CSE 1325: Object-Oriented Programming

Lecture

C++ Embedded Programming with State Diagrams

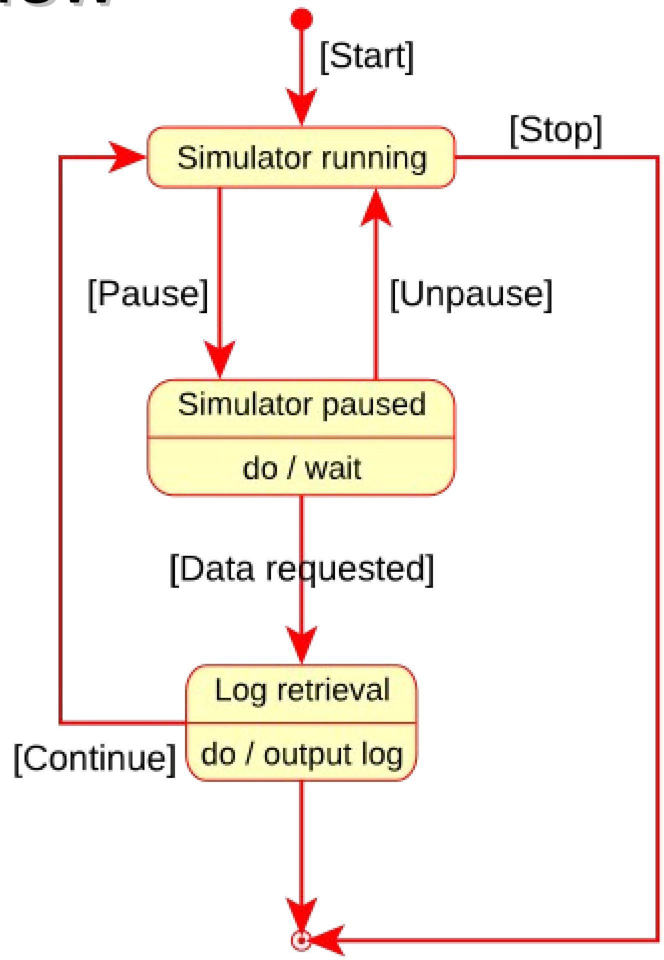
Mr. George F. Rice

george.rice@uta.edu

What do you call 2 crows on a power line?
Attempted murder.

Overview: UML State Diagrams

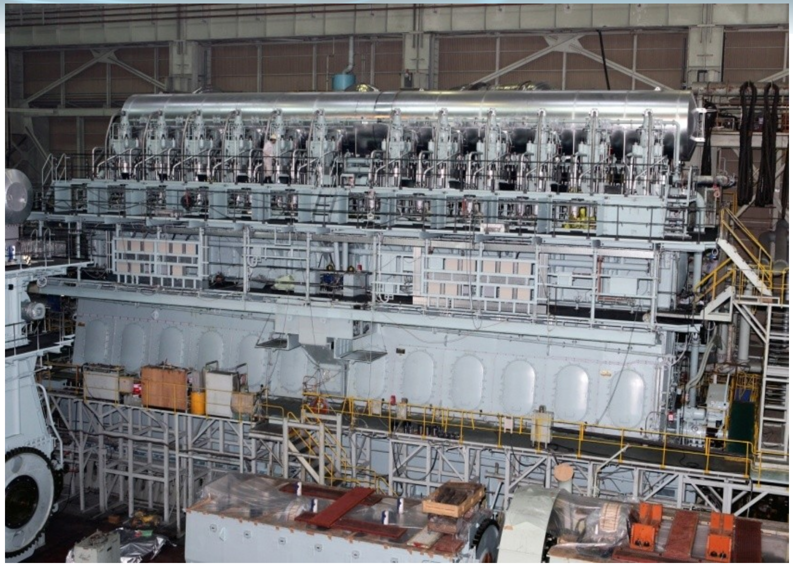
- Embedded Programming Overview
- UML State Diagrams
  - Elements
  - Mealy and Moore
  - Hierarchical State Machines
- State Design Pattern
- C++ Realization
  - Adding Event Handling
  - Dealing with Forward Refs
  - Testing



Embedded Programming

- Embedded programming comes in 2 flavors
  - Hard real-time – A correct answer after the deadline is the wrong answer
  - Soft real-time – A correct answer should be delivered before the deadline most of the time
- The software runs on special hardware
  - Hardware issues must be explicitly handled
  - Portability to next-gen hardware must be pre-planned
  - This is sometimes the bulk of the code

Embedded Systems



- Computers used as part of a larger system
  - That usually don't look like a computer
  - That usually control physical devices
- Often reliability is critical
  - "Critical" as in "if the system fails someone might die"
- Often resources (memory, processor capacity) are limited
- Often real-time response is important – or essential

Examples of Embedded Systems

- Assembly line quality monitors
- Bar code readers
- Bread machines
- Cameras
- Car assembly robots
- Cell phones
- Centrifuge controllers
- CD players
- Disk drive controllers
- “Smart card” processors
- Fuel injector controls
- Medical equipment monitors
- PDAs
- Printer controllers
- Sound systems
- Rice cookers
- Telephone switches
- Water pump controllers
- Welding machines
- Windmills
- Wrist watches

Who Works on Embedded Systems?

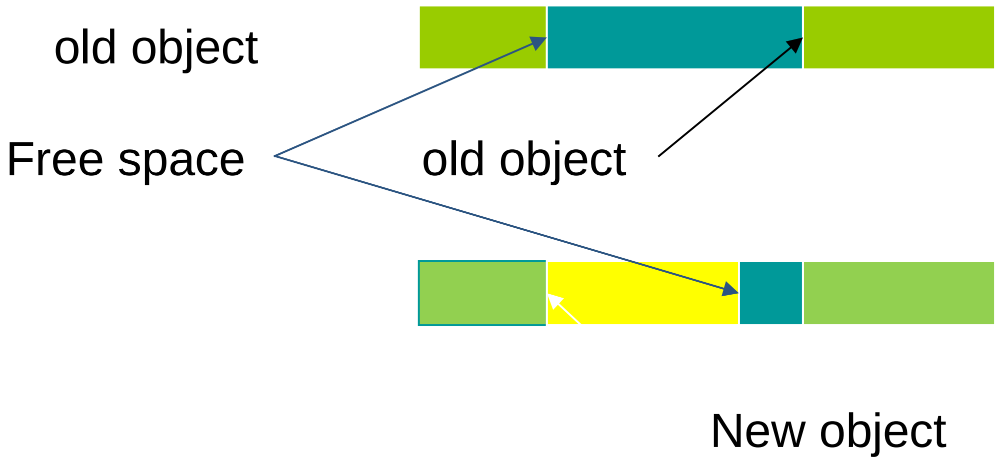
- Computer Scientists
  - Unless you only do web and client-server
- Computer Engineers
  - Hardware and software are complementary aspects of the overall system
  - The hardware must support the software <u>well</u>
- Electrical Engineers
  - You must be able to talk to the computer guys
  - Your concerns are also complementary – RF, power, feedback control systems, etc.

Hard Real-Time Uses a (Large) Subset of C++

- Execution time must be highly predictable
  - Memory allocation (new) times are highly variable
    - Depends on how fragmented heap is currently
  - Exception times are difficult to pin down
    - Execution path moves non-linearly up the call stack
  - System libraries must be included with caution
    - A single include can drag in a bloating of code
  - State machines often model real-world problems well – if the implementation is efficient and predictable
- Measure relentlessly and know your environment!

The Trouble with New and Delete

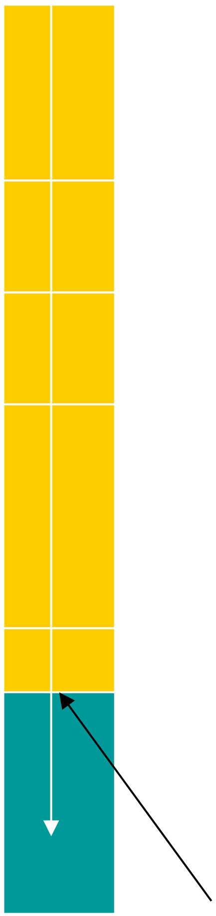
- C++ code refers directly to memory
  - Once allocated, an object cannot be moved (or can it?)



- Allocation delays
  - The effort needed to find a new free chunk of memory of a given size depends on what has already been allocated
- Fragmentation
  - If you have a “hole” (free space) of size N and you allocate an object of size M where M<N in it, you now have a fragment of size N-M to deal with
  - After a while, such fragments constitute much of the memory

The Solution is Preallocated Structures

- Global objects
  - Allocated during startup
  - Fixed memory size
- Stacks
  - Grow and shrink from the top only
    - No fragmentation, constant time
- Pools of fixed size objects
  - Allocate individually in any order
    - No fragmentation, constant time



Simple Pool Example

```
#include <array>

template<class T, int N>
class Pool {
  public:
    // get a T from the pool; return 0 if no free Ts
    T* get() {
        for (auto& t : items) {
            if (t.free) {t.free = false; return &t.value;}
        }
        return nullptr;
    }
    // return a T given out by get() to the pool
    void free(T* p) {
        for (auto& t : items) {
            if (p == &t.value) {t.free = true; break;}
        }
    }
  protected:
    class Item {
      public:
        Item() : free{true} { }
        T value;    // The value held in each pool slot
        bool free;  // True if this pool slot is unused
    };
    std::array<Item, N> items;
};
```

Testable_pool Derived Template

- Derive a template from Pool that can also display its contents to STDOUT

```
#include <iostream>
#include <iomanip>
#include “pool.h”

// CONSTRAINT: class T must overload operator<<

template <class T, int N>
class Testable_pool : public Pool<T, N> {
  public:
    // For demo only - print the contents of the pool
    void show_pool() {
        std::cout << std::hex << std::setw(2);
        for (auto& t : this->items) {
           if (t.free) std::cout << "-- ";
           else std::cout << t.value << ' ';
        }
        std::cout << std::endl;
    }
};
```

How to derive a template from a base template!

