CSE 1325: Object-Oriented Programming

Lecture 19

Embedded Programming, and UML State Diagrams with a C++ Implementation

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

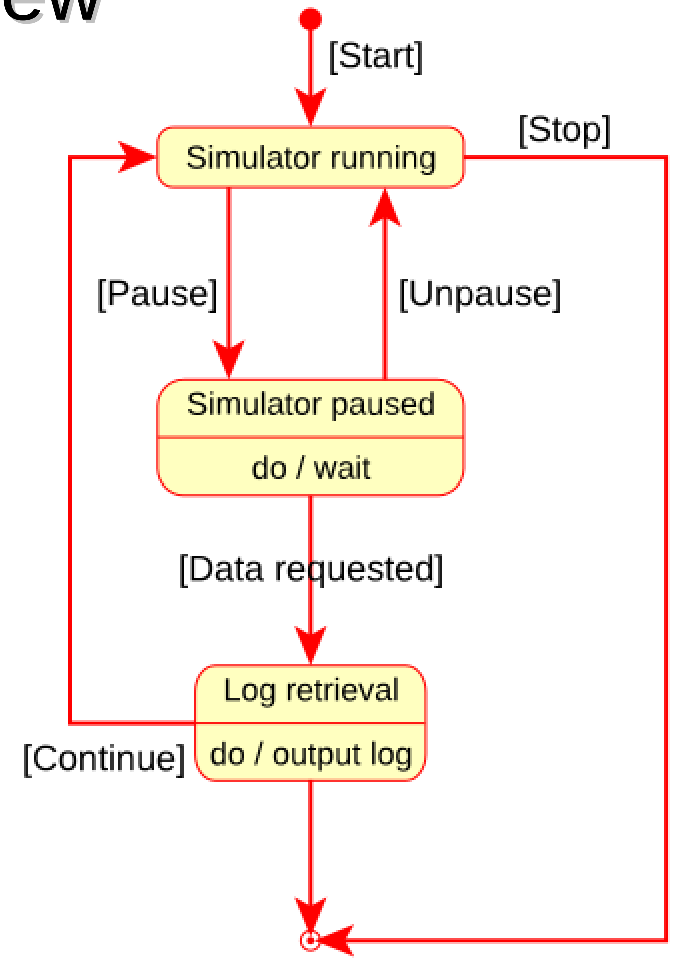
Or by appointment

Quick Review

- **True** or False: Most (but not all) stream operators are "sticky". If False, which is it? If True, give an example of each. **Hex is sticky, setw is not sticky.**
- True or **False**: Stream operators are constants and thus cannot accept parameters.
- To output hexadecimal numbers via cout, include the **hex** operator in the stream. To precede hexadecimal numbers with "0x", include **showbase** in the stream.
- What happens if a value exceeds the specified stream output width? **Correct data takes precedence over formats such as output width**
- **True** or False: Binary file operations are inherently less portable than text file operations.
- Stream operations can target string variables by using the **stringstream** class.
- The **Strategy** pattern enables an algorithm's behavior to be modified at runtime.
- The UML Activity diagram displays a sequence of activities at the **algorithm** level, particularly useful for documenting Use Cases.
- Which of the following is supported by the UML Activity Diagram?
(a) Friends (b) Interrupts (c) Swimlanes (d) Hierarchical Diagrams (e) Inheritance

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

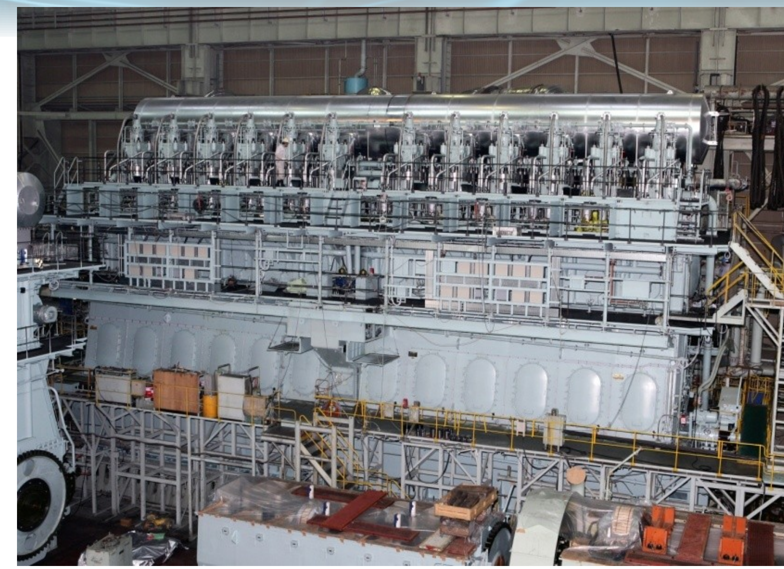


UML State Diagram (c) 2014 by Fred the Oyster, Used under Gnu FDL
https://commons.wikimedia.org/wiki/File%3AUML_State_diagram.svg

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

Tesla Model S photo by Unimog404 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=51113491
Tesla Instrument photo by Lklundin - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38233116

Who Works on Embedded Systems?

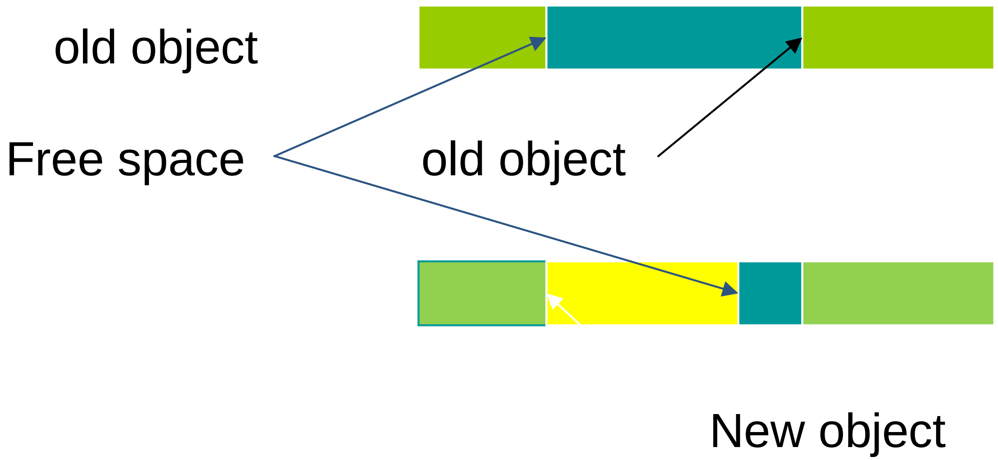
- Computer Scientists
  - Unless you only do web and client-server
- Computer Engineers
  - Hardware and software are complementary aspects of the overall system
  - The hardware must support the software <u>well</u>
- Electrical Engineers
  - You must be able to talk to the computer guys
  - Your concerns 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?)

