Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Software engineering techniques and tools

"A fool with a tool is still a fool."
- Grady Booch

Use case

- A use case in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system.
- In other words, a use case describes "who" can do "what" with the system in question.
- Use cases describe the system from the user's point of view.
- Each use case focuses on describing how to achieve a **goal** or task.

Use case - cont.

- Each use case should convey a primary scenario, or typical course of events, also called “basic flow”, “normal flow,” “happy flow” and “main path”.
  The main basic course of events is often conveyed as a set of usually numbered steps.
- Alternate paths can be written, e.g. next to the steps of the main path.

Use case: example

Main path	**Alternate path**
1. The system prompts the user to log on,	
2. The user enters his name and password	2.1 The user swipes an RFID card on a reader
3. The system verifies the logon information	
4. The system logs user on to system	

Use case - cont.

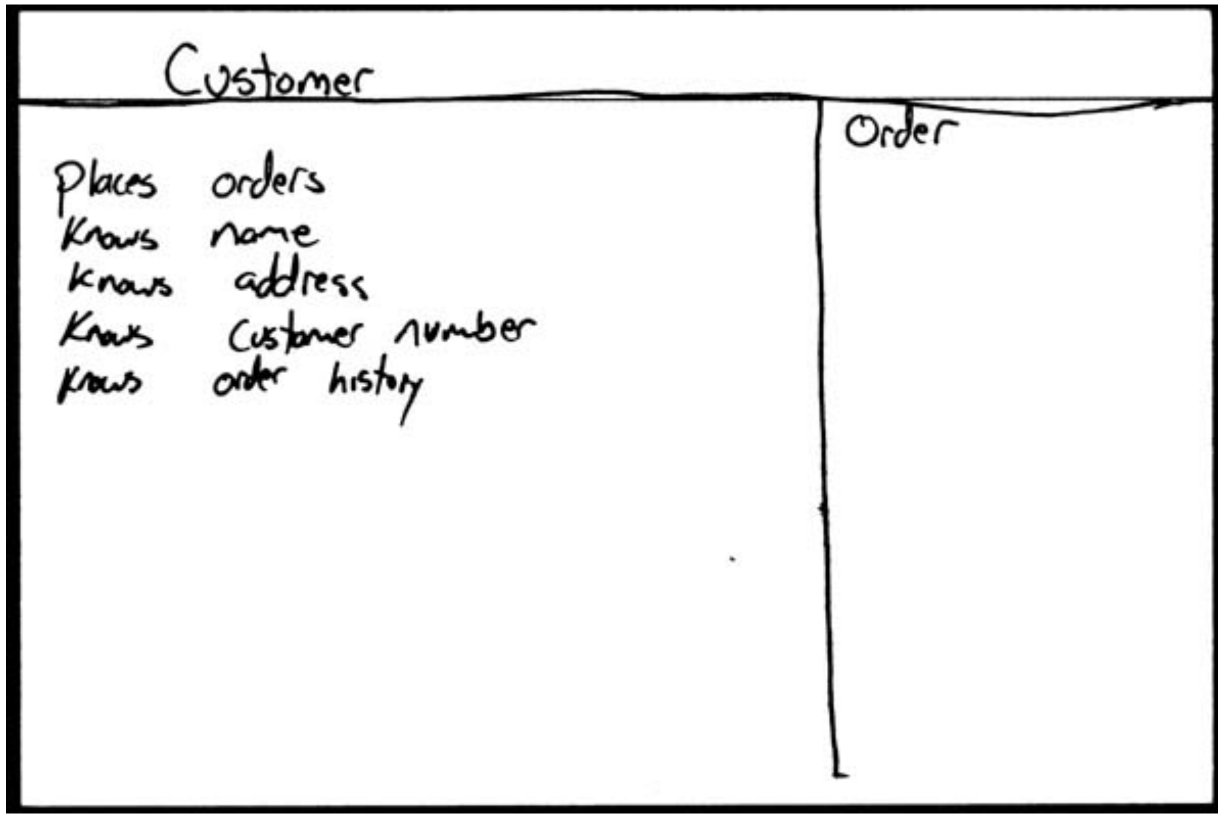
- Reread the use case, check it focuses on getting the task done.
- Pay attention to the nouns in the use case: they are candidates to identify the classes needed to model the system, and tell what to focus on
- Look at the verbs: they are candidates to identify the methods of the classes

CRC cards

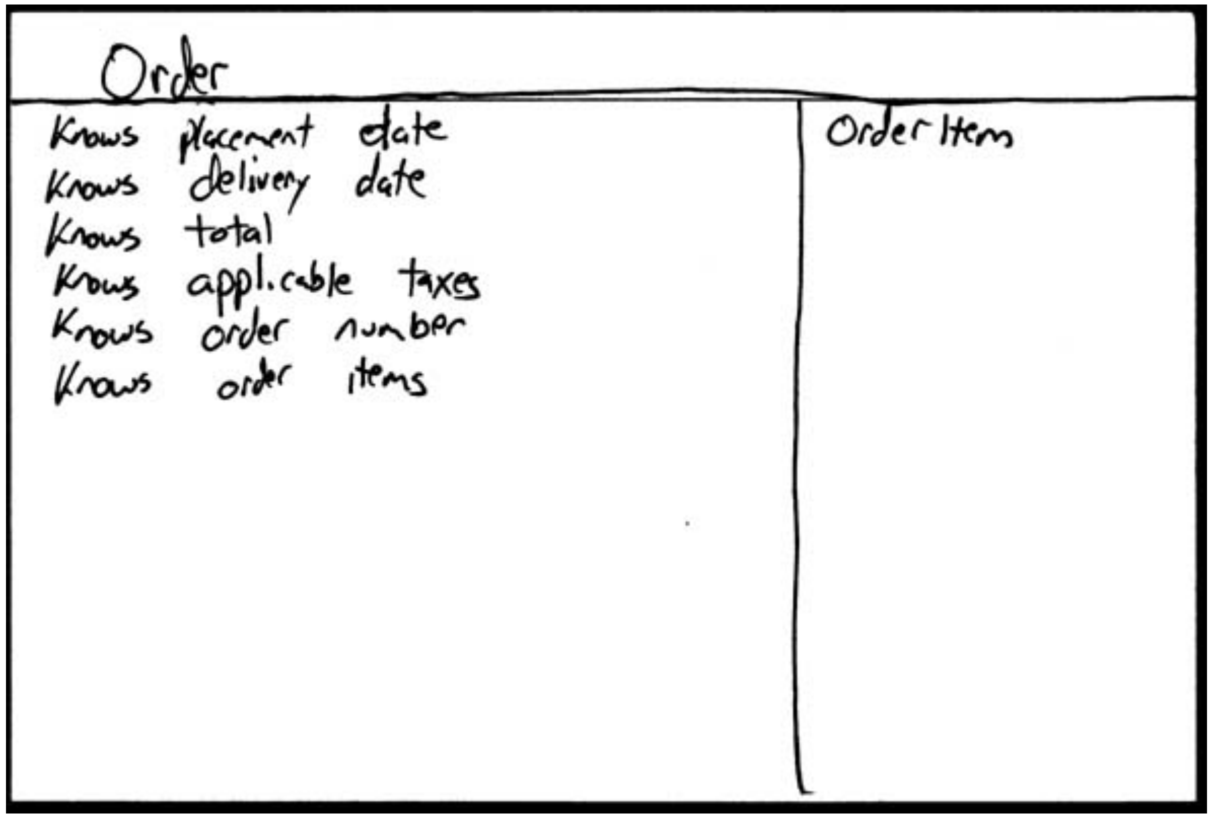
- CRC cards (Class, Responsibility, Collaborator) are a technique for discovering object classes, members and relationships in an object-oriented program.
- A **class** represents a collection of similar objects, a **responsibility** is something that a class knows or does, and a **collaborator** is another class that a class interacts with to fulfill its responsibilities.

CRC cards

Class Name	
Responsibilities	Collaborators



Customer	
Places orders Knows name knows address Knows customer number Knows order history	Order



Order	
Knows placement date Knows delivery date Knows total Knows applicable taxes Knows order number Knows order items	Order Item

CRC cards - cont.

- To create CRC classes iteratively perform the following steps:

1. Find classes: look for the three-to-five main classes
2. Find responsibilities: ask yourself what a class does as well as what information you wish to maintain about it.
3. Define collaborators: a class often does not have sufficient information to fulfill its responsibilities. Therefore, it must collaborate (work) with other classes to get the job done: requesting info or to perform a task
4. Move the cards around: it’s a method to understand the system: classes that collaborate should stay next each other

UML

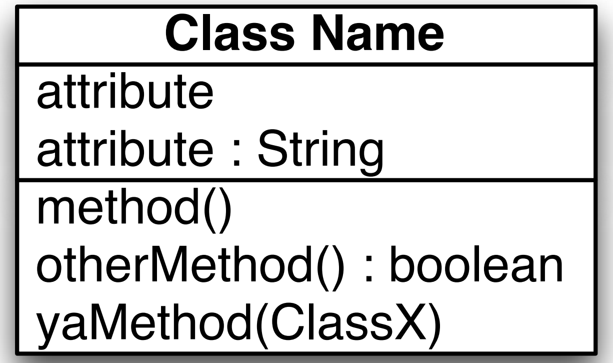
- UML (Unified Modeling Language) is a visual language for specifying, constructing, and documenting the artifacts of software-intensive systems.
- Complex software designs difficult for you to describe with text alone can readily be conveyed through diagrams using UML.
- Several tools help to draw UML diagrams, generate code from UML diagrams, generate UML diagrams from code.

UML Class diagram

- A UML class diagram describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.
- They are being used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code.

UML Class diagram - cont.

