

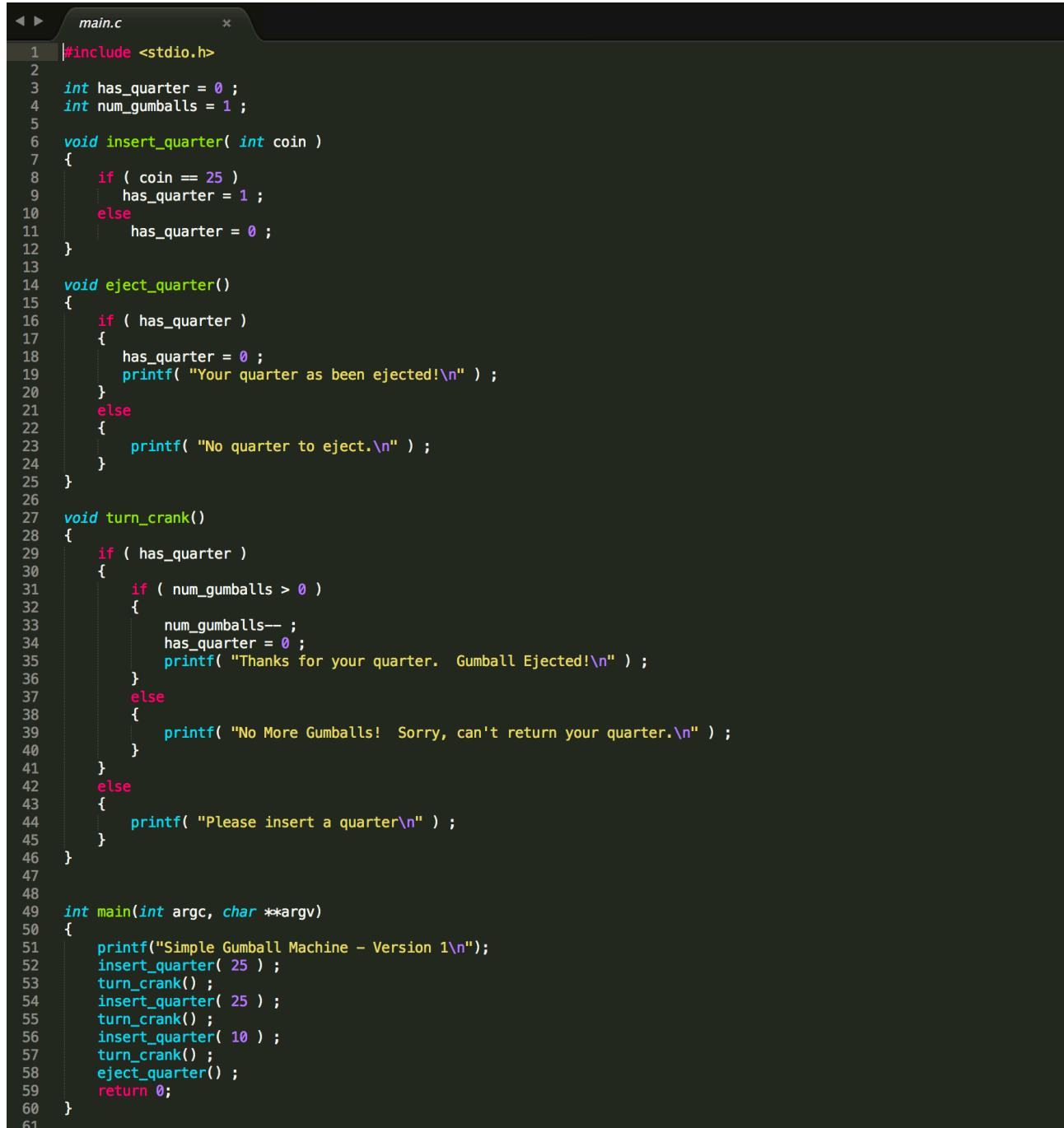
CMPE 202

“Classical” Procedural Programming
vs.
Object-Oriented Programming

The Gumball Machine



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



A screenshot of a code editor showing the file `main.c`. The code is a C program for a simple gumball machine. It includes functions for inserting a quarter, ejecting a quarter, and turning the crank. The `main` function initializes the machine and performs a sequence of operations.

```
1 // main.c
2
3 #include <stdio.h>
4
5 int has_quarter = 0 ;
6 int num_gumballs = 1 ;
7
8 void insert_quarter( int coin )
9 {
10     if ( coin == 25 )
11         has_quarter = 1 ;
12     else
13         has_quarter = 0 ;
14
15 void eject_quarter()
16 {
17     if ( has_quarter )
18     {
19         has_quarter = 0 ;
20         printf( "Your quarter has been ejected!\n" ) ;
21     }
22     else
23     {
24         printf( "No quarter to eject.\n" ) ;
25     }
26
27 void turn_crank()
28 {
29     if ( has_quarter )
30     {
31         if ( num_gumballs > 0 )
32         {
33             num_gumballs-- ;
34             has_quarter = 0 ;
35             printf( "Thanks for your quarter. Gumball Ejected!\n" ) ;
36         }
37         else
38         {
39             printf( "No More Gumballs! Sorry, can't return your quarter.\n" ) ;
40         }
41     }
42     else
43     {
44         printf( "Please insert a quarter\n" ) ;
45     }
46 }
47
48
49 int main(int argc, char **argv)
50 {
51     printf("Simple Gumball Machine - Version 1\n");
52     insert_quarter( 25 ) ;
53     turn_crank() ;
54     insert_quarter( 25 ) ;
55     turn_crank() ;
56     insert_quarter( 10 ) ;
57     turn_crank() ;
58     eject_quarter() ;
59     return 0;
60 }
```

**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>

```
1  public class GumballMachine
2  {
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 }
```

