Model-driven Engineering & Domain-Specific Languages

pictures: sxc.hu

UNIVERSITÉ CÔTE D'AZUR

Sébastien Mosser (UCA, I3S)

ENS Lyon, 14.09.2017

Sébastien Mosser

"geek, snowboarder & composition-driven guy"

- 12-...: Associate Professor, UNS
- 11-12: Research Scientist, SINTEF (NO)
- 10-11: Postdoc, Inria Lille Nord-Europe
- 2010: PhD Thesis, Software Composition

Sébastien Mosser

"geek, snowboarder & composition-driven guy"

- 12-...: Associate Professor, UNS
- 11-12: Research Scientist, SINTEF (NO)
- 10-11: Postdoc, Inria Lille Nord-Europe
- 2010: PhD Thesis, Software Composition MDE

Sébastien Mosser

"geek, snowboarder & composition-driven guy"

- 12-...: Associate Professor, UNS
- 11-12: Research Scientist, SINTEF (NO)
- 10-11: Postdoc, Inria Lille Nord-Europe MDE
- 2010: PhD Thesis, Software Composition MDE

Sébastien Mosser

"geek, snowboarder & composition-driven guy"

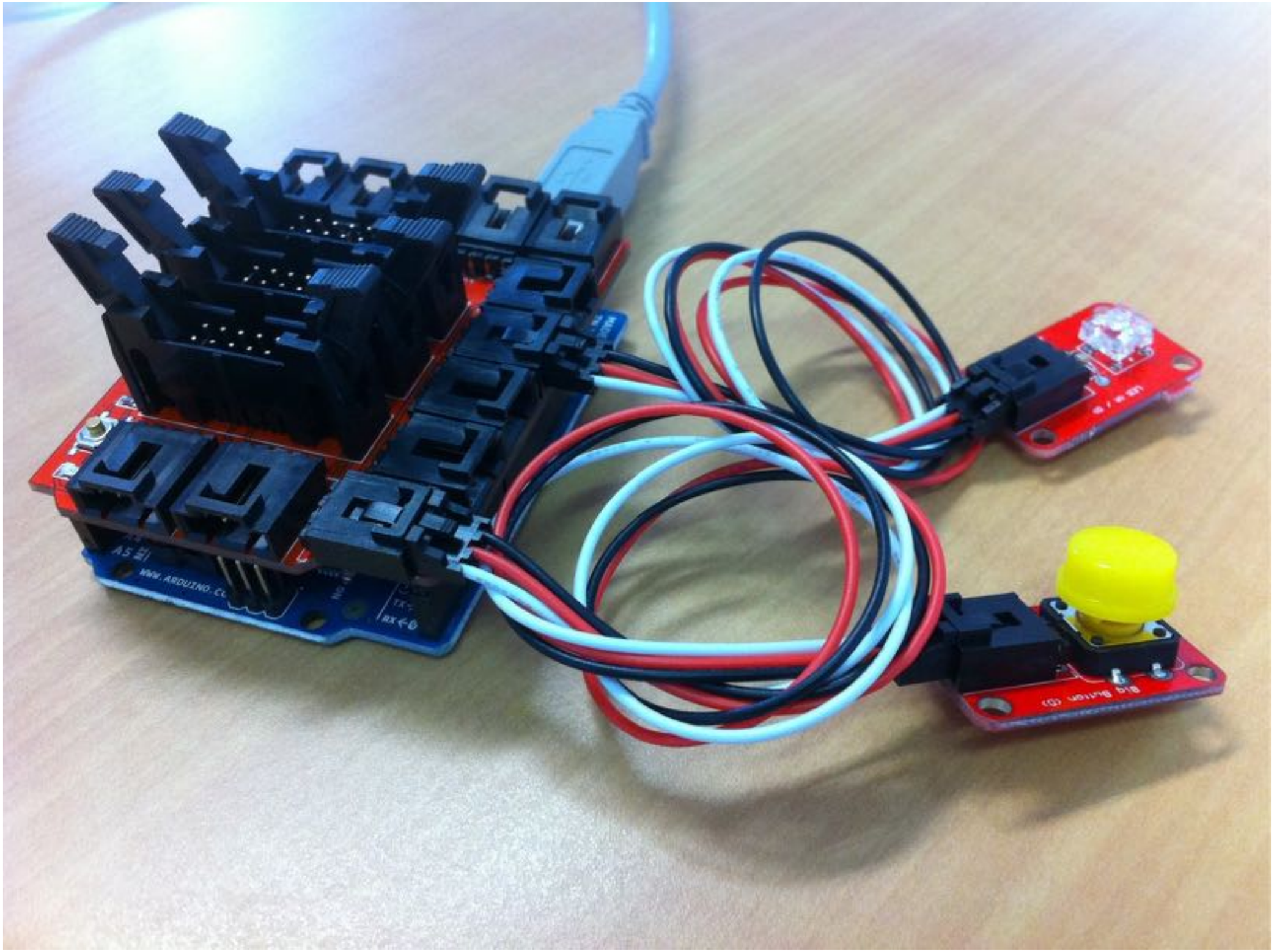
•12-...: Associate Professor, UNS

- 11-12: Research Scientist, SINTEF (NO) — MDE & DSL
- 10-11: Postdoc, Inria Lille Nord-Europe — MDE
- 2010: PhD Thesis, Software Composition — MDE

Sébastien Mosser

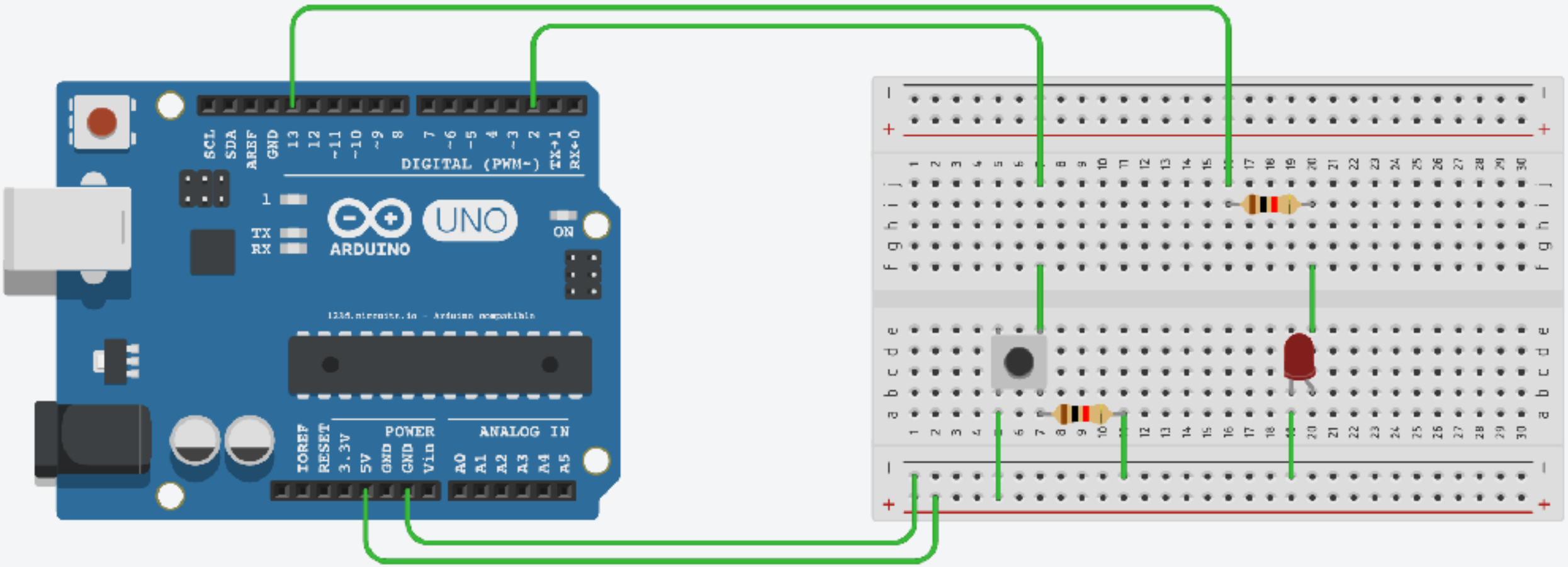
"geek, snowboarder & composition-driven guy"

- 12-...: Associate Professor, UNS — DSL
- 11-12: Research Scientist, SINTEF (NO) — MDE & DSL
- 10-11: Postdoc, Inria Lille Nord-Europe — MDE
- 2010: PhD Thesis, Software Composition — MDE



Sébastien Mosser, Philippe Collet, Mireille Blay-Fornarino. “Exploiting the Internet of Things to Teach Domain Specific Languages and Modeling” in Proceedings of the 10th Educators' Symposium at MODELS 2014 (EduSymp'14), ACM, IEEE, pages 1-10, Springer LNCS, Valencia, Spain, 29 september 2014

Arduino Simulator



https://circuits.io/circuits/2380885-big-red-button

```
int state = LOW; int prev = HIGH;
long t = 0; long debounce = 200;
void setup() {
  pinMode(8,INPUT);
  pinMode(11, OUTPUT);
}
void loop() {
 int reading = digitalRead(8);
 if (reading == HIGH && prev == LOW
    && millis() - t > debounce) {
   if (state == HIGH) {
     state = LOW;
   } else { state = HIGH; }
   time = millis();
 }
 digitalWrite(11, state);
 prev = reading;
}
```

```
int state = LOW; int prev = HIGH;
long t = 0; long debounce = 200;
void setup() {
  pinMode(8,INPUT);
  pinMode(11, OUTPUT);
}
void loop() {
 int reading = digitalRead(8)
 if (reading == HIGH && prev == LOW
     && millis() - t > debounce) {
   if (state == HIGH) {
    state = LOW;
   } else { state = HIGH; }
   time = millis();
 }
 digitalWrite(11, state);
 prev = reading;
}
```

Seriously?

```
app RedButton init_state : off {

  bricks :
    actuator red_led : 12
    sensor button : 9
  states :
    off {
      red_led <= low
      button is high => on
    }
    on {
      red_led <= high
      button is high => off
    }
}
```

Tom Guillermin @TomGuillermin · 9 déc.

@petitroll Je vois déjà un DSL pour définir les musiciens d'un groupe qui feraient une Jam session !

Afficher la conversation

http://www.dailymotion.com/video/x3b6q8g_toa-mata-band-le-premier-groupe-de-robots-lego_music?start=40

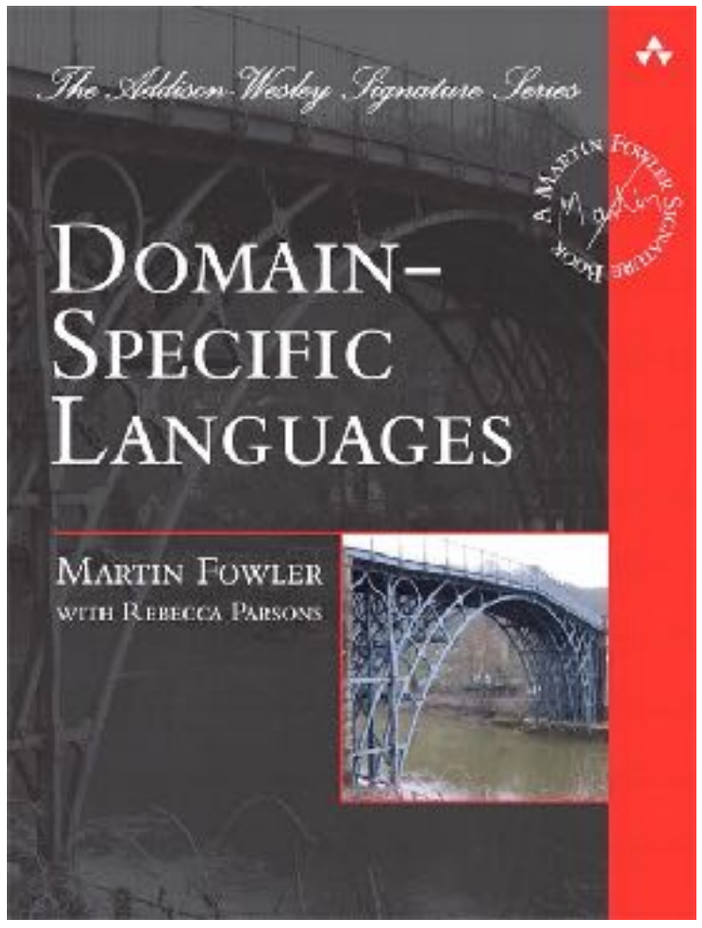
01:22 / 04:05 HD

«I see a **great future** for very **systematic** and **very modest** programming languages.»

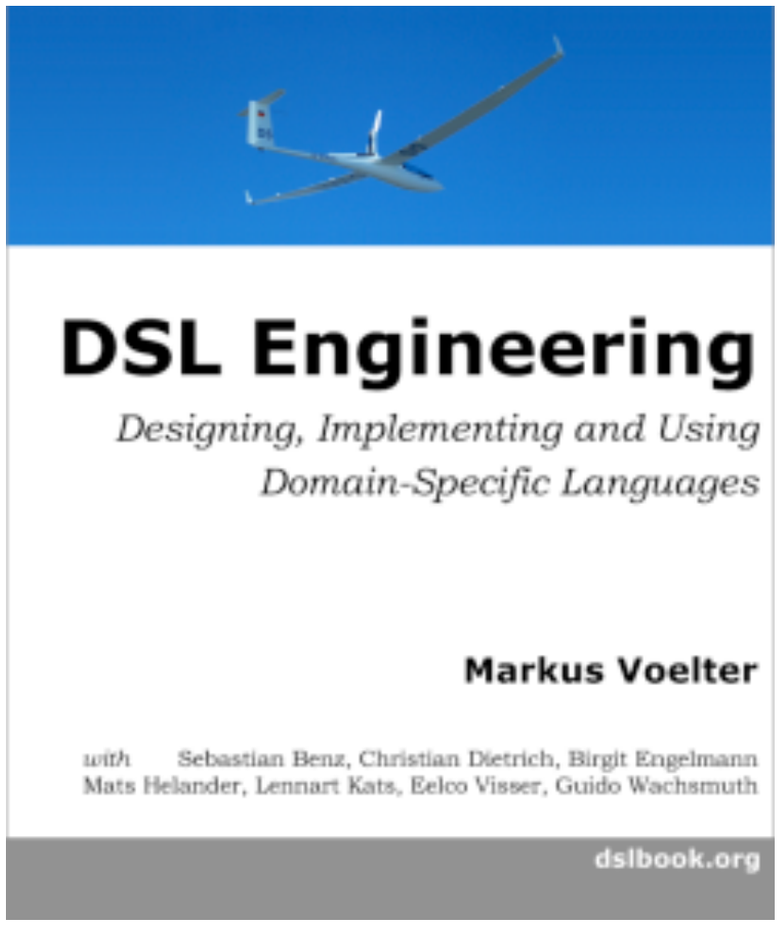
«I see a **great future** for very **systematic** and **very modest** programming languages.»

- E.W. Dijkstra (**1972**)

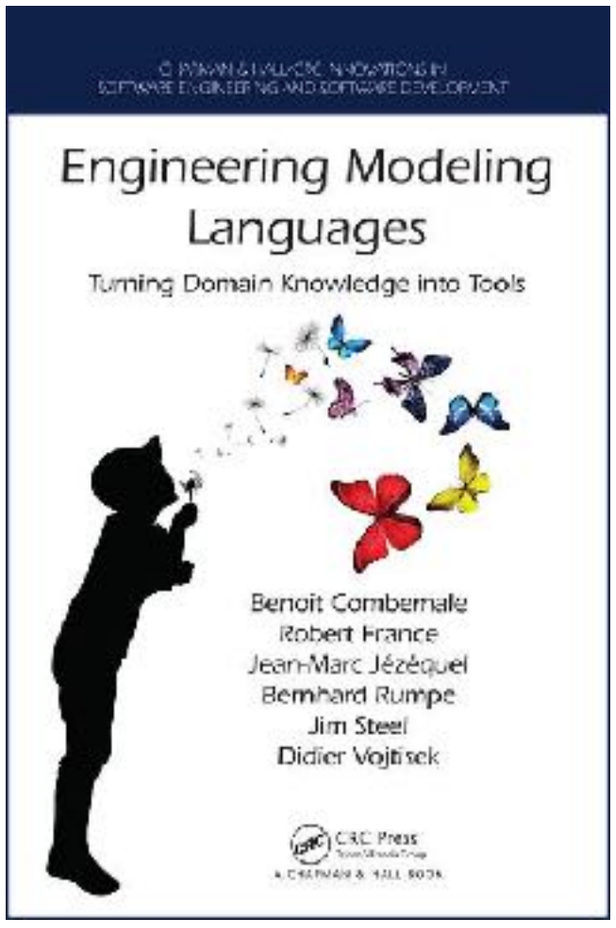
Bibliography



[Fowler]



[Voelter]



[Combemale et al]

Planning

- #37: Course introduction
- #38: Verification & Validation
- #39: Unsupervised lab

ArduinoML

- #40: Adv. code Generation
- #41: Variability modelling
- #42: Software Composition
- #48: Bibliography defence

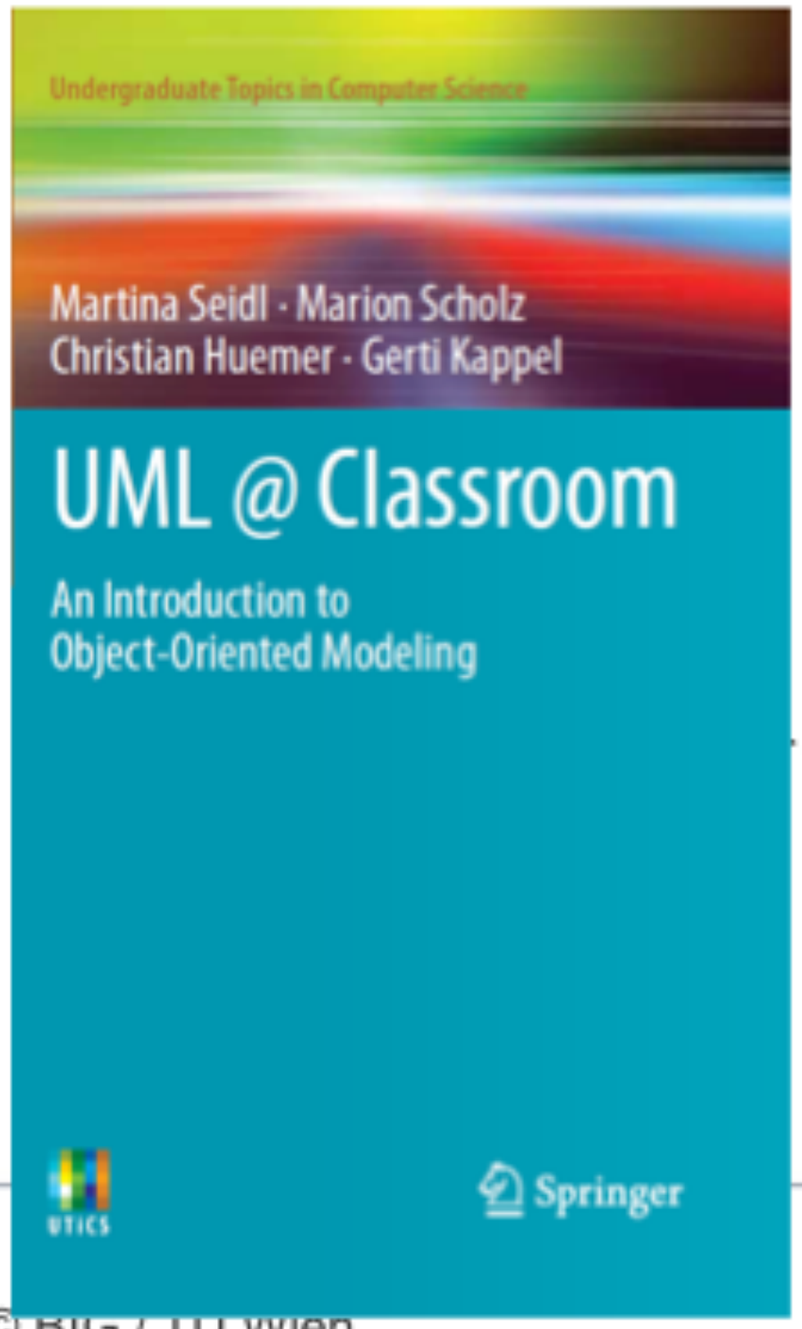
Scientific Workflows

(Object-oriented)

Models

Literature

- The lecture is based on the following book:



UML @ Classroom:
An Introduction to Object-Oriented Modeling
Martina Seidl, Marion Scholz, Christian Huemer and Gerti Kappel