“this->” is required to access protected base class members from a derived template

Testing Pool

```
#include “testable_pool.h”

int main() {
    typedef int seg7_led;  // Manage a pool of 10 2-digit LEDs
    Testable_pool<seg7_led, 10> p;
    p.show_pool();

    int* i1 = p.get();    *i1 = 42;
    int* i2 = p.get();    *i2 = 17;
    int* i3 = p.get();    *i3 = 0xFF;
    int* i4 = p.get();    *i4 = 0xCC;
    p.show_pool();

    p.free(i1);
    p.free(i3);
    p.show_pool();

    int* i5 = p.get();    *i5 = 0x4F;
    p.show_pool();
}
```

```
ricegf@pluto:~/dev/cpp/201801/23$ make pool
g++ -std=c++14    pool.cpp   -o pool
ricegf@pluto:~/dev/cpp/201801/23$ ./pool
-- -- -- -- -- -- -- -- -- --
2a 11 ff cc -- -- -- -- -- --
-- 11 -- cc -- -- -- -- -- --
4f 11 -- cc -- -- -- -- -- --
ricegf@pluto:~/dev/cpp/201801/23$
```

Stack Example

```
#include <array>

template<class T, int N>
class Stack {
  public:
    Stack() : next{stack.begin()} {stack.fill(T{});}

    // get a T from the stack; return 0 if no free Ts
    T* get() {
        if (next != stack.end()) return &(*next++);
        else return nullptr;
    }

    // return the most recent T given out by get() to the stack
    void free() {
        if (next != stack.begin()) {--next; *next = T{};}
    }

  protected:

    std::array<T, N> stack;
    typename std::array<T, N>::iterator next;
};
```

Design trade-off per project:
Initialize at construction?
Clear on free? OR
Ignore existing data
(not shown).

“typename” is required by g++ (though not by clang) to remove ambiguity in the template code that might result in errors for certain types of T at instantiation time.

Testable_stack Derived Template

- Derive a template from Stack that can also display its contents to STDOUT

```
#include <iostream>
#include <iomanip>
#include “stack.h”

// CONSTRAINT: class T must overload operator<<

template <class T, int N>
class Testable_stack : public Stack<T, N> {
  public:
    // For demo only - print the contents of the stack
    void show_stack() {
        int stack_elements = this->stack.size();
        std::cout << std::hex;
        for (auto t = this->stack.begin(); t != this->next; ++t)
            {std::cout << std::setw(2) << *t << ' '; --stack_elements;}
        while(stack_elements--) std::cout << "-- ";
        std::cout << std::endl;
    }
};
```

Testing Stack

```
#include "testable_stack.h"

int main() {
    typedef int seg7_led;  // Manage a stack of 10 2-digit LEDs
    Testable_stack<seg7_led, 10> p;
    p.show_stack();

    int* i1 = p.get();    *i1 = 42;
    int* i2 = p.get();    *i2 = 17;
    int* i3 = p.get();    *i3 = 0xFF;
    int* i4 = p.get();    *i4 = 0xCC;
    p.show_stack();

    p.free();
    p.free();
    p.show_stack();

    int* i5 = p.get();    *i5 = 0x4F;
    p.show_stack();
}
```

```
ricegf@pluto:~/dev/cpp/201801/23$ make stack
g++ -std=c++14    stack.cpp   -o stack
ricegf@pluto:~/dev/cpp/201801/23$ ./stack
-- -- -- -- -- -- -- -- -- --
2a 11 ff cc -- -- -- -- -- --
2a 11 -- -- -- -- -- -- -- --
2a 11 4f -- -- -- -- -- -- --
ricegf@pluto:~/dev/cpp/201801/23$
```

Templates Rock for Real-Time!

- Vector and Array classes (for example) are templates
  - Vector is not usually suitable for real-time!
  - Array, however, is *great* – low overhead, predictable
- Operations are inline
  - No run-time overhead
- No memory consumed for unused operations
  - Memory is often in short supply
- Reminiscent of the preprocessor
  - Now with class(es)!

Bit Representation in C++

- unsigned char uc; // 8 bits
- unsigned short us; // typically 16 bits
- unsigned int ui;// typically 16 bits or 32 bits
  - // (check before using)
  - // many embedded systems have 16-bit ints
- unsigned long int ul;// typically 32 bits or 64 bits
- std::vector<bool> vb(93); // 93 bits – C++ 14 and later only
  - true/false auto-converts to/from 1/0
  - Use only if you really need more than 32 bits with little memory
- std::bitset<16> bs{0xFAB}; // 16 bits, init as 0000111110101011
  - Similar as std::vector above, but clearer and more capable
  - Typically most efficient for multiples of sizeof(int)

std::bitset<*int* number_of_bits>

- Bitset is an “array of bits”
  - Thus, 8x more memory efficient than char
  - Size fixed when instanced (see vector<bool> for dynamic resizing – with restrictions)
  - Access a <u>bit</u> via operator[ ] or test(), e.g., foo[3]
- Can be constructed from int or binary strings
  - And can be read and written to streams in binary

Bitset Example

```
#include <iostream>
#include <bitset>

int main()
{
    // Create and manipulate a 16-bit bitset
    std::bitset<16> bitset1;
    std::cout << "Default 16-bit constructor: " << bitset1 << std::endl;
    bitset1.set();  // Sets ALL bits to 1 – see also reset(), flip()
    std::cout << "After set(): " << bitset1 << std::endl;
    for (int i=0; i<bitset1.size(); i+= 2) bitset1.reset(i);
    std::cout << "After resetting even bits: " << bitset1 << std::endl;
    for (int i=0; i<bitset1.size(); i+= 3) { // bitset1.test(i) would also work below
        std::cout << "Bit " << i << " is " << (bitset1[i] ? "set" : "reset") << std::endl;
    }
    std::cout << bitset1.count() << " bits are now set" << std::endl << std::endl;

    // Create a 32-bit bitset with explicit initialization
    std::bitset<32> bitset2{0xFEEDFACE};
    std::cout << "Initialized 32-bit constructor:" << bitset2 << std::endl;

    // Create a 13-bit bitset with string initialization
    std::bitset<13> bitset3(std::string("1001110100101"));
    std::cout << "String-initialized 13-bit constructor:" << bitset3 << std::endl;

}
```

Bitset Example

```
#include <iostream>
#include <bitset>

int main()
{
    // Create and manipulate a 16-bit bitset
    std::bitset<16> bitset1;
    std::cout << "Default 16-bit constructor: " << bitset1 << std::endl;
    bitset1.set(
    std::cout <<
    for (int i=0
    std::cout <<
    for (int i=0
        std::cou
    }
    std::cout <<

    // Create a
    std::bitset<
    std::cout <<

    // Create a
    std::bitset<
    std::cout <<

}
```

```
ricegf@nix:~/Documents/UTA/CSE1325/201801/dev/23$ g++ --std=c++14 bitset.cpp
ricegf@nix:~/Documents/UTA/CSE1325/201801/dev/23$ ./a.out
Default 16-bit constructor: 0000000000000000
After set(): 1111111111111111
After resetting even bits: 1010101010101010
Bit 0 is reset
Bit 3 is set
Bit 6 is reset
Bit 9 is set
Bit 12 is reset
Bit 15 is set
8 bits are now set

Initialized 32-bit constructor:11111101110110111110101100111O
String-initialized 13-bit constructor:1001110100101
ricegf@nix:~/Documents/UTA/CSE1325/201801/dev/23$
```

Bit Manipulation

a:	1	0	1	0	1	0	1	0	0xaa
b:	0	0	0	0	1	1	1	1	0x0f
a&b:	0	0	0	0	1	0	1	0	0x0a
a|b:	1	0	1	0	1	1	1	1	0xaf
a^b:	1	0	1	0	0	1	0	1	0xa5
a<<1:	0	1	0	1	0	1	0	0	0x54
b>>2:	0	0	0	0	0	0	1	1	0x03
~b:	1	1	1	1	0	0	0	0	0xf0

- & and
  (often checks a bit)
- | inclusive or
  (often sets a bit)
- ^ exclusive or
  (often flips a bit;
  also for graphics, crypto)
- << left shift
- >> right shift
- ~ one’s complement

Know Your Environment

- The disassembler is the embedded programmer’s best friend
  - What code did that C++ feature generate?
  - *Why* did it generate that code?
  - What alternate features generate “better” code?
- The system library source code is for bedtime reading
  - Which classes use unacceptable features?
  - Which interdependencies exist and how do they affect your memory and latency?
- The Real-Time Operating System (RTOS) is a key foundation
  - Especially the scheduler – know it well!
- The “non-hosted environment” (remote debugger) drives your productivity during integration to a surprising degree
  - Many “bugs” are *hardware* bugs, unlike in IT!

