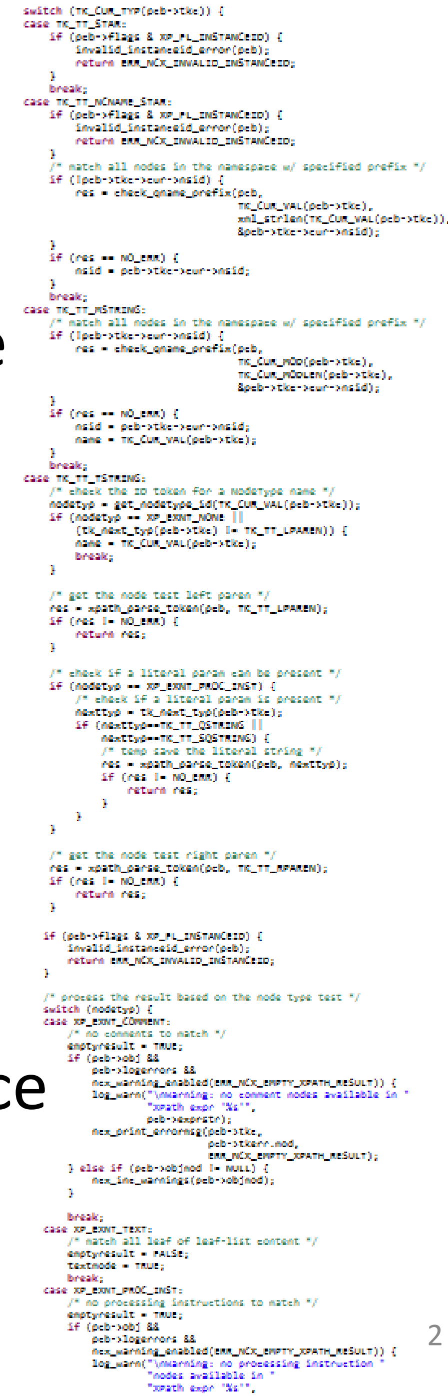
How To Write A Testable State Machine

Matthew Jones

ACCU Conference, 26^{th} April 2012

State Machines

- Describe your ‘nightmare’ state machine
- 1000 line file
- Mother of all switch() statements
- 10s of line per case
- Nested switch()es
- Freely calling other code to implement the state
- What is the cyclometric complexity?
- Often an entire thread’s code in one place
- Classic testing problem

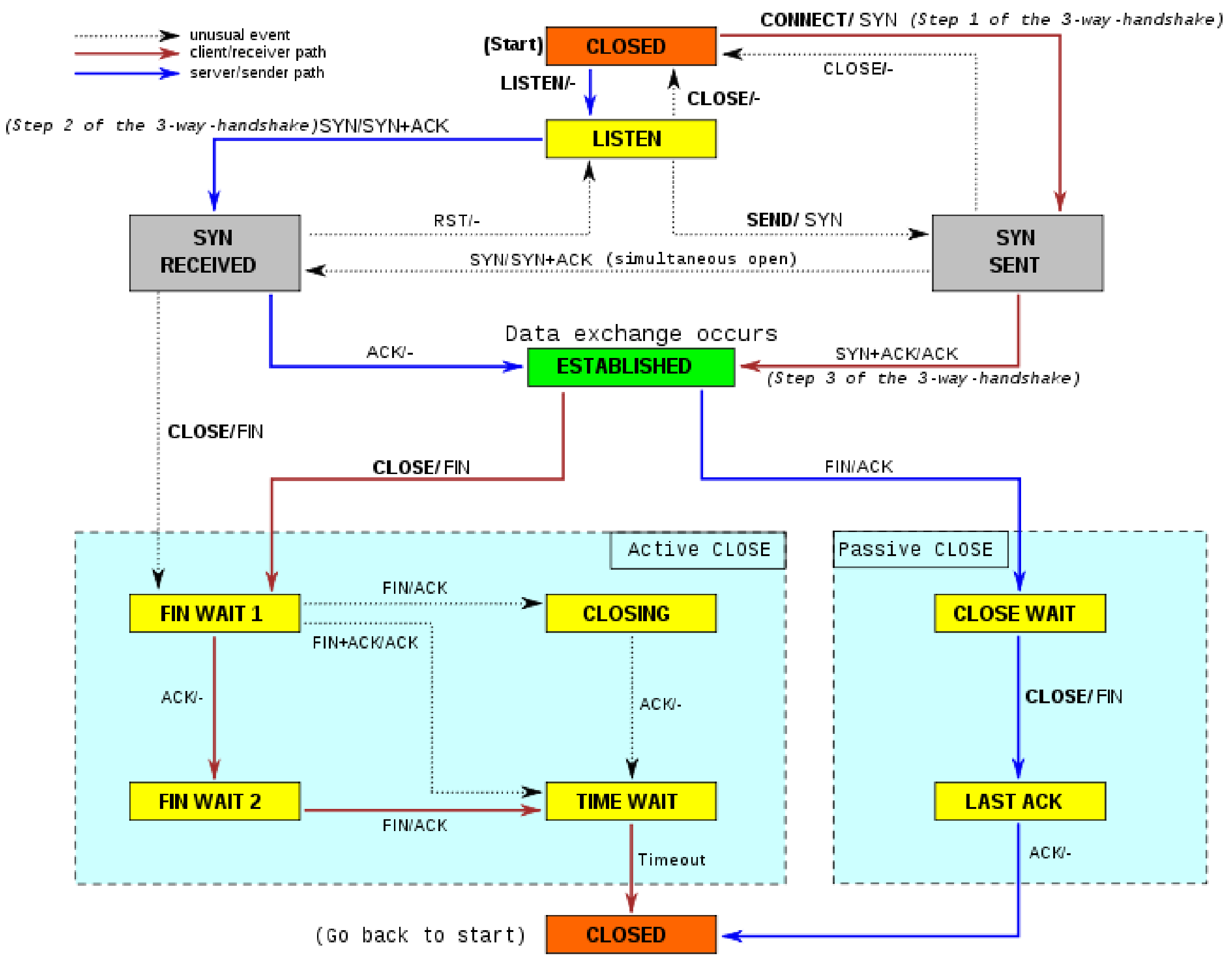


Example

- http://en.wikipedia.org/wiki/Event_driven_finite_state_machine
- Trivially simple
- Still has dependency on “application”

```c
/********************************************************************/
typedef enum {
        RADIO, CD
} STATES;

int readEventFromMessageQueue(void);

/********************************************************************/
int main(void)
  {
  /* Default state is radio */
  int state = RADIO;
  stationNumber=0;
  trackNumber=0;

  /* Infinite loop */
  while( 1 )
    {
    /* Read the next incoming event. Usually this is a blocking functio
    event = readEventFromMessageQueue()

    /* Switch the state and the event to execute the right transition.
    switch( state )
      {
      case RADIO:
        switch(event)
          {
          case mode:
            /* Change the state */
            state = CD;
            break;
          case next:
            /* Increase the station number */
            stationNumber++;
            break;
          }
        break;
      case CD:
        switch(event)
          {
          case mode:
            /* Change the state */
            state = RADIO;
            break;
          case next:
            /* Go to the next track */
            trackNumber++;
            break;
          }
        break;
      }
    }
```

