

7. Implementation Phase

7.1 Architecture Diagrams

7.2 OO Languages: Java

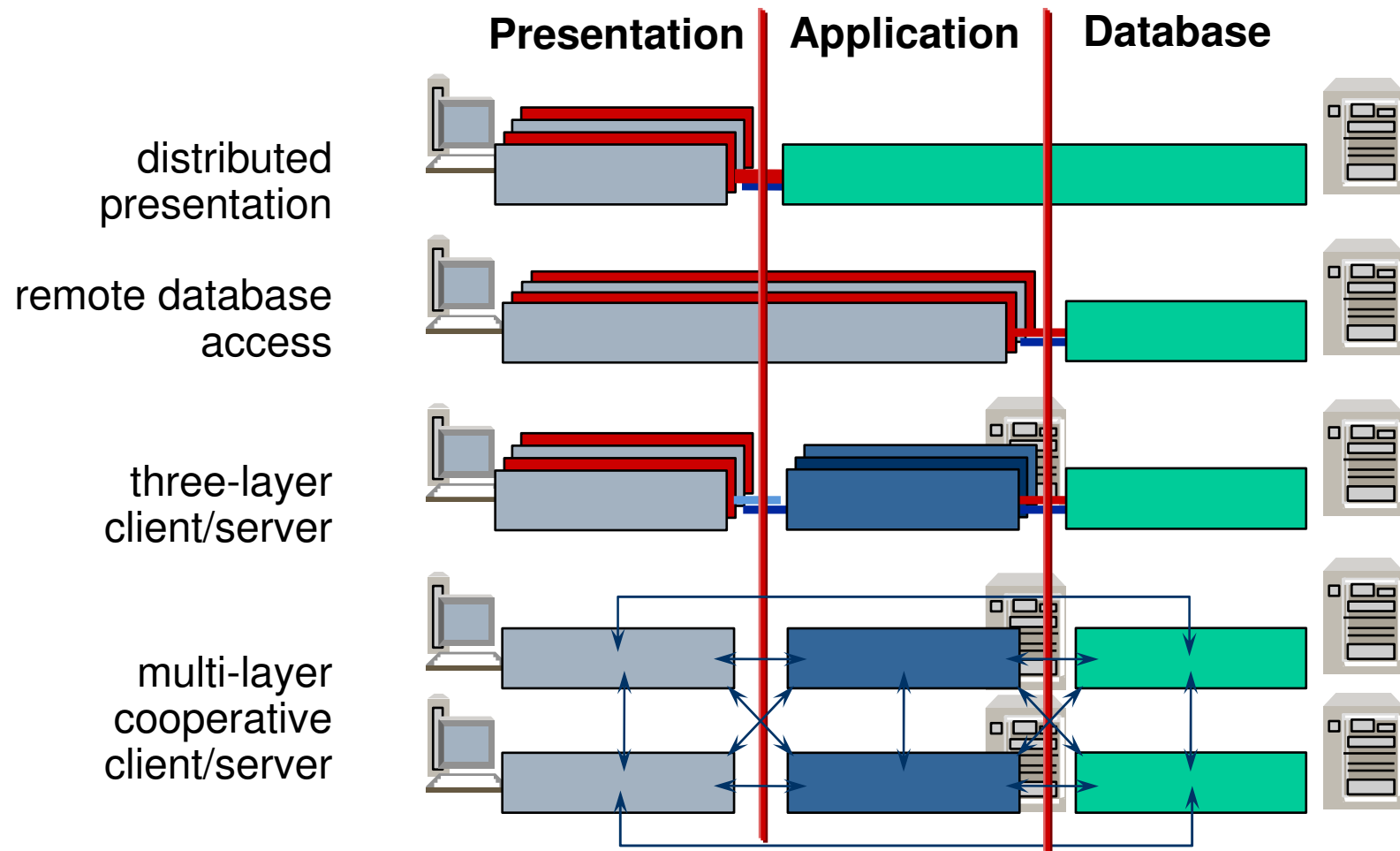
7.3 Constraint Languages: OCL

Architecture Design Models

- An **architecture model** (structure model) is a model of a data processing **system** describing the **static** structure of the components of a system.
- For large systems the architecture has to be modeled explicitly.
- Examples of Architectural Models:
 - network topology (hardware)
 - function tree (software)
 - deployment diagram, package diagram (software)
 - module diagram (hard/software)
 - organization chart (organigram) in a company model