Failing Hardware

- Hardware failures are <u>much</u> more common in embedded than in IT
  - Power surges and sags
  - Connectors vibrate loose
  - Physical damage
  - Environmental extremes
- The system may be remote when it fails
  - Like, on Mars... or in the Mariana Trench

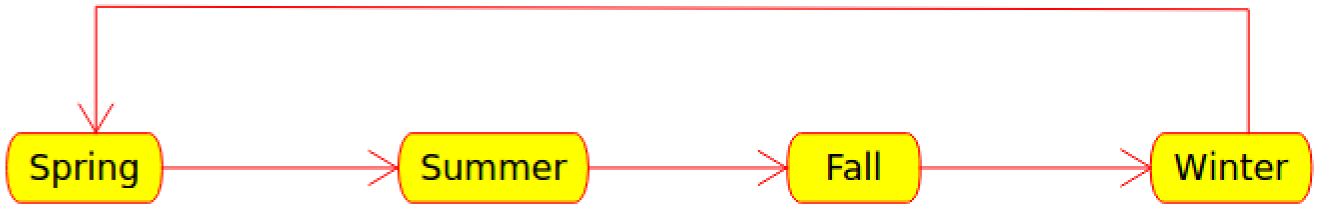
Beagle 2

- Strategies
  - Self-check
    - Heartbeat / Reset
    - Shuttle - 4+1
  - Redundancy
    - Fly by wire
    - Shuttle - 4+1
  - Degraded Modes
    - F-16 – 8 CPUs, 256 modes
  - Debug-only bus
    - I^2C (Inter-IC), step pins

Modeling the World in States

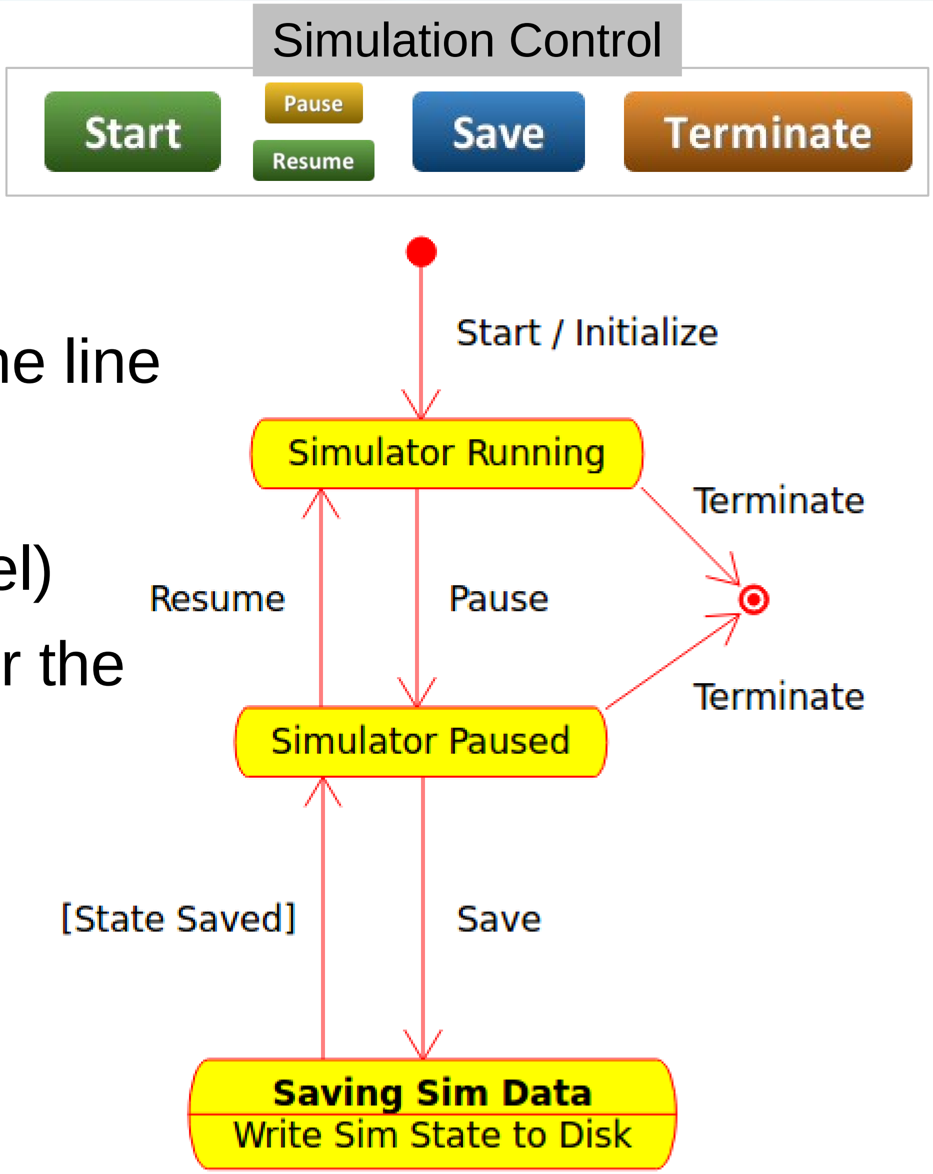
- Many embedded (and IT) systems exhibit state-like behavior
- A **State** is the cumulative value of all relevant stored information to which a system or subsystem has access.

  - The output of a stateful system is completely determined by its <u>current state</u> and its <u>current inputs</u>
  - A **State Diagram** documents the states, permissible transitions, and activities of a system

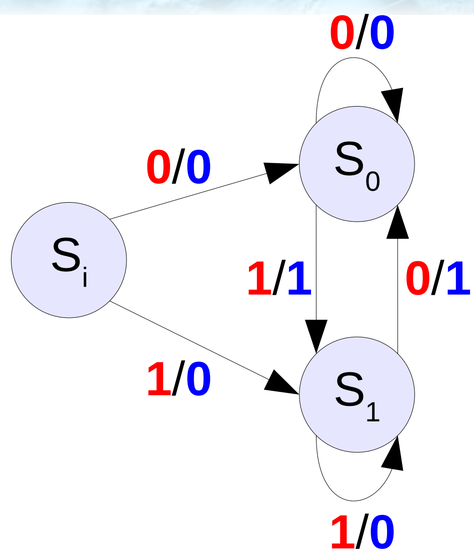


Basic UML State Diagram Elements

- Initial state (filled circle)
- State (rounded rectangle)
  - Name of state above the line
  - Optional activity / output below the line
- Transition (arrow line)
  - Event causing the transition (label)
  - Guard (bool) that must be true for the transition to occur (inside square brackets)
  - Activity / output during transition (after the /)
- Final state (target)

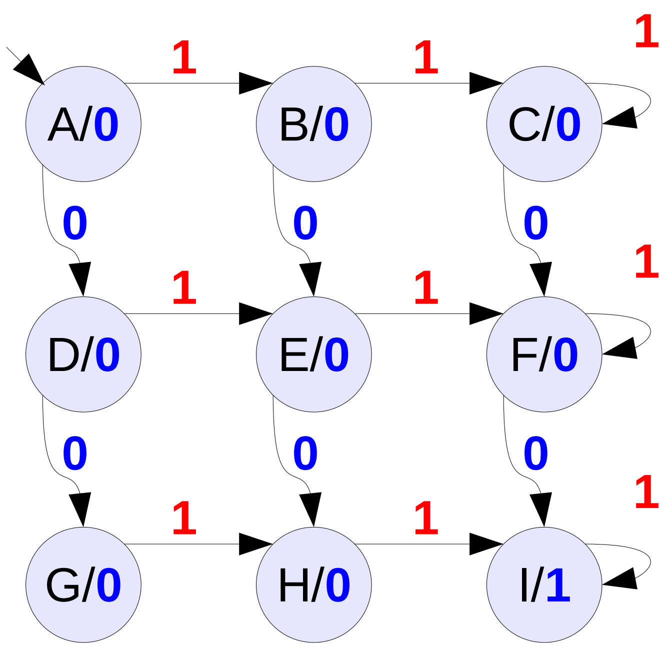


(Non-UML)
Mealy vs Moore Diagrams



A **Mealy** state machine defines outputs based on current state <u>and</u> current inputs

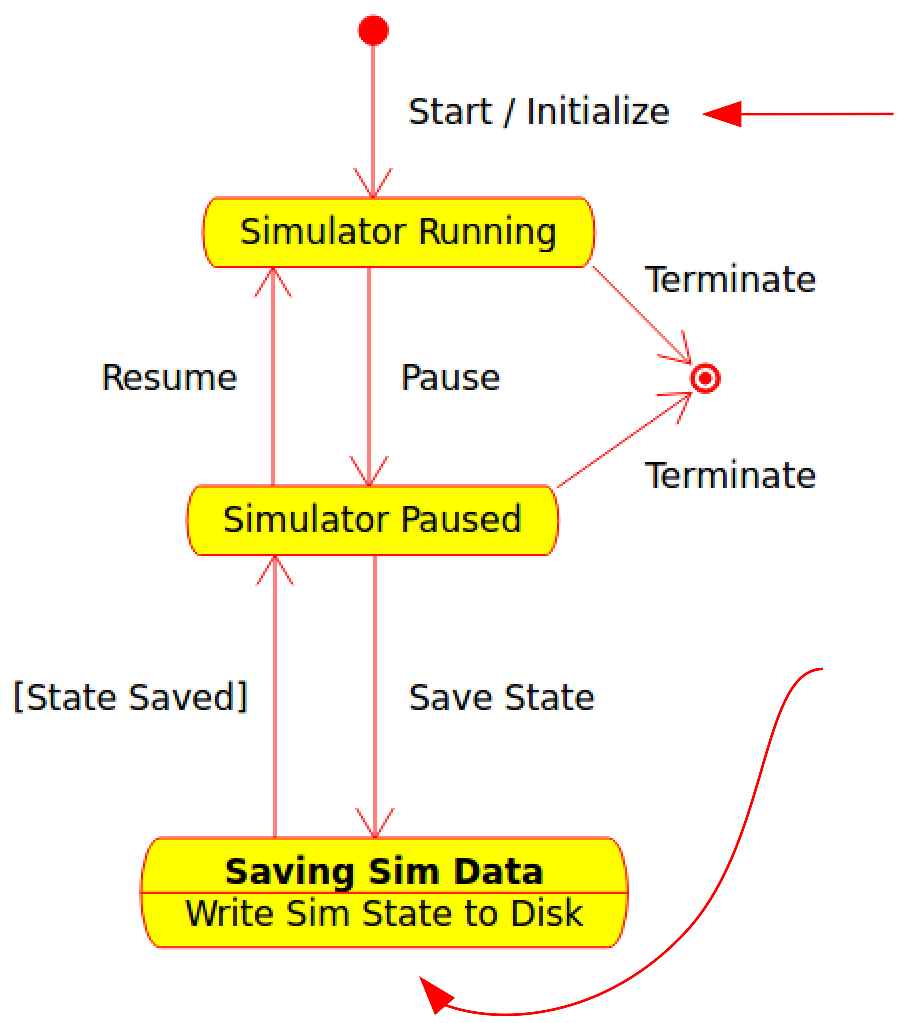
- Outputs are defined on *transitions*
- Outputs follow inputs immediately
- Often require fewer states