The TCP State Machine



Linux TCP State Machine

- tcp_states.h
- State logic scattered over dozens of large files in /net
- State changes often 'incidental' in other code.
- Completely untestable

```
enum {
    TCP_ESTABLISHED = 1,
    TCP_SYN_SENT,
    TCP_SYN_RECV,
    TCP_FIN_WAIT1,
    TCP_FIN_WAIT2,
    TCP_TIME_WAIT,
    TCP_CLOSE,
    TCP_CLOSE_WAIT,
    TCP_LAST_ACK,
    TCP_LISTEN,
    TCP_CLOSING,

    TCP_MAX_STATES
};
```

The Problem

- Not separating concerns: include code in SM that **implements** the state.
- Seemingly trivial, but introduces dependency(s) on the application
- Mixes state logic with application logic.
- Testing
  - Manual? Run application, stimulate it, observe outcome. Infer state machine operation
  - printf()
  - Stubbing & mocking to get a test suite to build→ tedious
  - Too hard → pressure not to.

So …

How should we go about writing a state machine from scratch so that we can test it easily?

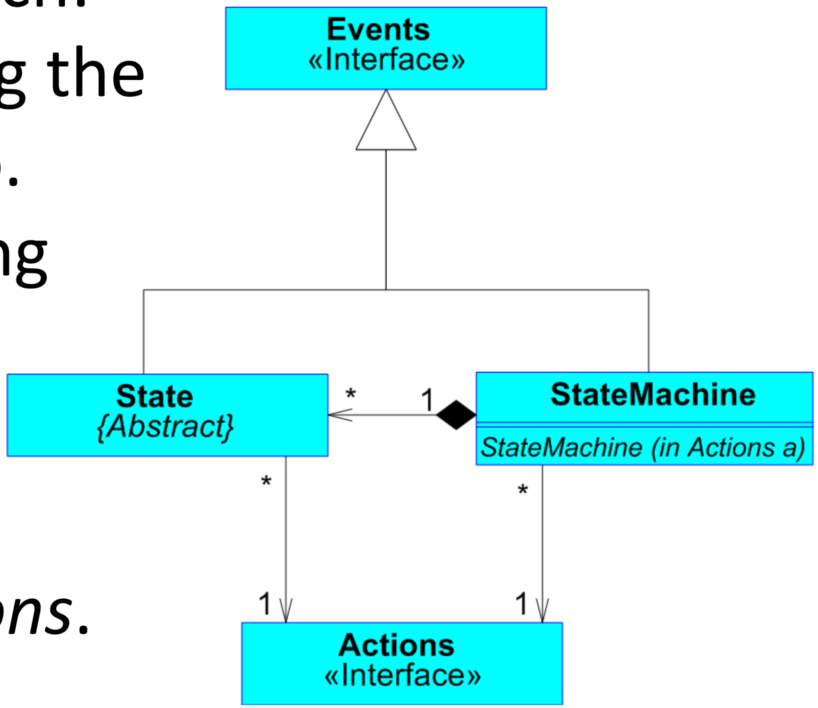
What is a State Machine?

- “Code that manages the state of something, responding to external events, and translating them into actions to be implemented by the system“ ©me.
- Transitions from one state to another in response to events.
- Transitions normally expected to cause actions, but aren't a requirement.
- Details of actions are NOT part of the state machine.

The Premise

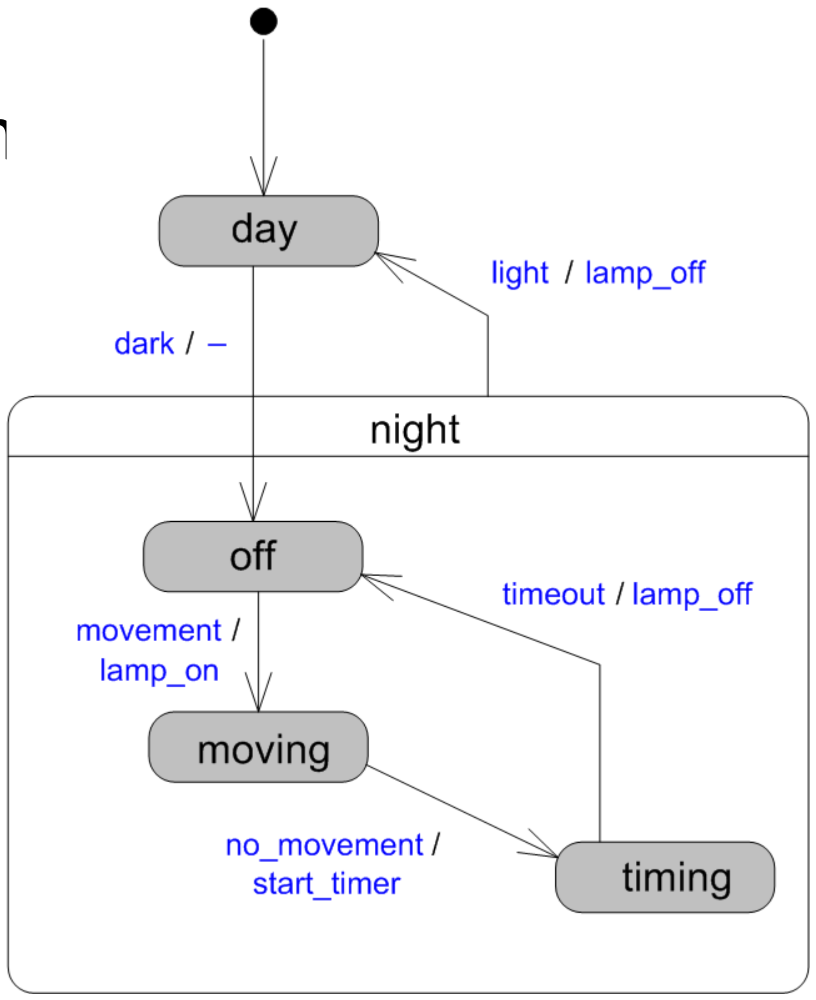
- SM is simply a way of turning events into actions.
- Any further details should be in the application.