Example: SAP R/3

- Flexible three-tier *client/server*-architecture

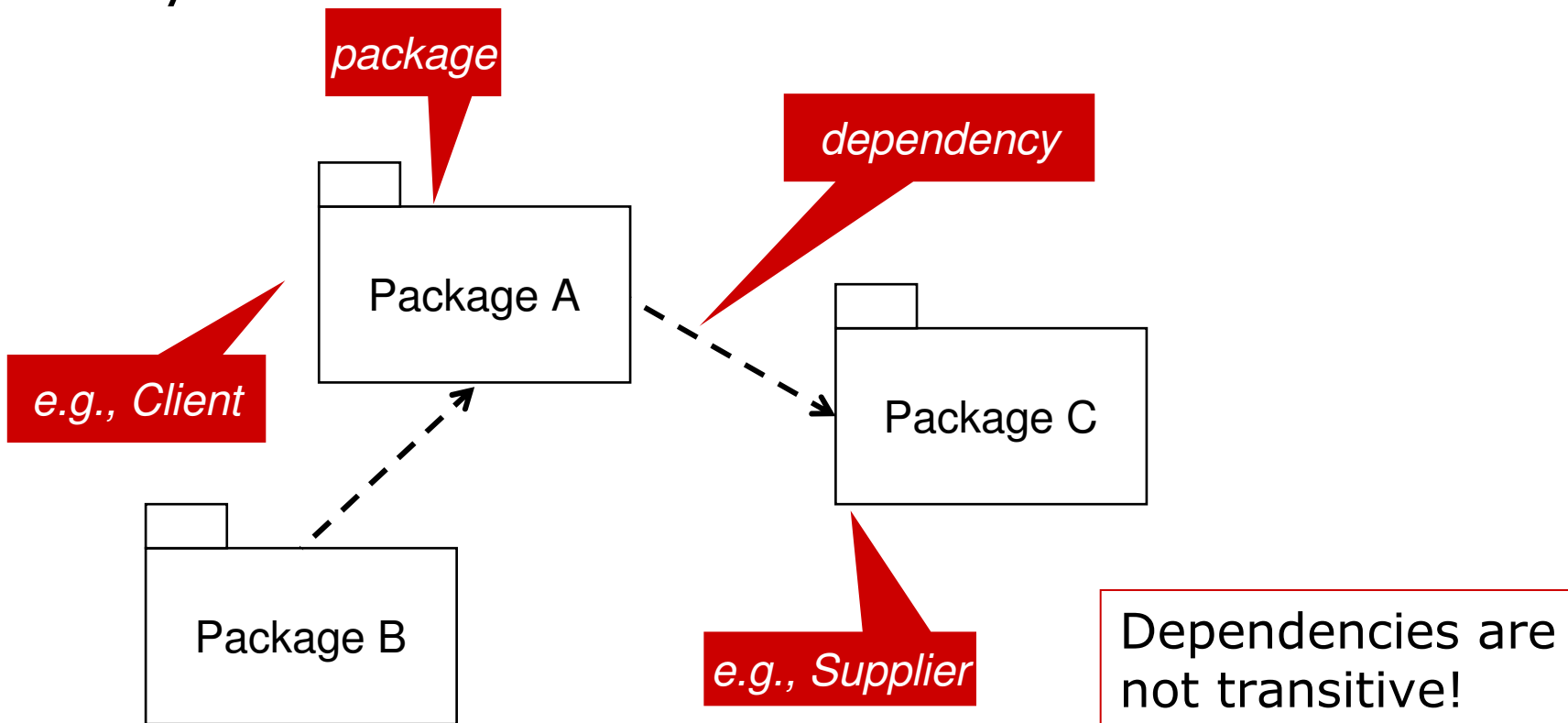


Specification Models in UML

- Architecture diagrams have the following elements:
 - Package Diagrams:
 - Classes, Interfaces & Packages
 - Nested Packages
 - Dependencies
 - Deployment Diagrams:
 - Nodes & Components
- Specification models for architecture design:
 - **logical** structures: packages with package diagrams; subsystems; interfaces
 - **physical** structures: components in component diagrams; nodes; deployment diagrams

Package Diagram

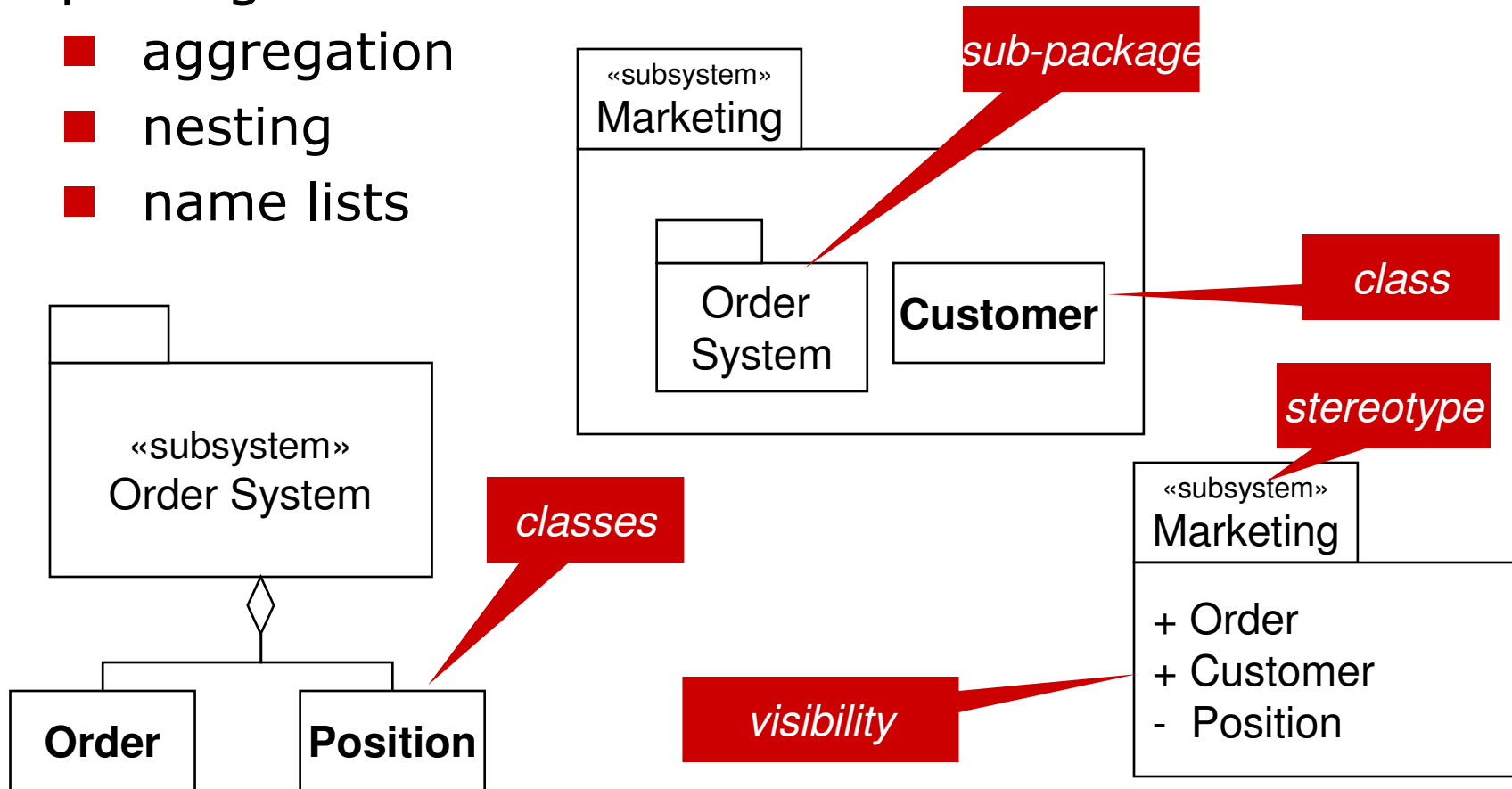
- A **package diagram** shows the coherence (*dependencies*) between different *packages* of the system.