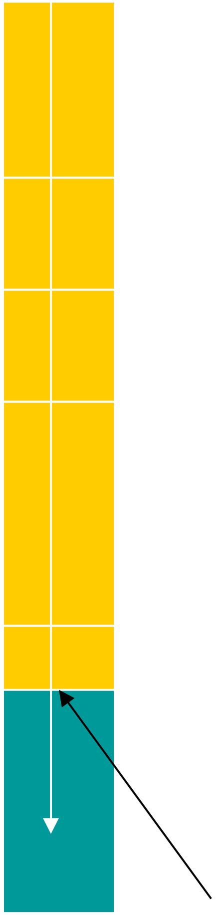


- Allocation delays
  - The effort needed to find a new free chunk of memory of a given size depends on what has already been allocated
- Fragmentation
  - If you have a “hole” (free space) of size N and you allocate an object of size M where M<N in it, you now have a fragment of size N-M to deal with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

Stack:



Top of stack

Pool:

Pool Example

```
// Note: element type is known at compile time
// Allocation times are completely predictable (and short)
// The user has to pre-calculate the maximum number of elements needed

template<class T, int N> class Pool {
public:
    Pool();            // make pool of N Ts – construct pools only during startup
    T* get();         // get a T from the pool; return 0 if no free Ts
    void free(T* p); // return a T given out by get() to the pool
private:
    // keep track of T[N] array (e.g., a list of free objects)
};

// Example use
Pool<Small_buffer, 10> sb_pool;
Pool<Status_indicator, 200> indicator_pool;
```

Stack Example

```
// Note: allocation times completely predictable (and short)
// The user has to pre-calculate the maximum number of elements needed

template<int N> class Stack {
public:
    Stack(); // make an N byte stack – construct stacks only during startup
    void* get(int N);   // allocate n bytes from the stack; return 0 if no free space
    void free(void* p); // return last block returned by get() to the stack
private:
    // keep track of an array of N bytes (e.g. a top of stack pointer)
};

// Examples

Stack<50*1024> my_free_store; // 50K worth of storage to be used as a stack

void* pv1 = my_free_store.get(256 * sizeof(int));   // allocate array of ints
int* pi = static_cast<int*>(pv1);                    // convert memory to objects

void* pv2 = my_free_store.get(50);
Pump_driver* pdriver = static_cast<Pump_driver*>(pv2);
```

Templates Rock for Real-Time!

- Vector class (for example) is a template
  - But not one that is suitable for real-time!
- 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 only
  - true/false auto-converts to/from 1/0
  - Use only if you really need more than 32 bits
- std::bitset bs(314);	// 314 bits
  - Use if you really need more than 32 bits
  - Typically efficient for multiples of sizeof(int)

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

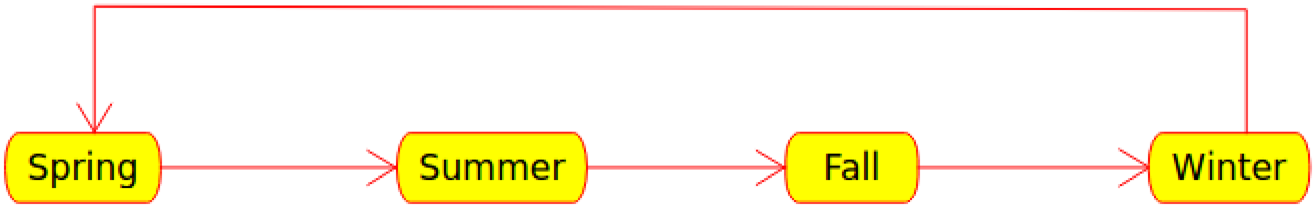
- Hardware failures are <u>much</u> more common in embedded than in IT
  - Power surges and sags
  - Connectors vibrate loose
  - Physical damage
  - Environmental extremes
- The system may be remote when it fails
  - Like, on Mars... or in the Mariana Trench

- 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

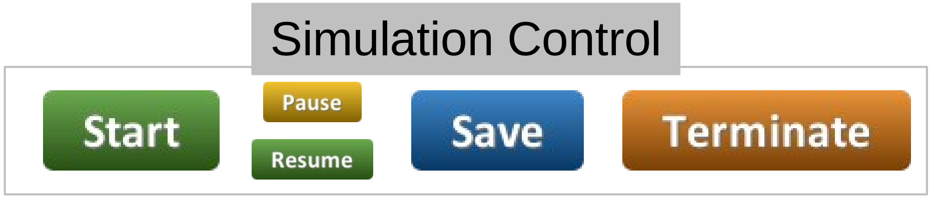
- Many embedded (and IT) systems exhibit state-like behavior
- A **State** is the cumulative value of all relevant stored information to which a system or subsystem has access.

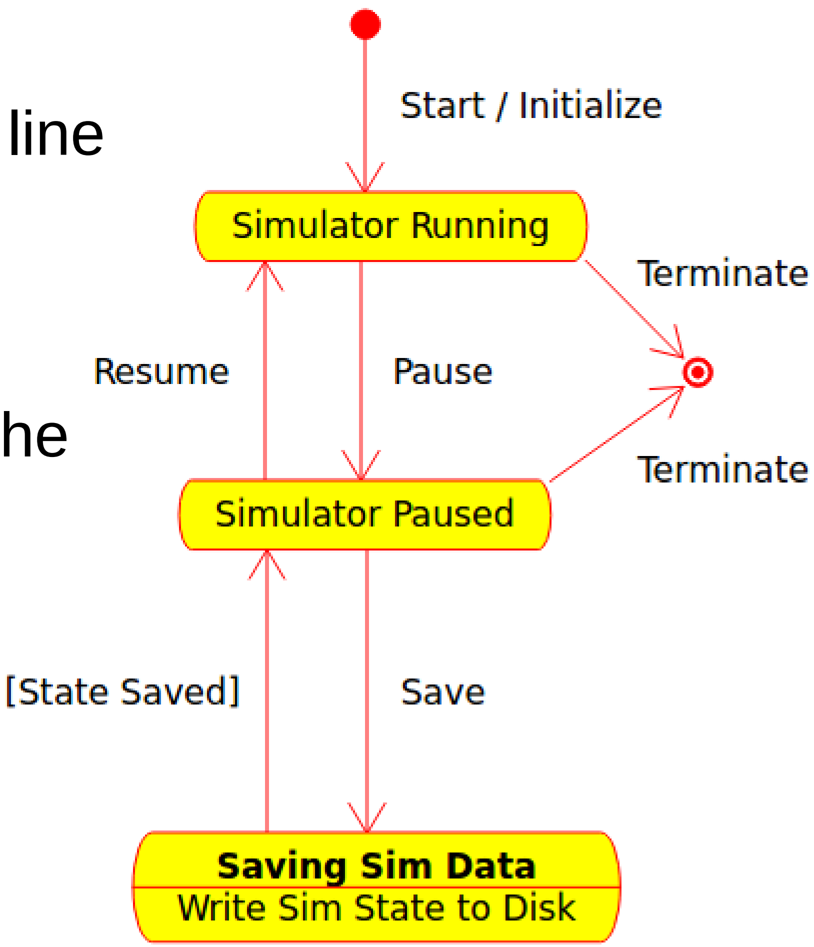
  - The output of a stateful system is completely determined by its current state and its current inputs
  - A **State Diagram** documents the states, permissible transitions, and activities of a system