- Maps nicely to an OO approach:
- An *events* interface describing the events the SM will respond to.
- An *actions* interface describing the actions the SM will output to the system.
- SM implements *events*.
- Application implements *actions*.



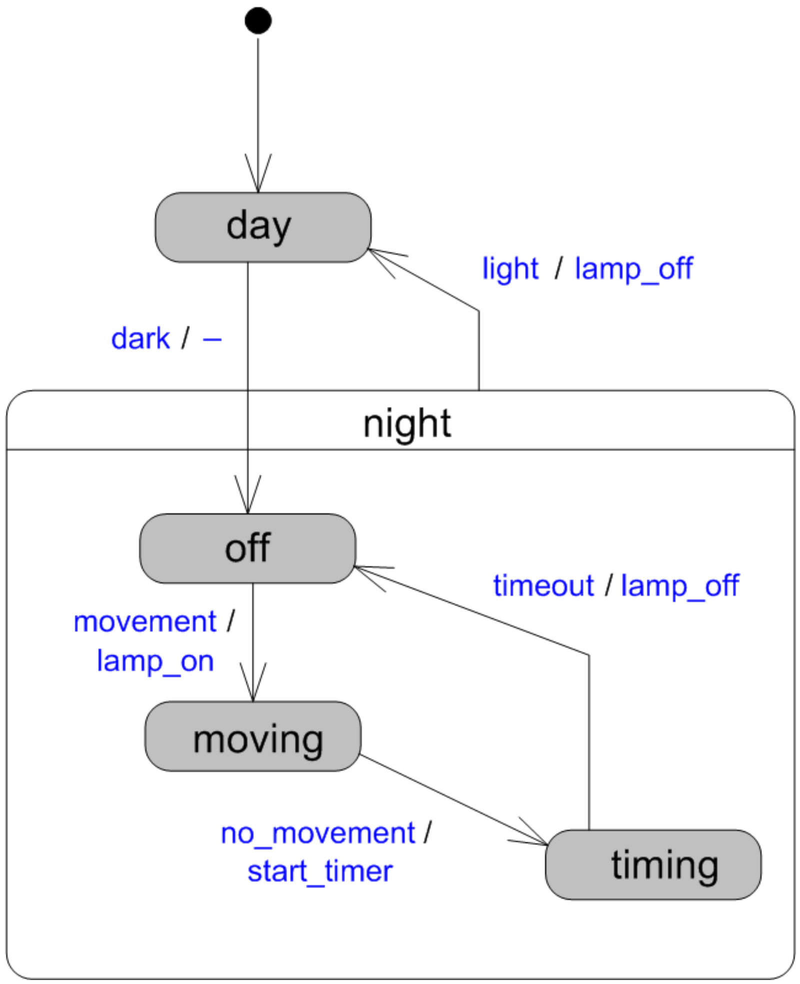
Making A Start

- Start from state diagram
- Identify inputs and outputs:
  - Inputs → events
  - Outputs → actions



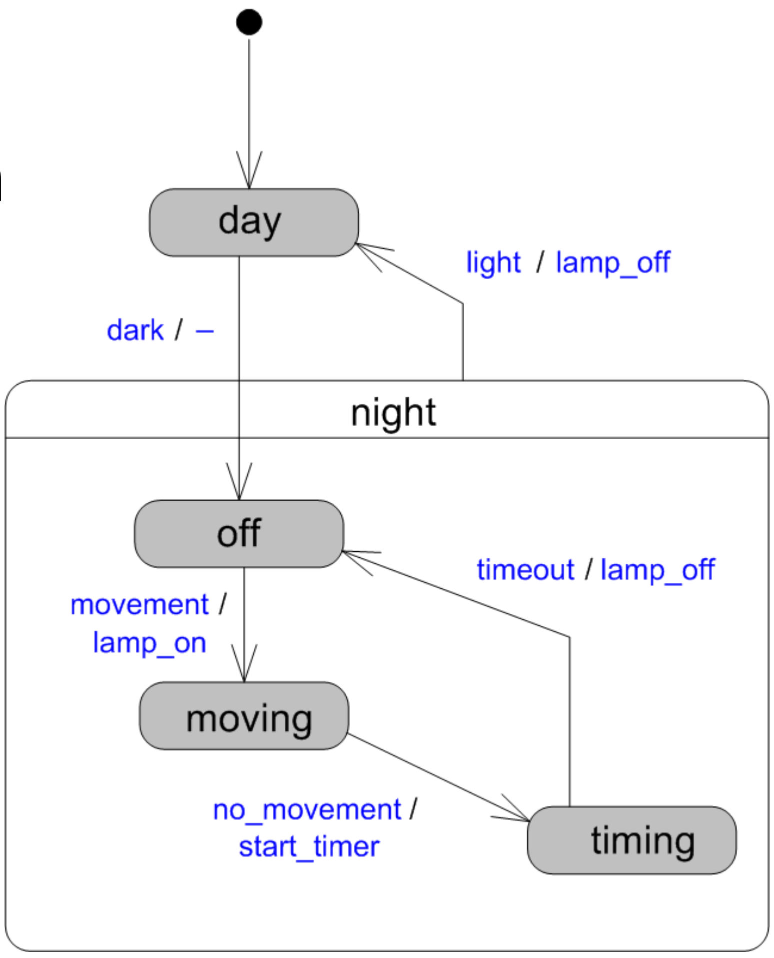
Events and Actions

```
class Events
{
public:

    virtual ~Events() {};

    virtual void Dark (void) = 0;
    virtual void Light (void) = 0;
    virtual void Movement (void) = 0;
    virtual void NoMovement (void) = 0;
    virtual void Timeout (void) = 0;

};

class Actions
{
public:

    virtual ~Actions() {};

    virtual void LampOn (void) = 0;
    virtual void LampOff (void) = 0;
    virtual void StartTimer (void) = 0;
};
```



Writing Some Tests

- Given the events we know the FSM can accept, we can write our first test to fire off an event and expect the action.
- Then another.
- And another.
- Soon we should have a test for every transition on the diagram.
- See worked example later



