Object Oriented Programming Report

Tudor Haba
2420645

Table of Contents

Object Oriented Foundations *3*

Overview Of the Car Park App *6*

Static vs Dynamic Memory Allocation *7*

Data Structures *8*

Unit Test Results *8*

User Guide to the App *10*

References *11*

Appendix *12*

- Video 12
- Code 12

Object Oriented Foundations

Inheritance is one of the principles of object-oriented programming. Inheritance allows for child classes to share the same methods from their parent classes along with all the variables [1]. A child class is a class that is more specialized than their parent and can override the parent's methods to make it their own as well as adding some methods that are not present in the parent class [1] [2]. Fig. 1 Shows how I implemented inheritance in the application, taking the IDReader and making it able to read number plates allowing for code reusability.

```java
public class NumberPlateReader extends IDReader {  7 usages  STudor1

    public NumberPlateReader() {  3 usages  STudor1
        super();
    }

    //Check if the incoming car has a valid plate
    //car plate number -> AA11BBB
    private boolean isValid(String carPlateNumber) {  1 usage  STudor1
        super.read(carPlateNumber);

        if(getSerialNumber().length() == 7) {
            if (getSerialNumber().matches( regex: "^[A-Z]{2}\\d{2}[A-Z]{3}$") || getSerialNumber().matches( regex: "^[A-Z]{2}\\d{2}[A-Z]{2}\\d{1}$")) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    @Override  STudor1
    public boolean update(String id) {
        if(isValid(id)) {
            System.out.println("Car plate: " + id + " is valid");
            return true;
        } else {
            System.out.println("Car plate: " + id + " is not valid");
            return false;
        }
    }
}
```

Fig. 1 Inheritance being implemented in the car park app

An interface defines the behaviour that a class which implements it must add. It specifies all the common methods, their parameters and what they should return while leaving the implementation up to the classes that implement them. Interfaces allow for code reusability and grouping similar classes together [1]. Fig. 2 shows how I used a Vehicle interface and defined all the common methods a vehicle might add and then how a car class implements their own version of vehicle. See appendix for full implementation.

```java
public interface Vehicle {  12 usages  1 implementation  STudor1
    String getPlateNumber();  2 usages  1 implementation  STudor1
    String generateBarcode();  1 usage  1 implementation  STudor1
    String getVehicleInfo();  1 usage  1 implementation  STudor1
    String getBarcode();  5 usages  1 implementation  STudor1
}
```

```java
public class Car implements Vehicle {  4 usages  STudor1
    private String make;  3 usages
    private String model;  3 usages
    private String plate;  3 usages
    private String barcode;  3 usages
    private VehicleType type;  2 usages

    public Car(String make, String model, String plate) {  2 usages  STudor1
        this.make = make;
        this.model = model;
        this.plate = plate;
        this.barcode = generateBarcode();
        this.type = VehicleType.car;
    }
```

Fig 2. Interfaces and implementation

An abstract class in object oriented is meant to act as a blueprint for other classes. With an abstract class you cannot create an object from the class, it is meant for other classes to inherit from the abstract class and implement the methods or overwrite existing ones [1]. While similar to interfaces, an abstract class lets you implement the methods and also declare variables that can be used by child classes. Fig. 3 shows that I used the IDReader as an abstract class as I don't want to let the user initiate a simple IDReader.

```java
public abstract class IDReader {  2 usages  2 inheritors  STudor1 *
    private String serialNumber;  4 usages

    IDReader() {  2 usages  STudor1
        this.serialNumber = "";
    }

    public void read(String carPlateNumber) {  3 usages  STudor1
        this.serialNumber = carPlateNumber;
    }

    public boolean update(String id) {  2 overrides  STudor1
        read(id);

        if (serialNumber.isEmpty()){
            return false;
        } else {
            return true;
        }
    }

    public String getSerialNumber() {  5 usages  STudor1
        return serialNumber;
    }
}
```

Fig.3 Abstract class

Polymorphism is when a method has multiple different implementations by different classes that inherit the method from a parent. Polymorphism allows us to have the same method with different outcomes based on the class that implements it [2]. I've used polymorphism in the app to make a Vehicle array and add any types of possible vehicles to the car park. Then I used the common method getVehicleInfo to get a slightly altered version for each vehicle type.

```
private final ArrayList<Vehicle> vehiclesParked;  5 usages
```

```
for (Vehicle vehicle : vehiclesParked) {
    vehicleList += vehicle.getVehicleInfo() + "\n";
}
```

Fig. 4 Polymorphism

Generics allow methods to take in any type passed as a parameter. Generics lead to code reusability and writing more concise code. This is due to the fact that you are defining how the method acts as soon as you pass in the type when you use them [3].

