

Lecture 06:

Formal Modeling with OCL

Christoph Lüth, Dieter Hutter, Jan Peleska

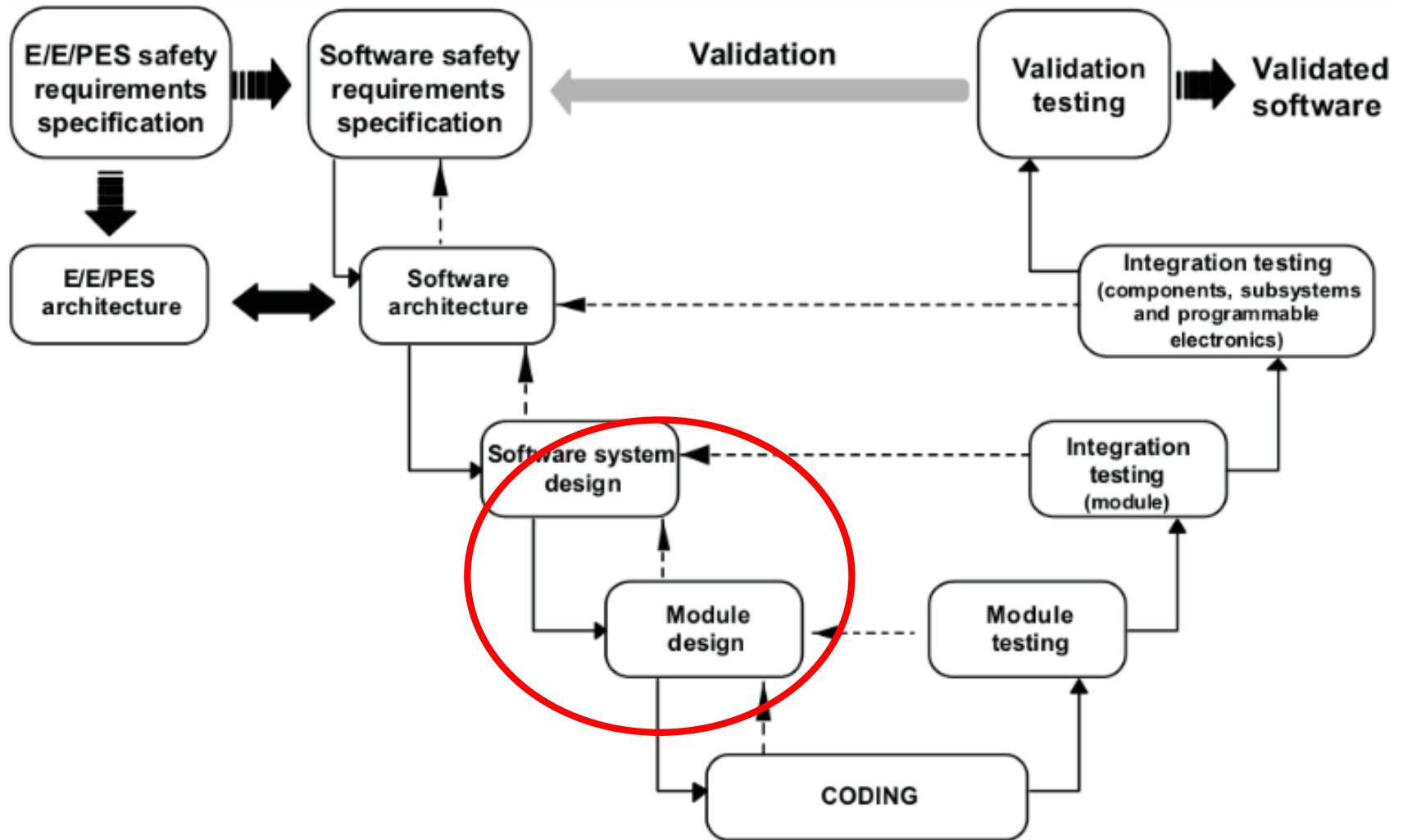


mit Folien v. Bernhard Beckert (KIT)

Where are we?

- ▶ 01: Concepts of Quality
- ▶ 02: Legal Requirements: Norms and Standards
- ▶ 03: The Software Development Process
- ▶ 04: Hazard Analysis
- ▶ 05: High-Level Design with SysML
- ▶ 06: Formal Modelling with OCL
- ▶ 07: Testing
- ▶ 08: Static Program Analysis
- ▶ 09-10: Software Verification
- ▶ 11-12: Model Checking
- ▶ 13: Conclusions

Formal Modeling in the Development Cycle



What is OCL?

- ▶ OCL is the **Object Constraint Language**.
 - ▶ Standardized by OMG actual version is OCL 2.4
 - ▶ Available at <https://www.omg.org/spec/OCL/>
- ▶ What is OCL?
 - ▶ *„A formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model.“* (OCL standard, §7)
- ▶ Why OCL?
 - ▶ *„A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model.“* (OCL standard, §7.1)

Characteristics of the OCL

- ▶ OCL is a pure **specification language**
 - ▶ OCL expressions do not have side effects
- ▶ OCL is **not** a programming language.
 - ▶ Expressions are not executable (though some may be)
- ▶ OCL is **typed** language
 - ▶ Each expression has a type; all expressions must be well-typed
 - ▶ Types are classes, defined by class diagrams

