

7. Implementation Phase

7.1 Architecture Diagrams

7.2 OO Languages: Java

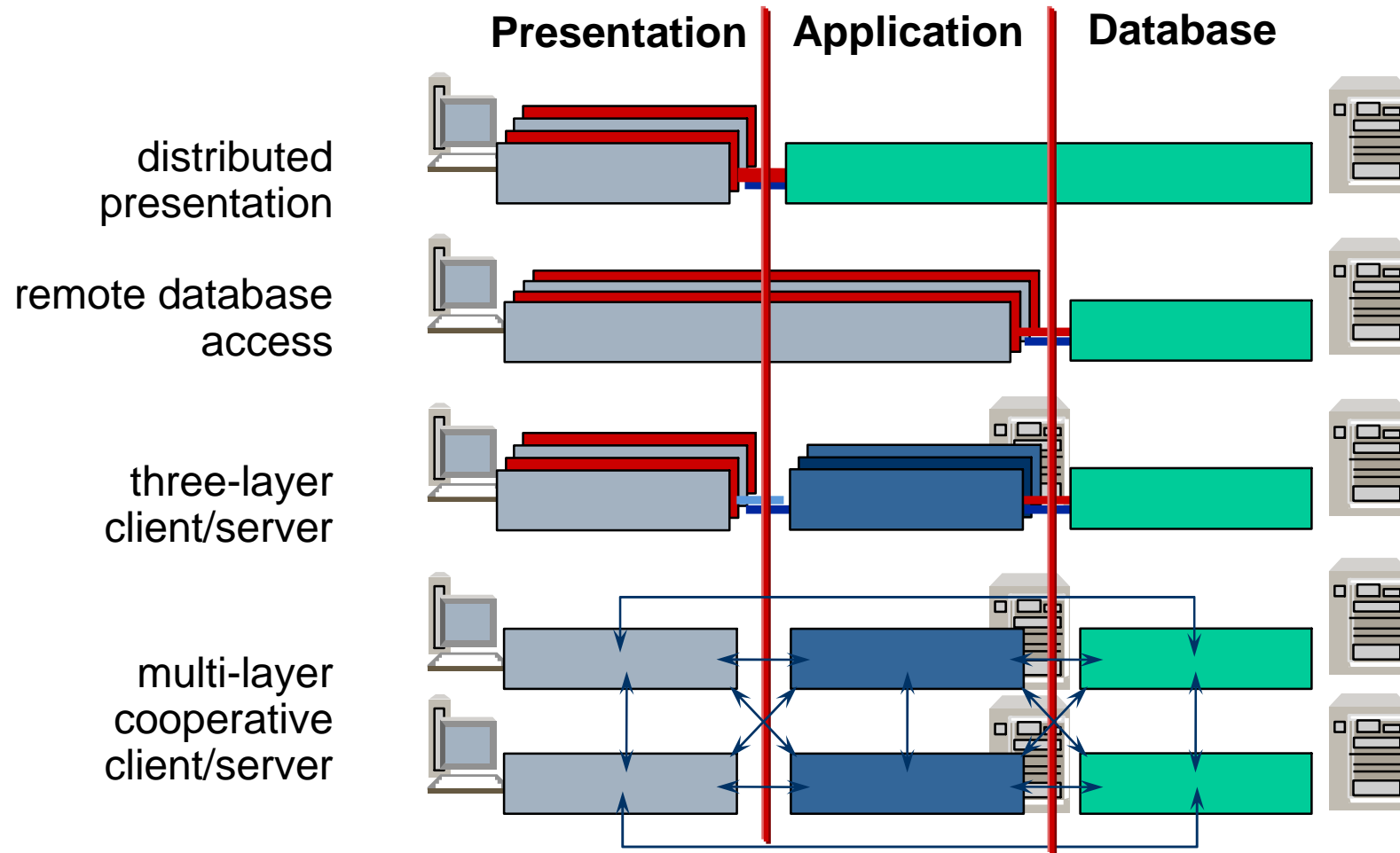
7.3 Constraint Languages: OCL

Architecture Design Models

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

Example: SAP R/3

- o Flexible three-tier client/server-architecture