Springer Publishing, 2015

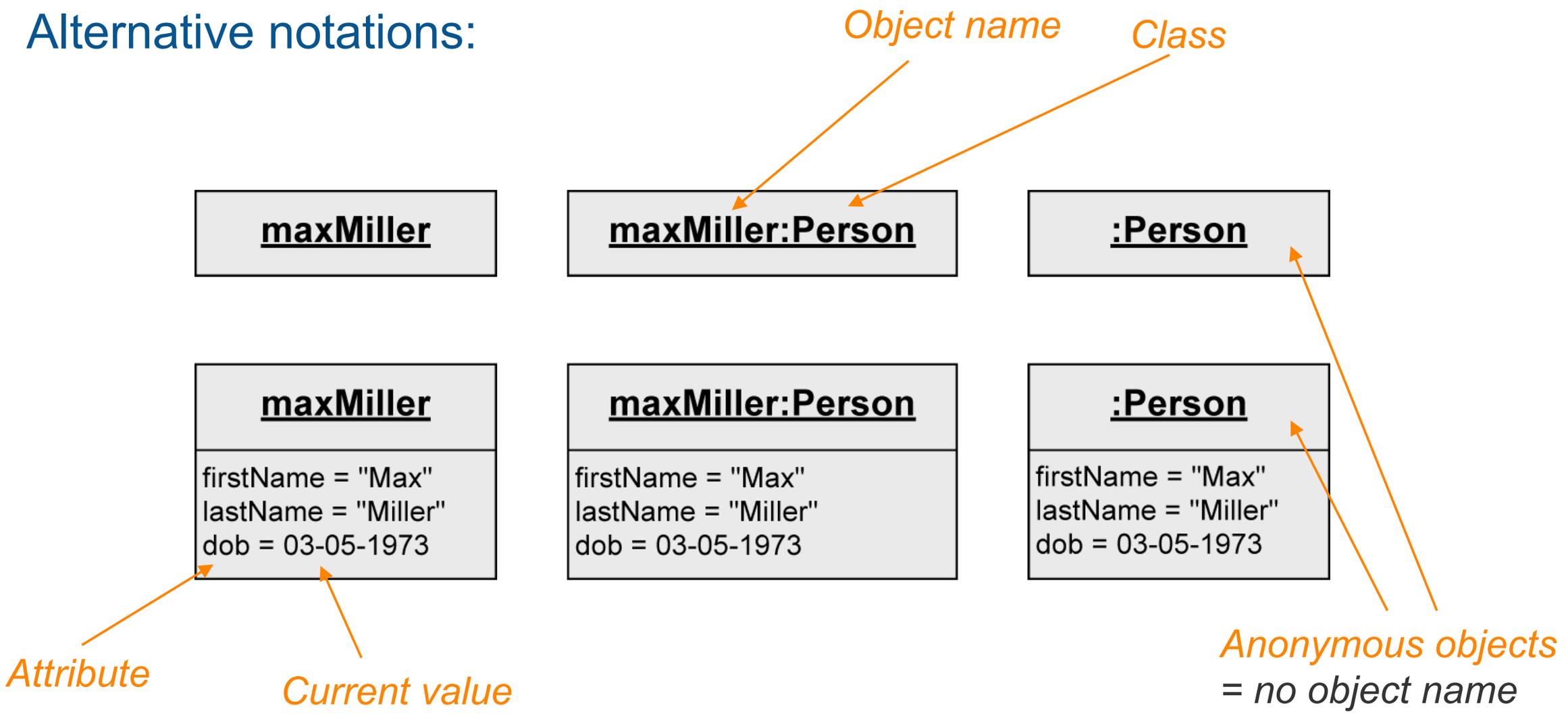
ISBN 3319127411

- Use Case Diagram
- **Structure Modeling**
- State Machine Diagram
- Sequence Diagram
- Activity Diagram

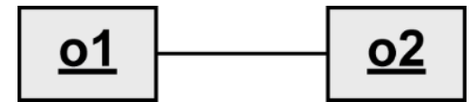
o:C

Object

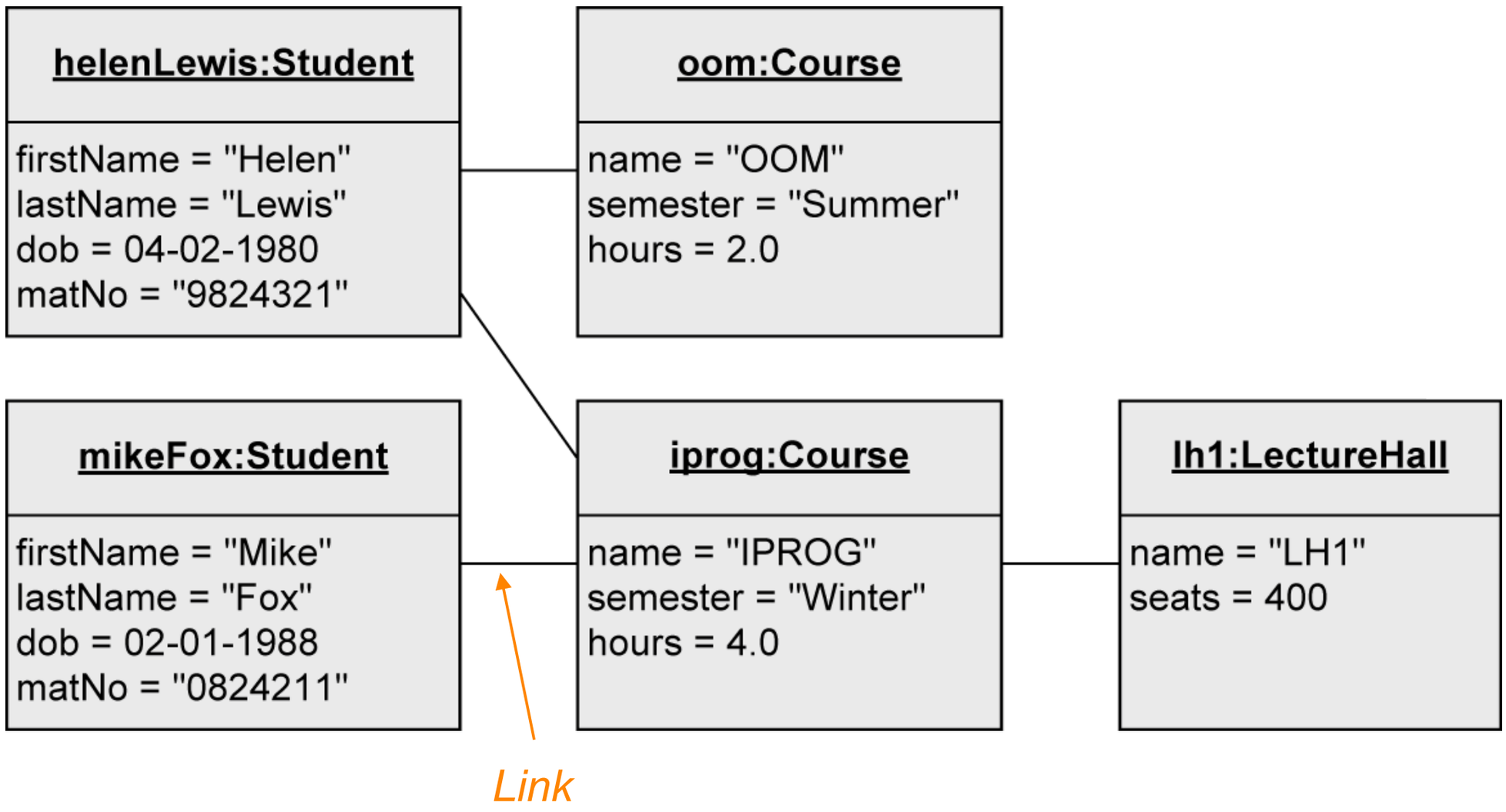
- Individuals of a system
- Alternative notations:



Object Diagram



- Objects of a system and their relationships (links)
- Snapshot of objects at a specific moment in time



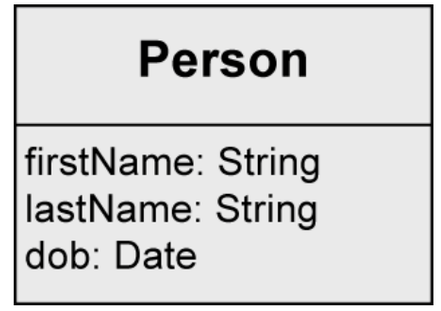
From Object to Class

- Individuals of a system often have identical characteristics and behavior
- A class is a construction plan for a set of similar objects of a system

- Objects are instances of classes

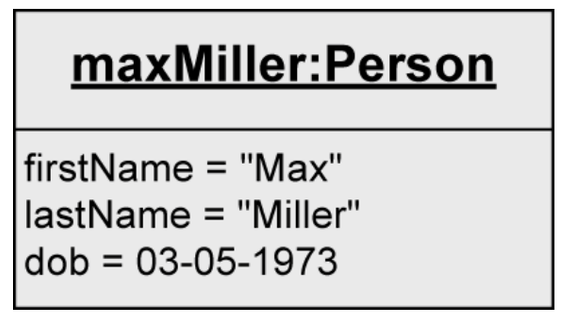
- Attributes: structural characteristics of a class
  - Different value for each instance (= object)

Class

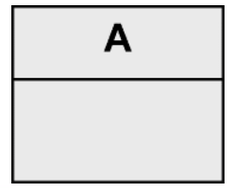


- Operations: behavior of a class
  - Identical for all objects of a class
    → not depicted in object diagram

Object of that class



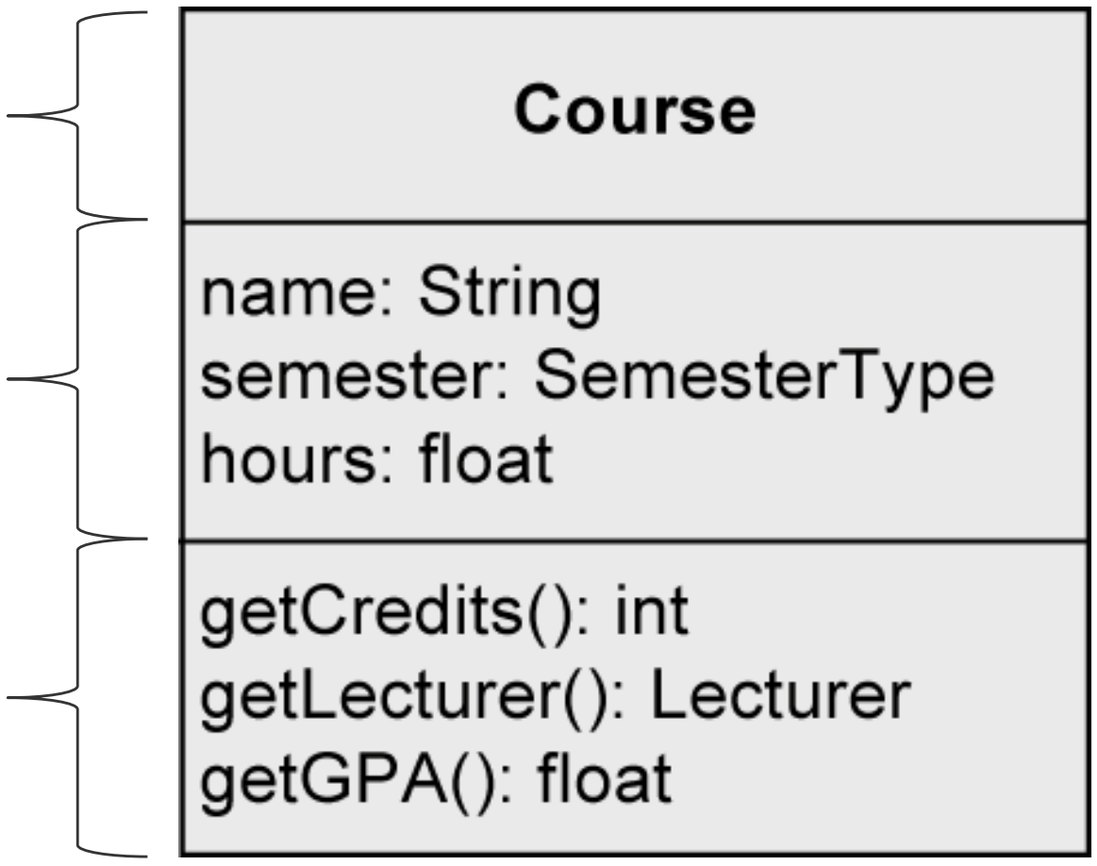
Class



Class name

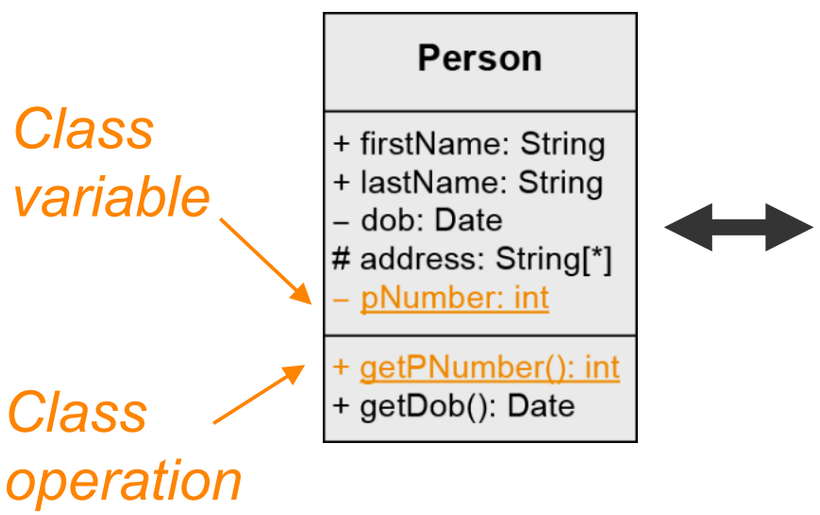
Attributes

Operations



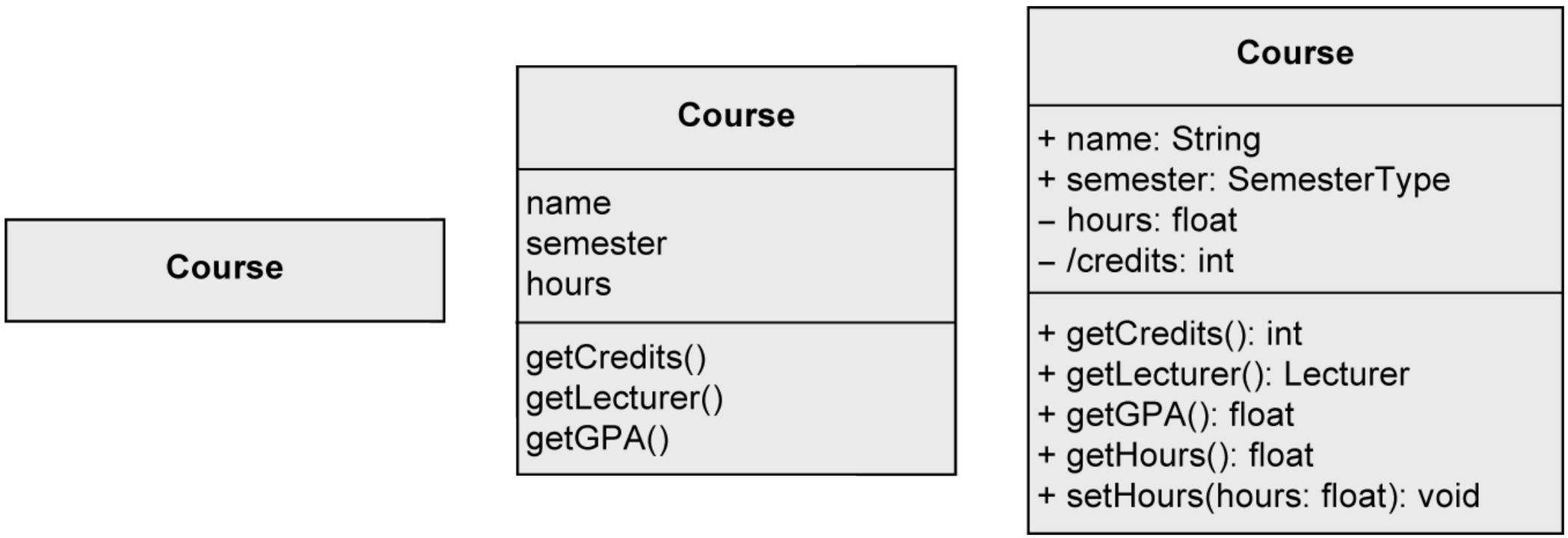
Class Variable and Class Operation

- Instance variable (= instance attribute): attributes defined on instance level
- Class variable (= class attribute, static attribute)
  - Defined only once per class, i.e., shared by all instances of the class
  - E.g. counters for the number of instances of a class, constants, etc.
- Class operation (= static operation)
  - Can be used if no instance of the corresponding class was created
  - E.g. constructors, counting operations, math. functions (sin(x)), etc.
- Notation: underlining name of class variable / class operation



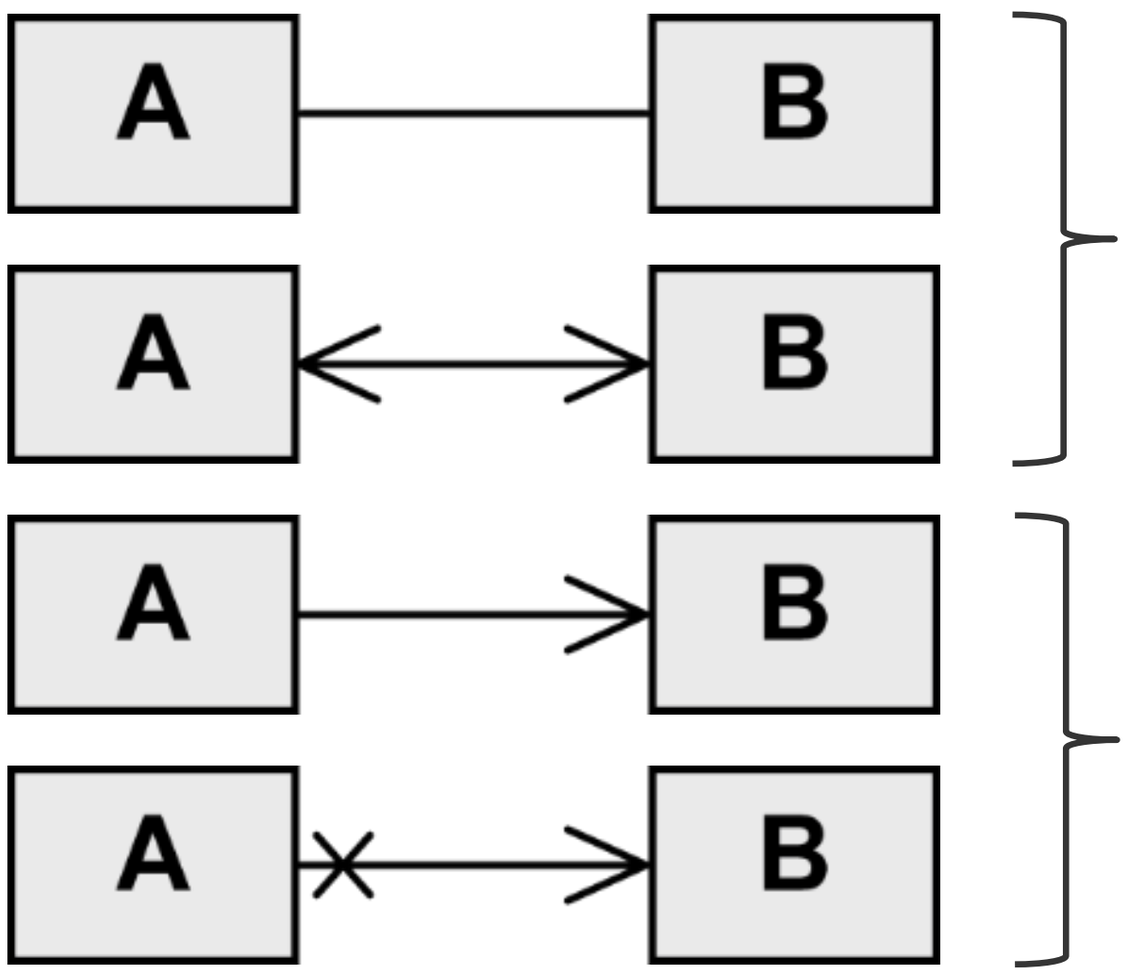

```
class Person {

   public String firstName;
   public String lastName;
   private Date dob;
   protected String[] address;
   private static int pNumber;
   public static int getPNumber() {…}
   public Date getDob() {…}
}
```

Specification of Classes: Different Levels of Detail

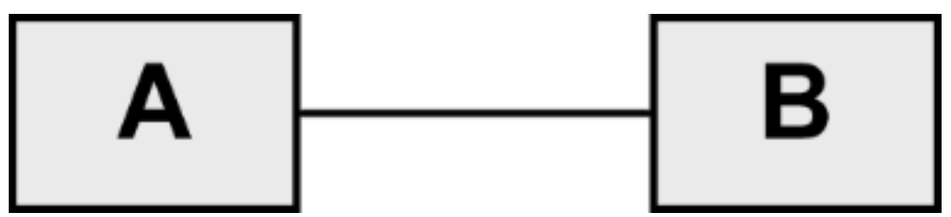


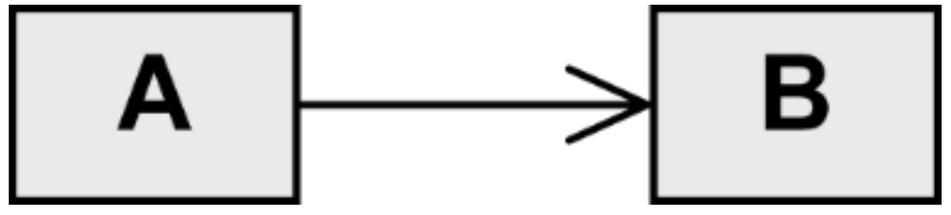
Navigability – UML Standard vs. Best Practice