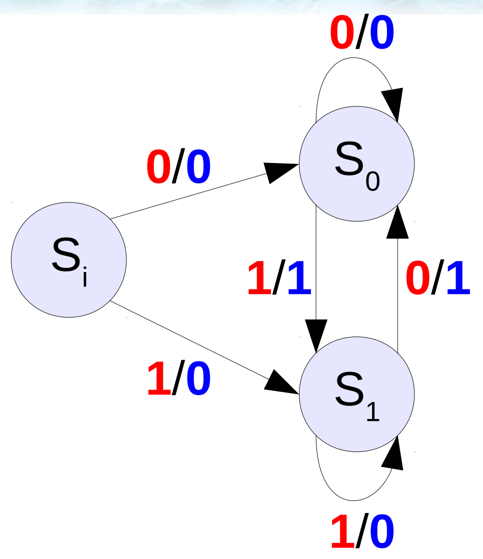
Basic UML State Diagram Elements

- 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)



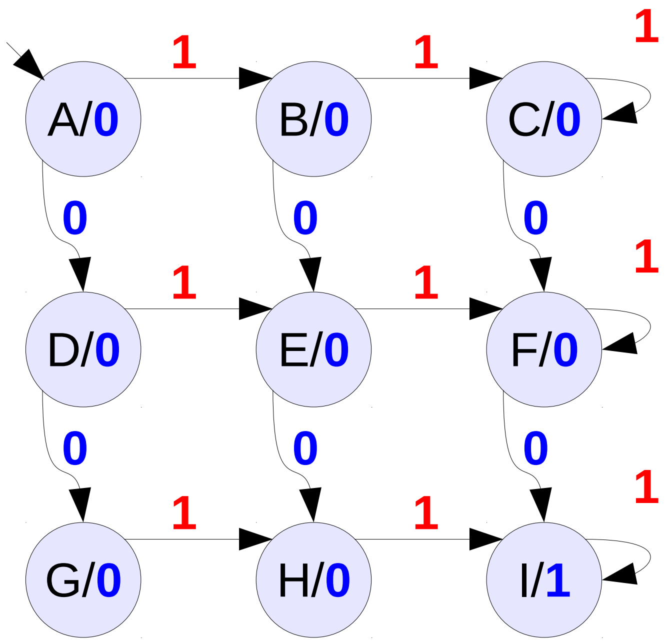


(Non-UML) Mealy vs Moore Diagrams



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

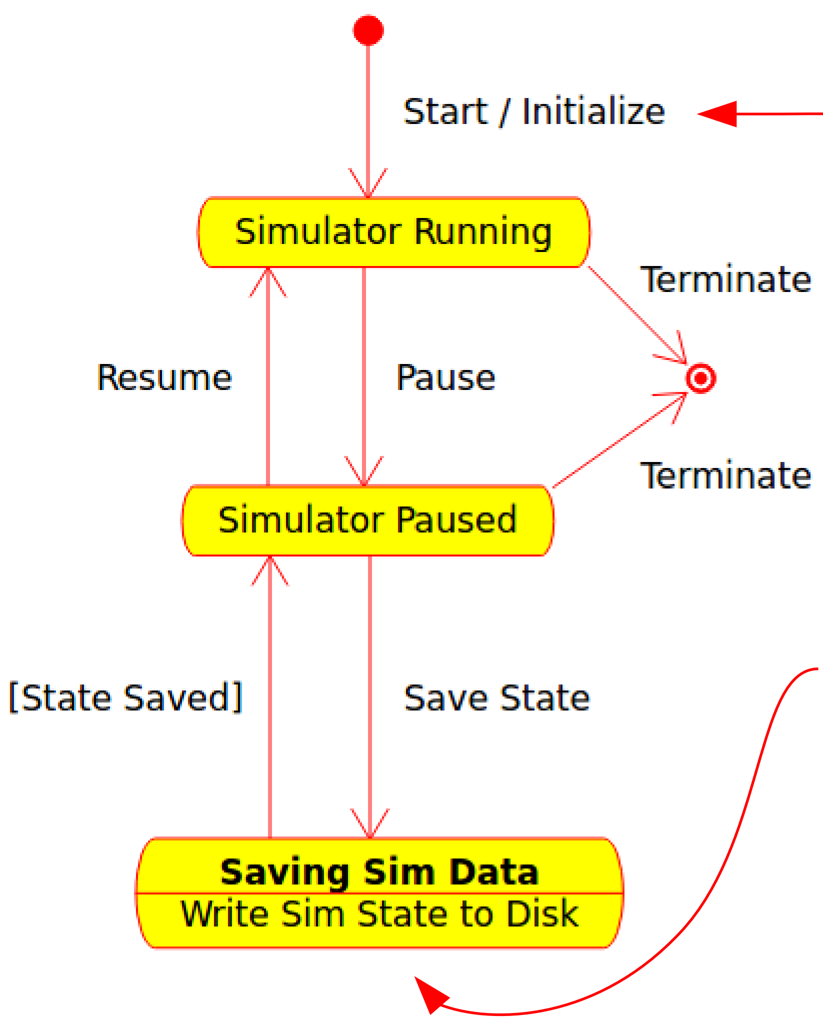
- Outputs are defined on *transitions*
- Outputs follow inputs immediately
- Often require fewer states



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

Isn't This Just an Activity Diagram?

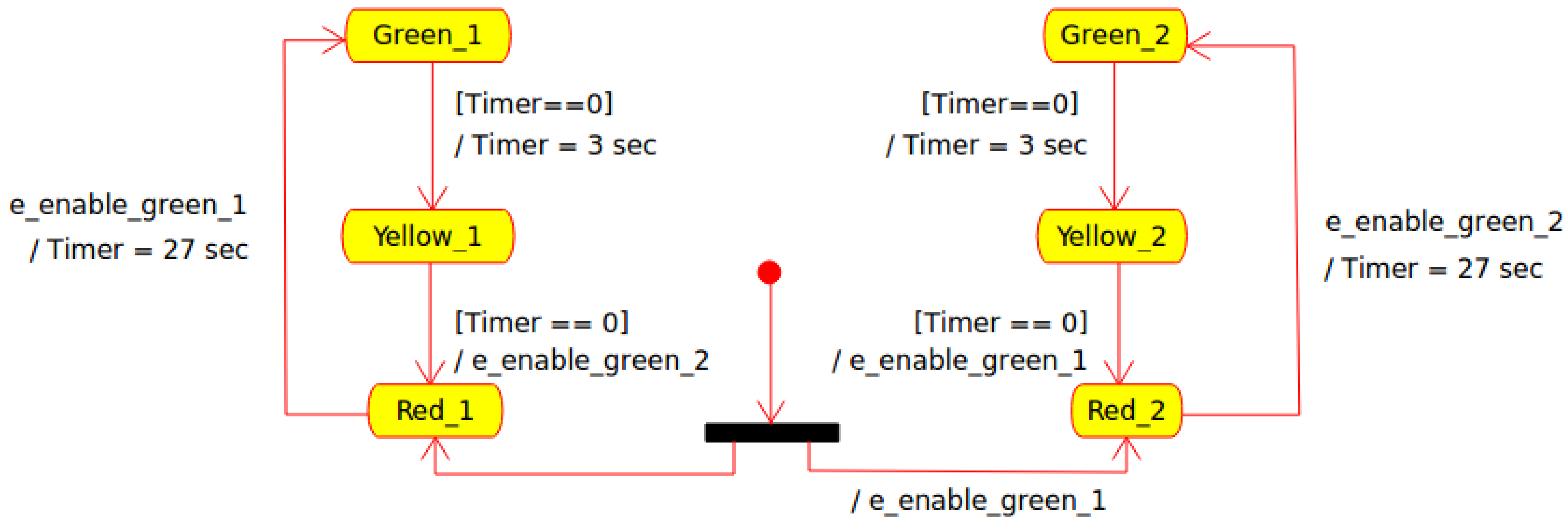
- Not exactly, but they are closely related
  - **An activity diagram is a special case of a state diagram**, in which the states consist only of activities and transitions are (almost) always triggered by completion of those activities rather than external events
  - Conceptually, activity diagrams may imply a data flow graph, while a state diagram may imply a structured, centralized control scheme
- Action and Activity states are conceptually different
  - An action state in a state digram cannot be further decomposed, and happens instantly (although an associated activity may consume time)
  - An activity state in an activity diagram can be further decomposed, and the state has a definite timespan before the next transition
- Either may be used to document a use case or to model the behavior of a class or subsystem
  - As may a Sequence Diagram

