

Coursework Specification

Module:	COMP1202: Programming 1		
Assignment:	Programming coursework	Weighting:	40%
Lecturers:	David Millard, Heather Packer, Steve Snow, Thai Son Hoang		
Deadline:	4/Dec/2018, 17:00	Feedback:	15/Jan/2019

Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object oriented programming.
- Can correctly use polymorphism and I/O.
- Can write code that it is understandable and conforms to good coding practice.

Contacts

General programming queries should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to Steve (s.snow@soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to Steve (s.snow@soton.ac.uk) or David Millard (dem@ecs.soton.ac.uk), ideally before the submission deadline.

Instructions

Late submissions will be penalised at 10% per working day.
No work can be accepted after feedback has been given.
You should expect to spend up to 50 hours on this assignment.
Please note the University regulations regarding academic integrity.

Smart Home Simulator



Specification

The aim of this coursework is to construct a simple simulation of the energy consumption- and energy generation- of a smart house. In the house are a number of appliances which consume/generate electricity or consume water when they operate, meters that record the amount of water and electricity consumed/generated and a battery which stores excess electricity if the house happens to generate more than it needs.

You are not required to have any knowledge of smart homes or household appliances in order to complete this coursework and no accuracy is claimed in the representation of these objects presented here. The smart home bears some superficial similarities to a real house but is grossly simplified and in most cases likely to be quite different to how real smart homes, meters and appliances might work.

Your task is to implement the specification as written.

In some cases, specific names are defined or specific properties given for classes. This is mainly to ensure that we can easily read your code, and find the parts that are most relevant to the marks scheme. You will not gain marks for deviating from the specification in order to increase the realism of the simulation, in fact it may cost you marks and make it harder to assess what you have written. In some cases we wish to know that you can use specific Java constructs, so if it specifies an `ArrayList` then we want to see you use an `ArrayList`. There might be other ways of implementing the same thing but we are trying to assess your ability to use specific constructs. Where the specification is less explicit in the way something is implemented (for example, by omitted what data types to use) you are free to make your own choices but will be assessed on how sensible those choices are. Comments in your code should explain the choices you have made.

A FAQ will be kept on the notes pages of answers that we give to questions about the coursework. If issues are found with the specification it will be revised if necessary. If questions arise as to how certain aspects might be implemented then suggestions of approaches may be made but these will be suggestions only and not defined parts of the specification.

How the Smart Home works

For this coursework you are expected to follow the specification of the house, meters and appliances as set out below. This will not correspond exactly to a real house but we have chosen particular aspects for you to model that help you to demonstrate your Java programming.

There are meters, appliances, and energy sources that will contribute to this simulation. For our purposes these include:

Meters: That monitor the consumption of different utilities (where a utility is a resource like electricity or water). In addition there is a variant of a meter, called a battery meter that has some ability to temporarily store some of that resource for a later time.

House: Meters are installed in a house that will contain a number of appliances connected to the meters and which can simulate those appliances being used.

Appliances: Different appliances are used during the day and different utilities. Some appliances such as solar panels and wind turbines can also generate a utility. All appliances are grouped together depending on how they consume/generate a utility and fall into one of four categories (1) *CyclicFixed*, (2) *CyclicVaries*, (3) *RandomFixed* and (4) *RandomVaries*. These categories are explained below.

Table 1. Categories of the appliances.