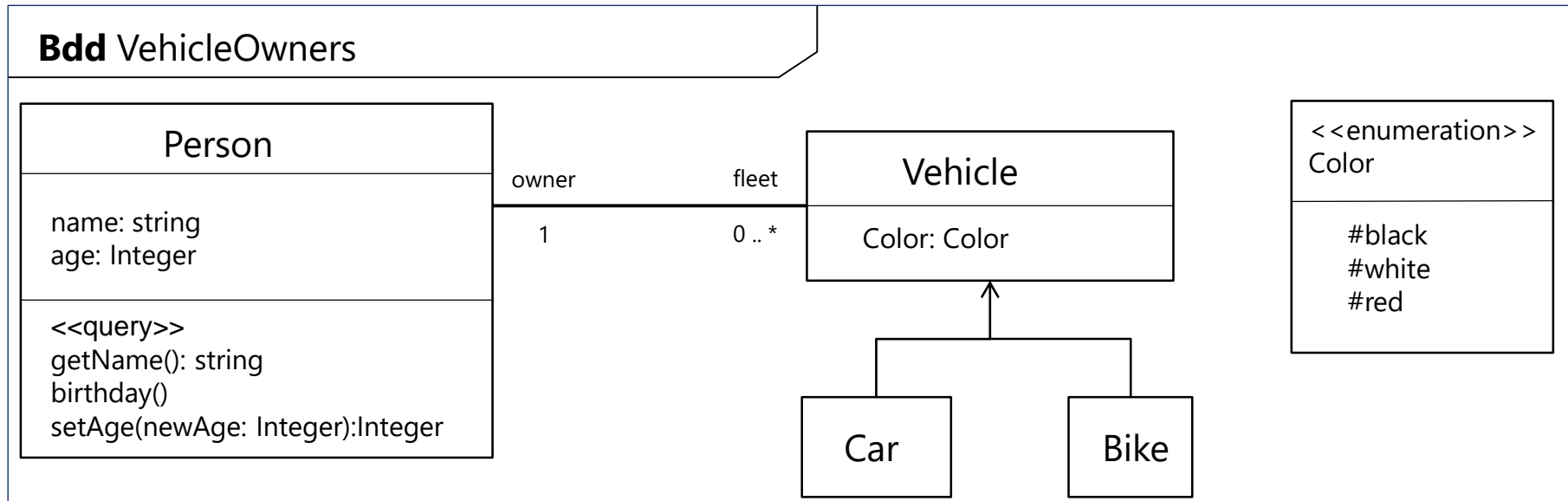
Usage of the OCL

- ▶ as a query language
- ▶ to specify invariants on classes and types in the class
- ▶ to specify type invariant for Stereotypes
- ▶ to describe pre- and post conditions on Operations and Methods
- ▶ to describe guards
- ▶ to specify target (sets) for messages and actions
- ▶ to specify constraints on operations
- ▶ to specify derivation rules for attributes for any expression over a UML model.

(OCL standard, §7.1.1)

OCL by Example

Why is SysML not enough?



What about requirements like:

- ▶ The minimal age of car owners
- ▶ The maximal number of cars (of a specific color) owned
- ▶ The maximal number of owners of a car

OCL Basics

- ▶ The language is **typed**: each expression has a type.
- ▶ Multiple-valued logic (true, false, undefined).
- ▶ Expressions always live in a **context**:
 - ▶ **Invariants** on classes, interfaces, types.

```
context Class
  inv Name: expr
```

- ▶ **Pre/postconditions** on operations or methods

```
context Class :: op(a1: Type, ..., an: Type) : Type
  pre Name: expr
  post Name: expr
```

OCL Types

▶ Basic types:

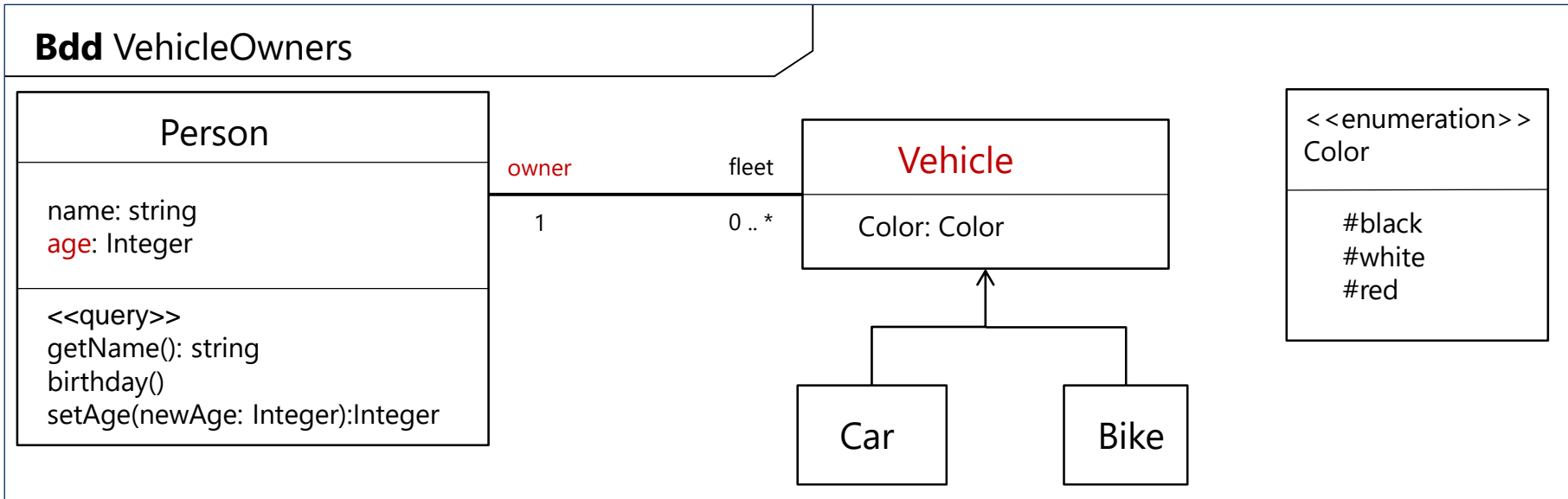
- ▶ `Boolean`, `Integer`, `Real`, `String`
- ▶ `OclAny` – Enthält alle Typen
- ▶ `OclVoid` – In allen Typen enthalten, nur eine Instanz `null`
- ▶ `OclInvalid` – Fehlerwert (nur eine Instanz `invalid`)

▶ Collection types:

- ▶ `Sequences`, `Bag`, `OrderedSet`, `Set`

▶ Model types

Invariants of Classes



“A vehicle owner must be at least 18 years old”

```
context    Vehicle
inv:      self.owner.age >= 18
```