**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**

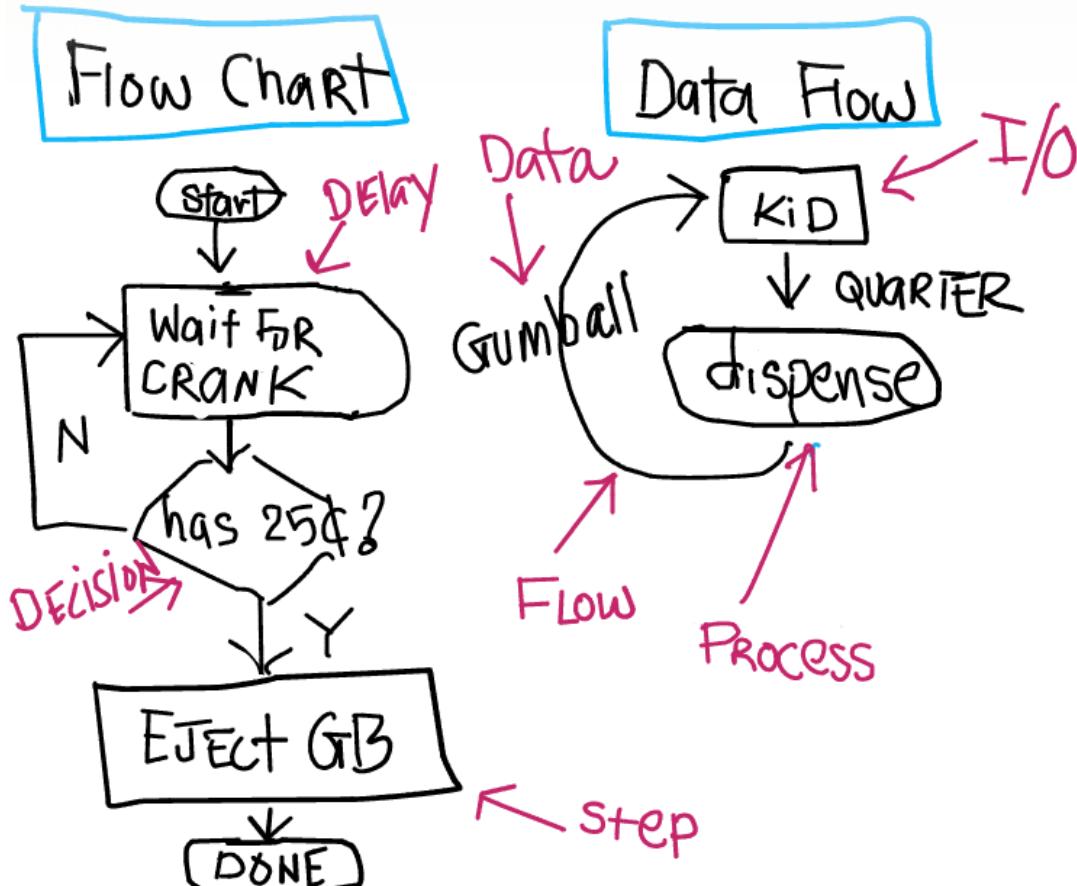


**Cost: 50 cents
Accepts All Coins
(i.e. nickels, dimes, &
quarters)**

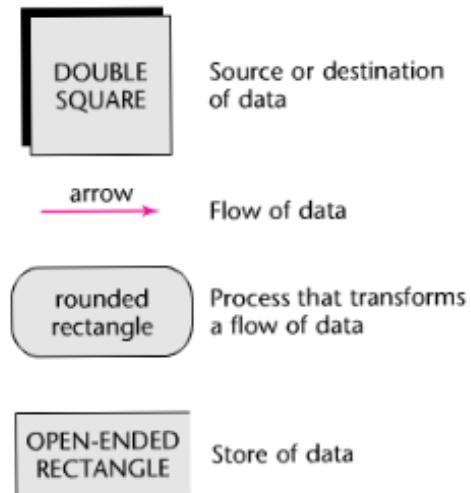
