CS 246 Spring 2012 – Assignment 5
Instructor: Jason Hinek
Bonus Due Date: Monday, July 23, 2012 at 23:59 (11:59pm)
Due Date: Wednesday, July 25, 2012 at 23:59 (11:59pm)

July 11, 2012

This assignment examines programming at the medium scale. Use it to become familiar with this new idea and make sure you use these concepts in your assignment solution.

WATCola is renown for its famous line of healthy soda pop, which come in the dazzling array of flavours: Blues Black-Cherry, Classical Cream-Soda, Rock Root-Beer, and Jazz Lime. The company recently won the bid to provide soda to vending machines on campus. The vending machines are periodically restocked by a WATCola truck making deliveries from the local bottling plant.

This assignment simulates a simple concession service using the objects and relationships in Figure 1. (Not all possible communication paths are shown in the diagram.)

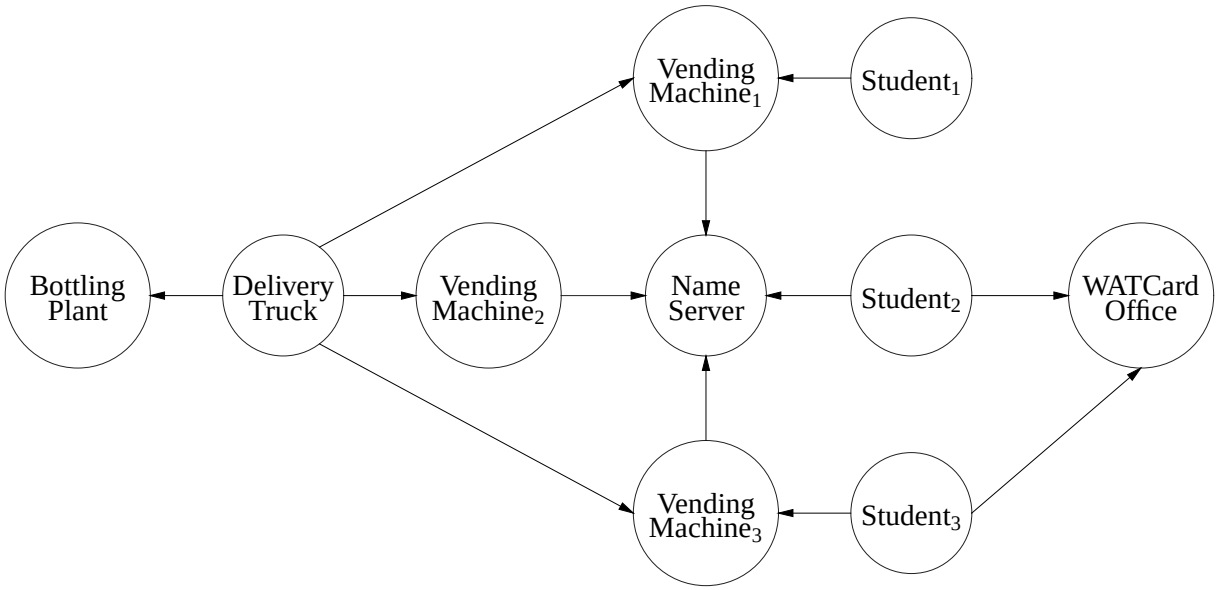

Figure 1: Concession Service

The following constants are used to configure the assignment, and are read from a text file:

```
SodaCost              1   # MSRP per bottle
NumStudents           4   # number of students to create
MaxPurchases          10  # maximum number of bottles a student purchases
NumVendingMachines    3   # number of vending machines
MaxStockPerFlavour    5   # maximum number of bottles of each flavour in a vending machine
MaxShippedPerFlavour  6   # maximum number of bottles of each flavour generated by the bottling
                          #     plant per production run
TimeBetweenShipments  5   # length of time between shipment pickup
```

Comments in the file (from # to the end-of-line), as well as blank lines, are ignored. The constants may appear in any order. Any number of spaces/tabs may appear around a constant name, value or comment. You may assume each constant appears in the configuration file, is syntactically correct, its value is within an appropriate range (i.e., no error

checking is required), and only one constant is defined per line. You may have to modify the values in the provided sample file to obtain interesting results when testing.

