













- 03 - Programming Foundations: Object-Oriented Des

 Certificate	Abschlusszertifikat_Programming Foundations ObjectOriented Design.pdf
 Completed Sections	9
 Course Links	https://www.linkedin.com/learning/programming-foundations-object-oriented-design-3/?resume=false
 Goal Sections	9
 Quick note	
 Start and Finish Date	
 Thumbnail	
 Time	2h 40m 37s
 Understanding %	40%-70%
 نسبة الإنتهاء	 Done

foundemntals

- What is the difference between **object-oriented programming** and **procedural programming**?
 - Procedural programming specifies a sequence of tasks, but object-oriented programming describes the properties of tools or items.

Correct

In object-oriented programming, there are some elements of procedural programming in the description of objects.

- How does dynamic polymorphism differ from static polymorphism?
 - It uses overriding instead of overloading.

Correct

Dynamic polymorphism creates a unique instance.

- What is overriding a method?
 - Creating a unique version of an inherited method.
-

- How are analysis and design different?
 - Analysis describes a problem; design describes a solution.
-

In addition to attributes and methods, what does a UML class diagram contain?

the class name

How do object behaviors and attributes differ?

- Attributes describe a state, but behaviors describe actions.
-

Shonzu has gathered the requirements for a new solution, described the application he is going to build, and identified the main objects in the solution. What should he do next?

Describe object interactions. , • This will be essential for understanding behaviors.

The purpose of **encapsulaiton** is to protect an object from unwanted changes

The Class contains

Attributes , behaviors and a name

- And we use abstraction when we define a class , to have something general like ,Focusing on the idea of a person instead of an individual

The **benefit** of using a programming language that has a **large library** , that many classes are already defined and can be used without have to re-deifen them.

If an attribute is added to a superclass, each subclass object automatically receives the additoinal attribute.

polymorphism

Dynamic Polmorphism

Uses the same interface for methods on different types of objects

Unterschiedeliche Kaffe Maschine machen Kaffe , weil sie Kaffe als Input bekommen haben .

Static Polymorphism

- Method Overloading
 - Implements mulple mthods with the same name , but different parameters
 - Selbe Maschine aber mit unterschiedliche Input , kaffe oder tee

Requirements

▼ Defining (Requirements)

- Non-functional requiremnets
 - How should it do it?
 - Legal
 - performance
 - Support
 - security
 - The appilication should be
 - availabel

- usable
- compauitble
- Functional Requiremnets
 - The system / application must do...

▼ FURPS+ requirements (Wiederholen) @April 17, 2022 → April 20, 2022

- Functionnality
- Usability
- Reliability
- Performace
- Supportabilty

▼ Challenge

- the system must
 - change the music if there are more than to in the row to let one hear his music

Use Cases

short use cases help identify problems but do not create the confusing work of full use cases. Casual use cases can be very helpful, but the use cases should be more fully developed for larger projects.

- Title: What is the goal?
 - that the austrounts listen muisk
 - all of them get the muisk they want
- primary Actor : Who desires it?
 - Astronouts
- Success Scenario: How is it accomplished?
 1. It can be written in steps one
 2. and than two

- a. the user 1 wish 3 songs firstly
- b. the user 2 wish just one song secondly
- c. The song of the second user got to the first place