UML standard



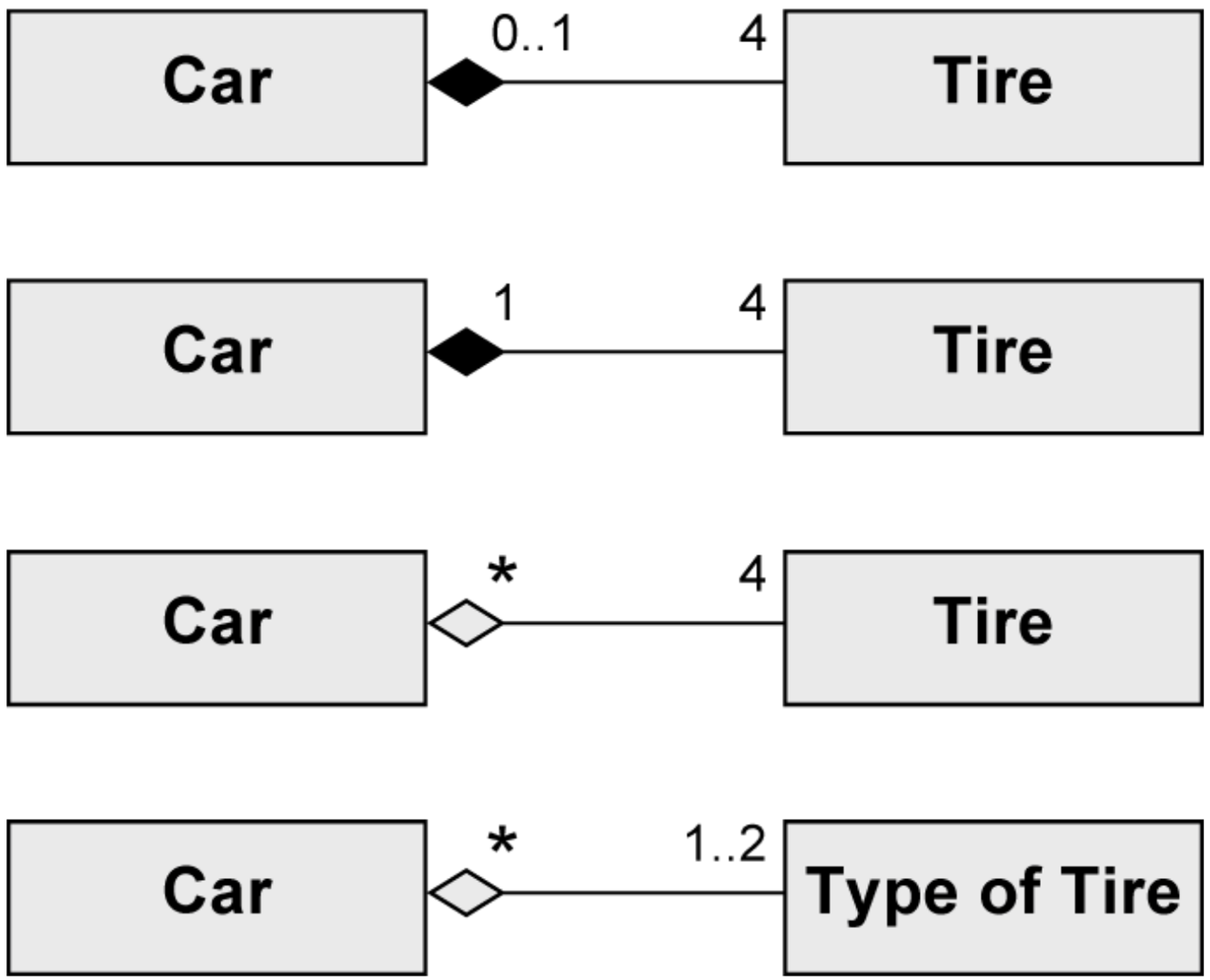
Best practice





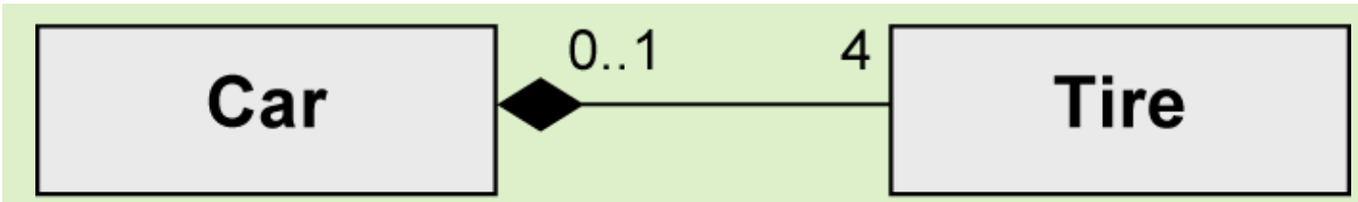
Shared Aggregation and Composition

- Which model applies?



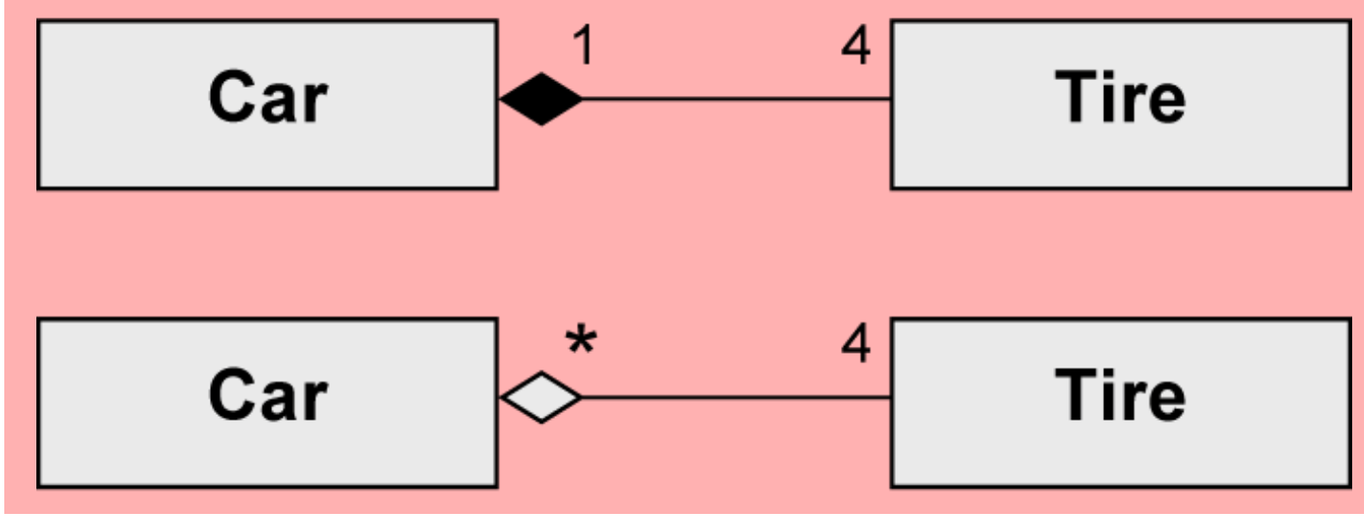
Shared Aggregation and Composition

- Which model applys?



A `Tire` can exist without a `Car`. A `Tire` belongs to one `Car` at most.

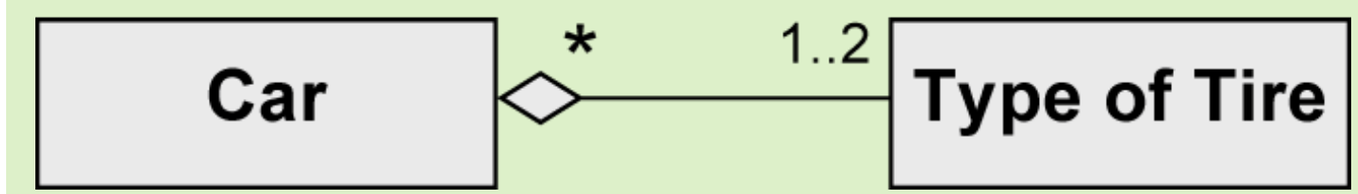
Yes



A `Tire` cannot exist without a `Car`.

No

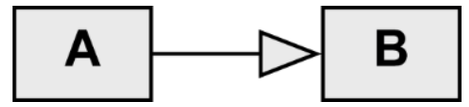
A `Tire` can belong to multiple `Cars`



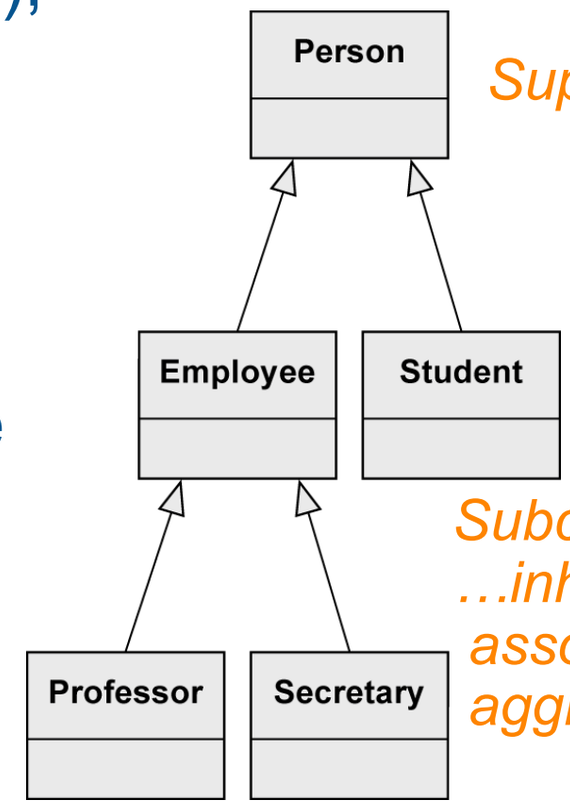
A `Car` has one or two types of `Tires`. Several `Cars` may have the same Type of `Tires`.

Yes

Generalization



- Characteristics (attributes and operations), associations, and aggregations that are specified for a general class (superclass) are passed on to its subclasses.
- Every instance of a subclass is simultaneously an indirect instance of the superclass.
- Subclass inherits all characteristics, associations, and aggregations of the superclass except private ones.
- Subclass may have further characteristics, associations, and aggregations.
- Generalizations are transitive.



Superclass

Subclasses …inherit characteristics, associations, and aggregations

A `Secretary` is an `Employee` and a `Person`

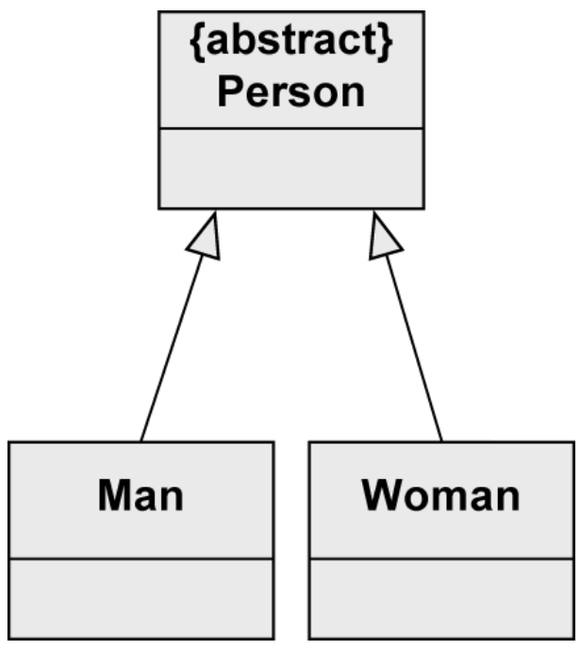
Generalization – Abstract Class

{abstract}
A

- Used to highlight common characteristics of their subclasses.
- Used to ensure that there are no direct instances of the superclass.
- Only its non-abstract subclasses can be instantiated.
- Useful in the context of generalization relationships.
- Notation: keyword **`{abstract}`** or class name in italic font.

{abstract}
Person

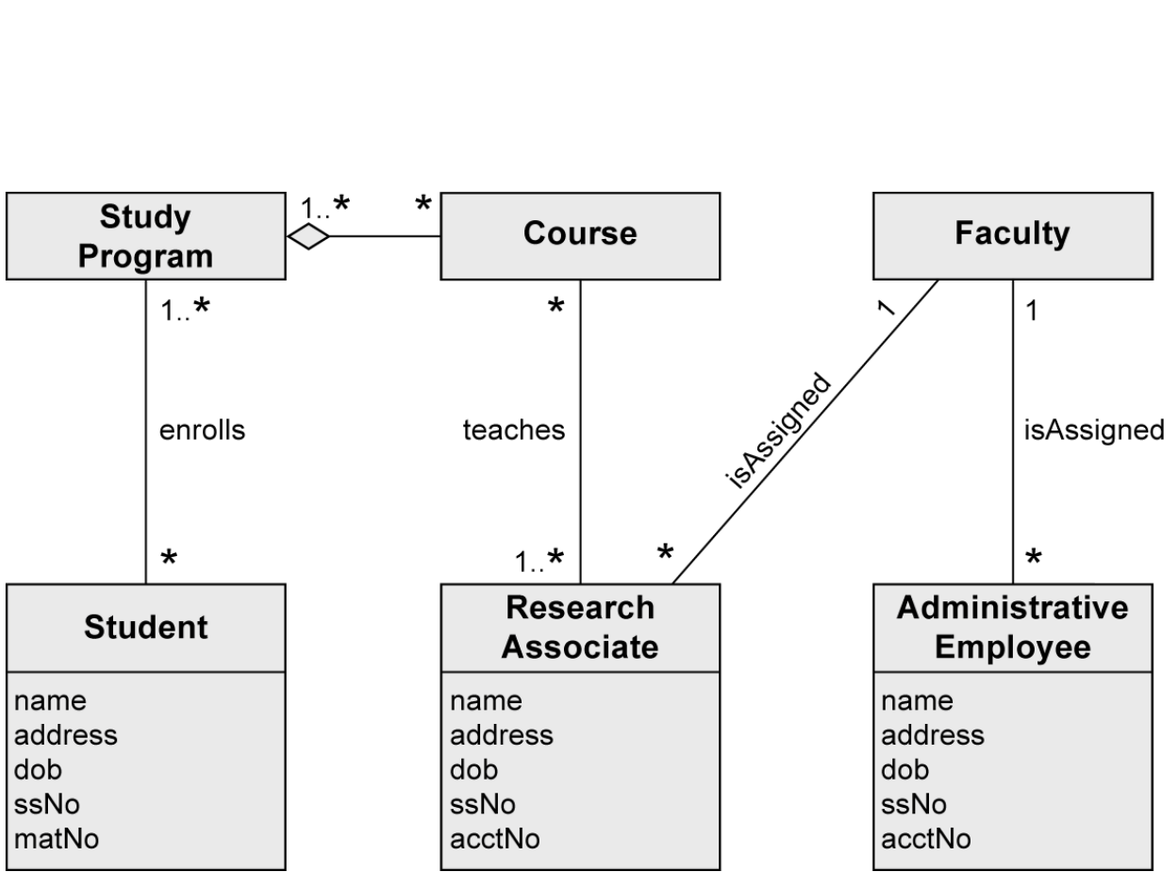
Person

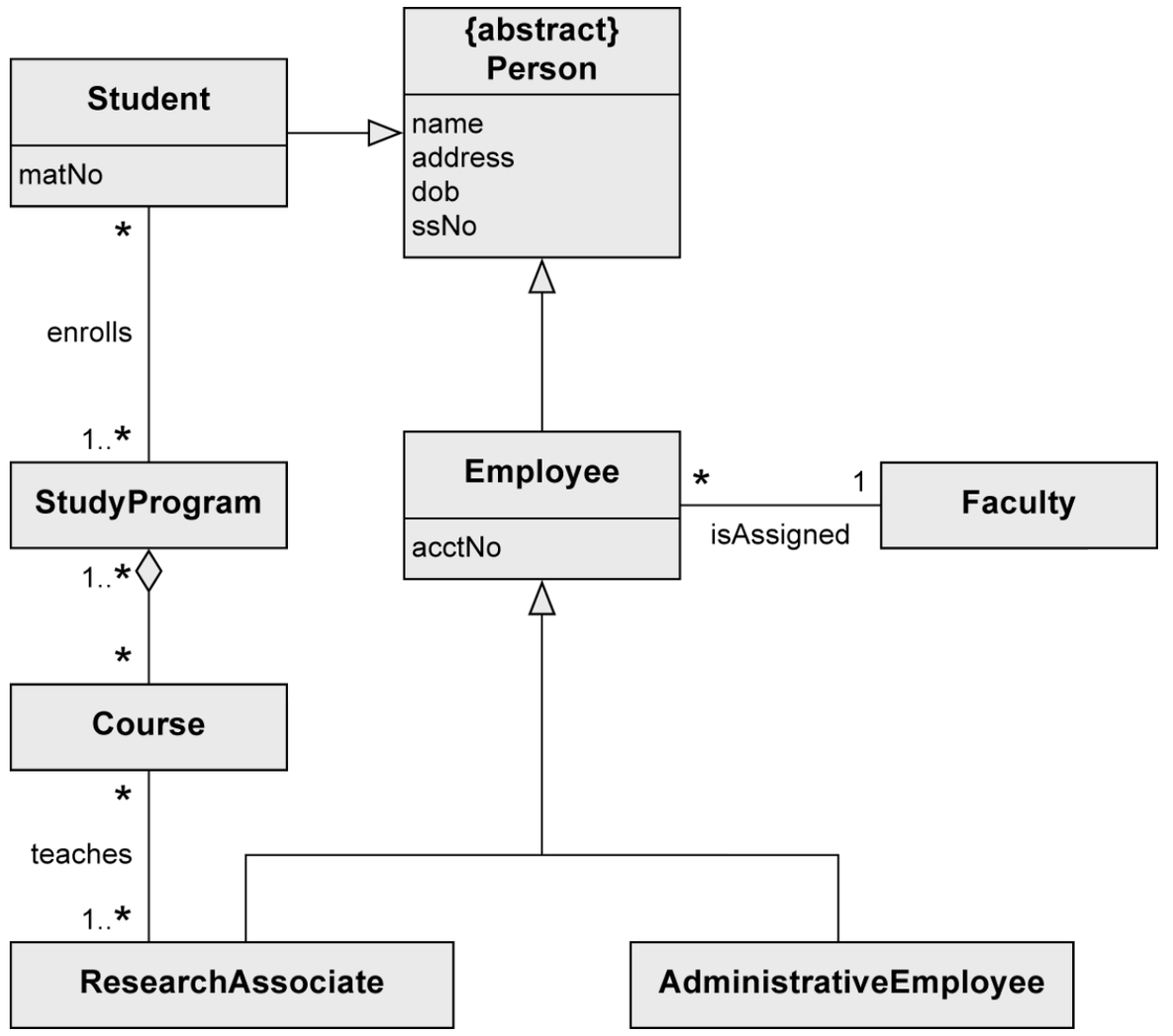


No `Person`-object possible

Two types of `Person`: `Man` and `Woman`

With and Without Generalization



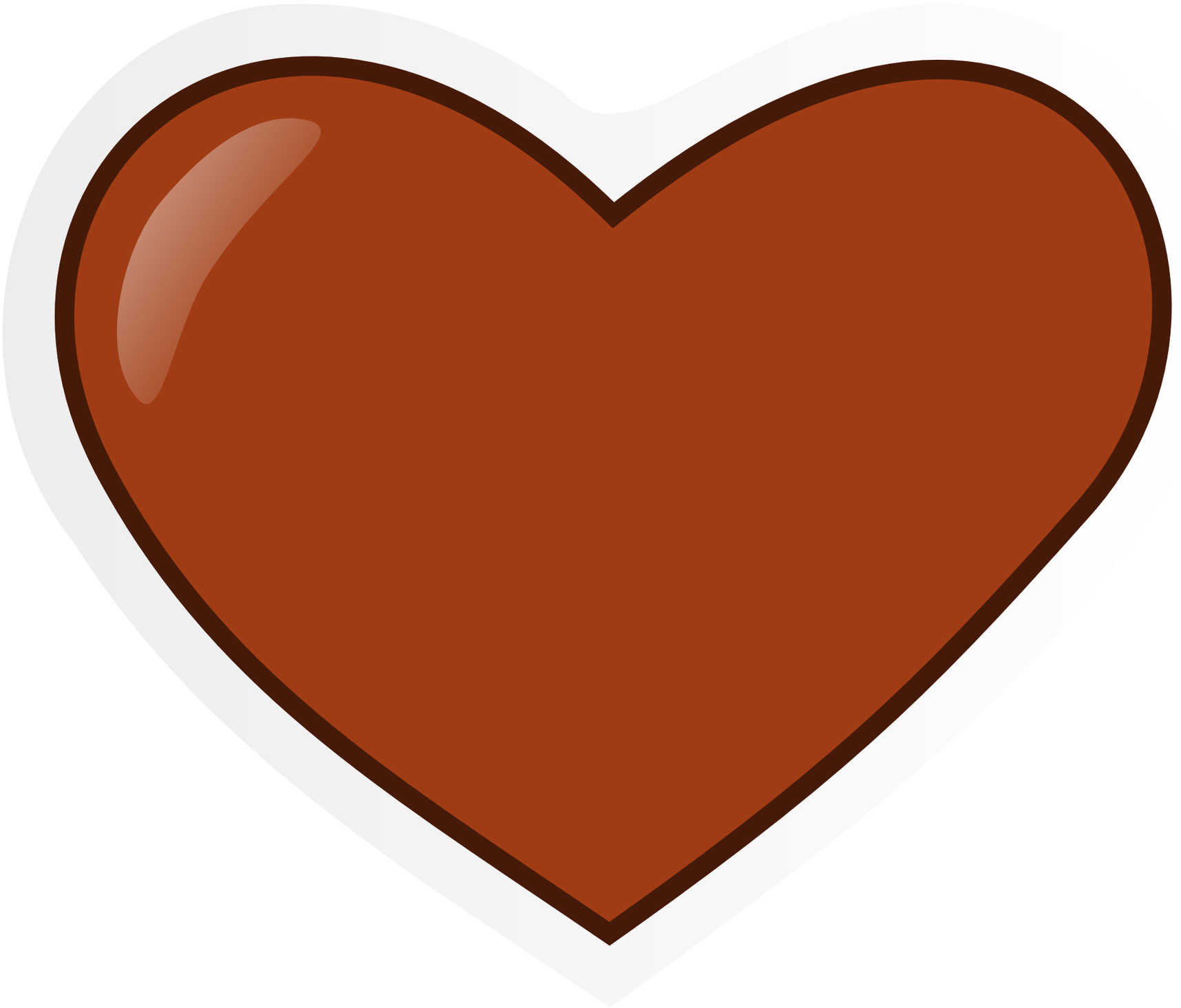


Model-driven Engineering

“

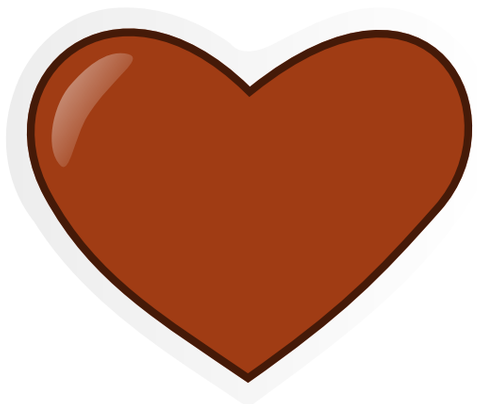
Why do we write programs, instead of writing programs that will write programs on our behalf?

