TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

# UML Class Diagrams: A Comprehensive Study of Aggregation, Composition, Dependencies, Interfaces, and Inheritance

Systementwurf und Systemdokumentation mit UML und SysML


**Vorgelegt von:** Ronny Pollak
**Matrikelnummer:** 123 4567
**Studiengang:** Master Informatik
**Dozent:** Prof. Dr.-Ing. Dipl.-Inf. Axel Hein

**Abgabedatum:** 13.01.2023

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

## 1.1 Motivation

This comprehensive study focuses on the various types of more complex relationships that can be represented in UML class diagrams. These relationships include *aggregation*, *composition*, *dependencies*, *interfaces*, *abstract* classes and *inheritance*. Each of these relationships reflects a specific type of connection between classes and serves a unique purpose in the design and implementation of object-oriented systems. The following sections define and provide examples of each of these relationships. This paper extends the work of Levin [insert paper citation] and does not delve into the basic concepts, that will be mentioned in the next section, in great detail. The focus is on furthering the research and providing examples to support the points made. The examples will build on a hypothetical delivery drone which is introduced in Levin's paper. By the end of this paper, readers should have a thorough understanding of the role of complex relationships in UML class diagrams and how to effectively use them to model object-oriented systems.

## 1.2 Introduction to UML class diagram

UML (Unified Modeling Language) class diagrams are a visual representation of the static structure of an object-oriented system. They are commonly used in software engineering to model the classes, attributes, operations, and relationships within a system, as well as the interactions between these components. A class in UML is a blueprint for an object, which is a runtime instance of that class. Classes are represented by rectangles in UML class diagrams, and typically contain three compartments:

- Class name
- *Attributes*
- *Operations*

Attributes are properties or characteristics of the class and are typically displayed in the middle compartment of the rectangle. Operations are the behaviors or actions that the class can perform and are displayed in the bottom compartment of the rectangle. In programming terms, these would be the variables and the methods or functions. Basic relationships are called *associations*. An Association indicates that one class has a reference to another class. [Rupp 12, p. 108-111] The association is shown using a solid line with an open arrow at one end, indicating the direction of the relationship. [Rupp 12, p. 142-143]

# 2 Complex UML class diagram relationships

This chapter will examine various types of complex UML class diagram relationships in detail and how they are depicted.

# 3 Aggregation and Composition

Aggregation and composition are two types of relationships that can be represented in a UML class diagram. They both are subtypes of associations and represent a relationship between two classes in which one class has a reference to another class, but they differ in the degree to which the lifetime of the referenced class is tied to the referencing class.

## 3.1 Aggregation

Aggregation is a weaker form of relationship than composition and is indicated in a UML class diagram by a solid line with a hollow diamond at one end. It represents a "part-of" relationship between two classes, in which the referencing class (the class with the diamond) is composed of one or more instances of the referenced class (the class on the other end of the line). The referencing class consists of the referenced class. However, the lifetime of the referenced class is independent of the referencing class. If the referencing class is destroyed, the referenced class may continue to exist. This means that the part of a whole can exist without the whole. In aggregation, as in associations, every combination of multiplicities is possible. [Rupp 12, p. 153]

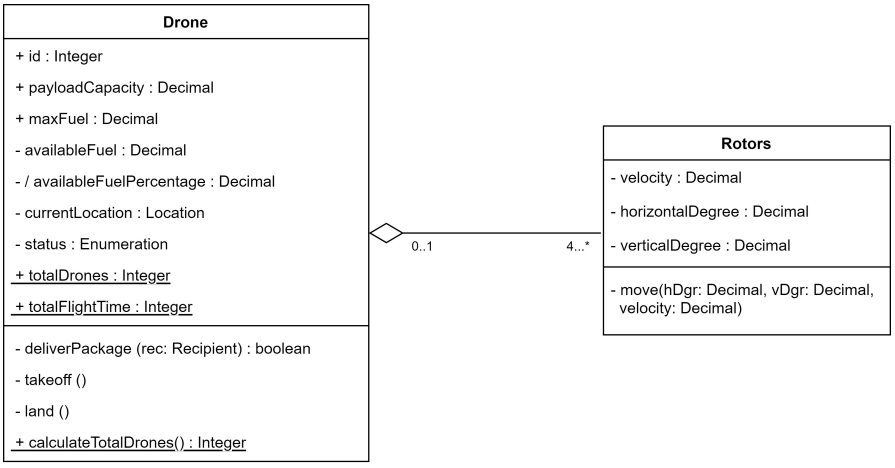


Figure 3.1: Example of an aggregation

An example of an aggregation relationship involving a `Drone` class and a `Rotor` class is demonstrated in Figure 3.1.