Category	Description	Examples
<i>CyclicFixed</i>	Switched on for a set cycle , i.e. a fixed number of hours each day and uses a fixed amount of resource each hour. Always-on appliances fall into this category, with a cycle length of 24 hours. For simplicity appliances will always start this activation at the start of the 24 hour period.	Fridge: Always on Treadmill: The occupant runs marathons and runs on the treadmill for 3 consecutive hours each day.
<i>CyclicVaries</i>	Switched on for a set cycle , i.e. a fixed number of hours each day (as per <i>CyclicFixed</i>), but use a variable amount of resources in each hour that they are switched on (a random number within a min-max range, see Table 2).	The cooker is used for 2 consecutive hours each day, but the amount of power used varies depending on what's being cooked. Solar panels generate only during daylight hours and generation varies each hour according to cloud cover.
<i>RandomFixed</i>	Switched on at random times during the day – each has a probability of being on during any given time increment (refer to the probabilities listed in Table 2), and consumes a fixed amount of resources when it is switched on.	TV, Computer, Kettle: the occupant has no set habits for watching TV, using the computer or boiling the kettle. Thus being used at random times of the day, all these appliances fall under <i>RandomFixed</i> .
<i>RandomVaries</i>	Switched on at random times during the day (as per <i>RandomFixed</i>), and use a variable amount of resources in each hour that they are switched on (a random number within a min-max range, see Table 2).	The wind turbine generates power at random times of the day depending on the wind and generates a variable amount of electricity per hour, depending on how strong the wind blows. The hose is used randomly during the day and consumes a variable amount of water according to how strong it is turned on each time.

Note: We model a day as 24 one-hour time increments

The table below describes the appliances in each category. Note one time increment = 1 hour.

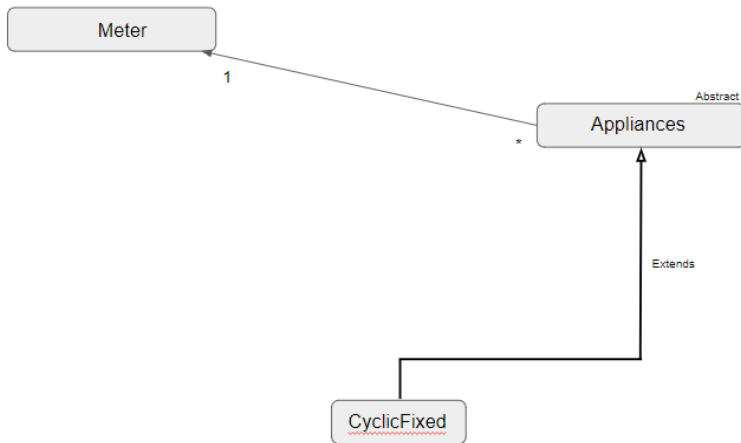
Table 2. Example appliances in each category (note this table is different to the configuration file, located on the Student Wiki, that we will introduce later in the document).

Appliance	Class	Min units consumed (per hour) (# units)	Max units consumed (per hour) (# units)	Fixed units consumed per hour (# units)	Probability switched on (1 in n)	Cycle length (/24)	Utility
Lights	CyclicFixed			6		5/24	Electric
Fridge	CyclicFixed			2		24/24	Electric
Cooker	CyclicVaries	4	18			2/24	Electric
TV	RandomFixed			2	1 in 6		Electric
Wind turbine	RandomVaries	-1	-25		1 in 2		Electric
Electricity = £0.013 per unit							
Water = £0.002 per unit							

Random number ranges (both here and on the configuration file) should be inclusive. For appliances that generate a utility, you should model this by having a negative value for their consumption. For the RandomFixed and RandomVaries appliances, run a probability each hour to determine whether or not it's switched on in that hour. Note that a 1 in 6 probability of it being switched on does not mean it will always be on for one hour in every six.

The next sections will take you through the construction of the various meters, appliances and buildings that are required. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your simulation. The first steps will provide more detailed information to get you started. It is important that you conform to the specification given. If methods are **written in red** then they are expected to appear that way.

Part 1 – Implement the meter class and appliance class



Meter

The first class you need to create is the Meter. This represents an object that manages the consumption and production of a particular utility. Later we will create two instances of this class – one to record water use and the other to record electricity use.

Define the **Meter** class. This should have three private properties:

- `utilityName` – this is a string that describes the type of utility (e.g. “electric”)
- `unitCost` – the cost of one unit of this type of utility (double)
- `meterReading` – a variable (float) representing the balance of units that have been used since the last meter reading. This is the variable that will be increased whenever an appliance consumes units, and decreased every time an appliance generates units.

Create a constructor that initialises these three values.

Now create a public method in the Meter class called **consumeUnits**, which takes a number of units as a parameter and adjusts the `meterReading` appropriately.