The following types and routines are required in the assignment (you may add only a public destructor and private/protected members):

1. ```
   class Student {
     public:
       Student( Printer &prt, NameServer &nameServer, WATCardOffice &cardOffice, unsigned int id,
                unsigned int maxPurchases );
       bool action();
   };
   ```

   A Student's function is to buy some of their favourite soda from a vending machine (a bottle costs whatever the vending machine is charging). Each student is passed an id in the range [0, NumStudents) for identification. A student begins by selecting a random number of bottles to purchase [1, MaxPurchases] and a random favourite flavour [0, 3], creating a WATCard (dynamically) with a $5 balance via the WATCardOffice, and obtaining the location of a vending machine from the name server. Subsequently, a student's behaviour is defined by member action as follows. When a student has purchased all the soda initially selected, then action returns false. A student then attempts to buy a bottle of soda from the vending machine. If the vending machine destroys a student's WATCard, the student must create a new one by creating a WATCard (dynamically) with $0 balance and transferring the same initial $5 to the card via the WATCardOffice. If the vending machine delivered a bottle of soda, the student drinks it and returns true. Otherwise, a student checks their WATCard for insufficient funds (less than VendingMachine::cost()) to purchase a bottle of soda. If there is insufficient funds, a student transfers $3 to their WATCard from the WATCard office until there is sufficient funds to buy at least one soda and returns true; the next time action is called, the student attempts to buy a soda with the new money. If the vending machine is out of the student's favourite flavour, the student must obtain a new vending machine from the name server and return true; the next time action is called, the student attempts to buy a soda at the new vending machine.

2. ```
   class WATCard {
     public:
       // you design a general WATCard
   };
   ```

   The WATCard manages the money associated with a card. When a WATCard is created it always has a $0 balance. You design the interface and implementation.

3. ```
   class WATCardOffice {
     public:
       WATCardOffice( Printer &prt );
       WATCard *create( unsigned int id, unsigned int amount );
       void transfer( unsigned int id, unsigned int amount, WATCard &card );
   };
   ```

   The WATCardOffice is used by a student to create a WATCARD with an initial deposit and transfer funds to their WATCard to buy a soda. For a transfer, there is a 1 in 4 chance a student only has 1/2 of the requested amount deposited; the other 1/2 is lost.

4. ```
   class VendingMachine {                  // general vending machine
     public:
       enum Status { BUY, STOCK, FUNDS }; // purchase status: successful buy, out of stock, insufficient funds
       VendingMachine( Printer &prt, NameServer &nameServer,
                       unsigned int id, unsigned int sodaCost, unsigned int maxStockPerFlavour );
       virtual ~VendingMachine();          // necessary to trigger destructors in inherited classes
       virtual Status buy( Flavours flavour, WATCard *&card ); // YOU DEFINE FLAVOURS
       virtual unsigned int *inventory();
       virtual void restocked();
       virtual unsigned int cost();
       virtual unsigned int getId();
   };
   ```

```
class VendingMachineCardEater : public VendingMachine { // specific vending machine
  public:
    VendingMachineCardEater( Printer &prt, NameServer &nameServer,
                             unsigned int id, unsigned int sodaCost, unsigned int maxStockPerFlavour );
    // member routines from VendingMachine
};
class VendingMachineOverCharger : public VendingMachine { // specific vending machine
  public:
    VendingMachineOverCharger( Printer &prt, NameServer &nameServer,
                             unsigned int id, unsigned int sodaCost, unsigned int maxStockPerFlavour );
    // member routines from VendingMachine
};
```

A vending machine's function is to sell soda to students at some cost. All vending machines have the functionality given by VendingMachine, but actual vending machines may behave differently. Each vending machine is passed an id in the range [0, NumVendingMachines) for identification, MSRP price for a bottle of soda, and the maximum number of bottles of each flavour in a vending machine. A new vending machine is empty (no stock) and begins by registering with the name server. A student calls buy to obtain one of their favourite sodas. If the specified soda is unavailable, buy returns STOCK. If the student has insufficient funds to purchase the soda, buy returns FUNDS. Otherwise, the student's WATCard is debited by the cost of a soda and buy returns BUY. One kind of vending machine has a problem *after* making a purchase: there is a 1 in 10 chance of it *eating* a student's WATCard by deleting it and setting card to NULL. In this case, the student must create a new WATCard (see Student task). The other kind of vending machine incorrectly charges double the cost of each bottle of soda; it always returns $2 \times sodaCost$ from its cost member.