- In the class diagram these classes are represented with boxes which contain three parts:
- The upper part holds the name of the class
- The middle part contains the attributes of the class (and their type)
- The bottom part gives the methods or operations the class can take or undertake

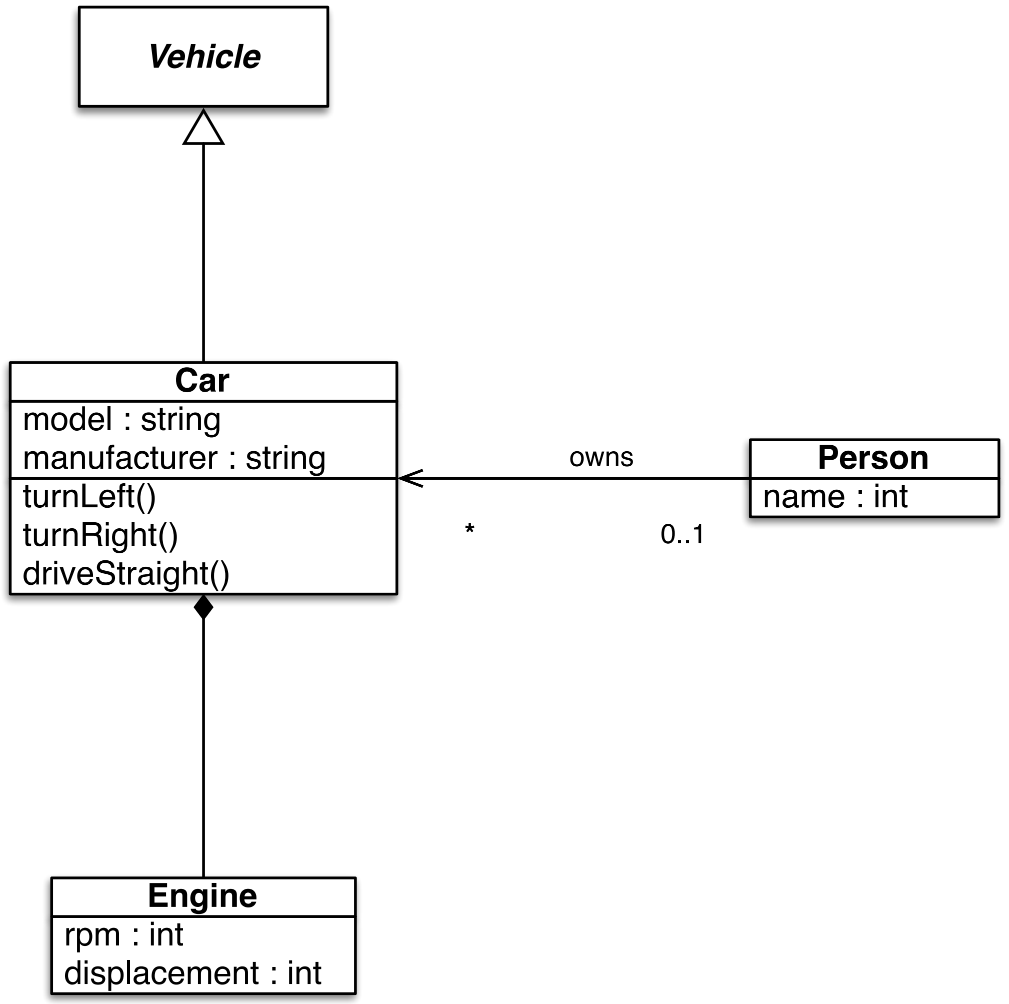


UML Class diagram - cont.

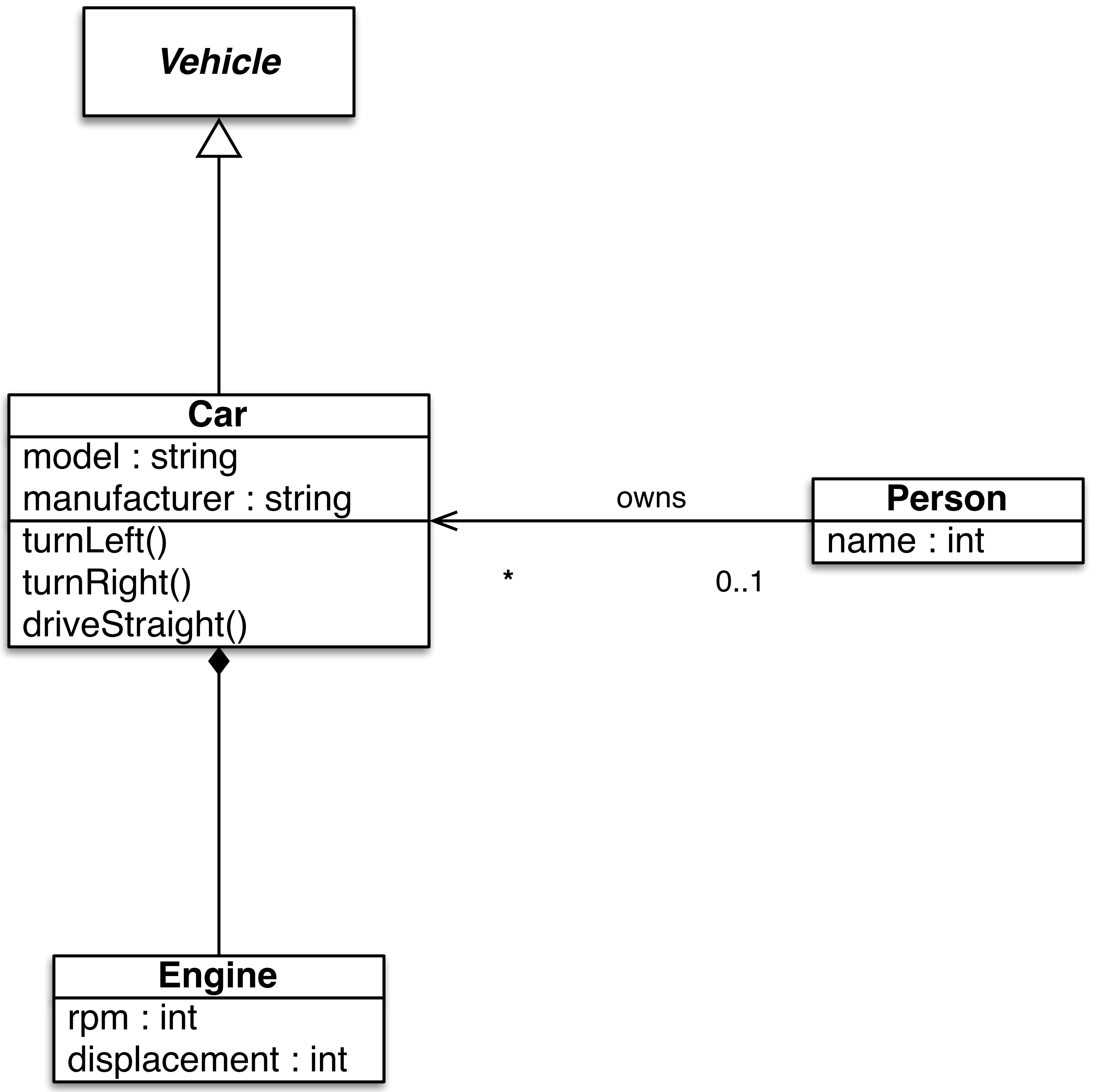
- In the conceptual design of a system a number of classes are identified and grouped together in a class diagram, which helps to determine the statical relations between those objects. With detailed modeling the classes of the conceptual design are often split in a number of subclasses.
- There can be several different types of relations among the classes, drawn as lines and arrows

UML Class diagram: example

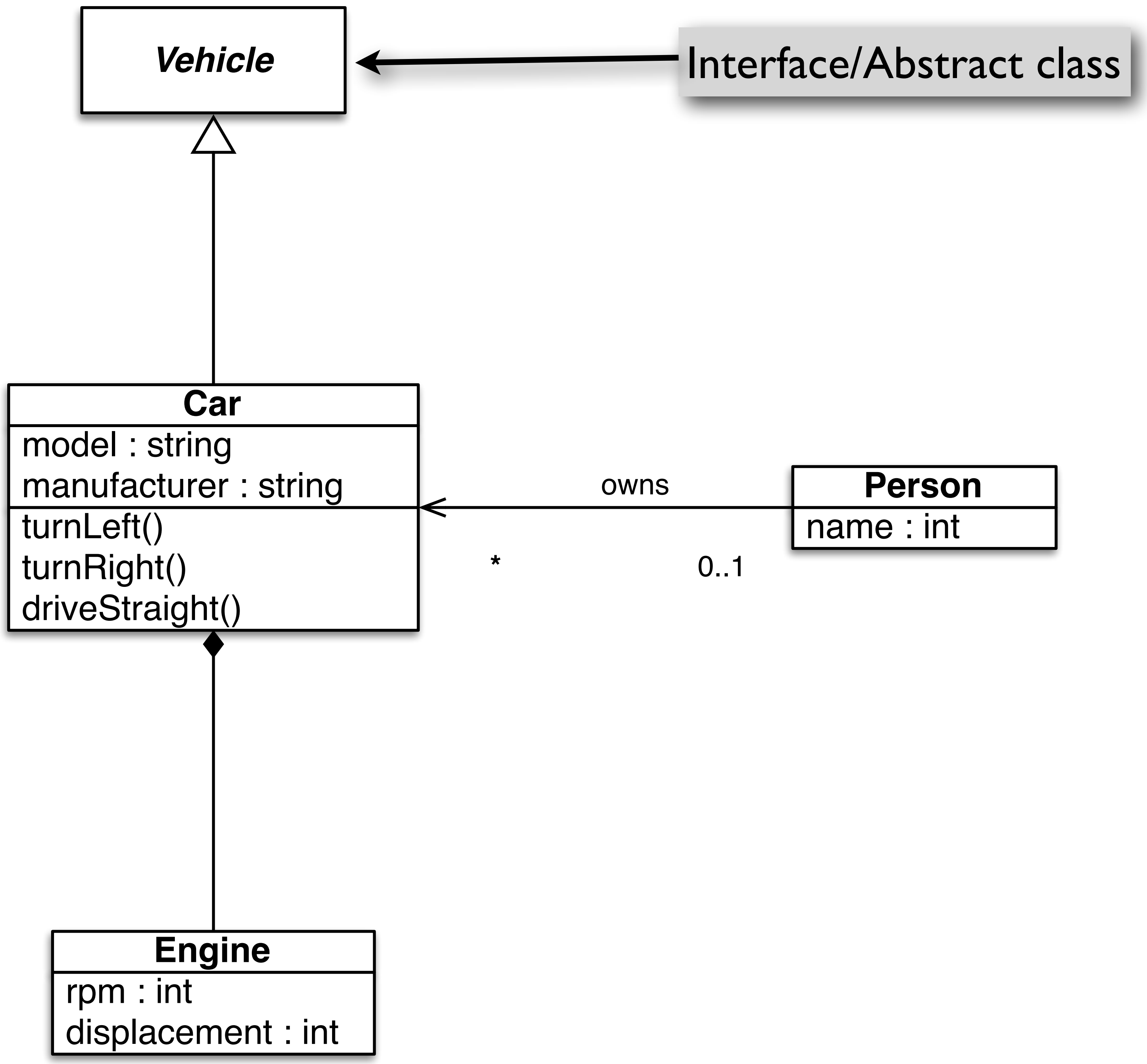
- Several tools allow to generate code from UML class diagrams, or reverse engineer code to UML class diagrams



