**PARKING LOT REQUIREMENTS AND ARCHITECTURAL DOCUMENT**

**How to write SRS document:** [**How to Write an SRS (Software Requirements Specification Document) (perforce.com)**](https://www.perforce.com/blog/alm/how-write-software-requirements-specification-srs-document)

**Purpose:**

* Management of a fleet of parking lots (for several customers, such as Q-park, throughout different locations in The Netherlands)

**Ideas:**

* Each parking can have a different layout (total number of parking spaces, number of floors and distribution of parking spaces (motorcycle, car, van (2 times a car), etc.)
* A camera system at the entrance and at the exit will be able to identify which vehicle type and the license plate in every parking lot. Once paid (paid confirmation has been received), the customer will have 15 minutes to leave the parking lot by default (15 minutes that will be rounded up from the next minute. Example: 15:45:21 to 16:01:00). Once these 15 minutes are expired, the customer will have to go back to the machine to extend another 15 minutes (free of charge). This 15 minutes can be adjusted for the client.
* Before entering to the parking lot, a sign will show if parking spots are still free. In case there are no free spaces, the sign will indicate “FULL”, but it will still be possible to access the parking lot once a car has left. Once inside, every floor will have its own sign indicating the free spots of that specific floor. A sensor will be placed in the ceiling of every parking spot to provide such info. Green for free, red for occupied (**maybe this is too difficult??**)
* The data for each parking lot will be transferred daily to the client. The provided data will be how many vehicles, which and how much time have stayed each day and the total amount paid by each in the parking lot (or for whatever purpose the client whishes). In terms of the parking, it will be possible to see how many vehicles were in the parking lot for a specific time (HH:MM:SS) or for a specific time frame (HH:MM:SS – HH:MM:SS).
* Apart from data, the client will be able to visualize in real time the status of every parking lot that wishes. It can be possible to visualize all the ones belonging to the customer at the same time and each of them can be seen by floor or fully.
* Fines can be defined by the client. In case a car stays more than a certain amount of time in the parking and no payment has been made, a fine will start from that point until the client defines so. The fees will be defined by the clients (**this data will be retrieved from a submodules folder in git, since it’s an external dependency. This repo will have to be maintained by the development team**)
* Define the folders architecture of the project (**different options**)
* Define the cycle life of the project (**important milestones, maintenance mode etc.**)
* Parking spots for residents are an option (**Limited amount should apply? To be defined**).
* Reservation for parking spots (online. Through app??) ([Online pre-booking (q-park.nl)](https://www.q-park.nl/en-gb/products/products/prebooking/)). **Make sure to understand how the process works so the implementation is correct**
* Reservation Park+Fly (**this has to be an option on the creation of the parking lot in case it makes sense and the parking is close to an airport**). If the parking has the option of Park+Fly, there is a special price per day up to a maximum amount of days.
* The price for each parking lot is stored in a file (database, maybe the cloud could be interesting. **This could be managed in the submodules of git, since it’s an external dependency**). The client can update the price whenever if wanted, but this should be discussed upfront, so that the development retrieves the latest update from the submodules repo.
* The opening times of the parking lot can vary. 24h service is also available. If a vehicle is still inside once the parking lot is closed it will count as parking time, unless the vehicle has a subscription.
* There are several options to manage the interaction of the customer with the payment process. Explained below.
  + The customer receives a ticket once entering the parking lot and in order to pay the ticket shall be introduced in one of the payment machines. Then the customer should introduce it in the toll just before leaving (full ticket option)
  + The customer receives a ticket once entering the parking lot and in order to pay the ticket shall be introduced in one of the payment machines. The customer does not need the ticket to leave the parking lot (hybrid option)
  + The customer does not need a ticket in order to enter or leave the parking lot and will be able to pay entering the number of the license plate in any of the payment machines (full digital)
* It will be possible to add/modify existing features of the parking lot. This will take (at least) two weeks to be implemented and tested before deploying it to the client. Deploying time can vary depending on the complexity of the add/modification.