A **Moore** state machine defines outputs based solely on the current state

- Outputs are defined in *states*
- Outputs change only on clock edges
- Safer for hardware realization due to greater immunity from race conditions and line noise

UML State Diagrams are Simultaneously Mealy and Moore



A **Mealy** state machine defines outputs based on current state <u>and</u> current inputs

- Outputs are defined on *transitions*
- Outputs follow inputs immediately
- Often require fewer states

A **Moore** state machine defines outputs based solely on the current state

- Outputs are defined in *states*
- Outputs change only on clock edges
- Safer for hardware realization due to greater immunity from race conditions and line noise

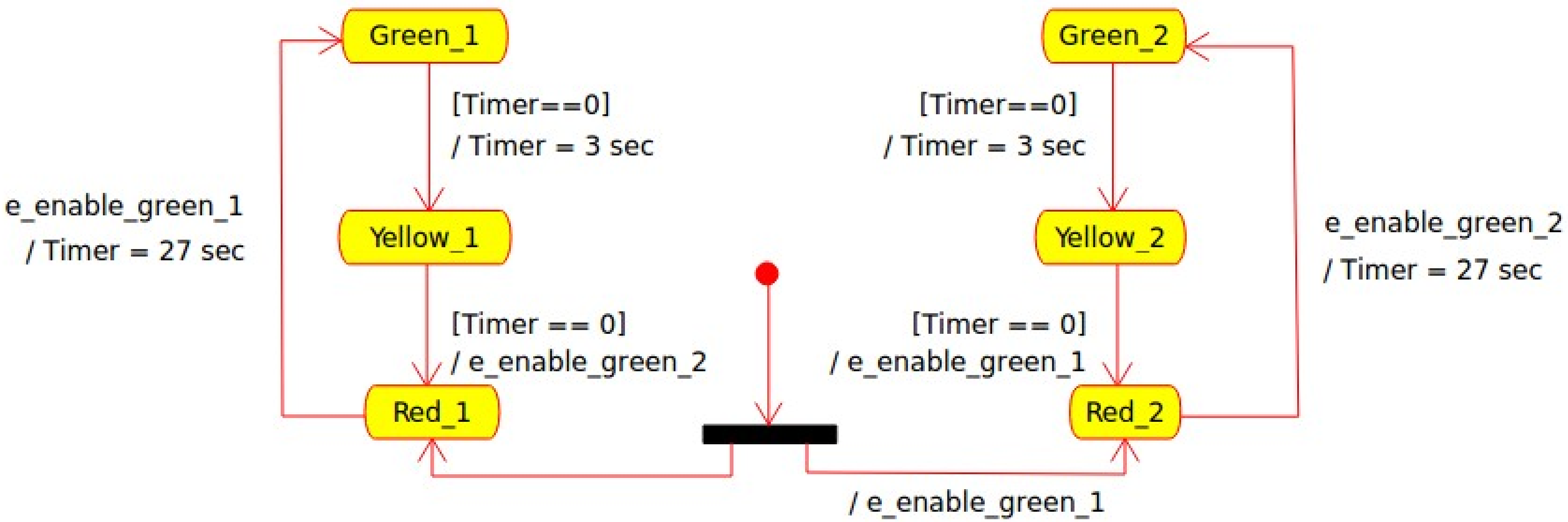
Simple State Logic

```
enum class State {PENDING, FILLED, PAID, COMPLETE, DISCARDED};

Order::Order() : _order_number{_next_order_number++}, _state{PENDING} { }

void Order::fill() { // For event FILL
  // States for which event FILL is invalid
  if (state == State::FILLED) throw std::runtime_error{"Try to fill FILLED order"};
  if (state == State::COMPLETE) throw std::runtime_error{"Try to fill COMPLETE order"};
  if (state == State::DISCARDED) throw std::runtime_error{"Try to fill DISCARDED order"};

  // States for which event FILL is valid
  if (state == PENDING) {state = FILLED;}
  else if (state == PAID) {state = COMPLETE;}

  // Detecting an invalid state for Order
  else {throw std::runtime_error{"Invalid Order state + std::to_string(state)};}

}
```

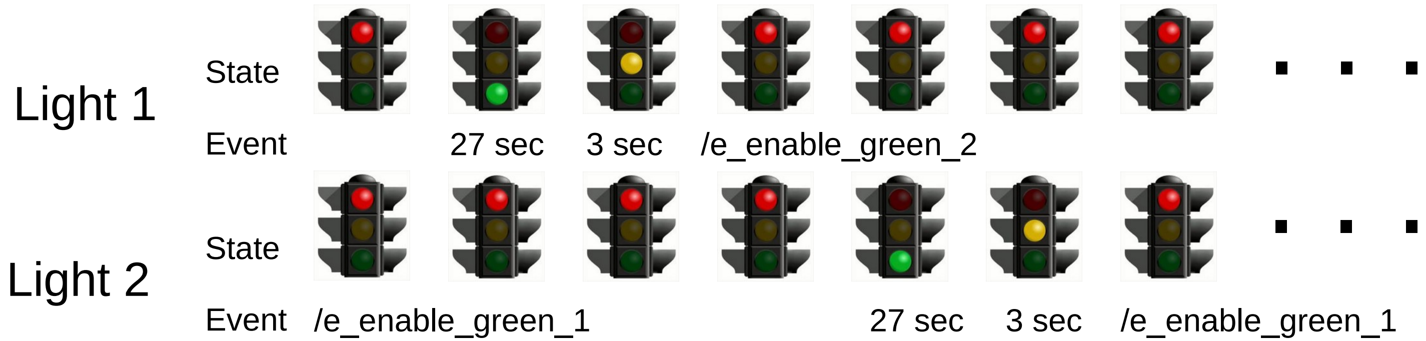
State machines can be built from simple logic, with events as methods that implement behaviors on those events as well as resulting state changes.

In a multi-threading environment, be certain to protect these state changes with a mutex!)

Object-Oriented State Logic: Modeling a Traffic Light

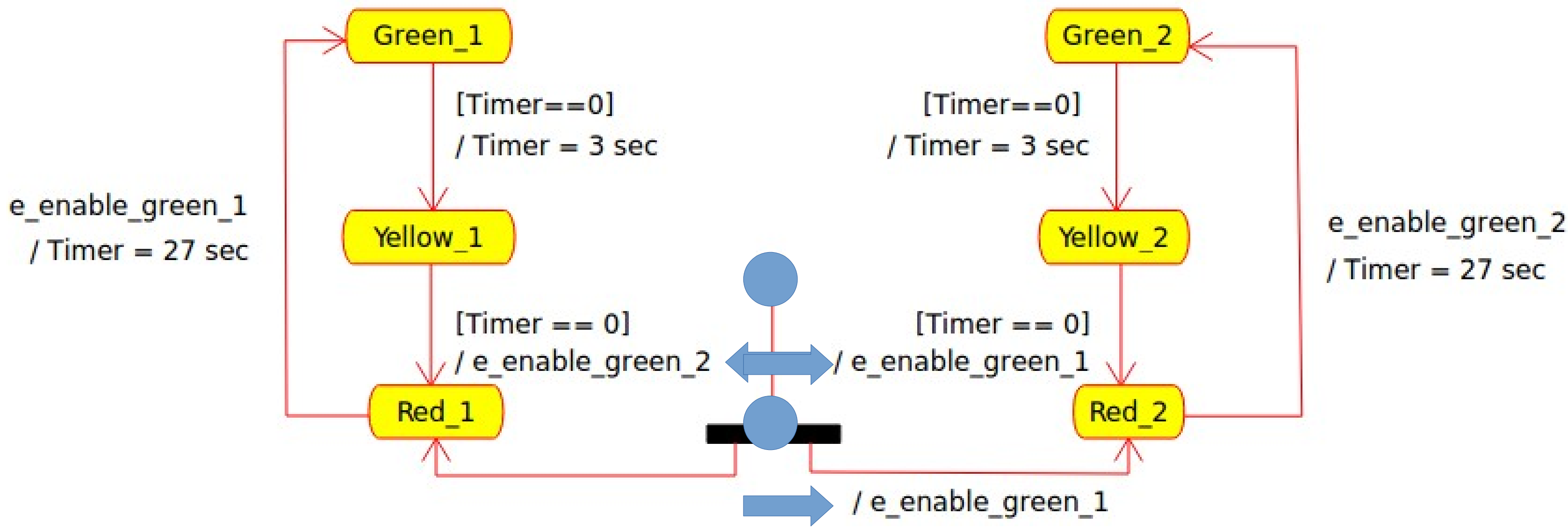


- Each direction is controlled by a separate state machine.
- The state of each machine selects the lights in that direction.
- The state machines share a timer and communicate via events.