Package Contents

- Structuring alternatives for the representation of package **content** are:

- aggregation
- nesting
- name lists



Subsystems

- Problem: How to break down a large system into smaller systems?

Functional Decomposition

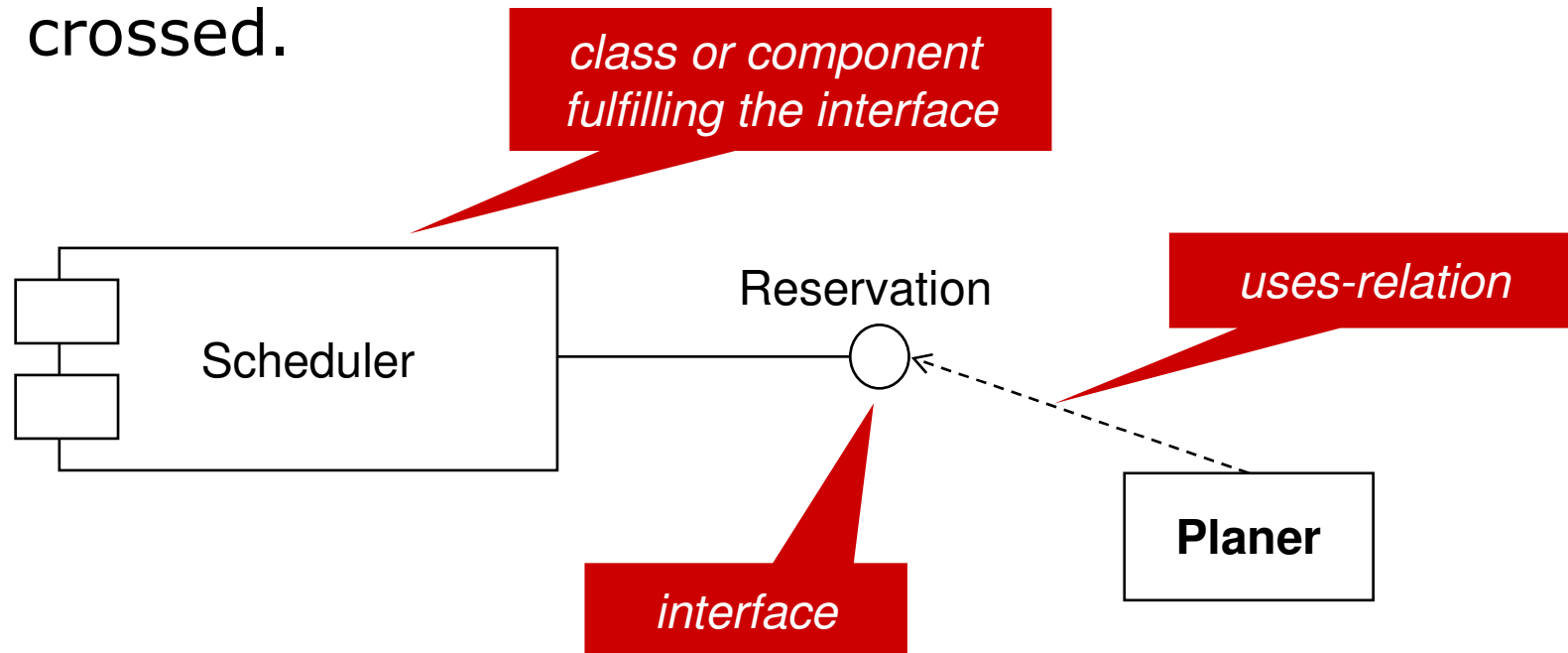
Map the total system on functions and sub-functions, starting with the use case.

OO Packages

- Collect classes into subsystems.
- Build layers of subsystems.
- Principles:
 - **Abstraction:** Concentrate on essentials.
 - **Locality:** Group together related components (data and algorithm).
 - **Hiding:** Restrict the visibility of details, so that only those parts of a system that need to know the details have access to them.

