01.10.2014

Christian Janeczek

5AHITT

INDINF-01

THE ART OF STATE MACHINES

Contents

[Task Description 2](#_Toc399934344)

[Design consideration 3](#_Toc399934345)

[REQUIREMENT: State-Centric State Machine Solution 3](#_Toc399934346)

[Advantages 3](#_Toc399934347)

[Disadvantages 3](#_Toc399934348)

[REQUIREMENT: State-Centric State Machine Solution + Hidden Transitions 4](#_Toc399934349)

[Advantages 4](#_Toc399934350)

[Disadvantages 4](#_Toc399934351)

[REQUIREMENT: Event-Centric State Machine Solution 5](#_Toc399934352)

[Advantages 5](#_Toc399934353)

[Disadvantages 5](#_Toc399934354)

[REQUIREMENT: State Pattern State-Machine Solution 6](#_Toc399934355)

[REQUIREMENT: Table Driven State-Machine Solution 7](#_Toc399934356)

[Apportionment of work with effort estimation 8](#_Toc399934357)

[Final Time Apportionment 9](#_Toc399934358)

[Task execution 10](#_Toc399934359)

[Test report 11](#_Toc399934360)

[Bibliography 12](#_Toc399934361)

# Task Description

Implement a component based C-Programm to show the difference of the 5 types of state machines presented in the book of Mrs. Elicia White "Making Embedded Systems" with traffic light system we discussed in the lesson. To test your implementation you can use simple output functions (e.g. fprintf), but be prepared to implement it also on hardware (GPIO with Leds, Timers, etc.).

Don't forget to document the differences (advantages/disadvantages) in your protocol.

# Design consideration

## REQUIREMENT: State-Centric State Machine Solution

As described in “*Making Embedded Systems*” by Mrs. Elicia White, the State-Centric State Machine Solution is plain and simple. In this case the State is being checked, and accordingly the command is being set. For an example I have got the following code snippet:

**case (state):**

**if event valid for this state**

**handle event**

**prepare for new state**

**set new state**

This means, that the state centric state-machine can change its context, and so move to a new state. This means, that each state needs to know about its sibling states, which might be a problem (see Disadvantages).

### Advantages

* Simple way of building the state machine
* Well organized

### Disadvantages

* The state-centric solution has got to know about each sibling and its states, which means, that they may not be able to work without knowing about each other. We once talked about this loosely-coupled way of engineering in our Software Engineering lesson.
* Completely static/no signs of dynamicity
* Each state HAS to know: “HOW and WHEN do I need to trigger each of the other states?”
* The higher the state count, the lower the efficiency

## REQUIREMENT: State-Centric State Machine Solution + Hidden Transitions

The other way to check by the state, is the State-Centric State Machine with Hidden Transitions. In this case we separate the transition information from the state machine itself and theoretically this model is better than the previous. Using this very method, you can separate the actions taken in each state FROM the specific state transitions:

**case (state):**

**make sure current state is actively doing what it needs**

**if event valid for this state**

**call next state function**

For our traffic-light example it would look like this (green light state):

**case (green light):**

**if (green light not on) turn on green light**

**if (event is stop)**

**turn off green light**

**call next state function**

**break;**

When a change occurs, the nextState function will be called:

**next state function:**

**switch (state) {**

**case (green light):**

**set state to yellow light**

**break;**

**case (yellow light):**

**set state to red light**

**break;**

**case (red light):**

**set state to green light**

**break;**

### Advantages

* State transitions are independent of the event -> good for implementation
* More encapsulation
* Fewer dependencies
* Much more efficient

### Disadvantages

* Calling the next state function would mean, that in this case, the REPEAT state is not going to be called, ever() as well as the YELLOW\_BLINKING state.
* Strict order of states, without any personal influence by commands(which could also be an advantage, in specific cases)
* Sometimes you are in need of inter-state dependencies, so this anatomy of a state-machine will be utter useless.

## REQUIREMENT: Event-Centric State Machine Solution

The implementation of the Event-Centric State Machine lets the events take the control of the flow, so that each event has an associated set of conditionals:

**case (event):**

**if state transition for this event**

**go to new state**

For example:

**switch (event)**

**case (stop):**

**if (state is green light)**

**turn off green light**

**go to next state**

**// else do nothing**

**break;**

### Advantages

* In some cases it is cleaner to associate the state machines with events.
* More efficient programming(less lines of code)

### Disadvantages

* The event-centric state machine makes it difficult to do housekeeping activities, for example: For checking specific timeouts, you may have to create an event for each housekeeping case.
* Completely static
* (See state-centric)