Meters should also have the ability to **report()** the amount of units of the utility that have been used and the corresponding cost incurred to the home occupier this report should be in the form of printing the `utilityName`, `meterReading`, and the cost associated with that meter reading (be multiplying `meterReading` by `unitCost`) to `System.out`. You may choose how to present the data, but it must be clear and readable. Calling the report method should reset the `meterReading` to zero. Finally, the method will end by returning the cost that you calculated.

*You will want to test your newly created **Meter** class by using a main method to create a new instance of a Meter, calling `consumeUnits` a number of times, and then calling `report` to see the effect.*

Appliance

The next class you will need to create is an abstract class called Appliance. This defines the basic properties and methods that all the different appliance classes (`CyclicFixed`, `CyclicVaries`, `RandomFixed`, `RandomVaries`) will use. The properties to define are:

- `name` – the name of this appliance as a string (e.g. for a fridge this would be “Fridge”)
- `meter` – the Meter instance that this appliance uses to record its consumption and production

Define an appropriate constructor for the Appliance that initialises the Appliance with a name. Define a public setter method to set the `meter`.

The abstract Appliance class should also have a public abstract method called **timePasses()**, which denotes one increment of time passing (1 increment of time = 1 hour). Each concrete class that inherits from

the abstract `Appliance` class will have to provide code that determines how it consumes or produces a utility when `timePasses` is called.

To model the various types of `Appliance`, you will use inheritance. You will need to create four new classes called `CyclicFixed`, `CyclicVaries`, `RandomFixed`, `RandomVaries` that will all inherit from the abstract class `Appliance`. **Start off by creating `CyclicFixed`.**

Define your third class, a concrete `CyclicFixed` class that inherits from `Appliance`. Define properties for how many units it consumes, and for how long in a 24 hour period it is active. The constructor should initialise these properties, and check that that the number of hours active is a value between 1 and 24.

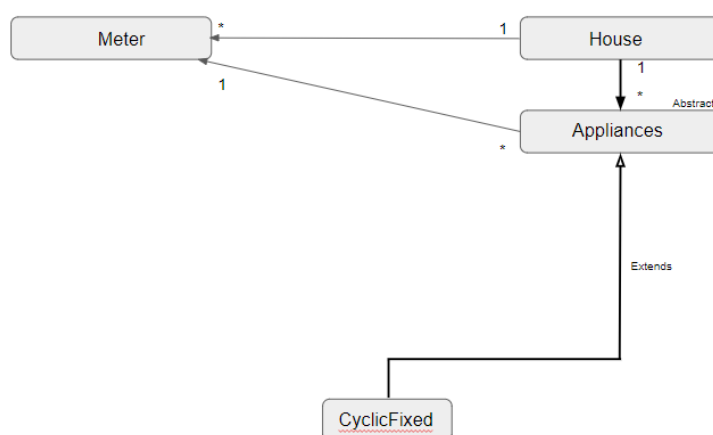
Next, override `timePasses` (in the `CyclicFixed` class) with code that uses the properties above to calculate how many units this appliance has used in this time period. Remember that this type of appliance is on and consumes its utility in each of the first n hours of each 24 period, where n is the number of hours it is active. You should add properties to the `CyclicFixed` class as necessary to enable this functionality.

Finally, you need to update the `Meter` associated with this appliance by calling `consumeUnits` with the value that you have just calculated. To do this you will need to add a protected method `tellMeterToConsumeUnits()` to the Abstract `Appliance` class that calls the public method `consumeUnits()` on the `Meter` associated with this appliance. Use whatever parameters you think are appropriate to ensure that the right number of units are recorded by the `Meter`.

You will want to test your newly created classes using a main method to create a new instance of a `CyclicFixed` `Appliance`, and another of a `Meter`, and then linking them together using the setter method in the appliance. Now you should be able to call `timePasses` on the appliance, and see it alter the values in the meter.

BY THIS STAGE you should have an abstract `Appliance` class, connected to a `Meter` class and the first subclass of `Appliance`: `CyclicFixed`.