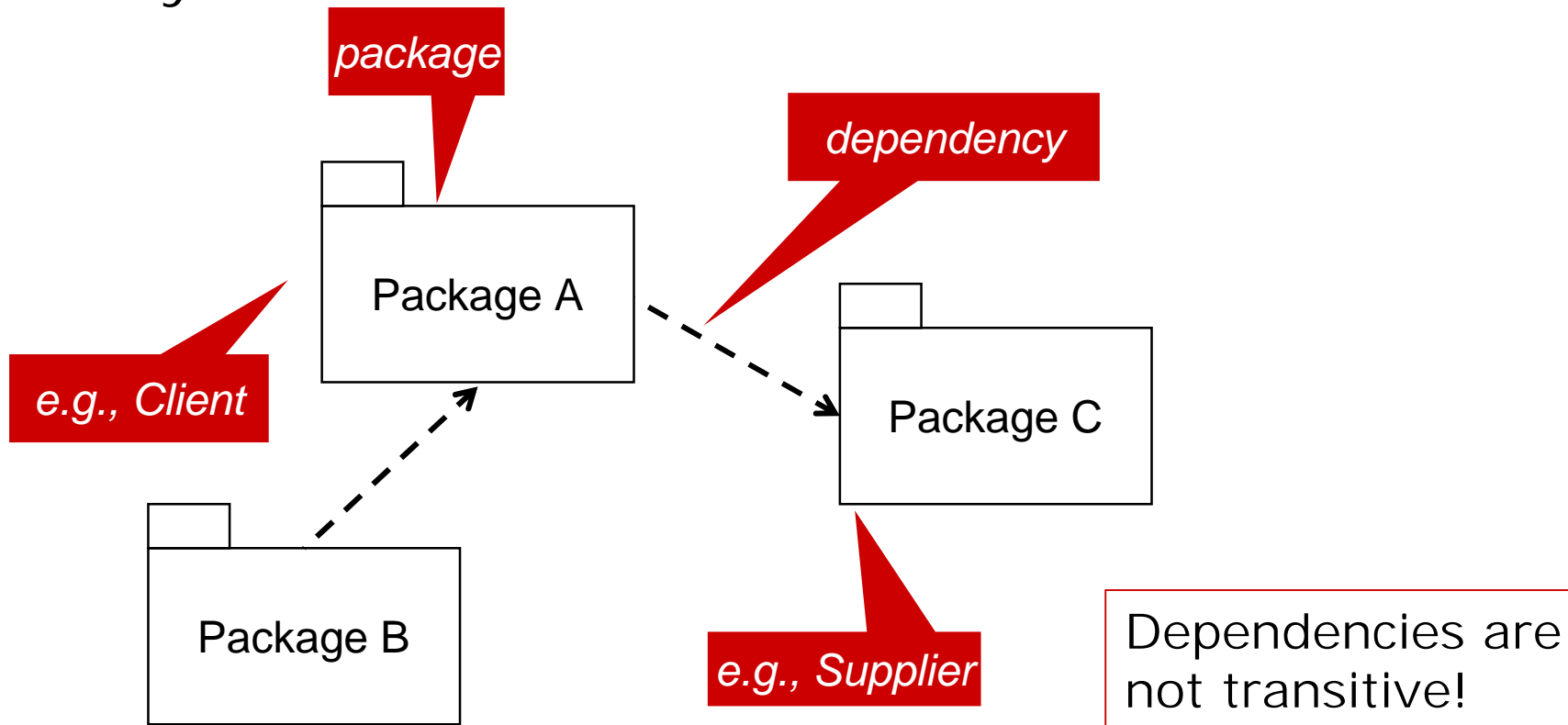


Specification Models in UML

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

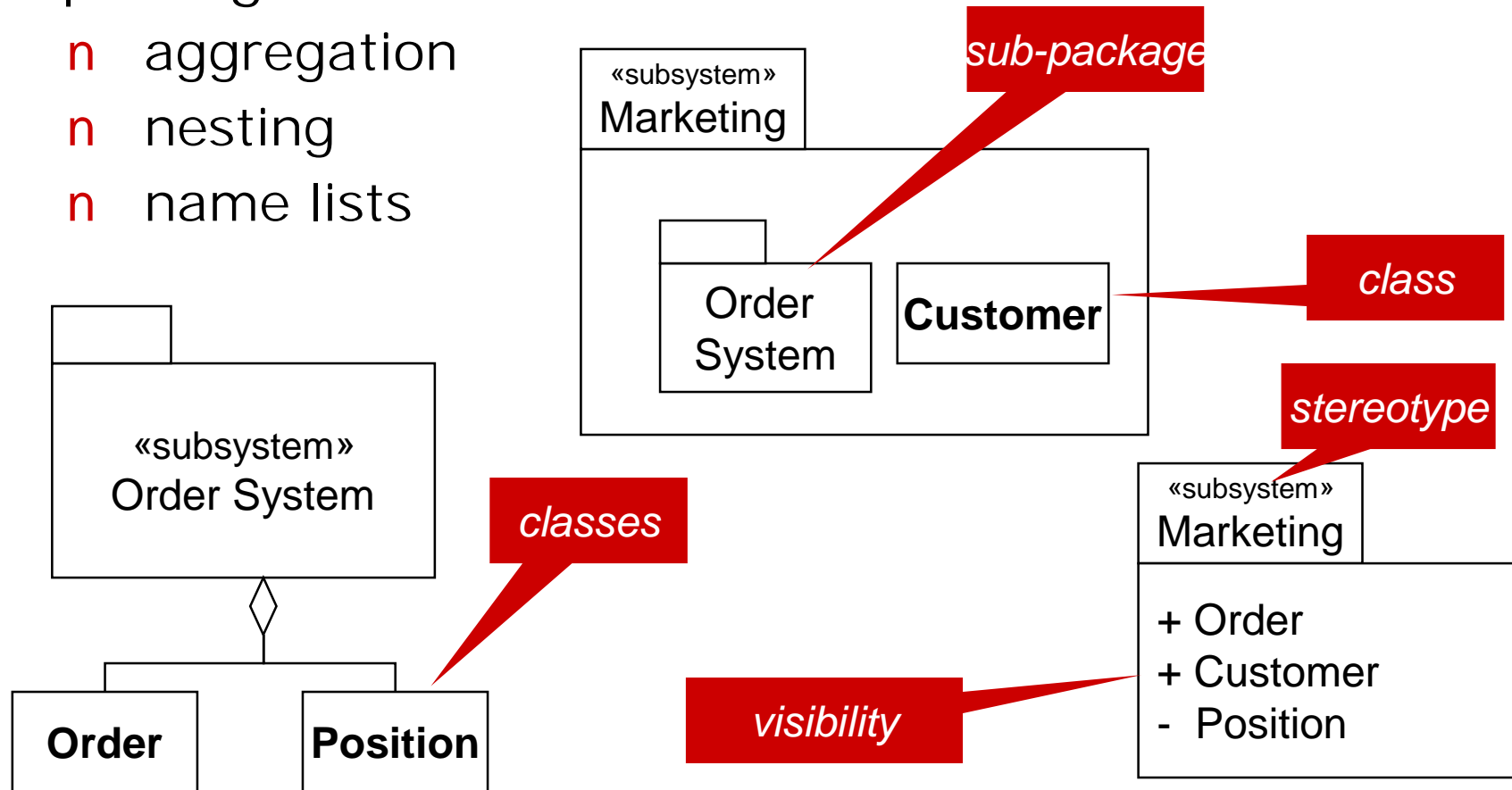
Package Diagram

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



Package Contents

- o Structuring alternatives for the representation of package content are:
 - n aggregation
 - n nesting
 - n name lists



Subsystems

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

Functional Decomposition

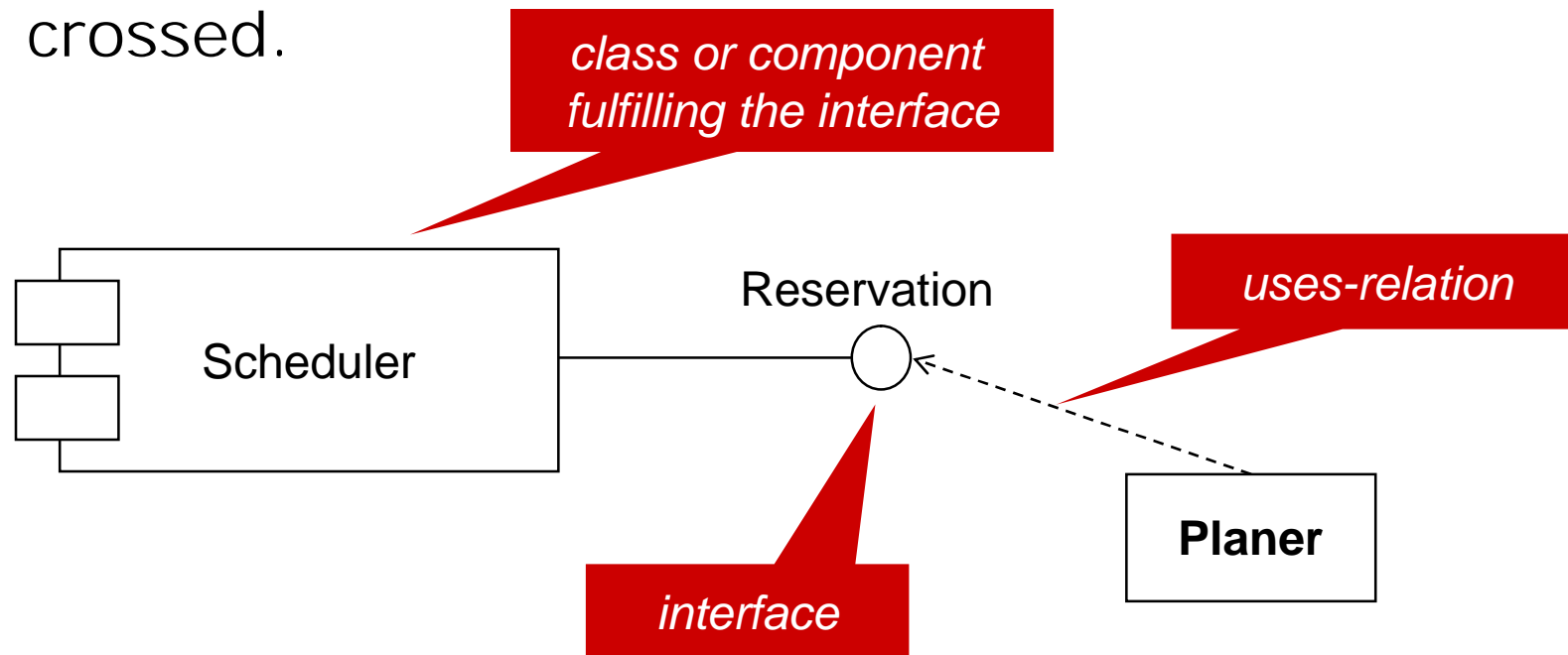
Map the total system on functions and sub-functions, starting with the use case.