The `Drone` class includes a private operation, `sendDroneToRecipient()`, which involves tasks such as navigating to the recipient's location, verifying the delivery details, and safely delivering the package. The `Rotor` class, on the other hand, contains a private operation, `move()`, which is called by the `Drone` class's `sendDroneToRecipient()` operation and three private attributes, `velocity`, `horizontalDegree`, and `verticalDegree`, which control the movement of the drone in the air.

The `Drone` class serves as the referencing class in this aggregation relationship and requires at least four `Rotor` instances to function properly. However, the drone has the capacity to utilize up to eight rotors at a time, as there are eight available slots for rotors on the drone. Each `Rotor` instance can be part of one or no drones at a given time.

If the `Rotor` instances become worn down or the weather conditions change, they can be replaced with new or specialized rotors. The destruction of a `Rotor` instance does not affect the existence of the `Drone`, and conversely, the destruction of a `Drone` instance does not affect the `Rotor` instances.

## 3.2 Composition

In a UML class diagram, composition is a stronger form of relationship than aggregation and is represented by a solid line with a solid diamond at one end pointing toward the composite class. It represents a whole-part relationship between two classes, in which the whole class, known as the composite, is composed of one or more instances of the part class, known as the component.

The composite class is responsible for the lifecycle of the component class, meaning that the composite class creates and destroys the component class as needed. The component class cannot exist independently of the composite class and is tightly bound to the composite class. This is in contrast to an aggregation relationship, where the component class can exist independently of the aggregate class.

The multiplicities available for use in a composition relationship are restricted compared to those in an aggregation relationship. While a whole can have any

number of parts, a part can only contribute to one whole. The multiplicity indicated by the solid diamond in a composition relationship is fixed at 0, 0..1, or 1. In many cases, the multiplicity is left out, which implies a multiplicity of 1. A multiplicity of 0 does not make sense in a composition relationship, and a multiplicity of 0..1 means that the parts are capable of existing independently for a certain period of time. [Rupp 12, p. 153-154]

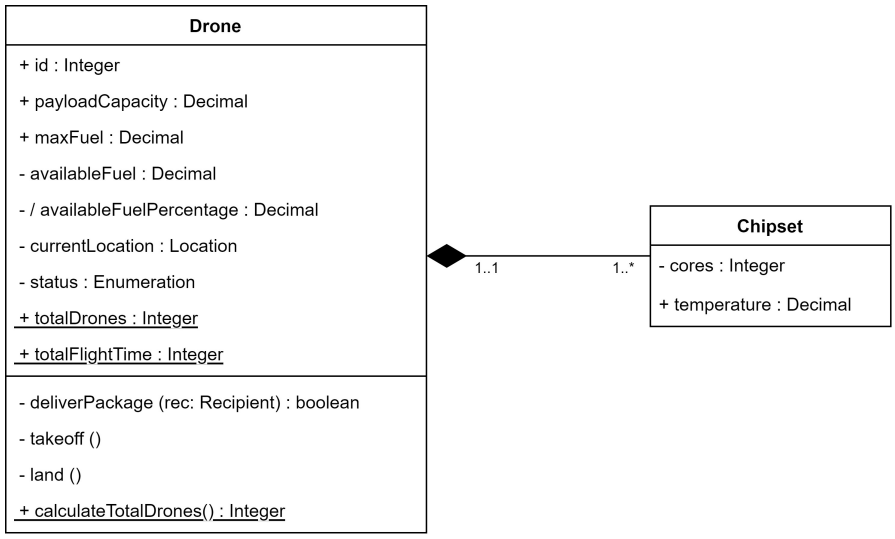


Figure 3.2: Example of a composition

In Figure 3.2, an example of a composition relationship between the `Drone` class and the `Chipset` class is displayed. The `Chipset` class contains the private attribute `cores`, which stores information about the number of cores present in the chipset, and the public attribute `temperature`, which reflects the temperature of the chipset. As a vital component of the drone, the chipset is responsible for processing data and issuing commands to other parts of the drone. Without a functioning chipset, the drone would be unable to operate.