– JM Jezequel



I love model-driven engineering

«is modelled by»



A sentence *contains* ***a subject***, ***a verb***, *and* ***a complement***.

«conforms to»

I love model-driven engineering

«is modelled by»

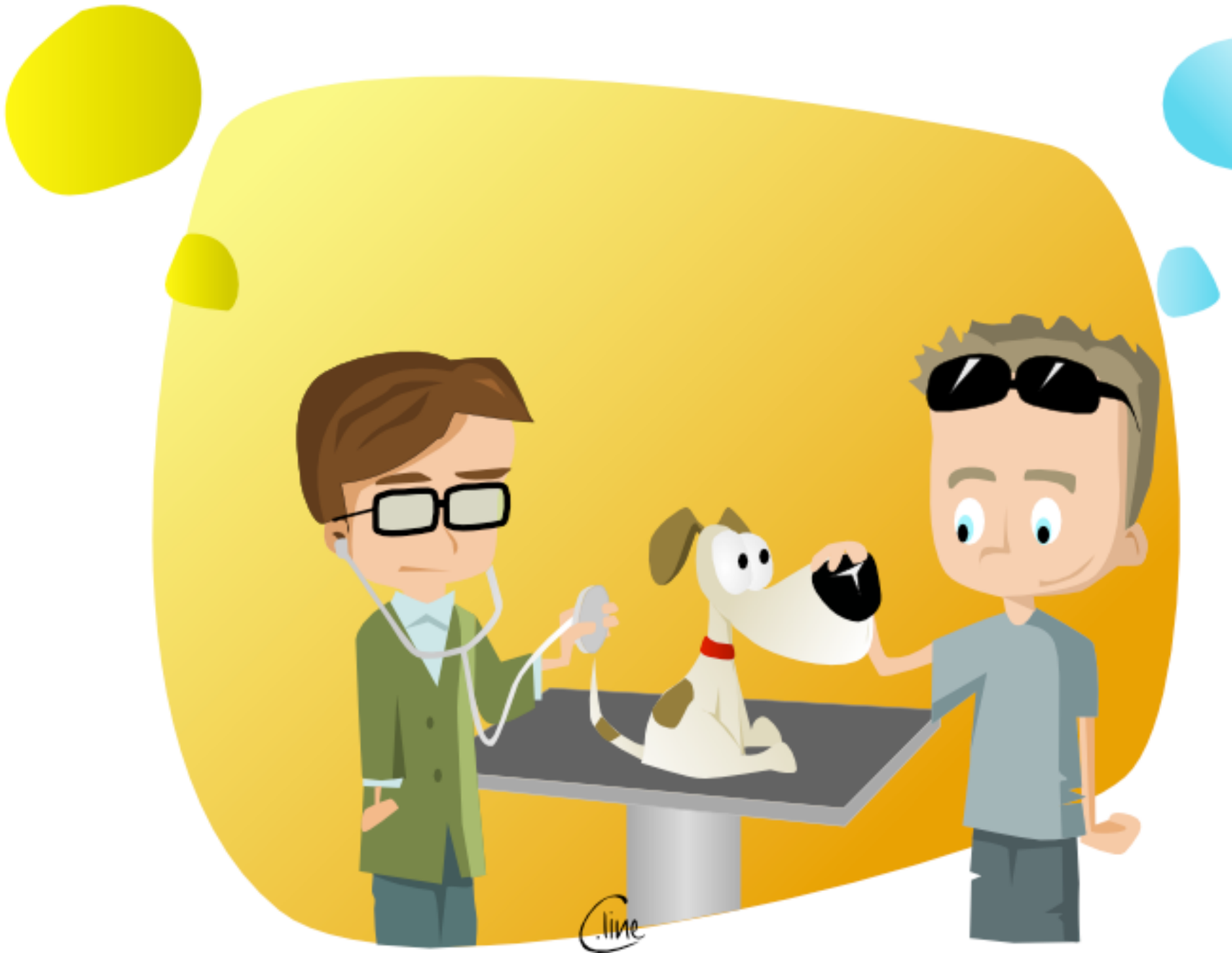
A sentence *contains* ***a subject***,
a verb, *and* ***a complement***.

«conforms to»

A sentence *contains* ***a subject***,
a verb, *and* ***a complement***.

«conforms to»

I love model-driven engineering



But, said Alice,

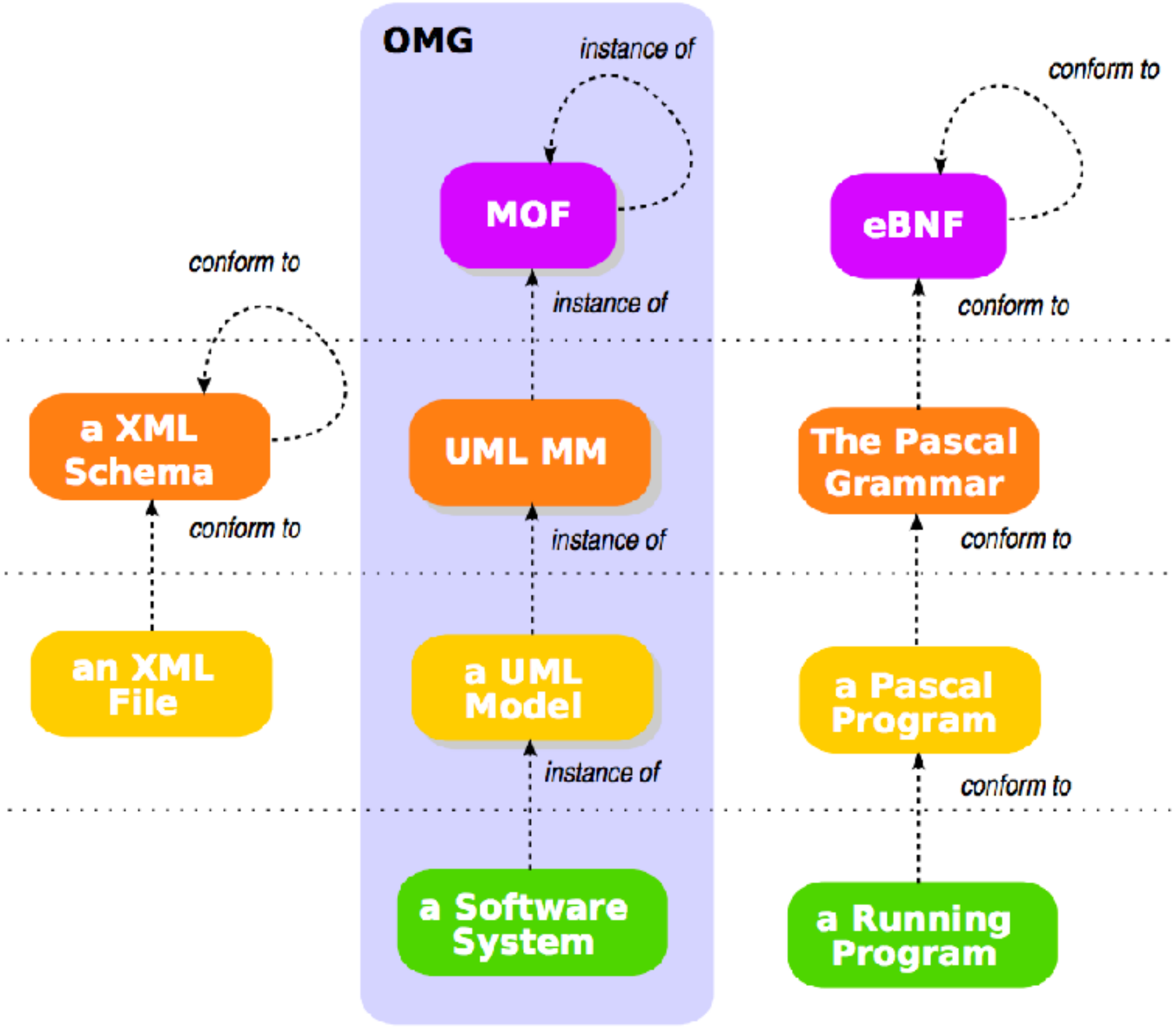
if the world has absolutely

No Sense,

who's stopping

us from

Inventing One?

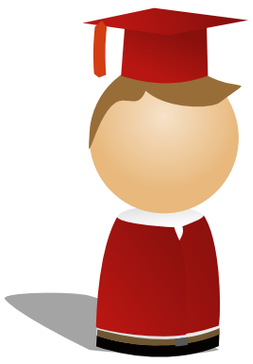

OMG
instance of
MOF
conform to
eBNF
conform to
instance of
conform to
a XML Schema
UML MM
The Pascal Grammar
conform to
instance of
conform to
an XML File
a UML Model
a Pascal Program
instance of
conform to
a Software System
a Running Program

Actors & Artifacts

Meta-modeling expert

Meta-meta-model

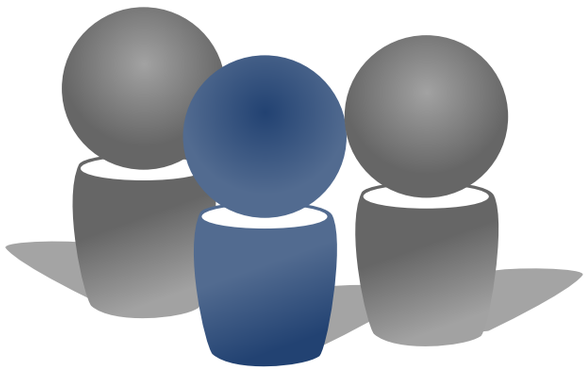
conforms to



Domain expert

Meta-model

conforms to



Users

Model

Capturing / Tooling Domain knowledge

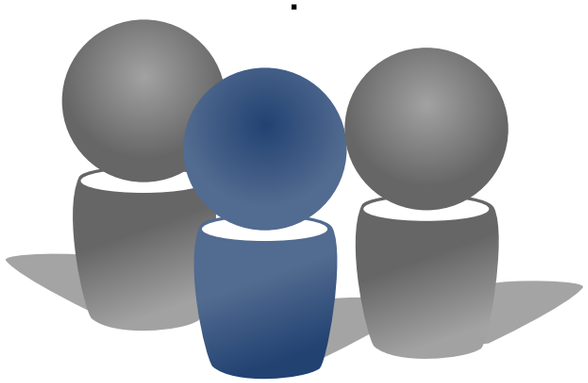
Meta-model relies on **Tool**

conforms to

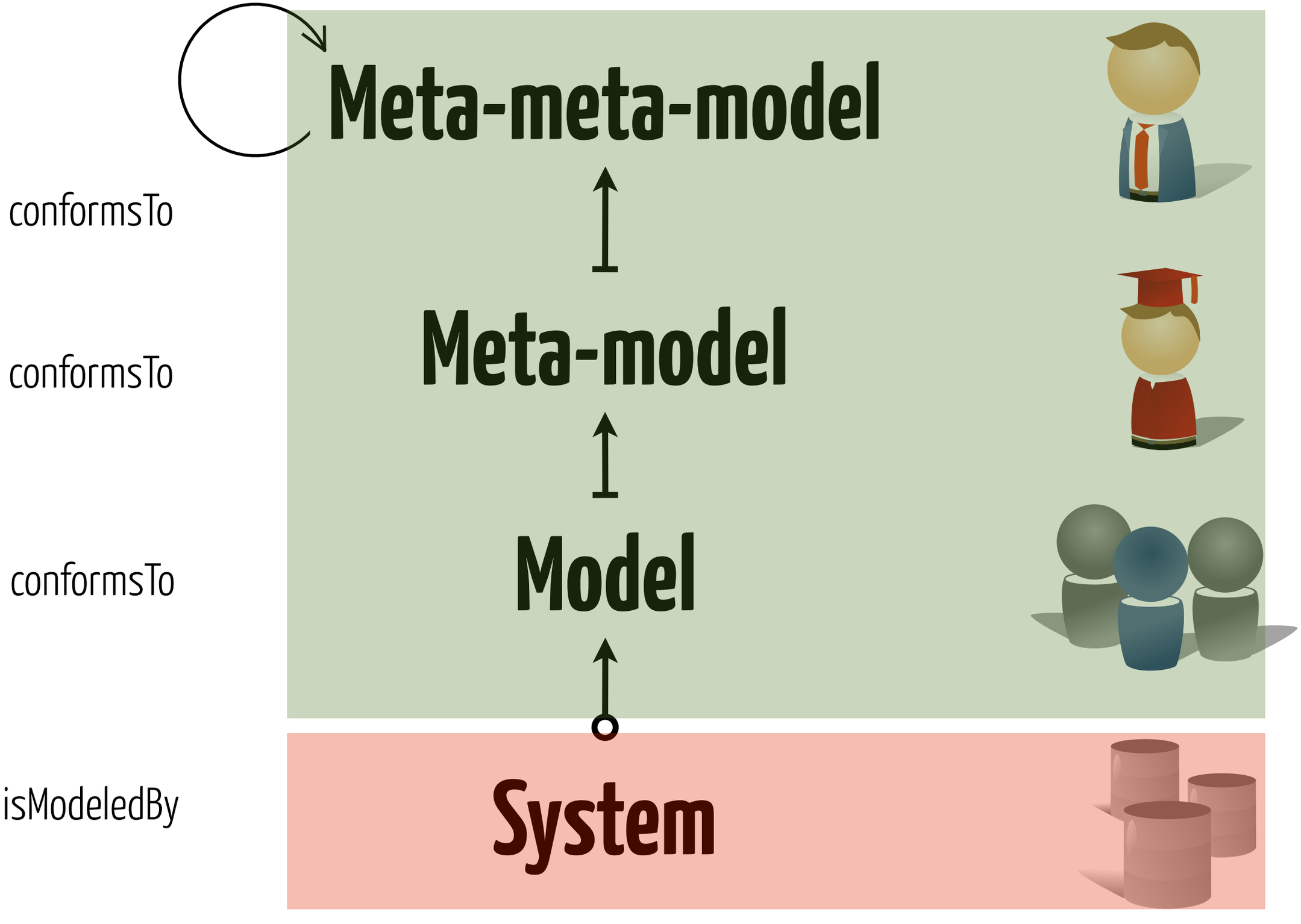
works on

uses

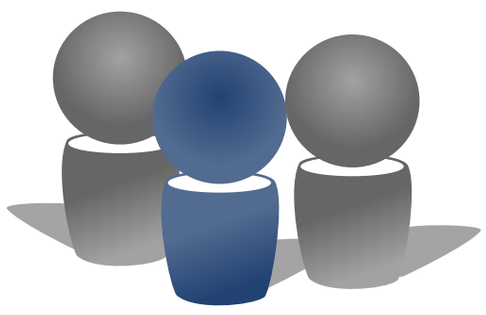
Model designs



The abstraction journey

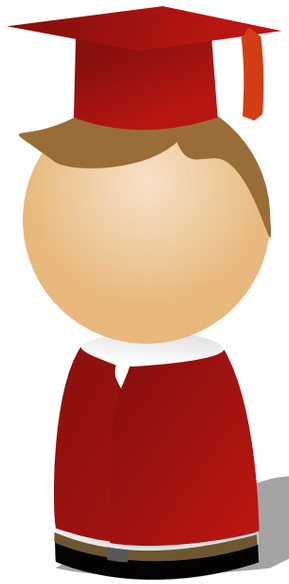


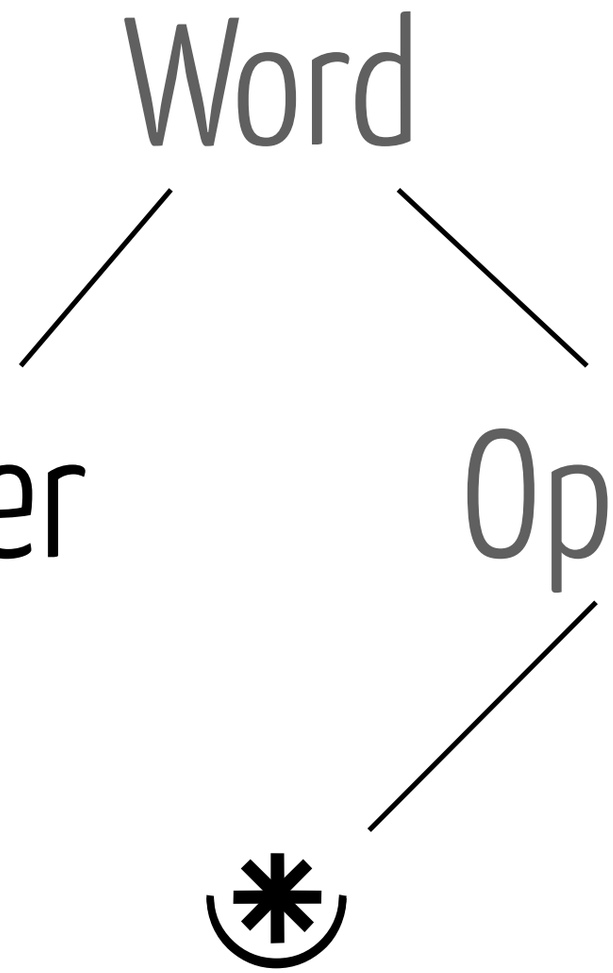
Example: Handling **Regular Expressions**

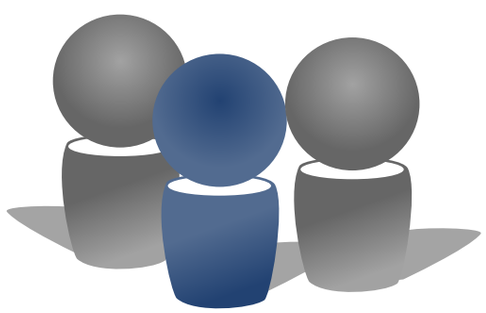


——((a⊙b)*⊙c+)*

Example: Handling **Regular Expressions**



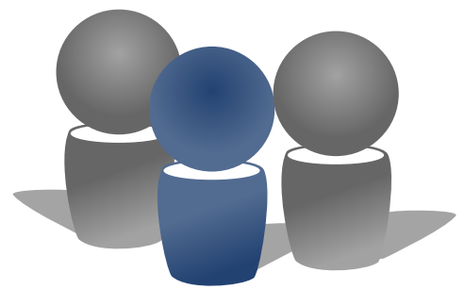




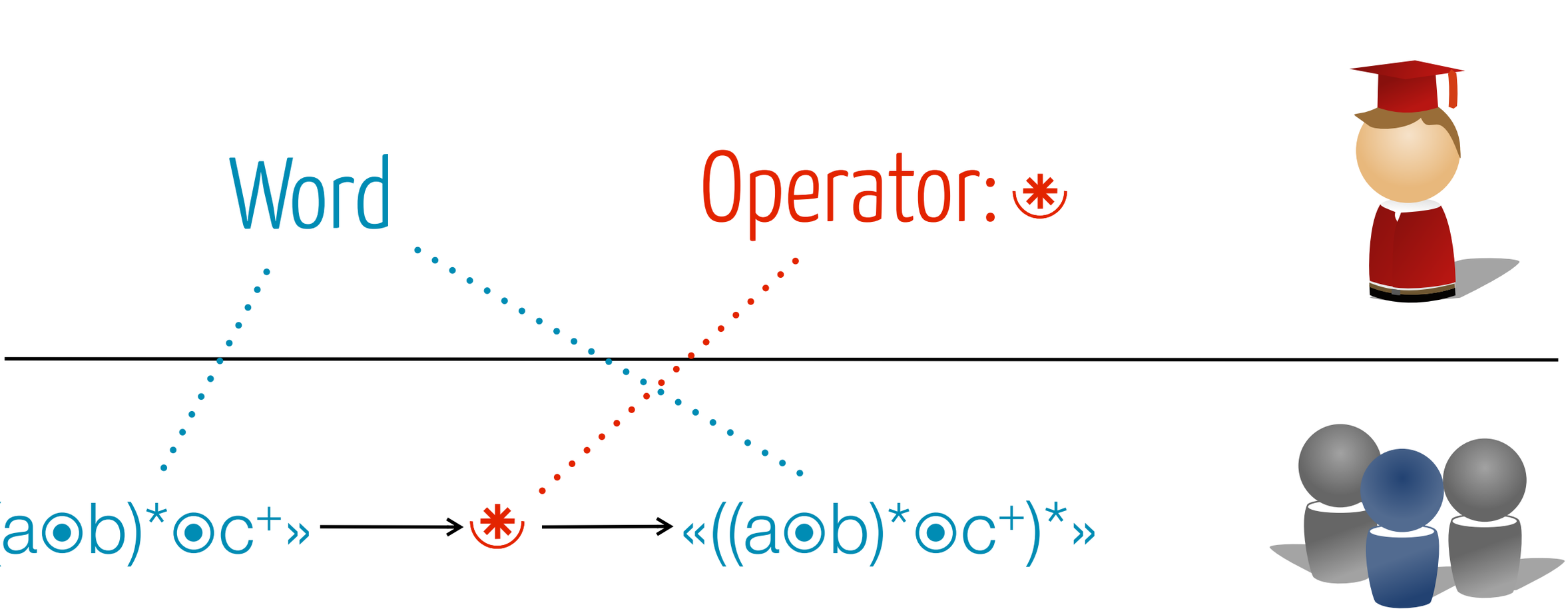
$((a \odot b)^* \odot c^+)^*$

Requirements #1: Modeling **concepts**

«(a⊙b)*⊙c⁺» → * → «((a⊙b)*⊙c⁺)*»