The “happy flow” of the parking will be the following one (**UML can be defined**):

* Car entering
* Car inside and not parked yet
* Car inside and parked
* Car inside, parked and not paid yet
* Car inside, parked and paid
* Car inside, not parked anymore and not paid yet
* Car inside, not parked anymore and paid
* Car has left

In resume, a parking lot will have:

* Defined number of floors, parking spots for motorbikes, cars and vans.
* Defined location (city and postcode)
* Access/leaving and payment method can be full ticket, hybrid or full digital
* Opening and closing time
* Subscriptions available
* Leaving time after paying
* Fees for not paying after a defined time has elapsed and the vehicle is still parked
* Price list based on day and vehicle type

**Ideas to split the functionalities into several classes, so that the code adheres to SRP (names of the classes are not final):**

* **CreateAndDeleteVehicle:** Responsible for the creation and of a new vehicle type once it enters the parking lot and the deletion of it once it leaves. Before that, it should be saved on the file (or somewhere in between, like an array? And then every 1h for example, it’s saved into the file).Use a constructor for creation and destructor for deleting vehicles (using delete)
* **ParkingPayment:** Responsible to manage the payment for parking of the vehicles. This class might be split into more sub-classes such as online payment, machine payment, etc.
* **ParkingPrices:** Responsible to update the prices of the parking lots (prices depend on location and vehicle type)
* **ParkingLayout:** Responsible for the initial layout (spots, floors) of the parking. It will be possible to re-arrange the spots and the layout of the floors, but not to create new floors once the parking is done (example: A floor can be assigned for residents on a later stage and not from the beginning).
* **StoreParkingData**: Store the data that might be interesting for the stakeholder (date, day of the week, vehicle plate, entering time, leaving time, vehicle type, etc.) in an Excel file and save it (use some service like the Cloud?).
* **ReadParkingData:** Responsible for reading the data from the Excel file.
* **DisplayParkingData**: Responsible for printing the state of the parking (layout, parked vehicles, free spots, occupation by floors, etc.). A GUI could be a nice add-on for the future.
* ResidentManagement: Responsible for adding/deleting a resident from the resident list of the parking lot. First it should be checked if a resident can be added/deleted.
* **ReserveParkingSpot**: License plate, entering time and leaving time should be provided, plus the payment in advance.
* **Vehicle**: A car, van or motorbike is a vehicle. They should inherit from the class Vehicle (think about the methods, attributes, etc.)

**Testing**

* Testing in C++ can range from simple manual tests to sophisticated automated test suites integrated into CI pipelines. For robust testing, use unit testing frameworks like Google Test, consider TDD for disciplined development, and leverage CI/CD systems to automate testing. This ensures that your code is well-tested, maintainable, and reliable.

**MainFlow (should be an infinite loop, always listening)**

* We assume that the parking lots are already created. The parking lots and their details can be stored in vectors, since they are dynamic arrays that can grow or shrink in size. We want to be able to track everything at the same time, so multithreading has to be a possibility (or any other feature that allows computing different parking’s at the same time). Why? Because this is a real-time system. Since tasks must be performed at regular intervals or within strict time constraints, multithreading can help manage these tasks efficiently. Example: Embedded systems controlling hardware devices, where certain sensors need to be read or controlled in real time.
* A vehicle is detected (a license plate is given together with the type of vehicle: car, van or motorbike), it should be determined if the vehicle is among the resident ones (then it does not need to pay, apart from the monthly fee, but data should be stored anyway).
  + A new Vehicle object should be created and depending on the type of vehicle, it should belong to the class Car, Van or Motorbike, which all of them inherit from Vehicle (a Car is-a Vehicle). The vehicle Class should be a blueprint, since all of them are able to do the same, it is not expected that the vehicles can perform different actions (a car should be able to do the same as a van and a motorbike in the parking lot)
  + Depending on the vehicle type a fee is applied per hour from the entering time
* Depending on the payment methods available from the parking itself, a ticket should be issued at the entrance in order to pay before exiting
* The vehicle gets into the parking lot and starts looking for a spot. A spot will be assigned to the vehicle once the sensor of the individual parking spot turns from green to red (**or is this too difficult to implement and does not bring any value??**).