Periodically, the truck comes by to restock the vending machines with new soda from the bottling plant. Restocking is performed in two steps. The truck calls inventory to return a pointer to an array containing the amount of each kind of soda currently in the vending machine. The truck uses this information to transfer into each machine as much of its stock of new soda as fits; for each kind of soda, no more than MaxStockPerFlavour per flavour can be added to a machine. If the truck cannot top-up a particular flavour, it transfers as many bottles as it has (which could be 0). After transferring new soda into the machine by directly modifying the array passed from inventory, the truck calls restocked to indicate the operation is complete. The cost returns the cost of purchasing a soda for this machine. The getId returns the identification number of the vending machine. You define the public type Flavours to represent the 4 different flavours of soda.

5.
```
class NameServer {
  public:
    NameServer( Printer &prt, unsigned int numVendingMachines, unsigned int numStudents );
    void VMregister( VendingMachine *vendingmachine );
    VendingMachine *getMachine( unsigned int id );
    VendingMachine **getMachineList();
};
```

The name server's function is to manage the vending-machine names. The name server is passed the number of vending machines, NumVendingMachines, and the number of students, NumStudents. It begins by logically distributing the students evenly across the vending machines in a round-robin fashion. That is, student id 0 is assigned to the first registered vending-machine, student id 1 is assigned to the second registered vending-machine, etc., until there are no more registered vending-machines, and then start again with the first registered vending-machine. Vending machines call VMregister to register themselves so students can subsequently locate them. A student calls getMachine to find a vending machine, and the name server must cycle through the vending machines *separately* for each student starting from the initial position via modulo incrementing to ensure a student has a chance to visit every machine. The truck calls getMachineList to obtain an array of pointers to vending machines so it can visit each machine to deliver new soda.

6.
```
class BottlingPlant {
  public:
    BottlingPlant( Printer &prt, NameServer &nameServer, unsigned int numVendingMachines,
                   unsigned int maxShippedPerFlavour, unsigned int maxStockPerFlavour,
                   unsigned int timeBetweenShipments );
    void getShipment( unsigned int cargo[] );
    void action();
};
```

The bottling plant produces random new quantities of each flavour of soda, [0, MaxShippedPerFlavour] per flavour. The bottling plant is passed the number of vending machines, NumVendingMachines, the maximum number of bottles of each flavour generated during a production run and subsequently shipped, MaxShippedPerFlavour, the maximum number of bottles of each flavour in a vending machine MaxStockPerFlavour, and the length of time between shipment pickups by the truck, TimeBetweenShipments. It begins by creating a truck, performing a production run, and distributing these bottles to initialize the registered vending machines using the truck. Subsequently, the bottling plant's behaviour is defined by member action as follows. There is a 1 in 5 chance the bottling plant is on strike. If on strike, the bottling plant returns immediately without doing any work; otherwise, the bottling plant checks if the truck has come back from the last delivery. There is a 1 in TimeBetweenShipments chance the truck has returned. If the truck has returned, the bottling plant performs a production run, and then calls the truck's action routine to deliver the current production run to the vending machines; otherwise, the bottling plant returns without doing any work. The truck calls getShipment to obtain a shipment from the plant (i.e., the production run), and the shipment is copied into the cargo array passed by the truck.

7.
```
class Truck {
  public:
    Truck( Printer &prt, NameServer &nameServer, BottlingPlant &plant,
           unsigned int numVendingMachines, unsigned int maxStockPerFlavour );
    void action();
};
```

The truck moves soda from the bottling plant to the vending machines. The truck is passed the number of vending machines, numVendingMachines, and the maximum number of bottles of each flavour in a vending machine maxStockPerFlavour. Subsequently, the truck's behaviour is defined by member action, which is called by the bottling plant. You must determine how the truck works from looking at the output from the sample executable.