Basic types and operations

- ▶ Integer (\mathbb{Z}) OCL-Std. §11.5.2
- ▶ Real (\mathbb{R}) OCL-Std. §11.5.1
 - ▶ Integer is a subclass of Real
 - ▶ round, floor from Real to Integer
- ▶ String (Zeichenketten) OCL-Std. §11.5.3
 - ▶ substring, toReal, toInteger, characters, etc.
- ▶ Boolean (Wahrheitswerte) OCL-Std. §11.5.4
 - ▶ or, xor, and, implies
 - ▶ Relationen auf Real, Integer, String

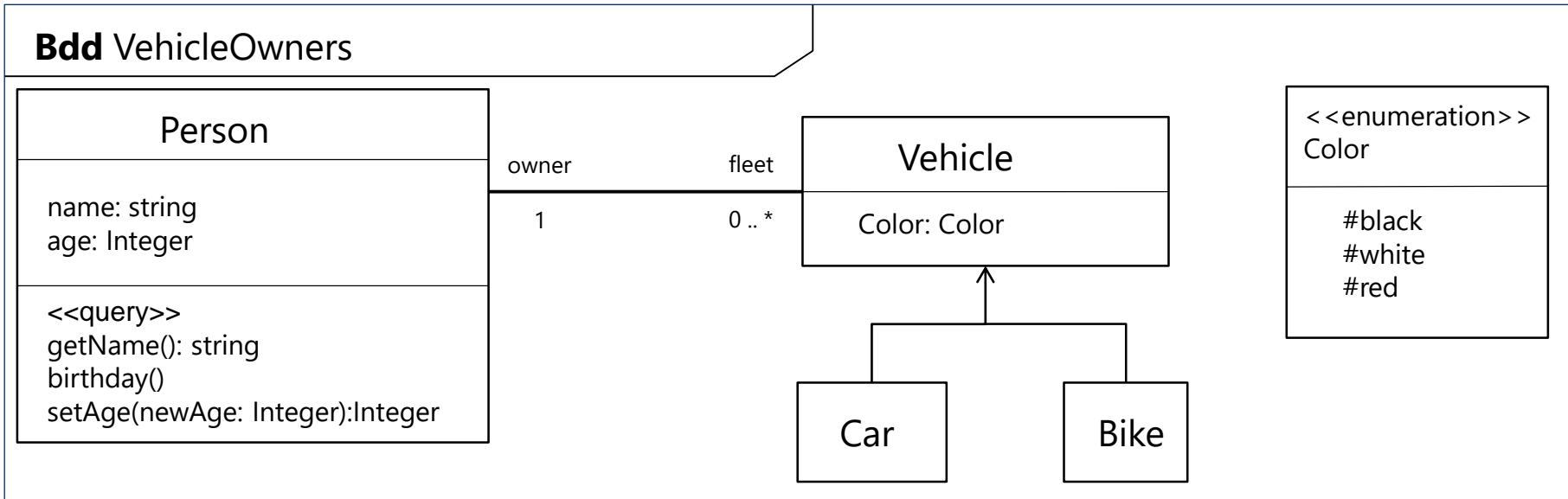
Collection Types

Sequence, Bag, OrderedSet, Set

OCL-Std. §11.6, §11.7

- ▶ Operations on all collections:
 - ▶ `size`, `includes`, `count`, `isEmpty`, `flatten`
 - ▶ Collections are always „flattened“
 - ▶ Syntax: `collection->operation(...)`
- ▶ Set, OrderedSet
 - ▶ `union`, `intersection`
- ▶ Bag
 - ▶ `union`, `intersection`, `count`
- ▶ Sequence (lists)
 - ▶ `first`, `last`, `reverse`, `prepend`, `append`

Collections



“Nobody has more than 3 vehicles”

```
context      Person
Inv:        self.fleet->size <= 3
```

Collection Types: Quantification

We can quantify over collections:

OCL-Std. §11.9.1

► **Universal quantification :**

```
coll->forall(elem: Type | expr[elem]) : Boolean
```

► **Existential quantification:**

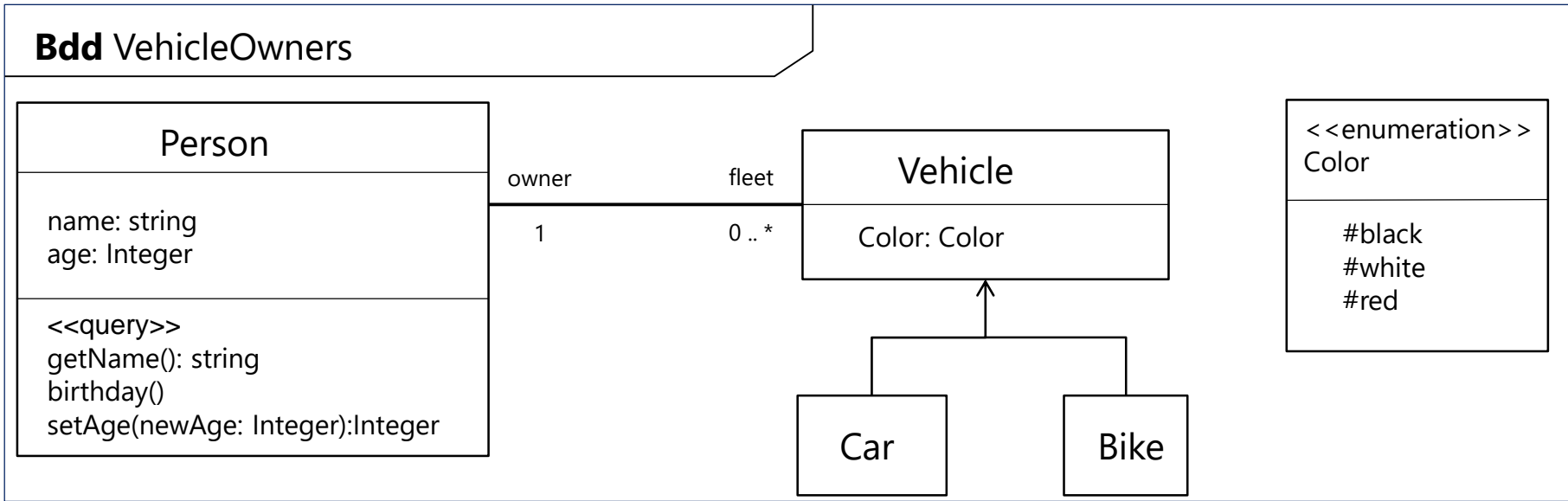
```
coll->exists(elem: Type | expr[elem]) : Boolean
```

► **Comprehension operator:**

```
coll->select(elem: Type | expr[elem]) : Coll[Type]
```

where `expr` is an expression of type `Boolean`.

