CMPE 202

"Classical" Procedural Programming
vs.
Object-Oriented Programming

The Gumball Machine

https://github.com/paulnguyen/cmpe202/blob/master/gumball/gumball-c-ver1/

main.c

```c
#include <stdio.h>

int has_quarter = 0 ;
int num_gumballs = 1 ;

void insert_quarter( int coin )
{
    if ( coin == 25 )
       has_quarter = 1 ;
    else
        has_quarter = 0 ;
}

void eject_quarter()
{
    if ( has_quarter )
    {
       has_quarter = 0 ;
       printf( "Your quarter as been ejected!\n" ) ;
    }
    else
    {
        printf( "No quarter to eject.\n" ) ;
    }
}

void turn_crank()
{
    if ( has_quarter )
    {
        if ( num_gumballs > 0 )
        {
            num_gumballs-- ;
            has_quarter = 0 ;
            printf( "Thanks for your quarter.  Gumball Ejected!\n" ) ;
        }
        else
        {
            printf( "No More Gumballs!  Sorry, can't return your quarter.\n" ) ;
        }
    }
    else
    {
        printf( "Please insert a quarter\n" ) ;
    }
}


int main(int argc, char **argv)
{
    printf("Simple Gumball Machine - Version 1\n");
    insert_quarter( 25 ) ;
    turn_crank() ;
    insert_quarter( 25 ) ;
    turn_crank() ;
    insert_quarter( 10 ) ;
    turn_crank() ;
    eject_quarter() ;
    return 0;
}
```

1. Start with this code (C Option)

2. Compile and Run it (See Makefile)

NOTE: This code is in "C". If you prefer to work in "Java", use the next slide as your starting point.

https://github.com/paulnguyen/cmpe202/tree/master/gumball/gumball-java-typical

GumballMachine.java

```java

public class GumballMachine
{

    private int num_gumballs;
    private boolean has_quarter;

    public GumballMachine( int size )
    {
        // initialise instance variables
        this.num_gumballs = size;
        this.has_quarter = false;
    }

    public void insertQuarter(int coin)
    {
        if ( coin == 25 )
            this.has_quarter = true ;
        else
            this.has_quarter = false ;
    }

    public void turnCrank()
    {
        if ( this.has_quarter )
        {
            if ( this.num_gumballs > 0 )
            {
                this.num_gumballs-- ;
                this.has_quarter = false ;
                System.out.println( "Thanks for your quarter.  Gumball Ejected!" ) ;
            }
            else
            {
                System.out.println( "No More Gumballs!  Sorry, can't return your quarter." ) ;
            }
        }
        else
        {
            System.out.println( "Please insert a quarter" ) ;
        }
    }
}

```

1. Start with this code (Java Option)

2. Compile and Run it (See Makefile)

NOTE: This code is in "Java". If you prefer to work in "C", use the previous slide as your starting point.

