# TRAFFIC LIGHT: A PEDAGOGICAL EXPLORATION THROUGH A DESIGN SPACE

*Viera K. Proulx. Jeff Raab, Richard Rasala*
*College of Computer Science*
*Northeastern University*
*Boston, MA 02115*
*617-373-2462*
*vkp@ccs.neu.edu, goon@ccs.neu.edu, rasala@ccs.neu.edu*

**ABSTRACT**

We present the representation of a traffic light as an example of an object that exhibits a rich behavior set and serves as a case study for a number of interesting design issues. We focus on the implementation of the internal state and corresponding control information of the traffic light, and discuss how various important kinds of behavior can be added to this extensible design. We present a number of design and programming exercises to enrich CS1 and CS2 courses.

## 1. INTRODUCTION

Teaching object oriented programming gives us the opportunity and the obligation to explore a wider variety of design issues than those discussed in courses which use imperative languages [2, 3, 5, 6]. To fulfill this obligation, we must provide students with exercises that are understandable, interesting, and present a large range of design issues. The design of a traffic light object is one such exercise [1, 4].

A traffic light is an object that is familiar to all students. Any student can provide a reasonable description of its functionality. With only a small amount of direction, students can quickly begin modifying such a description into an initial design for a traffic light class. Questions such as "How are states encoded and where are they stored?", "What causes transitions between states?", "Who is responsible for the control of timing?", "Who is in charge of the display", etc. immediately come up and shape the design. Students can understand a simple light sequence fairly quickly and enjoy the exercise where they design and implement the entire class. The discussion that follows presents an opportunity to explore design options, make implementation choices, and learn about new topics, such as event handling.

We present our initial design of this class together with the description of the first student project. We then follow with the discussion of additional features, the design problems they introduce, and our suggested implementation of these features. We conclude with suggestions for presenting this design problem in a classroom and as a student project.

## 2. CONTROLS AND VIEWS: A SIMPLE LIGHT SEQUENCE AT A FOUR WAY INTERSECTION

The first question that comes to mind when designing the traffic light class is whether there should be one object with four views, or four separate objects, for a four-way intersection. It is clear that if there are four objects, they will need to communicate with each other so they remain synchronized at all times. On the other hand, having a central control that drives four views seems to be a better design that lends itself to further extensions. The separation of control from view is an important design strategy. While each of the actual traffic light displays needs to be able to report on its state to the external viewers, the control of each state is centralized and synchronized for all lights at the intersection. So the key design decision is to represent the entire operation of the light system at one intersection as one class, though its member data may include an array of four light display objects, one for each direction at the intersection.

Next we consider how to represent the internal states of the object and how to program the transitions between those states. The design discussion focuses on appropriate methods to represent control and change, to maintain state information, to implement timing, and to encapsulate the entire object.

It is safe to assume that for a simple pattern the light is green for $G$ time units, yellow for $Y$ time units and red for $R$ time units, with the constraint that $R = G + Y$. This constraint states that the time for the red light in one direction equals the sum of the green and yellow lights in the other direction. This initial design uses a central clock, which initiates a state change when the appropriate time interval has elapsed. This design should be rejected immediately for two reasons. The first reason is that the constraint R = G + Y is appropriate for traffic moving in two of the four directions, but is ambiguous for traffic moving in the other two directions. A better constraint would be the pair of formulas $R_1 = G_2 + Y_2$ and $R_2 = G_1 + Y_1$, which correctly controls traffic in all directions. Second, changing the timing would necessitate a revision of the class itself. Although values for $G$ and $Y$ could be supplied as parameters, the resulting code would be more difficult for students to understand, and would still be limited to one type of pattern. A discussion of desired extensibility of the object leads us to use a state transition approach.

Once the discussion has proceeded this far, students can be asked to design and implement a simple traffic light class for a four-way intersection with timing given by values for $G_1$, $G_2$, $Y_1$, and $Y_2$. The design may include a graphical display of the changing lights.

## 3. A REAL TRAFFIC LIGHT: REPRESENTING STATES AND TRANSITIONS

We now look at timing and sequence variations, and present suggestions for design and implementation of the control center and its state information, and for programming transitions between states.

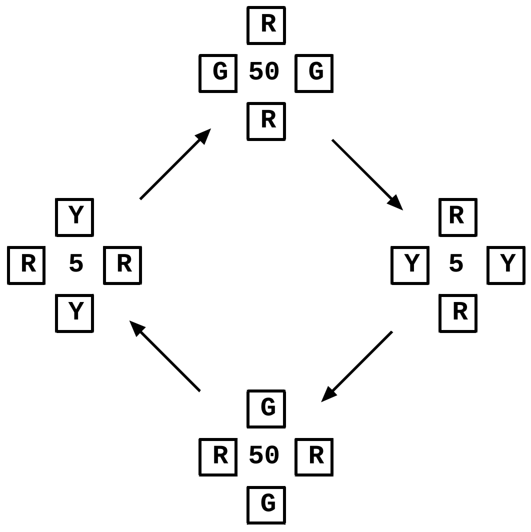
## 3.1 Representing the states and transitions

A discussion of real traffic light functionality makes it clear that the easiest and most versatile method for representing the state of the system is to allow a distinct state for every combination of lights, for each display. If the four displays in the four directions have three lights each, then 12 bits can encode the state of the entire system, at any discrete moment. More bits would be needed for a light with more than four displays, or for displays with more than three lights, such as a left turn signal. To encode the state transitions, each state can also include the interval of time for which it should persist. A circular sequence of states can be implemented either as a linked list, or a dynamic array. We prefer the second method as it has less overhead and is equally efficient in modern programming languages such as Java and C++.