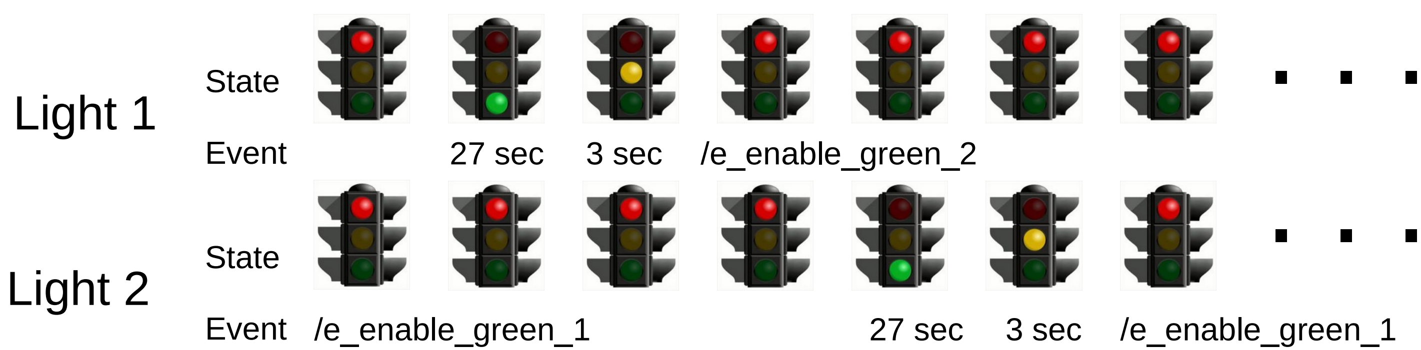


Images in the public domain from
http://www.freeclipartnow.com/transportation/traffic-lights/

Modeling a Traffic Light



- Each direction is controlled by a separate state machine.
- The state of each machine selects the lights in that direction.
- The state machines share a timer and communicate via events.



Images in the public domain from
http://www.freeclipartnow.com/transportation/traffic-lights/

States

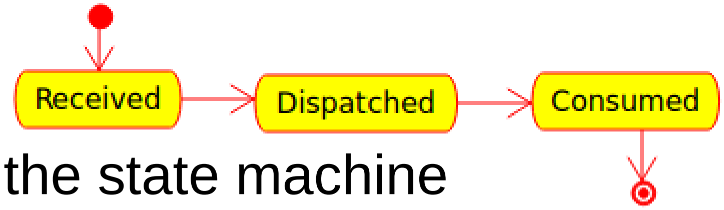
- Each state models a distinct aggregation of *relevant* memory and hardware in the system
  - Most of memory isn't relevant to the state machine
- UML states are *extended* states, allowed to contain complex data structures to complement the state itself
  - These should be used minimally to manage complexity of the state machine
  - Extended state variables raise the need for guards

Guards

- A **guard** is a Boolean expression that enables a state transition when true and disables it when false, e.g., [power == on]
  - The Boolean expression can reference any available object, but typically depends on related state machines, event parameters, or simple external signals
  - Guards should be minimized, as overuse can make a state machine and its associated code “brittle” and difficult to debug

Events

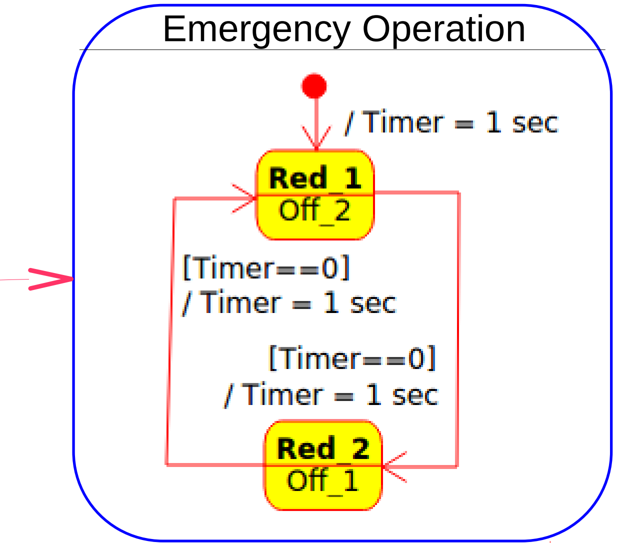
- An **event** is a type of occurrence that potentially affects the state of the system
  - The event may be generated e.g., by user action, timer expiration, or changes external to the system
  - The event may have one or more parameters
  - Events follow a 3-state lifecycle
    - Received, where it waits on an event queue until a machine reaches a state able to dispatch that event



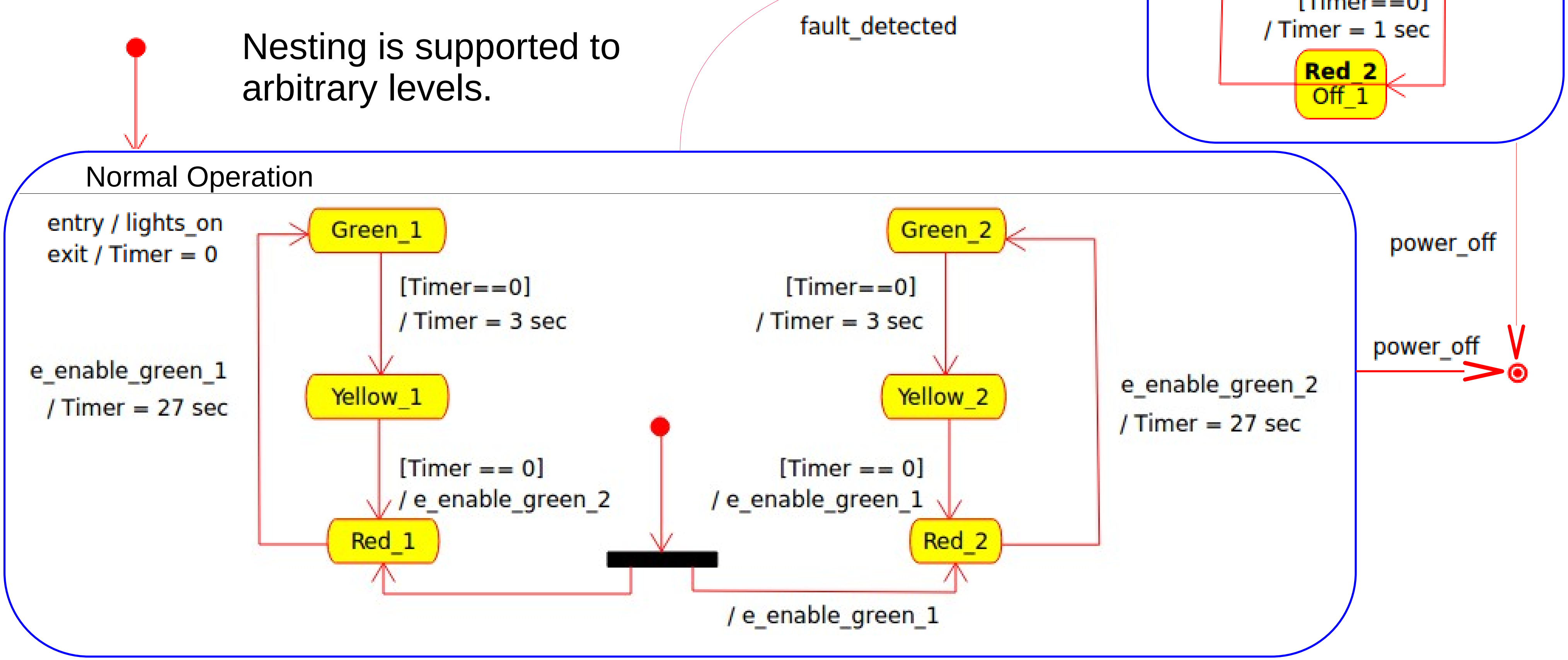
    - Dispatched, where it affects the state machine
    - Consumed, where it is no longer available for use
  - Only one event may be dispatched at a time
  - An event which no machine can handle is quietly discarded

Hierarchical State Machines
(HSM)

- Unique (originally) to UML state machines was hierarchy.
- On entry to a sub-state machine, the system:
  - Processes the sub-state entry / activity, if any
  - Initiates the sub-state machine from the initial state
  - Dispatches each event to the lowest level state machine that can receive it



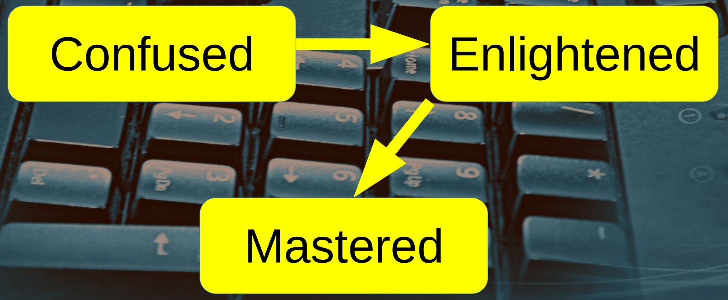
Nesting is supported to arbitrary levels.



Structural

State Design Pattern

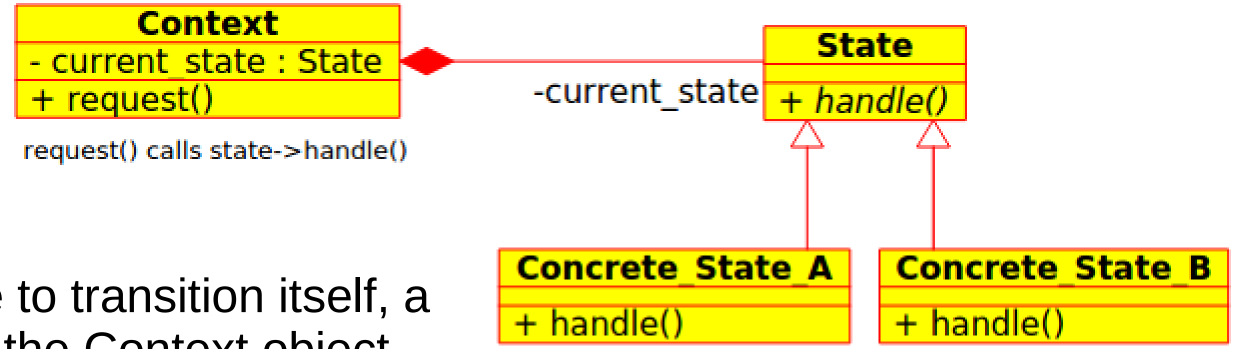
- The **State Design** pattern supports full encapsulation of unlimited states within a scalable context
  - This is a variation on the Strategy Pattern, optimized for state-dependent behaviors
  - The pattern allows the context (state machine) behavior to depend on the currently active State instance



State Design Pattern

Context is the state machine. It's current state is an instance of a concrete state class.