Writing Some Tests

- To add real world relevance to the work, add use case tests.
- May have implemented the diagram perfectly, but unless we put it through its paces it might not be apparent that the diagram is flawed.
- Mistake in example :-)

Approaches

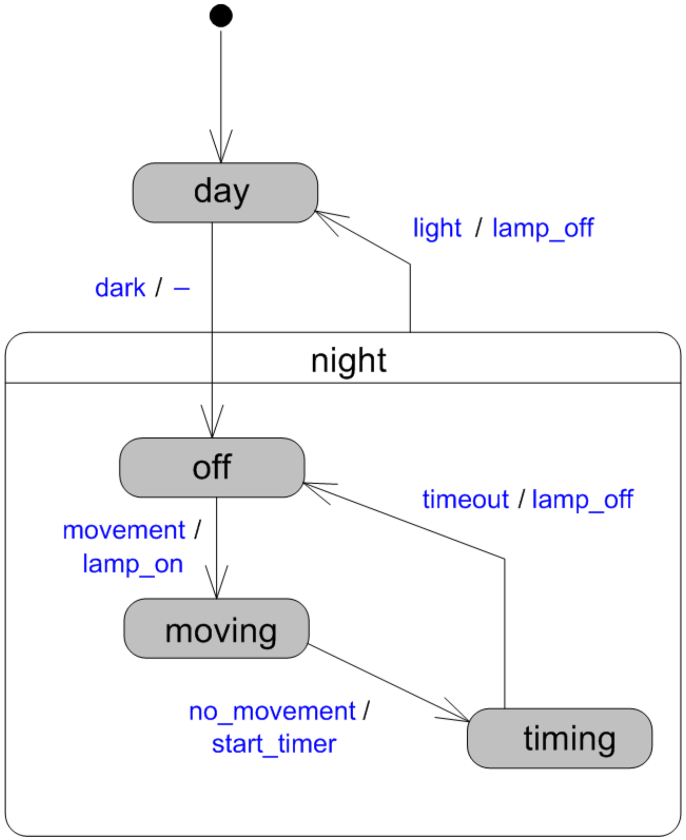
- Language, application, company, project specific.
- Derive test actions from *actions* interface and inject (i.e. Test doubles)
- Derive test implementation of SM to allow test code to sense transitions and actions
- Mock the *actions* interface.
- If non-00, stub action functions

Method: Transition Tests

- Bootstrap the SM, and test harness, into existence:
- Write a few transition tests: look for some expected actions and resulting states
- Get code & test framework into place and settled
- Once the state machine is starting to grow, move to test vectors to simplify the tests, and move faster:
  - [starting state, event(s), end state, expected action(s)]

Example Test Vector

```
struct TestVector
{
    // Type for a pointer to void void member function of StateMachine.
    typedef void (StateMachine::* SMFunctionPointer) (void);

    const char *          testTitle;
    StartingState         startingState;
    SMFunctionPointer     eventFunctionToApply;

    const char *          expectedState;
    const TestActions::ActionType *firstAction;
    const TestActions::ActionType *secondAction;
    const TestActions::ActionType *thirdAction;
};
```



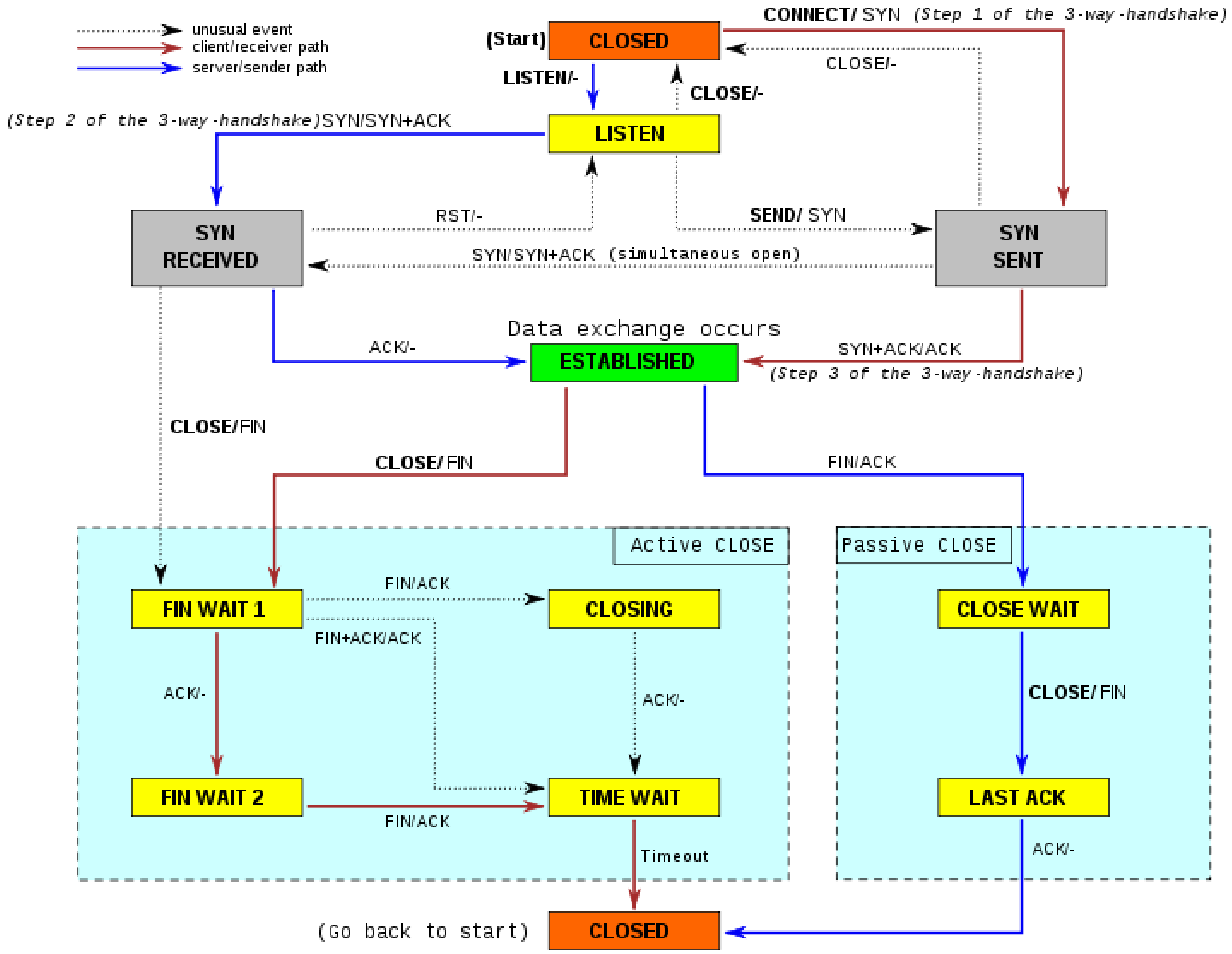
Method: Use Case Tests

- Add use case tests: inject >1 events, driving the SM round multiple transitions.
- Use cases should be good & bad, realistic & unlikely.
- Use cases apply the SM to the real world application: they are acceptance tests (GOOS).
- Each failing acceptance test lead us to TDD the requisite transitions:
  - Add the necessary single transition test vectors, and code the missing transitions / actions.
- Rinse and repeat