All output from the program is generated by calls to a printer, excluding error messages.

```
class Printer {
    struct PrinterImpl;
    PrinterImpl &impl;
  public:
    enum Kind { WATCardOffice, NameServer, Truck, BottlingPlant, Student, Vending };
    Printer( unsigned int numStudents, unsigned int numVendingMachines );
    ~Printer();
    void print( Kind kind, char state );
    void print( Kind kind, char state, int value1 );
    void print( Kind kind, char state, int value1, int value2 );
    void print( Kind kind, unsigned int lid, char state );
    void print( Kind kind, unsigned int lid, char state, int value1 );
    void print( Kind kind, unsigned int lid, char state, int value1, int value2 );
};
```

The printer generates output like that in Figure 2.

```
% soda soda.config 1009
WATOff  Names   Truck   Plant   Stud0   Stud1   Stud2   Mach0   Mach1
******* ******* ******* ******* ******* ******* ******* ******* *******
S       S                                                       S2
        R1                                              S2
        R0      S       S
                        G15
                P15     P                                       r
                d1,15
                U1,4
                D1,7                                    r       R
                d0,7
                U0,9
C1,5    N1,0    D0,4                    S1,1            R
C0,5    N0,1                    S2,2    V0
C2,5    N2,1                    V1              S3,5
                        G28     B3      B3      V1      B1,2    B2,2
                P28     P                                       r
                d1,28
                D1,23                                   r       R
                d0,23
                D0,13                           B3      R       B3,2
...     ...     ...     ...     ...     F       ...     ...     ...
                        X       B1                              B2,2
...     ...     ...     ...     F       ...     ...     ...     ...
                        X                       B1              B3,1
t2,3                                            t3
T2,3                                            T4              B3,0
        N2,0                                    B2
                                                V0      B3,2
                                                B0
t2,3                                            t3
T2,3                                            T3      B3,1
                                                B1
...     ...     ...     ...     ...     ...     F       ...     ...
...     ...     ...     ...     ...     ...     ...     ...     F
...     ...     ...     ...     ...     ...     ...     F       ...
...     ...     ...     F       ...     ...     ...     ...     ...
...     ...     F       ...     ...     ...     ...     ...     ...
...     F       ...     ...     ...     ...     ...     ...     ...
F       ...     ...     ...     ...     ...     ...     ...     ...
***********************
```

Figure 2: WATCola : Example Output

Each column is assigned to a particular kind of object. There are 6 kinds of objects: WATCard office, name server, truck, bottling plant, student, and vending machine. Student and vending machine have multiple instances. For the objects with multiple instances, these objects pass in their local identifier [0,N) when printing. Each kind of object prints specific information in its column:

- The WATCard office prints the following information:

State	Meaning	Additional Information
S	starting	
C	creating a WATCarder	student requesting creation, initial amount of deposit
t	start funds transfer	student requesting transfer, requested amount of transfer
T	complete funds transfer	student requesting transfer, actual amount of transfer
F	finished	

- The name server prints the following information:

State	Meaning	Additional Information
S	starting	
R	register vending machine	identifier of registering vending machine
N	new vending machine	student requesting vending machine, new vending machine
F	finished	

- The truck prints the following information:

State	Meaning	Additional Information
S	starting	
P	picked up shipment	total amount of all sodas in the shipment
d	begin delivery to vending machine	vending machine id, total amount remaining in the shipment
U	unsuccessfully filled vending machine	vending machine id, total number of bottles not replenished
D	end delivery to vending machine	vending machine id, total amount remaining in the shipment
F	finished	

States d and D are printed for each vending machine visited during restocking.

- The bottling plant prints the following information:

State	Meaning	Additional Information
S	starting	
X	on strike	
G	generating soda	bottles generated in production run
P	shipment picked up by truck	
F	finished	

- A student prints the following information:

State	Meaning	Additional Information
S	starting	favourite soda, number of bottles to purchase
V	selecting vending machine	identifier of selected vending machine
t	start funds transfer	amount of transfer
T	complete funds transfer	WATCard balance
B	bought a soda	WATCard balance
D	WATCard destroyed	
F	finished	

- A vending machine prints the following information:

State	Meaning	Additional Information
S	starting	cost per bottle
r	start reloading by truck	
R	complete reloading by truck	
B	student bought a soda	flavour of soda purchased, amount remaining of this flavour
F	finished	

Information is buffered until a column is overwritten for a particular entry, which causes the buffered data to be flushed. If there is no new stored information for a column since the last buffer flush, an empty column is printed. When an object finishes, the buffer is flushed immediately, the state for that object is marked with F, and all other objects are marked with "…". After an object has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in internal representation; **do not build and store strings of text for output**.