- 1. Now, Modify your code to support all three of the Machine Types above!**

Basic Requirements:

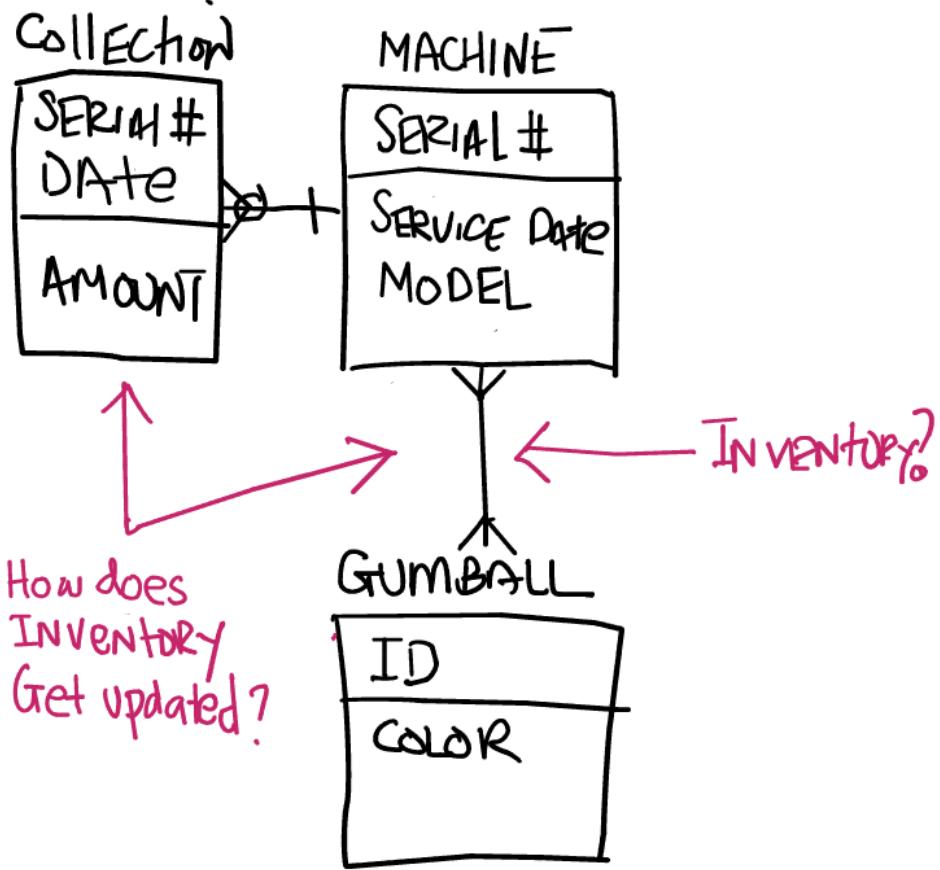
1. Insert a quarter, crank, get a gumball
2. If no more gumball, lose the quarter



