Design and Behaviour

Purpose of Simulation

The aims of this simulation consist of the following key features:

- Modelling a building, consisting of 7 floors, and a time resolution of 10 seconds (1 tick).
- The simulation should represent a working day (8 hours or 2880 ticks).
- A lift which transports different types of people to their randomly generated (within ranges) destination floor.
- Spawning a user specified number of employees and developers, who are split between two types of rival developers: Mugtome and Goggle.
- Spawning clients into the building with a probability of q, who are given priority when entering the lift.
- Employees and developers who choose to move to a different floor with probability of p.
- Spawning maintenance crew into the building with a probability of 0.005 per tick.

Data inputs	**Data outputs**
Seed number – This is used to create a consistent random object within the simulation.	Current tick number – This shows the current position in time in the simulation.
P – This value is used to determine the probability that a developer or employee will change floor.	Position of lift – This shows the current floor the lift is positioned in the simulation.
Q – This value is used to determine the probability that a client will arrive in the building.	Capacity of lift – This shows the current weighted capacity of the lift.
Number of employees – This sets the number of employees at the start of the simulation.	Movement of people – This shows the position of each person in the simulation.
Number of developers - This sets the number of developers at the start of the simulation.	Complaints and time of complaints – This shows the number of complaints, who made each complaint, and at what tick number the complaint was made.
Seed number – This is used to create a consistent random object within the simulation.	Average waiting times – This shows how long each person waited for the lift.

Key Design points:

An object-oriented approach (OOP) is used to help to reduce the complexity and improves the maintainability of the system. To achieve this, in our project we have used OOP in conjunction with encapsulation and polymorphism. An example of how OOP was implemented is illustrated in figure 1.

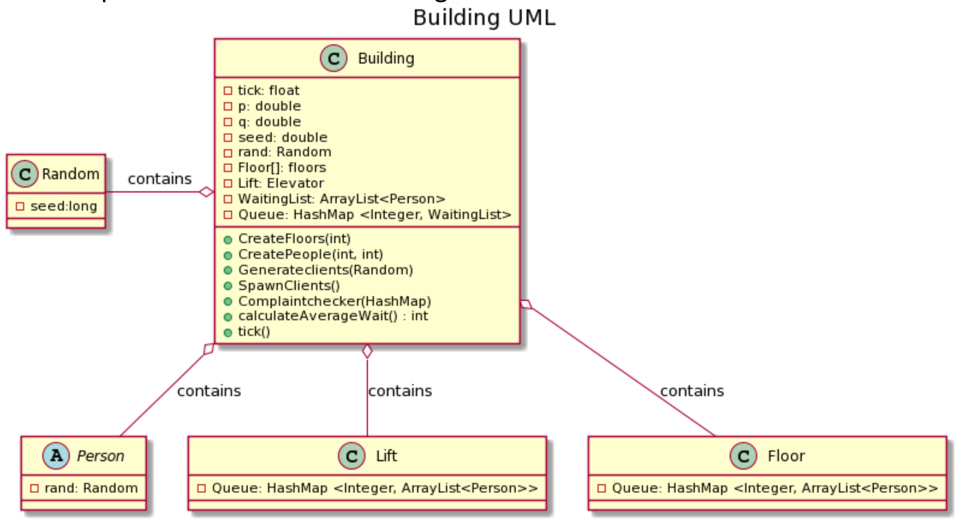

Figure 1: Represents a UML diagram which depicts the use of objects in our project

Figure 1 demonstrates how objects were used in the Building class. In our Building class numerous instances of the Floor class and a single instance of Lift and Random. Furthermore, it is illustrated how Person objects are passed around between each floor and lift object using the HashMap Queue, through the building class. A single instance of Random is also created here and passed to each Person when they are initialised in the "CreatePeople()" method.