This design also incorporates an anti-theft concept, in which each chipset is uniquely bound to a specific drone. The number of chipsets used by the drone may vary based on their efficiency and the desired level of efficiency for the drone.

# 4 Dependencies

In UML, a dependency is a relationship between two elements in which one element (the client) uses or depends on another element (the supplier) for specification or implementation. The dependency represents a semantic or structural relationship between the supplier and the client, indicating that the client is dependent on the supplier in some way. A dependency indicates that the connection between the two elements is at a higher level of abstraction than an association relationship. Dependencies can be represented in various UML diagrams, including class diagrams, component diagrams, deployment diagrams, and use-case diagrams. To represent a dependency in a UML diagram, a dashed line with an open arrow pointing from the client to the supplier is used. In some cases, stereotypes may be used to show the precise nature of the dependency. [Rupp 12, p. 159-160]

The following Table 4.1 shows the different types of dependencies, their stereotype, and a description.

| **Type of dependency** | **Stereotype** | **Description** |
|---|---|---|
| Abstraction | «abstraction», «derive», «refine», or «trace» | Relates two model elements, or sets of model elements, that represent the same concept at different levels of abstraction, or from different viewpoints |
| Realization | «realize» | Indicates that the client model element is an implementation of the supplier model element, and the supplier model element is the specification |

| | | |
|---|---|---|
| Substitution | «substitute» | Indicates that the client model element takes the place of the supplier; the client model element must conform to the contract or interface that the supplier model element establishes |
| Usage | «use», «call», «create», «instantiate», or «send» | Indicates that one model element requires another model element for its full implementation or operation |

Table 4.1: Types of dependency [IBM 21] [Rupp 12]

In the following sections, the different types of dependencies will be explained with examples.

## 4.1 Usage

In the context of modeling and software development, a usage relationship is a dependency in which one element (client) requires another element (supplier) for its full implementation or operation. The client element is considered incomplete without the supplier element. The usage relationship is represented using the stereotype `«use»` and is depicted using a dashed line with an arrow pointing from the client element to the supplier element. The client and supplier elements can be any named elements in the model. The difference between a usage relationship and a normal dependency relationship is minimal, with the main distinction being that the `«use»` stereotype may be used to specify that the dependency is specifically related to the actual use of the supplier element. [Rupp 12, p. 161]

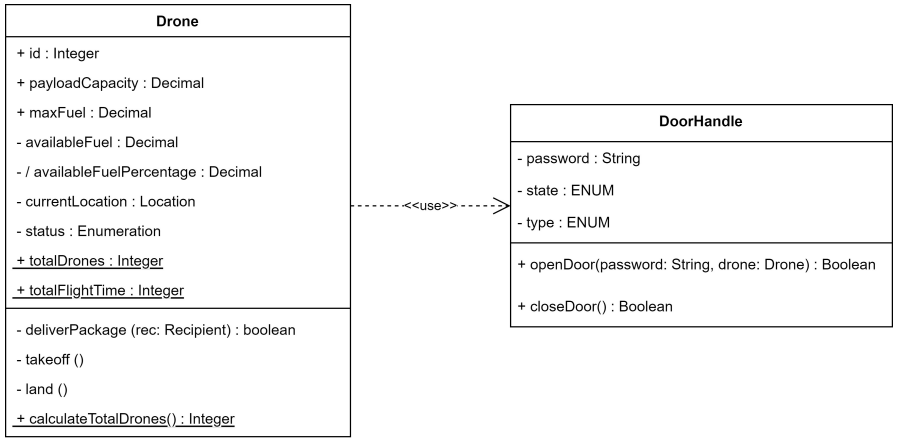


Figure 4.1: Example of usage

In Figure 4.1, an example of a usage relationship between the `Drone` class and the `DoorHandle` class is displayed. The `DoorHandle` class contains the private attribute `type`, which stores information about the type of door handle, and the public operation `open()` which allows the door to be opened with a password. This operation also utilizes the `state` enumeration to store the current state of the door handle, including `open`, `closed`, and `invalid` states. It enters the invalid state if an error occurs while opening or closing the door

The DoorHandle is built into various types of doors and is responsible for providing a mechanism for opening the door. The drone uses the DoorHandle to open doors as part of its delivery tasks and drops off the packages in the desired place. This can include houses, garages, or shacks. To open a door, the drone calls the public `open()` operation of the DoorHandle class and passes in the password for the door. If the password is correct, the door is opened and the Drone class can complete its delivery. Afterwards, the door will be closed again with the public operation `closeDoor()`.