Example Test Vectors

```
const TestVector v[] =
{
  { "Day + dark -> off",                START_IN_DAY,    StateMachine::Dark,       "Off",    0, 0, 0 },
  { "Day + light -> no change",         START_IN_DAY,    StateMachine::Light,      "Day",    0, 0, 0 },
  { "Day + movement -> no change",      START_IN_DAY,    StateMachine::Movement,   "Day",    0, 0, 0 },
  { "Day + no_movement -> no change",   START_IN_DAY,    StateMachine::NoMovement, "Day",    0, 0, 0 },
  { "Day + timeout -> no change",       START_IN_DAY,    StateMachine::Timeout,    "Day",    0, 0, 0 },

  { "Off + dark -> no change",          START_IN_OFF,    StateMachine::Dark,       "Off",    0, 0, 0 },
  { "Off + light -> lamp off; day",     START_IN_OFF,    StateMachine::Light,      "Day",    &LampOff, 0, 0 },
  { "Off + movement -> lamp on; moving", START_IN_OFF,   StateMachine::Movement,   "Moving", &LampOn,  0, 0 },
  { "Off + no_movement -> no change",   START_IN_OFF,    StateMachine::NoMovement, "Off",    0, 0, 0 },
  { "Off + timeout -> no change",       START_IN_OFF,    StateMachine::Timeout,    "Off",    0, 0, 0 },

  { "Moving + dark -> no change",       START_IN_MOVING, StateMachine::Dark,       "Moving", 0, 0, 0 },
  { "Moving + light -> lamp off; day",  START_IN_MOVING, StateMachine::Light,      "Day",    &LampOff, 0, 0 },
  { "Moving + movement -> no change",   START_IN_MOVING, StateMachine::Movement,   "Moving", 0, 0, 0 },
  { "Moving + no_movement -> start timer; timing",
                                        START_IN_MOVING, StateMachine::NoMovement, "Timing", &StartTimer, 0, 0 },
  { "Moving + timeout -> no change",    START_IN_MOVING, StateMachine::Timeout,    "Moving", 0, 0, 0 },

  { "Timing + dark -> no change",       START_IN_TIMING, StateMachine::Dark,       "Timing", 0, 0, 0 },
  { "Timing + light -> lamp off; day",  START_IN_TIMING, StateMachine::Light,      "Day",    &LampOff, 0, 0 },
  { "Timing + movement -> moving",      START_IN_TIMING, StateMachine::Movement,   "Moving", 0, 0, 0 },
  { "Timing + no_movement -> no change", START_IN_TIMING, StateMachine::NoMovement, "Timing", &StartTimer, 0, 0 },
  { "Timing + timeout -> lamp off; off", START_IN_TIMING, StateMachine::Timeout,   "Off",    &LampOff, 0, 0 },
};
```

Code Examples



Events and Actions

```
class Events
{
public:
    virtual ~Events() {}

    virtual void connect() = 0;
    virtual void listen() = 0;
    virtual void close() = 0;
    virtual void send() = 0;
    virtual void rst() = 0;
    virtual void ack() = 0;
    virtual void syn() = 0;
    virtual void syn_ack() = 0;
    virtual void fin() = 0;
    virtual void fin_ack() = 0;
    virtual void timeout() = 0;
};
```

```
class Actions
{
public:
    virtual ~Actions() {}

    virtual void syn() = 0;
    virtual void syn_ack() = 0;
    virtual void ack() = 0;
    virtual void fin() = 0;
};
```

One State & Transition

```
class StateMachine : public Events
{
public:
    Actions & actions;

    StateMachine( Actions & a )
        : actions( a ) {}

    virtual void connect() { actions.syn(); }
    virtual void listen() {}
    virtual void close() {}
    virtual void send() {}
    virtual void rst() {}
    virtual void ack() {}
    virtual void syn() {}
    virtual void syn_ack() {}
    virtual void fin() {}
    virtual void fin_ack() {}
    virtual void timeout() {}
};

class TestStateMachine : public StateMachine
{
public:
    TestStateMachine( Actions & a )
        : StateMachine( a ) {}

    const char * getState() { return "SYN SENT"; }
};
```

```
class MockActions : public Actions
{
public:
    MOCK_METHOD0(syn, void());
    MOCK_METHOD0(syn_ack, void());
    MOCK_METHOD0(ack, void());
    MOCK_METHOD0(fin, void());
};

TEST( Transitions,
      In_Closed_Connect_goes_to_SYN_SENT )
{
    MockActions actions;
    TestStateMachine sm( actions );
    sm.connect();
    ASSERT_STREQ( "SYN SENT", sm.getState() );
}

TEST( Transitions,
      In_Closed_Connect_does_syn )
{
    MockActions actions;
    EXPECT_CALL( actions, syn() );
    StateMachine sm( actions );
    sm.connect();
}
```

Two States & Transitions

```
enum TCPState
{
    CLOSED,
    SYN_SENT,
    LISTEN
};

class StateMachine : public Events
{
public:
    Actions & actions;
    TCPState currentState;

    StateMachine( Actions & a )
    : actions( a ),
      currentState( CLOSED )
    {}

    virtual void connect()
    {
        actions.syn();
        currentState = SYN_SENT;
    }
    virtual void listen()
    {
        currentState = LISTEN;
    }
    virtual void close() {}
    virtual void send() {}
    // ...
```

```
// Tests for end states

TEST( Transitions,
        In_CLOSED_connect_goes_to_SYN_SENT ) {
    MockActions actions;
    StateMachine sm( actions );
    sm.connect();
    ASSERT_EQ( SYN_SENT, sm.currentState );
}

TEST( Transitions,
        In_CLOSED_listen_goes_to_LISTEN ) {
    MockActions actions;
    StateMachine sm( actions );
    sm.listen();
    ASSERT_EQ( LISTEN, sm.currentState );
}

// Tests for actions

TEST( Transitions,
        In_CLOSED_connect_does_syn ) {
    MockActions actions;
    EXPECT_CALL( actions, syn() );
    StateMachine sm( actions );
    sm.connect();
}

TEST( Transitions,
        In_CLOSED_listen_does_nothing ) {
    MockActions actions;
    // expect no actions
    StateMachine sm( actions );
    sm.listen();
}
```