OO Packages

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

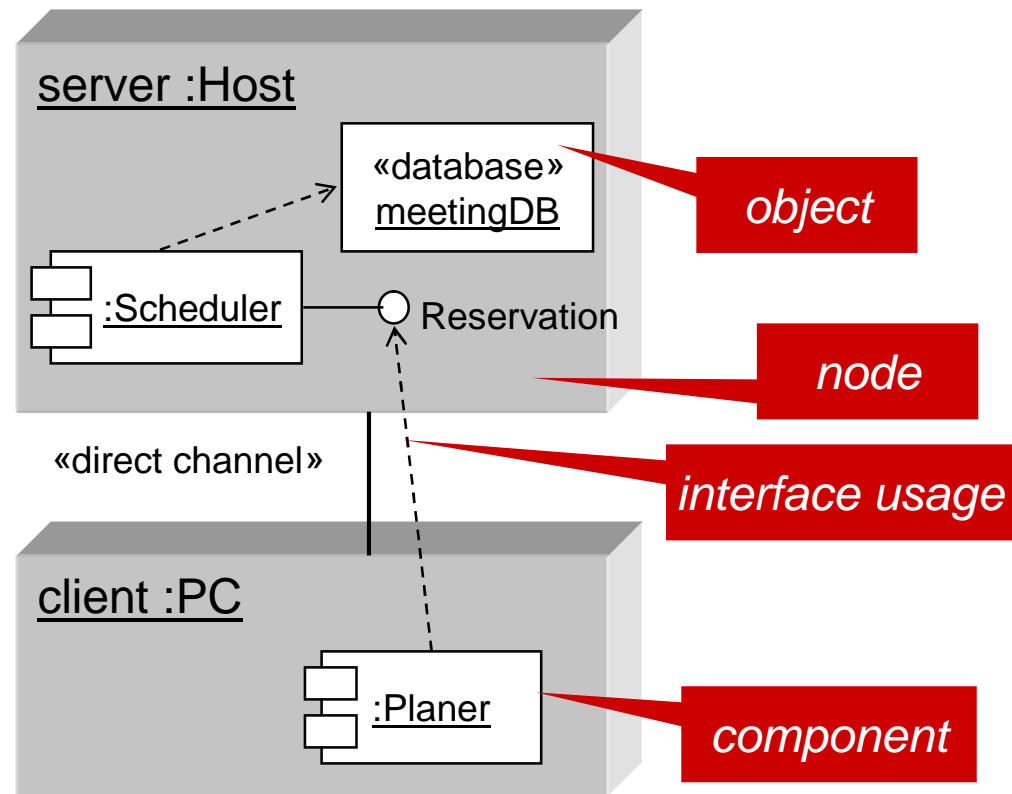
Interfaces in Architecture

- 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 .



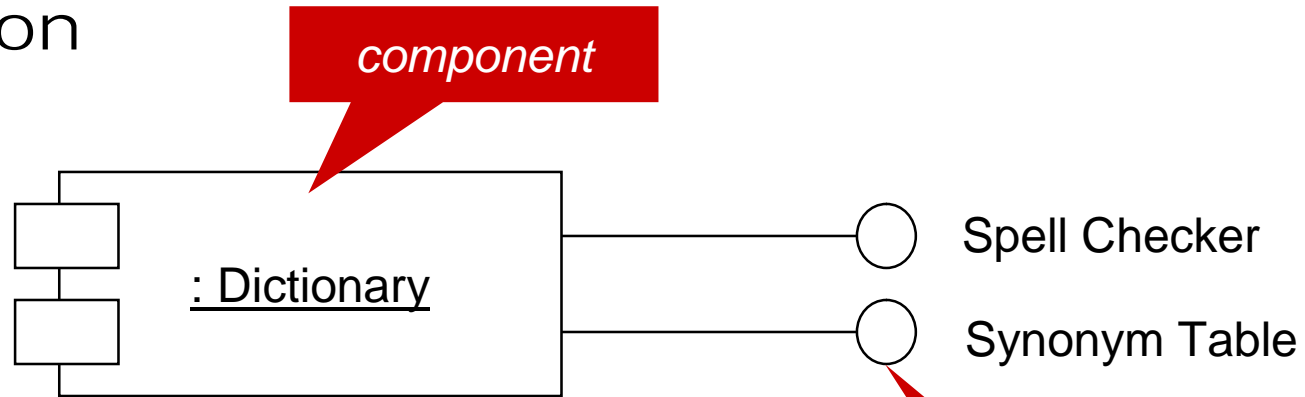
Components (1)

- o A component is a physical, replaceable part of a system containing an implementation which realizes a set of interfaces
- o Components have two aspects:
 - n Code: A component consists of code, components can contain or use components.
 - n 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,...)