Universal Quantification



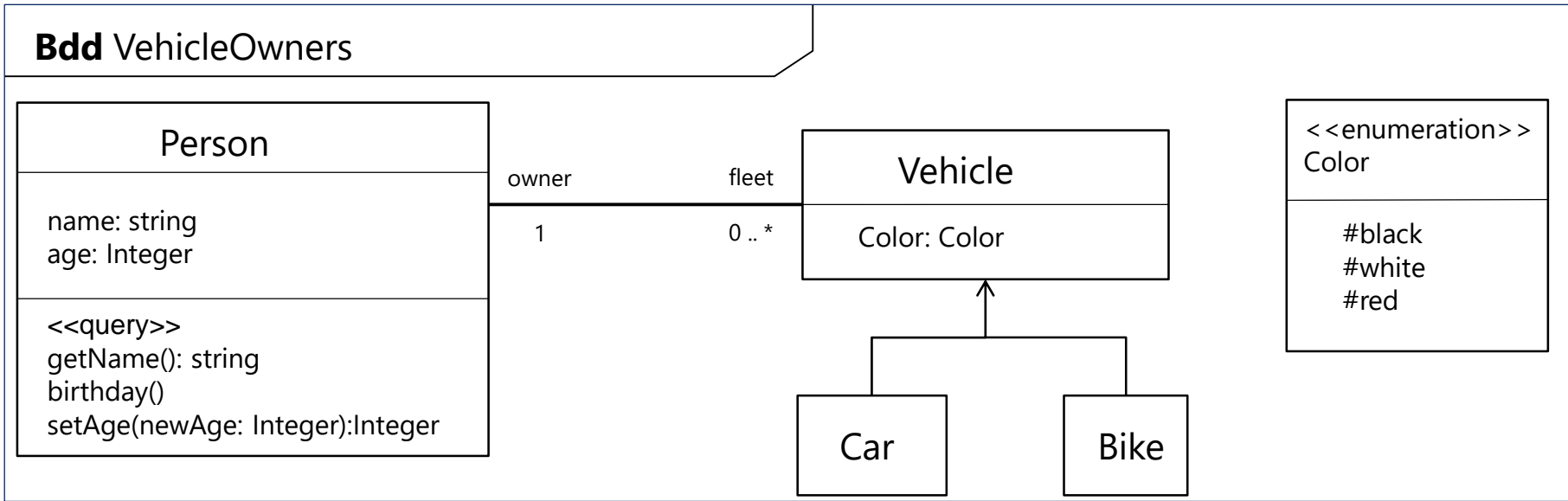
“All vehicles of a person are black”

```
context      Person
inv:         self.fleet->forAll(v | v.color = #black)
```

“No person has more than three black vehicles”

```
context      Person
inv:         self.fleet->select(v | v.color = #black)->size <= 3
```

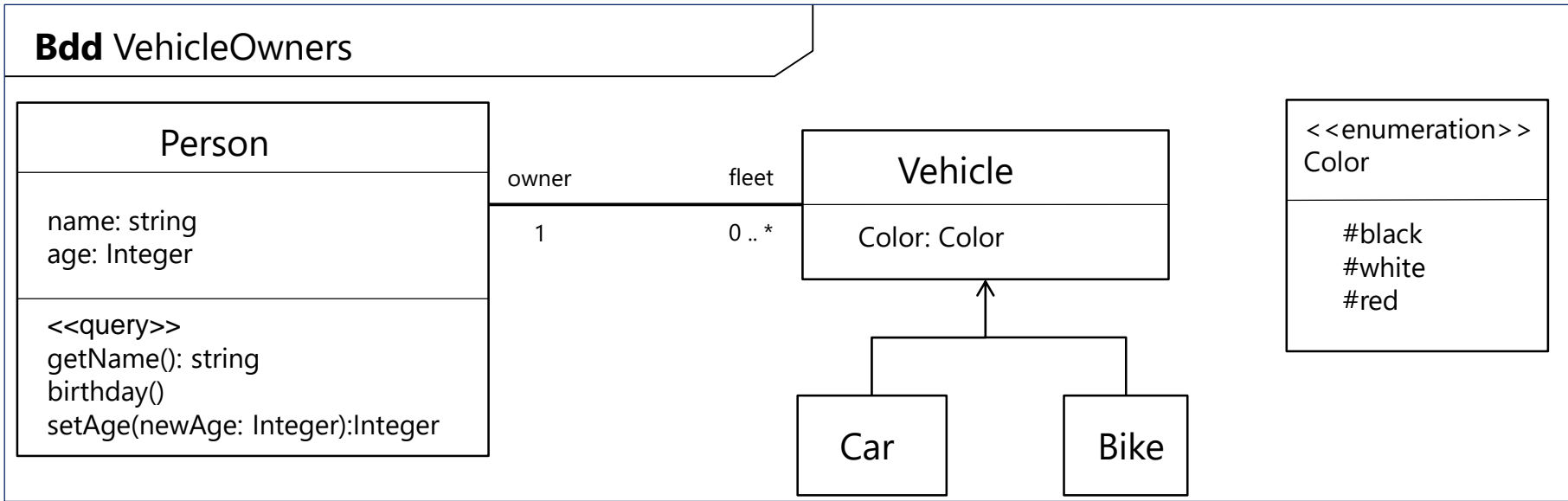

Universal Quantification



“A person younger than 18 owns no cars”

```
context Person
inv: self.age < 18 implies
    self.fleet -> forall(v | not v.isKindOf(Car))
```