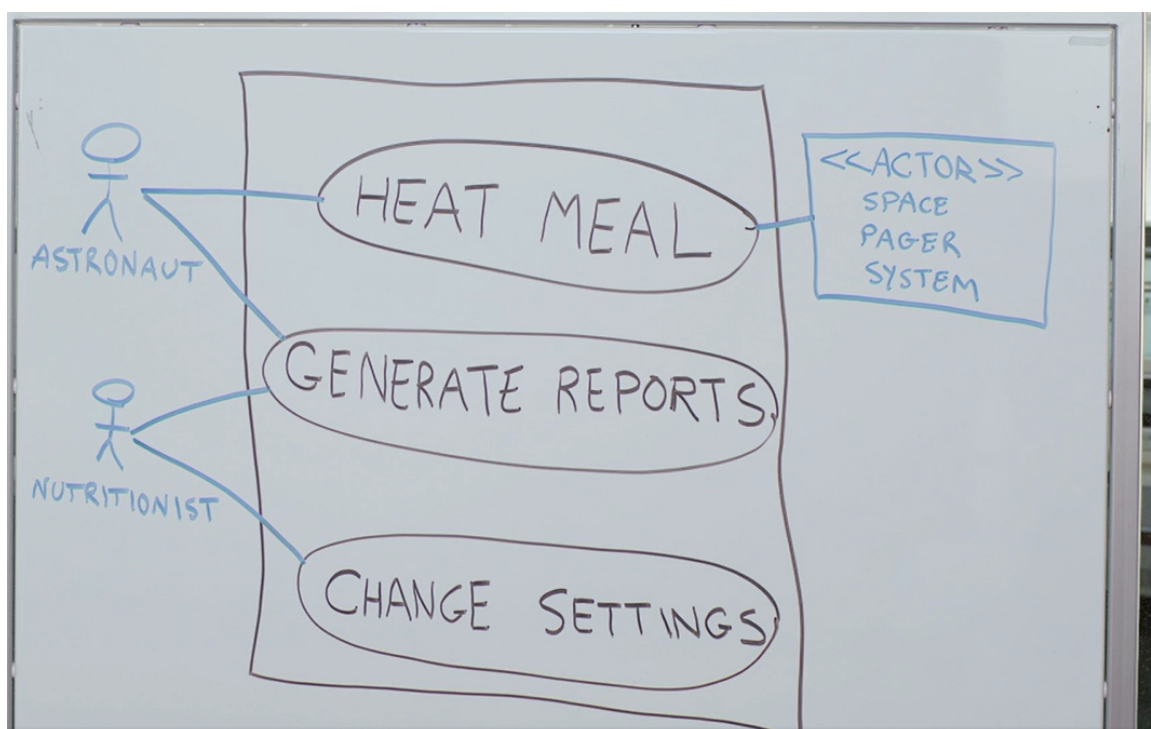
▼ identifying the actors

▼ identifying the scenarios

- User- Focused Goal
- It should be with own words
- Use case prompts
 - Who performs system administration tasks?
 - who manages users and security?
 - what happens if the system fails?
 - Is anyone looking at performance metrics or logs?

▼ Diagramming use cases

the function of a use case diagram is to connects actors to use cases



▼ User stories

- we have to follow 3 steps
 - As a (type of user)
 - Muisk listnner
 - I want (goal)
 - I want to listen to the muisk which i want
 - so that (reason)
 - nobody can play 6 muisk in the row and what other people finde boring
 - example:
 - As an astronaut , I want to heat up my food, so that I can eat a warm meal.

We have to leave techniqel stories

• Write two use cases

- Title
- Primary actor
- Scenario (paragraph or list)

• Write two user stories

- "As a ... I want ... so that ..."

