



UNIVERSITY  
OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

## 6. Object-Oriented Analysis

Matthias Book  
Spring 2022

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL  
ENGINEERING AND COMPUTER SCIENCE

# Mid-Semester Evaluation

Evaluate this course on Uglu!  
(10–15 February)



# Recap: Peer Feedback on Assignment Drafts

- **Current pre-submission feedback from tutors**
  - Show a draft of your assignment to your tutor on Wednesday
    - Tutor gives you spontaneous feedback
  - Submit your final assignment by Sunday
- **New: Additional optional peer feedback**
  - Submit an anonymous draft of your team's assignment by Wednesday
  - Your draft will be assigned to two other students for peer feedback
    - and you'll be assigned drafts of two random other students
  - Submit anonymous feedback to the received drafts by Friday
    - and you'll get anonymous feedback from two students on your draft as well
  - Rate how helpful the feedback that you've received has been
    - and you'll receive a rating of the quality of the feedback you've given
  - Submit your final assignment by Sunday

# Update: Peer Feedback on Assignment Drafts

- **You're invited to “test drive” a beta version the peer feedback this week:**
  - Submit an anonymized version of your team's user stories to the “Peer Feedback Test” assignment on Canvas by Wednesday
  - Give feedback on the submissions assigned to you by Friday
  - Rate the quality (depth, helpfulness) of the feedback you have received by next Wednesday
- Participation is optional, independent from the process for Ass. 1a and 1b, and ungraded.
- **Grading policy**
  - ~~a) Ungraded: You receive feedback in exchange for giving feedback; quality rating is just FYI~~
  - b) Optional: Quality of feedback you provide determines a bonus given on your project grade
    - Fallback: Presence of feedback determines bonus on your project grade
    - Similar to the quiz grade counting as a bonus on your exam grade
  - c) Mandatory: Quality of feedback you provide determines a personal feedback grade
    - Similar the presentation grade counting as a personal part of the project grade

This semester

# Online vs. On-site Teaching?

- COVID-19 restrictions have been eased to allow 200 students without masks in one classroom, regardless of distance.
- How would you prefer to follow the lectures?
  - live on Zoom
  - live in a classroom
  - asynchronously via recordings
- Lecture recordings will always be published on Canvas either way.
- Consultations will continue to proceed on Discord.
  - More convenient for students to drop in for their 20-minute slot
  - Safer for tutors not to be exposed to lots of people talking simultaneously for 3 hours in one room

# Recap: Velocity

- Velocity is a measure of how productively a team uses the available work days.
- a) Can **either** be used to calculate how many work days a user story will really need:
  - $\text{Person-days} / \text{people} / \text{velocity} = \text{required work days}$
- b) **or** can be used to calculate how many person-days fit into an iteration:
  - $\text{Work days} \times \text{velocity} \times \text{people} = \text{available person-days}$
- Begin project with an assumed velocity of 70%
- Calculate new project-specific velocity at the end of each iteration:
  - $\text{Person-days completed} / \text{Person-days originally available}^* = \text{new velocity}$ 
    - \*not accounting for old velocity
- Applying velocity may make it harder to fit your project into the available timeframe, but it will make your project plan more realistic.



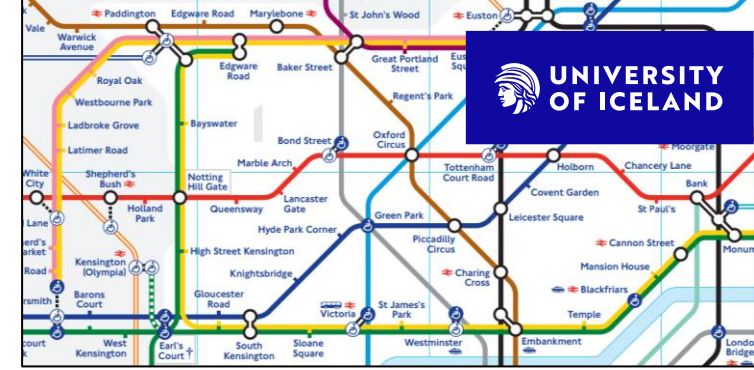
# In-Class Quiz #5 Solution

- Assume you are a 2-person team working on a project with 2-week sprints.
- a) Your initial velocity is 75%. How many *person-days* of work should you at most commit to for the first sprint?
  - $2 \text{ persons} \times 2 \text{ weeks} \times 5 \text{ days/week} \times 0.75 = 15 \text{ person-days}$
- b) At the end of the first sprint, you have managed to complete 10 person-days of work. What is your new *velocity*?
  - $10 \text{ person-days} / (2 \text{ persons} \times 2 \text{ weeks} \times 5 \text{ days/week}) = 0.5$
- c) At the beginning of the next sprint, you have an estimated 6 person-days of uncompleted work left to do from the previous sprint. How many *person-days* of *new* work can you at most commit to in this second sprint, given the new velocity and leftover work?
  - $2 \text{ persons} \times 2 \text{ weeks} \times 5 \text{ days/week} \times 0.5 - 6 \text{ person-days} = 4 \text{ person-days}$
- d) Assuming your team needs to complete all leftover work before you can start on anything new, how many *work days* will you probably have left for the *new* work in the sprint?
  - $2 \text{ weeks} \times 5 \text{ days/week} - 6 \text{ person-days} / 2 \text{ persons} / 0.5 = 4 \text{ work days}$



# Recap: Models and Modeling Languages

- A model is a simplified (abstract) representation that highlights or omits certain aspects of reality.
  - *Which aspects are highlighted or omitted depends on the model's purpose.*



## Unified Modeling Language (UML)

- Provides a number of diagram types to model various aspects of a system, e.g.
  - **Class diagrams** for system components
  - **Sequence diagrams** for component interaction
- Terminology in UML:
  - **Model:** The abstract representation of (existing or planned) reality
  - **Diagram:** The visual representation of a certain aspect of / view on the model
- A particular model element (e.g. a class/object) may occur in several diagrams.