Add Third Transition & State

```
enum TCPState
{
    CLOSED,
    SYN_SENT,
    LISTEN,
    SYN_RECEIVED
};

class StateMachine : public Events
{
    // ...
    virtual void connect() { actions.syn();      currentState = SYN_SENT; }
    virtual void listen()  {                     currentState = LISTEN; }
    virtual void close()   {}
    virtual void send()    {}
    virtual void rst()     {}
    virtual void ack()     {}
    virtual void syn()     { actions.syn_ack(); currentState = SYN_RECEIVED; }
    virtual void syn_ack() {}
    virtual void fin()     {}
    virtual void fin_ack() {}
    virtual void timeout() {}
};
```

```
TEST( States, In_LISTEN_syn_goes_to_SYN_RECEIVED )
{
    MockActions actions;
    StateMachine sm( actions );
    sm.currentState = LISTEN;
    sm.syn();
    ASSERT_EQ( SYN_RECEIVED, sm.currentState );
}
```

```
TEST( Transitions, In_LISTEN_syn_does_syn_ack )
{
    MockActions actions;
    EXPECT_CALL( actions, syn_ack() );
    StateMachine sm( actions );
    sm.currentState = LISTEN;
    sm.syn();
}
```

Introduce Test Vectors

```
struct TransitionTestVector
{
    typedef void (StateMachine::* EventMethod)();

    TCPState    startingState;
    EventMethod eventToInject;
    TCPState    expectedEndState;
};

void ExecuteOneTestVector( const TransitionTestVector& v )
{
    MockActions actions;
    StateMachine sm( actions );
    sm.currentState = v.startingState;
    (sm.*(v.eventToInject))();
    ASSERT_EQ( v.expectedEndState, sm.currentState );
}

TEST( States, In_CLOSED_connect_goes_to_SYN_SENT )
{
    const TransitionTestVector v =
        { CLOSED, &StateMachine::connect, SYN_SENT };
    ExecuteOneTestVector( v );
}

TEST( States, In_CLOSED_listen_goes_to_LISTEN )
{
    const TransitionTestVector v =
        { CLOSED, &StateMachine::listen, LISTEN };
    ExecuteOneTestVector( v );
}
```

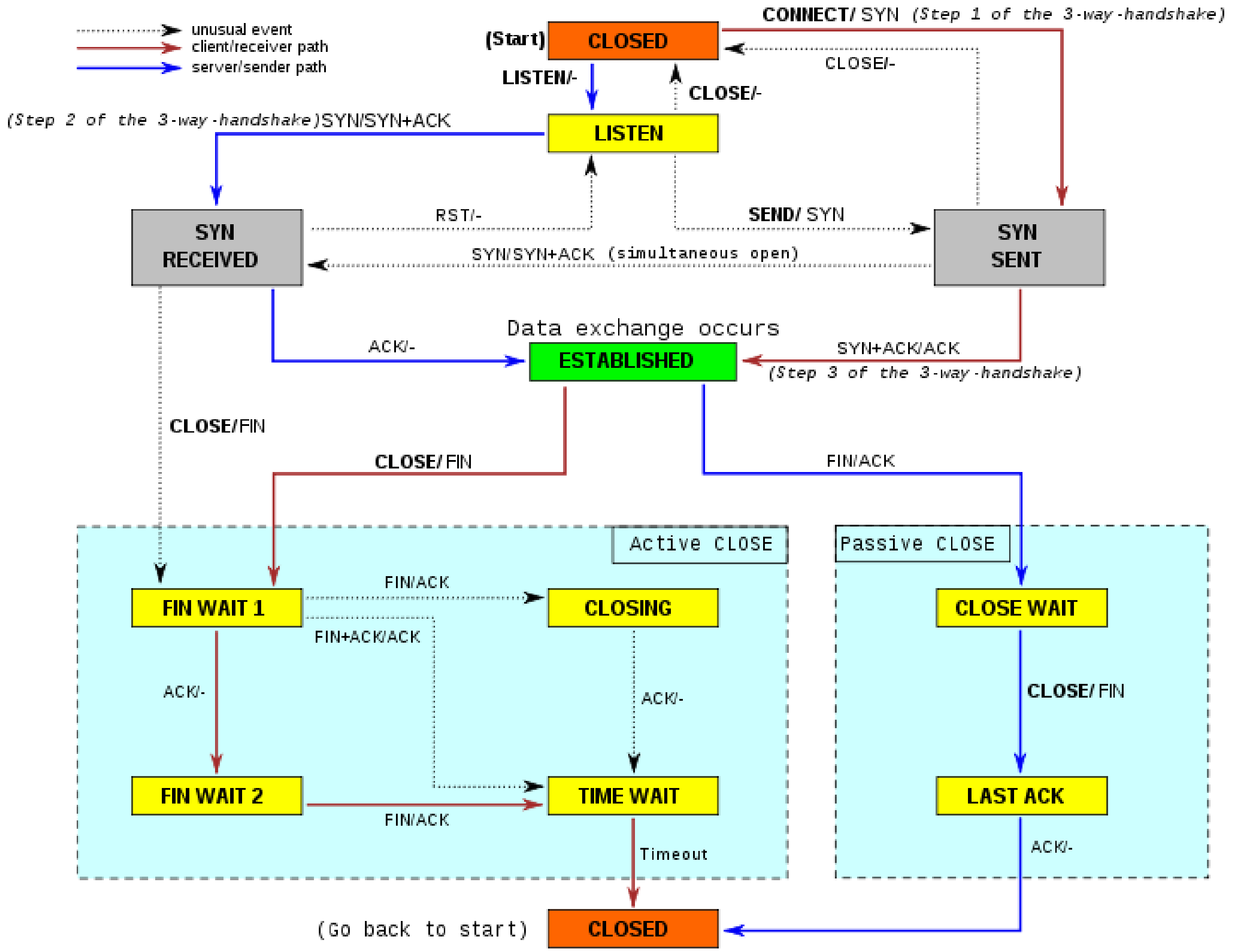
Add Actions To Test Vector

```
struct TransitionTestVector
{
    typedef void (StateMachine::* EventMethod)();

    TCPState    startingState;
    EventMethod eventToInject;
    TCPState    expectedEndState;
    int         expectedCallsToSyn;
    int         expectedCallsToSynAck;
    int         expectedCallsToAck;
    int         expectedCallsToFin;
};

void ExecuteOneTestVector( const TransitionTestVector& v )
{
    MockActions actions;
    EXPECT_CALL( actions, syn()     ).Times( v.expectedCallsToSyn );
    EXPECT_CALL( actions, syn_ack() ).Times( v.expectedCallsToSynAck );
    EXPECT_CALL( actions, ack()     ).Times( v.expectedCallsToAck );
    EXPECT_CALL( actions, fin()     ).Times( v.expectedCallsToFin );

    StateMachine sm( actions );
    sm.currentState = v.startingState;
    (sm.*(v.eventToInject))();
    ASSERT_EQ( v.expectedEndState, sm.currentState );
}

TEST( States, In_CLOSED_connect_goes_to_SYN_SENT )
{
    const TransitionTestVector v = { CLOSED, &StateMachine::connect, SYN_SENT, 1, 0, 0, 0 };
    ExecuteOneTestVector( v );
}
```