Domain Modeling

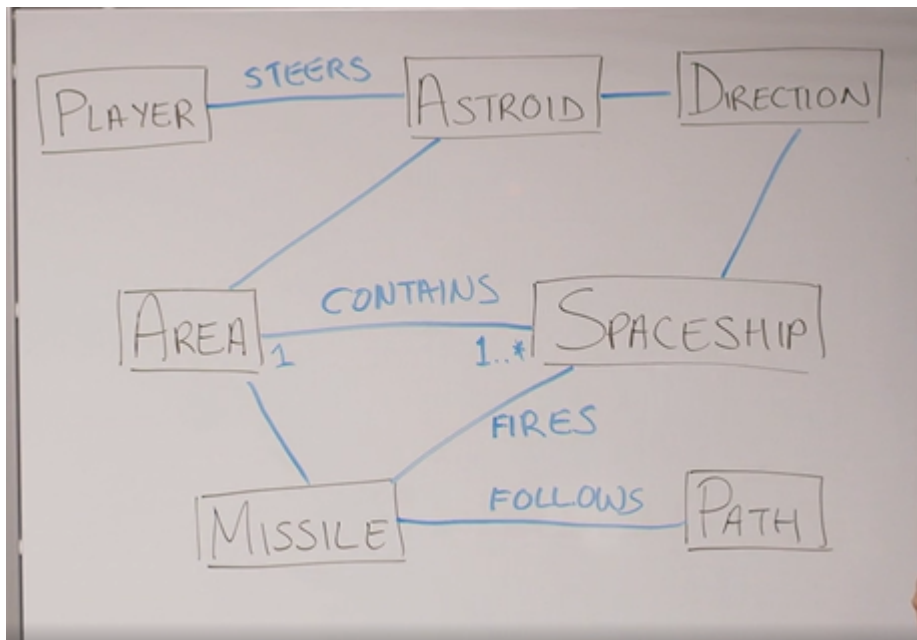
▼ Identifying the objects

- Conceptual Model
 - represents important objects and the relationships between them
 - After writing the scenario
 - Mark the names in the scenario and then look if something repeats or can not be objects

"Dodge" Use Case Scenario: System spawns enemy spaceship in play area. Spaceship flies towards player asteroid and fires missile at it. Player steers asteroid in direction to avoid missile path. Missile flies past player asteroid and disappear offscreen.