	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

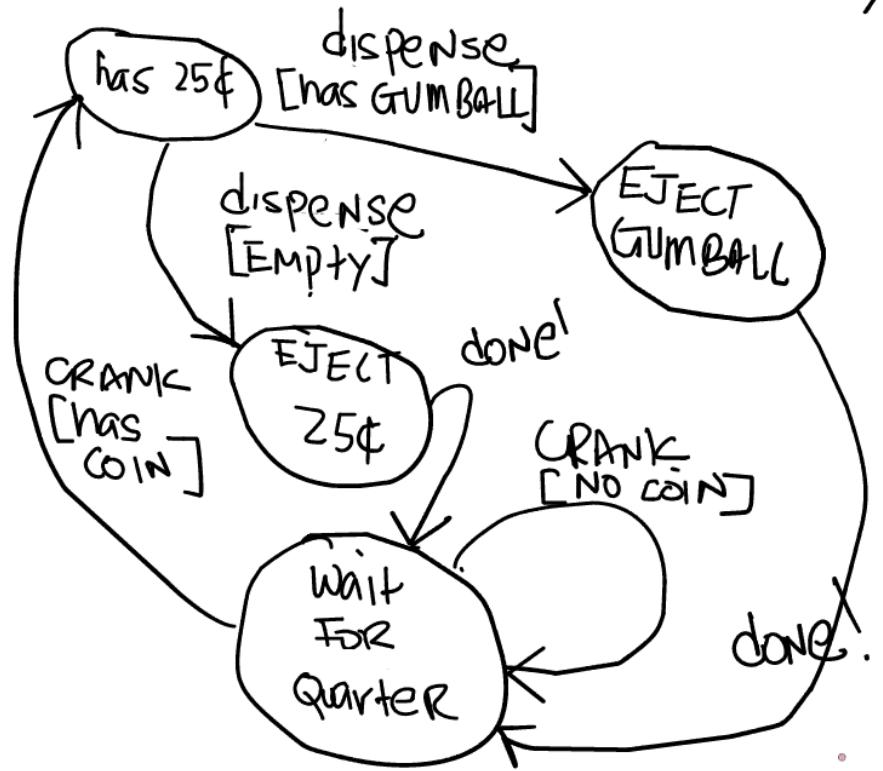


Data Model



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

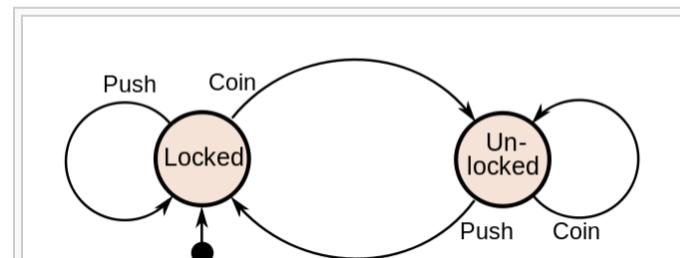
on the arm has no effect; no matter how many times it is done, the state remains

Putting a coin in – that is, giving the machine a coin – changes the state from the **locked** state to the **unlocked** state, putting additional coins in does not change the state. However, a customer pushing the arm while the machine is in the **unlocked** state

The turnstile state machine can be represented as a transition table:

Current State	Input	Next State	
Locked	coin	Unlocked	Unlocked
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	Wherever

It can also be represented by a [directed graph](#), showing the transitions from one state to another. Each row in the table (the [coin](#) input in the **Unlocked** state) are represented by a directed edge from the **Unlocked** state to the **Locked** state.



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		
		Current state	State A	State B
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.

Implementation in C (version 1) - Typical Solution

```
main.c ✘
1 #include <stdio.h>
2
3 int has_quarter = 0 ;
4 int num_gumballs = 1 ;
5
6 void insert_quarter( int coin )
7 {
8     if ( coin == 25 )
9         has_quarter = 1 ;
10    else
11        has_quarter = 0 ;
12 }
13
14 void turn_crank()
15 {
16     if ( has_quarter )
17     {
18         if ( num_gumballs > 0 )
19         {
20             num_gumballs-- ;
21             has_quarter = 0 ;
22             printf( "Thanks for your quarter. Gumball Ejected!\n" ) ;
23         }
24         else
25         {
26             printf( "No More Gumballs! Sorry, can't return your quarter.\n" ) ;
27         }
28     }
29     else
30     {
31         printf( "Please insert a quarter\n" ) ;
32     }
33 }
34
35 int main(int argc, char **argv)
36 {
37     printf("Simple Gumball Machine - Version 1\n");
38     insert_quarter( 25 ) ;
39     turn_crank() ;
40     insert_quarter( 25 ) ;
41     turn_crank() ;
42     insert_quarter( 10 ) ;
43     turn_crank() ;
44     return 0;
45 }
```