# Recap: Object-Orientation (OO)

- **Encapsulation:** Technical implementation of an item's internal operation
  - is usually neither known nor publicly documented
  - is usually (ideally!) not required for use
- Only the (ideally simple) interface must be known to the user.
- **The ability to remain ignorant of an item's internal implementation and state is what enables the simple use of most real-life items.**
- Object-orientation means that we view our software as **a system of cooperating objects.**
  - Think of real-world context, our models, and our software all in the same way (as objects)
  - Make responsibilities and access rights explicit and enforce them (encapsulation)
  - Reuse and extend functionality efficiently (inheritance, polymorphism → next week's class)



The object-oriented view permeates all phases of software construction:

- **Object-Oriented Analysis (OOA) → yields the Domain Model**
  - Identifying classes, their relationships and behavior in the reality of the application domain
    - Exploring the application domain, identifying business processes, identifying the objects handled by those processes, deriving classes from those objects, defining collaboration between classes...
- **Object-Oriented Design (OOD) → yields the Design Model**
  - Refining the models created during analysis to reflect technical (implementation) needs
    - Solutions for user interaction, data storage, component distribution, parallel/asynchronous execution; choice of suitable algorithms; optimization of data structures...
- **Object-Oriented Programming (OOP) → yields the Implementation**
  - Expressing the models created during design in a programming language (incl. refinement)
    - Adaptation to language specifics, use of libraries, additional low-level technical objects (e.g. for exception handling), conversion of types, implementation of algorithms...

# Domain Models: Classes and Class Diagrams

see also:

Learning UML 2.0, Chapters 4 and 5



**An object has a state, behavior and identity.**

- **State** is defined by the object's **attributes**
  - i.e. containers for changeable information
  - Typically encapsulated, i.e. not visible to the outside world, but only accessible by the object's internal implementation
- **Behavior** is defined by the object's **methods**
  - i.e. operations that the object can perform
  - Typically only some are visible to the outside world, constituting the object's interface, the rest is encapsulated
- **Identity** is defined by the object's **existence**
  - as a physical object in the real world
  - as an instance in computer memory addressable by a pointer (in Java: "reference")



# Objects with the Same Characteristics

- Example objects:
  - red VW Polo, 1.4L twin-charger straight-4 16-valve petrol, 130 kW, 4 doors, 2.344 km
  - blue VW Polo, 1.6L straight-4 16-valve petrol, 77 kW, 4 doors, 14.490 km
  - white VW Polo, 1.2L turbocharged straight-3 12-valve diesel, 55 kW, 4 doors, 90.341 km
- Same attributes
  - Color, engine, doors, mileage...
    - with *different* values
- Same methods
  - Drive, steer, brake, wipe, heat...
    - with *same* implementations
      - (we'll talk about *similar* implementations soon too)
- Can be described by a common “blueprint”
  - Class: Car



## ■ Class

- A definition of the common characteristics of a set of objects, i.e.
  - the types of its attributes
  - the implementations of its methods
- In software: Defined at design time



## ■ Object

- An individual instance of a class in which all of the class' characteristics manifest themselves
- In software: Created and destroyed at run time