This usage relationship between the Drone and DoorHandle classes can be represented with a directed line with an arrowhead pointing from the Drone class to the DoorHandle class and the stereotype «use». The Drone class is the client that uses the DoorHandle class, and the DoorHandle class is the supplier that provides the functionality for opening the door.

## 4.2 Abstraction

The abstraction relationship connects different levels of abstraction with each other. Stereotypes are a way to add additional meaning to an element in a UML2 class diagram. The common stereotypes that refine the basic abstraction relationship are: [IBM 21]

- `«derive»`: This stereotype indicates that the client element is calculated from the supplier element.
- `«refine»`: This stereotype indicates that the client element refines or improves upon the supplier element.
- `«trace»`: This stereotype can be used to link elements that have the same message content, but are each viewed in a different context or from a different perspective

In the following the two more common stereotypes `«derive»` and `«refine»`: will be explained with examples.

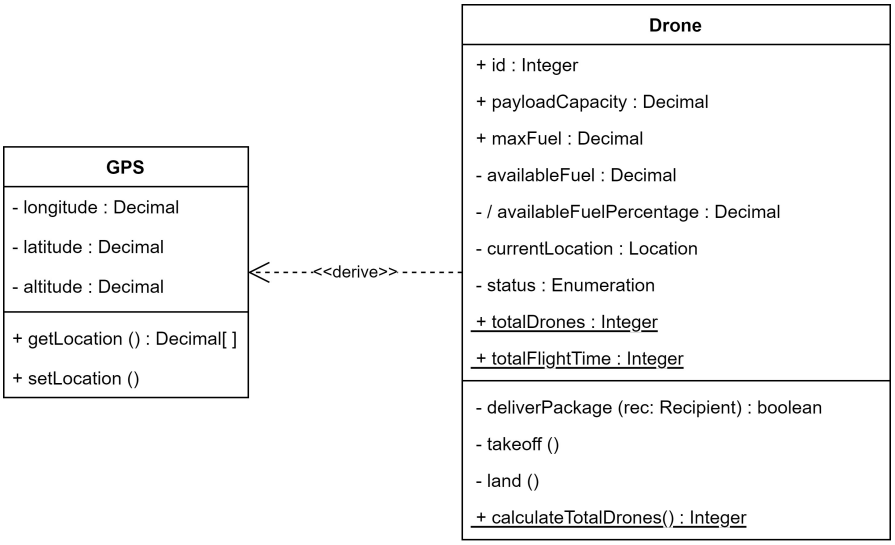


Figure 4.2: Example of a derive abstraction

In Figure 4.2, an example of a dependency relationship between the `Drone` class and the `GPS` class is displayed. The `GPS` class contains the private attributes `latitude`,

`longitude`, and `altitude`, which store information about the drone's current location. It also has the public operations `getLocation()` and `setLocation()`, which allow the Drone class to retrieve and update the location information provided by the GPS class. This dependency can be represented with a «derive» stereotype, as the drone calculates its location based on the information provided by the GPS.

The Drone class depends on the GPS class for accurate location data, as it is essential for the drone to be able to determine its position and navigate to the desired location. To retrieve the current location, the Drone class calls the `getLocation()` operation of the GPS class and stores the returned location in its `location` attribute. The Drone class can then use this information to perform actions such as flying to a specific location or avoiding obstacles.

This dependency between the Drone and GPS classes can be represented with a `«derive»` stereotype, as the drone calculates its location based on the information provided by the GPS. The GPS class is the supplier of this information, and the Drone class is the client that depends on it. Without the GPS, the drone would be unable to determine its location and would be unable to perform its intended functions.

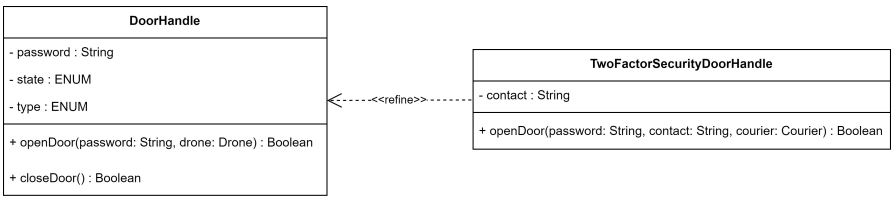


Figure 4.3: Example of a refine abstraction

In Figure 4.3, an example of a refinement relationship between the `DoorHandle` class and the `TwoFactorSecurityDoorHandle` class is displayed.