Virtual methods are methods that are declared in a parent class, for example an abstract class, and they can be overridden by any classes that extend implementation from the parent class [4]. I've declared a virtual method in my abstract class, IDReader, and then I've overridden the method in the classes that extend it, IDBarcodeReader and NumberPlateReader, Fig. 5.

```
public boolean update(String id) {  2
    read(id);

    if (serialNumber.isEmpty()){
        return false;
    } else {
        return true;
    }
}
```

```
@Override   STudor1
public boolean update(String id) {
    if(isValid(id)) {
        System.out.println("Barcode: " + id + " is valid");
        return true;
    } else {
        System.out.println("Barcode: " + id + " is not valid");
        return false;
    }
}
```

```
@Override   STudor1
public boolean update(String id) {
    if(isValid(id)) {
        System.out.println("Car plate: " + id + " is valid");
        return true;
    } else {
        System.out.println("Car plate: " + id + " is not valid");
        return false;
    }
}
```

Fig. 5 Virtual methods being implemented

Overview Of the Car Park App

One of the classes in my application is the barrier class. The barrier class consists of a Boolean isUp that defines the state of the barrier at any given point in time. This Boolean can be changed by two private methods, raise and lower. Those methods are accessible to other classes by the update method that takes in the state of the barrier and calls the according private methods to update the state. The barrier is used by the CarPark class to allow the cars to enter the car parks.

Another class in my application is the sensor class. The sensor is a simple class that has a private method to detect whether there is a car present. This method gets exposed to other classes using the update method that will pass into the class if a car was detected. This class is used by the CarPark class and is used to detect whether there is a car preset at the entrance or at the exit of the car park.

An IDReader class that has read method which set the private serialNumber within the class to a string we pass in as a parameter. The method gets called in the update class which then return a Boolean if the serialNumber has been set or not. This class is broad with the scope that it is then made more specialized by the NumberPlateReader which inherits the methods from the parent class and overrides the update method to pass in a number plate as the serial number. The class also has a private method to check whether the number plate that tries to get stored is valid and follows a pattern. Another method that extends the IDReader class is the IDBarcodeReader which overrides the update to take in a barcode and validates that the barcode according to the rules within the class. Both classes are used by the CarPark to validate incoming and outgoing cars using the number plate and to validate the barcode when the car leaves to pay for the car park.

A FullSign class is used by the CarPark class to update when the car park can no longer take in any cars. The FullSign class has a Boolean that is used as a state of the car park capacity and private methods to switch it on or off. They get called within the update method by the CarPark class.

An additional class I added was the Car class. The Car class implements a Vehicle interface. The interface was added with the idea that I can then implement multiple vehicles such as cars, trucks. The Car class has a make, model, car plate, barcode and a type. The car make, model and car plate will be set up once a car is created and the type is always assigned to be a car. These cannot be changed once a car is created but can be accessed through getters. The barcode is generated using a method impletemed from the interface, generateBarcode, it will create a random barcode that will then be used when the car wants to leave the car park. The user will have to enter the matching barcode assigned to the car to leave.

The most complex class is the CarPark class. It uses all the classes from above to simulate a working car park where cars can enter and leave. The car has a capacity that

is specified when creating a new car park object. When creating the object, it will also create all the classes needed for the car park to run, FullSign, Sensors, Barrier, Readers. When a car is detected in the update method then we use the method to pass in the car plate and a Boolean to say whether the car is parking or is leaving the car park. If the car is parking, then it will do all the checks to validate the car and allow it to park whilst calling a private method that decreases the available spaces. This method also checks to see if the available spaces reached 0 in order to turn the full light on. The car also gets added to a list of all the cars that are currently in the car park. If the car tries to park when the car park is full then the car will be refused until a car leaves the car park so that there is at least a space available for the car to park. When the cars leave the car park, they go up to the barrier activate the sensors to validate that car plate is valid and then they get asked to enter a barcode. The app will ask the user to enter the barcode until it is of a valid format and matches the one stored along with the car. The user has 5 tries until they are rejected and must try again. When everything is valid the car leaves the car park, and we delete it from our parked cars list and increase the available spaces.