Merge Existing Tests Into One

```
TEST( States, Test_all_state_to_state_transitions_and_resulting_actions )
{
    const int SYN = 1;
    const int S_A = 1;

    const TransitionTestVector v[] =
    {
        { CLOSED, &StateMachine::connect, SYN_SENT,     SYN, 0,   0, 0 },
        { CLOSED, &StateMachine::listen,  LISTEN,       0,   0,   0, 0 },
        { LISTEN, &StateMachine::syn,     SYN_RECEIVED, 0,   S_A, 0, 0 }
    };

    for( int i = 0; v[i].expectedCallsToFin != 99; ++i )
    {
        ExecuteOneTestVector( v[i] );
    }
}
```

(Reminder)



unusual event
client/receiver path
server/sender path
(Start)
CLOSED
CONNECT/ SYN (Step 1 of the 3-way-handshake)
CLOSE/-
LISTEN/-
CLOSE/-
(Step 2 of the 3-way-handshake)SYN/SYN+ACK
LISTEN
RST/-
SEND/ SYN
SYN RECEIVED
SYN/SYN+ACK (simultaneous open)
SYN SENT
Data exchange occurs
ACK/-
ESTABLISHED
SYN+ACK/ACK
(Step 3 of the 3-way-handshake)
CLOSE/FIN
CLOSE/ FIN
FIN/ACK
Active CLOSE
Passive CLOSE
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
FIN+ACK/ACK
ACK/-
ACK/-
CLOSE/ FIN
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
Timeout
ACK/-
(Go back to start)
CLOSED

Add More Transitions And States

```
StateMachine( Actions & a ) : actions( a ), currentState( CLOSED ) {}

virtual void connect() { actions.syn();     currentState = SYN_SENT; }
virtual void listen()  {                    currentState = LISTEN; }
virtual void close()   {                    currentState = CLOSED; }
virtual void send()    { actions.syn();     currentState = SYN_SENT; }
virtual void rst()     {                    currentState = LISTEN; }
virtual void ack()     {                    currentState = ESTABLISHED; }
virtual void syn()     { actions.syn_ack(); currentState = SYN_RECEIVED; }
virtual void syn_ack() { actions.ack();     currentState = ESTABLISHED; }
virtual void fin()     {}
virtual void fin_ack() {}
virtual void timeout() {}
```

```
const TransitionTestVector v[] =
{
    { CLOSED,       &StateMachine::connect, SYN_SENT,     SYN, 0,   0,   0 },
    { CLOSED,       &StateMachine::listen,  LISTEN,       0,   0,   0,   0 },

    { LISTEN,       &StateMachine::syn,     SYN_RECEIVED, 0,   S_A, 0,   0 },
    { LISTEN,       &StateMachine::send,    SYN_SENT,     SYN, 0,   0,   0 },
    { LISTEN,       &StateMachine::close,   CLOSED,       0,   0,   0,   0 },

    { SYN_SENT,     &StateMachine::close,   CLOSED,       0,   0,   0,   0 },
    { SYN_SENT,     &StateMachine::syn,     SYN_RECEIVED, 0,   S_A, 0,   0 },
    { SYN_SENT,     &StateMachine::syn_ack, ESTABLISHED,  0,   0,   ACK, 0 },

    { SYN_RECEIVED, &StateMachine::rst,     LISTEN,       0,   0,   0,   0 },
    { SYN_RECEIVED, &StateMachine::ack,     ESTABLISHED,  0,   0,   0,   0 },

    { CLOSED, &StateMachine::connect, CLOSED, 0, 0, 0, 99 }
};
```

Add Self Transitions For Closed

```
virtual void send()
{
    if( currentState != CLOSED )
    {
        actions.syn();
        currentState = SYN_SENT;
    }
}

virtual void rst()
{
    if( currentState != CLOSED )
    {
        currentState = LISTEN;
    }
}

virtual void ack()
{
    if( currentStat
    {
        currentStat
    }
}
```

```
const TransitionTestVector v[] =
{
    { CLOSED,       &StateMachine::connect, SYN_SENT,     SYN, 0,   0,   0 },
    { CLOSED,       &StateMachine::listen,  LISTEN,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::close,   CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::send,    CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::rst,     CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::ack,     CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::syn,     CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::syn_ack, CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::fin,     CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::fin_ack, CLOSED,       0,   0,   0,   0 },
    { CLOSED,       &StateMachine::timeout, CLOSED,       0,   0,   0,   0 },
```