- **A class is a “blueprint” for the creation of similar objects.**

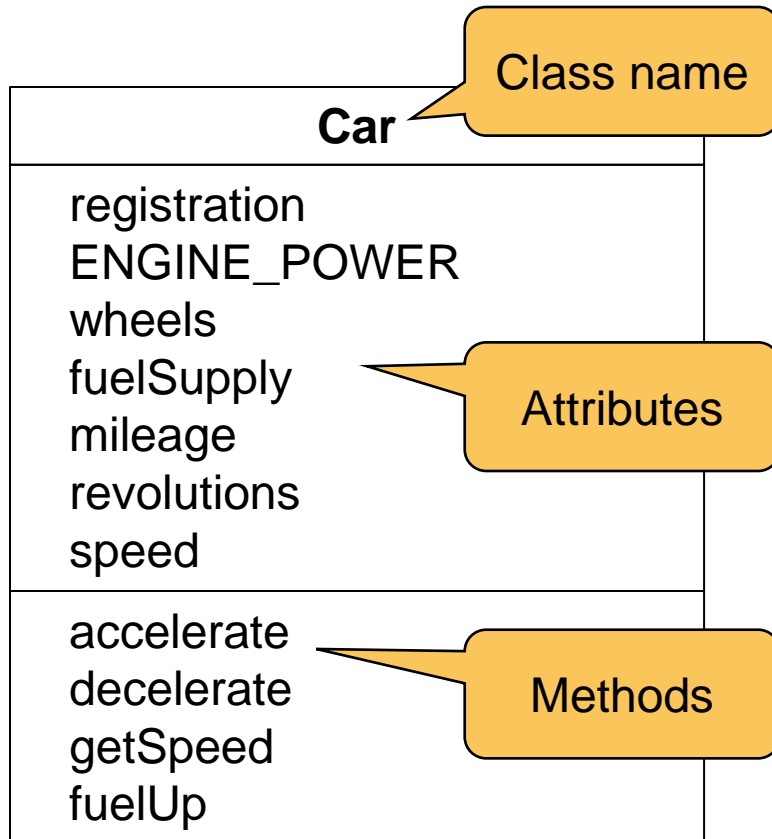




# Break



# Modeling Classes in UML Class Diagrams

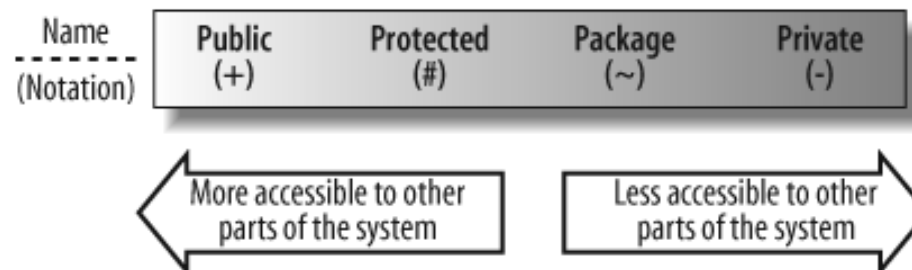


- A class is specified with
  - its name (choose a clear one; in singular form)
  - optionally: its attributes
  - optionally: its methods
- Refine a class as you learn more about it
  - Model only what you are certain about, never guess
    - Readers of your diagrams can't tell where you were sure and where you were not!
- Naming conventions
  - Classes: MixedCase, uppercase first letter
  - Attributes, methods: mixedCase, lowercase first letter
  - Constants: ALL\_CAPS with underscores

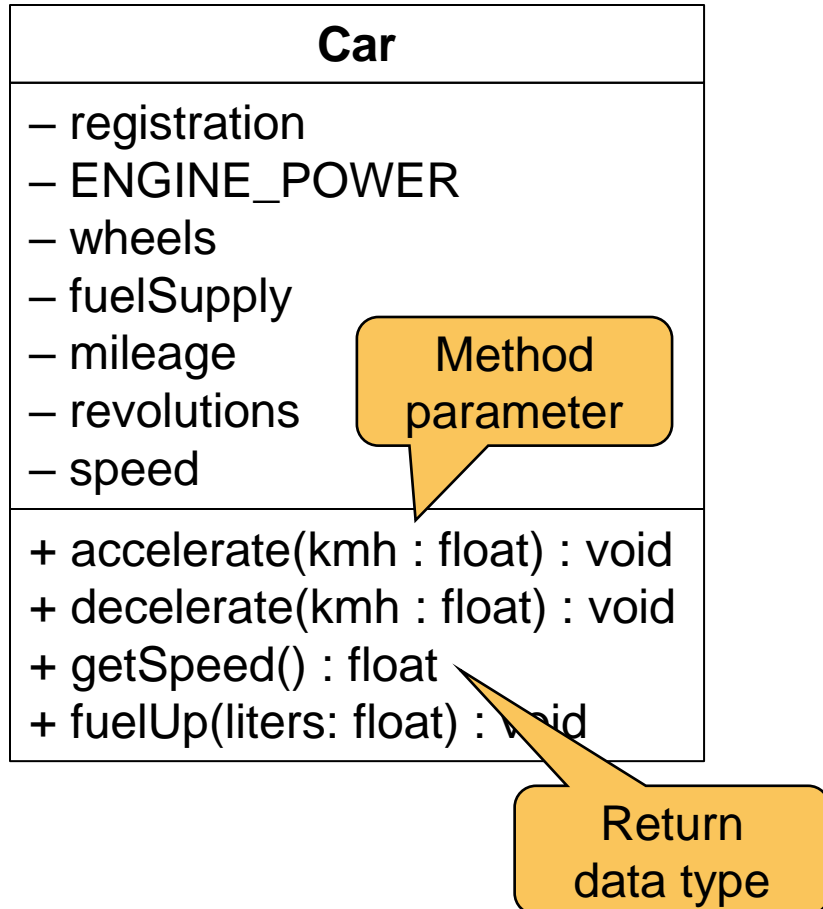
Car
<ul style="list-style-type: none"><li>– registration</li><li>– ENGINE_POWER</li><li>– wheels</li><li>– fuelSupply</li><li>– mileage</li><li>– revolutions</li><li>– speed</li></ul>
<ul style="list-style-type: none"><li>+ accelerate</li><li>+ decelerate</li><li>+ getSpeed</li><li>+ fuelUp</li></ul>

