Model::Main

```
RUN FUNCTION
while (!m_stopped.load()) {
        std::function<void()> message;
        {
            std::unique_lock<std::mutex> lock(m_mutex);
            m_condition.wait(lock,[&]{return !(m_messageQueue.empty()) || m_stopped.load();});
            if (!m_messageQueue.empty()) {
                message = m_messageQueue.front();
                m_messageQueue.pop();
            }
        }
        if (message) {
            message();
        }
}
```

ActiveObject

+m_thread: std::thread
+m_messageQueue: std::queue<std::function<void()>>
+m_mutex: std::mutex
+m_condition: std::condition_variable
+m_stopped: std::atomic_bool = false

+run()
+ActiveObject()
~ActiveObject()
+send()

```
Constructor ActiveObject
m_thread([this]() { run(); })
```

```
DESTRUCTOR
{
        m_stopped.store(true);
        m_condition.notify_all();
 }
 m_thread.join();
```

```
int main()
{
    ActiveObject myObject;
    myObject.send([]() {
        std::cout<<" first event "<<std::endl;
    });
    myObject.send([]() {
        std::cout<<" second event"<<std::endl;
    });
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout<<" hello world "<< std::endl;
    return 0;
}
```

```
 void send(std::function<void()> message) {
     {
         std::unique_lock<std::mutex> lock(m_mutex);
         m_messageQueue.push(message);
     }
     m_condition.notify_all();
 }
```