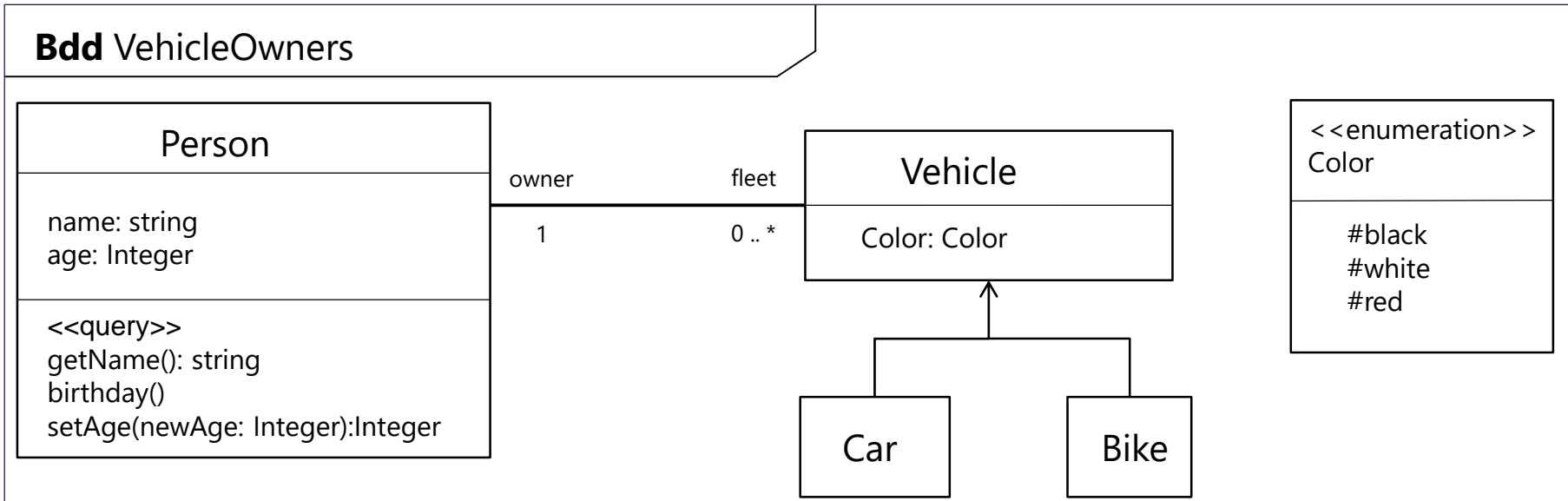
Existential Quantification



“There is a red car”

```
context    Car
inv:      Car.allInstances()->exists(c | c.color=#red)
```

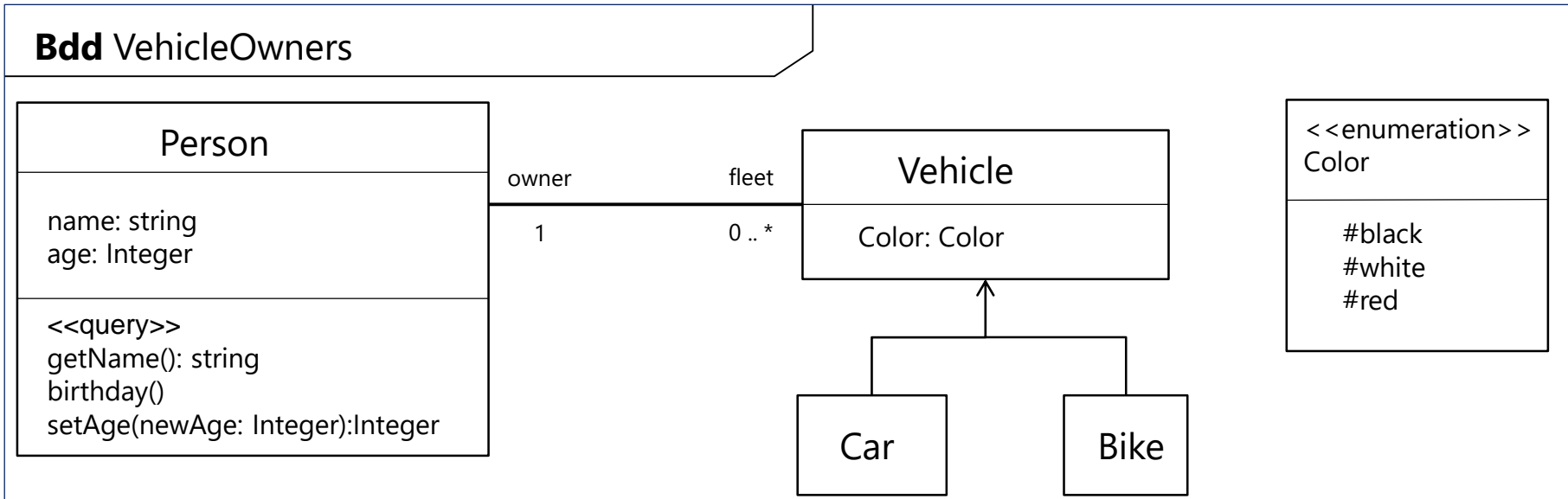
Pre/Post Conditions



“If `setAge(a)` is called with a non-negative argument `a`, then `a` becomes the new value of the attribute `age`.”

```
context    Person::setAge(a:int)
pre:      a >= 0
post:    self.age = a
```

Pre/Post Conditions



“Calling `birthday()` increments the age of a person by 1.”