Visibility  
modifier

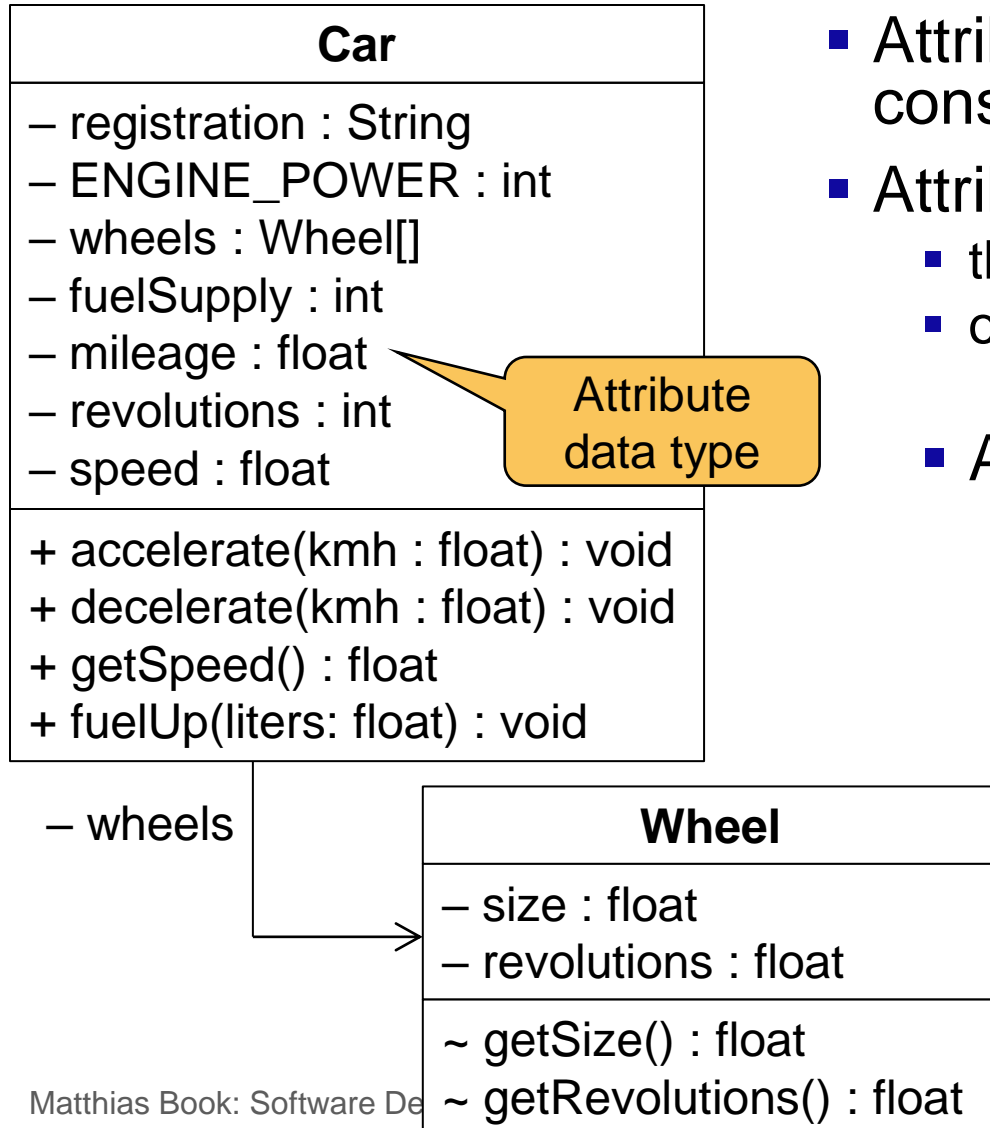
- To facilitate encapsulation, the visibility of classes, attributes and methods can be restricted with visibility modifiers.
- Attributes almost always have private visibility.
  - Instead of being directly manipulated by others, they should only be changed by methods in the same class.
  - Ensures that you have control over the attributes' values and can prevent any invalid values that may lead to errors.
- Methods may have any of the visibilities, depending on who they are providing functionality to:



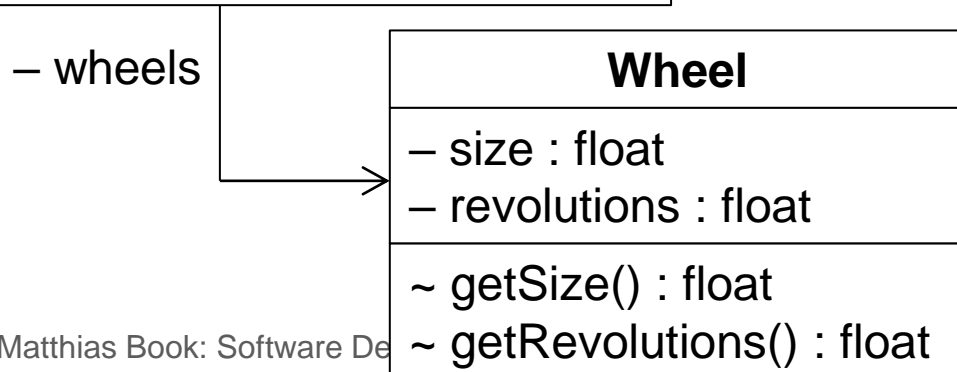
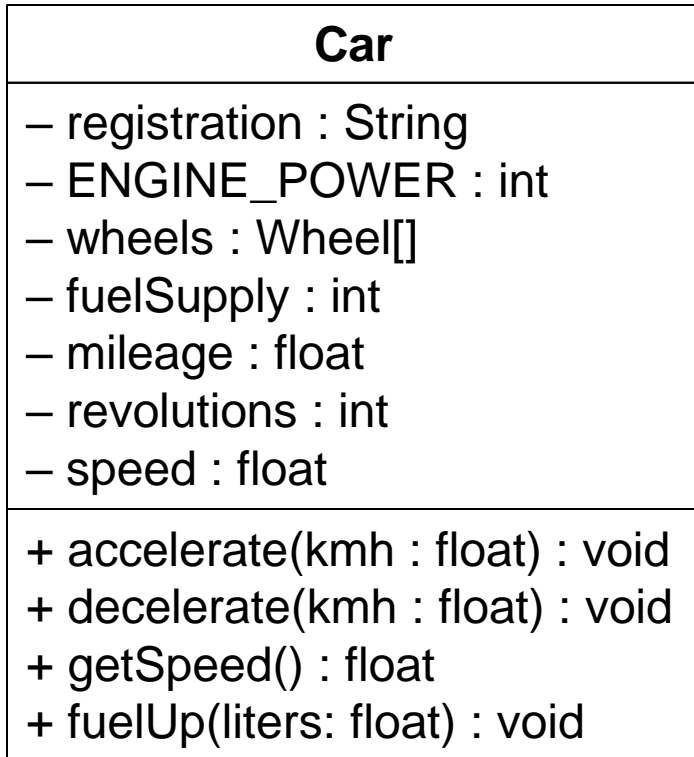
*(more on this later...)*