History of UML State Diagrams

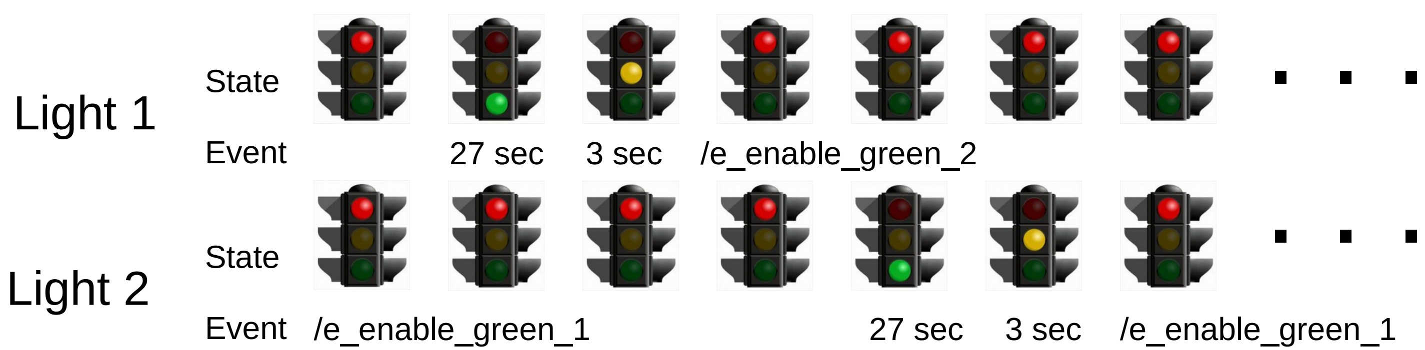
- Basic state diagrams have been around since the dawn of computers
  - States are fundamental to all clock-driven circuits
  - Hardware implementations have special challenges
- Dr. David Harel formalized and greatly extended hierarchical statechart semantics[1] in 1986, and is the “Father of Modern Statecharts”
- UML adopted an object-oriented variant of Harel Statecharts in 1997, and significantly extended the semantic model again

[1] http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/Statecharts.pdf

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

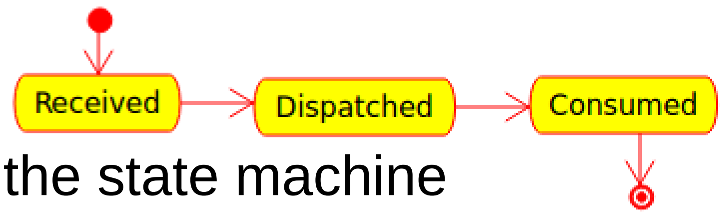
Guards

- 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

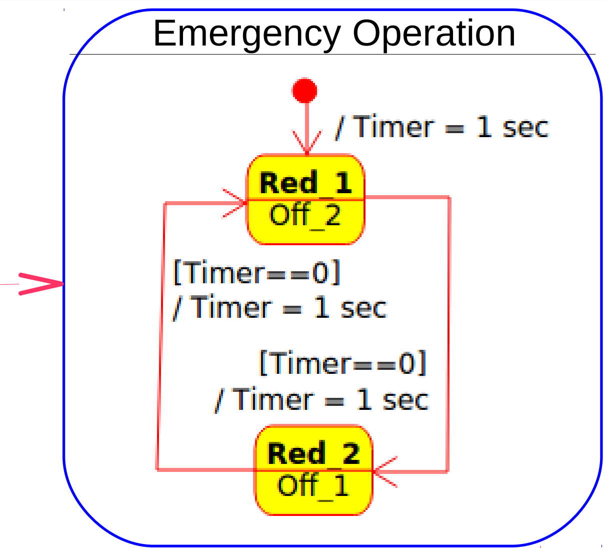
  - The event may be generated e.g., by user action, timer expiration, or changes external to the system
  - The event may have one or more parameters
  - Events follow a 3-state lifecycle
    - Received, where it waits on an event queue until a machine reaches a state able to dispatch that event



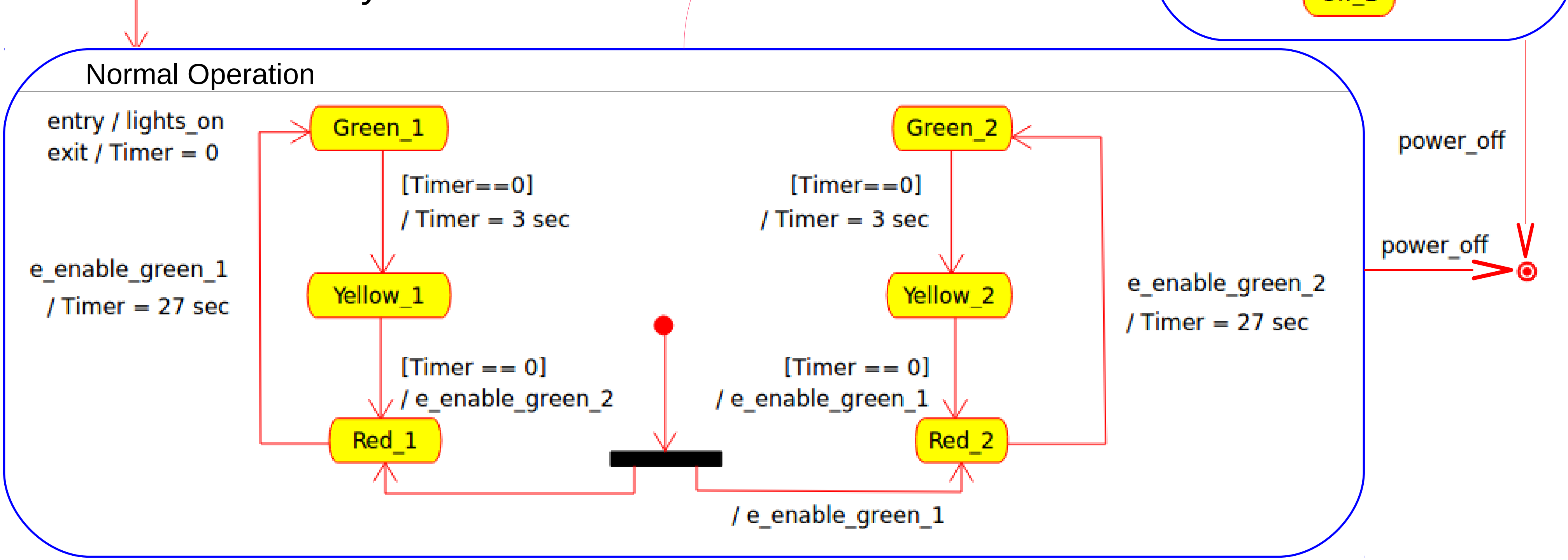
    - Dispatched, where it affects the state machine
    - Consumed, where it is no longer available for use
  - Only one event may be dispatched at a time
  - An event which no machine can handle is quietly discarded

