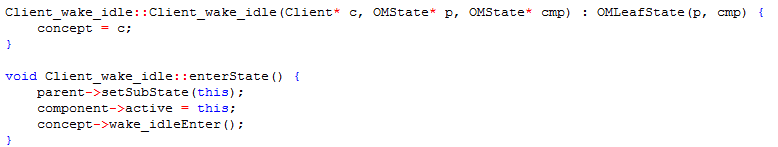
Class Client\_waitForResponse



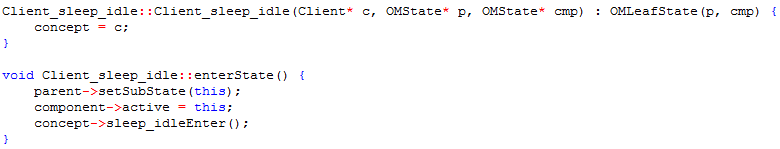
Class Client\_idle



Class Client\_wake\_idle

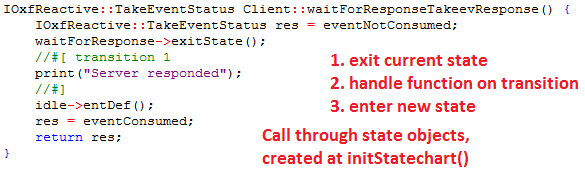


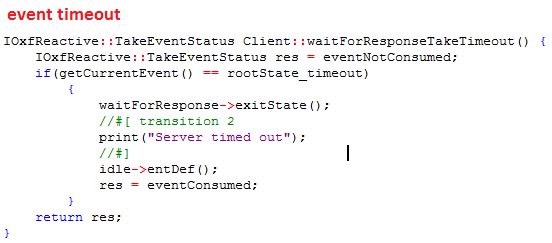
Class Client\_sleep\_idle



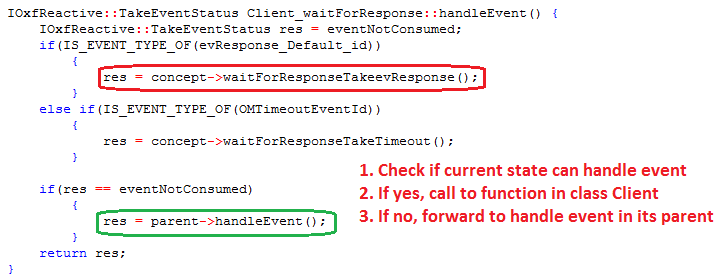
4- Event handle

Class Client



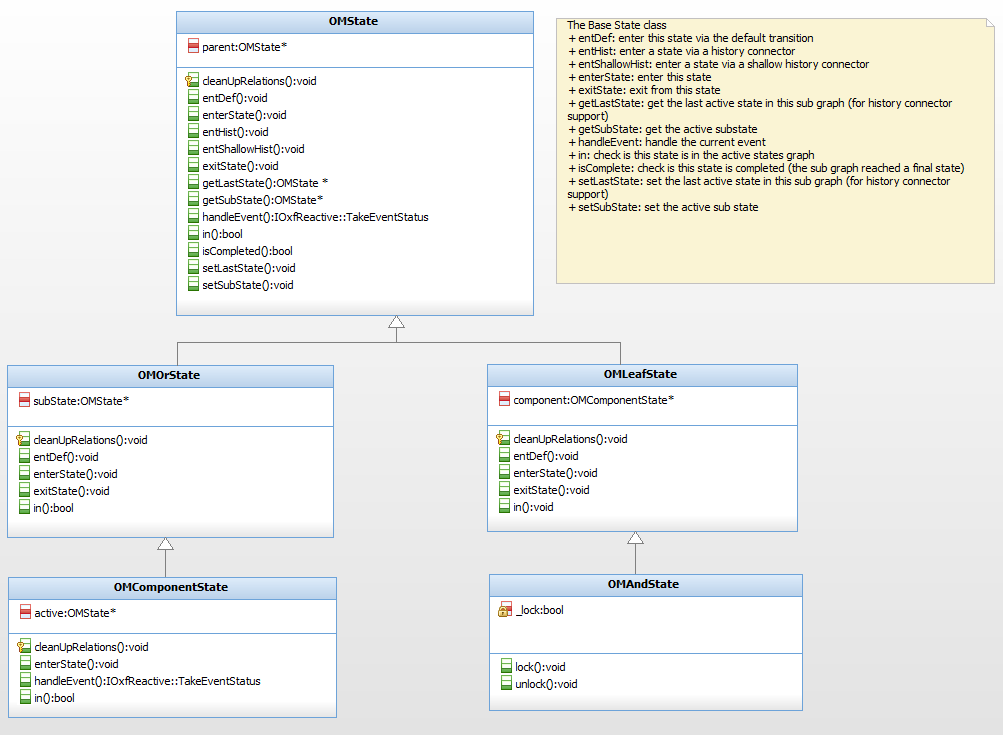


Class Client\_waitForResponse handles event



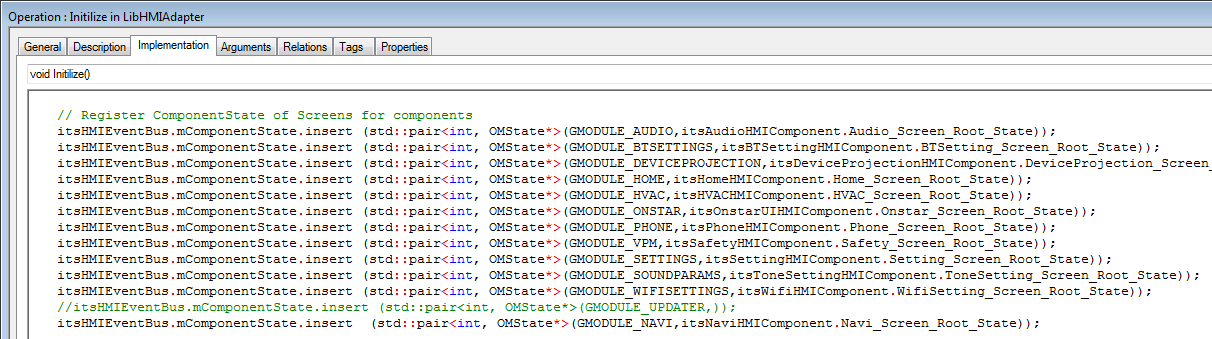
\*\*\* OXF State classes

{ProgramData}\IBM\Rational\Rhapsody\8.1.5\Share\LangCpp\oxf\state.cpp, state.h



\*\*\* Apply to our INFO3.5L project

1 – Register root state for each component