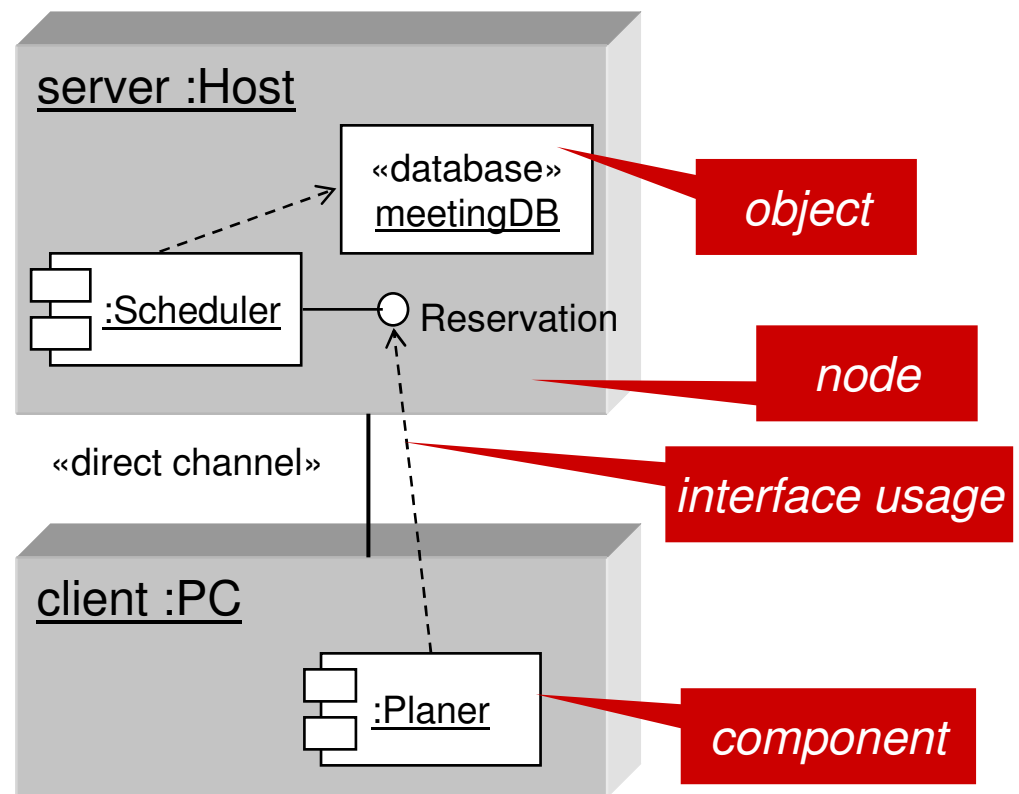
Interfaces in Architecture

- ❑ Interfaces let you specify the outward appearance of components.
- ❑ They are important in architecture diagrams because they show how system boundaries are crossed.



Deployment Diagram

- ❑ A **deployment diagram** shows the configuration of a *node* (piece of “hardware”) at runtime as well as the *components* (instances) and *objects* residing on it.
- ❑ Components not existing as runtime objects (instance) should appear in component diagrams only .



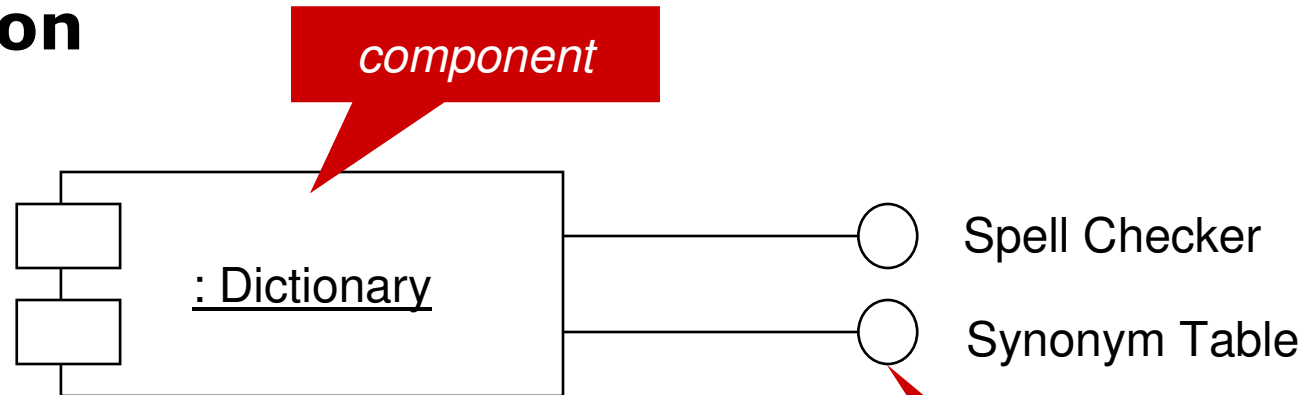
Components (1)

- A **component** is a physical, replaceable part of a system containing an implementation which realizes a set of interfaces
- Components have two aspects:
 - **Code:** A component consists of code, components can contain or use components.
 - **Identity:** A component can have identity and state represented by objects. An object which wants to use services of the component must use the appropriate instance.
(e.g., *Bean* reference, *DLL handle*, *CORBA-IOR*,...)

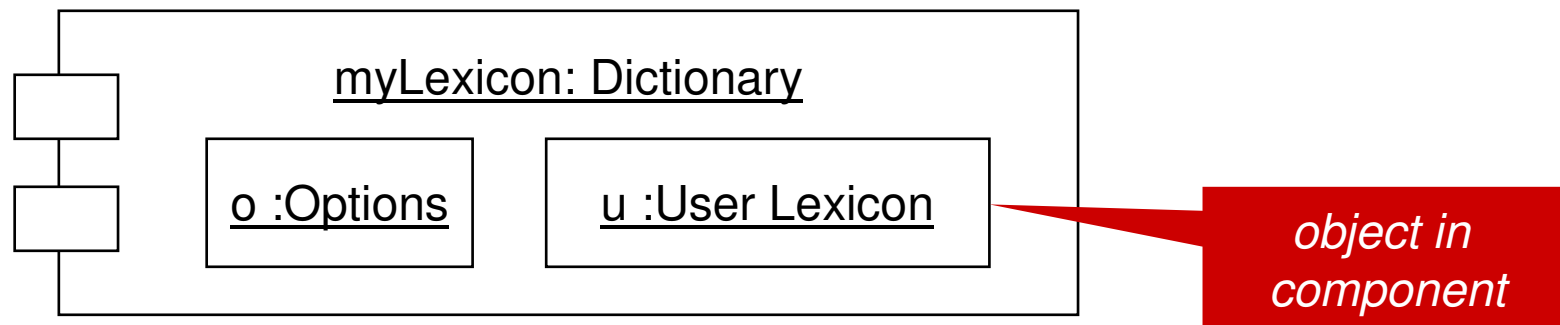
Example: Spelling checker as component:
- identity and state by user dictionary,
- different versions and languages.