The driver is supplied as source and runs the simulation (change it at your own risk). It reads and parses the simulation configurations, and creates in order the printer, WATCard office, name server, vending machines, bottling plant, and students. It then invokes the action routines of the students and the bottling plant in random order until all of the students have purchased their specified number of bottles. Finally, the students and then the vending machines are deleted in random order, followed by the remaining objects in reverse order to creation. The printer is supplied as an object file.

The executable program is named soda and has the following shell interface:

```
soda [ config-file [ random-seed ] ]
```

config-file is the text (formatted) file containing the configuration constants. If no value for config-file is specified, use the file name soda.config. random-seed is a positive integer used as the seed for the random number generator so it is possible to generate repeatable output. If no value for random-seed is specified, the random number generator is initialized with an arbitrary seed value.

To obtain repeatable results, all random numbers are generated using class PRNG. Excluding the given calls in the driver to obtain random values: there are four calls in the BottlingPlant (1 in BottlingPlant::BottlingPlant, 3 in BottlingPlant::action), there are two calls in Student::Student, there is one call in VendingMachineCardEater::buy, and there is one call in WATCardOffice::transfer. All random rolls (1 in N chance) are generated using prng(N-1) == 0.

Do not, under any circumstance, try to code this program all at once. Write each class separately and test it before putting the pieces together. Play with the sample executable to familiarize yourself with the system before starting to write code.

Road map:

July 11: All supplied files should be copied and a Makefile created. For each of the given classes, put in calls to the printer where possible to see how output is generated. For example, for the truck do the following:

```
Truck::Truck( ... ) { prt.print( Printer::Truck, 'S' ); }
Truck::~Truck() { prt.print( Printer::Truck, 'F' ); }
void Truck::action() {  prt.print( Printer::Truck, 'P', 5 ); }
```

You can play with different printer calls to see how to control output.

July 18: 4 of the classes should be coded, while the other 3 classes should just generate sufficient values to test the first 4 classes. Start preparing the test documentation by thinking of interesting aspects of the classes and test cases that illustrate these aspects. Use these test cases to begin testing the 4 classes under development.

July 25: The remaining 3 classes should be coded and the UML diagram should be complete. As well, finalize the test documentation.

Submission Guidelines

This assignment should be done by a team of two people because of its size. Both people receive the same grade (no exceptions). Either member may submit to Marmoset. The instructor and/or instructional-coordinator will not arbitrate team disputes; team members must handle any and all problems.

Please follow these guidelines carefully. ***Review the Assignment Guidelines and C++ Coding Guidelines* before *starting the assignment.*** Further information about the guidelines will be discussed in class. Please note that **each text file, i.e., *.*txt file, must be ASCII text and not exceed 750 lines in length, where a line is a maximum of 120 characters.**

You must create and use a makefile for this assignment:

Makefile – construct a makefile with target soda, which creates the executable file soda from source code in the makefile directory when the command make soda is issued. This Makefile must be submitted with your assignment and is used to build the programs, so it must be correct.

Submit the following to Maroset:

[100 Marks] 1. Place all of your program files (Makefile, *.{h,cc,C,cpp} and printer.o) into a5.zip and submit to project **a5code** of CS246_PROJECT.

The *.{h,cc,C,cpp} files are the code for this assignment. The program must be divided into separate compilation units, i.e., .h and *.{cc,C,cpp} files. **Program documentation must be present in your submitted code.**

Program correctness is based on output.

[25 Marks] 2. sodatest.txt – test documentation for the assignment, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.

[25 Marks] 3. soda.pdf – UML class diagram for your program. Do not get fancy and use complex tools to draw this picture. Any simple drawing tool should be able to generate the boxes and text, and generate .pdf output. A pdf copy/scan of a **well drawn** hand-made diagram is acceptable. ***The picture must fit on a single 8.5x11 page and may appear in either portrait or landscape format.***

Follow these guidelines. Your grade depends on it!

Note: The Marmoset public test will not be testing the correctness of your code. It will simply run the Makefile in your submission to generate the executable* soda *program.