Part 2 – Add House, run `timePasses()`



The next step is incorporating the `Meter` and `Appliances` within a class called `House`.

The `House` class will have two types of `Meter` attached to it, one for electric and one for water. The constructor should create these when an instance of the `House` is created.

Inside the House will be a number of Appliances, each attached to the appropriate Meter. You should use an **ArrayList** to represent the Appliances in the House.

The House will need a number of methods. These should include:

- **addElectricAppliance(Appliance)** – adds the specified Appliance to the House, and sets its meter to be the House electric meter.
- **addWaterAppliance(Appliance)** – adds the specified Appliance to the House, and sets its meter to be the House water meter.
- **removeAppliance(Appliance)** – removes the specified Appliance from the House.
- **numAppliances()** – this returns the number of Appliances in the House.

You should now have a House class that you can add Appliances to.

You will want to test your newly created House using a main method to create a new instance of a House and then adding instances of the CyclicFixed Appliances listed in Table 2.

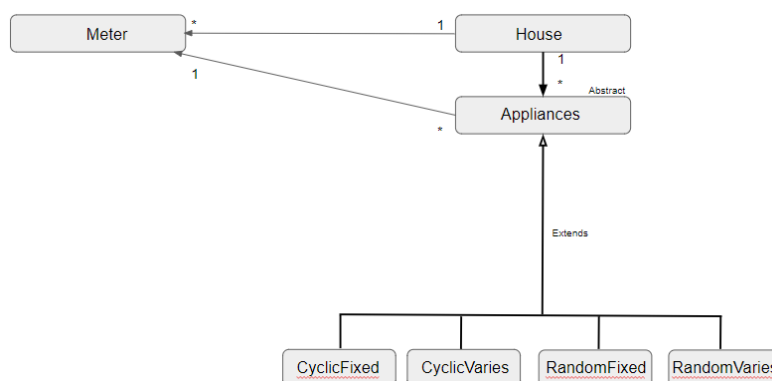
Calling timePasses()

Add a method to the House called **activate()** which simulates one unit of time (an hour) passing in the house, by calling **timePasses()** on each of the appliances in the house once. Once this has been done it will call **report()** on each of the house meters. The **activate** method should return the sum of the costs reported by both meters.

You will want to test your new activate method using a main method to create a new instance of a House and then adding the instances of the CyclicFixed Appliances listed in Table 2, and then calling activate.

BY THIS STAGE you should have implemented the appliances in the *CyclicFixed* class, connected the *Meter* and *Appliances* to the House, and be able to call **activate()** to invoke **timePasses()** on the appliances such that they increment the meter they are attached to (*CyclicFixed* appliances are all connected to the electricity meter).

Part 3 – Modelling other appliances



Next, you should implement the rest of the sub-classes of **Appliance**. This will involve working in the probabilities that a given appliance will be turned on in a given increment of time (e.g. **RandomFixed**, **RandomVaries** appliances) and determining the amount of power used in a given increment of time by calculating a random number within the range given in Table 2 (**CyclicVaries**, **RandomVaries**).

You will need to use a random number generator to determine the number of units consumed per hour (for `CyclicVaries` and `RandomVaries`) and the probability that it is switched on in a given hour (for `RandomFixed` and `RandomVaries`). You can do this with the `Toolbox` class that was introduced in Lab 1, or by using the methods in the `java.util.Random` class directly.

Once you have created each `Appliance` subclass you will need to create an appropriate constructor for it.

When `timePasses()` is called on an appliance it will typically go through the following procedure.