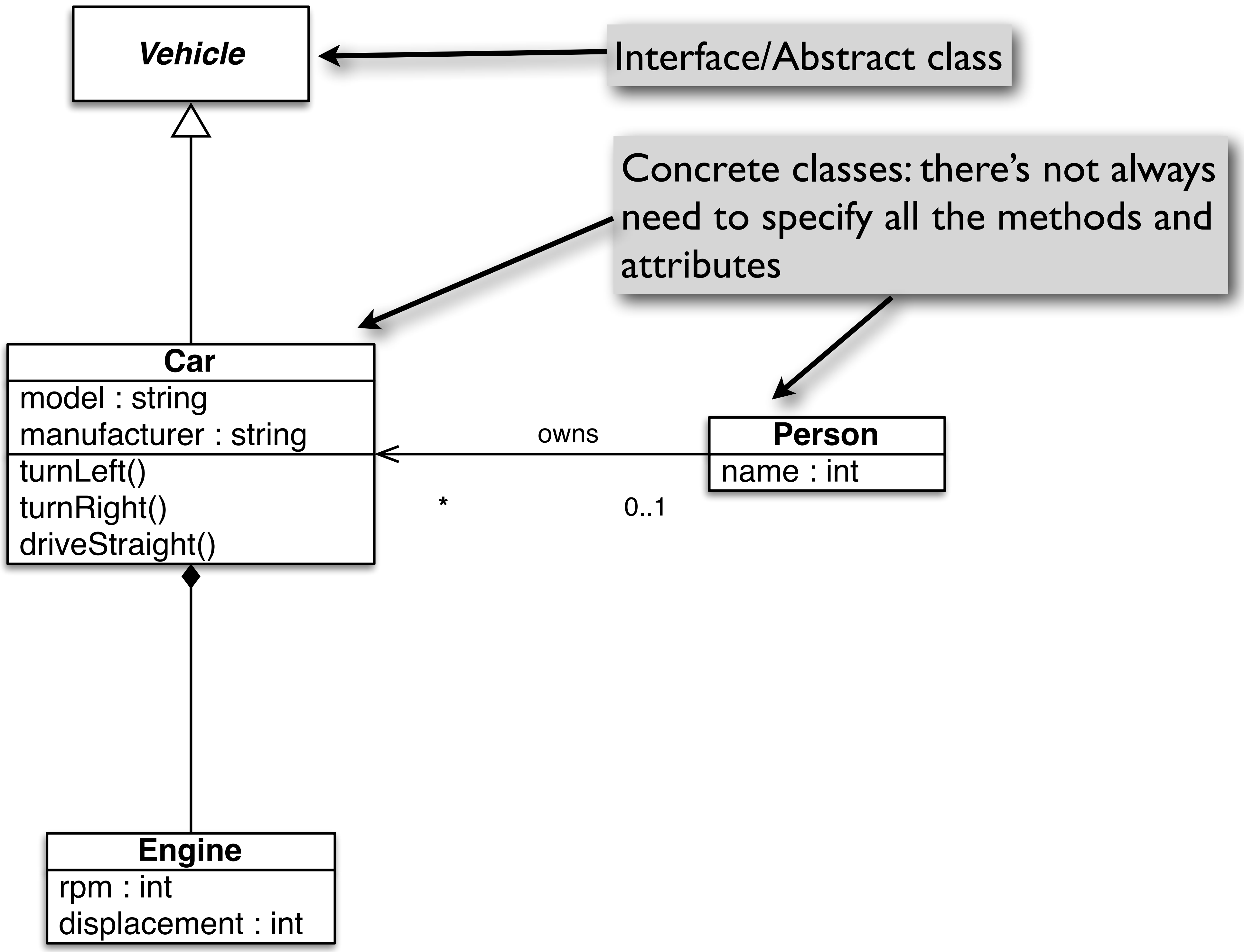
UML Class diagram



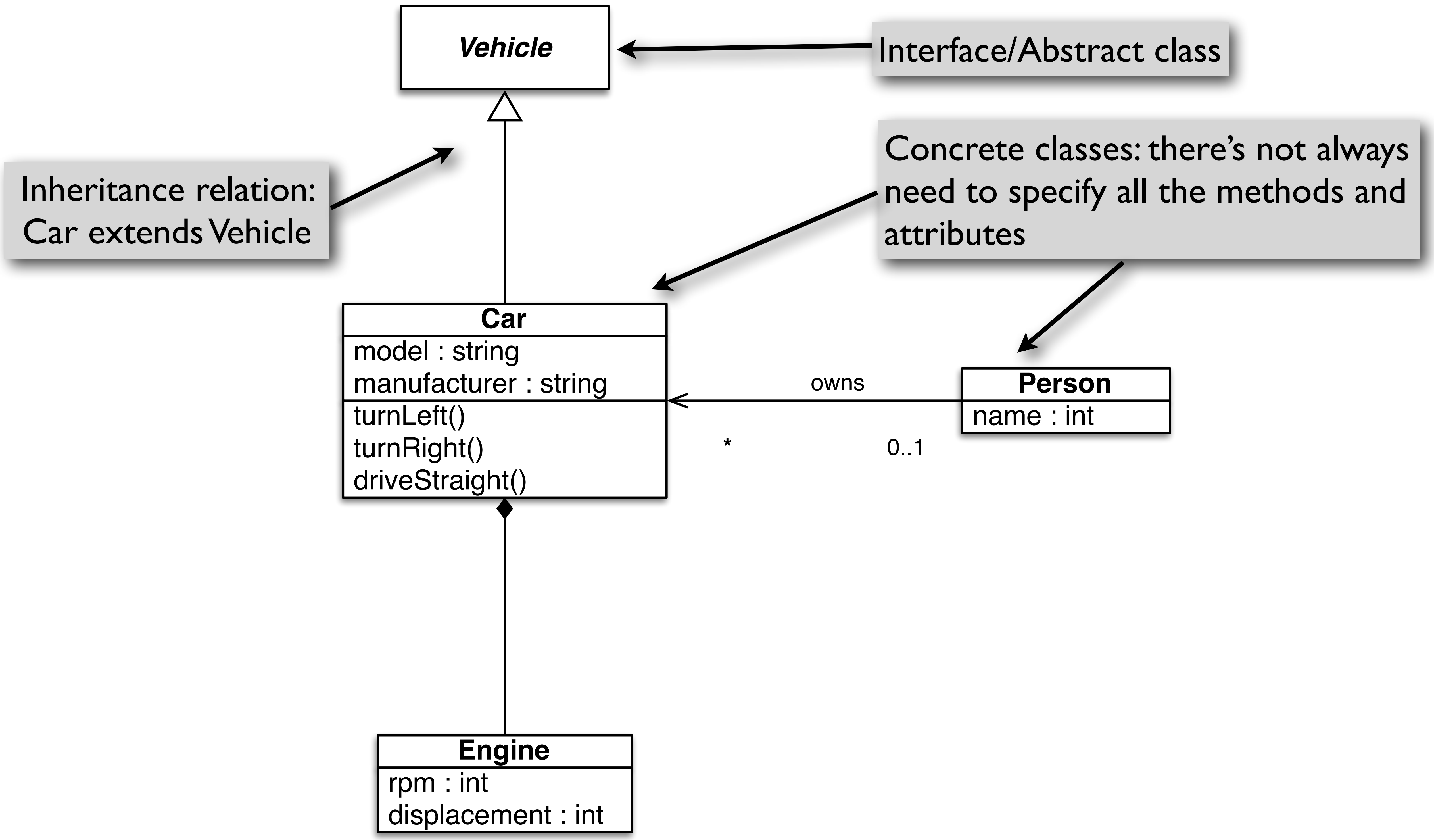
UML Class diagram



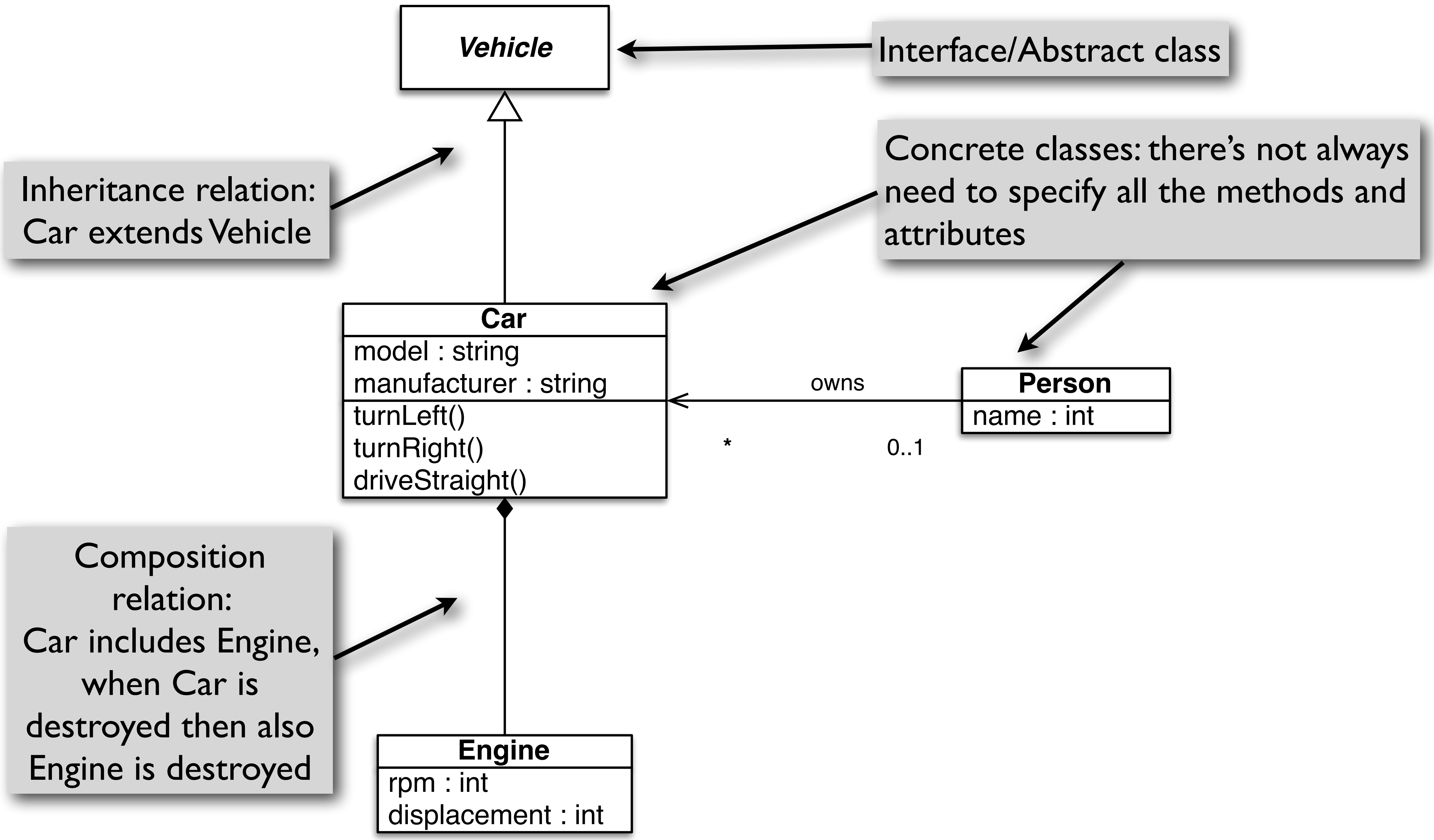
UML Class diagram



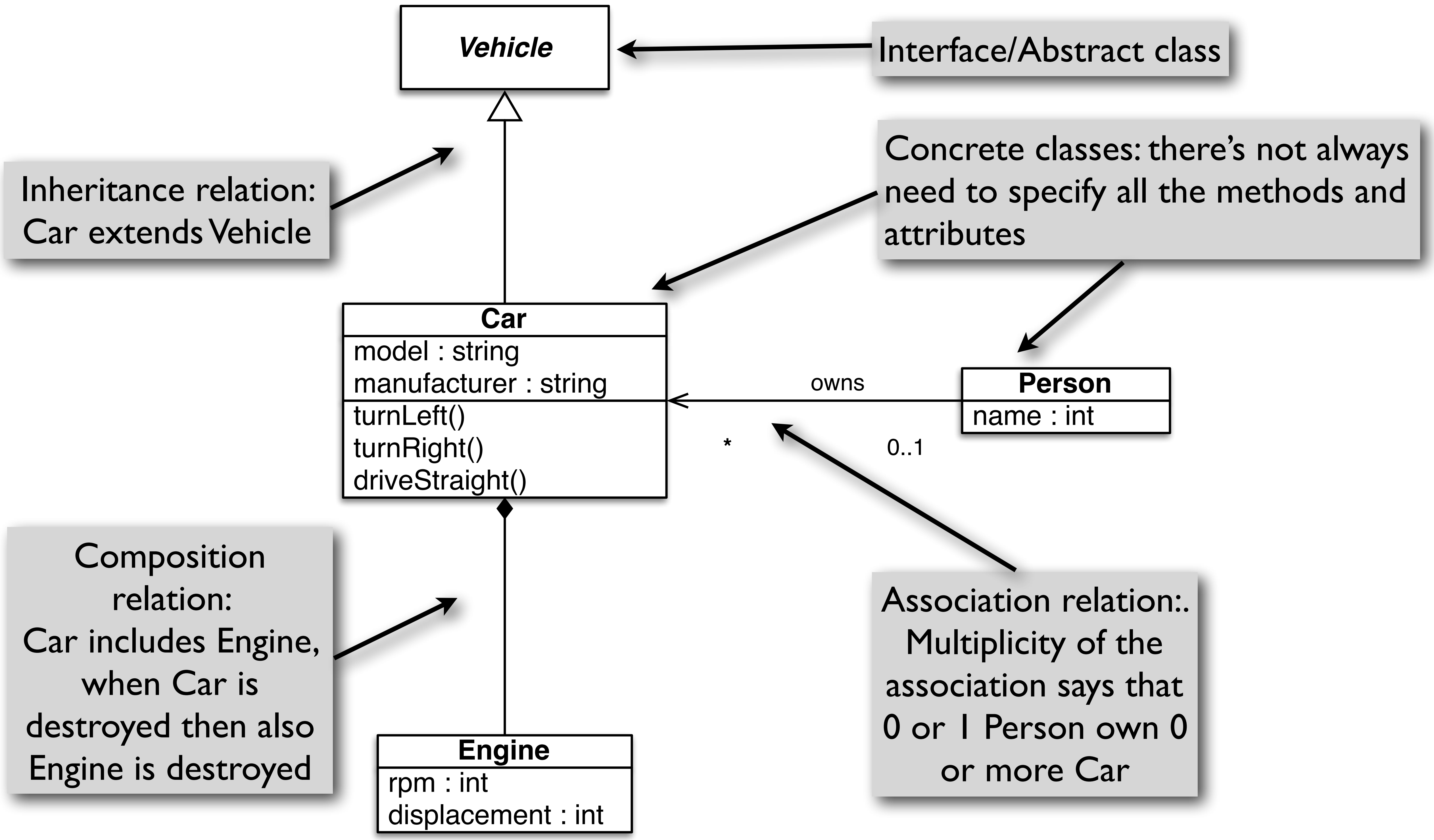
UML Class diagram



UML Class diagram

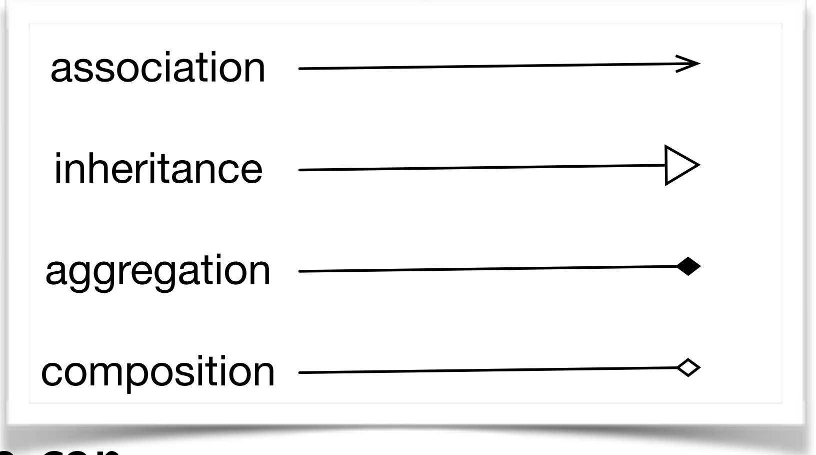

Vehicle
Interface/Abstract class
Inheritance relation: Car extends Vehicle
Concrete classes: there's not always need to specify all the methods and attributes
Car
model : string
manufacturer : string
turnLeft()
turnRight()
driveStraight()
owns
Person
name : int
*
0..1
Composition relation: Car includes Engine, when Car is destroyed then also Engine is destroyed
Engine
rpm : int
displacement : int

UML Class diagram


Vehicle
Interface/Abstract class
Inheritance relation: Car extends Vehicle
Concrete classes: there's not always need to specify all the methods and attributes
Car
model : string
manufacturer : string
turnLeft()
turnRight()
driveStraight()
owns
Person
name : int
*
0..1
Composition relation: Car includes Engine, when Car is destroyed then also Engine is destroyed
Association relation:. Multiplicity of the association says that 0 or 1 Person own 0 or more Car
Engine
rpm : int
displacement : int

Classes relationships

- A relationship is a general term covering the specific types of logical connections found on class and object diagrams.
- An **association** represents a family of links. A binary association is normally represented as a line. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, etc.
- **Aggregation** is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. The contents of the container are not automatically destroyed when the container is.
  Represented with an hollow diamond shape on the containing class
- The **Composition** relationship is more "physical" than aggregation: it is used when attempting to represent real-world whole-part relationships, e.g. an engine is a part of a car.
  It is represented with a filled diamond on the containing class.