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



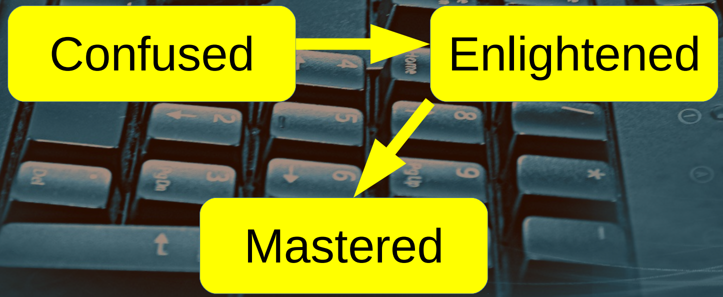
Nesting is supported to arbitrary levels.



Structural

State Design Pattern

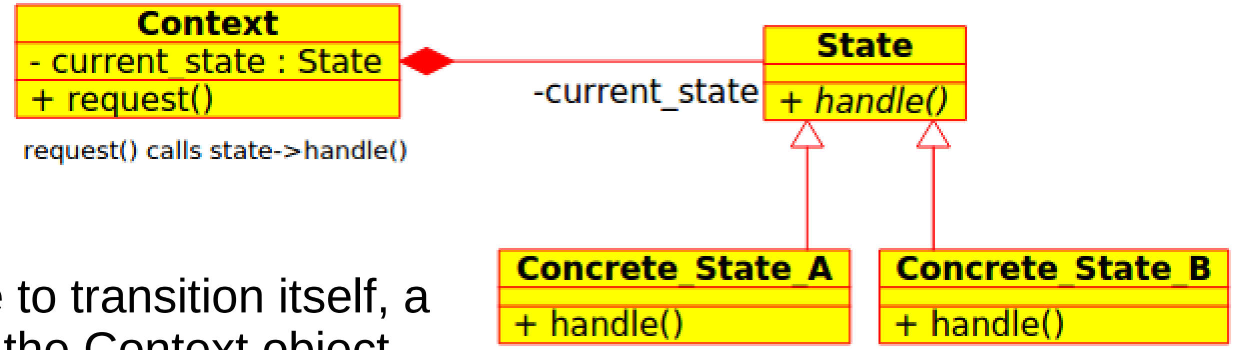
- The **<u>State Design</u>** pattern supports full encapsulation of unlimited states within a scalable context
  - This is a variation on the Strategy Pattern, optimized for state-dependent behaviors
  - The pattern allows the context (state machine) behavior to depend on the currently active State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



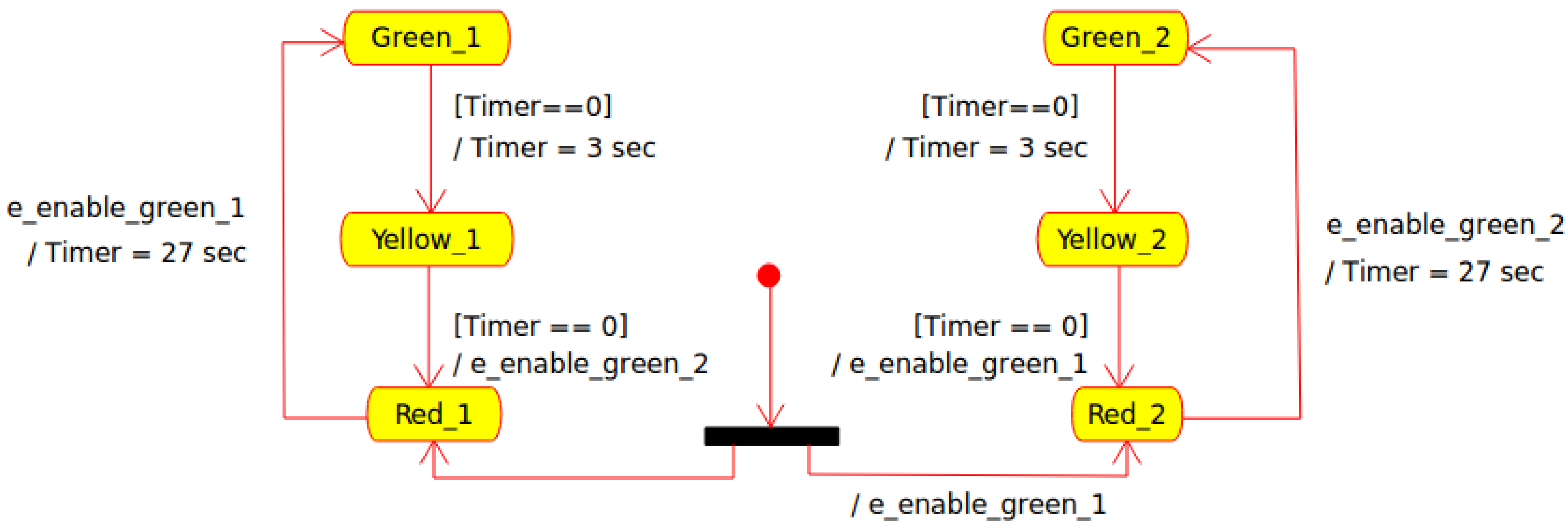
For the state to transition itself, a reference to the Context object must be passed to handle().

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

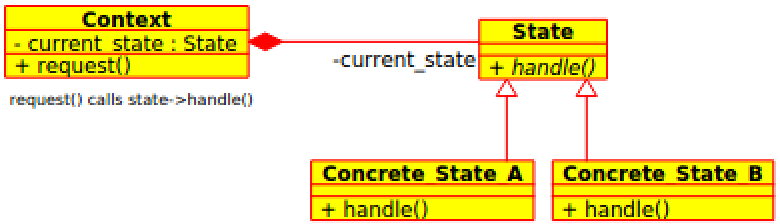
Implementing our Traffic Light

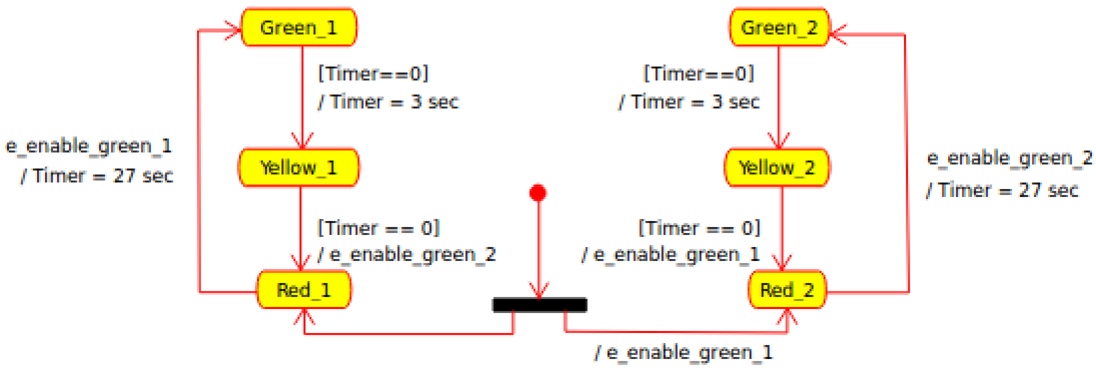
- We'll use the State Design Pattern, augmented by a basic event handler, to automate the traffic state diagram below