State is the virtual base class that declares (and in some cases defines) common state operations (represented here by *handle()*).



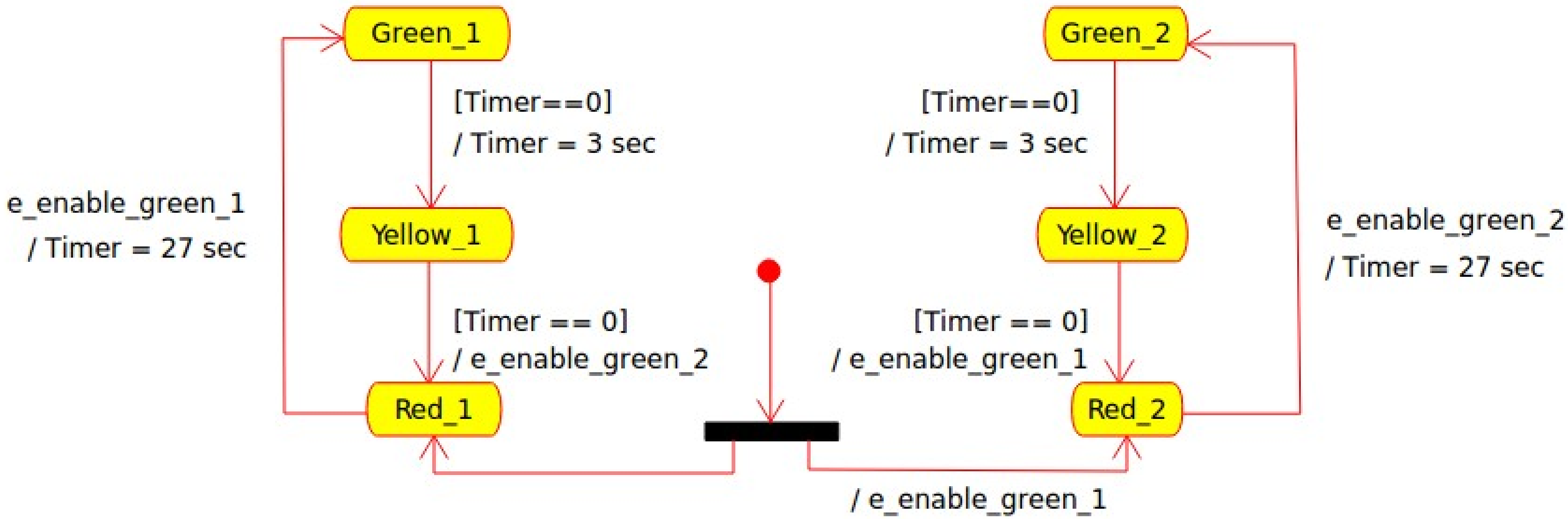
For the state to transition itself, a reference to the Context object must be passed to handle().

Note that event handling is not addressed by this pattern, and must be added to realize many UML state machine designs.

Classes derived from State represent the states defined within the state machine model. The handle() methods

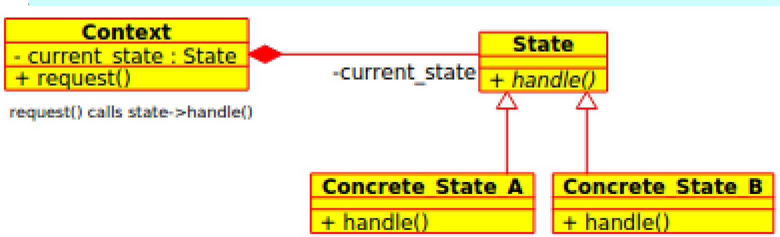
Implementing our Traffic Light

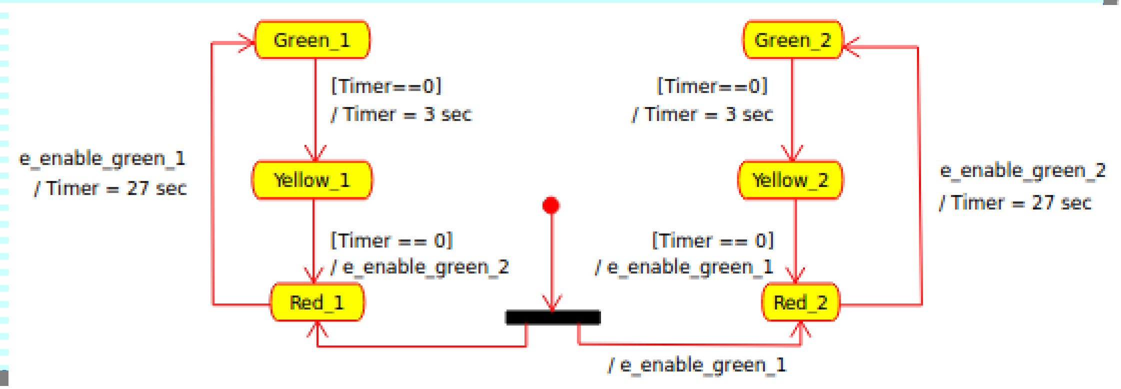
- We'll use the State Design Pattern, augmented by a basic event handler, to automate the traffic state diagram below



First, We Need Light Colors

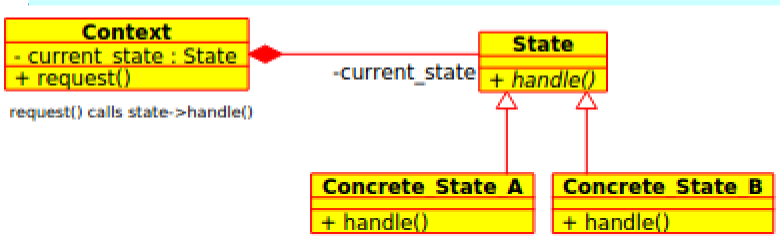
```
#include <stdexcept>
#include <iostream>

//
// Traffic light colors
//

enum class Traffic_light_color {GREEN, YELLOW, RED};
std::string ctos(Traffic_light_color color) {
  if (color == Traffic_light_color::GREEN) return "green";
  if (color == Traffic_light_color::YELLOW) return "yellow";
  if (color == Traffic_light_color::RED) return "red";
  throw std::runtime_error("ctos: invalid color");
}
```

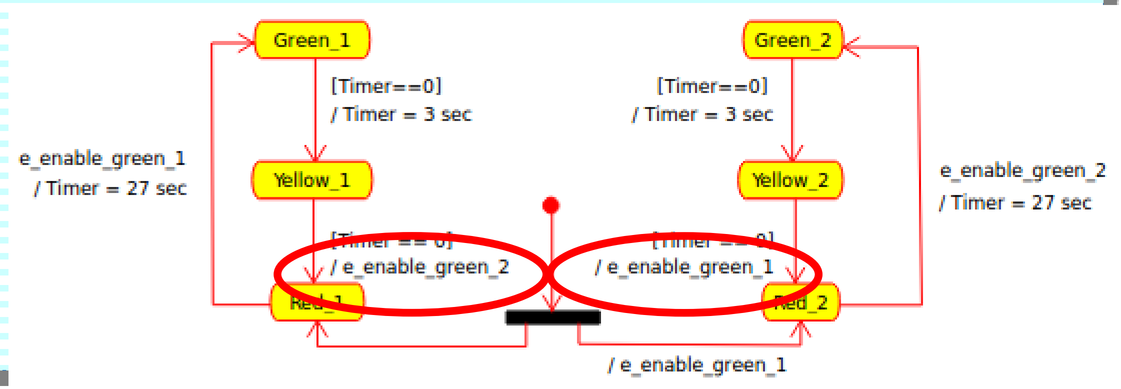




Next, a Simple Event Handler

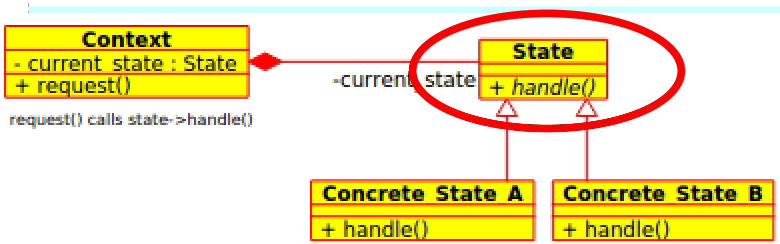
```
//
// Events
//

class Event {
  public:
    void generate() {++_pending;}
    bool consume() {
      if (_pending > 0) {--_pending; return true;}
      return false;
    }
  private:
    int _pending = 0;
};

Event e_enable_green_1;
Event e_enable_green_2;
```