Requirements #1: Modeling **concepts**



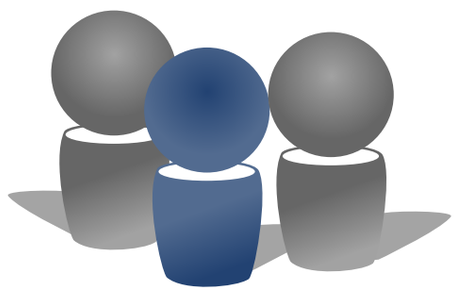
Requirements #1: Modeling **concepts**

Concept

Word

Operator: ⊛

«(a⊙b)*⊙c⁺» ⟶ ⊛ ⟶ «((a⊙b)*⊙c⁺)*»

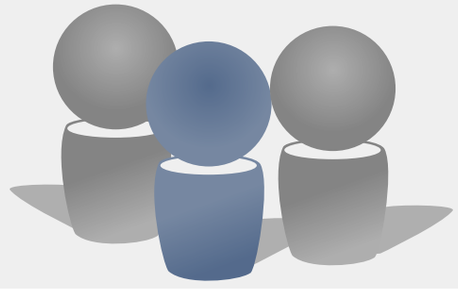


Starting at the beginning …

```
abstract public class NamedElement {
   protected String name;
}
public class Concept extends NamedElement {
   public Concept(String n) { this.name = n; }
}
```

```
Concept word = new Concept("Word");
Concept star = new Concept("KleenStar");
```

KleenStar(**Word**(‘**A**’)) ➜ A*

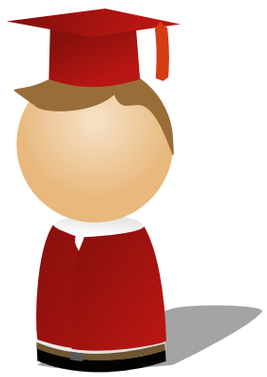


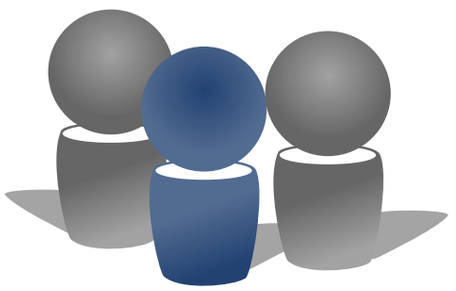
Requirements #2: Holding **Properties**

Concept

Letter

«a»





Requirements #2: Holding **Properties**

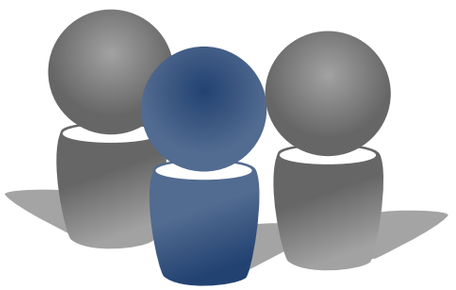
Concept

Letter

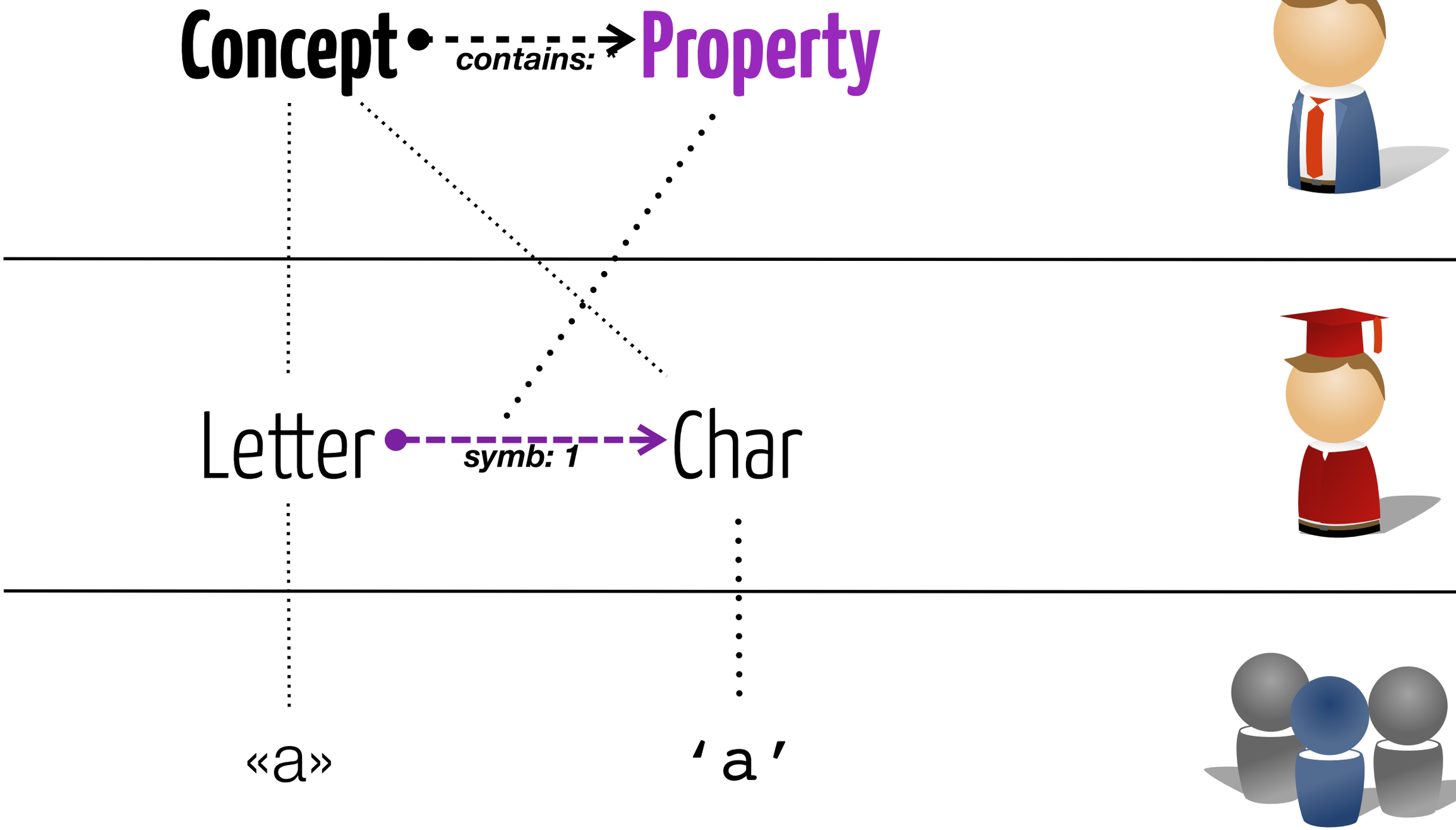
«a»

Char

'a'



Requirements #2: Holding **Properties**

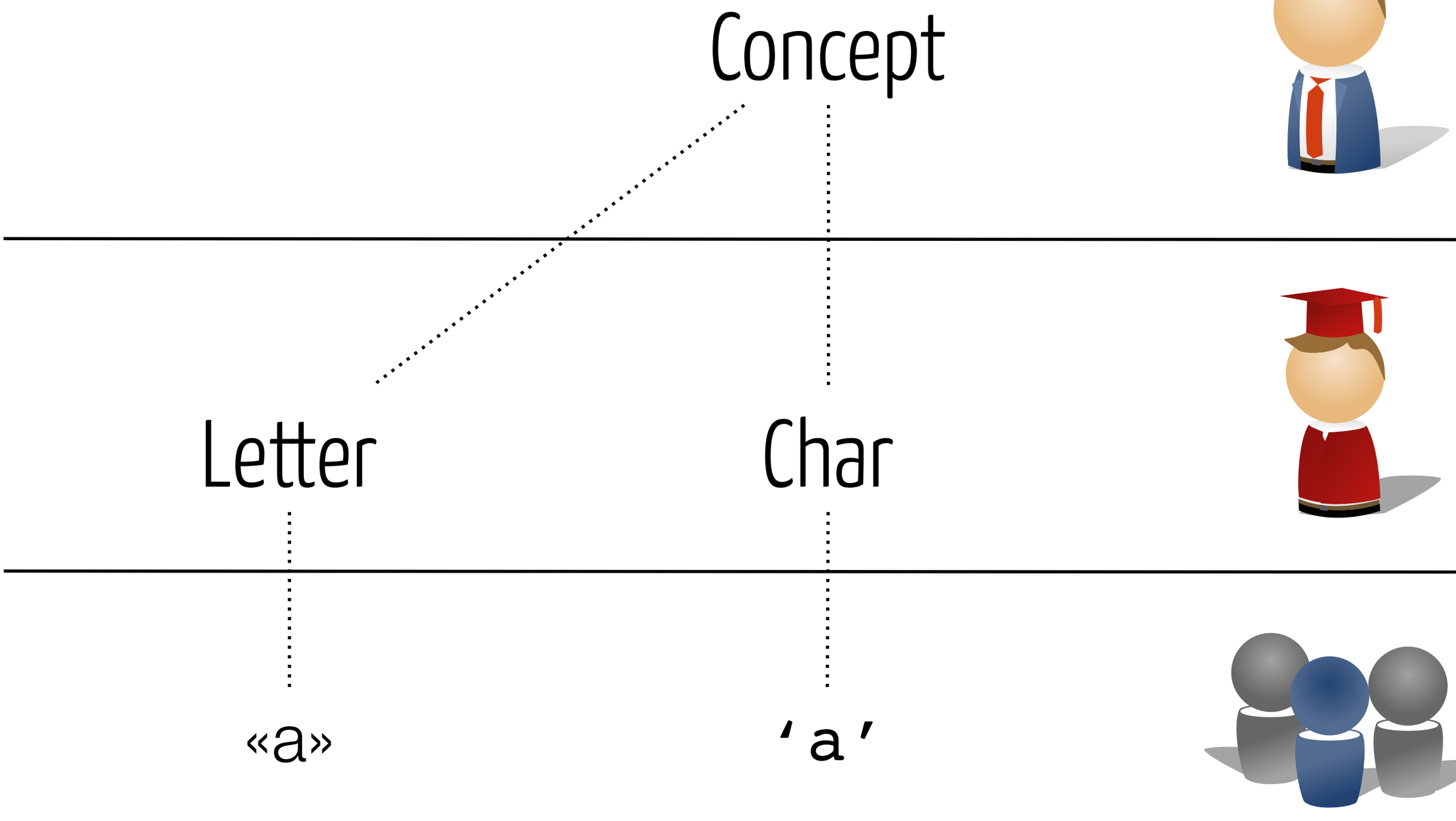


Implementing Properties

```
public class Property extends NamedElement {
  private Concept kind; private int mult;

  public Property(String n, Concept k, int m) {
    this.name = n; this.kind = k; this.mult = m;
  }
}
```

```
Concept ch = new Concept("Char");
Concept letter = new Concept("Letter");
letter.addProperty(new Property("symb", ch, 1));
```

Requirements #3: **Scoping** Concepts



Requirements #3: **Scoping** Concepts

Concept

Word_Lib

contains

Letter

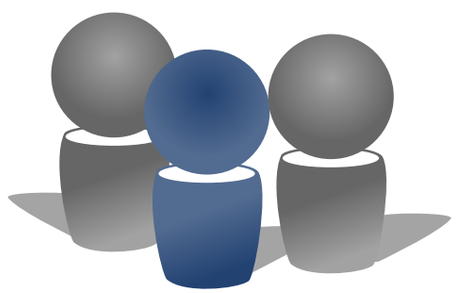
«a»

Std_Lib

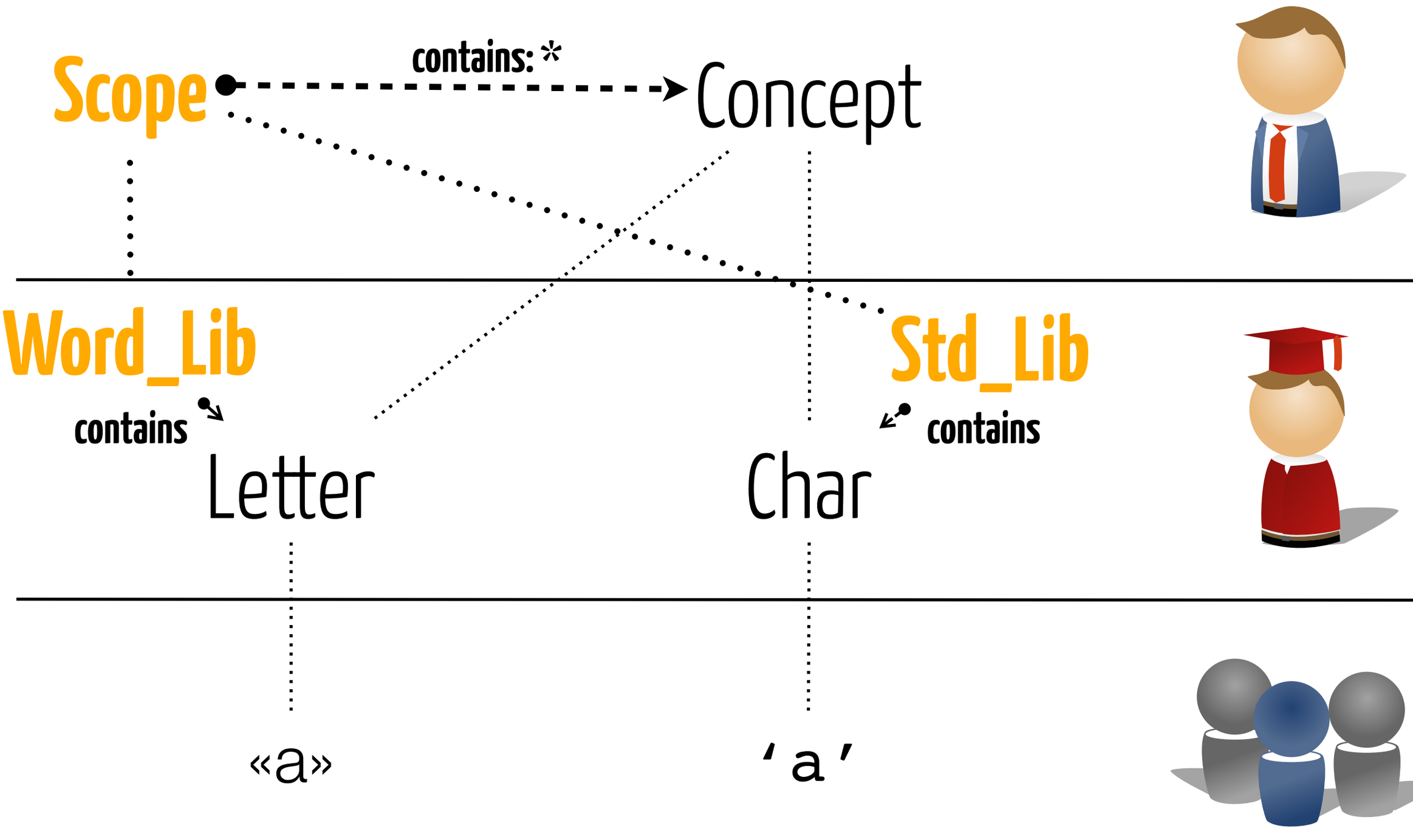
contains

Char

'a'



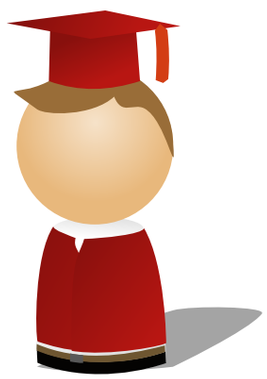
Requirements #3: **Scoping** Concepts



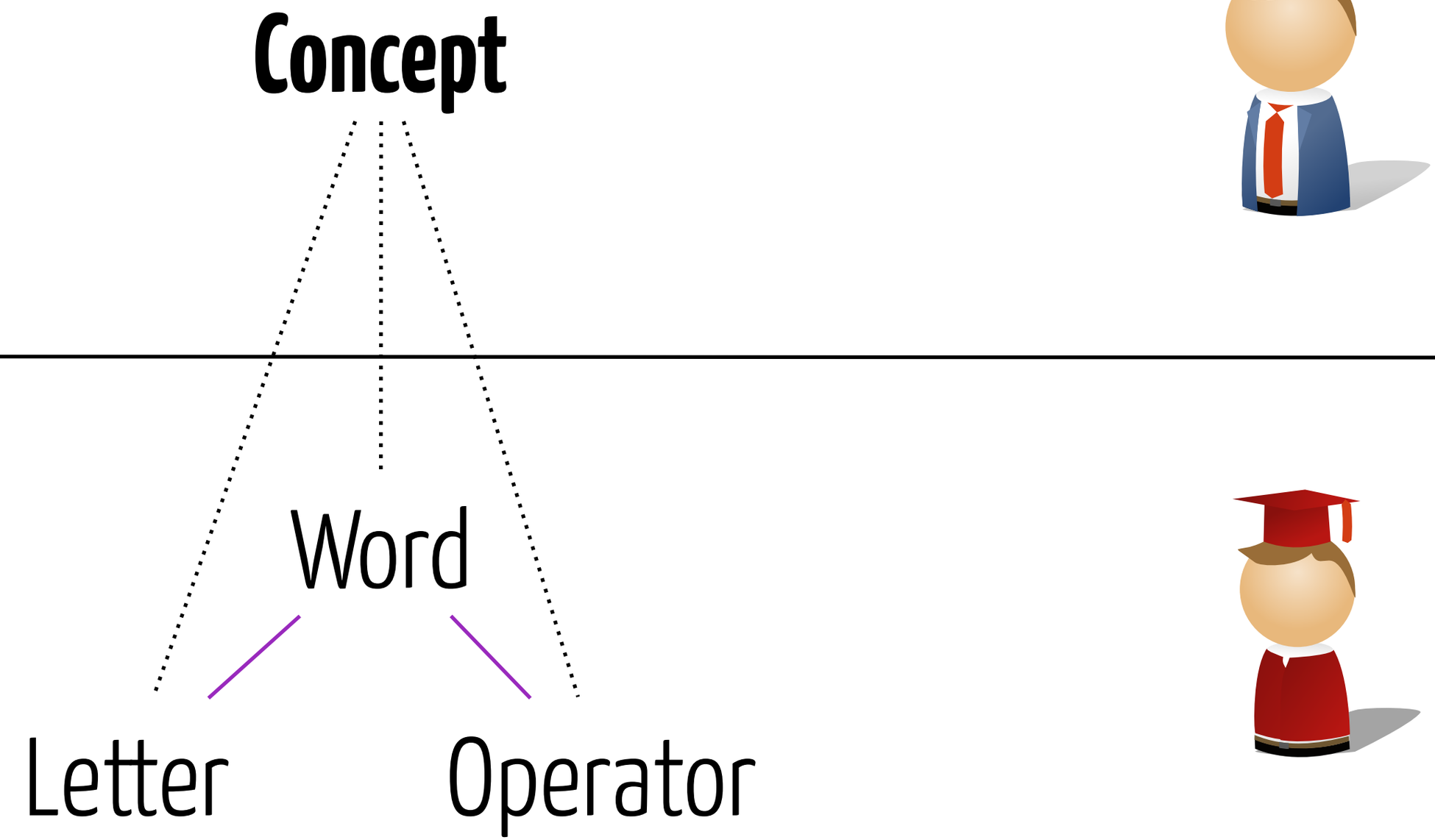
Implementing Concept Scoping

```
public class Scope extends NamedElement {
  private ArrayList<Concept> concepts;

  public Scope(String n) {
    this.name = n;
    this.concepts = new ArrayList<Concept>();
  }
}
```