Components (2)

□ Notation

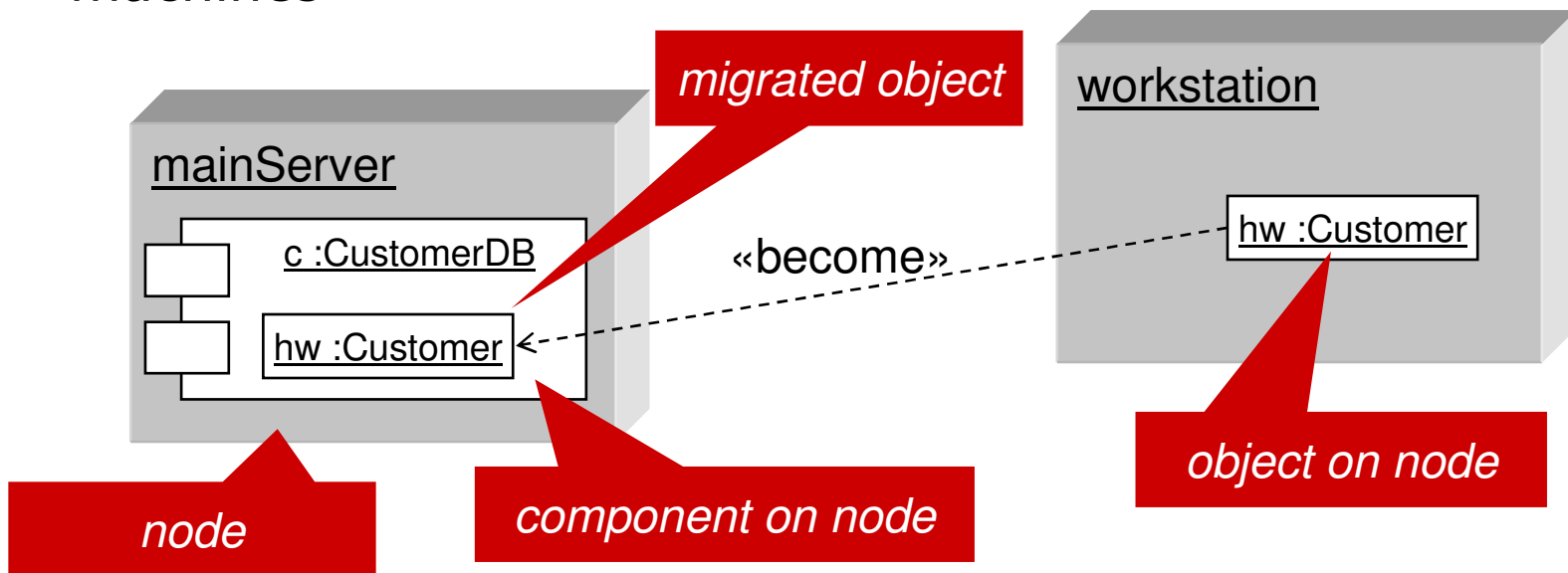


□ Component with identity



Node

- A **node** is a physical entity existing at runtime
 - represents processing resources
 - has storage and computation capacity
 - On a node, objects and components can be life
- Nodes can be computers, humans, or other devices or machines



Discussion

- Components ...
 - are conceptually and functionally larger than a classes.
 - unite behavior and collaboration of a group of classes (see collaboration diagrams)
 - are independent of other components but usually collaborate with them
 - are replaceable with other realizations of the same interface
 - are fundamental building blocks of **component-based architectures**
 - can be built recursively. A system can be a component on the next higher inspection level.

Languages

7.2 Object-Oriented Programming Language: Java

7.3 Constraint Language: OCL

Language Part

- **L**TODD - languages are used in all phases to describe the developed artifacts:
 - natural languages (non-formal, e.g. English)
 - all over, but
 - mainly used in the analysis phase
 - various kinds of diagrams (semi-formal, e.g. UML)
 - analysis & design
 - implementation (reading the created diagrams)
 - programming languages (fully formal, e.g. Java)
 - implementation
 - testing
 - constraint languages (mostly formal, e.g. OCL)
 - design and implementation

Languages

- Natural languages:
 - You speak (several of) them
 - Covered in High School
- Diagram languages (e.g., UML):
 - Covered in OOAD
- This leaves open:
 - Programming languages
 - Constraint languages, already done Alloy

Your Cup of Java (its Grounds)



- ❑ Not a full Java CUP (“Course Upon Programming”).
- ❑ We just list the basic object-oriented concepts, i.e., how to
 - declare classes, interfaces, methods, data members
 - create and manipulate instances
 - take advantage of object-oriented features like
 - ❑ inheritance
 - ❑ polymorphism, dynamic binding
 - ❑ overriding
 - exception handling (not really OO, but based on objects)
- ❑ All this is mainly syntax and thus not very difficult.
- ❑ The OO concepts were explained in the first two lectures.

Declaring Classes

- Basic syntax:

```
<visibility> <modifiers> class <classname> extends  
    <superclass-name> implements <list-of-interfaces>  
    { <class-body> }
```

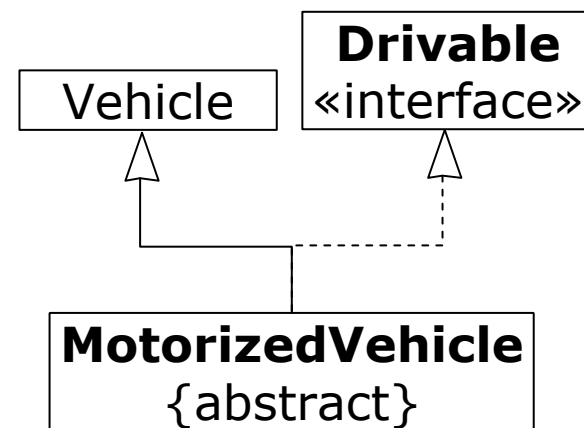
- For example:

```
public abstract class MotorizedVehicle  
    extends Vehicle implements Drivable { ... /*...*/ }
```

- Most elements can be omitted; mandatory is ...