## REQUIREMENT: State Pattern State-Machine Solution

In the State Pattern State-Machine Solution is described as follows:

Each and every state has got to be defined as some kind of object. We can use structures to create objects in C. The specific object has methods, uses for handling each event. We will need the following functions/methods:

* A method which is called, when the state is entered(LIGHT IS BEING TURNED ON)
* A method which is called, when leaving the state(LIGHT IS BEING TURNED OFF)
* Housekeeping methods
* Methods for starting as well as ending specific events

Here is some pseudo code taken from the book:

**class Context {**

**class State Red, Yellow, Green;**

**class State Current;**

**constructor:**

**Current = Red;**

**Current.Enter();**

**destructor:**

**Current.Exit();**

**Go:**

**if (Current.Go() indicates a state change)**

**NextState();**

**Stop:**

**if (Current.Stop() indicates a state change)**

**NextState();**

**Housekeeping:**

**if (Current.Housekeeping() indicates a state change)**

**NextState();**

**NextState:**

**Current.Exit();**

**if (Current is Red) Current = Green;**

**if (Current is Yellow) Current = Red;**

**if (Current is Green) Current = Yellow;**

**Current.Enter();**

**}**

### Advantages

* Dynamic
* Object oriented
* Encapsulated
* One method for handling each event

### Disadvantages

* Much more complex than the previous solutions
* Not as easy to understand
* Very hard to implement in C(there are no objects in C, only structs)

## REQUIREMENT: Table Driven State-Machine Solution

### Advantages

* Great solution for complex State Machines, because it’s re-useable
* Even though there is much data, nice overview

### Disadvantages

* Easy to make mistakes inside the table

# Apportionment of work with effort estimation

|  |  |  |  |
| --- | --- | --- | --- |
| **Competent person(s)** | **Task** | **Description** | **Estimated time in h** |
| Janeczek | Design consideration | How do I realize the state machines in C? | 2 |
| Janeczek | State-Centric Solution | Implementing the State-Centric solution | 3 |
| Janeczek | State-Centric Hidden Solution | Implementing the State-Centric solution with Hidden Transitions | 2 |
| Janeczek | Event-Centric Solution | Implementing the Event-Centric solution | 2 |
| Janeczek | State Pattern Solution | Implementing the State Pattern Solution | 3 |

# Final Time Apportionment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Competent person(s)** | **Task** | **Estimated time in h** | **Actual time in h** | **Comment** |
| Janeczek | Design consideration | 2 | 3 | I need to get better at Software-Planning! |
| Janeczek | State-Centric Solution | 3 | 4 | It took me quite some time to get comfortable working in C |
| Janeczek | State-Centric Hidden Solution | 2 | 3 | Hidden Transactions required some newly defined methods, which consumed quite an amount of time |
| Janeczek | Event-Centric Solution | 2 | 2 | „Same“ as the State-centric, just with events |
| Janeczek | State Pattern Solution | 3 | ? | ?s |

# Task execution

1. Brainstorming with other people: “How do I realize all this in C?”
2. Installing all needed packages, for example: clang, sublime text editor, etc.
3. Getting started with C again. Some warming-up with older tasks.
4. Writing the state-centric solution. Took more time than expected, I got quite sloppy in C.
5. Got more familiar with C and wrote the state-centric hidden solution in an adequate amount of time.
6. The event-centric solution was an advanced solution of the already written state-centric solution.
7. From what I have heard, the State Pattern solution is going to be “zach”.
8. I already know how to define data types in C, but object-oriented programming will be more difficult than expected.
9. I need to learn, how to efficiently build my program. Component-based programming is a must-do, because code duplication is more than just affecting the style of my code.

# Test report

# Bibliography

|  |
| --- |
| **Title:** How to define an enumerated type in C?  **Author:** Stéphane Gimenez  **Source:** <http://stackoverflow.com/questions/1102542/how-to-define-an-enumerated-type-enum-in-c>  **Last modified:** 2011/09/13  **Last seen:** 2014/09/30 |
| **Title:** Structs in C  **Author:** P. Böhme  **Source:** <http://www2.informatik.uni-halle.de/lehre/c/c_struct.html>  **Last modified:** 1996/02/15  **Last seen:** 2014/10/01 |
| **Title:** Creating your own header file in C  **Author:** Oliver Charlesworth  **Source:** <http://stackoverflow.com/questions/7109964/creating-your-own-header-file-in-c>  **Last modified:** 2011/09/18  **Last seen:** 2014/10/01 |
| **Title:** Makefiles  **Author:** C-HowTo  **Source:** <http://www.c-howto.de/tutorial-makefiles.html>  **Last modified:** ?  **Last seen:** 2014/10/01 |
|  |