System	It
Spaceship	Player
Area	Direction
Asteroid	Path
Missile	Offscreen

relationships



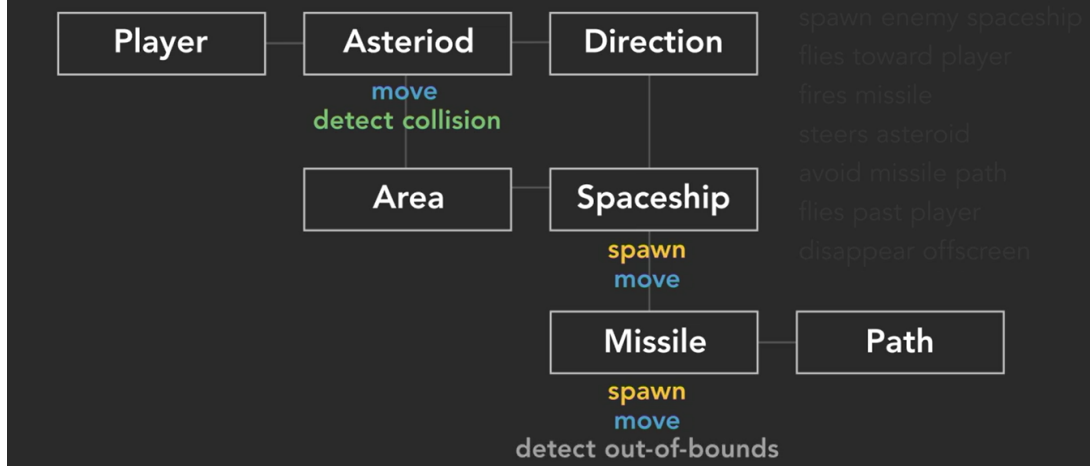
▼ Identifying class responsibilities

- we have to search the verbs to detect the responsibility
 -

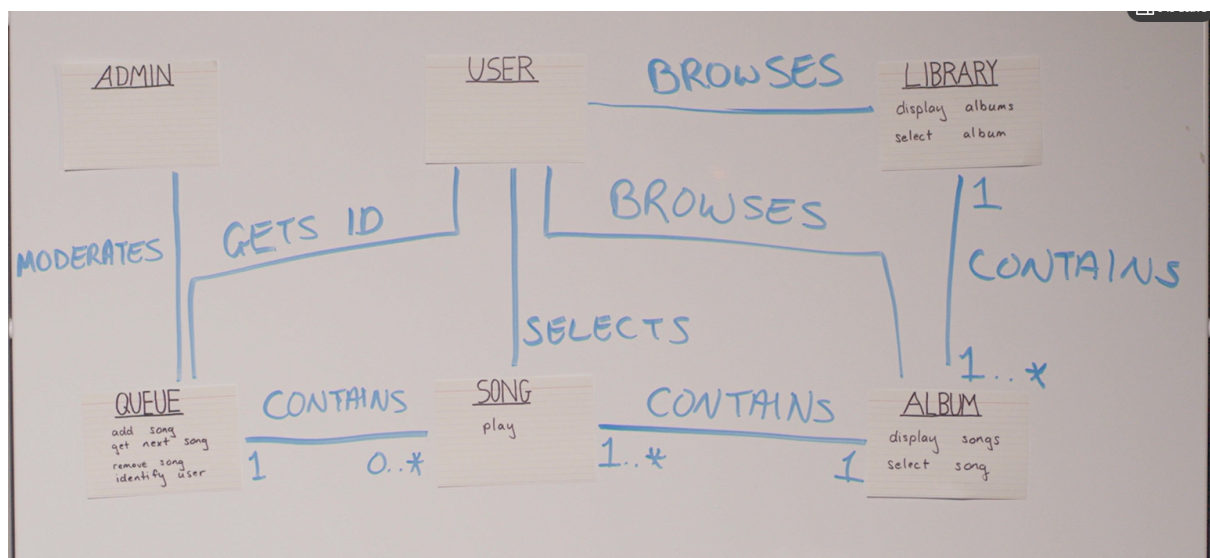
"Dodge" Use Case Scenario: System spawns enemy spaceship in play area. Spaceship flies towards player asteroid and fires missile at it. Player steers asteroid in direction to avoid missile path. Missile flies past player asteroid and disappears offscreen.

An object should be responsible for itself

Conceptual Object Model



Challenge

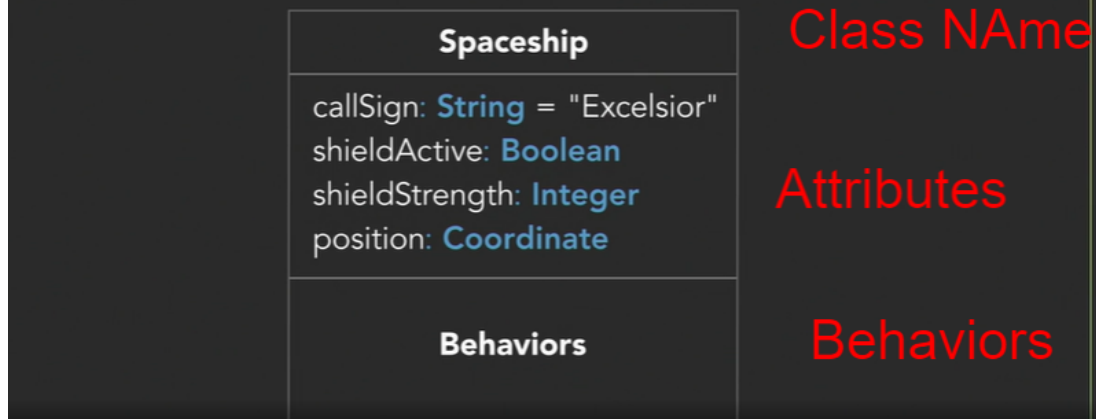


To identify candidates for objects we have to listing all for the nouns in the user stories , because Objects are nouns : they are things