- Determine whether the appliance is on, based on either
 - o (1) the time of the day (`CyclicFixed` and `CyclicVaries`), or
 - o (2) the outcome of running the probability coefficient specified in Table 2 (`RandomFixed` and `RandomVaries`).
- If the appliance is off then do nothing.
- Else if the appliance is on increment the meter it is attached to by the amount specified by either
 - o (1) the value specified on construction (`CyclicFixed`, `RandomFixed`), or
 - o (2) a random number within the range of Min and Max specified in Table 2 (`CyclicVaries`, `RandomVaries`)

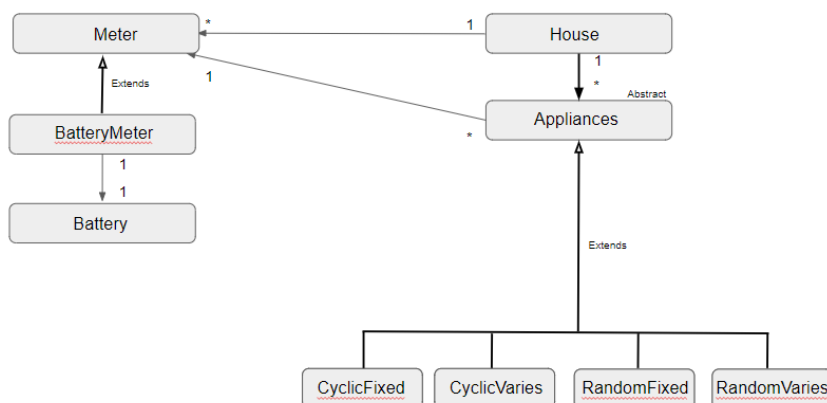
Note, that according to Table 2 some of these new appliances can generate a utility, which means that `consumeUnits` in `Meter` will be called with a negative number. It is therefore possible for the `meterReading` in `Meter` to go into the negative (if more units are produced by all the appliances than are consumed). However, when `report` is called any negative value should be ignored, in such cases the cost is simply reported as zero (i.e. the excess utility generation in any given time period is wasted).

Update the `report` method in `Meter` to reflect this.

You will want to test your newly created `Appliance` subclasses by creating instances of the appliances listed in Table 2 and adding them to the appropriate meters in the `House` class. Calling `activate` in succession will then show all of the utility usage within the `House` and give a total cost for each hour.

BY THIS STAGE you should have implemented ALL of the `Appliance` classes, and be able to add them to the `House` on the appropriate `Meter`, calling `activate()` to run the simulation, generating and consuming utilities on that `Meter`.

Part 4 - Modelling the battery



Add a second meter class `BatteryMeter` which extends the `Meter` class. The `BatteryMeter` is connected to an instance of a `Battery` class which has the ability to store a utility (in this case electricity). If the utility consumed by the house over a time increment is negative (i.e. production has exceeded the consumption of the other appliances), the excess units get stored in the battery. In the next time increment, if consumption exceeds production, the battery is used to power the house. For example, if the battery had a charge of 5 units and the following time increment the house consumed 40 units: then 5 units is drawn from the battery leaving it at zero and the cost calculated as if the house had drawn 35 units from the mains at the relevant `unitCost`. The `Battery` should have a capacity limit, and any production that cannot be stored (charge reaches that capacity) is lost.

Override the `report` method in `BatteryMeter` so that it is capable of recording and reporting the amount of power drawn from the battery and the total amount drawn from the mains.

Create a new overloaded constructor for the `House` that takes two references to `Meter` as parameters, and sets the first to be the electric meter and the second to be the water meter. You can now pass in an instance of a `BatteryMeter` as the first parameter when you create the house (and water `Meter` as the second parameter), and the house will use a `BatteryMeter` rather than the electric `Meter` for electric. In this way the `BatteryMeter` replaces the electric meter in this final step.

You will want to test your classes by adding appliances to a `House` with a `BatteryMeter`, in such a way that production (at least sometimes) outstrips demand.

BY THIS STAGE you should have implemented the battery and battery meter.

Part 5 – Running the simulation

You will have already created the class called `House`. It should have an `ArrayList` of `Appliances` that are linked to the `Meters` in the house. The `House` should have an `activate()` method which is used to run the simulation for one time increment.



Define an overloaded version of `activate()` that takes a number of hours as a parameter, and simulates that number of hours passing by calling `timePasses()` on each of the appliances in the house an appropriate number of times. After each hour it should call `report` on each of the house meters. At the end it should print the total cost accumulated over however many hours it was run.