Encapsulation is used to provide security and flexibility to a program. For example, encapsulation can be used to hide instance variables of a class from direct illegal access. Using an object-oriented approach, we can hide various methods and variables, in their corresponding package or class, using modifiers such as: protected or private respectively.
The flexibility that encapsulation offers, allows easy access to fetch and write data to a class, given that the legal access condition is met, for example when writing test scripts in the same package of a class we were able to

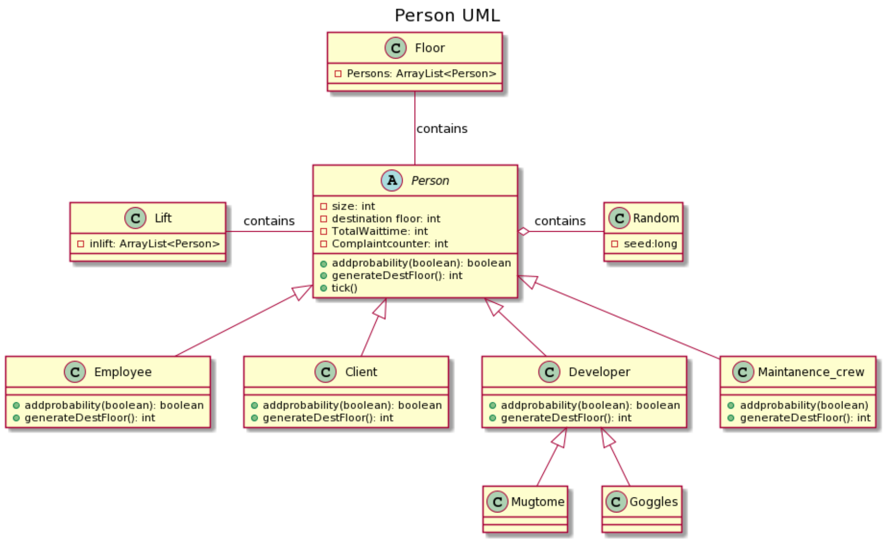

Figure 2: Represents a UML diagram which depicts the use of polymorphism in our project:

Polymorphism in conjunction with OOP will be implemented in our design, to allow some classes to share the same external interface, while also having different internal structures. In our project the abstract class “Person” is used as a base class, which every type of person inherits from, as depicted in figure 1. The methods for the Person class are overridden for each subclass of person. This is to allow the different types of people to do different things. For example, employees generate any destination floor, whereas clients are limited to only floors 0-3.
Furthermore, figure 1 also illustrates how the Mugtome and Goggles classes are also sub-classes of developer, thus inheriting its methods.
This use of polymorphism allows us to easily reuse consistent methods and variables across all subclasses. For example, setters and getter methods do not need to be re-written for each type of person, since they all share the methods in the “Person” class.

Design Pattern:
To complete our project, we attempted to follow a “Singleton Design Pattern”.
A Singleton Design Pattern was used to ensure that we could achieve a global access to the single instance of ‘Random’ object used throughout the program, in every class.
A static simulation class is used to hold a seed value. This seed is passed to the building class via a public constructor. It is in this class we created a version of a “Static factory method”. On every tick the building creates a single instance of Random, using a method called “singleinstanceofRandom()” and a seed from the simulation class. This Random object can be passed to any class via its constructor parameters whenever they are initialised.

Coupling and Cohesion
To ensure a high cohesion, we designed each class within the code to have a set of methods which are related to the intention of the class. For example, the client class focuses entirely on a client and what the client can do.
We have kept coupling low by ensuring a change in one class does not greatly affect other classes. For example, we have used getters and setters to ensure variables can still be accessed by other classes even as they change. Moreover, we designed our simulation to pass on a HashMap called “Queue” to hold data of where people, who want to change floors are, and which people are waiting to unload from the lift.
This meant, that changing the data only affects the HashMap (Queue), rather than the individual behaviour of functions. i.e. the only relationship between the floor and elevator class is a HashMap (Queue) containing people waiting for the elevator and the elevators destination floor, which is passed to the floors and elevator via the building class.