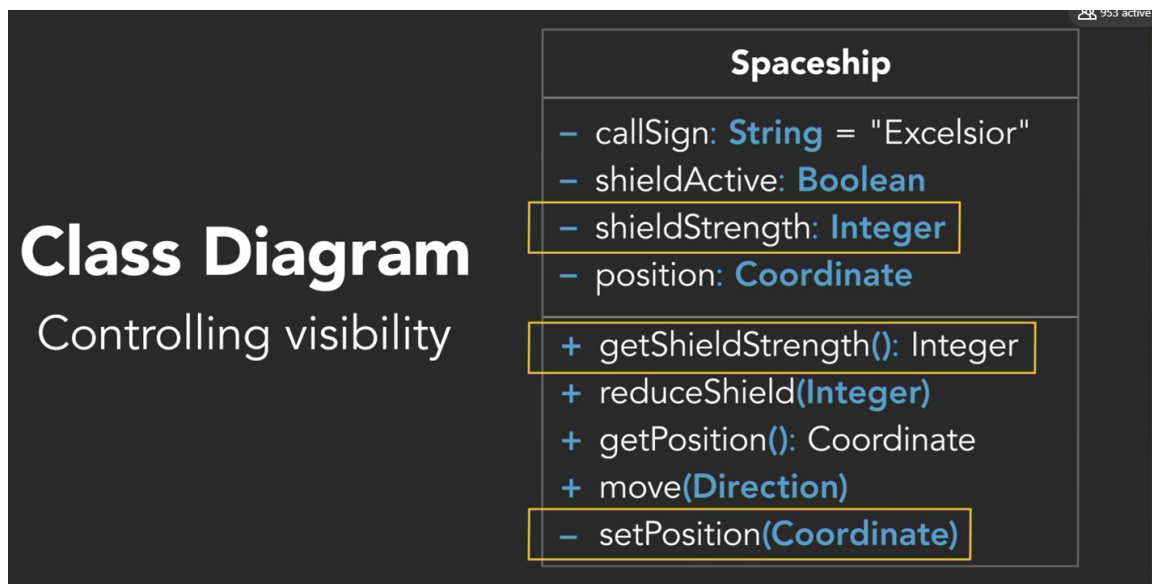
Class Diagramm

▼ Creating class diagrams: Attributes

Class Diagram



▼ Creating class diagrams: Behaviors



▼ Converting class diagrams into code

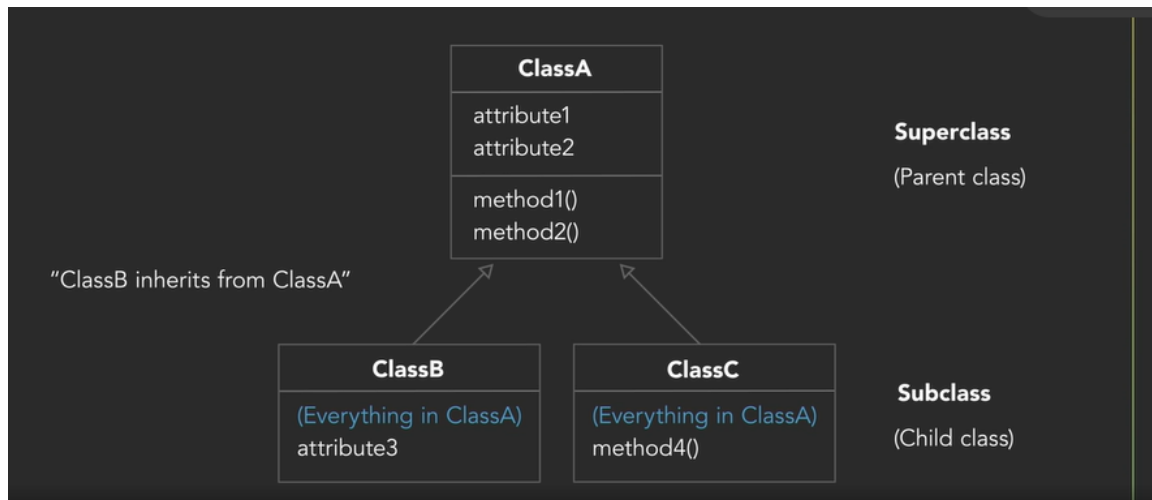
Python

```
class Spaceship():  
  
    def __init__(self):  
  
        # instance variables  
        self.callSign = ''  
        self._shieldStrength = 100  
  
        # methods  
        def fireMissile(self):  
            return "Pew!"  
  
        def reduceShield(self, amount):  
            self.shieldStrength -= amount
```

Inheritance and Composition

▼ Identifying inheritance situations

inheritance allowed one or more classes to get the same information of another class



- How can I know that i can inheritance something?