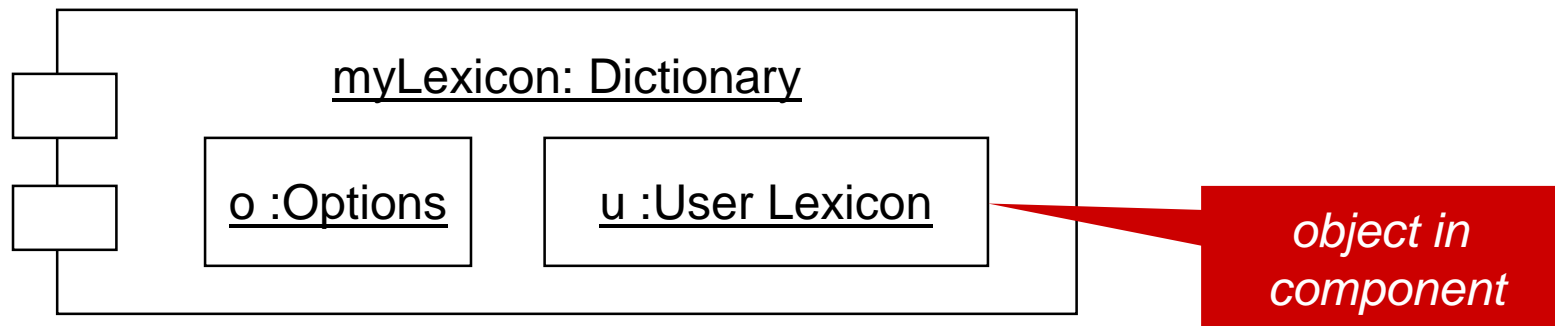
Example: Spelling checker as component:
- identity and state by user dictionary,
- different versions and languages.

Components (2)