Cost: 25 cents
Accepts only Quarters

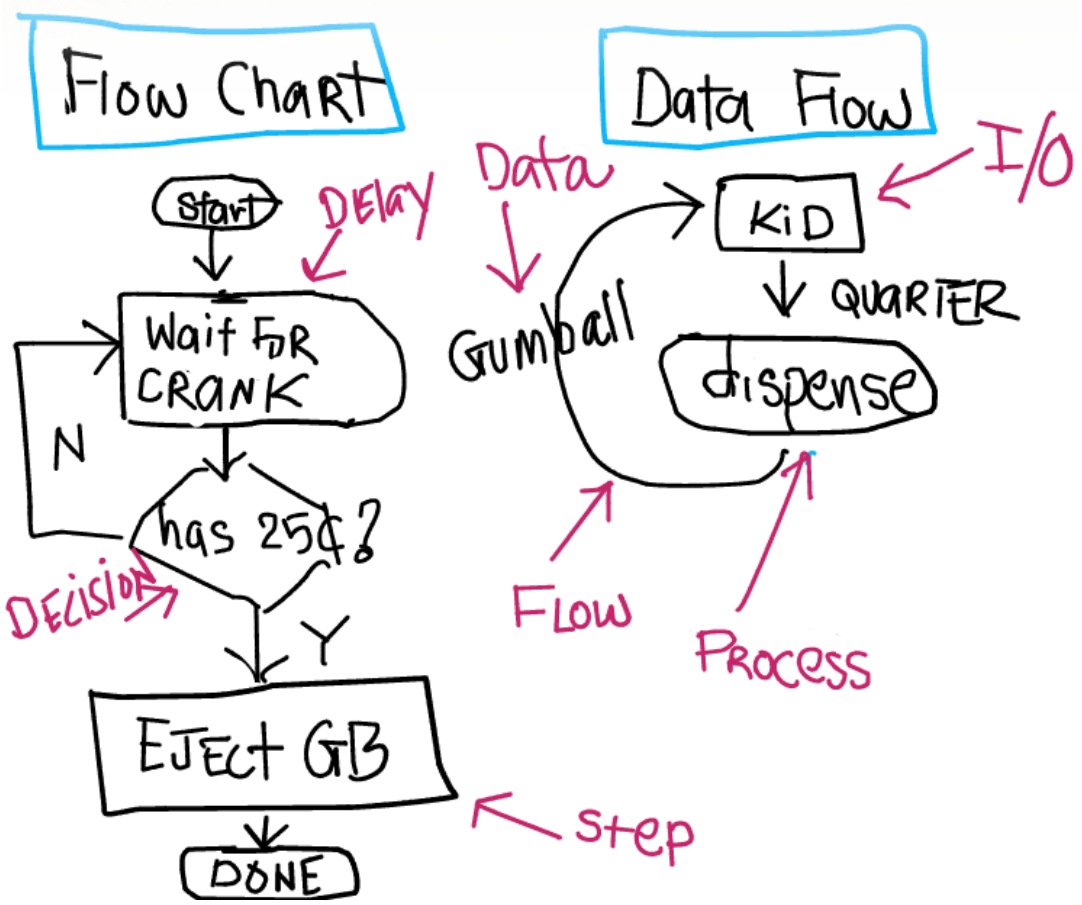
Cost: 50 cents
Accepts Two Quarters

Cost: 50 cents
Accepts All Coins
(i.e. nickels, dimes, & quarters)

1. *Now, Modify your code to support all three of the Machine Types above!*

Basic Requirements:

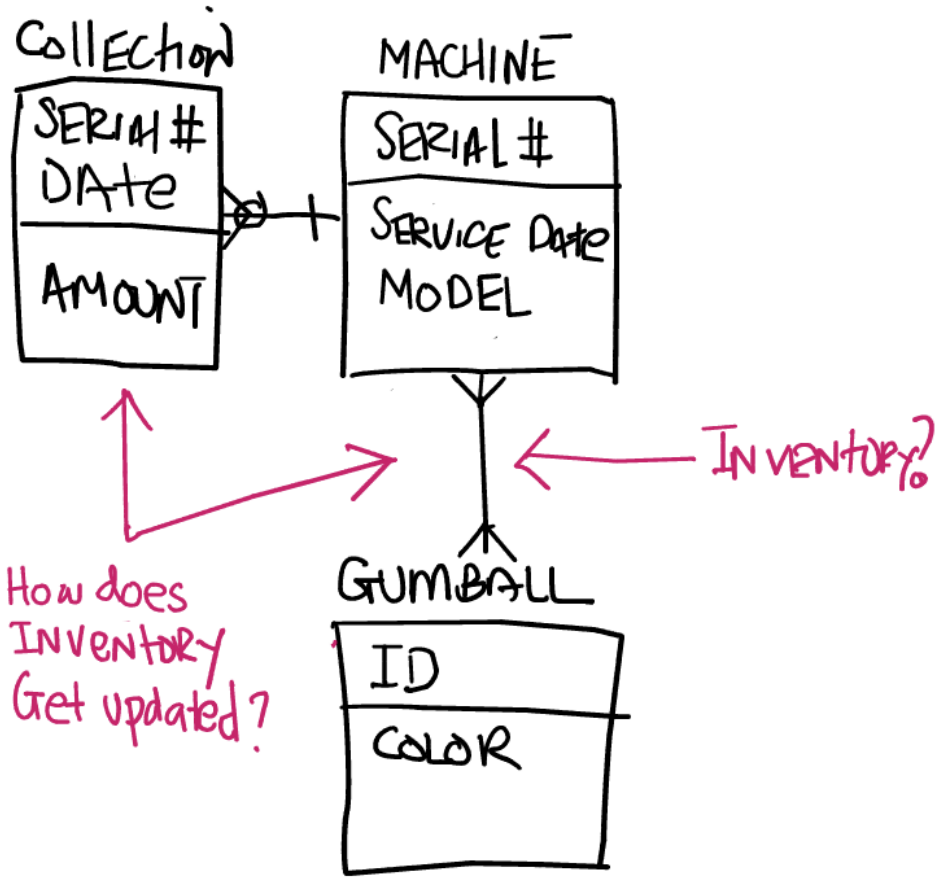
1. Insert a quarter, crank, get a gumball
2. If no more gumball, lose the quarter