The `DoorHandle` class, which was previously used by the drone to open and close doors as part of its delivery tasks, has been refined by the `TwoFactorSecurityDoorHandle` class to add an extra layer of security. The `TwoFactorSecurityDoorHandle` class has a private attribute `contact` that stores the phone number, email address, or recipient's ID in the delivery app.

When the drone calls the new `closeDoor` operation of the `TwoFactorSecurityDoorHandle` class, the recipient will receive a notifica-

tion that they must answer, in addition to the password that the drone inputs, for the door to be opened. This added security measure ensures that only the drone with the intended package can unlock the door.

The `TwoFactorSecurityDoorHandle` class is shown as the client element in the refinement relationship, which is depicted with a dashed line with an open arrow pointing from the `TwoFactorSecurityDoorHandle` class to the `DoorHandle` class. The stereotype `«refine»` is added to the relationship to indicate that the `TwoFactorSecurityDoorHandle` class refines the `DoorHandle` class.

## 4.3 Substitution

In UML, a substitution dependency is a relationship between classifiers that allows one classifier (the client) to be used in place of another classifier (the supplier) at runtime. The client classifier must meet all requirements specified by the supplier, including implementing interfaces and having corresponding ports. The substitution dependency is depicted in a UML diagram with a dashed line and open arrow pointing from the client to the supplier, and the stereotype `«substitute»` added to the relationship. Only classifiers, such as classes or components, can be connected in a substitution dependency. Substitution dependencies can be used when inheritance is not possible or appropriate, for example, if the supplier classifier is part of a third-party library and cannot be modified. The client and supplier classifiers in a substitution dependency do not need to have similar base structures, allowing for more flexible and complex designs. [Rupp 12, p. 165]

One common use case for substitution dependencies is when a classifier is being replaced or upgraded with a new version, but the new version needs to maintain compatibility with the original classifier. For example, consider a drone delivery system that originally used a fleet of traditional drones for deliveries. As the company develops and deploys a new line of robots that are capable of performing the same tasks as the drones, they may use a substitution dependency to indicate that the robots can be used as a substitute for the drones.

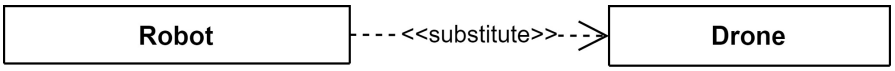


Figure 4.4: Example of a substitution

In Figure 4.4, an example of a substitution relationship between the `Drone` class and the `Robot` class is displayed.

The `Drone` class has been designated as the contract classifier, which means that it specifies a set of obligations that must be fulfilled by any substituting classifiers. In this case, the `Robot` class has been identified as a substituting classifier, which means that it can be used in place of the `Drone` class wherever instances of the `Drone` class are expected.

To fulfill the obligations specified by the `Drone` class, the `Robot` class must implement all interfaces and ports that are required by the `Drone` class. For example, the `Drone` class requires the implementation of a `deliverPackage()` operation, which must also be implemented by the `Robot` class to be used as a valid substitution.

## 4.4 Realization

A realization dependency relationship indicates that the client model element is an implementation of the supplier model element, and the supplier model element is the specification. A realization relationship can be used to model the relationship between a class and an interface, a component and a class, or a component and an interface. It represents the implementation of a contract defined by the supplier element and indicates that the client element provides the behavior specified by the supplier element. In UML2 a realization relationship is depicted as a solid line with a hollow triangle arrowhead pointing from the client element to the supplier element. Before UML2 the usual notation of dependencies, using a dashed line with an open arrow, was used as an alternative. [Rupp 12, p. 164]

An example of the realization relationship will be presented in the Interface chapter.

## 4.5 Multiple dependencies

In cases where a client has a dependency on several suppliers, a connector (represented as a small black dot) can be used to model the dependency. [Rupp 12, p.

160] In the following Figure 4.5 an example of possible multiple dependencies is displayed.