You can use print statements to record what is happening in your simulation. If you wish to slow things down (e.g. in order to better observe the house stepping through a full day), the following code will help you to pause for half a second between calls if you choose to include it.

```
try
{
    Thread.sleep(500);
    activate();
}
catch (InterruptedException e)
{
}
```

You should now be able to test your simulation. Appliances will be incrementing the relevant Meter and at the end of the time period (e.g. 24 for one day) your simulation can print out the amount of utility use recorded by each of the Meters.

Part 6 – Reading a simulation configuration file

A good simulation will allow you to set the starting conditions for your simulation and one way of doing this is for the simulation to read in a simple configuration file. For this step you will need to use file handling methods (see Lab 9 for examples) as well as split strings into different component parts.

You should modify your main method so that it can take a file on the command line. This will enable you to start your simulation by typing:

```
java House myHouse.txt
```

The configuration file (“config file”), located on the Student Wiki, will list a set of appliances (using a similar structure to Table 2) – we have provided an example file on the coursework page of the module wiki. Note that if an appliance appears twice it means that there are two instances of that appliance in the house.

You should complete this section of the coursework using a single House, with a Meter for water, and if possible a BatteryMeter for electric (unless you have not completed this, in which case a Meter for electric is acceptable).

When your program receives the configuration it should read the file one line at a time. For each line, it will need to identify the type of the appliance, and create a new instance of the appropriate class with the correct parameters, it should then add that instance to the House linked to the appropriate meter. You may find you need to create specific methods or a helper class to parse the configuration file and extract the information that the House needs. You should check for errors in the file and write appropriate error messages should they occur.

You should now have a simulator that configures itself from a file, and runs with a House containing multiple Appliances using different types of Meter. The Appliances will use resources that are recorded by the Meters in the House and the usage over a period of time will be written out by the simulation.

Part 7 – Set duration of simulation

Set a parameter on the main console as a maximum amount of time the simulation should run for. By default the simulation should run for 7 days (7 x 24 time units). The length of time the simulation runs for should also be able to be changed by passing a parameter to the main method.

Exceptions

You should be trying to use exceptions in the construction of your simulation where appropriate. You should be catching appropriate I/O exceptions but also might consider the use of exceptions to correctly handle:

- The input of a configuration file that does not conform to the specified file format.
- `timePasses` is called on an appliance but no meter has been set

- Currently there are no checks on what utility an appliance consumes and what meter it is attached to (i.e. it is possible to link a “Water” appliance to an “Electric” meter). You could add this information to the classes, and use it to raise an exception if they do not match when a meter is linked to an appliance.

Extensions

You are free to extend your code beyond the basic simulation as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. If you are in any doubt about the alterations you are making please include your extended version in a separate directory.

Scheduling and resource management can become very complicated very quickly, so we have deliberately not made efficient scheduling a part of this coursework. Be warned if you attempt to do anything clever in this regard for your extension.

Some extensions that you might like to include:

- Try adding new types of appliances such as an electric shower, dishwasher and washing machine, all of which use both electricity and water. Each of these appliances needs to be connected to both the electricity and water meters (you may want to consider using a Decorator Pattern for this). Each of these also typically have “eco” settings in which they will use less energy and water than in standard mode. Additional methods might need to be added to allow the appliance to be switched between modes.
- You might want to allow the simulation to save out the current state of a House to a file so that it can be reloaded and restarted at another time. To do this you may be able to use the current designed configuration file, or perhaps you need to extend it in some way.
- Currently the basic Meter wastes any excess production, and even the Battery wastes it once it has reached capacity. You could create a new subclass of BatteryMeter that instead uses a feed-in-tariff to reduce costs (i.e. excess utility is sold back to the utility company). You could generate a time-of-use pricing scenario, where the feed-in tariff fluctuates according to demand between one of three values, where the highest value is higher than the utility unit cost and the lowest value is lower than the utility unit cost. The cost of the utility in any given time increment varies according at random (or according to the time of day). Depending on the feed-in-tariff, your battery has to choose whether it uses its charge to power the house (if feed-in-tariff < unit cost) or to sell back to the grid (if feed-in tariff > unit cost).