First, We Need Light Colors

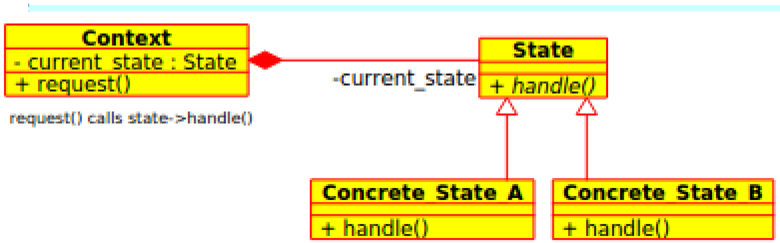
```
#include <stdexcept>
#include <iostream>

using namespace std;

//
// Traffic light colors
//

enum class Traffic_light_color {GREEN, YELLOW, RED};
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 runtime_error("ctos: invalid color");
}
```

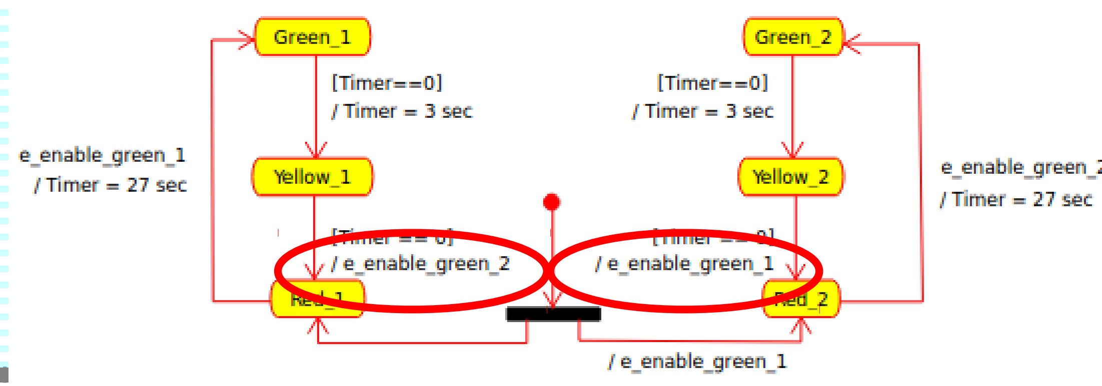




Next, a Simple Event Handler

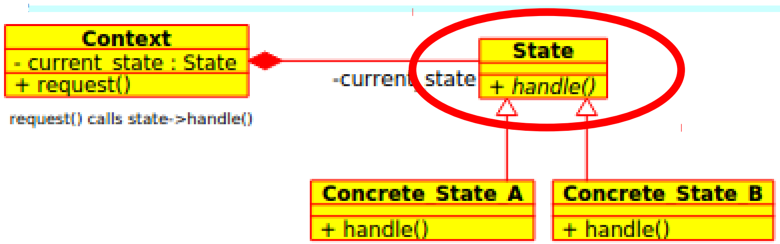
```
//
// Events
//

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;
```

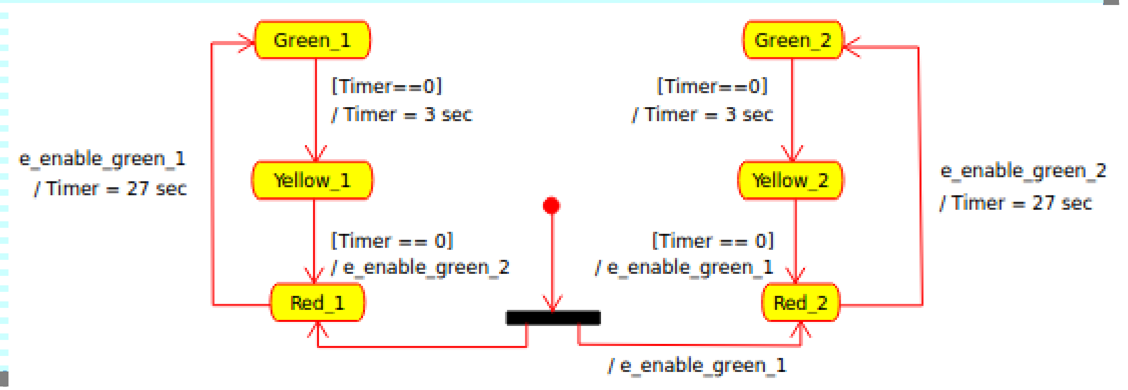




Our Virtual State Class

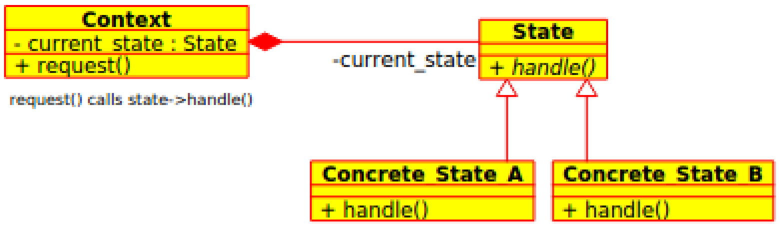
```
//
// States
//

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;
};
```