Technical documentation

- Sometimes reading code alone does not provide a full understanding of how something must be used or how it works: technical documentation is required, e.g. library manuals
- It's possible to create technical documentation from code comments using specialized tools like Doxygen (common in C++) or JavaDoc (common in Java)

Technical documentation

- Documentation tools require that programmers use specific tags in comments, parse code and generate documentation. An example of Doxygen comments:

```
/**
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param  Description of method's or function's input parameter
 * @param  ...
 * @return Description of the return value
 */
```

Technical documentation

- Documentation tools require that programmers use specific tags in comments, parse code and generate documentation. An example of Doxygen comments:

Note the double **

```
/**
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param  Description of method's or function's input parameter
 * @param  ...
 * @return Description of the return value
 */
```

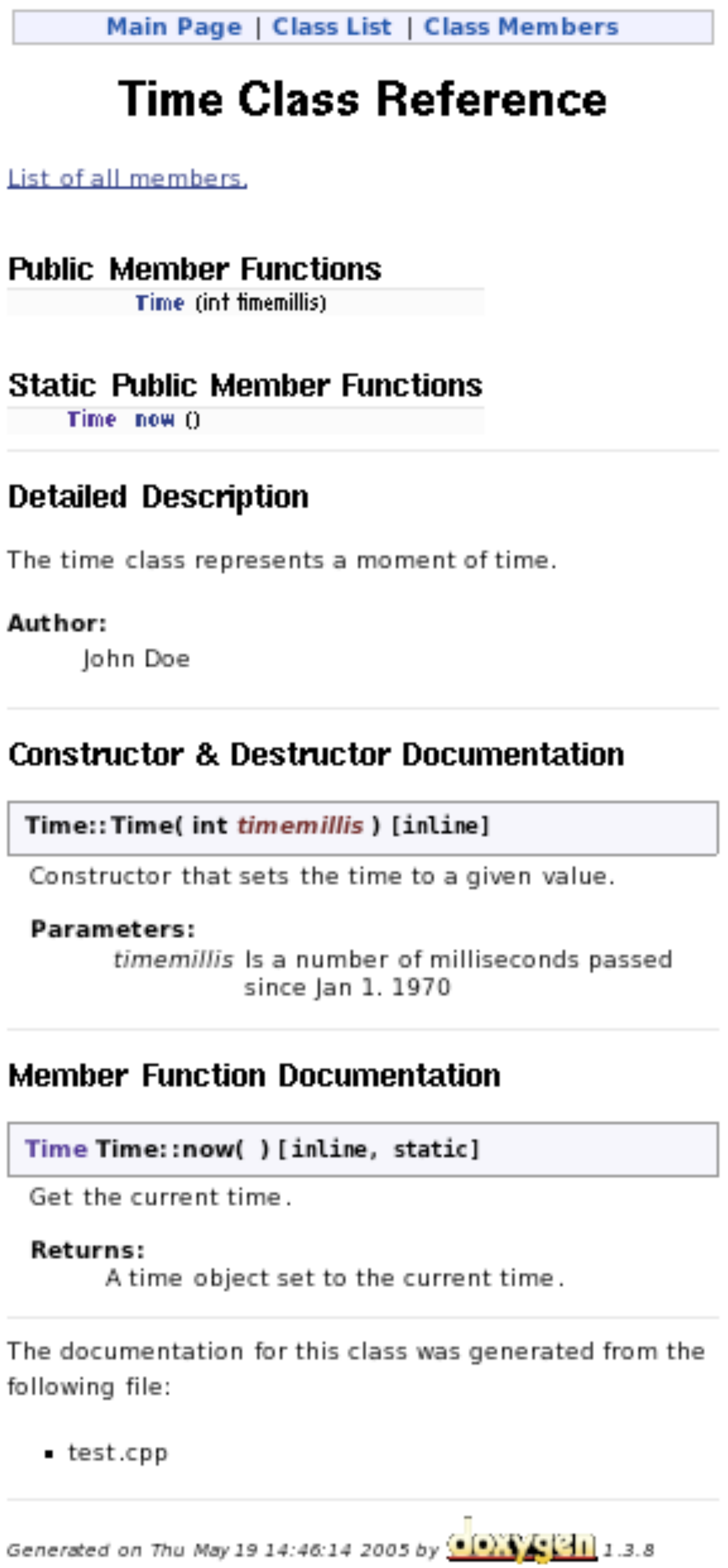
Doxygen example

```
/**
 * @file
 * @author  John Doe <jdoe@example.com>
 * @version 1.0
 *
 * @section LICENSE
 *
 * This program is free software; you can redistribute it
and/or
 * modify it under the terms of the GNU General Public
License as
 * published by the Free Software Foundation; either
version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be
useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty
of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU
 * General Public License for more details at
 * http://www.gnu.org/copyleft/gpl.html
 *
 * @section DESCRIPTION
 *
 * The time class represents a moment of time.
 */

class Time {

    public:

       /**
        * Constructor that sets the time to a given value.
        *
        * @param timemillis Number of milliseconds
        *        passed since Jan 1, 1970.
        */
       Time (int timemillis) {
           // the code
       }

       /**
        * Get the current time.
        *
        * @return A time object set to the current time.
        */
       static Time now () {
           // the code
       }
};
```

en example

```
/**
 * @file
 * @author
 * @version
 *
 * @section
 *
 * This prog                                   t
and/or
 * modify it
License as
 * publishec
version 2 of
 * the Licer
 *
 * This prog                                   be
useful, but
 * WITHOUT A                                   ty
of
 * MERCHANTA
See the GNU
 * General F
 * http://ww
 *
 * @section
 *
 * The time
 */

class Time {

    public:

        /**
```

```
                 * Constructor that sets the time to a given value.
                 *
                 * @param timemillis Number of milliseconds
                 *        passed since Jan 1, 1970.
                 */
                Time (int timemillis) {
                    // the code
                }

                /**
                 * Get the current time.
                 *
                 * @return A time object set to the current time.
                 */
                static Time now () {
                    // the code
                }
        };
```



Main Page | Class List | Class Members

Time Class Reference

List of all members.

Public Member Functions

Time (int timemillis)

Static Public Member Functions

Time now ()

Detailed Description

The time class represents a moment of time.

Author:
John Doe

Constructor & Destructor Documentation

Time::Time(int *timemillis*) [inline]

Constructor that sets the time to a given value.

Parameters:
timemillis Is a number of milliseconds passed since Jan 1. 1970

Member Function Documentation

Time Time::now( ) [inline, static]

Get the current time.

Returns:
A time object set to the current time.

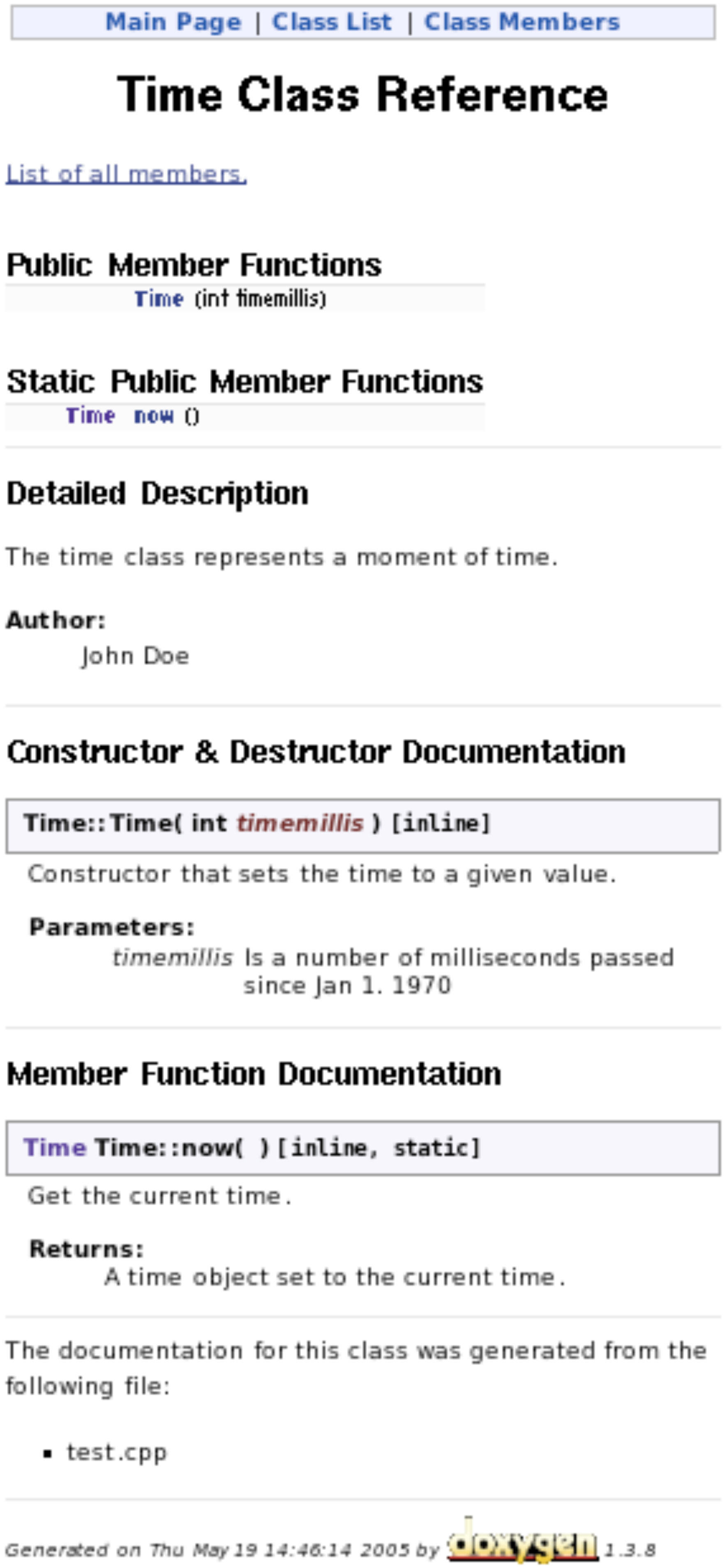
The documentation for this class was generated from the following file:

- test.cpp

Generated on Thu May 19 14:46:14 2005 by doxygen 1.3.8

en example

Doxygen parses the comments and produces different types of documents (HTML, Word, LaTeX), with all the required indexes



Main Page | Class List | Class Members

Time Class Reference

List of all members.