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
1
2     typedef struct
3     {
4         int num_gumballs ;
5         int has_quarter ;
6     } GUMBALL ;
7
8     extern void init_gumball( GUMBALL *ptr, int size ) ;
9     extern void turn_crank( GUMBALL *ptr ) ;
10    extern void insert_quarter( GUMBALL *ptr, int coin );
```

```
*main.c
1 #include <stdio.h>
2 #include "gumball.h"
3
4 int main(int argc, char **argv)
5 {
6     GUMBALL m1[1] ;
7     GUMBALL m2[1] ;
8
9     /* init gumball machines */
10    init_gumball( m1, 1 ) ;
11    init_gumball( m2, 10 ) ;
12
13    printf("Simple Gumball Machine - Version 2\n");
14
15    insert_quarter( m1, 25 ) ;
16    turn_crank( m1 ) ;
17    insert_quarter( m1, 25 ) ;
18    turn_crank( m1 ) ;
19    insert_quarter( m1, 10 ) ;
20    turn_crank( m1 ) ;
21
22    insert_quarter( m2, 25 ) ;
23    turn_crank( m2 ) ;
24    insert_quarter( m2, 25 ) ;
25    turn_crank( m2 ) ;
26    insert_quarter( m1, 10 ) ;
27    turn_crank( m2 ) ;
28
29    return 0;
```

```
gumball.c
1
2     #include <stdio.h>
3     #include "gumball.h"
4
5     void init_gumball( GUMBALL *ptr, int size )
6     {
7         ptr->num_gumballs = size ;
8         ptr->has_quarter = 0 ;
9     }
10
11     void turn_crank( GUMBALL *ptr )
12     {
13         if ( ptr->has_quarter )
14         {
15             if ( ptr->num_gumballs > 0 )
16             {
17                 ptr->num_gumballs-- ;
18                 ptr->has_quarter = 0 ;
19                 printf( "Thanks for your quarter. Gumball Ejected!\n" );
20             }
21             else
22             {
23                 printf( "No More Gumballs! Sorry, can't return your quarter.\n" );
24             }
25         }
26         else
27         {
28             printf( "Please insert a quarter\n" );
29         }
30     }
31
32     void insert_quarter( GUMBALL *ptr, int coin )
33     {
34         if ( coin == 25 )
35             ptr->has_quarter = 1 ;
36         else
37             ptr->has_quarter = 0 ;
38 }
```

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 public class GumballMachine
2 {
3
4     private int num_gumballs;
5     private boolean has_quarter;
6
7     public GumballMachine( int size )
8     {
9         // initialise instance variables
10        this.num_gumballs = size;
11        this.has_quarter = false;
12    }
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}
```

Native support for
Encapsulation
in OO Languages
(like Java)

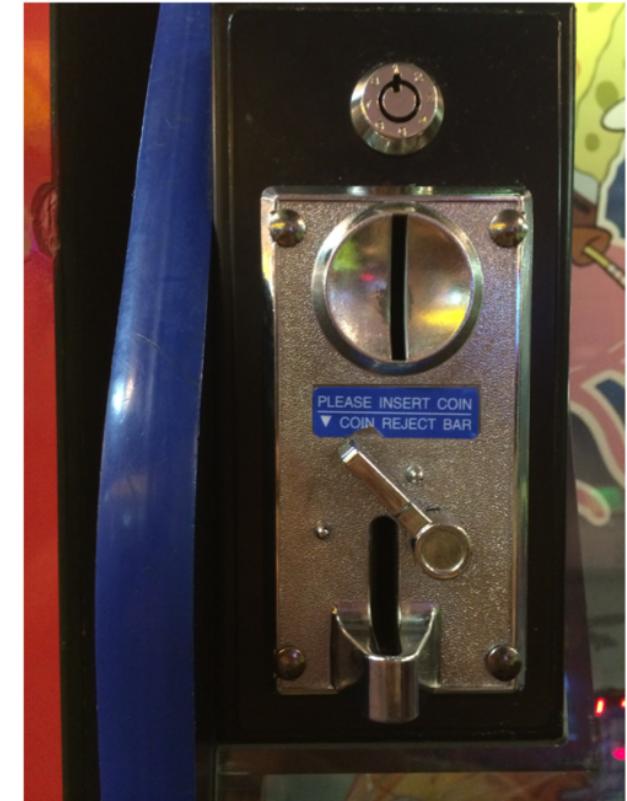
Dealing with Product Variants



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)

