# Simulation of a Petrol Station Level 1

**REQ:**

* **Custom #**. of Pumps & Tills
* **Diff** Vehicle Types

**L1\_REQ:**

* **Inc** of store,
* **Inc/Excl** of Trucks
* **GUI**

**Future Changes:**

* Store: **probability of purchases**
* Vehicles: **Size of gas tanks and their behavior**

**EX\_REQ:**

* **More** Vehicle Types
* **Multi** Fuel types **@ Diff** Prices
* Away Parking (**During** Shopping)
* Vehicle **Breakdown**
* **Adjust**able Vehicle probabilities
* **Adjust**able Vehicle Gas Tanks

**Extra Details:**

Runs at **1T**ick = **10S**econds time frame. [tfore **6T**icks = **1M**inuite]

Every **Range** **R** has a **uniform probability**.

**Pump**s **Fills** @ **1G**allon per **T**ick

**Pump** Default **Cap**acity: **3U**nits

**Paying at Till** can take **R** **2-3M**in ~ **R** **12-18T**icks

**One Vehicle** appears **per Tick** [based on stacked spawn probability]

Vehicle always **go to least occupied Pump/Till**

Vehicles **always fully fill tank**

**Vehicle -> front** of Pump, **waits a tick** before stating fill

**Vehicle Full, waits a tick -> goto Till**

**L1 might goto store**

**Spend /+time at store**

Goto till on next tick??????????????????????

**Spend** **2-4M**inutes **@Till**

Track:

**Money Made** (from Pump)

**Money Lost** (from pumps being full)

**L1 Money Made** (from sales)

**L1 Money Lost** (from missed sales)

Need at least **5 JUnit Test** Classes

# Steps of Simulation:

**Main Process**

Sub Process

**Input ->**

**UserInput**

Args[] >> UserInput.input

Accessor get p, q, pricePerGallon, numTicks, numPumps, numTills, Seed

**L1:** withTrucks

**Set-up**

Create Pumps[numPumps] pumps, Tills[numTills] tills, Random generator(seed),

double #PRICE\_PER\_GALLON,

int #GlobalCounter.

--Start of Tick

**Construct Vehicle**

generator.nextDouble() , < p…2p < …2p+q < L1:….2p+q+t0

(true) -> create Motorbike/Car/FamiltySedan/L1:Truck

**Vehicle -> Pump**

Pump.checkSpace()

(True) -> add to queue

(False) -> Vehicle.getPumpRevenue() & **(L1)** Vehicle.getStoreRevenue() BREAK

**Pump fill-up**

While Vehicle.isFull is false

//needs to wait one tick once getting to front of pump before starting to fill

decrement fuelTillFull

if fuelTillFull =0, make isFull true

//on next tick after becoming full, goto next place – stays at front of Pump

**(L1) Shop**

//spends TicksShopping Ticks shopping

While Vehicle.isDoneShopping

// same as in fill-up – may use Local class to reduce duplicate functions

shopRevenue+=Vehicle.getShopExpenditure;

**Pay at Till**

//basically same as Shop – but at the end add to Pumprevenue

//spends TicksPaying Ticks at the Till

While Vehicle.isDonePaying

// same as in fill-up – may use Local class to reduce duplicate functions

pumpRevenue+=Vehicle.getPumpExpenditure;

**Vehicle leaves**

If Vehicle.isFull & L1.isDoneShopping & .isDonePaying

Remove from front of Till

Remove from front of Pump

--end of Tick

**Deconstruction /Reset**

Clear all collections

reset values to after “Set-Up”

can ignore fields such as “output” that will get overwritten

**Output Results**

// maybe also output details of station (the input)

String output = fuelRevenue, fuelLost, shopRevenue, shopLost, totalRevenue, totalLost

“feildName: “+fieldValue+”\t”

output (System.out.println) or write to file.