- Methods in a class diagram express *what* an object derived from this class will be able to do at run-time
  - but not *how* it will do it (that will be defined during OOP)
- Methods should only
  - operate on the attributes of their own class, and
  - call methods of this or other classes to access other data
- Methods are specified with
  - their name (choose a clear one, ideally start with a verb)
  - optionally: their parameters (may lead into OOD)
    - i.e. data being passed to the method by the caller
  - optionally: their return type (may lead into OOD)
    - i.e. data returned to the caller



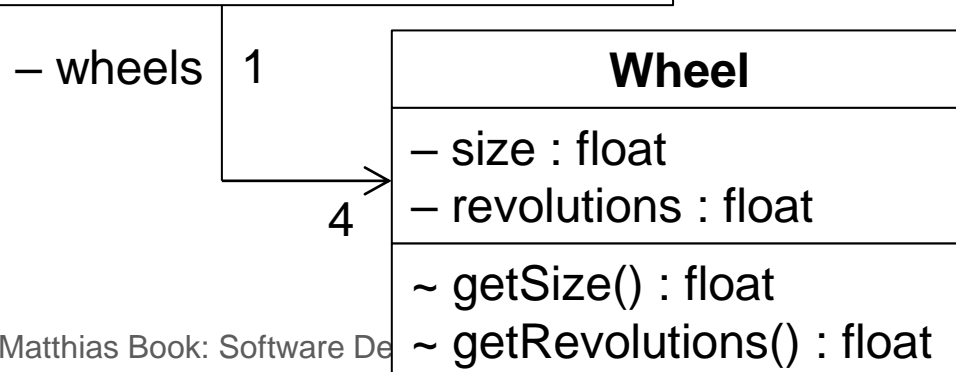
- Attributes are data stores for the information that will constitute an object's state at run time.
- Attributes are specified with
  - their name (choose a clear one!)
  - optionally: their data type (leads into OOD)
- Attributes can be specified
  - always: inline (i.e. listed in the attributes box)
    - Always done for primitive types (int, float etc.) and simple classes (String, Date etc.) from class libraries
  - optionally also: by association (i.e. drawn as a separate class)
    - Usually done for more complex classes of your own system / business domain
    - To visualize complexity, dependencies, multiplicities



- An association indicates that instances of this class will contain a reference to an instance of another class as an attribute at run-time.
- Associations are specified as arrows with
  - an open-tip arrowhead pointing to the referenced class
  - the name of the attribute containing the reference at the arrow's foot
  - optionally: multiplicities at the arrow's ends
    - \* = infinite amount
    - $n$  = finite amount (e.g. 1, 4 etc.)
    - $m..n$  = interval (e.g. 0..1, 0..\*, 1..\*, 2..4 etc.)
- read: “has [a]” (e.g. “a car has a wheel / has wheels”)

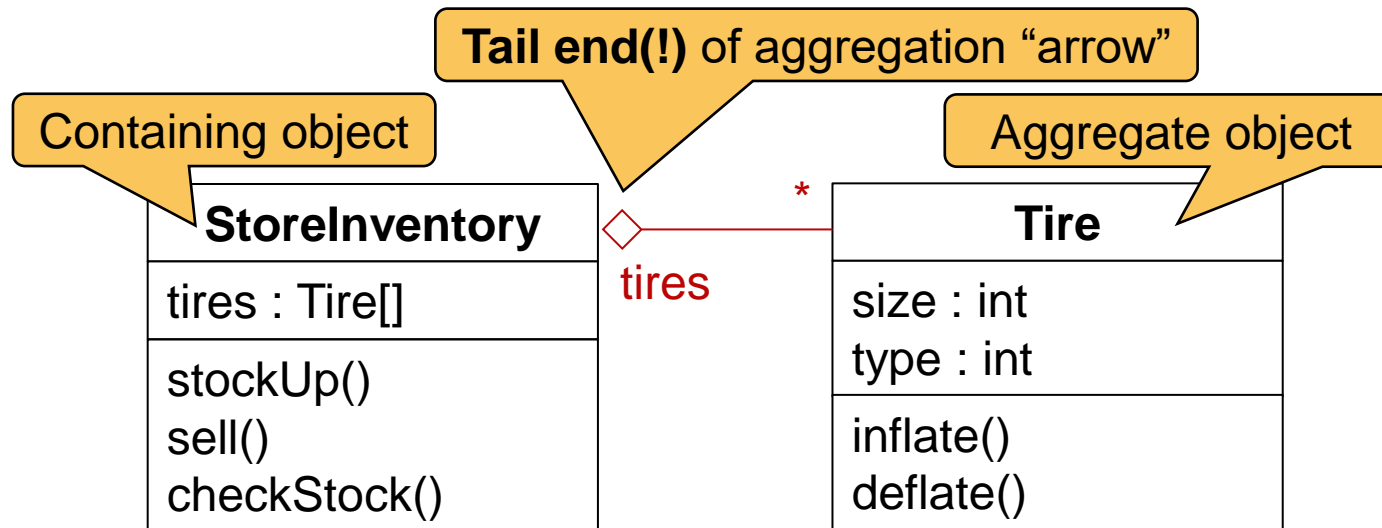


Car
<ul style="list-style-type: none"><li>– registration : String</li><li>– ENGINE_POWER : int</li><li>– wheels : Wheel[]</li><li>– fuelSupply : int</li><li>– mileage : float</li><li>– revolutions : int</li><li>– speed : float</li></ul>
<ul style="list-style-type: none"><li>+ accelerate(kmh : float) : void</li><li>+ decelerate(kmh : float) : void</li><li>+ getSpeed() : float</li><li>+ fuelUp(liters: float) : void</li></ul>