The screenshot shows the Cloud9 IDE interface with the following details:

- Workspace:** design
- File:** main.c (active tab)
- Code Content:**

```
1 #include <stdio.h>
2 #include "gumball.h"
3
4 void OUT_OF_GUMBALL( GUMBALL *ptr )
5 {
6     printf( "No More Gumballs! \n" );
7     ptr->current_state = 0 ; /* No Gumballs! */
8 }
9
10 void NO_QTR( GUMBALL *ptr )
11 {
12     printf( "No Quarter Inserted! \n" );
13     ptr->current_state = 2 ; /* No Quarter! */
14 }
15
16 void HAS_QTR( GUMBALL *ptr )
17 {
18     printf( "Quarter Inserted! \n" );
19     ptr->current_state = 1 ; /* Has Quarter! */
20     ptr->has_quarter = 1 ;
21 }
22
23 void EJECT_GUMBALL( GUMBALL *ptr )
24 {
25     if( ptr->num_gumballs>0 )
26     {
27         printf( "Your Gumball has been ejected!\n" );
28         ptr->has_quarter = 0 ;
29         ptr->num_gumballs-- ;
30     }
31     if( ptr->num_gumballs <= 0 )
32         ptr->current_state = 0 ; /* Out of Gumballs! */
33     else
34         ptr->current_state = 2 ; /* No Quarter */
35 }
36
37 void (*machine[3][4])(GUMBALL *ptr) = {
38     {OUT_OF_GUMBALL, HAS_QTR, NO_QTR, EJECT_GUMBALL },
39     /* crank */ {OUT_OF_GUMBALL, EJECT_GUMBALL, NO_QTR, 0 },
40     /* insert qtr */ {OUT_OF_GUMBALL, HAS_QTR, HAS_QTR, 0 },
41     /* eject qtr */ {OUT_OF_GUMBALL, NO_QTR, NO_QTR, 0 }
42 } ;
43
44
45
```
- IDE Status:** (1 Bytes) 20:24 C and C++ Tabs: 4
- Right Sidebar:** Collaborate, Outline, Debugger

Implementation in Java (Pattern Version)

The screenshot shows the Cloud9 IDE interface with the following details:

- Header:** design - Cloud9
- Address Bar:** https://ide.c9.io/paulnguyen/design
- Toolbar:** LIBRARY, WEB, ADMIN, CODE, EDU, LABS, SJSU, 202, 281, 279, DOCS, ZAPP, BOOKS
- Cloud9 Menu:** File, Edit, Find, View, Goto, Run, Tools, Window, Support, Preview, Run
- Memory/CPU/Disk Monitor:** Shows current usage.
- Share and Settings:** Share, Settings, Cloud9 icon.
- Sidebar:** Collaborate, Outline, Debugger.
- Left Sidebar:** Workspace, Navigate, Commands. Under Workspace, the "java7" folder is expanded, showing subfolders "concurrency", "generics", and "gumball". The "gumball" folder contains files: gumball-java-pattern.asta, GumballMachine.ctx, GumballMachine.java (selected), GumballMachineTestDrive.ctx, GumballMachineTestDrive.java, HasQuarterState.ctx, HasQuarterState.java, Makefile, NoQuarterState.ctx, NoQuarterState.java, package.bluej, README.TXT, SoldOutState.ctx, SoldOutState.java, SoldState.ctx, SoldState.java, State.ctx, State.java, innerclass, junit4, patterns, reflection.
- Code Editor:** Title: GumballMachine.java x. Content:

```
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;
46        }
47    }
48}
```
- Bottom Status Bar:** 1:1 Java Tabs: 4

Question: What's the State Diagram?