**Technical implementation:**

The code will be developed and tested in Python, following an OOD approach (**give reasons of why this is the best approach or a good approach at least**). The idea is to develop in Python and later convert this code into C++ code for production, where a better performance is expected. Think about what should be more beneficial, either use inheritance (a monstera ***is a*** plant) or composition (a monstera ***has*** leaves). A hybrid approach may be an option as well, depending on the needs (**Another approach is to follow a TDD as well. Analyze the different options. It’s compatible with what has been mentioned above**).

Cybersecurity is a topic that should be taken into consideration.

It is desirable to follow SOLID principles when an OOP approach is implemented. SOLID is an acronym that groups five core principles that apply to object-oriented design. These principles are the following:

* **S**ingle-responsibility principle (SRP)
* **O**pen-closed principle (OCP)
* **L**iskov substitution principle (LSP)
* **I**nterface segregation principle (ISP)
* **D**ependency inversion principle (DIP)

If we follow the ***Single Responsibility Principle***, we can’t assign much logic to a unique class. For instance, we can’t define a class “ParkingLot” that handles the layout, the opening and closing time, location, etc. This makes it harder for the code to be maintained, understood and tested.

A screen shot of a computer program

Description automatically generated

The previous class “Order” is not following the ***Single Responsibility Principle***, since it is both responsible for managing the order and the payment. This results in our code being highly coupled and makes it harder to understand, maintain and test. It can be refactored so that it adheres to the Single Responsibility Principle.

A screen shot of a computer program

Description automatically generated

By dividing our concerns, we are able to add new methods of payment with ease without having to modify the Order class. Note: The code still violates the SRP because the order is responsible for both the prices and the quantities.

The architecture should follow the ***Open-Closed principle*** as well. Classes should be open for extension and closed for modification. We can create an abstract class for that. If we have a base class and other classes can inherit from it, then we are following the Open-Closed principle.

Imagine that we wish to add a new payment method, we would have to make modifications to the PaymentProcessor class. This violates the Open-Closed Principle, which states that software entities should be open for extension but closed for modification. Let’s rework the code to adhere to OCP:

A screen shot of a computer code

Description automatically generated

Now the code adheres to the OCP, since it’s closed for modification and open for extension because we can add new payment methods without modifying the PaymentProcessor class. This is possible by the use of abstract classes (and methods).

Abstract classes:

It is recommended to use abstract classes whenever possible, since this would bring a scenario where the Open-Closed principle for OOP is applied. For instance, all the parking’s will have some common characteristics (as we saw before): parking spots, signs, at least a ground floor, payment, opening and closing time, etc.

However, among all this similarities there will be differences as well. For instance one parking can open at 9am and another one may be open 24h. And here is where we can get the benefit of using abstract classes to fulfill the Open-Closed principle.

What is going to differ between the different parking lots?

1. Payment method (We can create a Payment class and subclasses for each payment method via abstract methods from the Payment class, which can be used as a blueprint. The subclasses should inherit from the base class)
2. Opening/closing times
3. Layout (what if a client decides that all the parking spots are going to be for cars? And none for motorbikes)
4. Subscriptions (some clients may chose to apply it and others not). Since subscriptions might differ, it makes sense to follow the same approach as for the payment method.

As an example, in our code we might have initially a class PaymentProcessor, which manages the payment process.

A screen shot of a computer code

Description automatically generated

But then we acknowledge that the payment process/method may differ based on the choices that the clients have made. To solve this situation we can make use of abstract classes, which can be considered as a blueprint for other classes. We use an abstract class while we designing large functional units or when we want to provide a common interface for different implementations of a component.

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes (ABC) and that module name is ABC. ABC works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod. Following an example with the payment processor:

A screen shot of a computer program

Description automatically generated

The payment processor interacts with the object order (from the class Order). Independently of what we have ordered, we’ll to pay it. The payment method can differ for transactions.