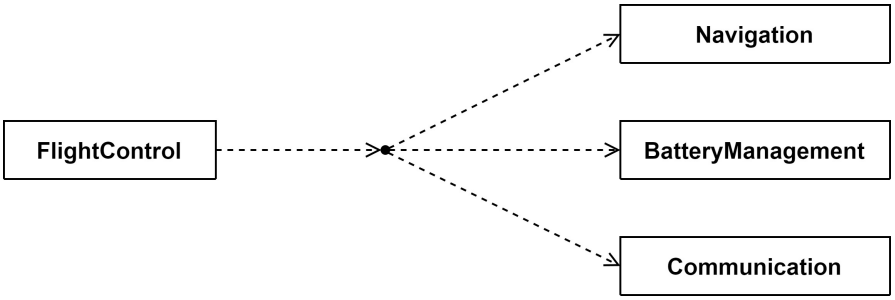


Figure 4.5: Example of a multiple dependency

For the class `FlightControl` to work properly, it is dependent on the classes `Navigation`, `BatteryManagement`, and `Communication`. Furthermore, the way the classes are dependent can be elaborated by stereotypes and added notes.

## 4.6 Interface

An interface is a type of classifier that represents a set of related behaviors that a class or component must implement to conform to the interface. An interface defines a set of abstract operations, which are public and do not have any implementation. Interfaces are represented by a classifier rectangle with the stereotype `«interface»` at the top, followed by the name of the interface. The operations and attributes of the interface are listed inside the rectangle. [Rupp 12, p. 130-131]

There are two equivalent ways of displaying an interface relationship which are shown in the following Figure 4.6:

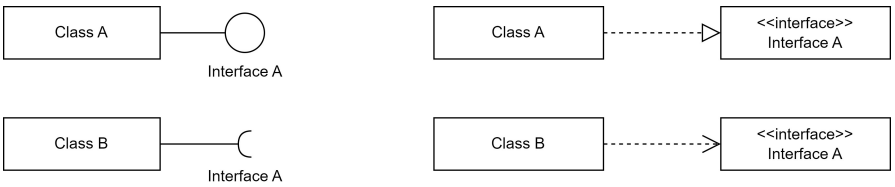


Figure 4.6: Notation of an interface relationship

The left side shows the lollipop and the right side the arrow notation. The first row displays the implementation of the interface with the class. The second row displays that the class requires an implementation of this interface. [Rupp 12, p. 129] In the following the arrow notation will be used.

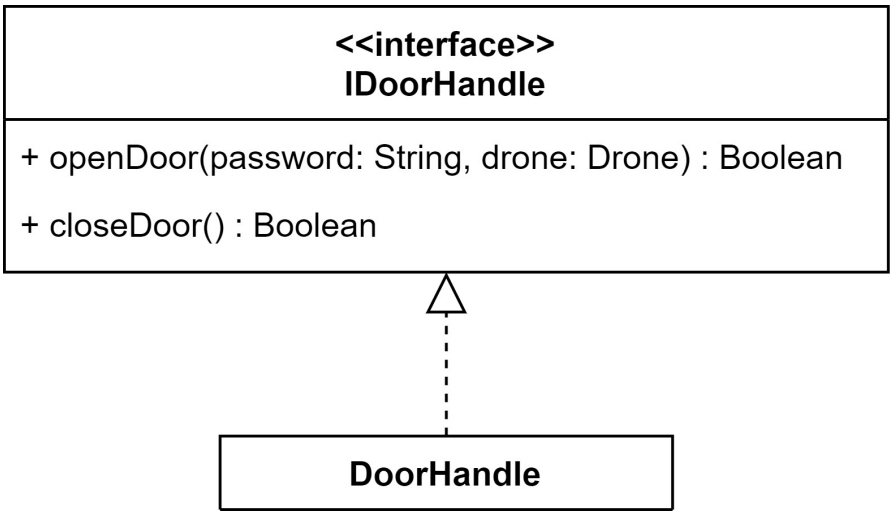


Figure 4.7: Example of a DoorHandle interface

An example of an interface is shown in Figure 4.7, where the `IDoorHandle` interface has a realization relationship with the `DoorHandle` class. The `DoorHandle` class implements the `openDoor()` and `closeDoor()` operations defined in the interface, allowing for flexibility in the implementation. For instance, a different version of the `DoorHandle` class that also implements these operations could be used in place of the original. Interfaces in UML allow for easy substitution of different class implementations while maintaining the same set of required operations.