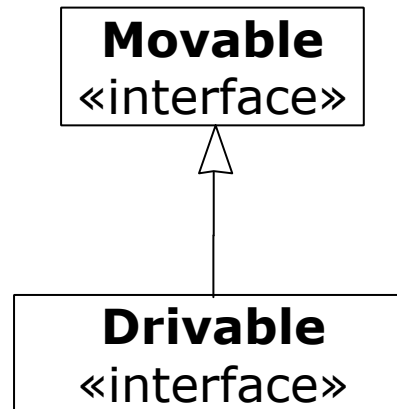
- **class**
- <classname>
- { <class-body> }

Note: Anything enclosed in `/*` and `*/` in Java is a comment. Everything on a line after `//` is a comment as well.



Declaring Interfaces

- Very similar to classes
 - not all visibilities make sense
 - there is no **implements** clause
- Syntax:
`<visibility> <modifiers> interface <classname>
 extends <superinterface-name> { <interface-body> }`
- For example:
`public interface Drivable extends Movable { .../*...*/ }`



Methods

- ❑ Declared in classes or interfaces
- ❑ Implemented in classes
- ❑ Basic Syntax:
 <visibility> *<modifiers>* *<return-type>*
 <method-name>(*<parameter-list>*)
 throws *<exception-list>* *<method-body>*
- ❑ Most of these terms are optional, here is an example:
`public static User getLoggedInUser() {.../* ... */}`
- ❑ Abstract methods and method declarations in interfaces do not have a body.

Overriding

- ❑ Re-implementation of methods in subclasses
- ❑ All you have to do is give a method with the same signature

```
class Person {  
    String name;  
    public String getName() {  
        return name;  
    }  
}  
class King extends Person {  
    public String getName() {  
        return "King " + super.getName();  
    }  
}
```

Note: The return type can be more specific (covariant) Java 5, parameter types invariant or new method.

Invocation of the overridden method

Larger Example

Specialization

Class

```
public class Employee extends Person {
```

```
    private double salary;
```

```
    public Employee() {
```

```
        setSalary(1000);
```

```
        // this.salary = 1000;
```

```
    }
```

```
    public void setSalary(double newSalary) {
```

```
        salary = newSalary;
```

```
    }
```

```
    public void raiseSalary(double percent) {
```

```
        double factor = percent / 100;
```

```
        setSalary(getSalary() * (1 + factor));
```

```
    }
```

```
    public double getSalary() {
```

```
        return salary;
```

```
    }
```

```
}
```

Data member

Constructor

Method

local variable

Return type

Instances

- To create a new instance of a class, use the **new** operator:

```
new Person("John", 42);
```

- new instance is created
- constructor is called

- The new instance is usually bound to a variable:

```
Person p = new Person("John", 42);
```

- you need the variable to later access the instance
- can store instances in lists etc.

Using Instances

❑ Invoking methods:

```
employee.raiseSalary(3.0);
```

- uses late binding to call the right implementation of the method

❑ Accessing fields:

```
person.name = "Maria";
```

- fields usually are not public

Bad practice! Just an example, use `setName("Maria")` in real life.

❑ In Java you don't need to destroy instances.

The runtime system does this for you:

- Garbage Collection
- Reachability

Control Flow (1)

- You can actually also write algorithms in Java ...
 - loops: **for**, **while**

```
List<String> list = ...  
for (String s : list) {  
    System.out.println(s);  
}
```

Prints lists

```
int i = 1; int j = 1; int k = 0;  
while (j < 100) {  
    k = j; j += i; i = k;  
    System.out.print(" " + k);  
}
```

Prints Fibonacci
numbers < 100

Control Flow (2)

- conditionals: **if**, **switch**

```
Person p = ...  
if (("John".equals(p.getFName())) &&  
    ("Edwards".equals(p.getLName()))) {  
    System.out.println("Found John Edwards");  
} else {  
    System.out.println("This is not him");  
}
```

```
int i = Random.nextInt(10);  
int k = 0;  
switch (i) {  
    case 1: k = 7; break;  
    case 2: k = 5; break;  
    case 5: k = 9; break;  
    default: k = -1;  
}
```

Error Handling: The Old Way

□ Good Old Pascal:

```
{ $I- }
```

```
reset (F);
```

```
{ $I+ }
```

```
if IOresult<>0 then
```

```
begin
```

```
    writeln('Error encountered in reading file.');
```

```
    halt;
```

```
end;
```

```
{ use the file here }
```

Reopen file for writing

Check whether error occurred

Theoretically you have to execute
if IOresult<>0 then ...
after every operation on the file.

□ Manual error handling

□ Control flow becomes much more complicated

□ Nobody is forced to do error handling

- Programs just crash when an error occurs that the programmer did not anticipate.

Exceptions: The New Way

□ Java:

```
try {  
    InputStream in = new FileInputStream("a.txt");  
    // ... use the file here  
} catch (IOException e) {  
    System.out.println("Error");  
}
```

□ You are forced to do exception handling at some point in your program.

- Methods can defer exception to the caller:

```
public void test() throws IOException { ... }
```

- Have to declare which exceptions can be thrown during a methods execution

□ Control flow does not become cluttered because you handle the exception where you can deal with it.

Exception Handling (2)

- Making the caller responsible for handling:

```
class Reading {  
    private byte[] buffer;  
    private InputStream in;  
    private void readBytes() throws IOException {  
        in.read(buffer, ...);  
    }  
    public Person getData() {  
        Person p;  
        try {  
            readBytes();  
            p = Person.convertData(buffer);  
        } catch (IOException e) {  
            showErrorMessage("Could not read file");  
        }  
        return p;  
    }  
}
```

The diagram illustrates the flow of control and exception handling. The **Callee** is the `readBytes()` method, which is responsible for performing the I/O operation and throwing an `IOException`. The **Caller** is the `getData()` method, which calls `readBytes()` and is responsible for handling the exception by catching it and displaying an error message.