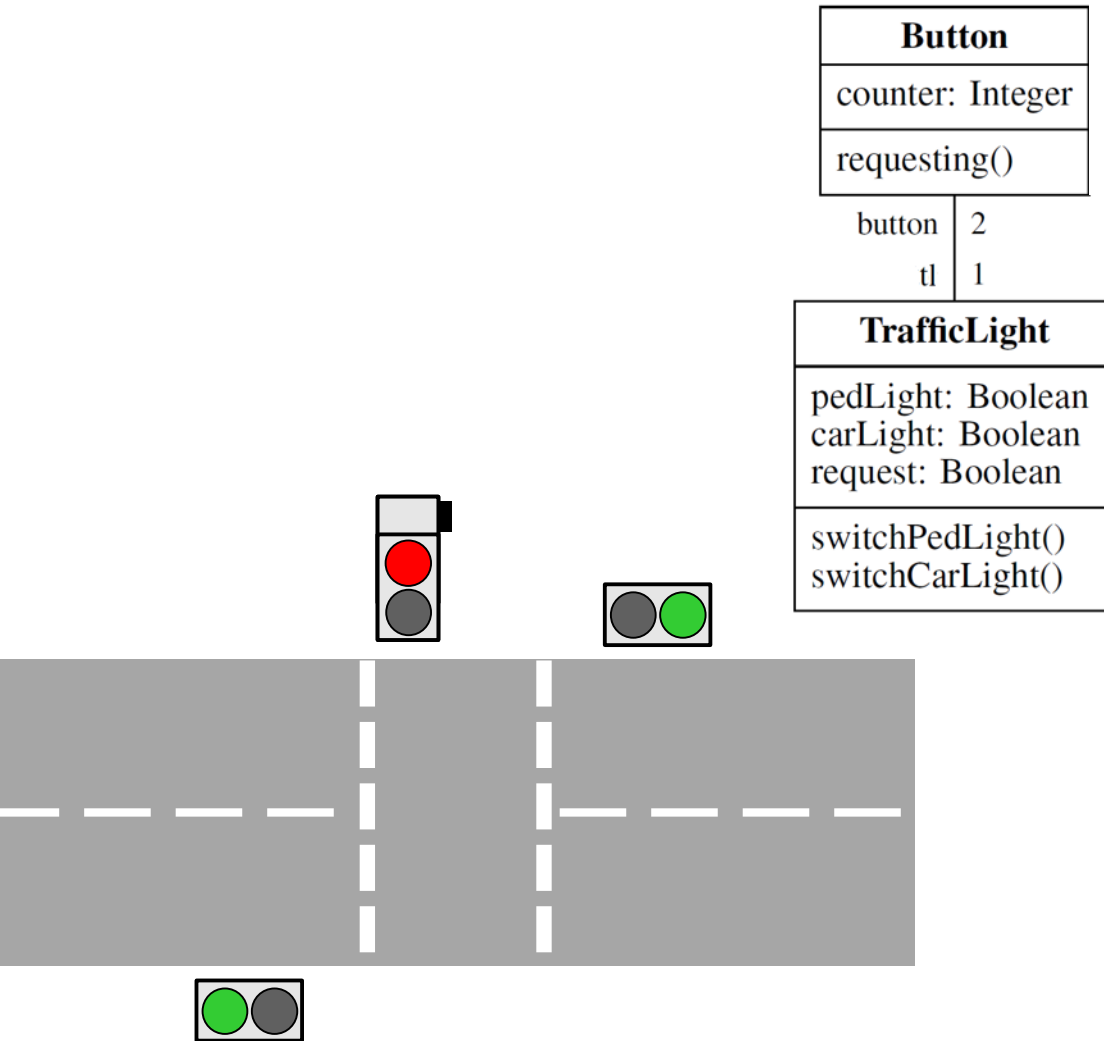
```
context    Person::birthday()
post:     self.age = self.age@pre + 1
```

Dynamic Aspects

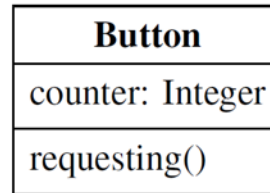
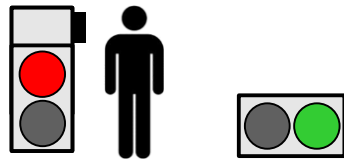
Modelling Dynamic Aspects

- ▶ Block diagrams model the **static structure** of the system: classes, attributes and the type of the operations. The possible **system states** are all instances of these model types.
- ▶ Invariants and pre/post conditions can be used to model the **dynamic aspects** of the system. In particular, they model all possible **state transitions** between the system states.
- ▶ An operation can become **active** (there is a state transition emanating from it) if the invariant holds, and the precondition holds. If there are no active state transitions, the system is **deadlocked**.
 - ▶ *Deadlocks should be avoided.*

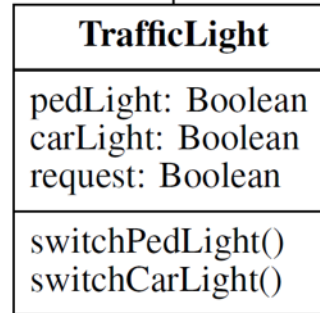
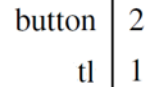
Example: The Traffic Light



Example: The Traffic Light



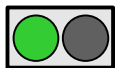
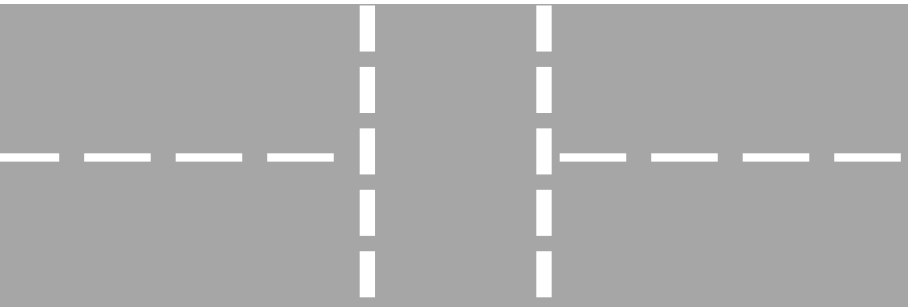
```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```