	Terminator	A start or stop point in a process
	Process	A computation step or an operation
	Decision	Decision making or branching
	Delay	A waiting period
	Data I/O	Input or output operation

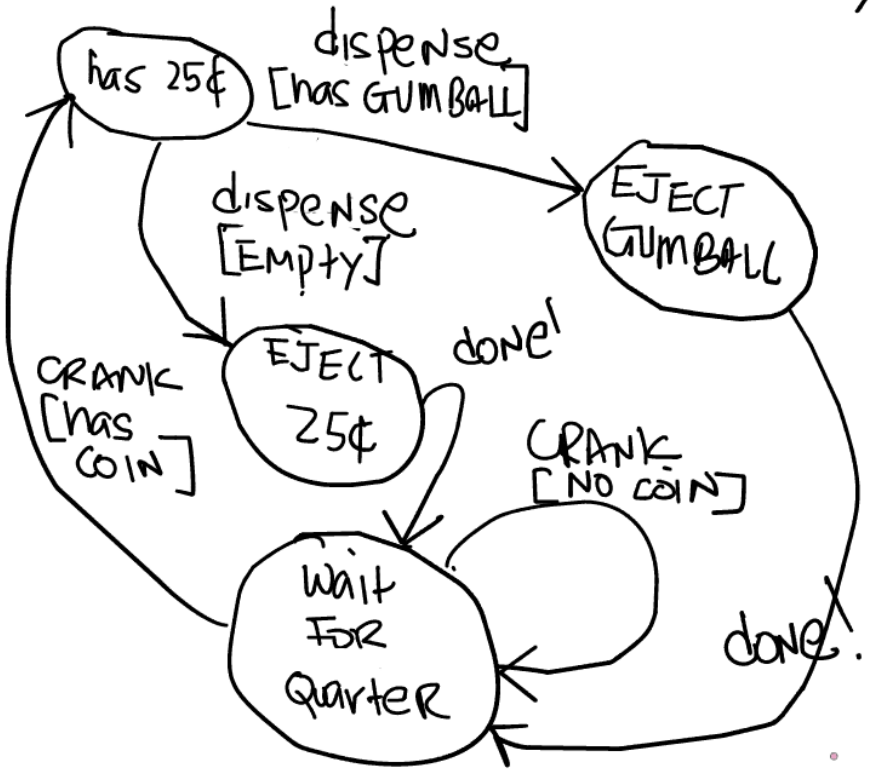
DOUBLE SQUARE	Source or destination of data
arrow	Flow of data
rounded rectangle	Process that transforms a flow of data
OPEN-ENDED RECTANGLE	Store of data

Data Model



MACHINE LOGIC AS FSM.

(Finite State Machine)

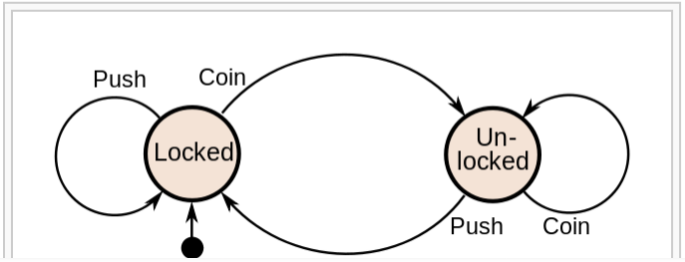


A "NICE" GUMBALL MACHINE!

State Machines (FSM) as Transition Tables

Example: coin-operated turnstile [edit]

An example of a very simple mechanism that can be modeled by a state machine is a turnstile.[2][3] A turnstile, used to control access to subways and amusement park rides, is a gate with three rotating arms at waist height, one across the entryway. Initially the arms are locked, blocking the entry, preventing patrons from passing through. Depositing a coin or token in a slot on the turnstile unlocks the arms, allowing a single customer to push through. After the customer passes through, the arms are locked again until another coin is inserted.