Static vs Dynamic Memory Allocation

Static memory allocation is memory that is given to a variable at compile time and lasts for the length of the program without changing [5]. Some of the benefits of allocating a piece of the memory for the entire length of the program are avoiding memory leaks which can cause the program to fail [6]. Also, since we are only allocating it once this means it is easy to handle as we do not need to free it up again. However, because of that it can be a waste of memory as the space we allocated might get used once and never again and it is kept occupied for the entirety of the program.

Dynamic memory allocation on the other hand is when we are allocating memory space while the program is running [5] [7]. We are allocating a new space in memory when it is needed and then we must deallocate it when we are no longer using it in order to avoid possible memory leaks and optimize the program. However, one drawback of dynamic memory allocation is that it is more complex to keep track of when we need to deallocate the memory [7]. In Java, we do not have this drawback due to Java's garbage collector. It is designed to check that the address is still in use and if not then reclaim the space to allocate it to other objects when they need it so the application stays optimized [8].

Data Structures

For storing the cars parked in the car park app I chose to use an array list. I went with an array list over other data structures because of the ease of access to the data since for the car park app I would only have to add and delete data to the list. Adding data is easy using an array list as it is a dynamic and I can initiate an array list then resize it when more data is added or removed [9]. Deleting from an array list is also easy as we can delete an item by either its name or by the index it is stored at. As I am using a Car class that gets stored in the array list I then look up the car plate assigned to each Car item and then delete it if I find it.

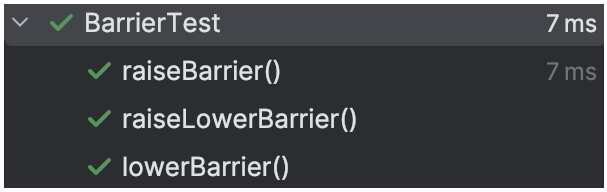
I decided to use the built-in array list data structure that comes from java util over implementing my own array list using a node class. While testing the access times I found out that implement my own data structure would make it significantly slower than using the one the comes from Java, Fig 6.

```
Elapsed time in nanoseconds for node LinkedList: 353416
Elapsed time in nanoseconds for LinkedList: 20375
```

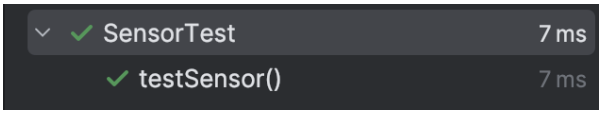
Fig. 6 Difference in time between a java util linked list and a node linked list

Unit Test Results

The unit tests cover all the classes from the app and a number of possible scenarios that could occur within the car park class. Starting with the barrier test, I test the barrier status is correctly updated, and I can raise and lower the barrier.



The sensor test tests that the car is present.



The full sign test will check that the light will be turned on when the car park is full. Another test will check that the state of the light is off when the car park is not full. The last test will cover a possible scenario happening, the car park gets full and then cars start leaving, the light should be turned on and the off.

FullSignTest	55 ms
fullSignOffTest()	47 ms
fullSignOnTest()	1 ms
fullSignOnOffTest()	7 ms

The car plate test and the bar code test will check that the entered car plate and barcode are valid or invalid.

NumberPlateReaderTest	11 ms
invalidNumberPlateTest()	11 ms
validNumberPlateTest()	
IDBarcodeReaderTest	11 ms
validBarcodeTest()	10 ms
invalidBarcodeTest()	1 ms

The unit test for the car class tests that it generates the barcode in the correct format.