15% of the marks are available for implementing a reasonable sized extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt lots of extensions in order to gain these marks but complexity will be considered, so simply adding more appliances wouldn’t in itself be enough to gain all the marks. Please describe your extension clearly in the readme file that you submit with your code along with any special instructions required to run it.

Space Cadets

You might want to add a GUI to your simulator so that you can visualise the state of the *House* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it, although you can reasonably expect praise and glory for your efforts.

Submission

Submit **all Java source files** you have written for this coursework in a zip file **house.zip**. **Do not submit class files**. As a minimum this zip will include:

House.java, Meter.java, BatteryMeter.java, Battery.java, Appliance.java, CyclicFixed.java, CyclicVaries.java, RandomFixed.java, RandomVaries.java.

Also include a text file **readme.txt** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Tuesday 4th December 2018 16:00** to: <https://handin.ecs.soton.ac.uk>

Relevant Learning Outcomes

1. Simple object oriented terminology, including classes, objects, inheritance and methods.
2. Basic programming constructs including sequence, selection and iteration, the use of identifiers, variables and expressions, and a range of data types.
3. Good programming style
4. Analyse a problem in a systematic manner and model in an object oriented approach

Marking Scheme

Criterion	Description	Outcomes	Total
Compilation	Applications that compile and run.	1	20
Specification	Meeting the specification properly.	1,2,4	40
Style	Good coding style.	3	25
Extensions	Completion of one or more extensions.	1, 2,4	15

REFER FULL MARKING SCHEME BELOW

Programming 1 Coursework Marking Scheme 2018/19

Compiles (Out of 10)

- 0 - doesn't compile at all
- 2.5- almost nothing compiles
- 5 - some bits compile but not all, or all compiles but only a few classes presented
- 7.5- has classes for Meter, House, Appliance, and most sub-classes, almost all compile
- 10 - Has classes for Meter, House, Appliance, Battery and all subclasses in specifications- all compile

Runs (Out of 10)

- 0 - doesn't run at all
- 5 – runs, but limited or unclear functionality
- 10 – runs and has all the functionality requested in the specification

Specification (Out of 40)

Criteria is broken down to look at FIVE core parts of the specification (*8 marks x 5 core parts = 40 marks*). (1) House and Meters, (2) Appliances (abstract class and sub-classes), (3) timePasses (including duration of simulation parameter- see Part 7), (4) Battery, and (5) I/O configuration parsing (from the Config file). Each of these five core parts will be awarded marks according to:

- 0 - Part not attempted.
- 1-3 Some code, class shells, no significant steps to working code, no real functionality.
- 3 – 4.5 Class attempted showing some promise but buggy or incomplete, not meeting spec.
- 4.5 - 6 Class is functional, but not always meeting spec, and/or shows poor programming structure.
- 6 -7.5 Complete and correctly meets the spec, but some room for improvement. This is a comfortable first.
- 8 Part completed almost flawlessly. Very little room for improvement here

Style (Out of 25)

- 0 - 10 Very poor coding style, bad use of methods and class structure, hard to follow. (Failing)
- 11-13 Poor style, lots of errors. Might include bad variable and class naming, poor method specification, odd parameter and return types, bad use of scoping, odd use of loops or if statements, unnecessary code or variables, poor choice of types, inappropriate commenting etc.
- 14-17 Passable work, some mistakes along the lines of the above, but generally acceptable code.
- 18-21 Very good coding, minor problems and things that could be improved. Could always understand what the code did but there may occasionally be better ways of achieving it. This is a comfortable first.
- 22-25 Excellent work, very little that could be improved on, would be a joy to maintain or extend this code.

Extensions (Out of 15)

Students should be able to get full marks from implementing one modest extension well, so pick their most significant extension (hopefully described in their text document) and look at

0 No extensions attempted.

1-6 Poor extension that doesn't work properly, is poorly specified or clumsily implemented.

7-11 Good extension, works, does something interesting, but might have been better implemented and may be hard to easily work out how it works or what it is intended to do.

12-15 Very good extension, well thought out, provides interesting additional functionality and is implemented in a well-structured and elegant way.