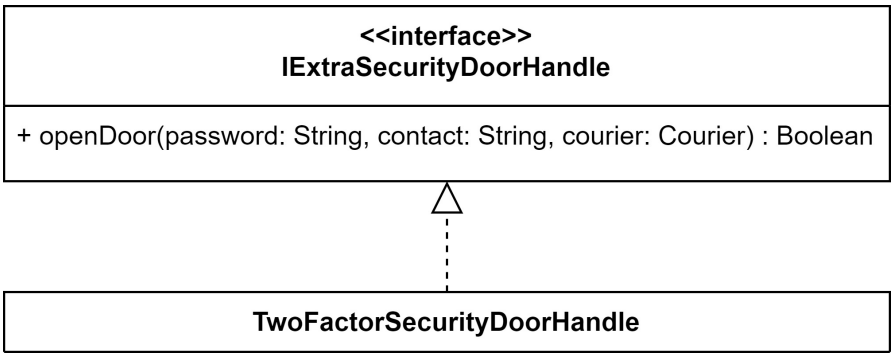


Figure 4.8: Example of an ExtraSecurityDoorHandle interface

In Figure 4.8, another example of an interface relationship is shown. The `IExtraSecurityDoorHandle` interface allows for multiple different implementations of secure DoorHandles to be easily swapped in and out. The `TwoFactorSecurityDoorHandle` class implements the `openDoor()` operation of the `IExtraSecurityDoorHandle` interface.

# 5 Inheritance

In UML class diagrams, inheritance is a relationship between two classes in which one class (the subclass or derived class) is a specialization of the other class (the superclass or base class). This means that the subclass inherits the attributes, operations and associations of the superclass, and can add additional of its own. Inheritance is displayed by a solid line with a closed, non filled, arrow pointing from the subclass to the superclass. The two ways of designing inheritance relationships are described in the following.

Generalization is a term used to describe the process of creating a superclass from one or more subclasses. This is typically done to identify common attributes and behaviors shared by the subclasses and to create a more generalized, abstract representation of these concepts. [Seid 15, p. 69-70] [IBM]

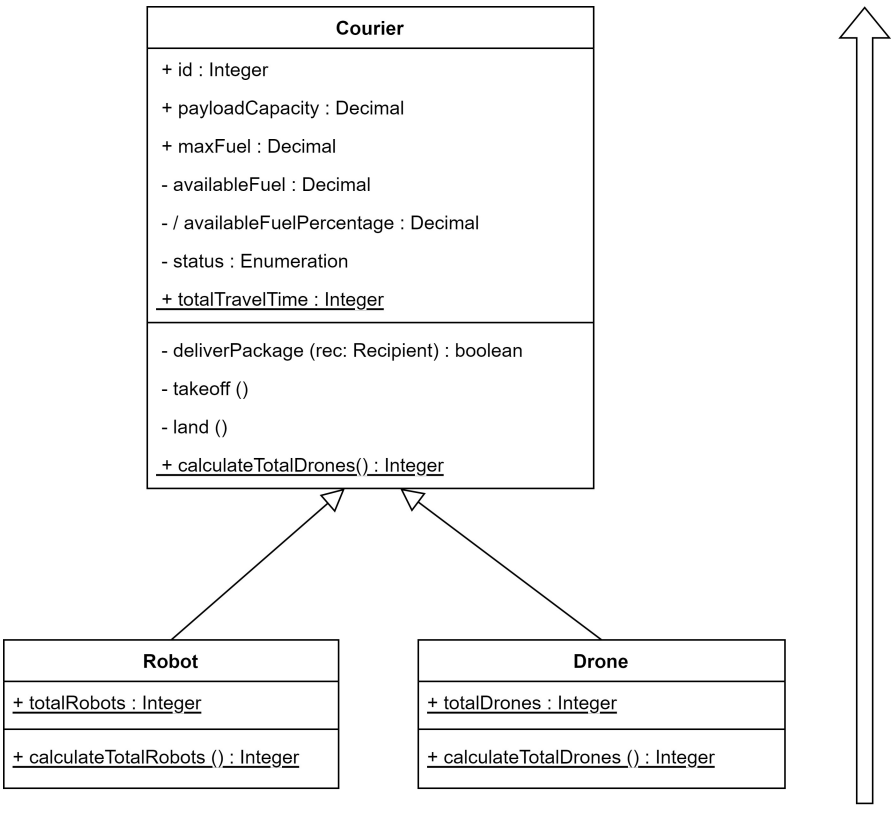


Figure 5.1: Example of generalization

In Figure 5.1, an example of inheritance and generalization is shown. The class `Robot` has been added to the system as an additional form of transportation. Since the classes `Robot` and `Drone` have many attributes and operations in common, they have been extracted into a superclass, the `Courier`. This means that both subclasses inherit the attributes, operations, and associations of the superclass. The superclass `Courier` has replaced the `Drone` class in its previous relationships that have been shown in prior examples. The arrow on the right side indicates that the superclass is formed from its two subclasses, which have been designed before the superclass.