CarTest	12 ms
generateBarcodeTest()	12 ms

The last of the unit tests cover the car park class. This class tests that if a car enters the car park it is then added to the list, and if the car leaves the car park and is removed from the list. I also test that the spaces available decrease when a car enters the car park and if the spaces increase when a car leaves the car park. The capacity tests the capacity being updated when a car enters and leaves the car park. Similar to the FullSign unit tests I also test the capacity being 0 and if the full sign gets turned on in this class. The final test check whether the correct vehicle gets returned when a plate number is entered.

CarParkTest	24 ms
carEntersTest()	15 ms
fullCarParkTest()	
incrementSpaces()	7 ms
capacityTest()	
decrementSpaces()	1 ms
carExitsTest()	1 ms
getVehiclesByPlateTest()	

User Guide to the App

To run the car park run Main.java. Then interact with the terminal to pass information to the app. “Incoming car” asks the user for a car that wants to park in the car park. It will only accept data in the format “AA11BBB”, anything else will be rejected. The app uses a random generator to decide whether a car parks or leaves the car park. When leaving the car park, you’ll get shown a list of all the parked cars. You select a car that wants to leave and then have to enter the matching barcode in order to pay for the car park and leave. The barcode is also shown in the same list of all the parked cars. If the barcode is entered wrong more than 5 times the car can no longer exit and will have to try again later.

References

[1] I. Lavy, Rashkovits ,Rami, and R. and Kouris, ‘Coping with abstraction in object orientation with a special focus on interface classes’, *Comput. Sci. Educ.*, vol. 19, no. 3, pp. 155–177, Sep. 2009, doi: 10.1080/08993400903255218.

[2] C. Dixon, ‘An introduction to object-oriented programming through Turbo Pascal’, *ACM SIGCSE Bull.*, vol. 23, no. 2, pp. 33–35, May 1991, doi: 10.1145/122106.122113.

[3] N. Kumar and R. Kumar, ‘Evolution of Generic Programming in OOPLs’, *ACM SIGSOFT Softw. Eng. Notes*, vol. 44, no. 1, pp. 35–43, Mar. 2019, doi: 10.1145/3310013.3310033.

[4] G. Hedin and J. L. Knudsen, ‘On the role of language constructs for framework design’, *ACM Comput. Surv.*, vol. 32, no. 1es, p. 8, Mar. 2000, doi: 10.1145/351936.351944.

[5] I. Lamprou and Z. Zhang, ‘Safe Optimized Static Memory Allocation for Parallel Deep Learning’.

[6] D. L. Heine and M. S. Lam, ‘A practical flow-sensitive and context-sensitive C and C++ memory leak detector’, in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, San Diego California USA: ACM, May 2003, pp. 168–181. doi: 10.1145/781131.781150.

[7] W. H. Lee and M. Chang, ‘A study of dynamic memory management in C++ programs’, *Comput. Lang. Syst. Struct.*, vol. 28, no. 3, pp. 237–272, Oct. 2002, doi: 10.1016/S0096-0551(02)00015-2.

[8] C. JMTauro, M. V Prabhu, and V. J Saldanha, ‘CMS and G1 Collector in Java 7 Hotspot: Overview, Comparisons and Performance Metrics’, *Int. J. Comput. Appl.*, vol. 43, no. 11, pp. 27–32, Apr. 2012, doi: 10.5120/6149-8524.

[9] ‘ArrayList (Java Platform SE 8)’. Accessed: Apr. 05, 2025. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Appendix