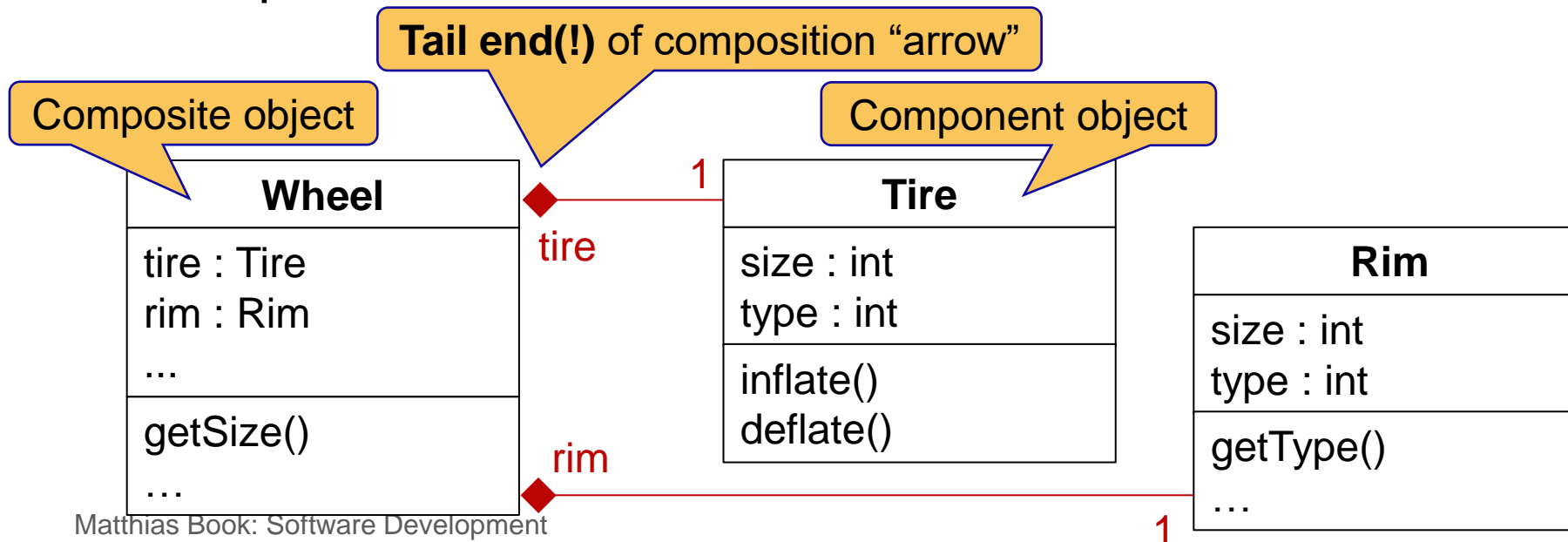


- Multiplicities indicate the allowed number of associated class instances (objects).
  - No default if not specified: clarify with client unless obvious
- A multiplicity at the arrow's head indicates how many objects the attribute can refer to
  - here e.g.: A car must have exactly four wheels.
  - or: 0..\* ("X can reference none or many instances of Y")
- A multiplicity at the arrow's foot indicates by how many of these objects an object can be referenced
  - here e.g.: A wheel can only be part of one car.
  - or: 1..\* ("Y can be referenced by one or many instances of X")





- To indicate that one object serves as a container for a collection of other objects, we can use an **aggregation** instead of an association arrow.
  - Both the container and the contained objects can exist independently of each other.
- Example: A store's inventory holds many tires.
  - The store exists independently of the tires in its inventory, the tires exist independently of the store, and can be temporarily associated with the store as they are in its inventory.



- To indicate that one object consists of other objects, we can use a **composition** instead of an association arrow.
  - Composition implies that the composed objects' existence depends on each other, i.e. if the composite object is destroyed, so are its individual components.
  - Can sometimes imply even stronger definition: Components cannot meaningfully (in the context of the system's application domain) exist separately from the composite object.
- Example: A car's wheel consists of a tire and a rim.



# Association vs. Aggregation vs. Composition

- **Association** expresses a “uses” relationship: 
    - Objects store references to other objects that they rely on for functionality
  - **Aggregation** expresses a “contains” relationship: 
    - Objects store collections of references to other objects that exist independently of them
  - **Composition** expresses a “consists of” relationship: 
    - Objects store references to other objects that are integral parts of them
-  Arrow direction
- All three relations are implemented in a similar way – as attributes of a class
    - Difference is in who is creating the instances, and who else gets references to them
  - **It is recommended to always use association, unless you specifically want to convey the special semantics of aggregation or composition.**
    - But make sure the multiplicities make sense (e.g. no  $n..m$  composition can exist)

# Association vs. Aggregation vs. Composition

- When to use which relationship cannot be answered in general. In particular...
  - whether some objects are integral components of an object (composition)
  - whether they are “collectible” elements of a container object (aggregation)
  - whether they are independent objects that are just referred to (association)
  - or whether their behavior requires the definition of an individual class at all
- ...depends highly on the application domain and purpose of your software.
  - Use common sense
  - Don't overthink things
  - **Use plain association when in doubt**