Constraint Language

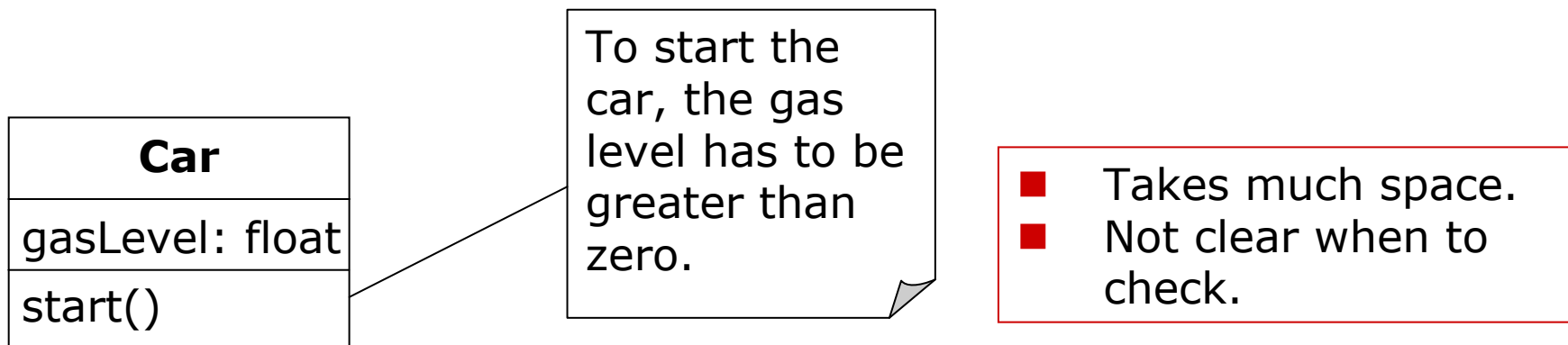
Algorithmic Programs
versus
Formal Logic Declarations

Constraints

- You can express constraints ...
 - in natural language (e.g., as notes in UML diagrams)
 - in a formal language (e.g., in OCL, Object Constraint Language, Alloy)
- Reasons for having formal constraints on models:
 - can express semantics beyond the expressiveness of diagrams
 - are formal and can be used to verify system's properties
 - for test during development
 - in disputes with customer
 - can generate working code from constraints (to some extent; issue still subject to research)

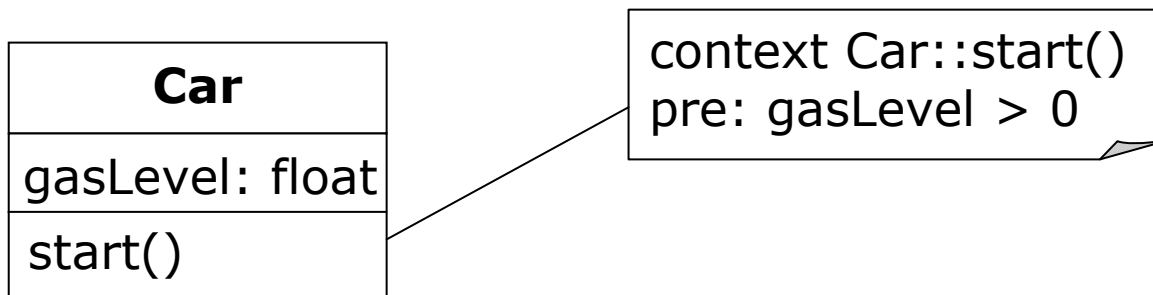
Constraint Example

- You can specify constraints in natural language and attach them as notes in UML diagrams
- Natural language has disadvantages:
 - Even simple constraints take much space.
 - Complex constraints are hard to understand.
 - Complex constraints have to be formulated very carefully in order not to be ambiguous.



Formal Constraint Language

- ❑ Can formulate the constraints more concisely
- ❑ Less ambiguity



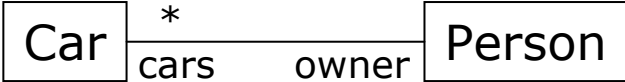
Object Constraint Language

- ❑ OCL is part of UML
- ❑ Constraints are given as logical expressions which evaluate to true or false
- ❑ Kinds of constraints:
 - pre-condition: is true before a method execution
 - post-condition: is true after a method execution
 - invariant: is always true, stays true
 - guard: must be true before state transition fires
- ❑ OCL has no side effects (no updates)