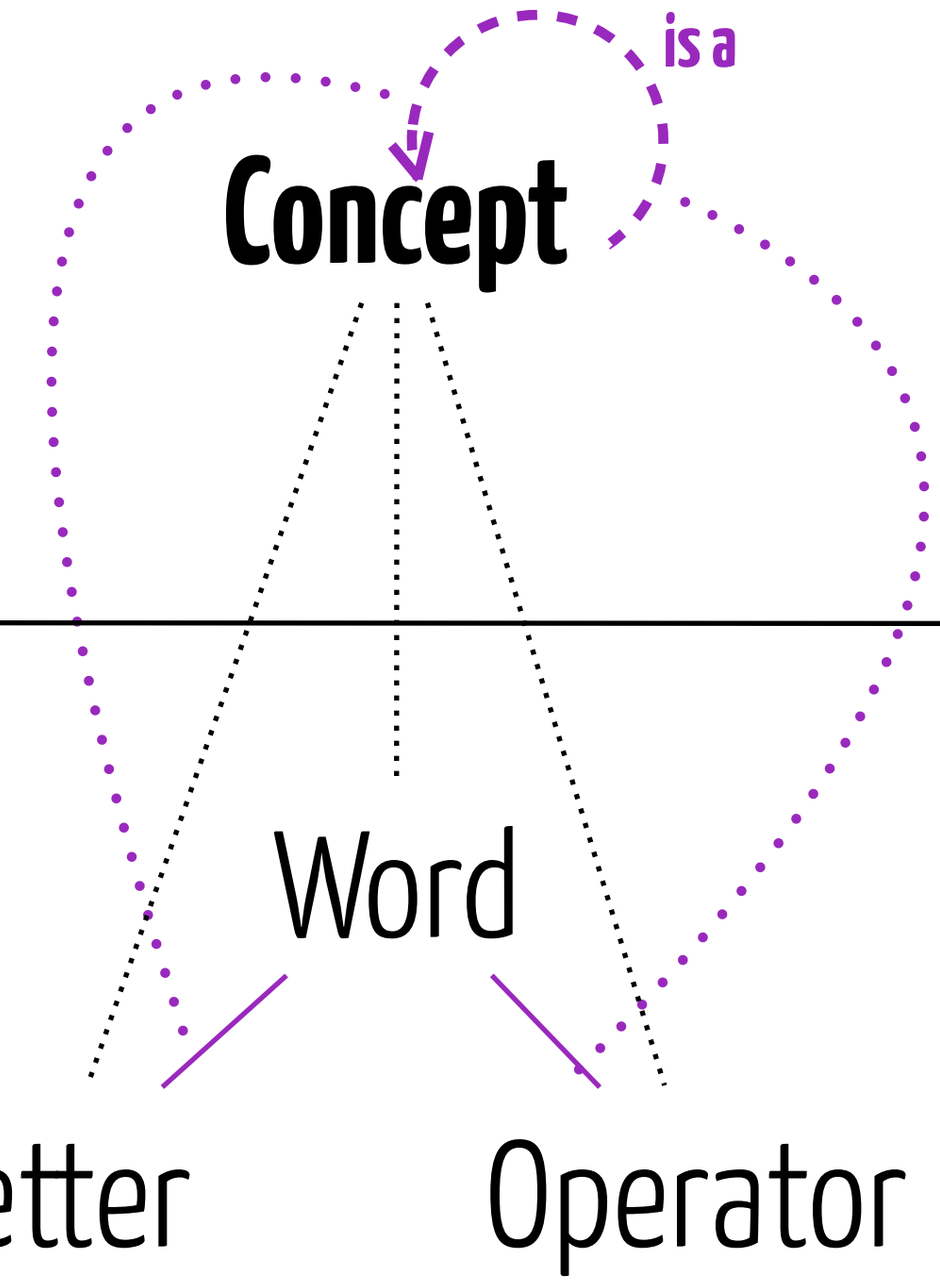
```
Scope words = new Scope("word_lib");
Scope std = new Scope("std_lib");
std.addConcept(char);
words.addConcept(letter);
```



Requirements #4: **Categorizing** Concepts



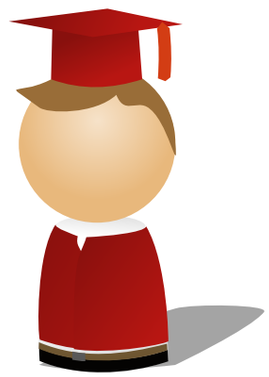
Requirements #4: **Categorizing** Concepts



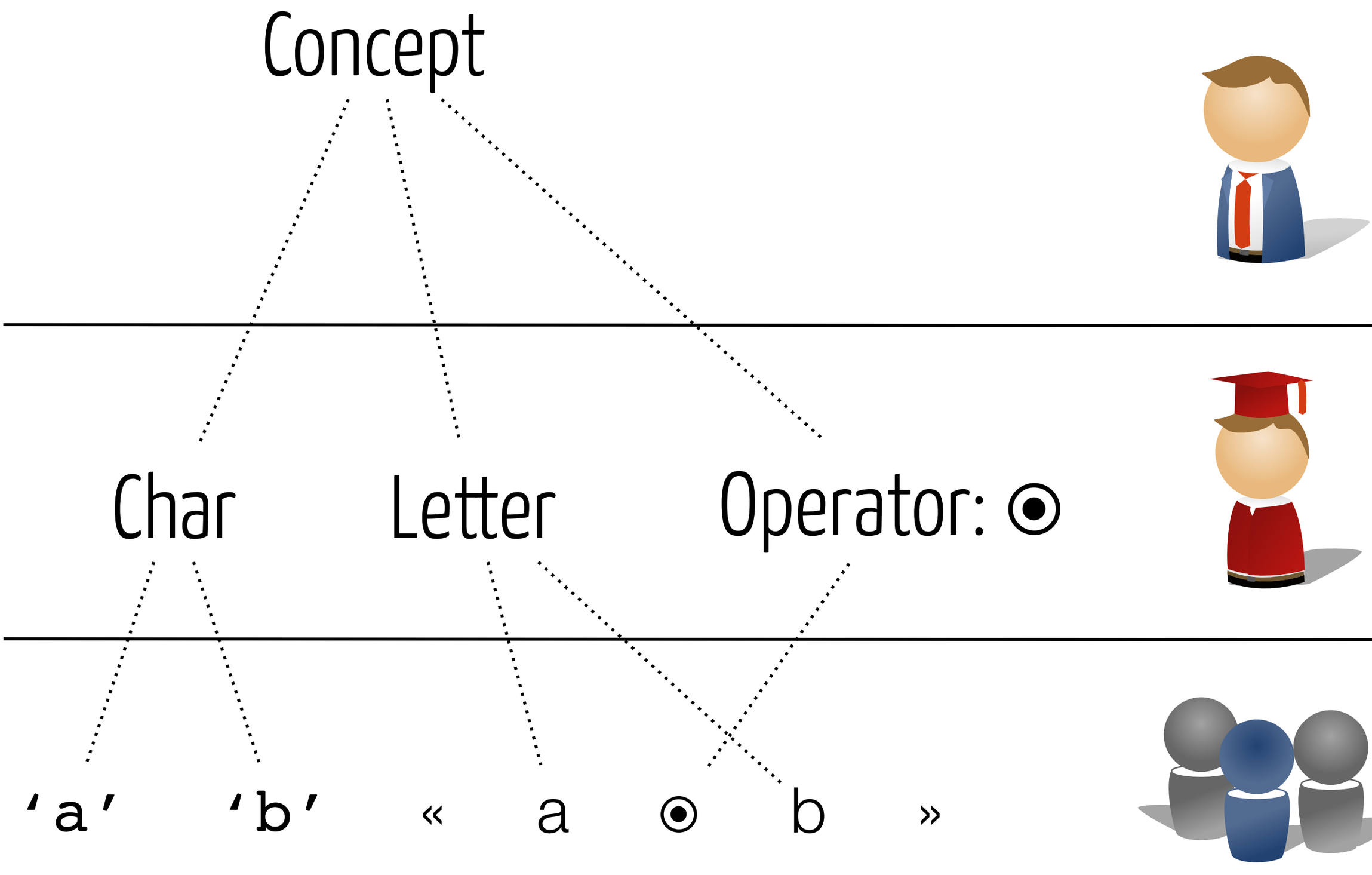
Implementing Relations between Concepts

```
public class Concept extends NamedElement {
   private Concept isA;
  // ...
   public void refines(Concept parent) {
      this.isA = parent;
   }
}
```

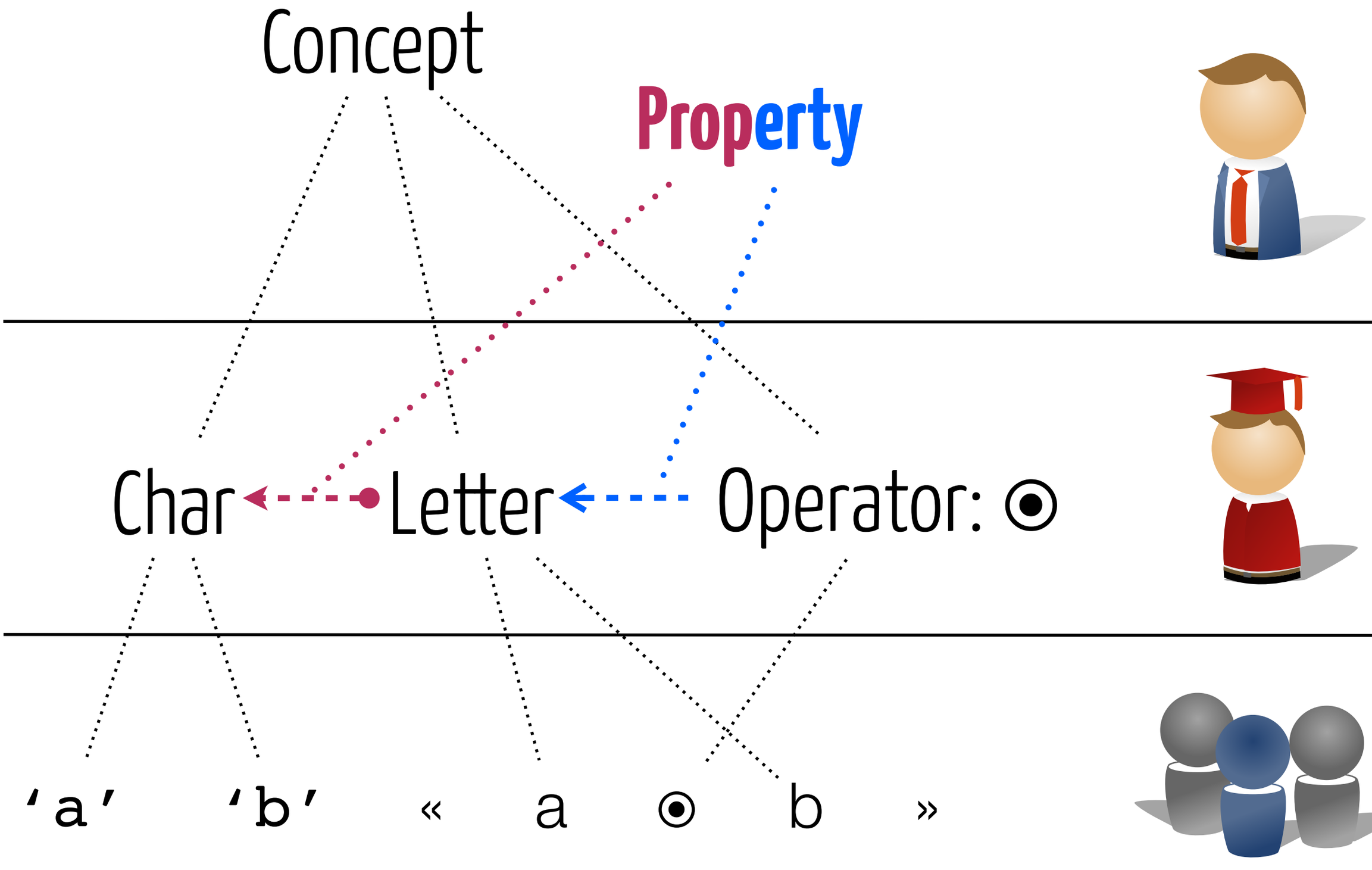
```
Concept word     = new Concept("Word");
Concept letter   = new Concept("Letter");
Concept operator = new Concept("Operator");
letter.refines(word);
operator.refines(word);
```



Requirements #5: Containment ≠ Reference



Requirements #5: Containment ≠ Reference

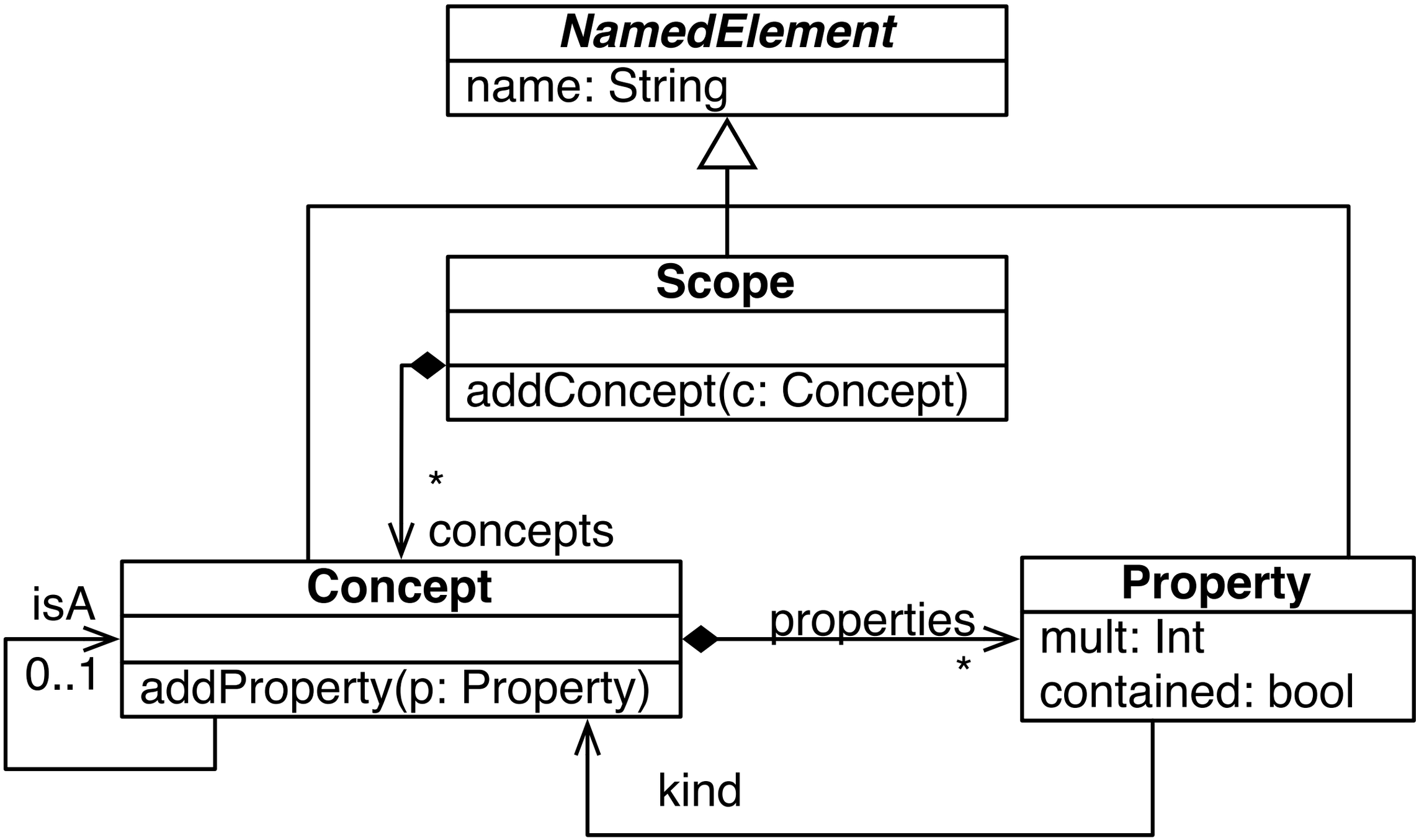


Containment = property of «Property»

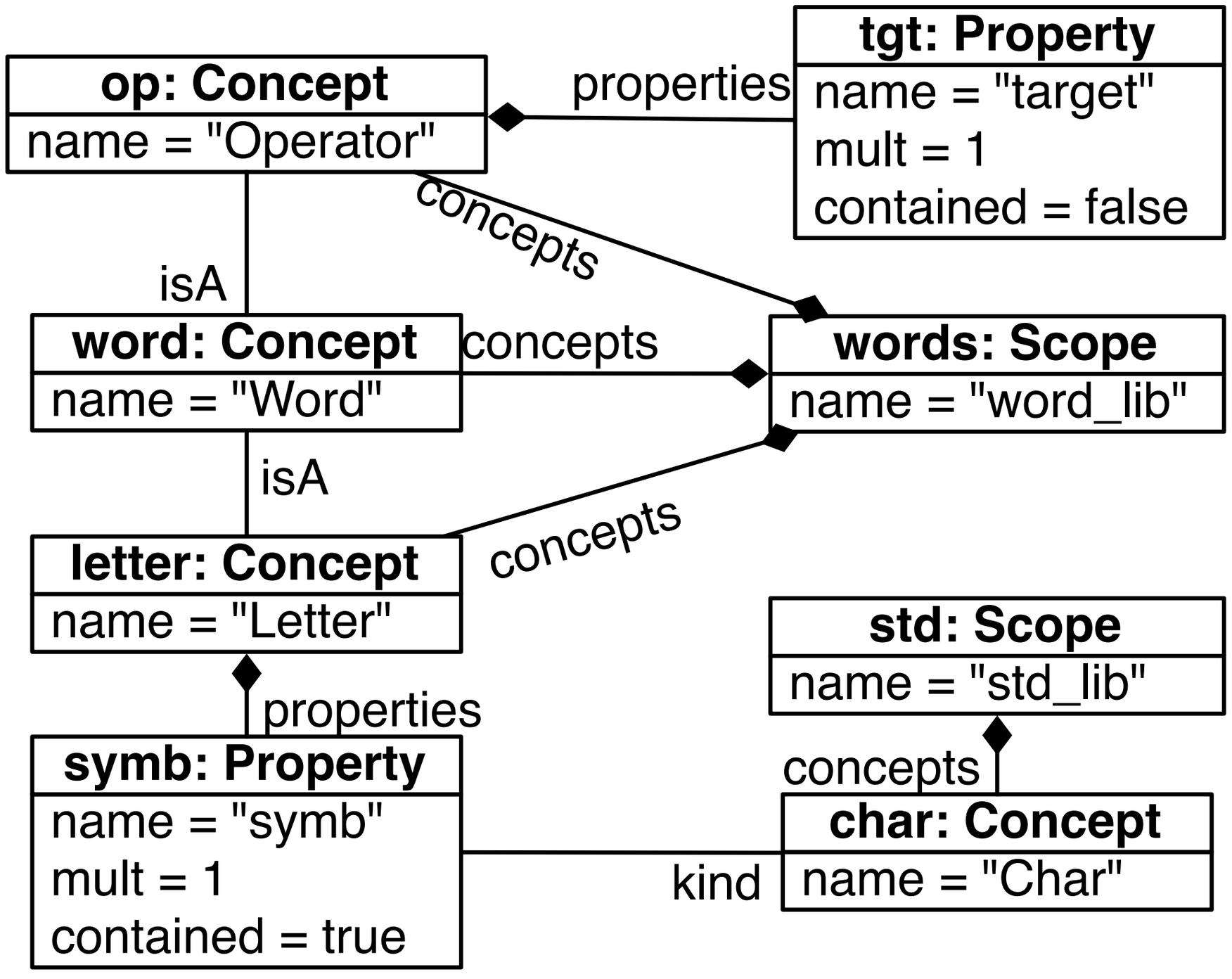
```
public class Property extends NamedElement {
   // ...
   private bool contained = false;
   public void setContained(bool b) {
      this.contained = b;
   }
}
```

```
Property letSymb = new Property("symb",ch,1);
symb.setContained(true);
letter.addProperty(letSymb);

Property opWord  = new Property("word",word,-1);
operator.addProperty(opWord);
```

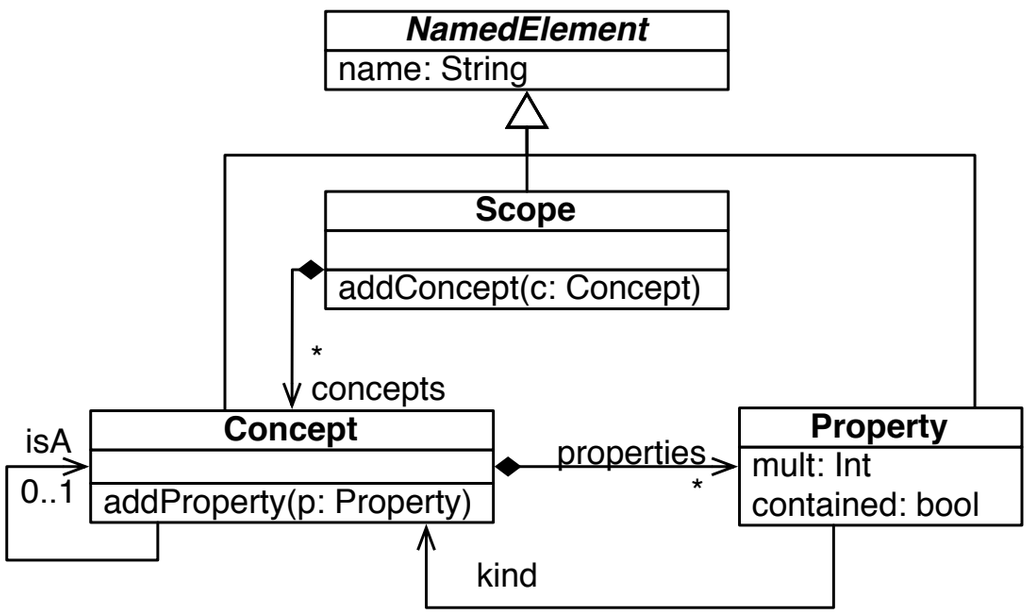
Meta-meta-model Summary (UML Class Diag.)