```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

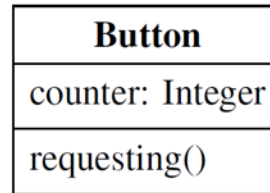
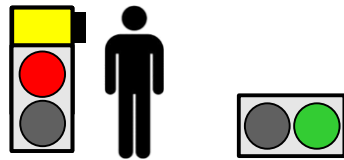
```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not (pedLight = true and
  carLight = true)
```

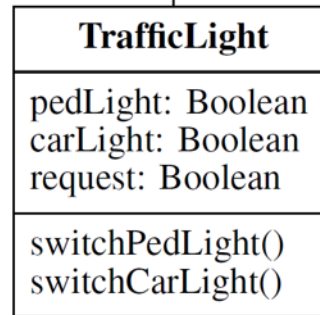
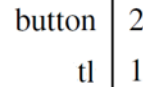


pedLight: False
 carLight: True
 request: False
 counter: 0

Example: The Traffic Light



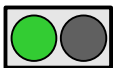
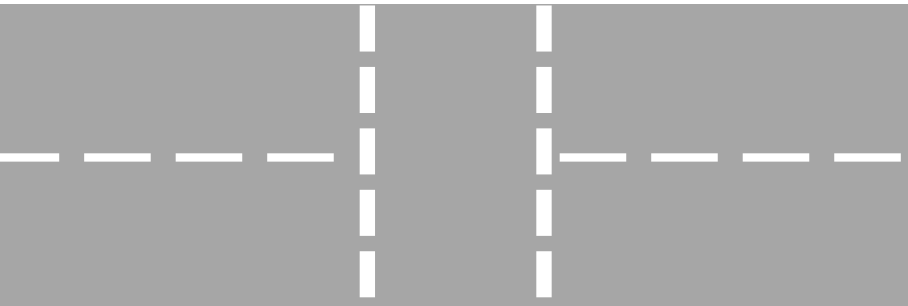
```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```



```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

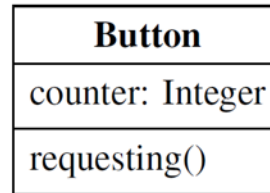
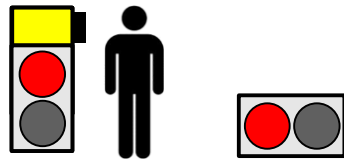
```
context switchCarLight()
  post: carLight != carLight@pre
```

```
inv: not (pedLight = true and
  carLight = true)
```

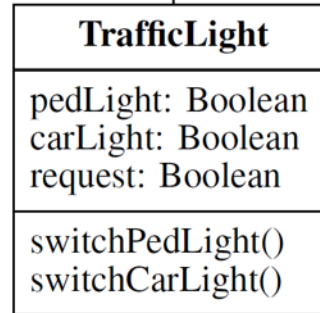
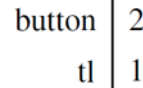


pedLight: False
 carLight: True
 request: True
 counter: 1

Example: The Traffic Light



```
context requesting()
  pre: tl.pedLight = false
  post: tl.request = true
  post: counter = counter@pre + 1
```

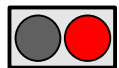


```
context switchPedLight()
  pre: request = true
  post: pedLight != pedLight@pre
  post: request = false
```

```
context switchCarLight()
  post: carLight != carLight@pre
```

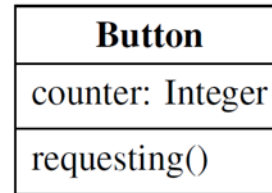
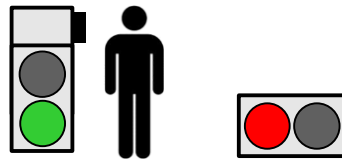
```
inv: not (pedLight = true and
  carLight = true)
```

pedLight: False
 carLight: False
 request: True
 counter: 1

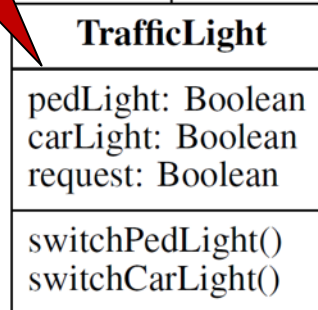


Example: The Traffic Light

Deadlock



| | |
|--------|---|
| button | 2 |
| tl | 1 |



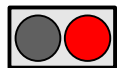
```
context requesting()
pre: tl.pedLight = false
post: tl.request = true
post: counter = counter@pre + 1
```

```
context switchPedLight()
pre: request = true
post: pedLight != pedLight@pre
post: request = false
```

```
context switchCarLight()
post: carLight != carLight@pre
```

```
inv: not (pedLight = true and
carLight = true)
```

pedLight: True
carLight: False
request: False
counter: 1



OCL Details

Model types

- ▶ Model types are given by
 - ▶ Attributes,
 - ▶ Operations, and
 - ▶ Associations of the model
- ▶ Navigation along the association
 - ▶ If cardinality is 1, type is of target type \mathbf{T}
 - ▶ Otherwise, it is `Set (T)`
- ▶ User-defined operations in expressions have to be stateless (stereotype `<<query>>`)

Collection Types: Iterators

- ▶ Quantifiers are a special case of iterators.
 - ▶ Think of `all/any` in Haskell defined via `fold`

