2009 November 23

Example Test Questions for Designing Systems

1.

You will design and implement classes for cars. There are three types of cars: **Sedan**, **Compact**, and **Sports**. Each car contains an integer amount of gas. All cars respond to the *gas* feature, which increments the amount of gas by 1 up to a maximum of 50, and prints out "Gas!" to the standard output. Except the **Compact** car, which after printing "Gas!" also prints "Well, I have to work hard again". All cars respond to the *accelerate* feature, which decreases the amount of the gas by 1 and increases the speed of a car by 1. A car prints out "Faster!" when it receives the *accelerate* feature. However, **Sports** cars get so excited that they prints "It really feels good!" which costs the Sports car two more units of gas, but increases the speed by another one. All the cars respond to the *brake* feature, which decreases the amount of the gas, as well as speed, by 1. The speed is represented by an integer between 0 and 200, except for a **Sports** car, which can reach 300. When the speed is greater than (3*gas – 50), a car responds by printing "Speeding!" A **Sedan** car will further complain "Why hurry?" The exertion of printing the extra words costs Sedan an additional unit of gas

Draw a class hierarchy in BON, which specifies the relationships among all classes you designed (no interface details). Also, implement your classes in Eiffel with suitable contracts and invariants. Your design should be general enough to easily add new types of cars.

2.

A vending machine sells Coke and Powerade. It accepts either one dollar (loonies) or two dollar (twoonies) coins. A Coke costs one loonie while a Powerade costs one twoonie (or two loonies). Assume that the machine is fair: it will always give the requested product if enough money has been deposited, and it will never accept money if a product is unavailable. The machine return changes by printing out the amount that should be returned.

You're to write an Eiffel class, VENDING_MACHINE (or VM) that models the vending machine. Use following features:

deposit_coin : puts in a loonie or twoonie
no_coke : true iff there are no Coke left
no_powerade : true iff there are no Powerade left
cokes : an array of Cokes
powerades : an array of Powerades
paid : returns the total amount of money paid so far
get_coke : returns a Coke
get_powerade : returns a Powerade

Assume that you have two classes: COKE and POWERADE for use. Therefore, you don't have to implement these two classes. The only feature they have is `make` to return a new instance of each, and `count` to to state how many can still be "made". Now complete the implementation of the class VM in Eiffel including class attributes, routines and contracts (assertions). You can add any number of new features to VM, but they must be private. Furthermore, please explicitly state any assumptions you made.

3.

For this problem you will design classes for a cactus patch. It turns out there are three types of cacti: *Sad*, *Stoic*, and *Angry*. Each cactus contains some integer amount of water. The only two events which ever happen in the life of a cactus are intense sunshine and occasional rain. Cacti lead pretty simple lives, waiting eagerly for weather systems to pass.

All cacti respond to the *water* method. Whenever a cactus receives a *water* message its amount of water goes up by 1 up to a maximum of 100. The cactus is so excited that something happened that it prints "*Water!*" to standard output. Except the *Stoic* cactus which, after the "*Water!*", also prints "*(well ... not that I care.)*".

All cacti also respond to the *sun* method. Whenever a cactus receives a *sun* message, it prints to standard output "*Sun!*". The sun also cause the cactus' amount of water to go down by 1, but it cannot go below zero. This may cause the cactus to become unhappy. The happiness factor of a cactus is generally (2**water*) – 5, but with a maximum value of 20.

Sad cacti are the exception – they are always exactly 10 units less than a normal cactus with a maximum happiness of 10. A cactus is "unhappy" whenever its happiness factor is negative. When unhappy, and only when unhappy, a cactus responds by printing to standard output "*Not happy!*". Except the *Angry* cactus, which instead responds by printing "*REALLY not happy!*". The exertion of printing the extra word so reduces the *Angry* cactus an additional unit of water.

Now draw a top level class hierarchy in BON and implement your classes in Eiffel. Make sure your design is general enough for any one to add a new type of cactus easily.

4.

A vending machine sells Coke and Powerade. It accepts either one-dollar (loonies) or two-dollar (toonies) coins. A Coke costs one loonie while a Powerade costs one toonie, or two loonies. Assume that the machine is fair: it will always give the requested product, if enough money has been deposited, and it will never accept money if a product is unavailable. The machine returns change by printing out the amount that should be returned.

You are to write an Eiffel class VM (for Vending Machine) that models the vending machine. Use the following features.

- deposit_coin: puts in a loonie or a toonie
- no_coke: true iff there is no Coke left
- no_powerade: true iff there is no Powerade left
- cokes: an array of cokes
- powerades: an array of Powerades
- paid: returns the total amount of money paid so far
- get_coke: returns a Coke
- get_powerade: returns a Powerade

5.

Design a COKE_MACHINE class. You can assume that you have two reusable classes: COKE, representing a soft drink, and COIN, representing a coin. The class has three attributes: a list of COIN's called **money**, which represents the money that has been inserted by customers; a list of COKE's; and a constant **size**, the maximum number of cokes in the machine.

Write interfaces and contracts for the following features of COKE_MACHINE.

- **paid**: an attribute that is true whenever a coin has been entered and a coke can be taken out.
- **make**: the creation procedure
- **empty**: a feature that is true whenever the machine is out of cokes
- **put_in_money**: a feature that puts a coin in the machine
- **take_out_coke**: a feature that takes a coke from the machine, providing that the customer has already paid.

Your contracts should be written using Eiffel's assertion language; you should not need to use BON assertions. Provide implementations for the features **empty**, **put_in_money** and **take_out_coke** in Eiffel.

Give a class invariant expressing the constraints on using the machine.