Video

https://youtu.be/N3UY5y0FhzE

Code

```
public class Barrier {
    private boolean isUp;

    public Barrier() {
        isUp = false;
    }

    private void raise() {
        isUp = true;
    }

    private void lower() {
        isUp = false;
    }

    public void update(boolean barrierStatus) {
        if (barrierStatus) {
            raise();
            System.out.println("Barrier raised");
        } else {
            lower();
            System.out.println("Barrier lowered");
        }
    }

    public boolean getIsUp() {
        return isUp;
    }
}
```

```
public class FullSign {
    private boolean isLightOn;

    public FullSign() {
        this.isLightOn = false;
    }

    private void switchOn() {
        isLightOn = true;
    }

    private void switchOff() {
        isLightOn = false;
    }

    public void update(boolean lightOn) {
        if (lightOn) {
            switchOn();
```

```
            System.out.println("Car park full");
        } else {
            switchOff();
        }
    }

    public boolean getIsLightOn() {
        return isLightOn;
    }
}
```

```
package org.example;

public class Sensor {
    private boolean carPresent() {
        System.out.println("Car detected");
        return true;
    }

    public boolean update(boolean isCarPresent) {
        if (isCarPresent) {
            return carPresent();
        } else {
            return false;
        }
    }
}
```

```
package org.example;

public abstract class IDReader {
    private String serialNumber;

    IDReader() {
        this.serialNumber = "";
    }

    public void read(String carPlateNumber) {
        this.serialNumber = carPlateNumber;
    }

    public boolean update(String id) {
        read(id);

        if (serialNumber.isEmpty()){
            return false;
        } else {
            return true;
        }
    }

    public String getSerialNumber() {
        return serialNumber;
    }
}
```

```java
package org.example;

public class IDBarcodeReader extends IDReader {

    public IDBarcodeReader() {
        super();
    }

    private boolean isValid(String barcode) {
        super.read(barcode);

        if(getSerialNumber().length() == 6) {
            if (getSerialNumber().matches("^[A-Z]{2}\\d{4}$")) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    @Override
    public boolean update(String id) {
        if(isValid(id)) {
            System.out.println("Barcode: " + id + " is valid");
            return true;
        } else {
            System.out.println("Barcode: " + id + " is not valid");
            return false;
        }
    }
}
```

```java
package org.example;

public class NumberPlateReader extends IDReader {

    public NumberPlateReader() {
        super();
    }

    //Check if the incoming car has a valid plate
    //car plate number -> AA11BBB
    private boolean isValid(String carPlateNumber) {
        super.read(carPlateNumber);

        if(getSerialNumber().length() == 7) {
            if (getSerialNumber().matches("^[A-Z]{2}\\d{2}[A-Z]{3}$")
|| getSerialNumber().matches("^[A-Z]{2}\\d{2}[A-Z]{2}\\d{1}$")) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
```

```
    }

    @Override
    public boolean update(String id) {
        if(isValid(id)) {
            System.out.println("Car plate: " + id + " is valid");
            return true;
        } else {
            System.out.println("Car plate: " + id + " is not valid");
            return false;
        }
    }
}
```

```
public interface Vehicle {
    String getPlateNumber();
    String generateBarcode();
    String getVehicleInfo();
    String getBarcode();
}
```

```
public class Car implements Vehicle {
    private String make;
    private String model;
    private String plate;
    private String barcode;
    private VehicleType type;

    public Car(String make, String model, String plate) {
        this.make = make;
        this.model = model;
        this.plate = plate;
        this.barcode = generateBarcode();
        this.type = VehicleType.car;
    }

    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }

    @Override
    public String getBarcode() {
        return barcode;
    }

    @Override
    public String getPlateNumber() {
        return plate;
    }

    @Override
    public String generateBarcode() {
```

```java
        Random random = new Random();

        char c1 = (char) ('a' + random.nextInt(26));
        char c2 = (char) ('a' + random.nextInt(26));

        String barcodeStart = ("" + c1 + c2).toUpperCase();
        String barcodeEnd = ("" + (random.nextInt(9000) + 1000));

        return barcodeStart + barcodeEnd;
    }

    @Override
    public String getVehicleInfo() {
        return "Plate: " + plate + " Barcode: " + barcode + "\tMake: "
+ make + " Model: " + model + " Type: " + type;
    }
}
```

```java
package org.example;

import java.util.ArrayList;
import java.util.Scanner;

public class CarPark {
    private final int capacity;
    private int freeSpaces;