The screenshot shows the Cloud9 IDE interface with a dark theme. The top bar includes tabs for 'design - Cloud9', 'Search', and various project and file management icons. The main workspace consists of two tabs: 'main.c' and 'gumball.c'. The 'main.c' tab contains a simple 'Hello World' program. The 'gumball.c' tab displays the following C code:

```
1 #include <stdio.h>
2 #include "gumball.h"
3
4 void OUT_OF_GUMBALL( GUMBALL *ptr )
5 {
6     printf( "No More Gumballs! \n" );
7     ptr->current_state = 0 ; /* No Gumballs! */
8 }
9
10 void NO_QTR( GUMBALL *ptr )
11 {
12     printf( "No Quarter Inserted! \n" );
13     ptr->current_state = 2 ; /* No Quarter! */
14 }
15
16 void HAS_QTR( GUMBALL *ptr )
17 {
18     printf( "Quarter Inserted! \n" );
19     ptr->current_state = 1 ; /* Has Quarter! */
20     ptr->has_quarter = 1 ;
21 }
22
23 void EJECT_GUMBALL( GUMBALL *ptr )
24 {
25     if( ptr->num_gumballs>0 )
26     {
27         printf( "Your Gumball has been ejected!\n" );
28         ptr->has_quarter = 0 ;
29         ptr->num_gumballs-- ;
30     }
31     if( ptr->num_gumballs <= 0 )
32         ptr->current_state = 0 ; /* Out of Gumballs! */
33     else
34         ptr->current_state = 2 ; /* No Quarter */
35 }
36
37
38 void (*machine[3][4])(GUMBALL *ptr) = {
39     /* OUT_OF_GUMBALL,      HAS_QTR,      NO_QTR,      EJECT_GUMBALL */
40     /* crank */           {OUT_OF_GUMBALL,   EJECT_GUMBALL, NO_QTR,    0 },
41     /* insert qtr */       {OUT_OF_GUMBALL,   HAS_QTR,     NO_QTR,    0 },
42     /* eject qtr */        {OUT_OF_GUMBALL,   NO_QTR,     NO_QTR,    0 }
43 } ;
```

The code defines four functions: OUT_OF_GUMBALL, NO_QTR, HAS_QTR, and EJECT_GUMBALL. It also initializes a 3x4 array 'machine' with pointers to these functions. The array dimensions are [3][4], where the first dimension represents states and the second represents actions. The states are likely represented by the values 0, 1, and 2 assigned in the HAS_QTR function. The actions are likely represented by the function names themselves.

Solution (try this out using Astah UML)

```
2 #include <stdio.h>
3 #include "gumball.h"
4
5 void OUT_OF_GUMBALL( GUMBALL *ptr )
6 {
7     printf( "No More Gumballs! \n" );
8     ptr->current_state = 0 ; /* No Gumballs! */
9 }
10
11 void NO_QTR( GUMBALL *ptr )
12 {
13     printf( "No Quarter Inserted! \n" );
14     ptr->current_state = 2 ; /* No Quarter! */
15 }
16
17 void HAS_QTR( GUMBALL *ptr )
18 {
19     printf( "Quarter Inserted! \n" );
20     ptr->current_state = 1 ; /* Has Quarter! */
21     ptr->has_quarter = 1 ;
22 }
23
24 void EJECT_GUMBALL( GUMBALL *ptr )
25 {
26     if( ptr->num_gumballs>0 )
27     {
28         printf( "Your Gumball has been ejected!\n" );
29         ptr->has_quarter = 0 ;
30         ptr->num_gumballs-- ;
31     }
32     if( ptr->num_gumballs <= 0 )
33         ptr->current_state = 0 ; /* Out of Gumballs! */
34     else
35         ptr->current_state = 2 ; /* No Quarter */
36 }
37
38 void (*machine[3][4])(GUMBALL *ptr) = {
39     /* OUT_OF_GUMBALL,      HAS_QTR,      NO_QTR,      EJECT_GUMBALL */
40     /* crank */           {OUT_OF_GUMBALL,   EJECT_GUMBALL,  NO_QTR,      0 },
41     /* insert qtr */       {OUT_OF_GUMBALL,   HAS_QTR,      HAS_QTR,      0 },
42     /* eject qtr */        {OUT_OF_GUMBALL,   NO_QTR,      NO_QTR,      0 }
43 } ;
```

