EVENT DRIVEN PROGRAMMING

- Embedded Real Time Systems
- Ron Barker

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN · FH
MÜNCHEN

Review of Last Week

STATE MACHINE IMPLEMENTATIONS

State Machines and OO

- State Machine Implementions Ought Reflect the mapping of 3 Basic OO Concepts:
  - Encaplusation
  - Inheritance
  - Reuse of Resources:
    - Models
    - Code
- Do this is our primary goal

State Machine Basics

- The basis of any real time event driven state machine:
  - Event Capture Control
  - Event Dispatch
  - Event Processing
- State Machines are the data objects that provide these facilities.
- State Machines are inherently data variables abstracting event instances

Standard Implementations

- There are certain "standard" implementation methods that are mainly applicable to traditional non-hierarchical FSMs.
  - Nested Switch
  - State Tables
  - Object Oriented Design Patterns

Bomb1 Overview

- Implementation Attributes
  - Wholly in “c” – Note C++ type Constructor
  - Introduces Event Abstraction
    - Sig defined as Event Object “event”
    - Time event Object “inherits” event and adds time
    - Bomb Object IS THE State machine
      - State + extended variables
  - Introduces “indirect” Execution Control
    - -single inheritance charactistic of “c”
  - Dispatcher - knows only events – states –NO HISTORY

Bomb1 Analysis

- Analyze according to formal definition
- What are the States?
- What are the Events?
- What are the Transitions?
- Where is the Mapping function
- Where are the Problem ?

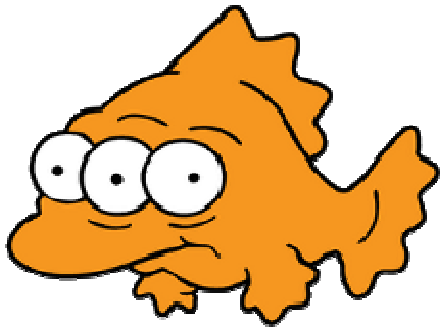
PRACTICAL EXERCISE
BLINKY : SWITCH CASE MODEL

Blinky Reloaded

- Rewrite Blinky as a State Machine
  - Nested Switched Case
  - Use Bomb1 as guide

Blinky: Steps toward a FSM

- Analyse Code to “reverse engineer” a requirements doc
- Create a BSP for Blinky
- Use the Requirements Doc to:
  - Identify Set of States
  - Identify Set of Events
  - Identify Initial State
  - Describe a Mapping function



Blinky: Requirements

- What must Blinky Do?

  When the MCB2300's poti wheel is rotated, it generates a specific voltage that is read from the systems analog-digital unit. The program is required to obtain the digital value via the units ISR and display this value on 3 different output devices. The value must be scaled to be displayed proportionally in accordance with the range of display points on the individual devices

Blinky: Requirements

- How does Blinky Do it?

LED: Values Scaled to illuminate a set of Leds such that the number of leds switched on/off correlates to the scaled value

LCD: Display the scaled value as a bar graph on the LCD Display such that the number of LCD segments correlates to the scaled value

V24: Display periodically the numerical value in hex Format as output to a serial Terminal.

The LED and LCD display are to be synchronized such that they display simultaneously the same scaled value.

Blinky the State Machine

- State Machine Design
  - State Machine Maintains History
  - Aggregate Data Object with:
    - Scalar variable “the state”
    - N Scalar “extended state variables”
  - Mapping Function
    - No History, No Globals
    - Establish Transition Rules
    - Generate Internal Signals

Blinky: BSP

- Blinky Board Support Package
  - BSP Isolates App from HW

Isolate specific HW components and implement each in separate compilation units. Use existing code.