The representation of a simple pattern would require a minimal amount of space, and would be easy for a student to understand. An example pattern follows, for a traffic light with four displays with three lights per display:

| | North | East | South | West | *Time* |
|---|---|---|---|---|---|
| i | RYG | RYG | RYG | RYG | |
| 0 | 100 | 001 | 100 | 001 | 50 |
| 1 | 100 | 010 | 100 | 010 | 5 |
| 2 | 001 | 100 | 001 | 100 | 50 |
| 3 | 010 | 100 | 010 | 100 | 5 |

This corresponds directly with the simple state transition diagram below:



3.2 Adding "delayed green" functionality and other fixed-timing variations

The flexible design of the state representation enables us to add complex timing functionality, such as "delayed green" timing, with no further programming. A delayed green is simply a state where a single display shows a green light, and all other displays show red. Since the state representation maintains an individual light combination for each display, adding this type of functionality requires only the addition of a new state to an existing pattern. For example, state 0 above, which persists for 50 units of time, could be split into two states in order to implement a delayed green. The first of the two states would have only the East display showing a green light, and would have a persistence of some $n < 50$. The second of the two states would have the same display as 0 above, but would have a persistence of $50 - n$ units of time.

In the same way as for delayed green, any type of functionality that uses complex timing can be implemented by adding new states to a sequence. Examples of other complex timing include situations where the duration of the green light in one direction is different than that in the perpendicular direction, where there is a left turn signal, and where a pedestrian walk sign is shown once per cycle.

3.3 Adding a pedestrian "walk" button

In many cases it is desirable to provide time for pedestrians to safely cross an intersection by showing a red light on all displays. It is simple to add such a state to a given sequence, but such an approach would be inefficient; it cannot be guaranteed that there will be pedestrians waiting to cross the intersection every time the light changed into that state. The solution is to allow conditional state transitions, such that a transition could lead to two different states depending on the evaluation of a conditional expression. For this example, a state in the sequence would change to an "all red" walk state only if the walk button was pressed, and otherwise would continue to the next normal state in the sequence.

Implementing this functionality may seem difficult, but it is actually quite simple. By including a boolean value *button* in the state representation, we can designate whether a given state is part of the normal pattern or is a conditional state to be entered only if the button has been pressed. Additional programming is required in this case, as each state change must test to see if the following state in the pattern should be entered or skipped. Because of the extensible design of the state representation, however, only a small amount of new code is necessary.

## 3.4 Sensor driven traffic lights

Some traffic lights use external sensor information to force state changes that are appropriate to the traffic currently moving through the intersection. Although these sensors are similar to pedestrian walk buttons, there are important differences that affect the overall design. First, there are often multiple different sensors for a given intersection, while there is usually just a single walk button. Second, the sensor information is not used to choose between two possible transitions, rather it is used to interrupt a state and cause a transition before the usual time interval has elapsed. Although such functionality is more difficult to implement than the pedestrian walk button, its implementation is still simple enough to provide pedagogical benefits without confusing students.

To implement a sensor driven traffic light, the state representation must be augmented to maintain not only a normal time interval, but also a minimum persistence interval, as well as a list of sensors to listen to. With the addition of this new information, a state can then be programmed to persist at least until the minimum interval has elapsed, and at most until the normal interval has elapsed, with the caveat that a state change can be initiated by any of the registered sensors in the mean time. This implementation makes use of event handling in a straightforward manner, and introduces interrupts as a fundamental and important concept.

## 3.5 Using multiple sequences

Many real world traffic lights do not simply follow a single sequence ad infinitum. Such lights follow different sequences based on local data such as time of day or traffic density, or external data such as holidays. Under the object oriented paradigm, implementation of this kind of functionality is extremely simple. Assuming that a sequence is an encapsulation of an ordering of discrete states, a higher level object could easily be created to encapsulate an ordering of discrete sequences, representing a state transition cycle of sequences. These intersequence transitions could be initiated in the same various ways as intrasequence transitions, e.g. through expiration of time intervals, by pushing external buttons, or by listening to sensor data. In order to avoid awkward, possibly dangerous, intersequence transitions, a constraint could be added to ensure that such transitions may only occur after a sequence has completed its internal state transition cycle.