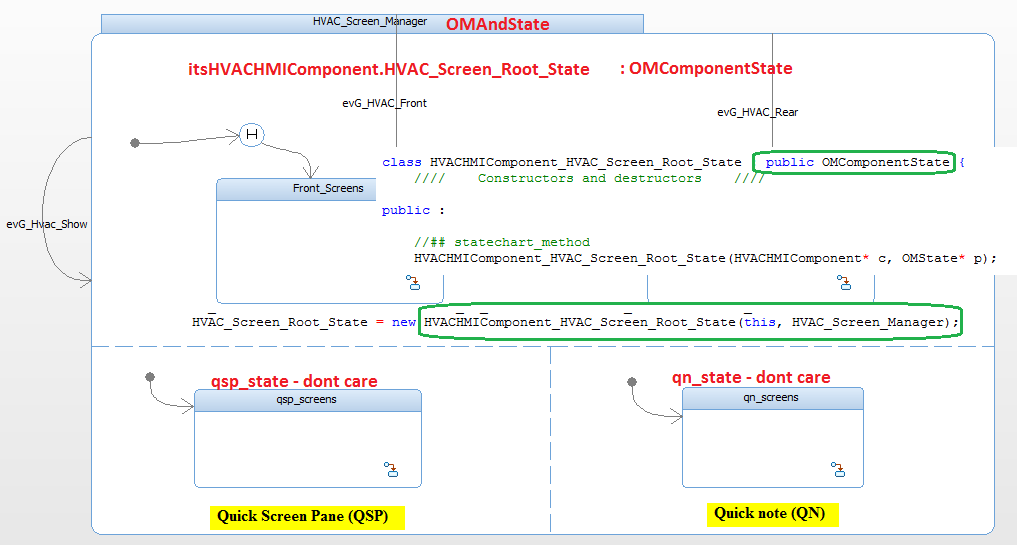
Why do we need to define a root state?

+ We always divide the first state of each component into 2 or more sub-states by AND-line (one for screen, others for overlay states)

+ As a result, first state will be type of OMAndState

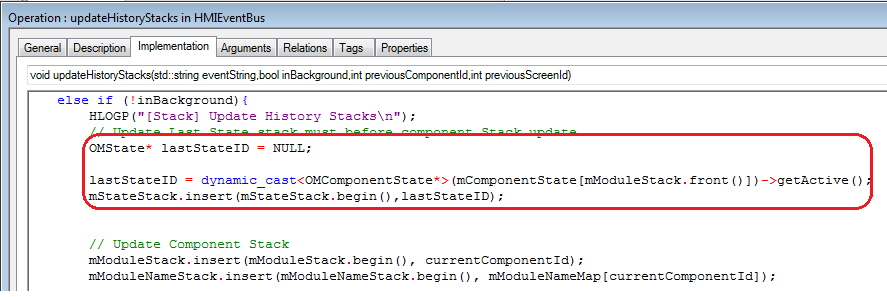
+ But OMAndState does not have subState (means cannot use history)

+ So, we keep a list of root state of every components

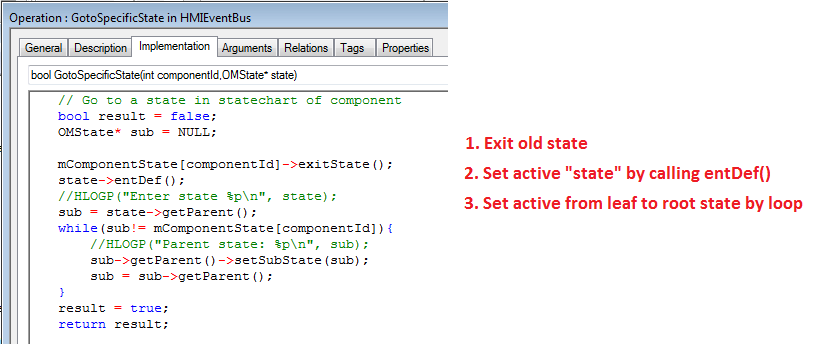


2 – Get current Active state

Purpose: save last state to mStateStack for back



3 - GotoSpecificState(int componentId, OMState\* state)



Part 3. Event Queue