- Serial
- LED
- LCD
- Timer
- AD Converter

Design and Implement Functions to

- Retrieve HW Signals
- Initialize HW
- Register Events

Consequences of Nested Switch

- Pros:
  - Simple – both states and events are ENUMs
  - Small (usually) memory footprint – single state variable
- Cons:
  - Elements are coded specifically for problem – no reuse
  - Dedicated "monolithic" coding –prone to code creep
  - Performance of 2nd-level switch determines dispatching
  - Degrades as number of cases increases
  - Tends to promote duplicate code in states
  - Not hierarchical

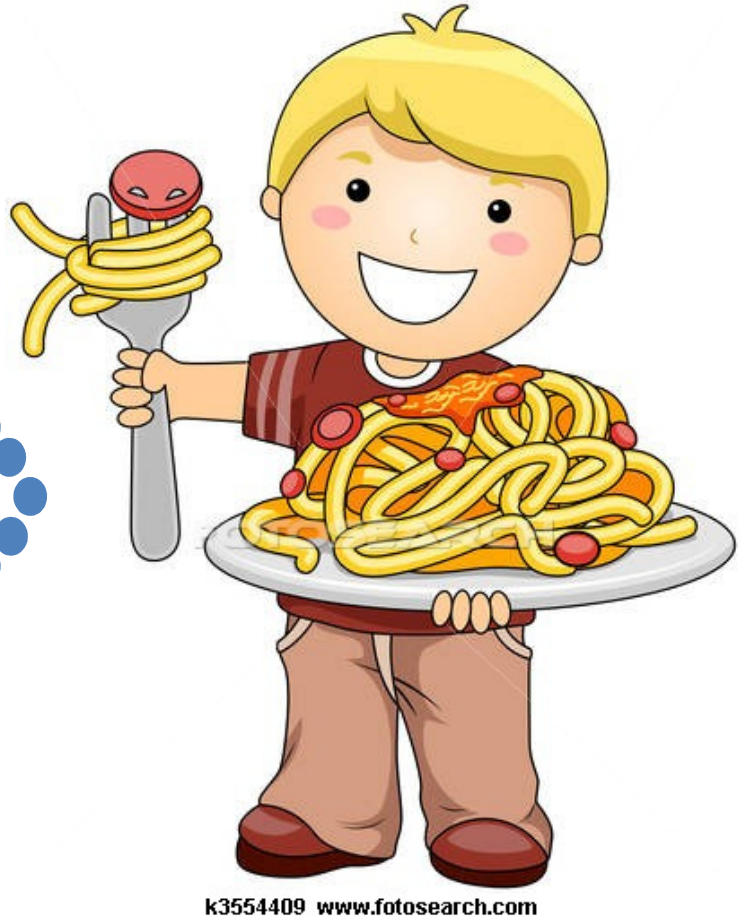
..and eventually..

Leads to architectural decay

begets convolution

Leads to

Spag hetti code



HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN · FH MÜNCHEN

Standard State Machine Implementations

GENERIC STATE TABLES

Generic State Tables

- Commonly Accepted "state of art" FSM Implementation
- State Tables are basically Truth Tables in which each vector is the address of a state handler.
- State Tables are represented asTwo Dimensional Arrays
  - Events in the horizontal axis
  - States in the Vertical Axis
- Numeric Values represent States / Event
- State Transition is implemented in a dispatch function:
  - Calculates the offset of state handler in table
  - Calls via function pointer that state handles

Generic State Tables

	Events			
	UP	DOWN	ARM	TICK
States				
Setting	Setting_UP, setting	Setting_DOWN setting	Setting_ARM(), timing	Empty(), setting
Timing	Timing_UP, timing	Timing_DOWN(), timing	Timing_ARM(), setting	timing_TICK() timing

Practical Exercise

- Bomb 2 – Check Out in Redmine

- Compile – Run
- Analyse

Practical Exercise

- Bomb 2
- Implements 2 Dimensional Array
  - 4 Events – UP DOWN TICK ARM
  - 2 States – Set – ARM
  - 8 Entries –
- Introduces the concept of „opaque pointer“
- Opaque pointer is basis of late binding / Dynamic Dispatch
-

Generic State Tables-The Good

- Maps state table representation directly to handlers
- Provides relatively good performance for event dispatching (single instance of dispatching)
- Event Process promotes reuse of code
- Tables can be stored in ROM to accommodate resource constrained devices

Generic State Tables-The Bad…

- Require States and Signals as ENUMS
- States and Signals – indices into state table .. must be contiguous and start with 0
- Initialisation of State Table is difficult to maintain:
- Adding a new state requires add and initialisation of a new row