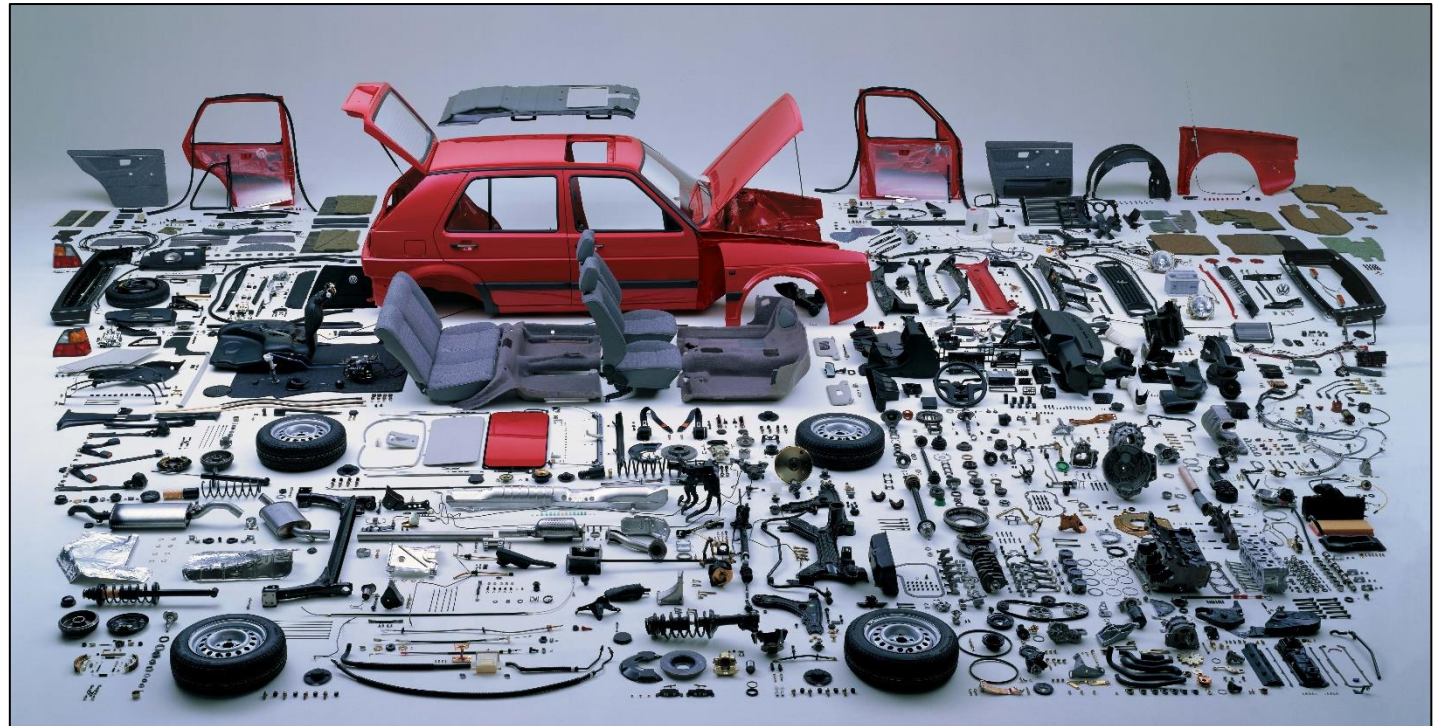


Image: Volkswagen

Car
<ul style="list-style-type: none"><li>– registration : String</li><li>– ENGINE_POWER : int</li><li>– wheels : Wheel[]</li><li>– fuelSupply : int</li><li>– mileage : float</li><li>– revolutions : int</li><li>– speed : float</li></ul>
<ul style="list-style-type: none"><li>+ accelerate(kmh : float) : void</li><li>+ decelerate(kmh : float) : void</li><li>+ getSpeed() : float</li><li>+ fuelUp(liters: float) : void</li></ul>

```
public class Car {  
    private String registration;  
    private final int ENGINE_POWER = 75;  
    private Wheel[] wheels;  
    private int fuelSupply;  
    private float mileage;  
    private int revolutions;  
    private float speed;  
  
    public void accelerate(float kmh) {...}  
    public void decelerate(float kmh) {...}  
    public float getSpeed() {...}  
    public void fuelUp(float liters) {...}  
}
```



# Team Assignment #2: OO Domain Model

- Create a domain model that defines:
  - The application-domain concepts that your system will need to handle (→ classes)
  - The properties that are characterizing instances of those classes (→ attributes)
  - The operations that can be performed on instances of those classes (→ methods)
  - The association, aggregation or composition relationships between those classes
- By **Sun 27 Feb**, submit in Canvas (as a PDF document):
  - A **UML class diagram** showing your project's **domain model**:
    - All classes that describe your team's application domain
      - including their attributes and methods (no need for data types)
    - Relationships (association, aggregation, composition) between classes
      - with multiplicities where appropriate
- On **Wed 2 Mar**, present and **explain** your model to your tutor:
  - Why did you structure the classes, their attributes and associations the way you did?
  - How will your system work with these classes via their methods?