- o Notation

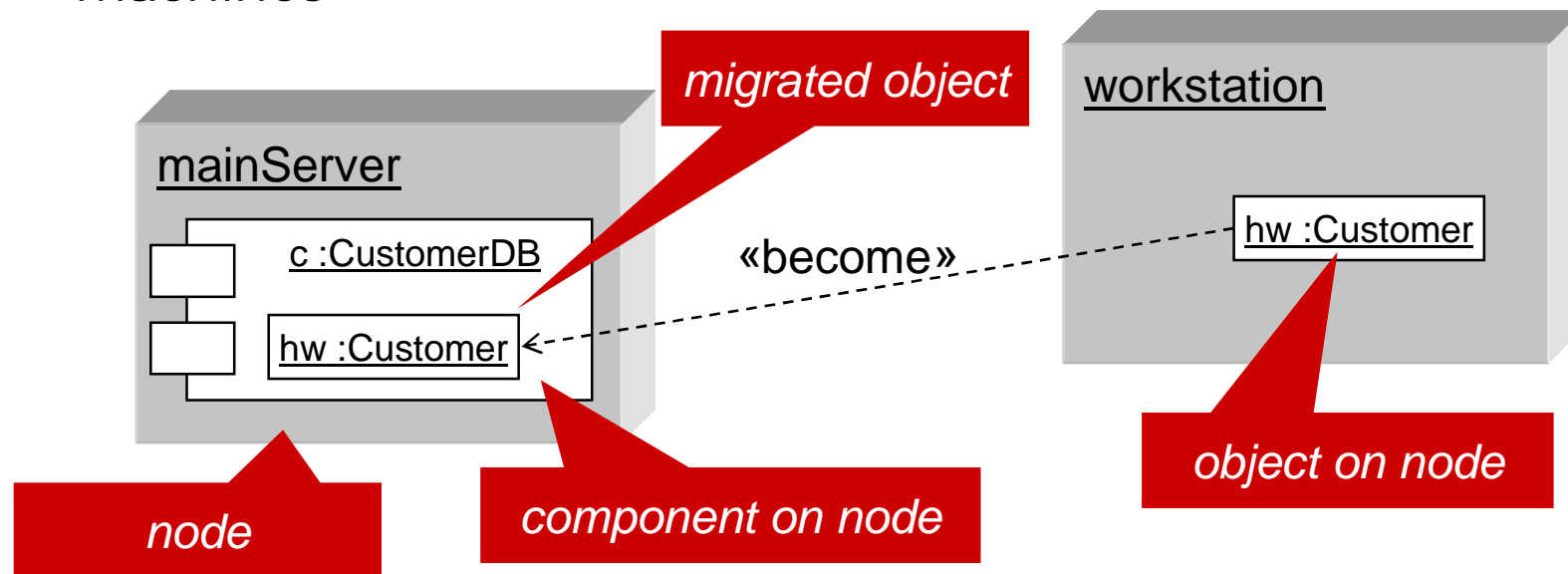


- o Component with identity



Node

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



Discussion

- o Components ...
 - n are conceptually and functionally larger than a classes.
 - n unite behavior and collaboration of a group of classes (see collaboration diagrams)
 - n are independent of other components but usually collaborate with them
 - n are replaceable with other realizations of the same interface
 - n are fundamental building blocks of component-based architectures
 - n 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

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

Languages

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

Your Cup of Java (its Grounds)

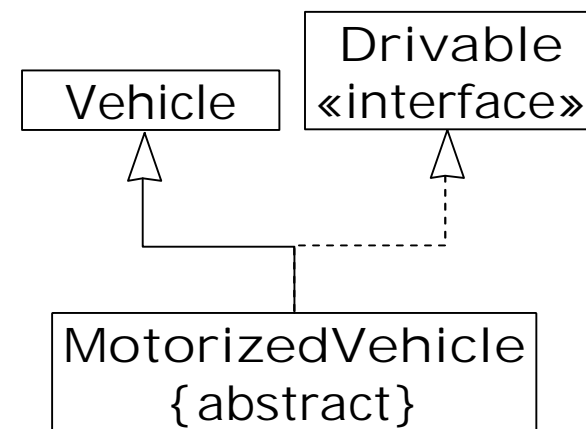


- o Not a full Java CUP (“Course Upon Programming”).
- o We just list the basic object-oriented concepts, i.e., how to
 - n declare classes, interfaces, methods, data members
 - n create and manipulate instances
 - n take advantage of object-oriented features like
 - o inheritance
 - o polymorphism,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

Declaring Classes

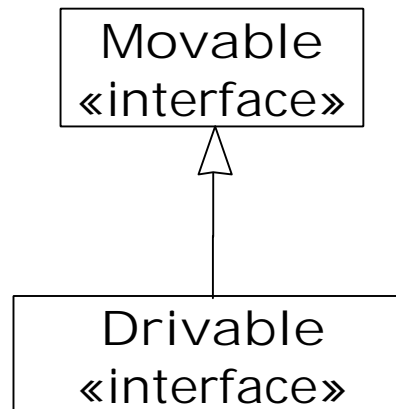
- o Basic syntax:
`<visibility> <modifiers> class <classname> extends
 <superclass-name> implements <list-of-interfaces>
 { <class-body> }`
- o For example:
`public abstract class MotorizedVehicle
 extends Vehicle implements Drivable { ... /*...*/ }`
- o Most elements can be omitted; mandatory is ...
 - n class
 - n <classname>
 - n { <class-body> }

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



Declaring Interfaces

- o Very similar to classes
 - n not all visibilities make sense
 - n there is no implements clause
- o Syntax:
`<visibility> <modifiers> interface <classname>
 extends <superinterface-name> { <interface-body> }`
- o 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

- o To create a new instance of a class, use the **new** operator:
`new Person("John", 42);`
 - n new instance is created
 - n constructor is called
- o The new instance is usually bound to a variable:
`Person p = new Person("John", 42);`
 - n you need the variable to later access the instance
 - n can store instances in lists etc.

Using Instances

- o Invoking methods:
`employee.raiseSalary(3.0);`
 - n uses late binding to call the right implementation of the method
- o Accessing fields:
`person.name = "Maria";`
 - n fields usually are not public
- o In Java you don't need to destroy instances. The runtime system does this for you:
 - n Garbage Collection
 - n Reachability

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

Control Flow (1)

- o You can actually also write algorithms in Java ...
 - n 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)

n 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

- o Good Old Pascal:

```
{ $I- }  
reset(F);  
{ $I+ }
```

Reopen file for writing

Check whether error occurred

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

```
begin
```

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

```
    halt;
```

```
end;
```

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

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

- o Manual error handling

- o Control flow becomes much more complicated

- o Nobody is forced to do error handling

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

Exceptions: The New Way

- o Java:

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

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

- n Methods can defer exception to the caller:

- public void test() **throws IOException** { ... }

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

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

Exception Handling (2)

- o 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 relationship between the caller and the callee. A red box labeled "Callee" points to the `readBytes()` method in the `Reading` class. A red box labeled "Caller" points to the `try` block in the `getData()` method of the same class, which is responsible for handling the exception thrown by the callee.