Considered as a state machine, the turnstile has two states: ***Locked*** and ***Unlocked***.[2] There are two inputs that affect its state: putting a coin in the slot (***coin***) and pushing the arm (***push***). In the locked state, pushing on the arm has no effect; no matter how m
Putting a coin in – that is, giving the machi
unlocked state, putting additional coins in
change the state. However, a customer pu

The turnstile state machine can be represe
input

Current State	Input	Next State	
Locked	coin	Unlocked	Unlo
	push	Locked	Non
Unlocked	coin	Unlocked	Non
	push	Locked	Whe

It can also be represented by a directed gr
transitions from one state to another. Each
coin input in the ***Unlocked*** state) are repr

State/Event table [edit]

Several state transition table types are used. The most common representation is shown below: the combination of current state (e.g. B) and input (e.g. Y) shows the next state (e.g. C). The complete action's information is not directly described in the table and can only be added using footnotes. A FSM definition including the full actions information is possible using state tables (see also virtual finite-state machine).

State transition table

Input \ Current state	State A	State B	State C
Input X	…	…	…
Input Y	…	State C	…
Input Z	…	…	…

UML state machines [edit]

The Unified Modeling Language has a notation for describing state machines. UML state machines overcome the limitations of traditional finite state machines while retaining their main benefits. UML state machines introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines. They support actions that depend on both the state of the system and the triggering event, as in Mealy machines, as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines.

https://en.wikipedia.org/wiki/Finite-state_machine

Implementation in C (version 1) - Typical Solution

main.c

```c
#include <stdio.h>

int has_quarter = 0 ;
int num_gumballs = 1 ;

void insert_quarter( int coin )
{
    if ( coin == 25 )
       has_quarter = 1 ;
    else
        has_quarter = 0 ;
}

void turn_crank()
{
    if ( has_quarter )
    {
        if ( num_gumballs > 0 )
        {
            num_gumballs-- ;
            has_quarter = 0 ;
            printf( "Thanks for your quarter.  Gumball Ejected!\n" ) ;
        }
        else
        {
            printf( "No More Gumballs!  Sorry, can't return your quarter.\n" ) ;
        }
    }
    else
    {
        printf( "Please insert a quarter\n" ) ;
    }
}

int main(int argc, char **argv)
{
    printf("Simple Gumball Machine - Version 1\n");
    insert_quarter( 25 ) ;
    turn_crank() ;
    insert_quarter( 25 ) ;
    turn_crank() ;
    insert_quarter( 10 ) ;
    turn_crank() ;
    return 0;
}
```

Problems:

1. Global variables are not a good practice! Why?
2. Can only have one gumball machine at a time.
3. What about a 50¢ gumball machine?
4. How about the "nice" gumball machine that returns coins?

Implementation in C (version 2) - Abstract Data Type

gumball.h

```c
typedef struct
{
    int num_gumballs ;
    int has_quarter ;
} GUMBALL ;

extern void init_gumball( GUMBALL *ptr, int size ) ;
extern void turn_crank( GUMBALL *ptr );
extern void insert_quarter( GUMBALL *ptr, int coin );
```

*main.c

```c
#include <stdio.h>
#include "gumball.h"

int main(int argc, char **argv)
{
    GUMBALL m1[1] ;
    GUMBALL m2[1] ;

    /* init gumball machines */
    init_gumball( m1, 1 ) ;
    init_gumball( m2, 10 ) ;

    printf("Simple Gumball Machine - Version 2\n");

    insert_quarter( m1, 25 ) ;
    turn_crank( m1 ) ;
    insert_quarter( m1, 25 ) ;
    turn_crank( m1 ) ;
    insert_quarter( m1, 10 ) ;
    turn_crank( m1 ) ;

    insert_quarter( m2, 25 ) ;
    turn_crank( m2 ) ;
    insert_quarter( m2, 25 ) ;
    turn_crank( m2 ) ;
    insert_quarter( m1, 10 ) ;
    turn_crank( m2 ) ;

    return 0;
}
```