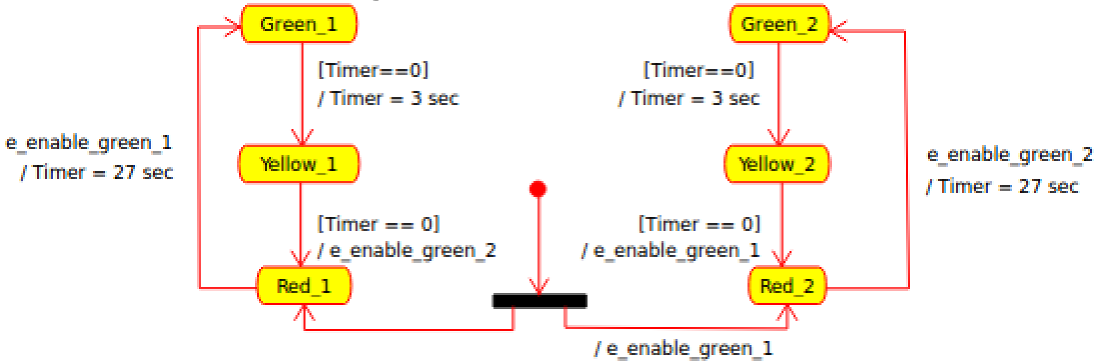




A Minor Problem

- 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



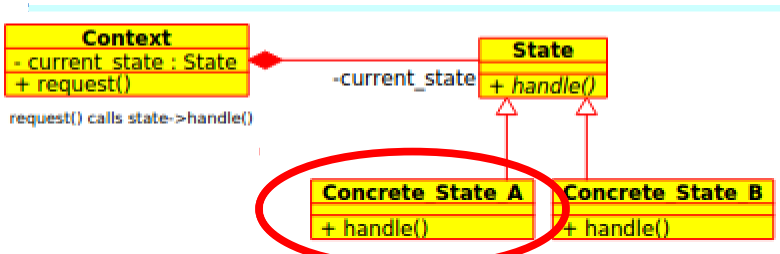


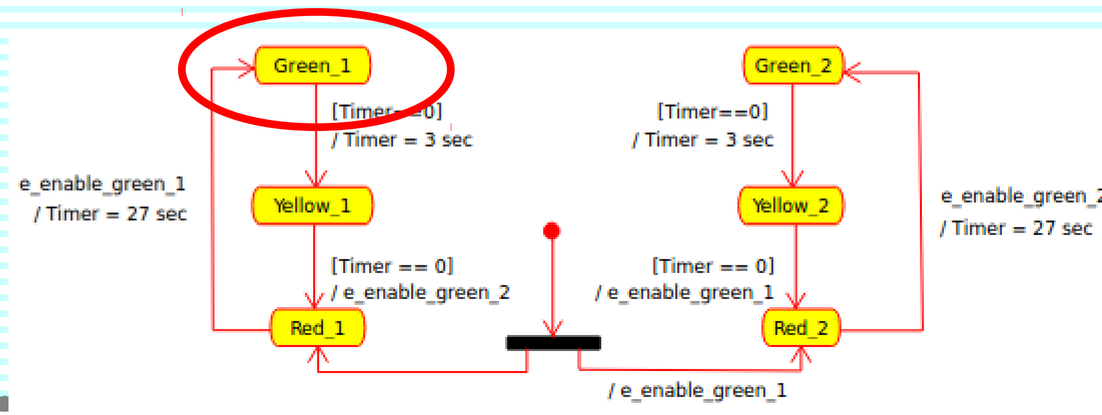
Our First State - Green_1

```
// 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





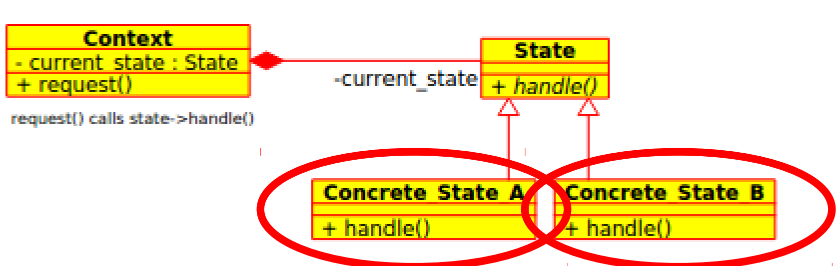
Yellow_1 and Red_1

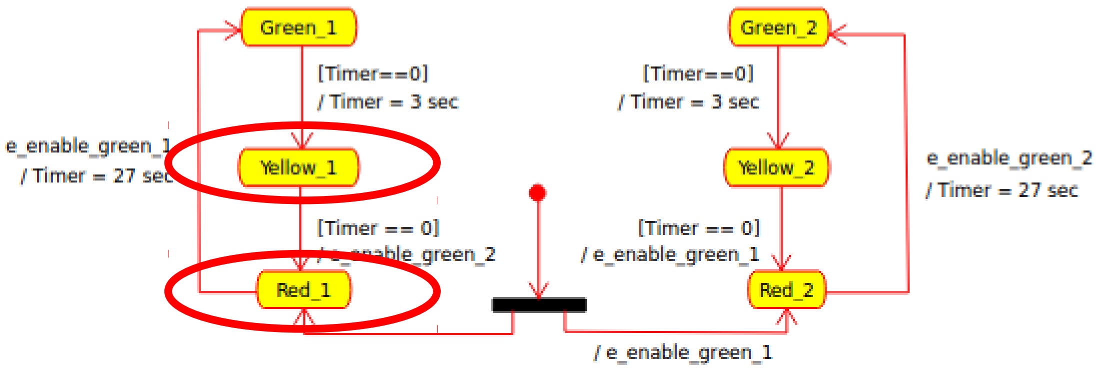
```
class Yellow_1 : public State {
  public:
    Yellow_1() : State(3) { }
    Traffic_light_color color() override {
      return Traffic_light_color::YELLOW;
    }
  protected:
    State* handle() override {
      if (_seconds <= 0) {
        e_enable_green_2.generate();
        return state_factory("Red_1");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