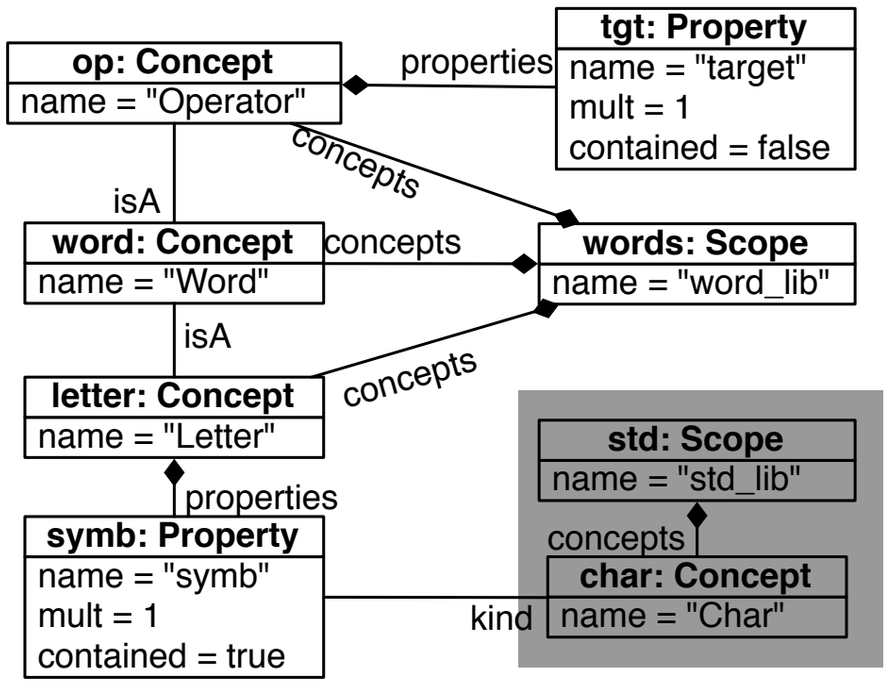


Meta-model Summary (UML Object Diag.)

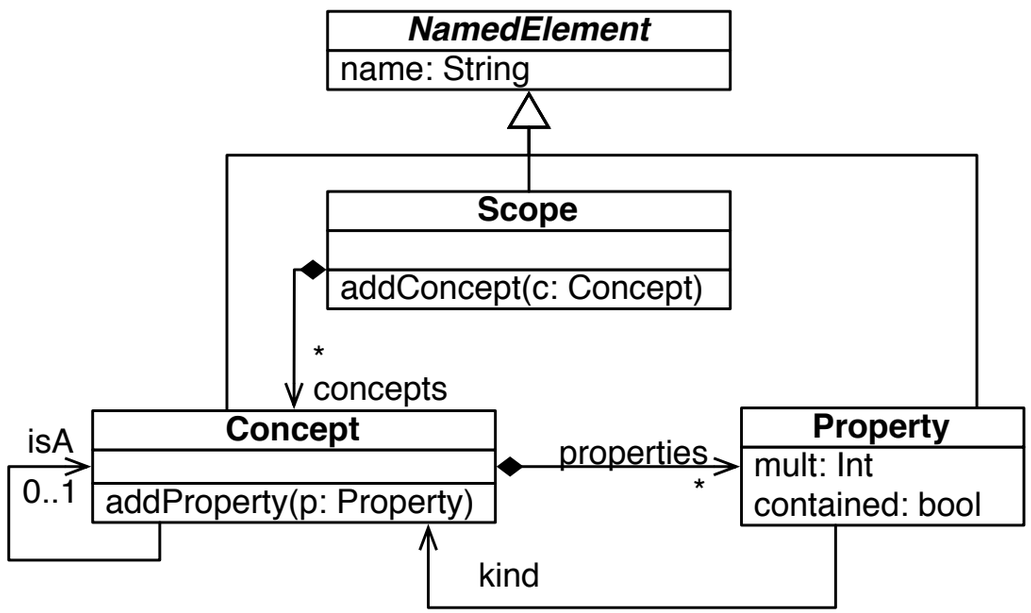


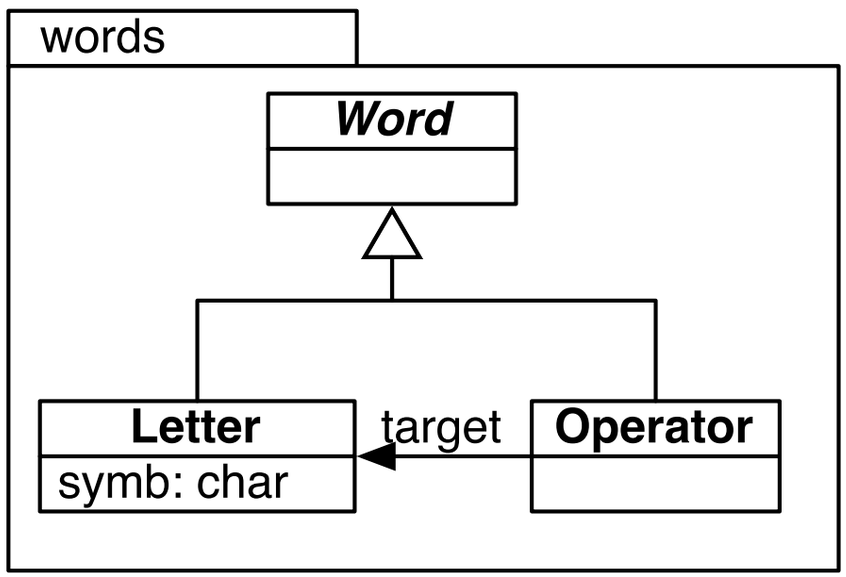
From Meta-models to Object-Oriented Classes

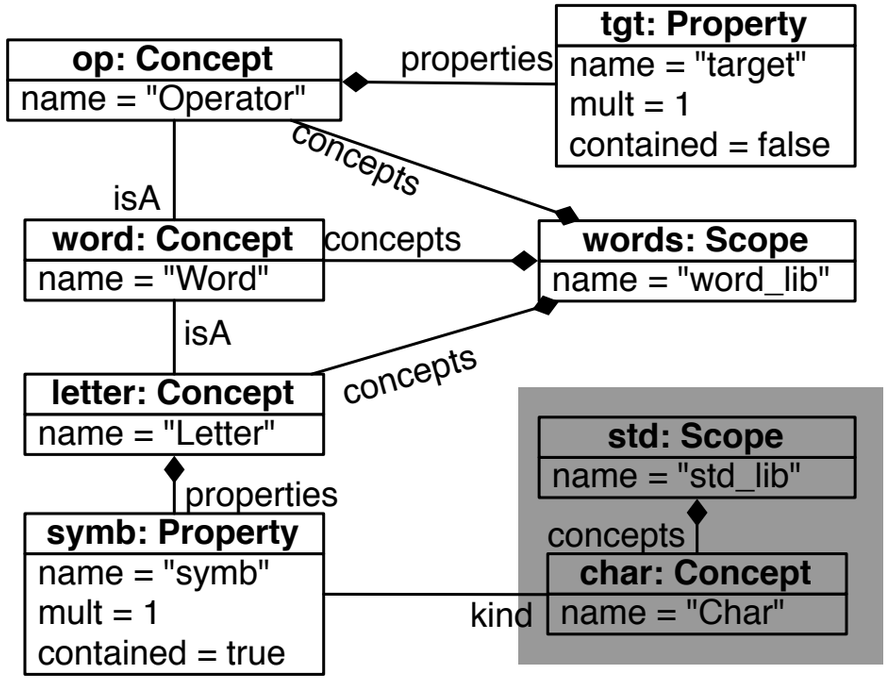




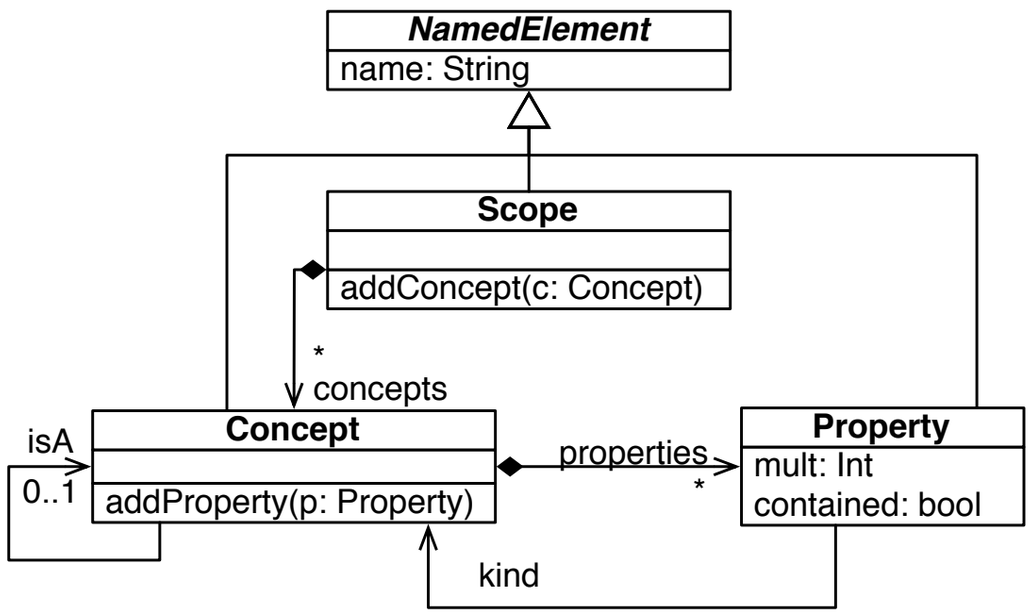
From Meta-models to Object-Oriented Classes