- Inheritance describes an "Is a" Relationship
- Example
 - A Starfighter is a Spaceship

▼ Problem of

- Powerful , but it can lead to inflexible desings
- All classes inherit the same behavior

▼ Using inheritance

891 aktive Nutzer:innen

Calling a Method in the Super/Parent/Base Class

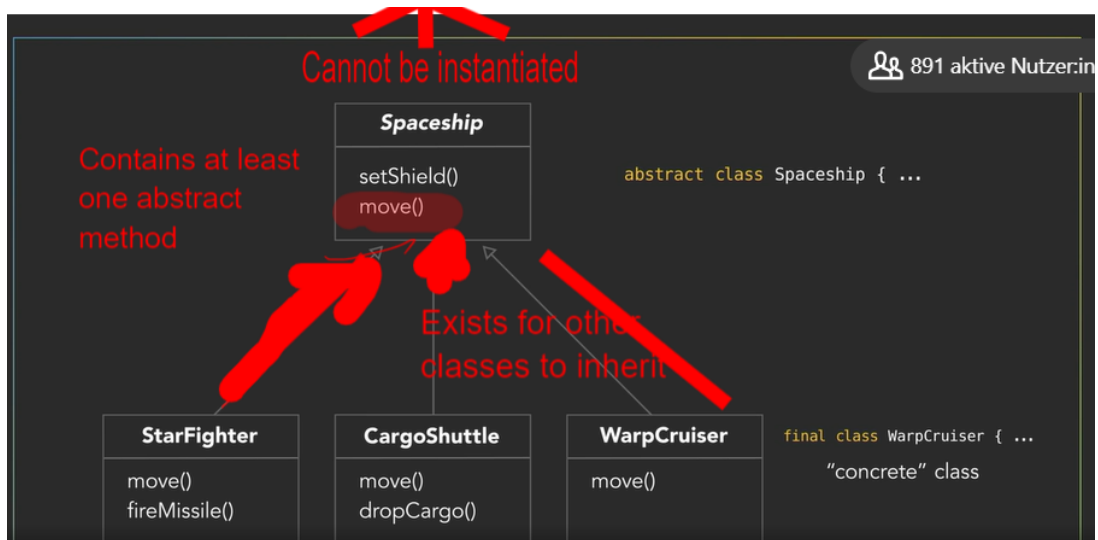
```

classDiagram
    class Spaceship {
        setShield()
    }
    class CargoShuttle
    Spaceship <|-- CargoShuttle
  
```

Java	<code>super.setShield()</code>	Ruby	<code>super set_shield</code>
C#	<code>base.setShield()</code>	C++	<code>Spaceship::setShield()</code>
Swift	<code>super.setShield()</code>		
Python	<code>super().setShield()</code>		