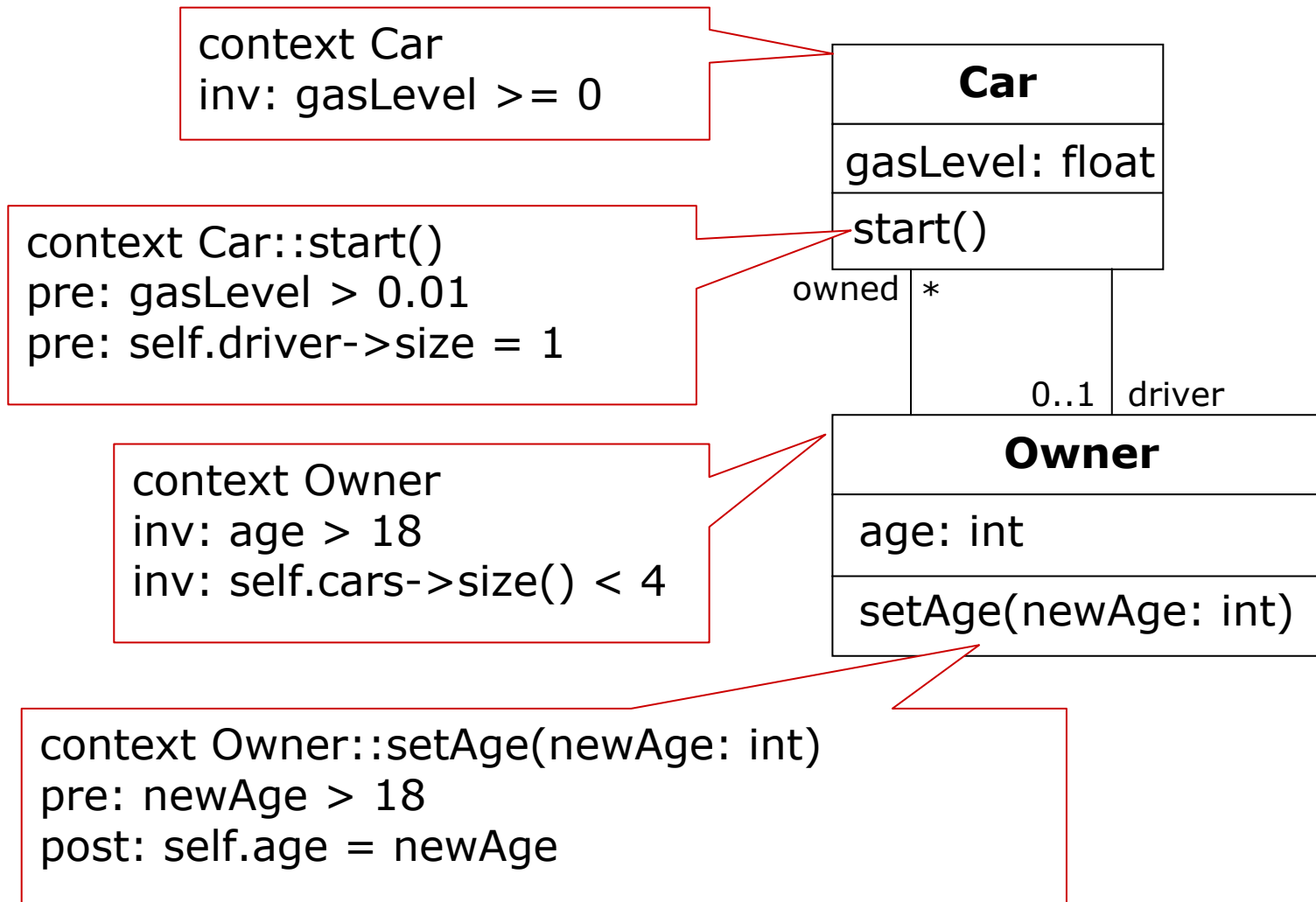
OCL Basics

- General syntax:
 - context** *<name>*
 - pre:** *<expression>*
 - post:** *<expression>*
 - inv:** *<expression>*
- context: specifies the entity under consideration.
- pre: expression that must hold before execution
- post: expression that must hold after execution
- inv: expression that stays true
- Can repeat or omit any of pre/post/inv
- There are more, but these are the most important ones. Also see the specification [1].

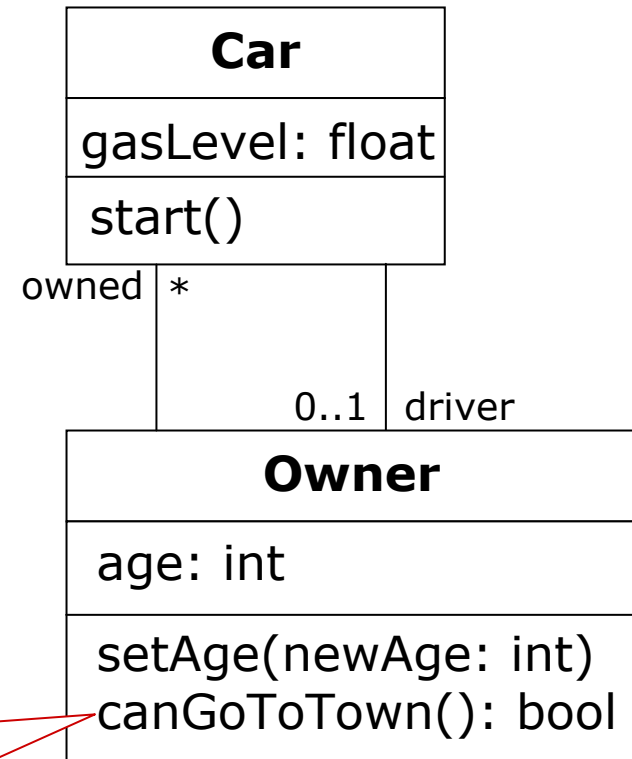
OCL Basics (2)

- ❑ OCL has the usual arithmetic (e.g. +, -, *, /, mod...) and string (concat, size) operations
- ❑ OCL also has the usual comparison operators
- ❑ Reference to context instance: self
- ❑ Can access fields, e.g.: self.age 
- ❑ Can navigate associations, e.g.: self.owner
- ❑ Built-in simple and collection types
 - special functionality for collection types, e.g. self.cars->size = 3 means that you have to have exactly three cars.
- ❑ Methods which are stereotyped <<query>> can be used in OCL expressions
- ❑ ... and much more ...

OCL Examples (1)



OCL Examples (2)

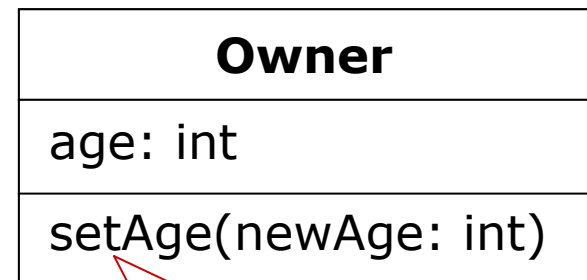


```
context Owner::canGoToTown(): bool
pre: self.owned->size > 0
pre: self.owned.gasLevel > 2
pre: self.age > 18
```

OCL to Code

- (Subcases of) OCL constraints can be translated into code

```
class Owner {  
  int age;  
  void setAge(int newAge) {  
    assert(newAge > 18);  
    ...  
  }  
}
```



```
context Owner::setAge(newAge: int)  
pre: newAge > 18
```

- This translation is not always possible
 - When to check? (e.g. invariants)
 - "transactional boundaries"
- ⇒ topic of research

References

□ OCL

■ Introduction to OCL

http://www.parlezuml.com/tutorials/ocl/index_files/frame.htm

■ [1] OCL Specification

<http://www.omg.org/docs/ptc/03-10-14.pdf>

□ Java

■ Sun's site

<http://java.sun.com>