**Motorbike**:

Fixed Probability of appearing (**p**)

Unit\_**Size: 0.75**

**Tank**\_Size: **5** (gallons)

**L1**: goto**Shop** **{false}**

**Small Car:**

Fixed Probability of appearing (**p**)

Unit\_**Size: 1**

**Tank**\_Size: **R{7-9}** (gallons)

**L1**: goto**Shop** **{if(Timer < 5M|30T~prob=0.3 ) spend £5-10, 2-4Minutes** (12-24T)**}**

**Family Sedan:**

Fixed Probability of appearing (**q**)

Unit\_**Size: 1.5**

**Tank**\_Size: **R{12-18}** (gallons)

**L1**: goto**Shop** **{if(Timer < 10M|60T~prob=0.4 ) spend £8-12, 2-5Minutes** (12-30T)**}**

**L1: Truck:**

Dynamic Probability of appearing [**Initial t0**]

**(Unhappy)** - > reduce *t* by 20% of its current value: ***t*0 =0*.*8*t*.**

**(Happy)** -> increase it by 5% of its current value, up to the original value of ***t*: *t*0 = min{1*.*05*t,t*0}.**

Unit\_**Size: 2**

**Tank**\_Size: **R{30-40}** (gallons)

**L1**: goto**Shop** **{if(Timer <= 8M|48T~always ) spend £15-20, 4-6Minutes** (24-36T)**}**

# Results:

Need to run simulation for 4 Hours ~ **1440 Ticks**

Results needs to be an average of results from **10 different seeds**

**p**: 0.01, 0.02, 0.03, 0.04, 0.05

**q**: 0.01, 0.02, 0.03, 0.04, 0.05

**#of Pumps**: 1,2 & 4

**#of Tills**: 1,2 & 4

**L1: /w/wo Trucks**

225 / L1 450 combinations

(\*seeds \* ticks) 3,240,000 / L1 6,480,000

# Submission:

**Deliverable Deadline: 5:00pm Thursday 4th May**

Online trough Blackboard: .zip file ~ Needs to run on Eclipse running JRE 8

1. **Source files** (Inc. JUnit test classes)
2. **HTML Doc**umentation (Javadoc)
3. Written (group) **Report**
   1. .PDF file,
   2. *[3-4 pages]* “brief” **Description of design**. **Distinguishing between** **library** **components** (reusable) and **client code** (specific to application) **Explain**ing **rational** **behind design decisions**.
   3. **L1:** *[0.5 pages]* “brief” Description of **changes** needed to our library classes for simulation **to support extra req**uirements. @see: **EX\_REQ**
   4. *[1-2 page(s)]* **UML: Class** Diagram.
   5. **L1:** *[1 page]* **UML: Sequence** Diagram.
   6. Simulation **Results** (tabular form).
   7. *[.5 page]* “brief” **discussion of results**, their **implications**.
   8. *Listing of* ***source code as PDF*** *files*
   9. *Instructions on* ***how to Build/Run*** *the program*

Need to state Level 1, Group 16

Oral Exam for each group member about the submission

Individual Single Page Report about:

1. “brief” description of contribution.
2. Discussion of what you’ve learnt from carrying out the implementation.

# Marking:

Level 1: 100/100: 25-Design,

Level 2: 75/100:

As well as the group submission, marks will be determined by

Individual:

1. Report
2. Performance in the oral exam
3. Attendance at meetings

Marks will be awarded for

* Quality of the implemented software
  + Flexible & Extendable design
  + Good Commenting
  + Robust & Defensive programming
  + Consistent programming style

# Milestones:

Requirement Analysis: Analyse the spec and generate a system architecture (noun-verb method).

Design: Create a UML: Class Diagram.

Unit Tests: Try writing some Unit Tests before writing any of the body of the program.

GUI Design: sketch GUI design (wireframe).

# Strengths

1. Good use of encapsulation in the design.

2. Good subsystem decomposition (including model/interface).

3. Well-structured GUI code.

4. Clear separation between service and client functions.

5. Good use of inheritance and dynamic binding for run-time polymorphism.

6. Good use of abstract base classes and interfaces.

7. Object creation correctly performed.

8. Avoiding global variables.

9. Appropriate use of final and static in member functions.

10. Appropriate use of final variables in place of numeric constants.

11. Headers with filename, description, author, and version in all files.

12. Clear design documentation.

13. Unit tests that adequately cover potential errors