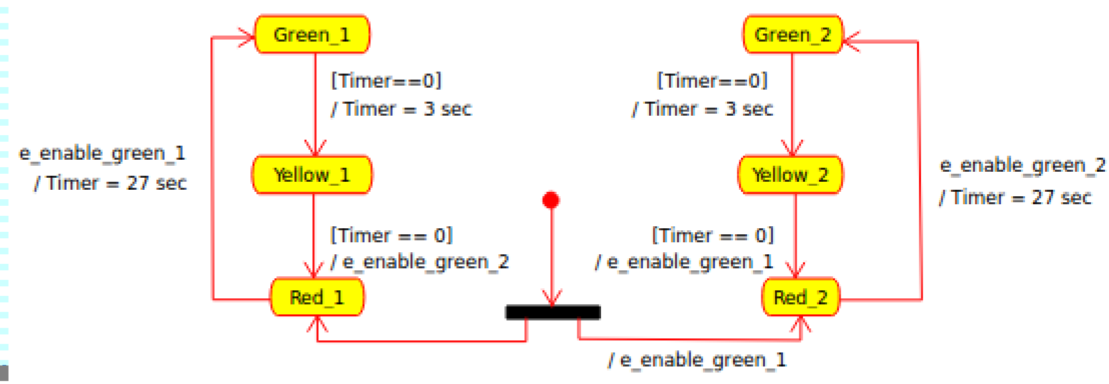




Our Virtual State Class

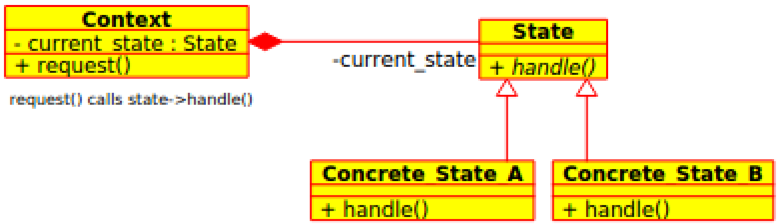
```
//
// States
//

class State {
  public:
    State(int seconds) : _seconds{seconds} { }
    virtual Traffic_light_color color() {throw runtime_error("State color");}

    State* tic() {
      if (_seconds > 0) --_seconds;
      return handle();
    }

  protected:
    virtual State* handle() {throw runtime_error("State handle");}
    int _seconds = 0;
};
```

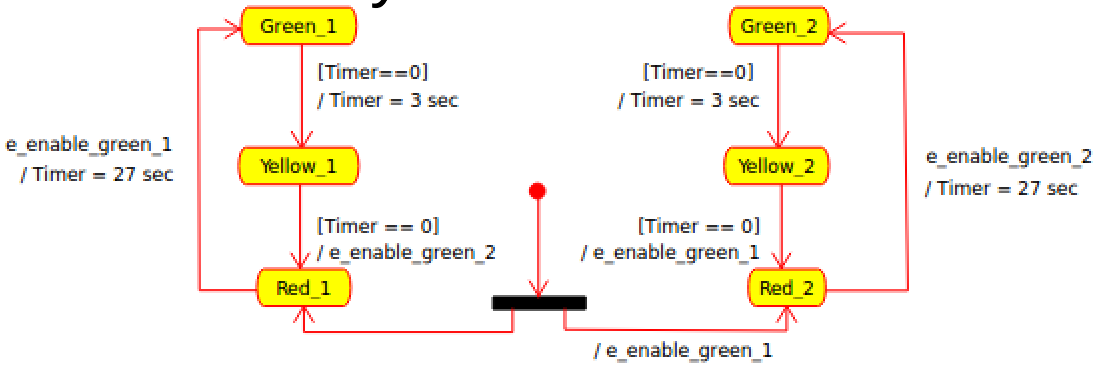




A Minor Problem

- Each state must be able to (potentially) create instances of every other state
- We can forward reference *pointers* in C++, but not *instances*
- SOLUTION: We'll create a state_factory function that generates a state on the heap and returns a pointer, based on a *string* descriptor, e.g., “Green_1”
  - We'll need to remember to delete each state as we transition away from it to avoid memory leaks



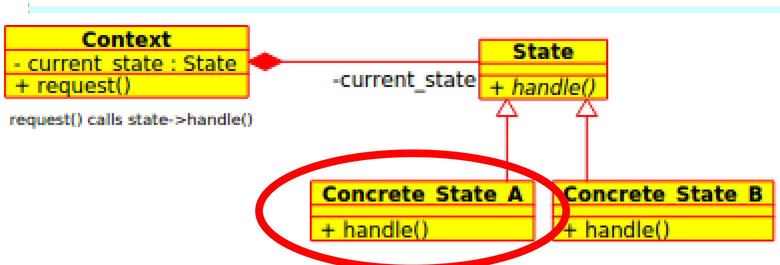


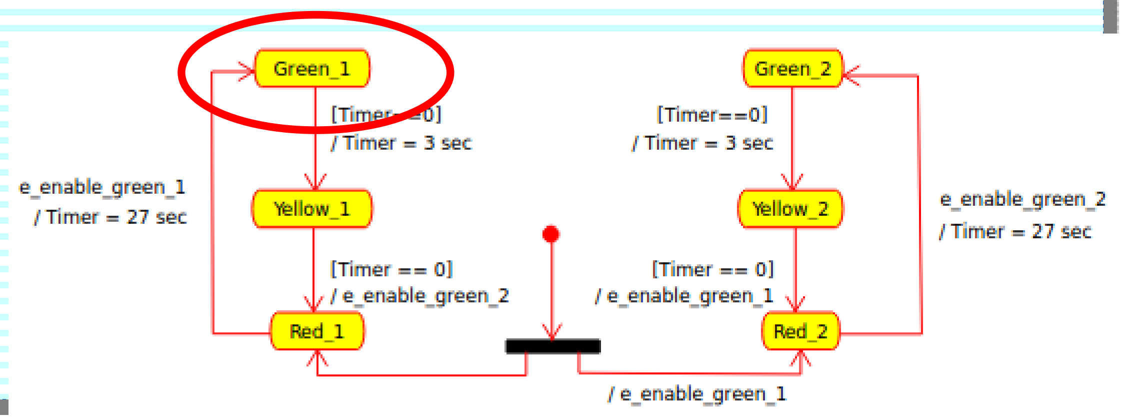
Our First State - Green_1

```
// Provide every state access to every other state
State* state_factory(string state);

class Green_1 : public State {
  public:
    Green_1() : State(27) { }
    Traffic_light_color color() override {
      return Traffic_light_color::GREEN;
    }
  protected:
    State* handle() override {
      if (_seconds <= 0) {
        return state_factory("Yellow_1");
      } else {
        return this;
      }
    }
};
```

The Green_1 to Yellow_1 transition depends only on the timer, not events

Our state_factory in action





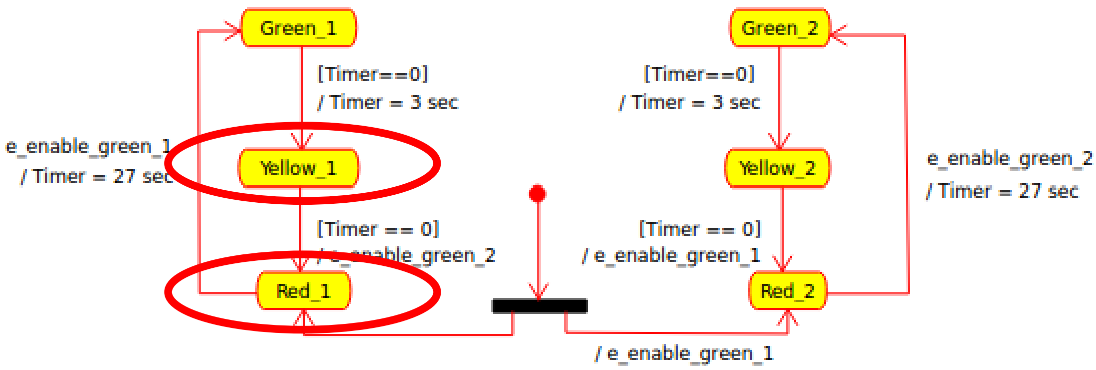
Yellow_1 and Red_1

```
class Yellow_1 : public State {
  public:
    Yellow_1() : State(3) { }
    Traffic_light_color color() override {
      return Traffic_light_color::YELLOW;
    }
  protected:
    State* handle() override {
      if (_seconds <= 0) {
        e_enable_green_2.generate();
        return state_factory("Red_1");
      } else {
        return this;
      }
    }
};
```

The Yellow_1 to Red_1 transition depends only on the timer, but generates an event

```
class Red_1 : public State {
  public:
    Red_1() : State(0) { }
    Traffic_light_color color() override {
      return Traffic_light_color::RED;
    }
  protected:
    State* handle() override {
      if (e_enable_green_1.consume()) {
        return state_factory("Green_1");
      } else {
        return this;
      }
    }
};
```

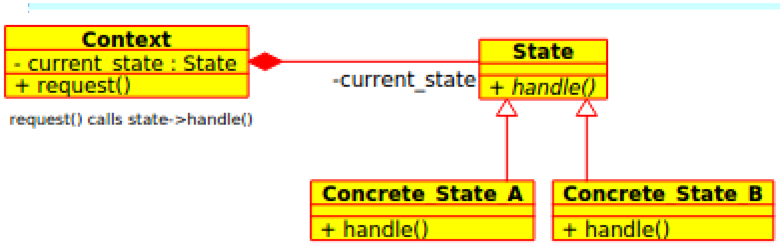
The Red_1 to Green_1 transition depends only on an event

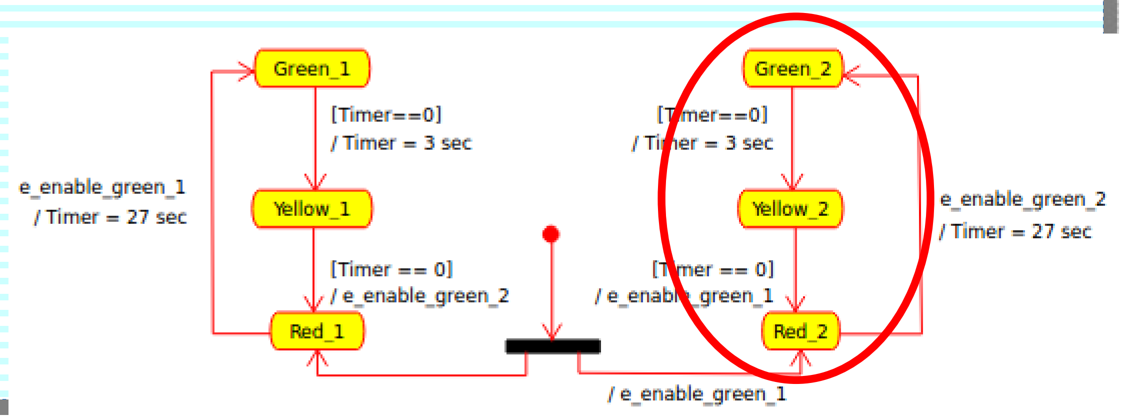


Our state_factory

```
State* state_factory(string state) {
    if (state == "Green_1") return new Green_1{};
    if (state == "Green_2") return new Green_2{};
    if (state == "Yellow_1") return new Yellow_1{};
    if (state == "Yellow_2") return new Yellow_2{};
    if (state == "Red_1") return new Red_1{};
    if (state == "Red_2") return new Red_2{};
    throw runtime_error("state_factory: Invalid state: " + state);
}
```