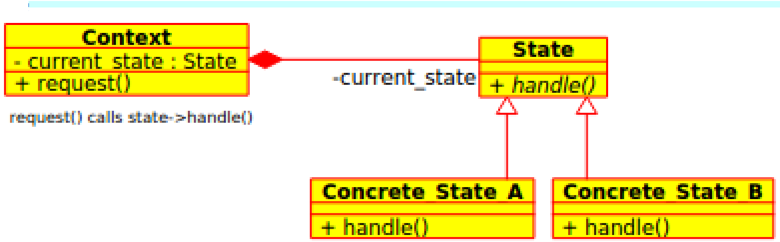


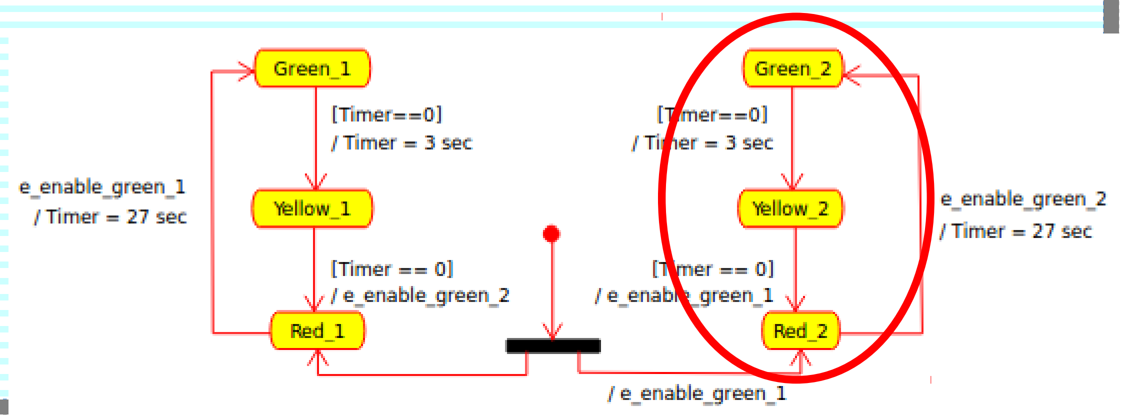


Our state_factory

```
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. :-)



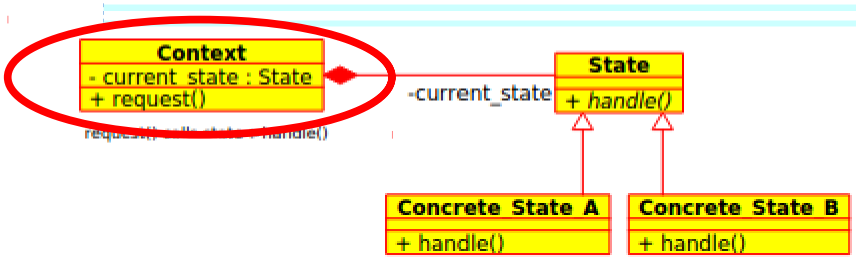


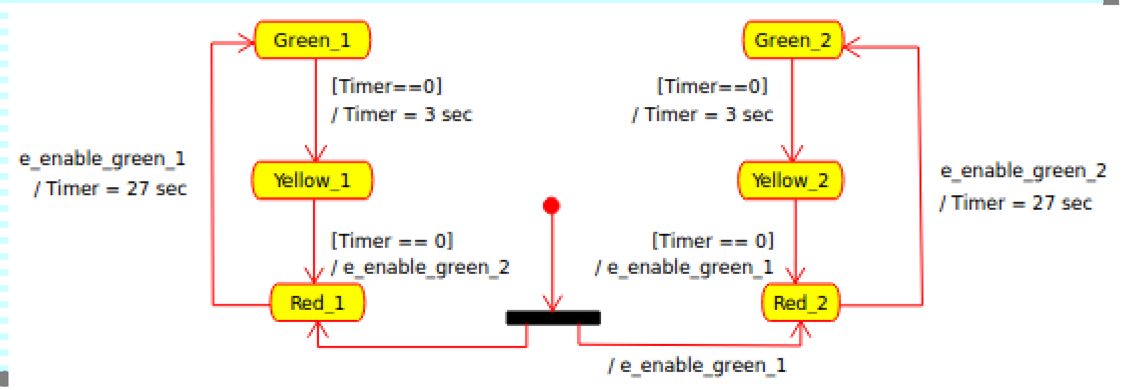
The State Machine Class

```
//
// 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.



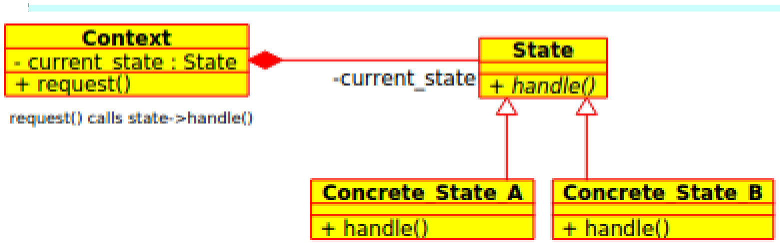


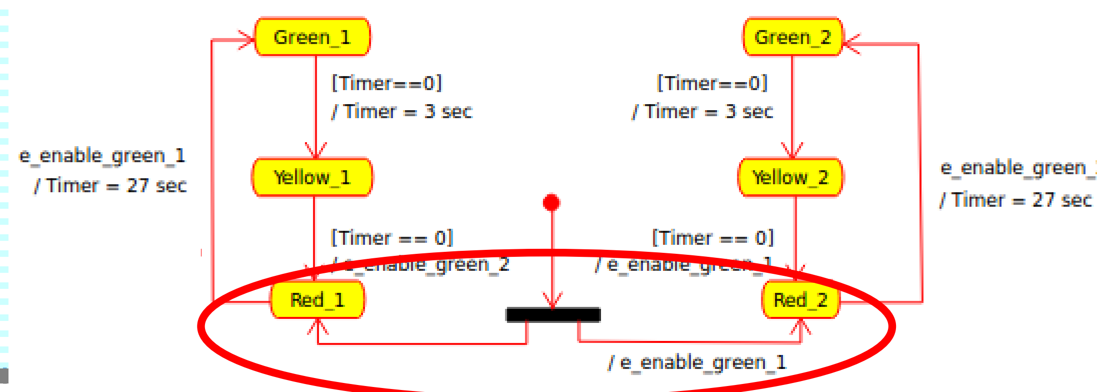
Finally, main!

```
//////////
// 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.

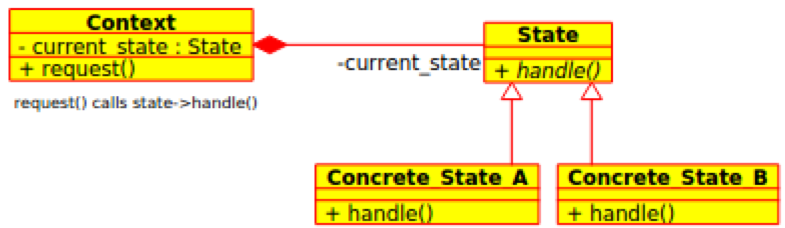
Then we generate an event to kick things off, then tic off 300 seconds.

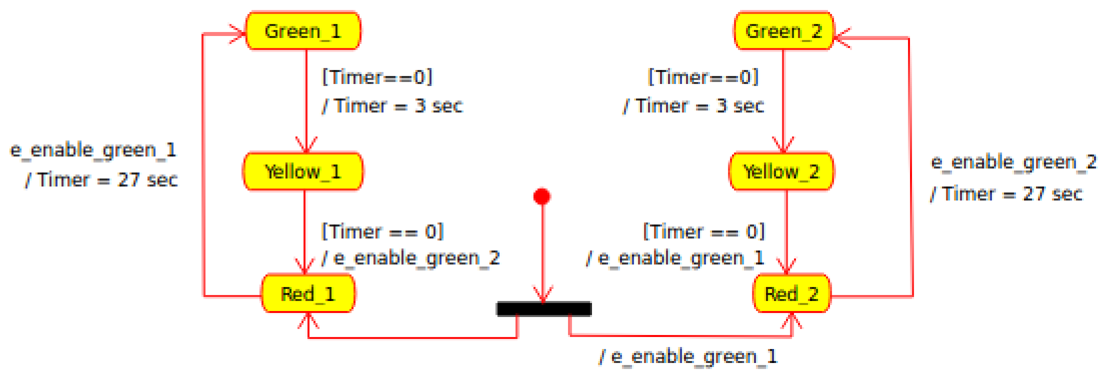




Testing

```
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
  -

Quick Review

- A _____ is the cumulative value of all relevant stored information to which a system or subsystem has access.
- A ____ ________ documents the states, permissible transitions, and activities of a system.
- An ________ is a type of occurrence that potentially affects the state of the system
- A _____ is a Boolean expression that enables a state transition when true and disables it when false, e.g., [power == on]
- True or False: A UML State Diagram is a special case of the UML Activity Diagram.
- The _____ _______ _______ supports full encapsulation of unlimited states within a scalable context. It does not, however, support events as is. It is closely related to the _________ _________.
- A _____ state machine defines outputs based on current state <u>and</u> current inputs, while a _____ state machine defines outputs based on current state only.

For Next Class

- (Optional) Read Chapter 25 in Stroustrop
  - Do the Drills!
- Skim Chapters 17-19 for next week
- **Sprint #3 now in progress: Our first ice cream order!**
  - **Individuals**: Time to create your servers and customers, and package it all together into your first multi-serving order!
  - **Duos:** Also create the Emporium itself (perhaps via an emporium class), and start loading and saving data files!
  - **Trios:** Save and load data files, and add some enticing photos; and the manager looks forward to the first reports!
  - **Quattros:** *Lots* of reports, enticing photos, restocking the serving prep counter, and then pay those servers!