gumball.c

```c
#include <stdio.h>
#include "gumball.h"

void init_gumball( GUMBALL *ptr, int size )
{
    ptr->num_gumballs = size ;
    ptr->has_quarter = 0 ;
}

void turn_crank( GUMBALL *ptr )
{
    if ( ptr->has_quarter )
    {
        if ( ptr->num_gumballs > 0 )
        {
            ptr->num_gumballs-- ;
            ptr->has_quarter = 0 ;
            printf( "Thanks for your quarter.  Gumball Ejected!\n" ) ;
        }
        else
        {
            printf( "No More Gumballs!  Sorry, can't return your quarter.\n" ) ;
        }
    }
    else
    {
        printf( "Please insert a quarter\n" ) ;
    }
}

void insert_quarter( GUMBALL *ptr, int coin )
{
    if ( coin == 25 )
        ptr->has_quarter = 1 ;
    else
        ptr->has_quarter = 0 ;
}
```

ADT --> think of the GB Machine in terms of its operations. (insert quarter, turn crank)
Data Encapsulation --> hide implementation details. Not supported in C Language

Implementation in Java (BlueJ)

```
1
2  public class GumballMachine
3  {
4
5      private int num_gumballs;
6      private boolean has_quarter;
7
8      public GumballMachine( int size )
9      {
10         // initialise instance variables
11         this.num_gumballs = size;
12         this.has_quarter = false;
13     }
14
15     public void insertQuarter(int coin)
16     {
17         if ( coin == 25 )
18             this.has_quarter = true ;
19         else
20             this.has_quarter = false ;
21     }
22
23     public void turnCrank()
24     {
25         if ( this.has_quarter )
26         {
27             if ( this.num_gumballs > 0 )
28             {
29                 this.num_gumballs-- ;
30                 this.has_quarter = false ;
31                 System.out.println( "Thanks for your quarter.  Gumball Ejected!" ) ;
32             }
33             else
34             {
35                 System.out.println( "No More Gumballs!  Sorry, can't return your quarter." ) ;
36             }
37         }
38         else
39         {
40             System.out.println( "Please insert a quarter" ) ;
41         }
42     }
43 }
44
```

Native support for Encapsulation in OO Languages (like Java)