Constraint Language

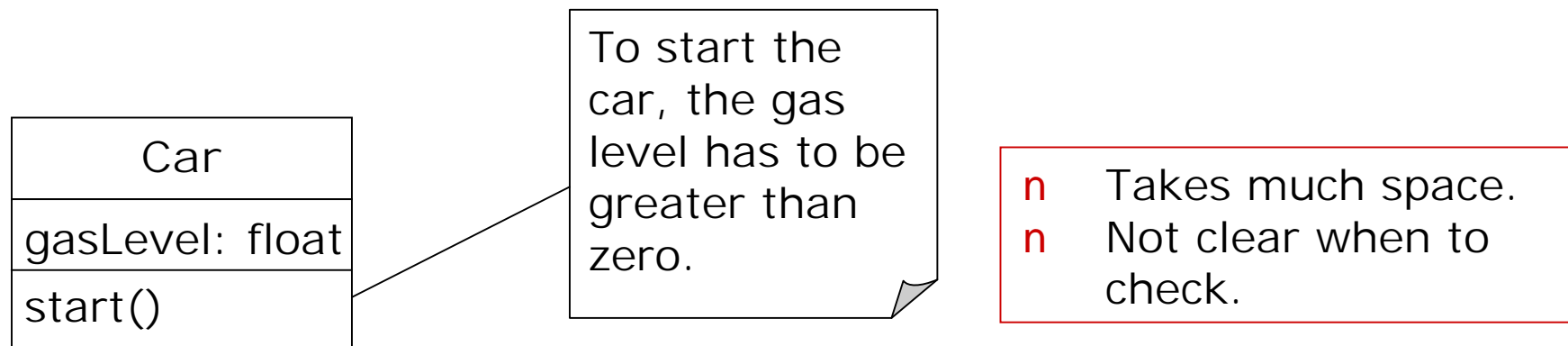
Algorithmic Programs
versus
Formal Logic Declarations

Constraints

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

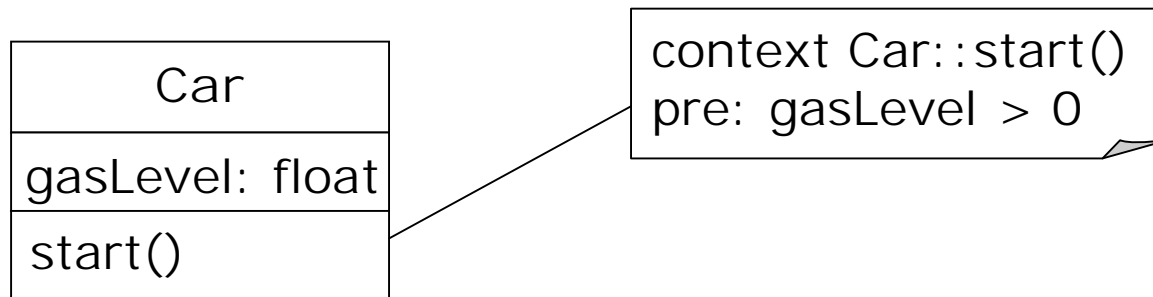
Constraint Example

- o You can specify constraints in natural language and attach them as notes in UML diagrams
- o Natural language has disadvantages:
 - n Even simple constraints take much space.
 - n Complex constraints are hard to understand.
 - n 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