Recommended simple  
and free drawing tool:  
<http://draw.io>

# In-Class Quiz #6: Object-Orientation

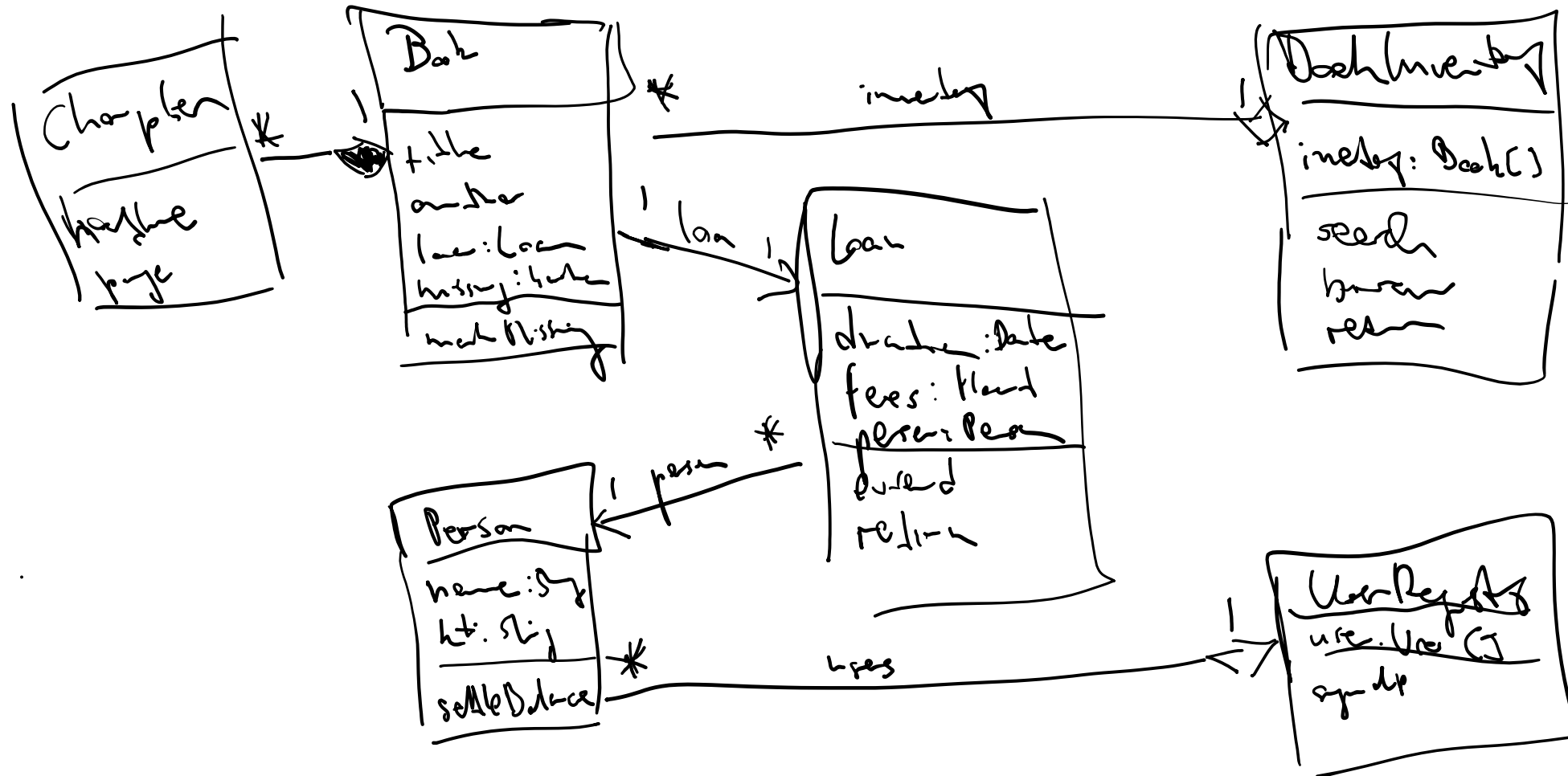
- Fill in the blanks with the words *attributes*, *class*, *implementations*, *methods*, *objects*, or *values*:
  - a) In the OO paradigm, we design a system as a set of collaborating \_\_\_\_\_.
  - b) A \_\_\_\_\_ defines the common characteristics of a set of \_\_\_\_\_.
  - c) A \_\_\_\_\_ is defined at a system's design time, while \_\_\_\_\_ are created and destroyed at run time of the system.
  - d) An object's state is defined by the \_\_\_\_\_ of its \_\_\_\_\_, while its behavior is defined by the \_\_\_\_\_ of its \_\_\_\_\_.
  - e) The \_\_\_\_\_ of different objects of the same class can have different \_\_\_\_\_, but their \_\_\_\_\_ have the same \_\_\_\_\_.
  - f) \_\_\_\_\_ should only be accessed by \_\_\_\_\_ of the same \_\_\_\_\_.

# Break



- Identify **classes** representing concepts of application domain
  - Usually, these are “data classes”
  - Use singular nouns to label these classes – each instance will represent one manifestation of the concept (e.g. one book, one student)
- Determine properties that characterize instances of these classes
  - These will be the **attributes** (e.g. a book’s title, publication year, etc.)
  - Many of these will be primitive types (e.g. title, year)
  - Some may have more complex structure of their own (e.g. a book’s current borrower)
    - These will be other classes related e.g. through association
- Determine functions these classes can perform / that can be performed on them
  - Those functions that work with one instance’s attributes become the class’ **methods** (e.g. retrieving a book’s title, or extending a book’s rental deadline)
  - Those functions that work with many instances need to be implemented in a different class (e.g. a BookInventory class that knows all Book instances)
- Identify classes representing complex relationships between other classes
  - E.g. a Loan class containing all details of the relationship of a Book being loaned by a Student

# Example Domain Model of a Library



# Thank you!

book@hi.is