Specialization is the process of creating a subclass from a superclass. This is typically done to add more specific attributes and behaviors to the subclass, or to specialize the subclass for a particular purpose.

In the following Figure 5.2, an example of specialization is depicted. The figure illustrates that, given the existence of a superclass, the subclasses can be derived from it, resulting in a specialization relationship. The arrow on the right side of the figure indicates that the subclasses, in this case `Drone` and `Robot`, are derived from the superclass `Courier`, which was designed prior to them. This inheritance relationship allows the subclasses to inherit the attributes, operations, and associations of the superclass.

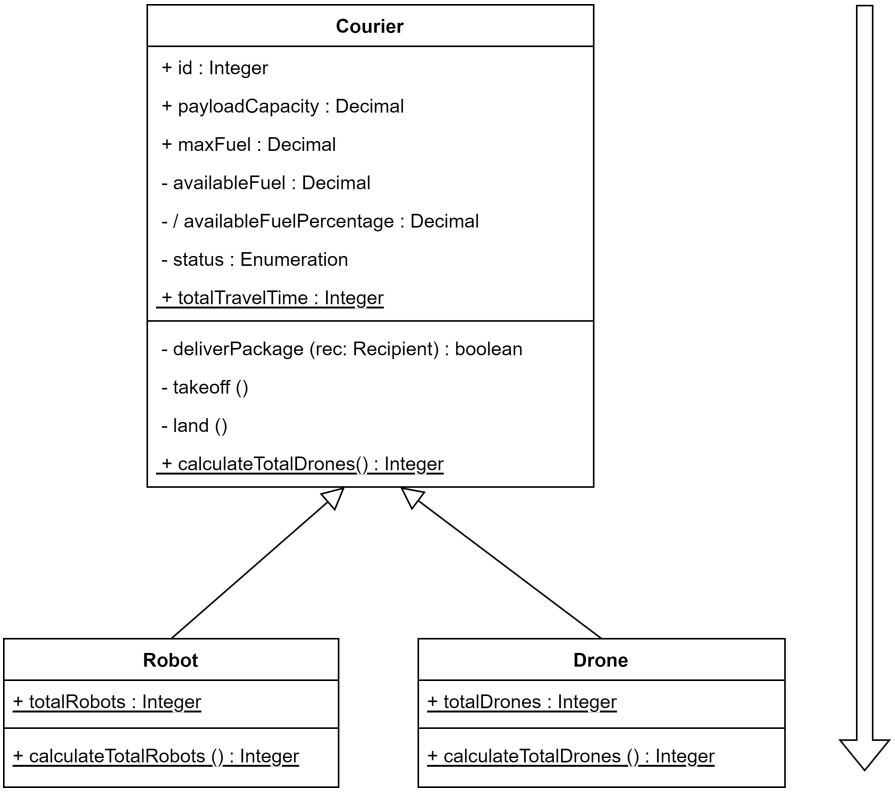


Figure 5.2: Example of specialization

UML also allows multiple inheritance, which means classes can have multiple superclasses. [Seid 15, p. 71]

# 6 Resumee and outlook

# Glossary

**library** A suite of reusable code inside of a programming language for software development. i

**shell** Terminal of a Linux/Unix system for entering commands. i

# Bibliography

[IBM] IBM. “Generalization in UML”.

[IBM 21] IBM. “Dependency relationships”. March 2021.

[Rupp 12] C. Rupp, S. Queins, and die SOPHISTen. *UML 2 glasklar*. Carl Hanser Verlag GmbH & Co. KG, München, 4., aktualisierte und erweiterte auflage Ed., 2012.

[Seid 15] M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML@ classroom*. Springer, 2015.

# Supplemental Information

Hier könnte Ihr Anhang stehen!