In the case of the parking lot, this can be applied to the payment process as well, since there are three options: full ticket, hybrid or full digital. Therefore, in can be a good idea to define a base case for Payment and subsequently classes that inherit from the basic Payment. The abstract method can be defined as “pay”.

Another principle to follow is the ***Liskov Substitution Principle*** (LSP). The principle states that “subtypes must be substitutable for their base types”. In practice, this principle is about making your subclasses behave like their base classes without breaking anyone’s expectations when they call the same methods.

In our case, Paypal uses email addresses for verification, whereas credit and debit cards use security codes. This means we are abusing the Liskov Substitution Principle because we are using a subclass in a way that is not compatible with its parent class.

Notice that the parent class is defining the need to use both the attributes “order” and “security code” in order to process a payment:

A screenshot of a computer program

Description automatically generated

A computer screen shot of a code

Description automatically generated

However, if the method pay of the parent class PaymentProcessor is adjusted and we define a constructor for each subclass so that the Paypal class can be used with the email attribute instead of the security code, then the Liskov Substitution Principle is fulfilled.

The principle is that if we substitute some code that works under the PaymentProcessor class, it should be possible to substitute that class with any of its subclasses, such as DebitPaymentProcessor, CreditPaymentProcessor or PaypalPaymentProcessor.

PaymentProcessor becomes the type that can be substituted through polymorphism with either DebitPaymentProcessor, CreditPaymentProcessor or PaypalPaymentProcessor. Notice that they have distinct set of attributes and different initializer methods. The only thing that hey have in common is the ability to process the payment.

With this implementation in place, we can make use of the PaymentProcessor type interchangeably with its DebitPaymentProcessor, CreditPaymentProcessor and PaypalPaymentProcessor subtypes when we only care about their common behavior (process the payment). This is the essence of the Liskov substitution principle.

A screen shot of a computer code

Description automatically generated

A computer screen with text

Description automatically generated

The ***Interface Segregation Principle*** states that “Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies”.

In this case, *clients* are classes and subclasses, and *interfaces* consist of methods and attributes. In other words, if a class doesn’t use particular methods or attributes, then those methods and attributes should be segregated into more specific classes.

Let’s see an example where we add the ability to send the user an SMS to authenticate their payment.

A screen shot of a computer code

Description automatically generated

This code violates the ISP because the CreditPaymentProcessor class is forced to implement the auth\_sms method, even though it does not use it. This not only means we end up writing more code, but it could potentially cause bugs if we forget to implement the method.

Let’s refactor this code to adhere to the ISP.

A computer screen shot of code

Description automatically generated

A class SmsPaymentProcessor (which inherits from PaymentProcessor) with an abstract method auth\_sms has been created, so that the DebitPaymentProcessor inherits from it. In this way the CreditPaymentProcessor is not obliged to implement a method that does not need.

We could make this code even better by separating the authorization logic from the payment processor.



The main idea behind the ***Dependency Inversion Principle*** is that high-level modules should not depend on low-level modules, but both should depend on abstractions. This means that you should not have to change other sections of your code when you change the implementation of a class.

In practice, this means that our payment processor shouldn’t be concerned with how its payment is validated, whether that be by SMS, a robot check, or an email. Our previous code can be refactored so that it adheres to the ***Dependency Inversion Principle***.

A screenshot of a computer program

Description automatically generated

An abstract class Authorizer has been created so that any possible authorizer inherits directly from it, as it is for the case of the classes SMSAuthorizer and NotARobotAuthorizer.

A screen shot of a computer code

Description automatically generated

Before the class SMSAuthorizer, which did not inherit from any abstract method, was used by the DebitPaymentProcessor for the payment authentication via the is\_authenticated method. Now the SMSAuthorizer class inherits from the class Authorizer, which has an abstract method is\_authenticated. And the Authorizer can be any of the possible ones, not necessarily a SMS one.

This means we can change the implementation of the Authorizer class we use without having to change the DebitPaymentProcessor class.

Note that the dependency inversion principle can be applied in order to make the classes depend on abstractions rather than on concrete implementations (as it was for the DebitPaymentProcessor).