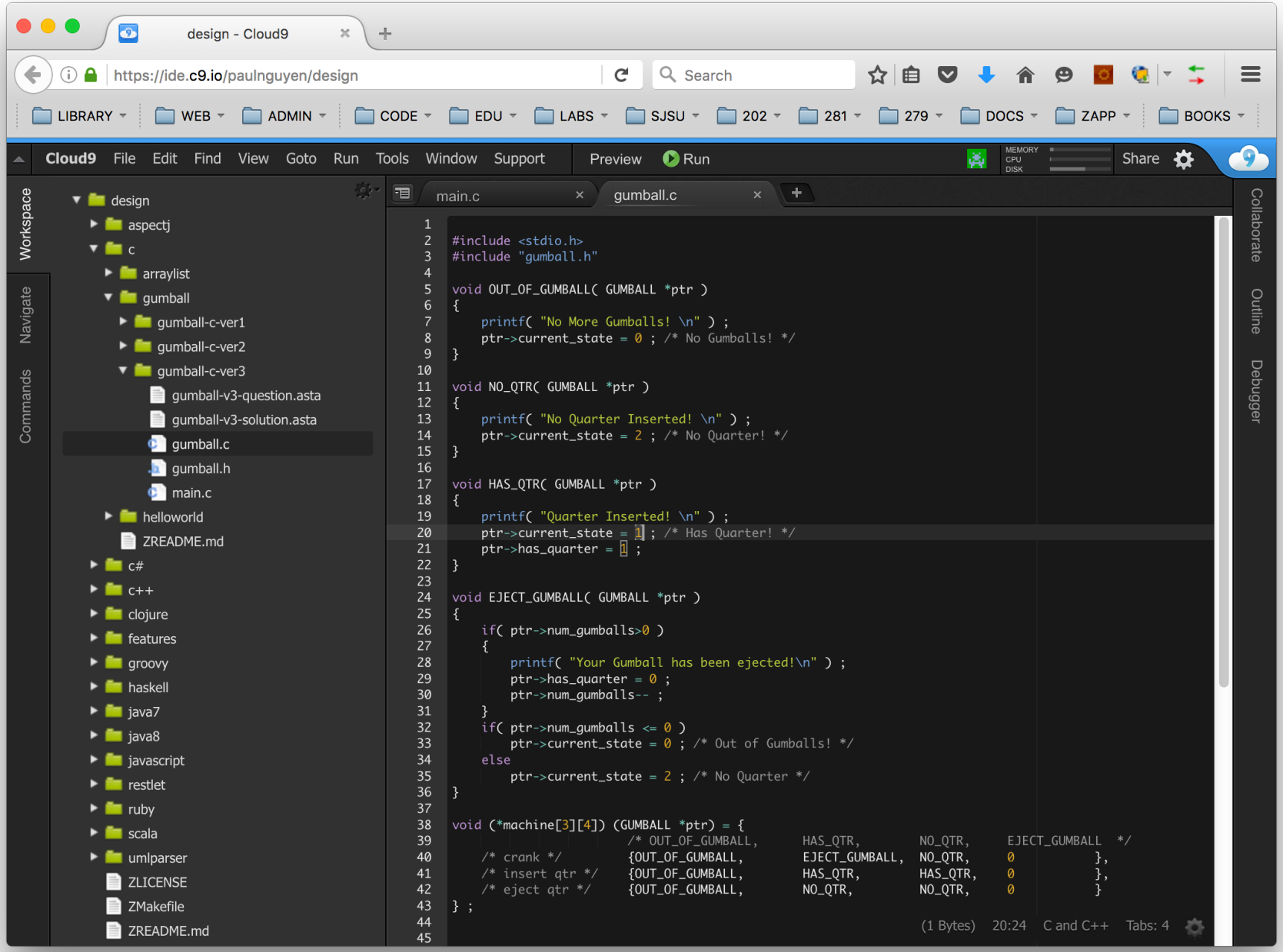
Dealing with Product Variants

Cost: 25 cents
Accepts only Quarters

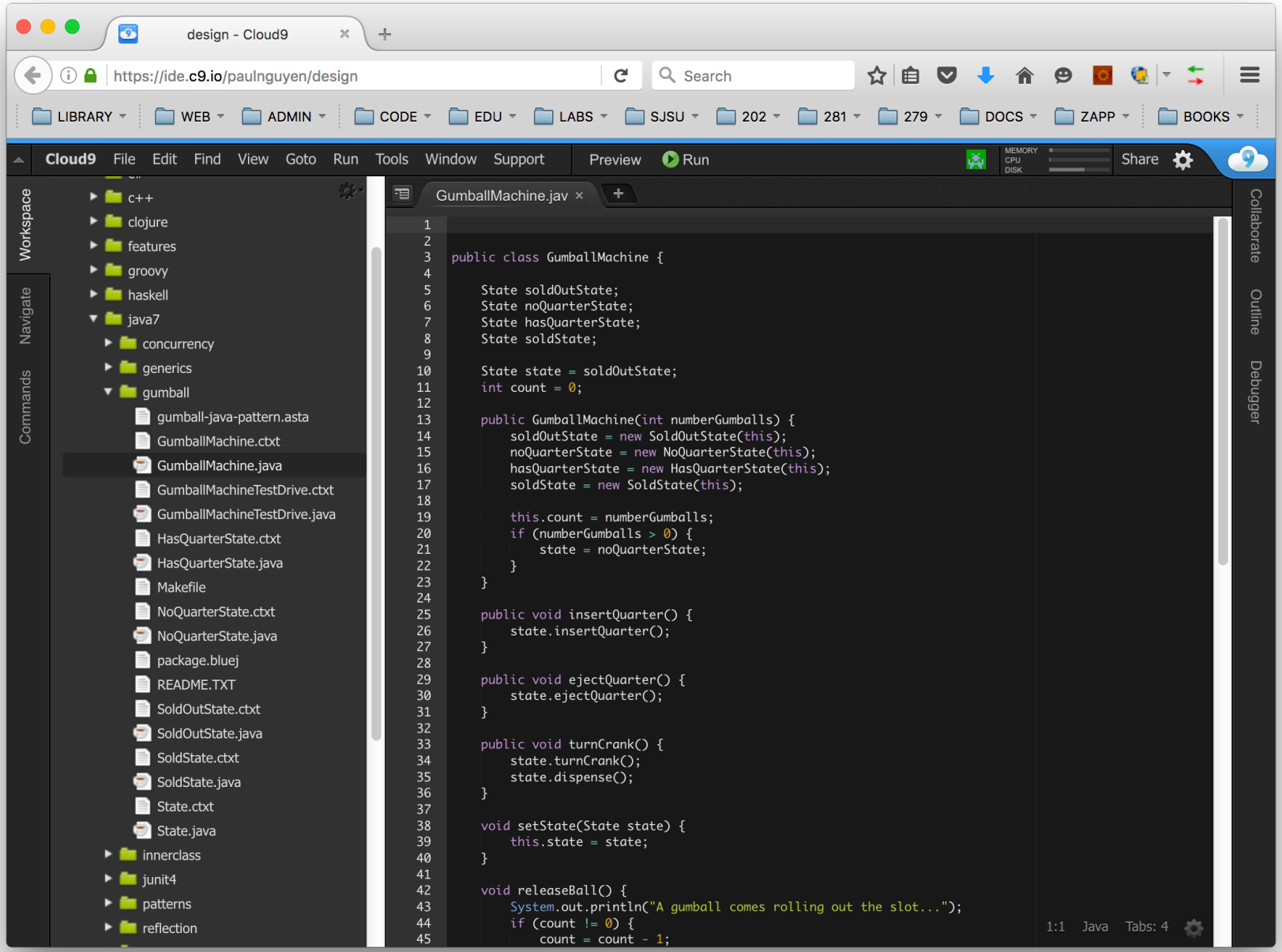
Cost: 50 cents
Accepts Two Quarters

Cost: 50 cents
Accepts All Coins
(i.e. nickels, dimes, & quarters)

Implementation in C (FSM Version)