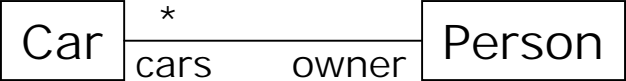
Object Constraint Language

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

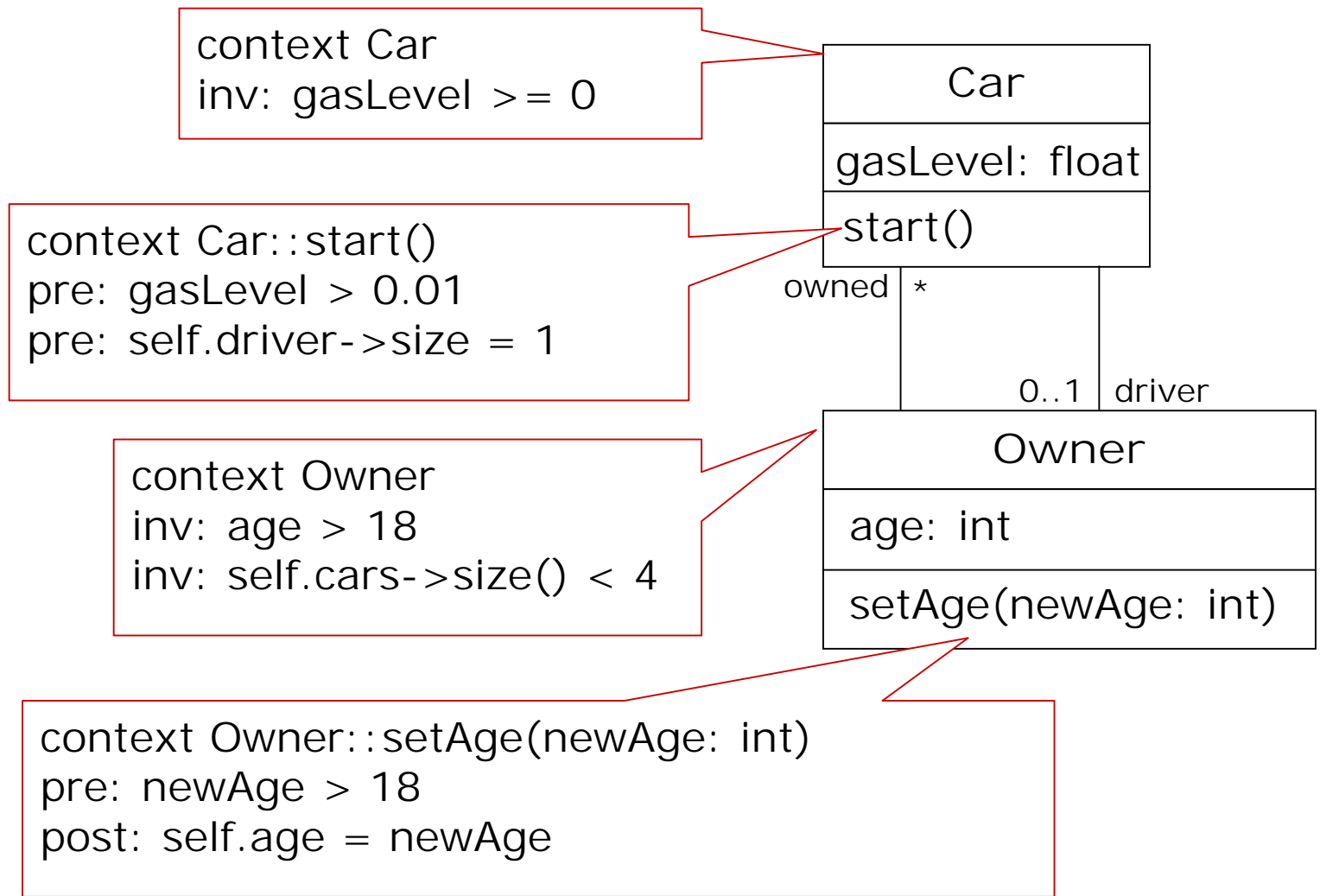
OCL Basics

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

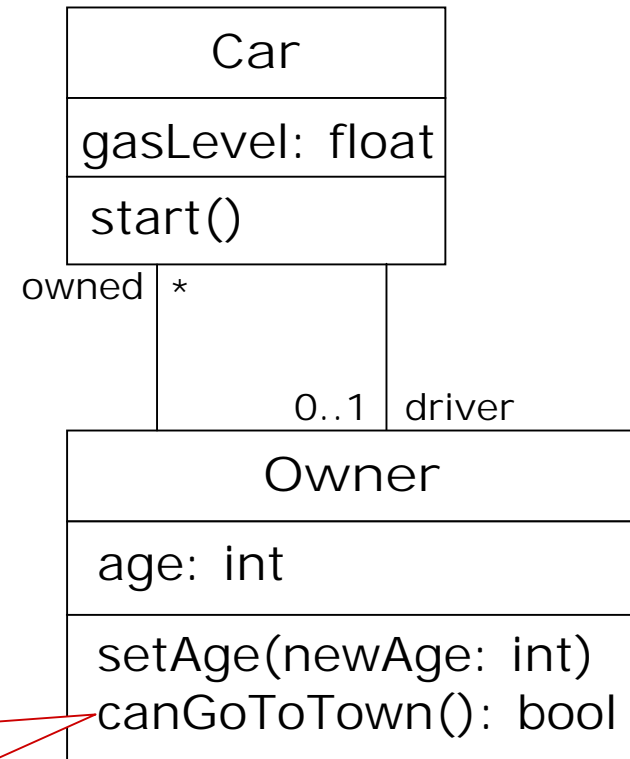
OCL Basics (2)

- 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)

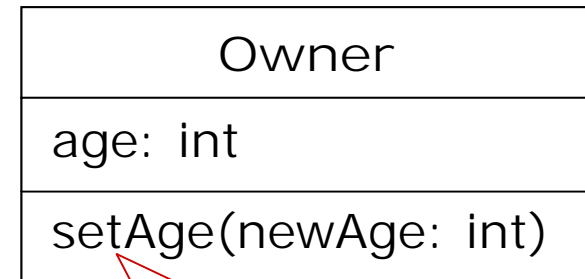


```
context Owner::canGoToTown(): bool
pre: self.owned->size > 0
pre: self.owned.gasLevel > 2
pre: self.age > 18
```

OCL to Code

- o (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
```

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

References

- o OCL
 - n Introduction to OCL
http://www.parlezuml.com/tutorials/ocl/index_files/frame.htm
 - n [1] OCL Specification
<http://www.omg.org/docs/ptc/03-10-14.pdf>
- o Java
 - n Sun's site
<http://java.sun.com>