This idea could be extended to any desired level. In order to accommodate holiday traffic patterns, or seasonal differences in roadway conditions, further levels could be fashioned with very little programming effort. A traffic light could maintain a complex, multiple sequence schedule using basically the same tools needed to maintain a simple two minute light sequence.

## 3.6 Adding emergency capabilities

Regardless of the level of encapsulation used to determine the overall schedule for a traffic light, it may be necessary to provide the traffic light with a state that can be triggered during an emergency situation, such as when a fire station is responding to an alarm. This functionality could be added at the sequence level, in the same way as the passenger walk button, by adding states that are skipped by default. Such a solution would be inefficient, as many such states would have to be added in order to assure that the emergency state could be entered at any point, in any sequence, in any time of day, etc.. It is a more intuitive design to provide the emergency state as an alternative to the entire schedule, which can be entered or left by the push of a button.

The implementation of an emergency state would be similar to the implementation of sensor lights, in that it would be interrupt driven, but the emergency state implementation is more difficult in other respects. In order to eliminate the possibility of dangerous state transitions between sequences, a constraint was added which allowed sequence transitions to occur only at the end of a sequence's cycle; this constraint may cause the transition into the emergency state to take unacceptably long. It is not safe, however, to immediately enter the emergency state.

By adding yet another boolean value *emergency* to the lowest level state representation, we can mark states as safe or unsafe to change to the emergency state. Typically, states where one or more displays show green would be marked unsafe, and states where all displays show only yellow or red would be marked safe. In the case of an emergency, the light would immediately move to the next safe state in the current pattern. When the time interval for the safe state has elapsed, the light would change to the emergency state, which would persist until the emergency button was released. Upon leaving the emergency state, it would be desirable to have the light change to the state that follows the safe state that preceded the emergency. By treating the emergency state as a subroutine, an obvious implementation possibility becomes clear: the current state at the time of the emergency could be stored on a stack, and entered upon return from the emergency call.

An alternative way of assuring safe transition into the emergency state is to encode several emergency transition patterns and label each state with an index that identifies its appropriate emergency transition sequence. Upon receiving the emergency signal, the light would follow the appropriate transition sequence, and then enter the emergency state. The transition back to the normal sequence may require its own transition sequence as well.

## 4. PEDAGOGICAL CONSIDERATIONS

## 4.1 Problems and Projects for Students

There are several ways that this problem can be used in a course. The design and implementation of the basic light class with fixed timing and display is a nice exercise. The class discussion provides an opportunity to practice the design of state transition diagrams, and introduces the tradeoffs between the different methods for representing the states and implementing the transitions in a program. The introduction of the pedestrian push button gives the context for discussing the handling of events and external interrupts. The overall discussion of the design explains the importance of making the initial design extensible and of properly encapsulating the components of the system. When discussing how to implement the displays and the interactions of the traffic light with the traffic on the road, the idea of separating the control from the view becomes prominent. Another interesting issue is the control of timing. A simple simulation would be controlled by a centralized timer that updates the state of the system (both the light and the relevant traffic) once at every discrete time interval. However, for a more realistic simulation one needs to give the traffic light its own way of checking the system clock and updating itself as an independent thread with the highest priority. The traffic in each direction can then be implemented as four separate threads, running with somewhat lower priority than the light thread.

### 4.2 Our Experiences

We used the assignment to design a simple traffic light in out Object Oriented Design course. To focus student's attention on the design of the controls and interactions with the external views, we provided the code for a graphical display of the lights. The assignment came fairly early in the course, when students had very little experience with designing classes and understanding the proper encapsulations. The exercise provided an excellent example for explaining proper class design and encapsulation. The better students added extra features for additional credit and greatly enjoyed watching the actual simulation of traffic. Students clearly enjoyed the project and produced many interesting questions and design variations.

## 5. ACKNOWLEDGEMENTS

Partial support for this work has been provided by the National Science Foundation Leadership in Laboratory Development, award #DUE-9659552 and by the Microsoft Corporation.

## 6. REFERENCES

[1] Berman, A. M., Data Structures via C++: Objects by Evolution, Oxford University Press, New York, 1997, pp. 275-277.

[2] Holland, S., Griffith, R., Woodman, M., Avoiding Object Misconceptions, *ACM SIGCSE Bulletin* 29 (1), 131-134, 1997.

[3] Parlante, N., Teaching With Object-Oriented Libraries, *ACM SIGCSE Bulletin* 29 (1), 140-144, 1997.

[4] Proulx, V. K., Traffic Simulation: A Case Study for Teaching Object Oriented Design, *ACM SIGCSE Bulletin* 30 (1), 48-52, 1998.

[5] Rasala, R.. Function Objects, Function Templates, and Passage by Behavior in C++. *ACM SIGCSE Bulletin* 29 (1), 35-38, 1997.

[6] Schoenfeld, D. A., Object-oriented Design and Programming: An Eiffel, C++ and Java Course for C Programmers, *ACM SIGCSE Bulletin* 29 (1), 135-139, 1997.