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. 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.

TODO Drone beispiele erwähnen

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]

1.3 Structure

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.

2.1 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.

2.1.1 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.

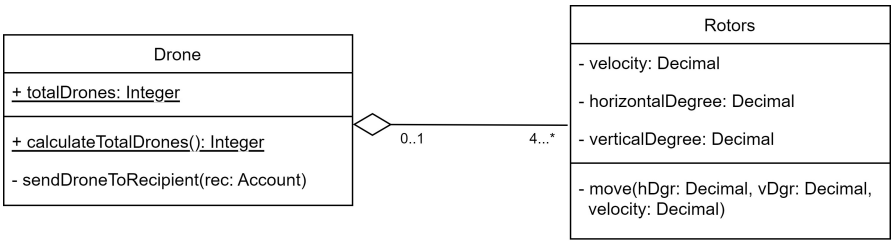

Figure 2.1: Example of an aggregation

An example of an aggregation relationship involving a `Drone` class and a `Rotor` class is demonstrated in Figure 2.1. The `Drone` class contains a static attribute, `totalDrones`, which keeps track of the total number of drones, as well as a static operation, `calculateTotalDrones()`, which calculates this value. The class also includes a private operation, `sendDroneToRecipient()`, which is responsible for sending the drone to its intended recipient. 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. [Rupp 12, p. 153]

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.

TODO: Qualifizierte Assoziation Darstellung einer ternären Assoziationn

2.1.2 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 towards 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.(e.g. beispiel). (e.g., the pineapple juice before it is mixed with the other ingredients of the cocktail). [Rupp 12, p. 153-154]

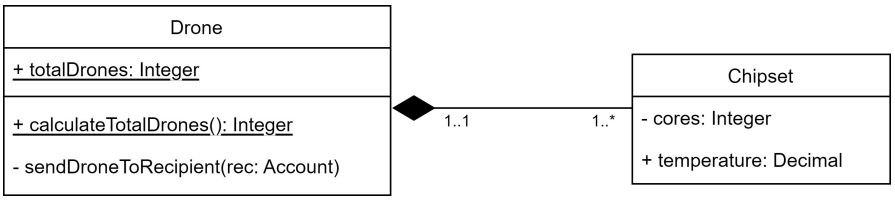

Figure 2.2: Example of a composition

In Figure 2.2, 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.

2.2 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 (Booch, Rumbaugh, und Jacobson, 1998). 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.

The following Table 2.1 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
Binding	«bind»	Connects template arguments to template parameters to create model elements from templates
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 2.1: Types of dependency [IBM 21]

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

2.2.1 Usage

In Figure **??**, 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 method `open(password: string)` which allows the door to be opened with a password. This method 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 occures 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 the houses, garages or shacks. To open a door, the drone calls the `open(password: string)` method 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.

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.

2.2.2 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 sterotypes 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 Figure **??**, 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 methods `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 a desired location. To retrieve the current location, the Drone class calls the `getLocation()` method of the GPS class and stores the returned location in its own `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.

2.2.3 Binding

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. This allows designers and developers to represent complex dependency relationships in a clear and concise manner, making it easier to understand the dependencies and interactions between different parts of a system.

2.3 Foo Foo

Hier folgt nun ein Listing, was basically Code ist. Und hier verlinke ich darauf: Listing 2.1.

```
1 def sexyFunction(x):
2     return x + 69
```

Listing 2.1: Pythonbeispiel nach Albrecht et al. [**?**]

2.3.1 Foo Foo Foo

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3 Bar

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a

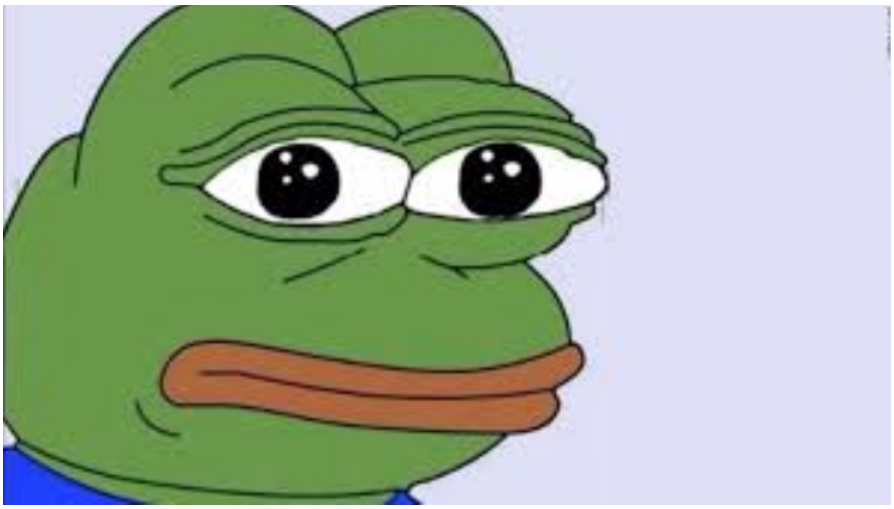

Figure 3.1: Ein äußerst wichtiger Frosch ist hier auf diesem Bild zu sehen. Diese Caption wäre zu lang für die Anzeige im Abb.verzeichnis, weswegen es diese eckigen Klammern gibt. Zitieren kannst du hier auch [**?**]

meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3.1 Bar Bar

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.
Es gibt eine wichtige Tabelle, nämlich die nun verlinkte Table 3.1.

3.1.1 Bar Bar Bar

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a

	Frosch	Frosch
Strukturierter Frosch	Datenabfrage	Frosch Mining
Unstrukturierter Frosch	Inhaltsabfrage	Text Frosch

Table 3.1: Die Tabellencaption: Für Tabellen empfiehlt sich der https://www.tablesgenerator.com/

meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

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 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.

Supplemental Information

Hier könnte Ihr Anhang stehen!