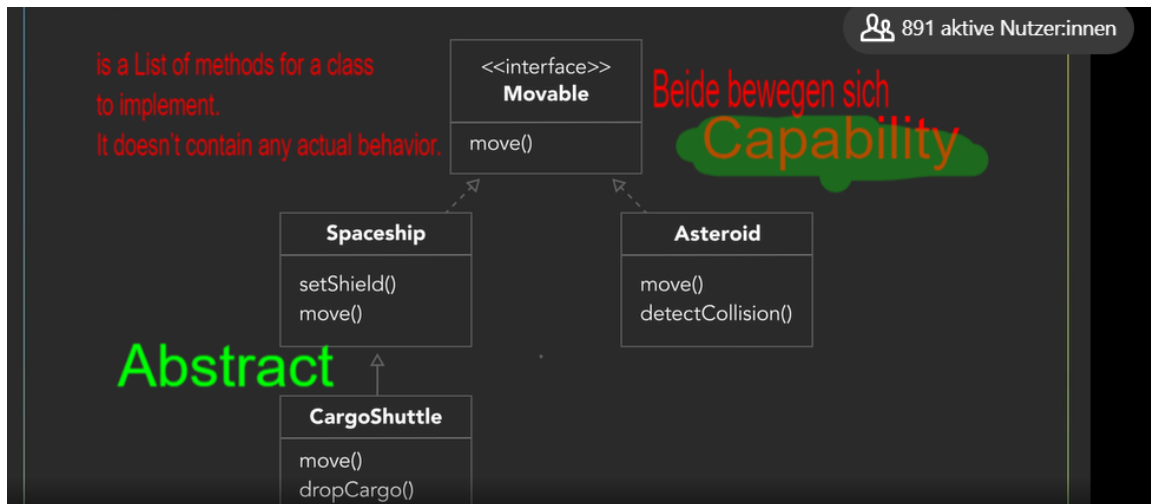
▼ Abstract and concrete classes

- Abstract Class : represent a **type**
 - Exists for other classes to inherit
 - Cannot be instantiated
 - Contains at least one abstract method
 -



▼ Interfaces

- is a List of methods for a class to implement. It doesn't contain any actual behavior.
- Interfaces represent a capability.
-

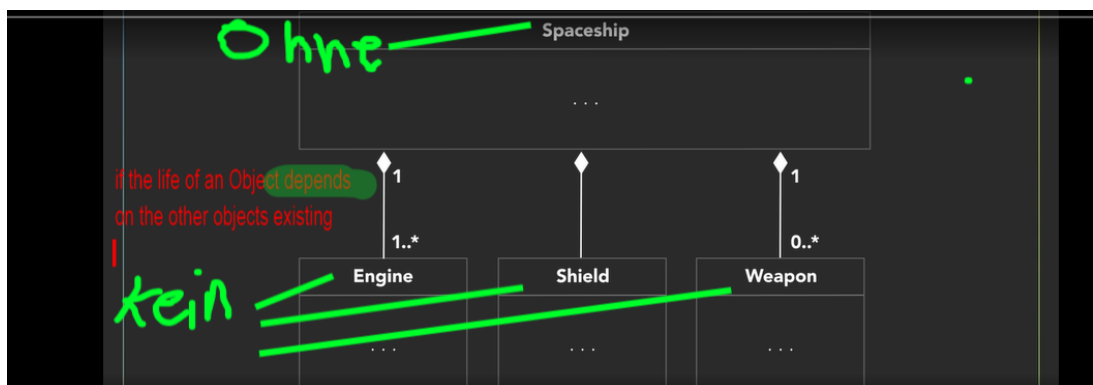


▼ Aggregation

- One Object is build on other objects
- How can I know when I have to use it?
 - Aggregation Describes a “Has a” Relationship
 - Example
 - A Fleet has a Spaceship
 - or has many
 -

▼ Composition

- A Spaceship owns an Engine
 - if the life of an Object depends on the other objects existing
 -



- We can still “inherit “ behavior
- We can make dynamic runtime decisions
- We can add new behavior without altering existing code
- We can include behaviors not considered by the creator

Software development

▼ OOP support in different languages

Object-Oriented Languages 893 aktive Nutzer:innen

Language	Inheritance	Call to Super	Typing	Interfaces	Abstract Classes
Java	single	super	static	Yes	Yes
C#	single	base	static	Yes	Yes
Python	multiple	super	dynamic	Abstract Class	Yes
Swift	single	super	static	Protocols	No
C++	multiple	name of class::	static	Abstract Class	Yes
Ruby	mixins	super	dynamic	n/a	n/a
JavaScript	prototypes	n/a	dynamic	n/a	n/a

▼ General development principles

- Code should be : S O L I D
 - Single responsibility principle
 - No Godobjects , classes should be splitted , when there are to big
 - Don't repeat yourself
 - Code Smell
 - Any characteristic in a program's code that possibly indicates a deeper problem
 - Open / closed principle
 -
 - Liskov substitution principle

- Interface segregation principle
- Dependency inversion principle