- ▶ All iterators defined via `iterate`

OCaml-Std. §7.6.6

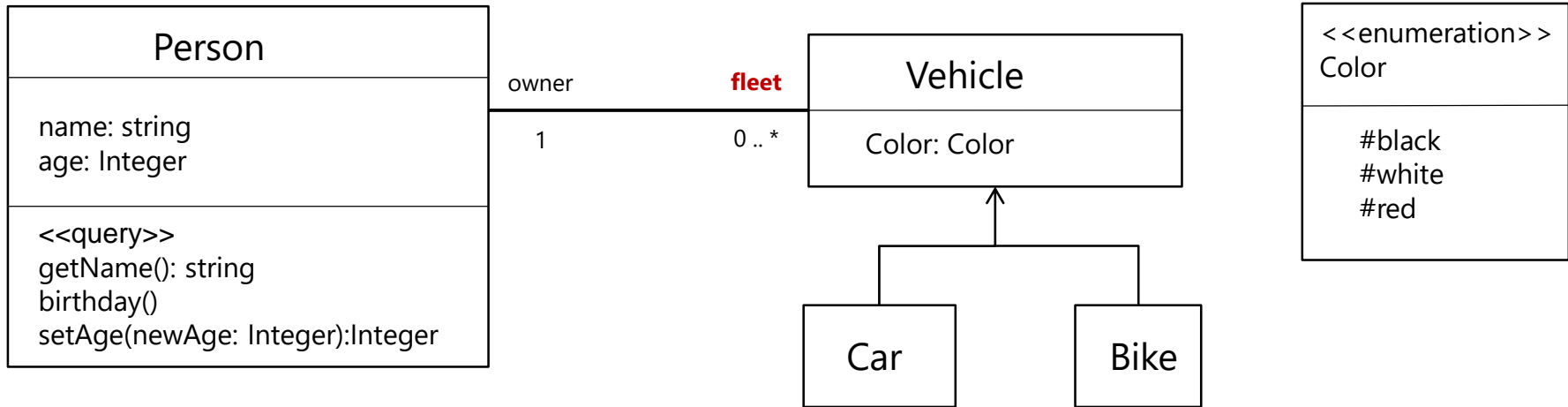
```
coll->iterate(elem: T; acc: T2 = initial_expr
             | expr[elem, acc]) : T2
```

where `expr` of type `T` denotes a function on `elem` and `acc`

```
c.iterate(e: T, acc: T2 = v) = {
  acc= v;
  for (Enumeration e= c.elements(); e.hasMoreElements();) {
    acc= expr[e, acc];
    e= e.nextElement();
  }
  return acc;
}
```

acc stands for
"accumulator"

Collection Types: Iterators



“A person owns at most 3 black vehicles”

```
context    Person
inv:      self.fleet->iterate(v; acc:Integer = 0
          | if (v.color = #black)
            then acc + 1 else acc
          endif ) <= 3
```

Undefinedness in OCL

- ▶ Each domain of a basic type has two values denoting “**undefinedness**”:
OCL-Std §A.2.1.1
 - ▶ *null* or ε stands for “undefined”, e.g. if an attribute value has not been set or is not defined (Type `OclVoid`)
 - ▶ *invalid* or \perp stands for “invalid” and signals an error in the evaluation of an expression (e.g. division by 0, or application of a partial function) (Type `OclInvalid`)
 - ▶ As subtypes: `OclInvalid` \subseteq `OclVoid` \subseteq all other types
- ▶ Undefinedness is **propagated**.
 - ▶ In other words, all operations are **strict**: „an *invalid* or *null* operand causes an *invalid* result“.

The OCL Logic

- ▶ Exceptions to strictness:

- ▶ Boolean operators (see below)
- ▶ Case distinction
- ▶ Test on definedness: `oclIsUndefined` with

$$\text{oclIsUndefined}(e) = \begin{cases} \text{true} & \text{if } e = \perp \vee e = \text{null} \\ \text{false} & \text{otherwise} \end{cases}$$

- ▶ The domain type for `Boolean` also contains null and invalid.

- ▶ The resulting logic is **four-valued**.
- ▶ It is a **Kleene-Logic**: $A \rightarrow B \equiv \neg A \vee B$
- ▶ Boolean operators (`and`, `or`, `implies`, `xor`) are **non-strict on both sides**.
- ▶ But equality (like all other relations) is strict: $\perp = \perp$ is \perp

OCL Boolean Operators: Truth Table

| b_1 | b_2 | b_1 and b_2 | b_1 or b_2 | b_1 xor b_2 | b_1 implies b_2 | not b_1 |
|---------------|--------------------------|-----------------|----------------|-----------------|---------------------|---------------|
| false | false | false | false | false | true | true |
| false | true | false | true | true | true | true |
| true | false | false | true | true | false | false |
| true | true | true | true | false | true | false |
| false | ε | false | ε | ε | true | true |
| true | ε | ε | true | ε | ε | false |
| false | \perp | false | \perp | \perp | true | true |
| true | \perp | \perp | true | \perp | \perp | false |
| ε | false | false | ε | ε | ε | ε |
| ε | true | ε | true | ε | true | ε |
| ε | ε | ε | ε | ε | ε | ε |
| ε | \perp | \perp | \perp | \perp | \perp | ε |
| \perp | false | false | \perp | \perp | \perp | \perp |
| \perp | true | \perp | true | \perp | true | \perp |
| \perp | \perp or ε | \perp | \perp | \perp | \perp | \perp |

► Legend: \perp is *invalid*, ε is *null*.

OCL-Std §A .2.1.3, Table A.2

OCL Style Guide

- ▶ Avoid **complex** navigation („Loose coupling“).
 - ▶ Otherwise changes in models break OCL constraints.
- ▶ Always choose **adequate context**.
- ▶ „Use of `allInstances ()` is **discouraged**“
- ▶ Split up invariants if possible.
- ▶ Consider defining **auxiliary operations** if expressions become too complex.

Summary

- ▶ OCL is a typed, state-free specification language which allows us to denote constraints on models.
- ▶ We can define our models much more precisely.
 - ▶ Ideally: no more natural language needed.
- ▶ OCL is part of the more „academic“ side of UML/SysML.
 - ▶ Tool support is not great, some tools ignore OCL, most tools at least type-check OCL, hardly any do proofs.
- ▶ However, in critical system development, the kind of specification that OCL allows is **essential**.
- ▶ Try it yourself: USE – Tool <http://useocl.sourceforge.net>
Martin Gogolla, Fabian Büttner, and Mark Richters. [USE: A UML-Based Specification Environment for Validating UML and OCL](#). Science of Computer Programming, 69:27-34, 2007.