Add Some Use Case Tests

No changes to SM code

```
struct MultiTransitionTestVector
{
    typedef void (StateMachine::* EventMethod)();

    TCPState    startingState;
    EventMethod eventToInject1;
    EventMethod eventToInject2;
    EventMethod eventToInject3;
    TCPState    expectedEndState;
    int         expectedCallsToSyn;
    int         expectedCallsToSynAck;
    int
    int
    int
};
```

```
const MultiTransitionTestVector v[] =
{
    { CLOSED, &StateMachine::connect, &StateMachine::syn,     &StateMachine::ack,
        ESTABLISHED, SYN, S_A, 0,    0 },

    { CLOSED, &StateMachine::connect, &StateMachine::syn_ack, NULL,
        ESTABLISHED, SYN, 0,   ACK, 0 },

    { CLOSED, &StateMachine::connect, &StateMachine::syn,     &StateMachine::rst,
        LISTEN,      SYN, S_A, 0,    0 },

    { CLOSED, &StateMachine::listen,  &StateMachine::syn,     &StateMachine::ack,
        ESTABLISHED, 0,   S_A, 0,    0 },

    { CLOSED, &StateMachine::listen,  &StateMachine::close,   NULL,
        CLOSED,      0,   0,   0,    0 },
```

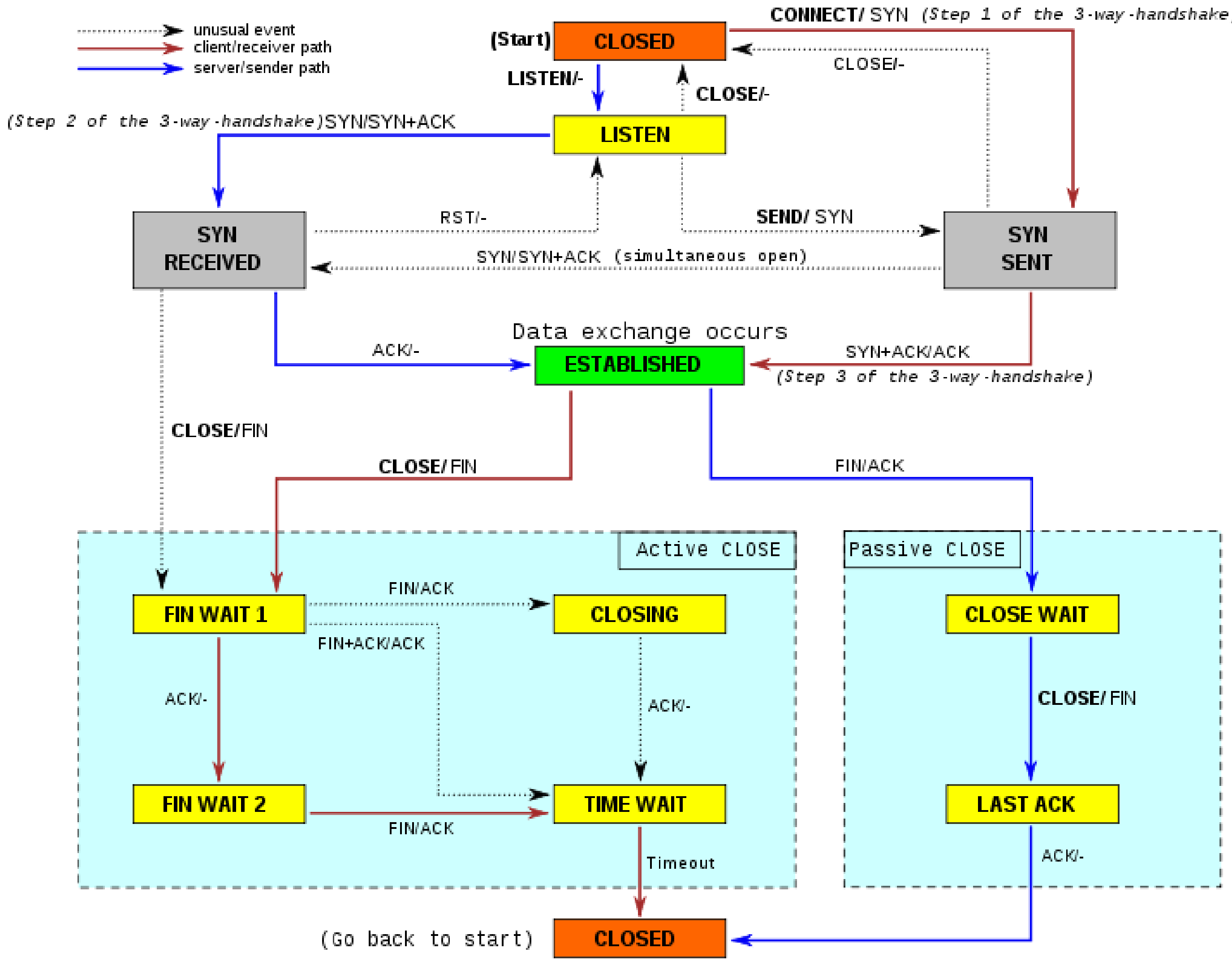
Add Use Case for ESTABLISHED → CLOSED

```
virtual void close()
{
    if( currentState == CLOSE_WAIT )
    {
        actions.fin();
        currentState = LAST_ACK;
    }
    else
    {
        currentState = CLOSED;
    }
}
virtual void ack()
{
    if( currentState == LAST_ACK )
    {
        currentState = CLOSED;
    }
    else if( currentState != CLOSED )
    {
        currentState = ESTABLISHED;
    }
}
virtual void fin()
{
    if( currentState != CLOSED )
    {
        actions.ack();
        currentState = CLOSE_WAIT;
    }
}
```

```
const MultiTransitionTestVector v[] =
{
    {
      ESTABLISHED,
      &StateMachine::fin,
      &StateMachine::close,
      &StateMachine::ack,
      CLOSED,
      0, 0, ACK, FIN
    },
```

```
const TransitionTestVector v[] =
{
  { ESTABLISHED, &StateMachine::fin,
          CLOSE_WAIT, 0, 0, ACK, 0 },

  { CLOSE_WAIT,  &StateMachine::close,
          LAST_ACK,   0, 0, 0, FIN },

  { LAST_ACK,    &StateMachine::ack,
          CLOSED,     0, 0, 0,   0 },
```

Finished Code



Conclusions

- If we are growing the code in response to the tests, what does it look like?
  - Who cares? It works.
  - TDD: it should be pretty simple
  - GoF OO state machine pattern is unlikely to occur spontaneously.
  - The same goes for hierarchical states, even if shown on the state diagram.
  - But once tests are in place, we are free to refactor to this if we want.

Conclusions

- Maintain separation of concerns (app/SM)
- Test for every event in every state to prevent surprises & prove completeness
- Jump between acceptance tests (use case) and unit tests (transitions, actions)
- Use case tests can drive design in absence of state diagram
- Given a state transition diagram we can **test the SM into existence** and **prove it completely**

Questions ?

matthew.jones@garmin.com