Public Member Functions

Time (int timemillis)

Static Public Member Functions

Time now ()

Detailed Description

The time class represents a moment of time.

Author:
John Doe

Constructor & Destructor Documentation

Time::Time(int *timemillis*) [inline]

Constructor that sets the time to a given value.

Parameters:
timemillis Is a number of milliseconds passed since Jan 1. 1970

Member Function Documentation

Time Time::now( ) [inline, static]

Get the current time.

Returns:
A time object set to the current time.

The documentation for this class was generated from the following file:

- test.cpp

Generated on Thu May 19 14:46:14 2005 by doxygen 1.3.8

```
/**
 * @file
 * @author
 * @version
 *
 * @section
 *
 * This prog                                                     t
and/or
 * modify it
License as
 * publishec
version 2 of
 * the Licer
 *
 * This prog                                                     be
useful, but
 * WITHOUT A                                                     ty
of
 * MERCHANTA
See the GNU
 * General F
 * http://ww
 *
 * @section
 *
 * The time
 */

class Time {

    public:

        /**
```

```
             * Constructor that sets the time to a given value.
             *
             * @param timemillis Number of milliseconds
             *        passed since Jan 1, 1970.
             */
            Time (int timemillis) {
                // the code
            }

            /**
             * Get the current time.
             *
             * @return A time object set to the current time.
             */
            static Time now () {
                // the code
            }
        };
```

An exercise in software engineering

How to start ?

- Discuss with client, to get a description of the desired system
- Condense it in a feature list
- Create use cases
- Identify the most important aspects, and focus on them
- Apply OO principles to add flexibility
  - aim for a maintainable and reusable design

An example

- Client: create a Dungeon&Dragons/Rogue-like videogame
- Discussing with the client we get this list of features:
  - Players can play using different fighting characters
  - Each character has different specialities
  - Each character may change different weapons while game progresses
  - Characters move within a map
  - A map is composed by different tiles
  - The game allows to buy add-ons like maps and characters

Most relevant elements

- From the feature list we find out what is most important by asking ourselves if a part:

1. is essential to the system: e.g. the game could exist without a "character" ?
2. has a clear meaning. If not spend time to figure it out.
3. you know how to do it. E.g. how to manage the movement of characters in the map.

Most relevant elements

- From the feature list we find out what is most important by asking ourselves if a part:

1. is essential to the system: e.g. the game could exist without a "character" ?

Considering the previous list of features it is clear that a "character" is a significant element, while the add-ons are less important (though there's need to figure out the exact meaning of it)

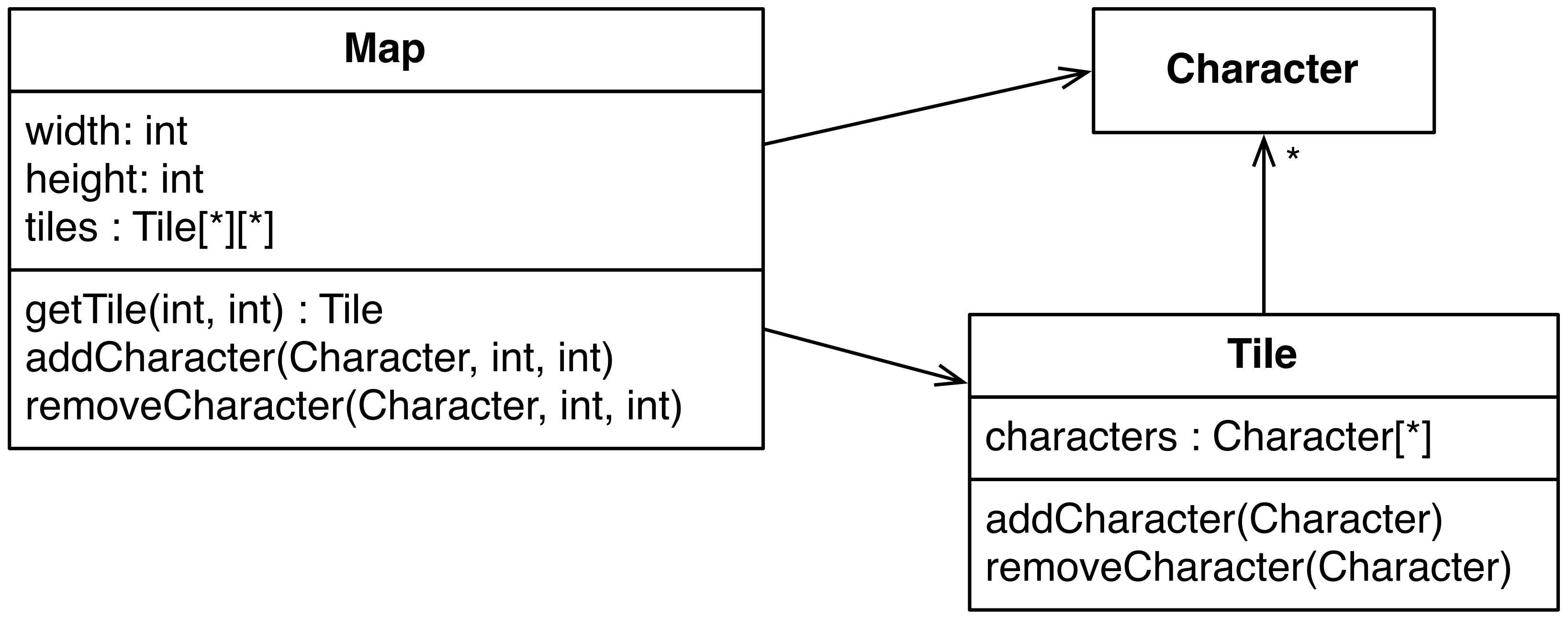
Most relevant elements

The key features from our list are:
- The map - essential
- The character - essential, check meaning
- The movement coordination - check meaning and how to do it

Considering the previous list of features it is clear that a “character” is a significant element, while the add-ons are less important (though there’s need to figure out the exact meaning of it)

Designing the objects

- A map has a certain size
- A map has different tiles
- It’s possible to add/remove characters on tiles