    private final Barrier barrier;
    private final FullSign fullSign;
    private final Sensor entranceSensor;
    private final Sensor exitSensor;
    private final NumberPlateReader numberPlateReader;
    private final IDBarcodeReader barcodeReader;
    private final ArrayList<Vehicle> vehiclesParked;

    public CarPark(int capacity) {
        this.capacity = capacity;
        this.freeSpaces = capacity;
        this.barrier = new Barrier();
        this.fullSign = new FullSign();
        this.entranceSensor = new Sensor();
        this.exitSensor = new Sensor();
        this.numberPlateReader = new NumberPlateReader();
        this.barcodeReader = new IDBarcodeReader();
        this.vehiclesParked = new ArrayList<>();
    }

    private void incrementSpaces() {
        if (freeSpaces == 0) {
            fullSign.update(false);
        }
        this.freeSpaces++;
    }

    private void decrementSpaces() {
        this.freeSpaces--;
        if (this.freeSpaces == 0) {
            fullSign.update(true);
```

```
        }
    }

    private boolean checkBarcodeMatchesPlate(String plate) {
        Scanner scanner = new Scanner(System.in);
        boolean matchFound = false;
        boolean tooManyTries = false;
        int tries = 0;

        System.out.print("Enter the barcode to pay: ");

        while (!matchFound && !tooManyTries) {
            String barcode = scanner.nextLine();
            boolean isValidBarcode = barcodeReader.update(barcode);
            Vehicle vehicle = getVehicleByPlate(plate);

            if (isValidBarcode && vehicle.getBarcode().equals(barcode))
{
                matchFound = true;
            } else {
                System.out.print("Barcode does not match the car. Try
again: ");
            }

            if (tries == 5) {
                tooManyTries = true;
            }

            tries++;
        }

        return matchFound;
    }

    private void letCarOut(String carPlate) {
        System.out.println("Parking paid");
        barrier.update(true);
        incrementSpaces();
        System.out.println(carPlate + " has left the car park");
        vehiclesParked.remove(getVehicleByPlate(carPlate));
        barrier.update(false);
    }

    private void letCarIn(String carPlate) {
        barrier.update(true);
        System.out.println(carPlate + " has entered the car park");
        barrier.update(false);
        decrementSpaces();
        vehiclesParked.add(new Car("make", "model", carPlate));
    }

    public boolean getFullSign() {
        return fullSign.getIsLightOn();
    }

    public int spacesLeft() {
       return this.freeSpaces;
    }
```

```java
    public String getVehiclesParked() {
        String vehicleList = "";

        for (Vehicle vehicle : vehiclesParked) {
            vehicleList += vehicle.getVehicleInfo() + "\n";
        }

        return vehicleList;
    }

    public int getCapacity() {
        return capacity;
    }

    public Vehicle getVehicleByPlate(String plate) {
        for (Vehicle vehicle : vehiclesParked) {
            if (vehicle.getPlateNumber().equals(plate)) {
                return vehicle;
            }
        }

        return null;
    }

    public String getLeavingCar(){
        Scanner scanner = new Scanner(System.in);
        boolean isCarParked = false;
        String carPlate = "";

        System.out.print("\n" + getVehiclesParked());
        System.out.print("\nEnter the plate number: ");

        while (!isCarParked) {
            carPlate = scanner.nextLine();

            if (getVehicleByPlate(carPlate) != null) {
                isCarParked = true;
            } else {
                System.out.print("Please pick from the list above: ");
            }
        }

        return carPlate;
    }

    /** isParking represents whether a car is parking or leaving the
car park
     * true -> parking
     * false -> leaving
     * */
    public void update(String carPlate, boolean isParking) {
        boolean isCarParkFull = fullSign.getIsLightOn();
        boolean isValidCar = numberPlateReader.update(carPlate);
        boolean carAtEntrance = false;
        boolean carAtExit = false;

        if (isValidCar){
```

```
            if (isParking) {
                carAtEntrance = entranceSensor.update(isParking);
            } else {
                carAtExit = exitSensor.update(!isParking);
            }
        }

        if (isCarParkFull && isParking) {
            System.out.println("Car park is full at the moment. Try
later!");
        } else {
            if (carAtEntrance) {
                letCarIn(carPlate);
            }
            if (carAtExit) {
                boolean isValidBarcode =
checkBarcodeMatchesPlate(carPlate);

                if (isValidBarcode) {
                    letCarOut(carPlate);
                } else {
                    System.out.println("Too many tries. Try again
later.");
                }
            }
        }

        System.out.println(spacesLeft() + " spaces left\n");
    }
}
```