Green_2, Yellow_2, and Red_2 are very similar – just swap the 1's and 2's. :-)



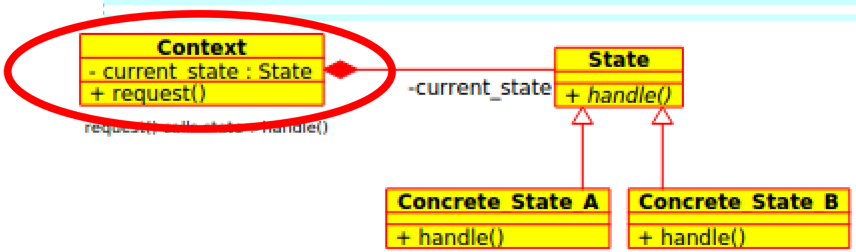


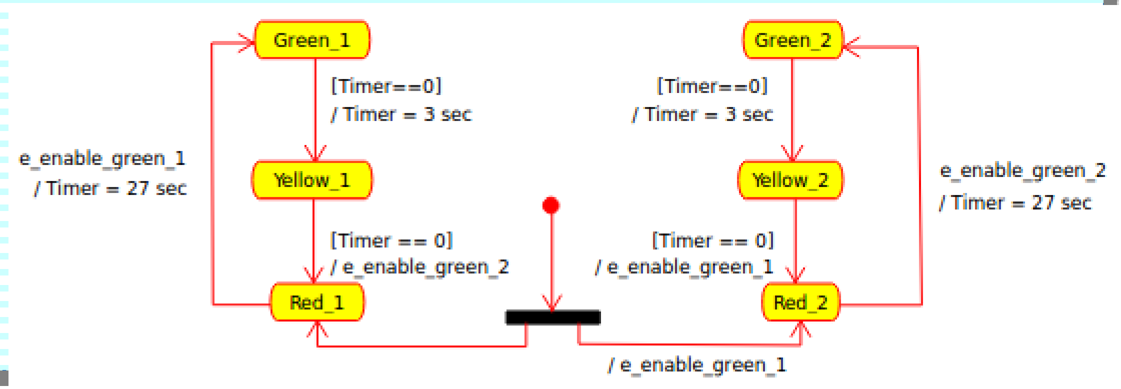
The State Machine Class

```
//
// State machines
//

class Light { // Context
  public:
    Light(State* state) : _state{state} { }
    Traffic_light_color color() {return _state->color();}
    void tic() {
      State* _newstate = _state->tic();
      if (_newstate != _state) {
        delete _state;
        _state = _newstate;
      }
    }
  private:
    State* _state;
};
```

The transition logic has been delegated to the states, so the state machine itself is very simple and reusable!

When a state is no longer needed, we must delete it to avoid memory leaks.



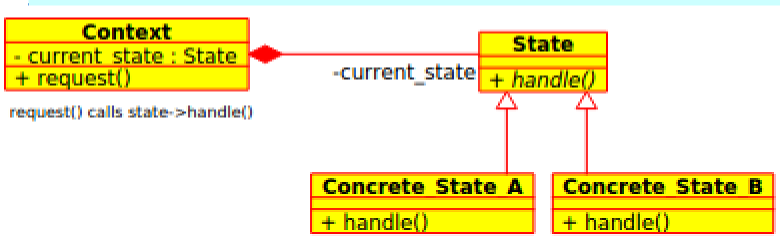


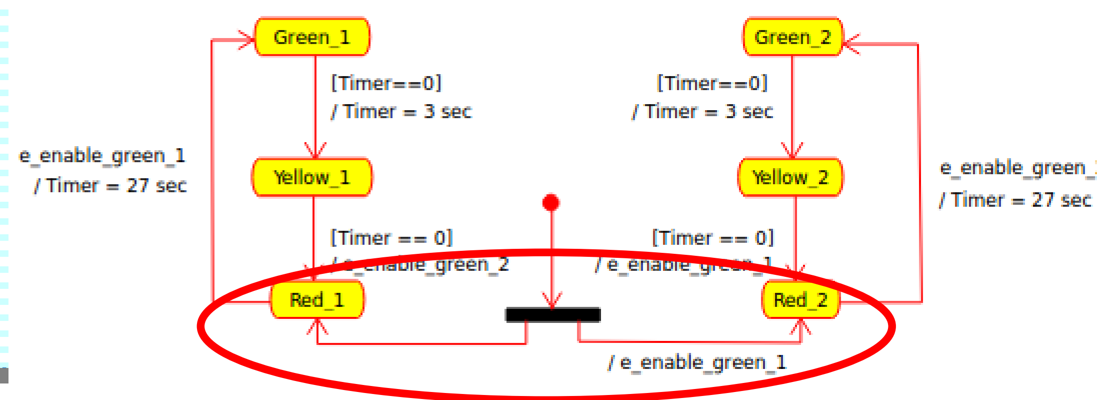
Finally, main!

```
//////////
// Main //
//////////
int main() {
  Light north_south{state_factory("Red_1")};
  Light east_west{state_factory("Red_2")};
  e_enable_green_1.generate();

  for (int i=0; i < 300; ++i) {
    cout << i << ": " << ctos(north_south.color()) << " "
                      << ctos(east_west.color()) << endl;
    east_west.tic();
    north_south.tic();
  }
}
```

Two traffic lights are created, distinguished by their initial state.

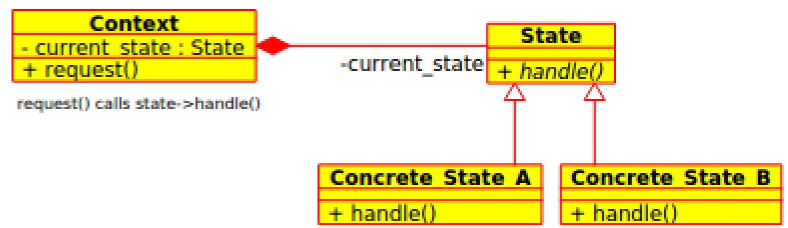
Then we generate an event to kick things off, then tic off 300 seconds.

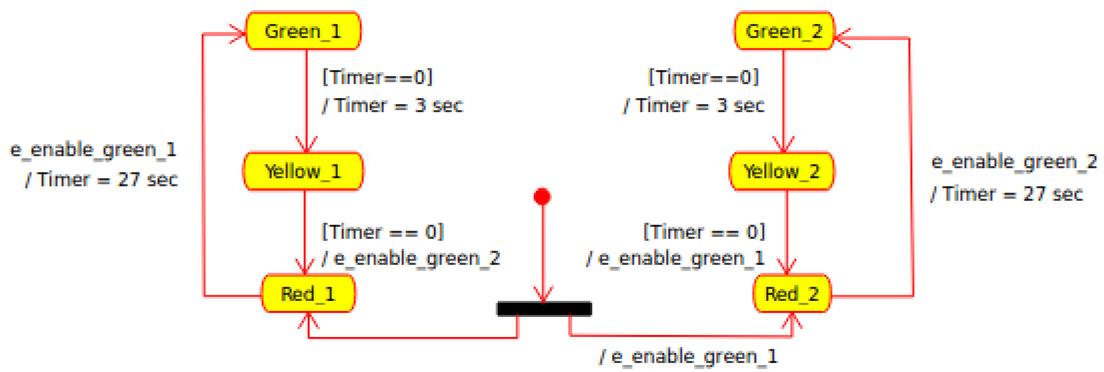




Testing

```
0: red red
1: green red
2: green red
...
25: green red
26: green red
27: green red
28: yellow red
29: yellow red
30: yellow red
31: red red
32: red green
33: red green
34: red green
...
54: red green
55: red green
56: red green
57: red green
58: red green
59: red yellow
60: red yellow
61: red yellow
62: green red
63: green red
64: green red
...
86: green red
87: green red
88: green red
89: yellow red
90: yellow red
91: yellow red
92: red red
93: red green
94: red green
95: red green
...
```





Output has been truncated at the ellipses to fit multiple cycles on the screen.

```
github.com/AlDanial/cloc v 1.71  T=0.03 s (35.9 files/s, 7141.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              1             22             16            161
-------------------------------------------------------------------------------
```

```
117: red green
118: red green
119: red green
120: red yellow
121: red yellow
122: red yellow
123: green red
124: green red
125: green red
...
147: green red
148: green red
149: green red
150: yellow red
151: yellow red
152: yellow red
153: red red
154: red green
155: red green
156: red green
...
178: red green
179: red green
180: red green
181: red yellow
182: red yellow
183: red yellow
184: green red
185: green red
186: green red
...
208: green red
209: green red
210: green red
211: yellow red
212: yellow red
213: yellow red
214: red red
215: red green
216: red green
217: red green
218: red green
```

Other C++ State Machine Implementations

- Professional UML tools offer sophisticate state machine implementations with code generation
  - IBM Rhapsody, MagicDraw, Visual Paradigm...
- Some frameworks provide generalized support
  - Boost MSM, Quantum Platform
- Writing a tailored framework based on the State Design Pattern is always an option