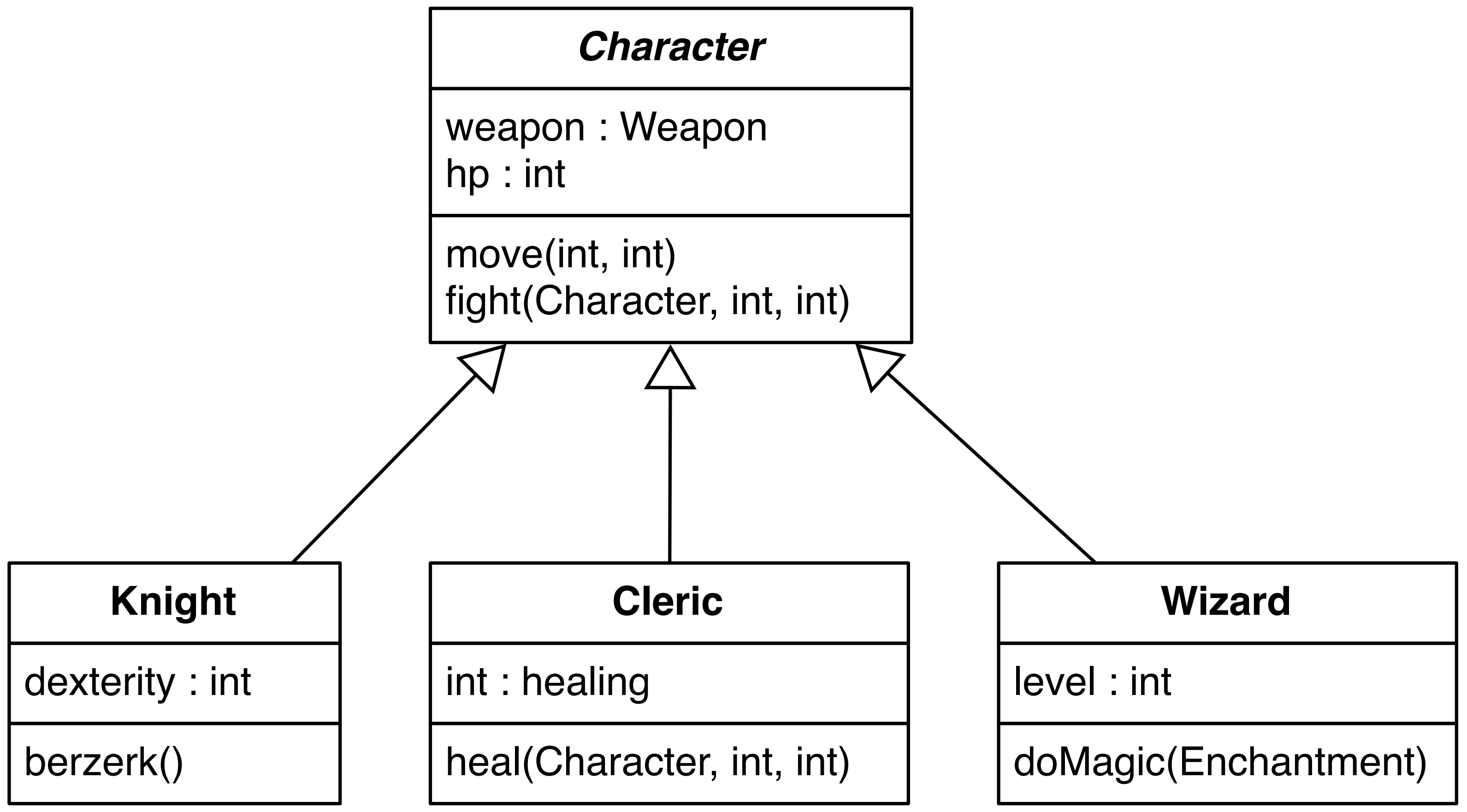

Map
width: int
height: int
tiles : Tile[*][*]
getTile(int, int) : Tile
addCharacter(Character, int, int)
removeCharacter(Character, int, int)
Character
*
Tile
characters : Character[*]
addCharacter(Character)
removeCharacter(Character)

Designing the objects - 2

- Build on what you already have... let's continue to expand on character...
    - There are different types of characters, with specific actions...
    - ...but they have common attributes
    - A super class holds commonalities, sub-classes manage specific functions

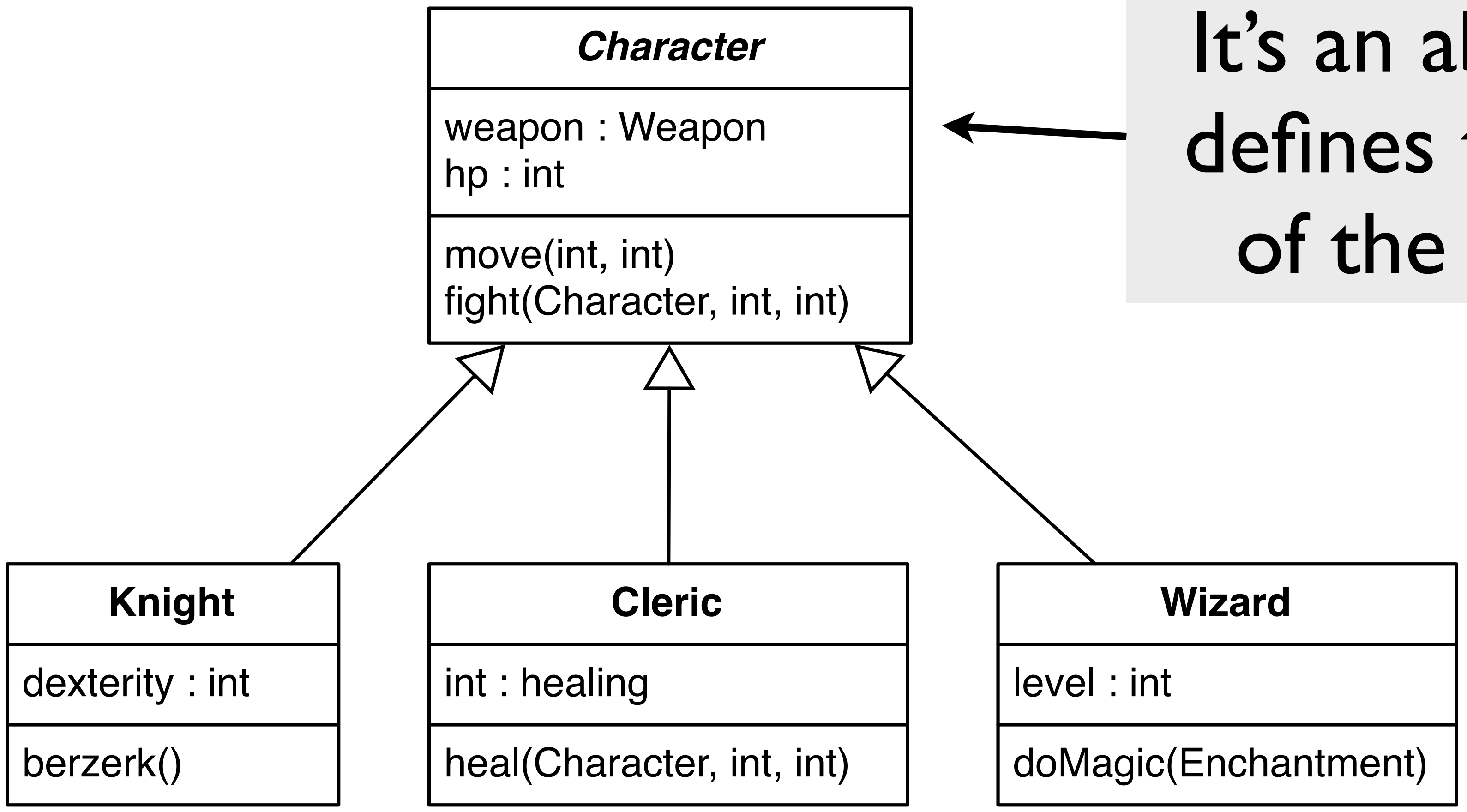
Designing the objects - 3

- Develop the Character class seen before
- A possible solution is:



Designing the objects - 3

- Develop the Character class seen before
- A possible solution is:

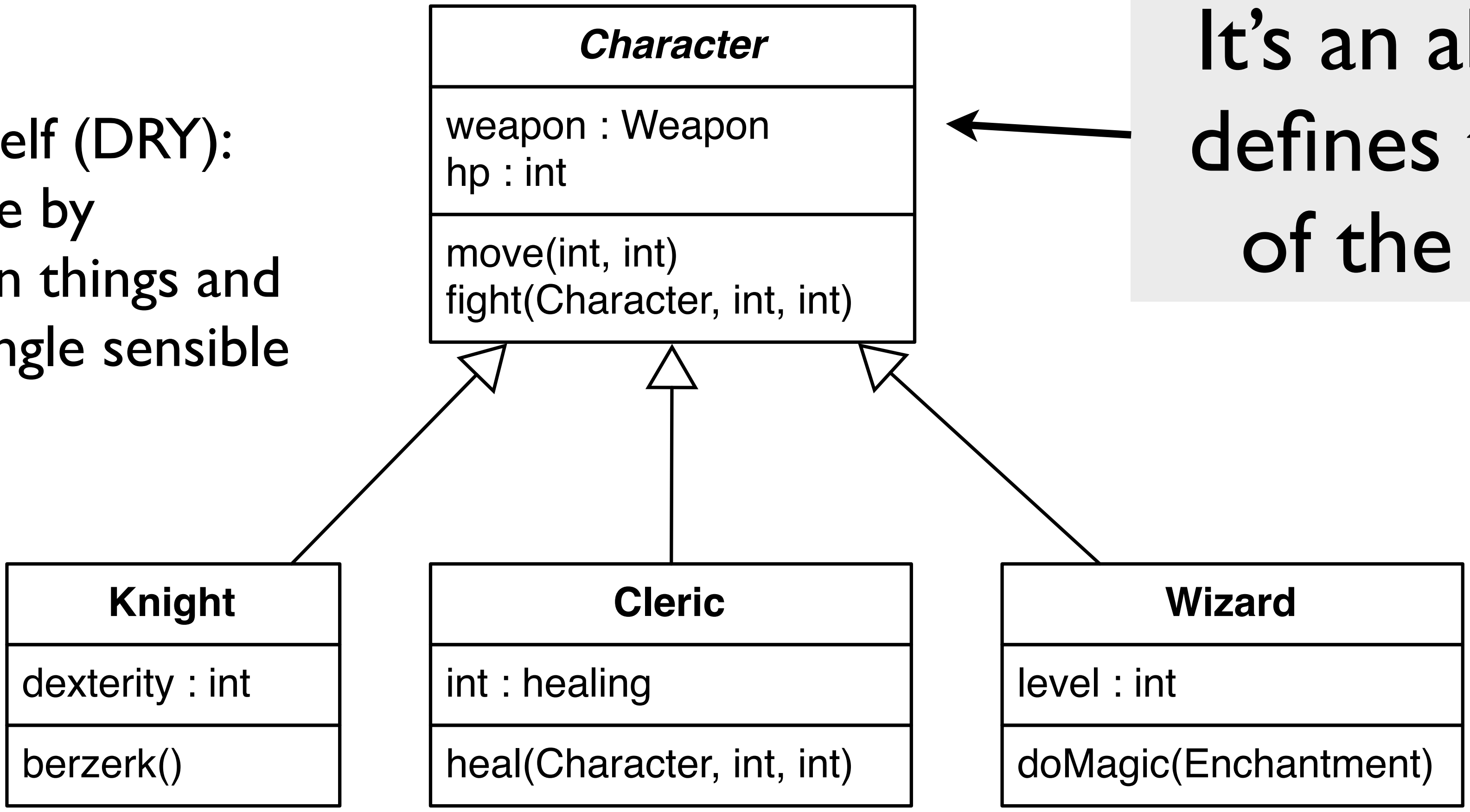


It’s an abstract class: defines the interface of the sub classes

Designing the objects - 3

- Develop the Character class seen before
- A possible solution is:

Don't Repeat Yourself (DRY): avoid duplicate code by abstracting common things and placing them in a single sensible location



It's an abstract class: defines the interface of the sub classes

Designing the objects - 4

- Some useful guidelines:
  - Open-Closed Principle (OCP): classes should be open for extension and closed for modification
    - move() is defined in the base class and doesn’t change. If a new character will need to change it will just override it (so think in advance and make it virtual)
  - Don’t Repeat Yourself (DRY): avoid duplicate code by abstracting common things and placing them in a single sensible location
    - hit points are common to all characters, code to manage them is in the super class
  - Single Responsibility Principle (SRP): every object should have just one responsibility and all services should focus on it
  - Liskov Substitution Principle: a subtype must be substitutable for their base type
  - Delegation: hand over the responsibility for a particular task to another class or method

Liskov substitution principle

- Let’s suppose we want to add aerial fighting with dragons: we need a 3D map.
  - Extending the base class makes the 3D map to inherit all the methods that work on 2D coordinates... but these methods are of no use. LSP shows us that a 3D map is NOT a 2D map !
  - Instead of inheriting consider delegating the management of each layer of a 3D map to a 2D map

Liskov substitution principle

- Let's suppose we want to add aerial fighting with dragons: we need a 3D map.
  - Extending the base class makes the 3D map to inherit all the methods that work
    ut these methods
    ws us that a 3D map

Use delegation when you want to use the functionality of another class without changing it's behaviour

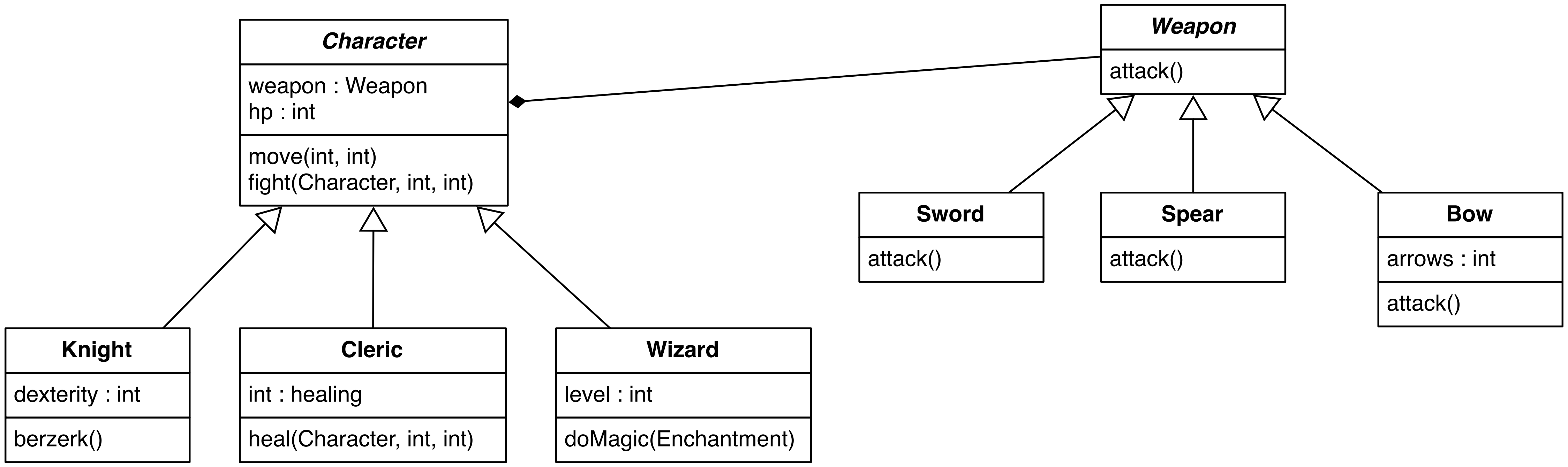
  - Instead of inheriting consider delegating the management of each layer of a 3D map to a 2D map

When to use composition ?

- Use composition to assemble behaviours of different classes
- Composition allows to use behaviour from a family of other classes, changing their behaviour at runtime
    - if the object that is composed of other objects is destroyed so are the behaviours
    - if it’s not destroyed then it’s called aggregation

Composition: an example

- When the Character dies we destroy the Weapon



Inheritance et al.

- In OO inheritance is just one of the solutions we can use to design good software. We have also:
- Delegation
- Composition
- Aggregation

Example of bad inheritance

- Let us suppose we want to have also different races for our characters (e.g. Elf, Orc, Human, …): using multiple inheritance would lead to an “explosion” of classes (e.g. HumanKnight, ElfKnight, etc.)
  - Probably composition or adding some attributes to Knight/Mage/etc. is enough
  - Exercise: can we use compositional for the special abilities of our characters, without having to use RTTI ?

Some C++ style suggestions

from Bjarne Stroustrup*

*original interview (http://www.artima.com/intv/goldilocks.html)

Avoid Object-Orientaphilia

- Do NOT do everything by creating a class as part of a class hierarchy with lots of virtual functions:
  "You can program with a lot of free-standing classes. If I want a complex number, I write a complex number. It doesn't have any virtual functions. It's not meant for derivation."
- Use inheritance only when a class hierarchy makes sense from the point of view of your application, from your requirements.

Classes Should Enforce Invariants

- A class invariant is an **invariant** used to constrain objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object: an invariant allows you to say when the object's representation is good and when it isn't.
- Rule of thumb: you should have a real class with an interface and a hidden representation if and only if you can consider an invariant for the class.

Classes Should Enforce Invariants

- A class invariant is an **invariant** used to constrain objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object: an invariant allows you to say when the object's representation is good and when it isn't.
- Rule of thumb: yo an interface and a only if you can co class.

a condition that can be relied upon to be true during execution of a program, or during some portion of it

Classes Should Enforce Invariants - 2

- You can write the interfaces so that they maintain that invariant. Operations that don't need to mess with the representation are better done outside the class. This results in a clean, small interface that you can understand and maintain.
- The invariant is a relationship between different pieces of data in the class. If every data can have any value, then it doesn't make much sense to have a class.

Classes Should Enforce Invariants - 3

- Example: a data structure containing a name and address in which any string is a good name and address then should be implemented as struct... no private attributes and getter/setter or base classes with virtual methods.
- But... if the representation may change often or there's need to have different representations of the object then use the class.
- The constructor establishes the environment for the member functions to operate in: it establishes the invariant.

Designing Simple Interfaces

- A method that is using data but not defending invariant may not need to be in the class.
  Example: operations that need direct access to representations should be in class.
- In a *Date* class changing day/month/year should be members, instead a function that finds the next Sunday given a date could be a function built in a supporting library.

Class design suggestions

OOP key concepts and class design

- Abstraction - responsibilities (interface) is different from implementation
- A class provides some services, takes on some responsibilities, that are defined by the public interface. How it works inside doesn't matter. Distinguish between interface and implementation.
- Client should be able to use just the public interface, and not care about the implementation.

OOP key concepts and class design

- Encapsulation - guarantee responsibilities by protecting implementation from interference
- Developer of a class can guarantee behavior of a class only if the internals are protected from outside interference. Specifying private access for the internals puts a wall around the internals, making a clear distinction between which code the class developer is responsible for.

Some design principles

- Design a class by choosing a clear set of responsibilities
- Make classes responsible for working with their own data.
- Domain classes should be based on actual domain objects.
- What kinds of objects are in the domain?
  - Which classes -
- What characterizes each domain object?
  - Members -
- How are different kinds of objects related to each other?
  - Inclusion versus association -
    - Part-of relation versus "using" or "interacts with"
  - Relative lifetimes -
    - Do they exist independently of each other?

Red flags

- If class responsibilities can’t be made clear, then OOP might not be a good solution
    - Lots of problems work better in procedural programming than in OOP, so there is no need to force everything into the OO paradigm. OO is no silver bullet.
- Beware of classes that do nothing more than a C struct.
    - Is it really a “Plain Old Data” object, like C struct, or did you overlook something? If it is a simple bundle of data, define it as a simple struct.

Red flags - cont.

- Avoid heavy-weight, bloated, or “God” classes - prefer clear limited responsibilities.
    - If a class does everything, it is probably a bad design. Either you have combined things that should be delegated to derived classes or peer classes, or you have misunderstood the domain.
- Put in the public interface only the functions that clients can meaningfully use.
    - Reserve the rest for private helpers.
    - Resist the temptation to provide getters/setters for everything.

Design principles for methods

- Make member functions const if they do not modify the logical state of the object.
- Make a class fully responsible for initializing itself with constructor functions.
  - It is error-prone and bad design if the client has to "stuff" initial data into the object. Take care that all member variables get a good initial value.
  - Only supply these where necessary - if the member variable is a class type, the compiler will call its default constructor for you.
  - Do not write constructors, assignment operators, or destructors when the compiler- supplied ones will work correctly. Unnecessary code is simply places for bugs to hide, especially when revisions are made!

Methods’ responsibilities

- Constructor methods allow a class to be responsible for its initialization.
- Destructor methods allow a class to be responsible for cleaning up after itself.
- Copy/Move constructor and assignment operator functions allow a class to be responsible for how it is copied or its data moved.

Methods’ responsibilities

- Constructor methods allow a class to be responsible for its initialization.
- Destructor methods allow a class to be responsible for cleaning up after itself.
- Copy/Move constructor and assignment operator functions allow a class to be responsible for how it is copied or its data moved.

More on “Move” constructors in next lectures...

Credits

- These slides are (partly) based on the material of:
  - Prof. David Kieras, Univ. of Michigan