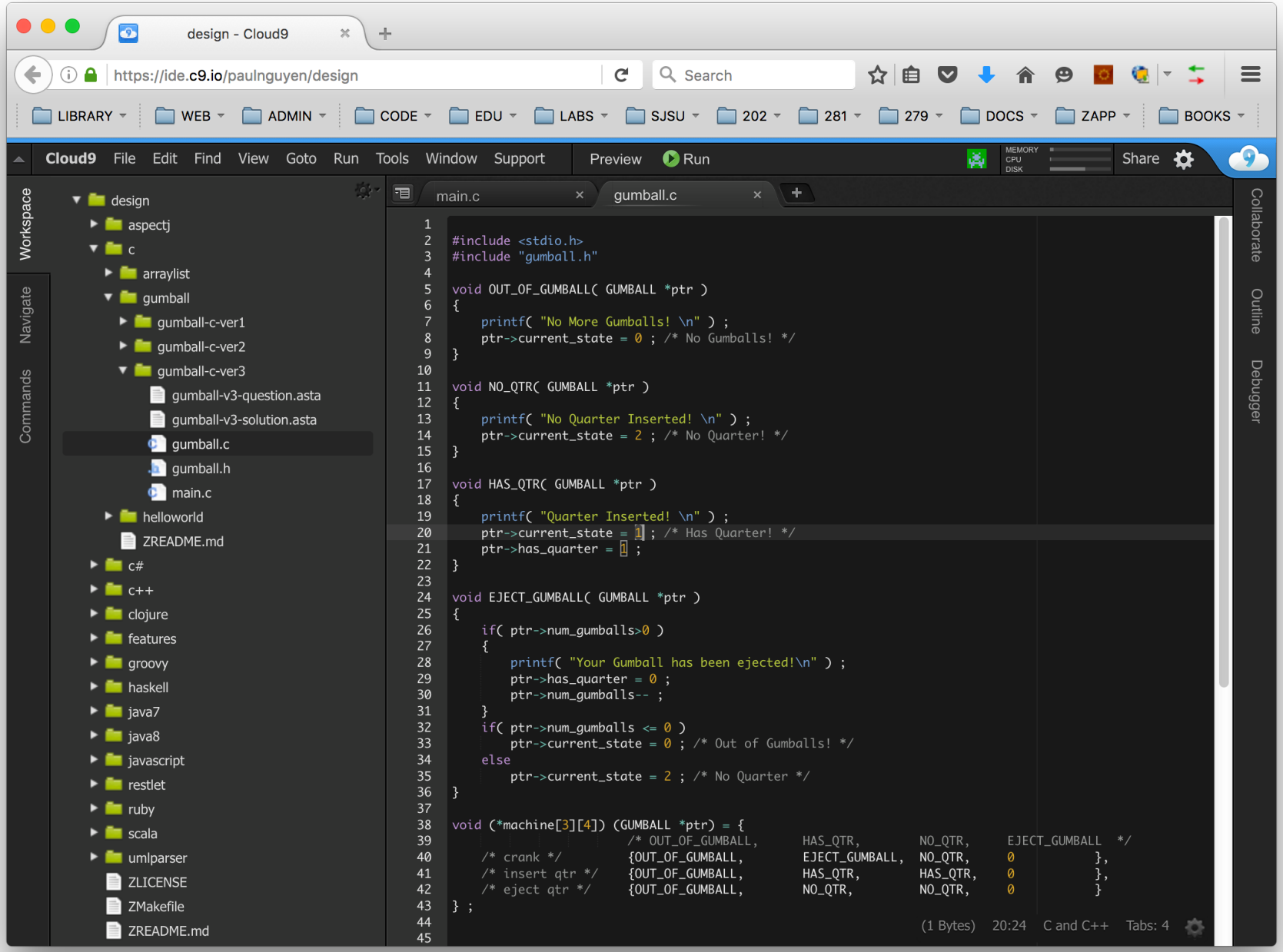


Implementation in Java (Pattern Version)




```
1
2
3   public class GumballMachine {
4
5       State soldOutState;
6       State noQuarterState;
7       State hasQuarterState;
8       State soldState;
9
10      State state = soldOutState;
11      int count = 0;
12
13      public GumballMachine(int numberGumballs) {
14          soldOutState = new SoldOutState(this);
15          noQuarterState = new NoQuarterState(this);
16          hasQuarterState = new HasQuarterState(this);
17          soldState = new SoldState(this);
18
19          this.count = numberGumballs;
20          if (numberGumballs > 0) {
21              state = noQuarterState;
22          }
23      }
24
25      public void insertQuarter() {
26          state.insertQuarter();
27      }
28
29      public void ejectQuarter() {
30          state.ejectQuarter();
31      }
32
33      public void turnCrank() {
34          state.turnCrank();
35          state.dispense();
36      }
37
38      void setState(State state) {
39          this.state = state;
40      }
41
42      void releaseBall() {
43          System.out.println("A gumball comes rolling out the slot...");
44          if (count != 0) {
45              count = count - 1;
```

Question: What's the State Diagram?



Solution (try this out using Astah UML)

```c
#include <stdio.h>
#include "gumball.h"

void OUT_OF_GUMBALL( GUMBALL *ptr )
{
    printf( "No More Gumballs! \n" ) ;
    ptr->current_state = 0 ; /* No Gumballs! */
}

void NO_QTR( GUMBALL *ptr )
{
    printf( "No Quarter Inserted! \n" ) ;
    ptr->current_state = 2 ; /* No Quarter! */
}

void HAS_QTR( GUMBALL *ptr )
{
    printf( "Quarter Inserted! \n" ) ;
    ptr->current_state = 1 ; /* Has Quarter! */
    ptr->has_quarter = 1 ;
}

void EJECT_GUMBALL( GUMBALL *ptr )
{
    if( ptr->num_gumballs>0 )
    {
        printf( "Your Gumball has been ejected!\n" ) ;
        ptr->has_quarter = 0 ;
        ptr->num_gumballs-- ;
    }
    if( ptr->num_gumballs <= 0 )
        ptr->current_state = 0 ; /* Out of Gumballs! */
    else
        ptr->current_state = 2 ; /* No Quarter */
}

void (*machine[3][4]) (GUMBALL *ptr) = {
                    /* OUT_OF_GUMBALL,      HAS_QTR,        NO_QTR,     EJECT_GUMBALL  */
    /* crank */     {OUT_OF_GUMBALL,        EJECT_GUMBALL,  NO_QTR,     0           },
    /* insert qtr */{OUT_OF_GUMBALL,        HAS_QTR,        HAS_QTR,    0           },
    /* eject qtr */ {OUT_OF_GUMBALL,        NO_QTR,         NO_QTR,     0           }
} ;
```