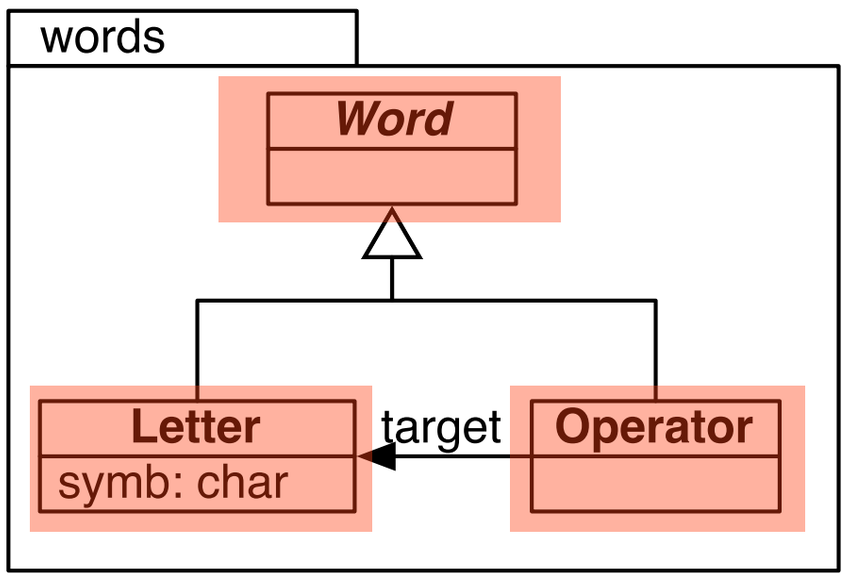


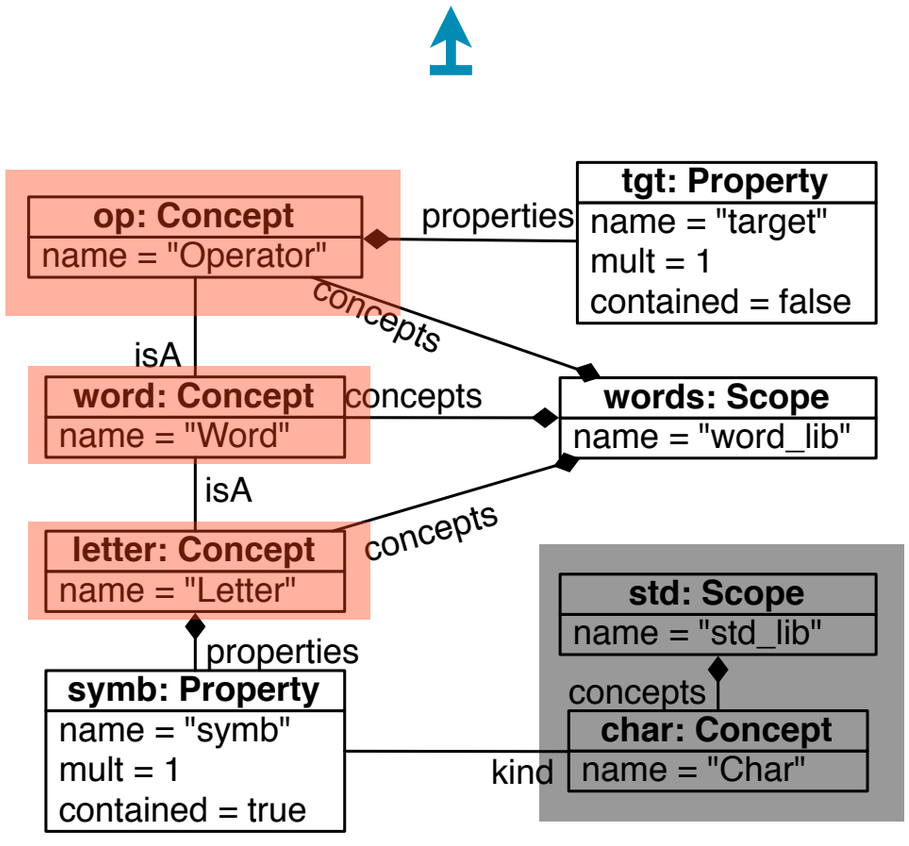




From Meta-models to Object-Oriented Classes

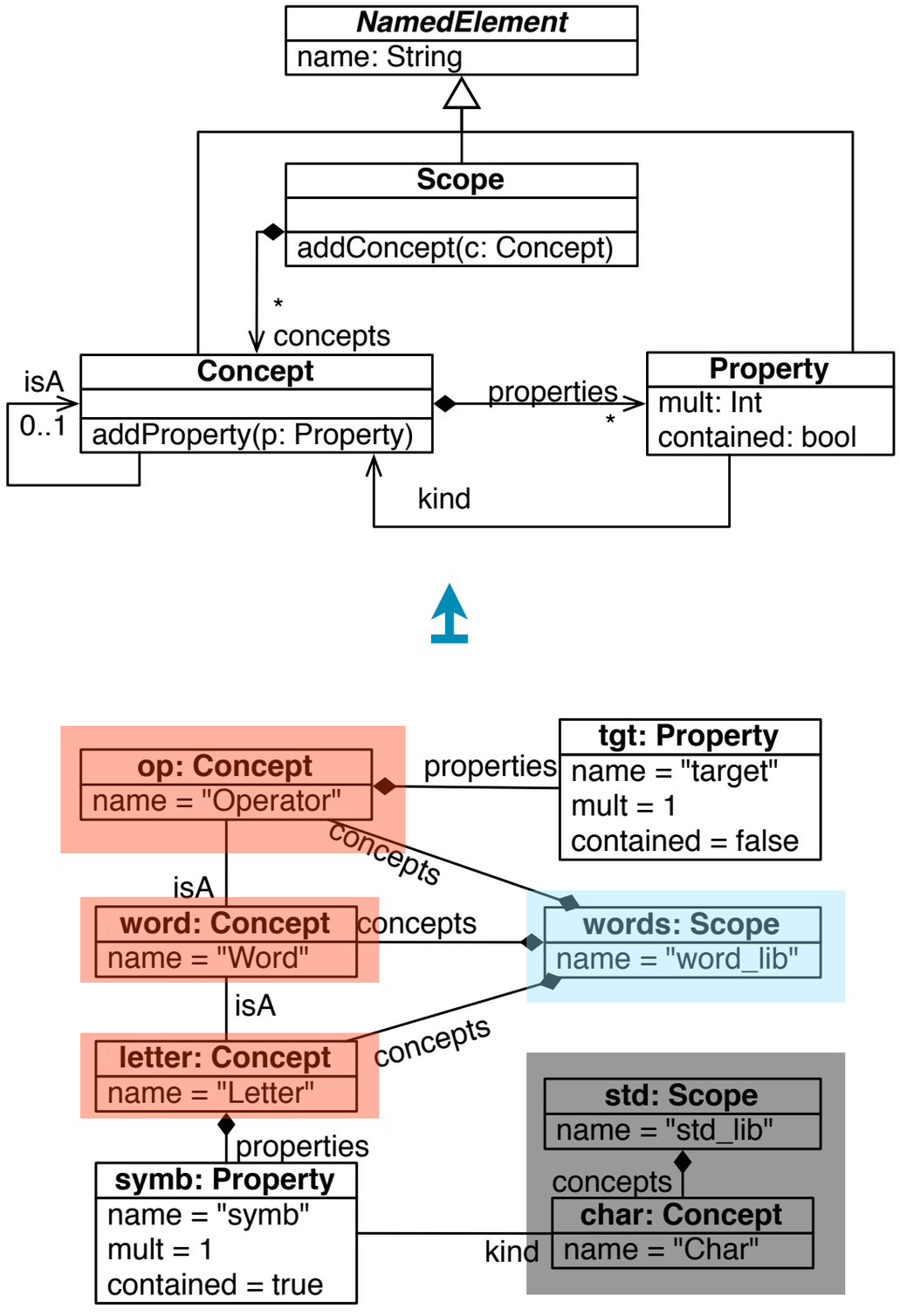


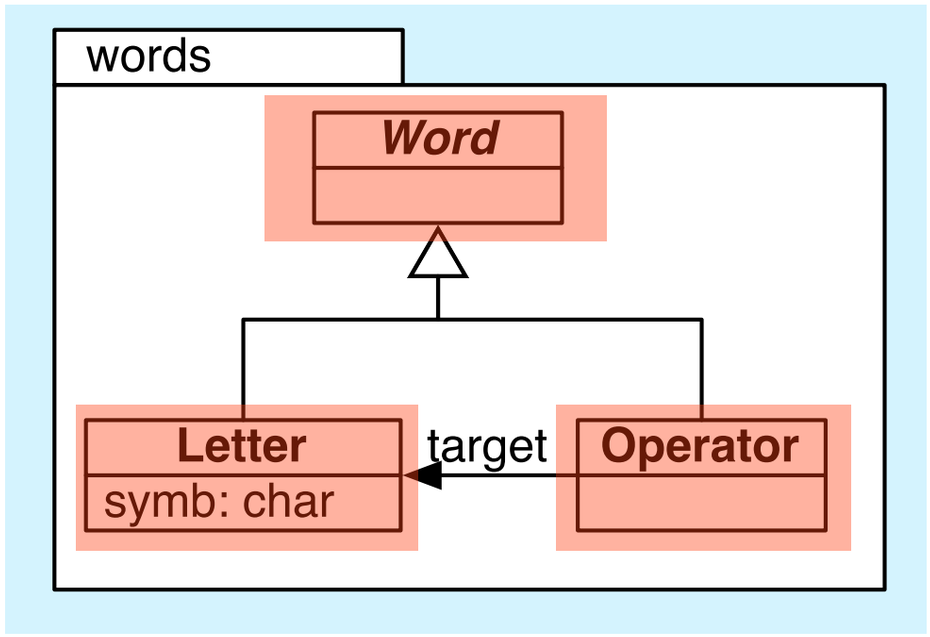




Concept → *Class*

From Meta-models to Object-Oriented Classes

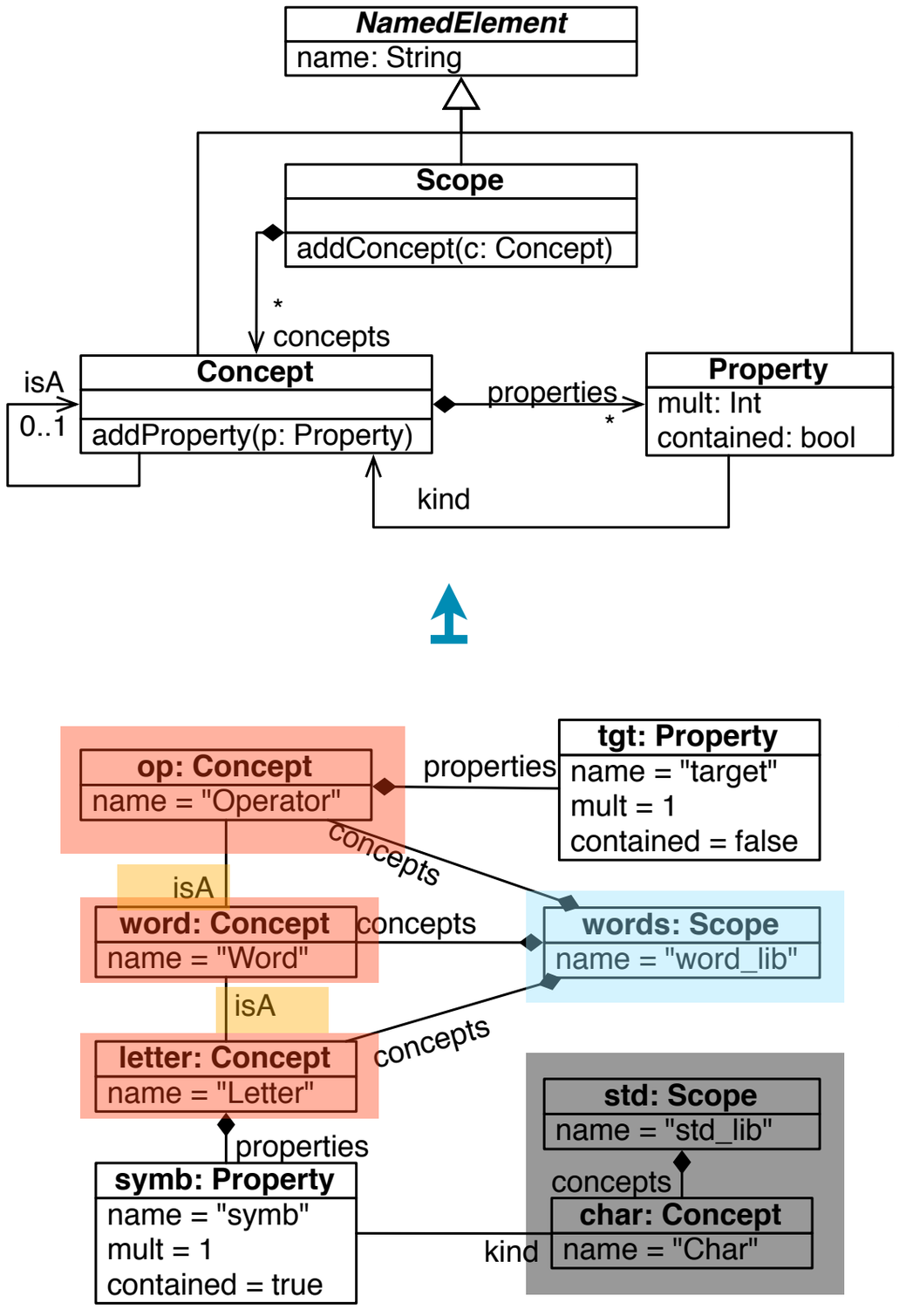


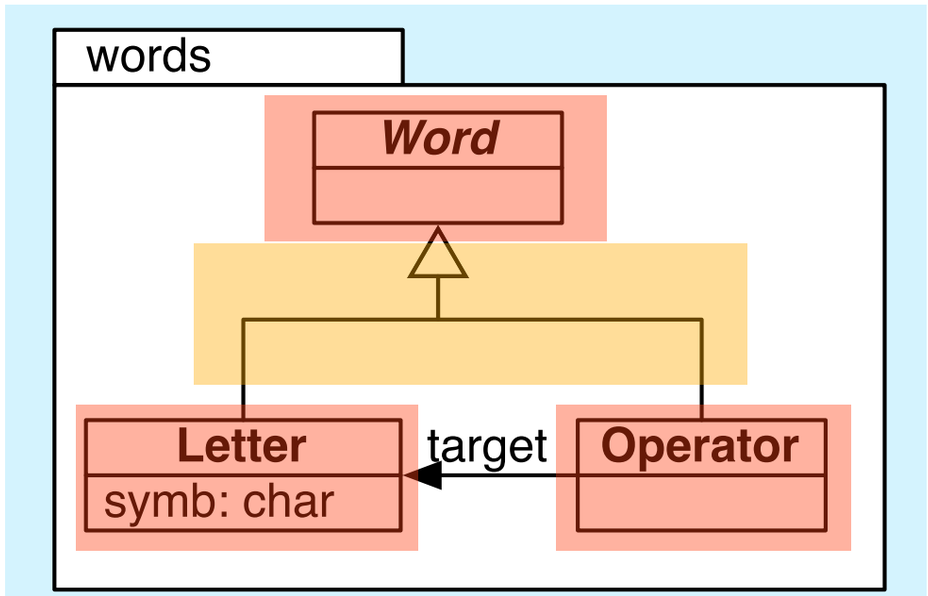


Concept → *Class*

Scope → *Package*

From Meta-models to Object-Oriented Classes



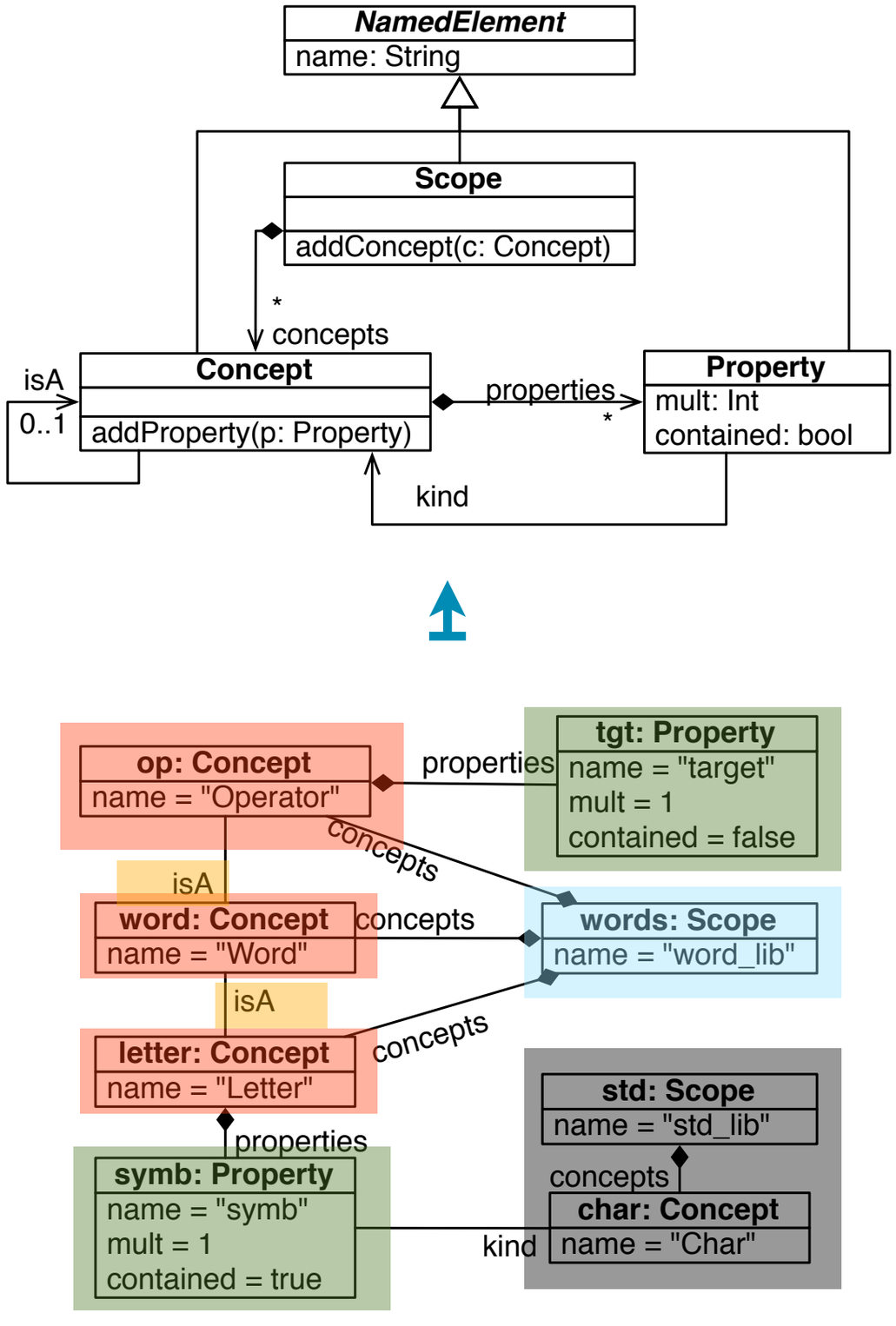


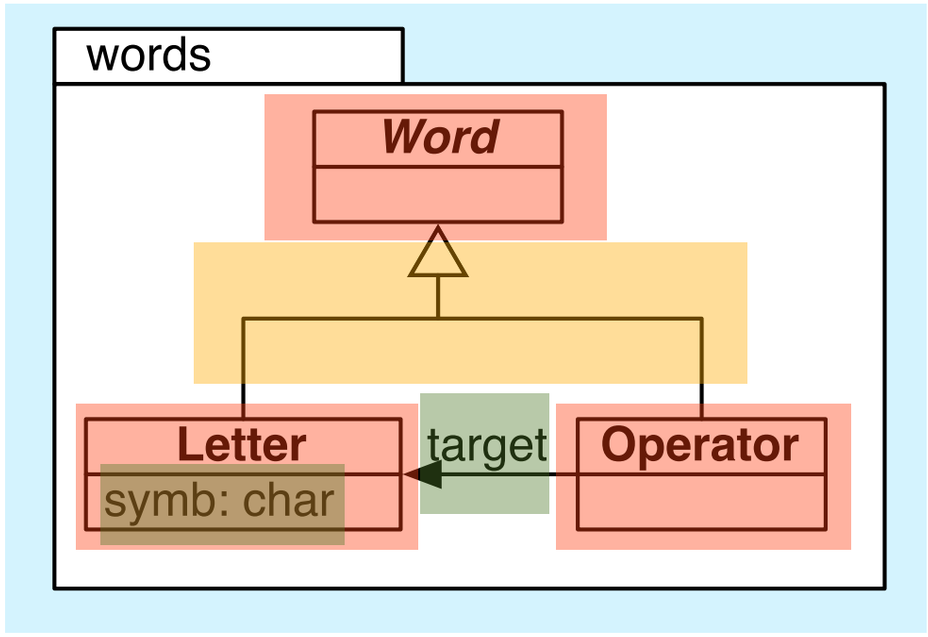
Concept → *Class*

Scope → *Package*

isA → *Inheritance*

From Meta-models to Object-Oriented Classes





Concept → *Class*

Scope → *Package*

isA → *Inheritance*

Property → *Attribute*

Bootstrapping a Modelling Language

A sentence *contains* ***a subject***,
a verb, *and* ***a complement***.

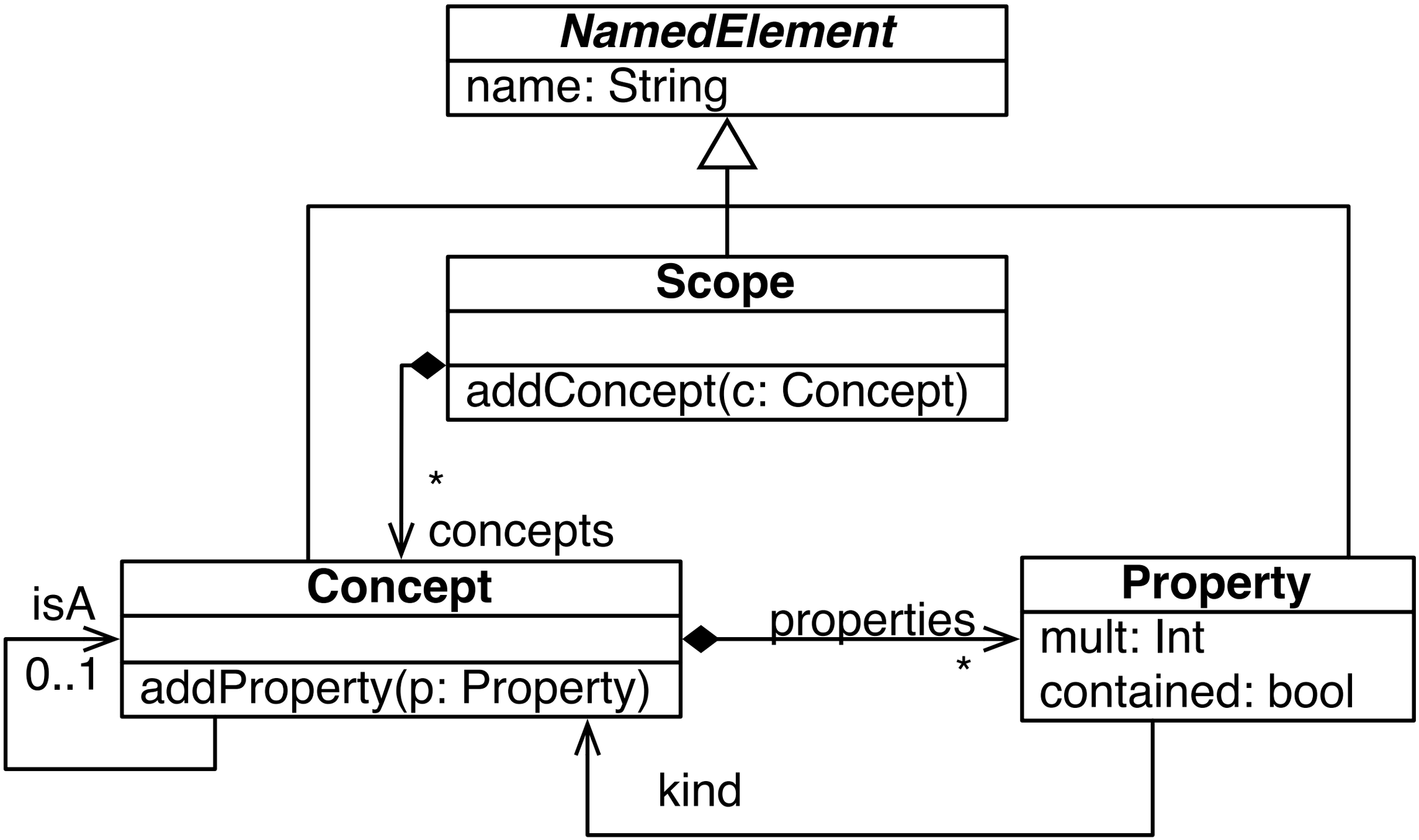
«conforms to»

A sentence *contains* ***a subject***,
a verb, *and* ***a complement***.

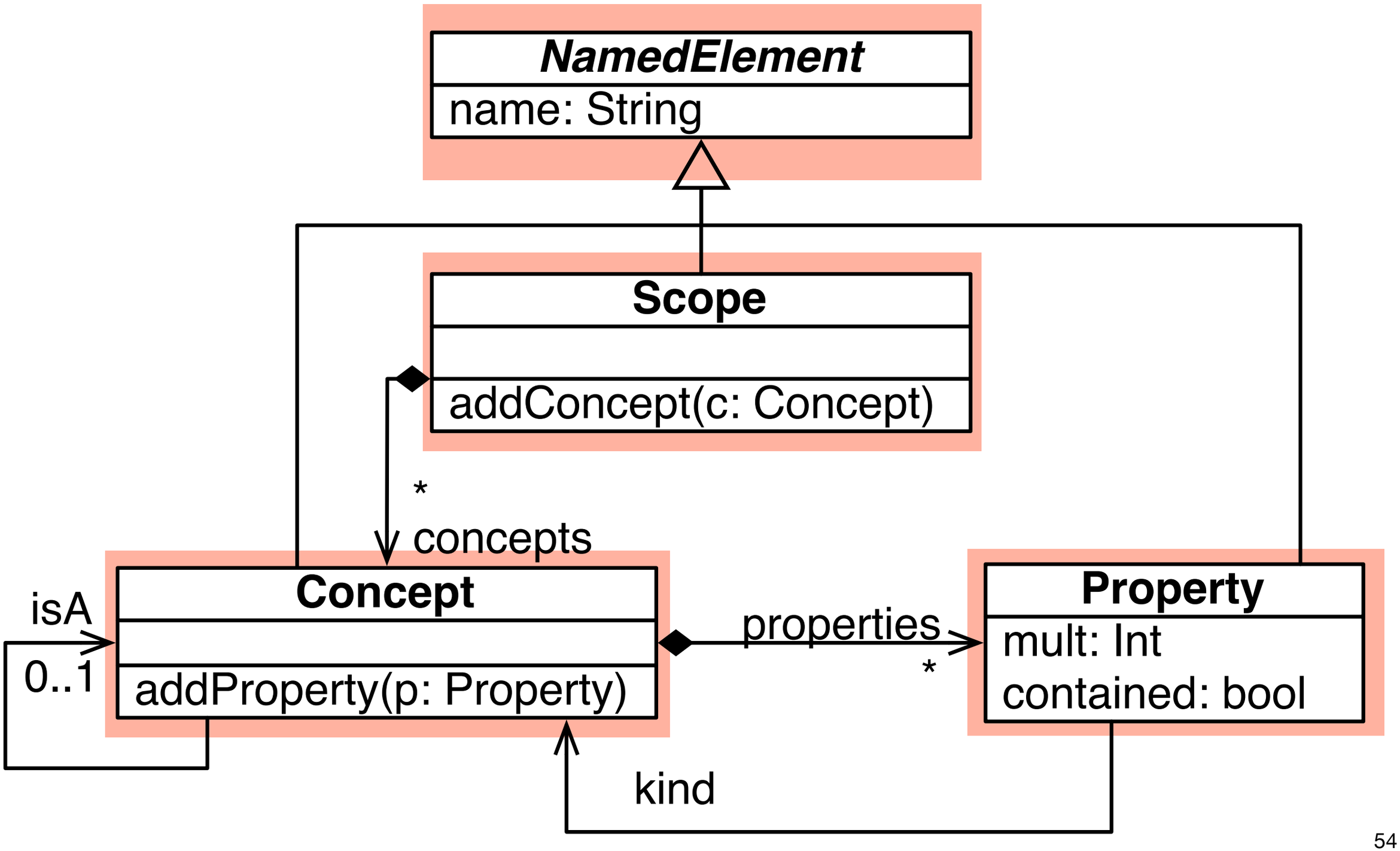
«conforms to»

I love model-driven engineering

Meta-meta-model **bootstrap**



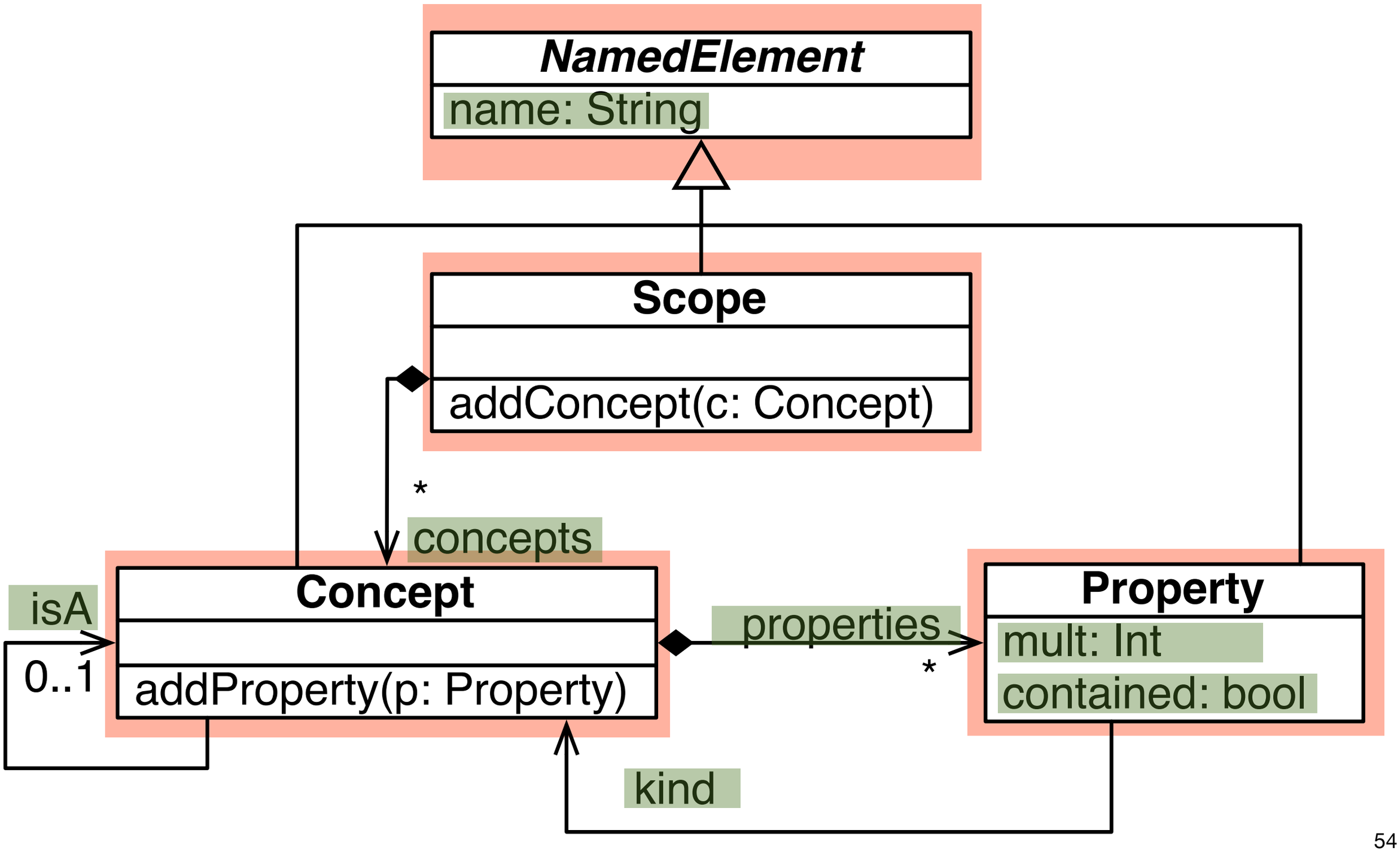
Meta-meta-model **bootstrap**



Meta-meta-model **bootstrap**

Concept

Property

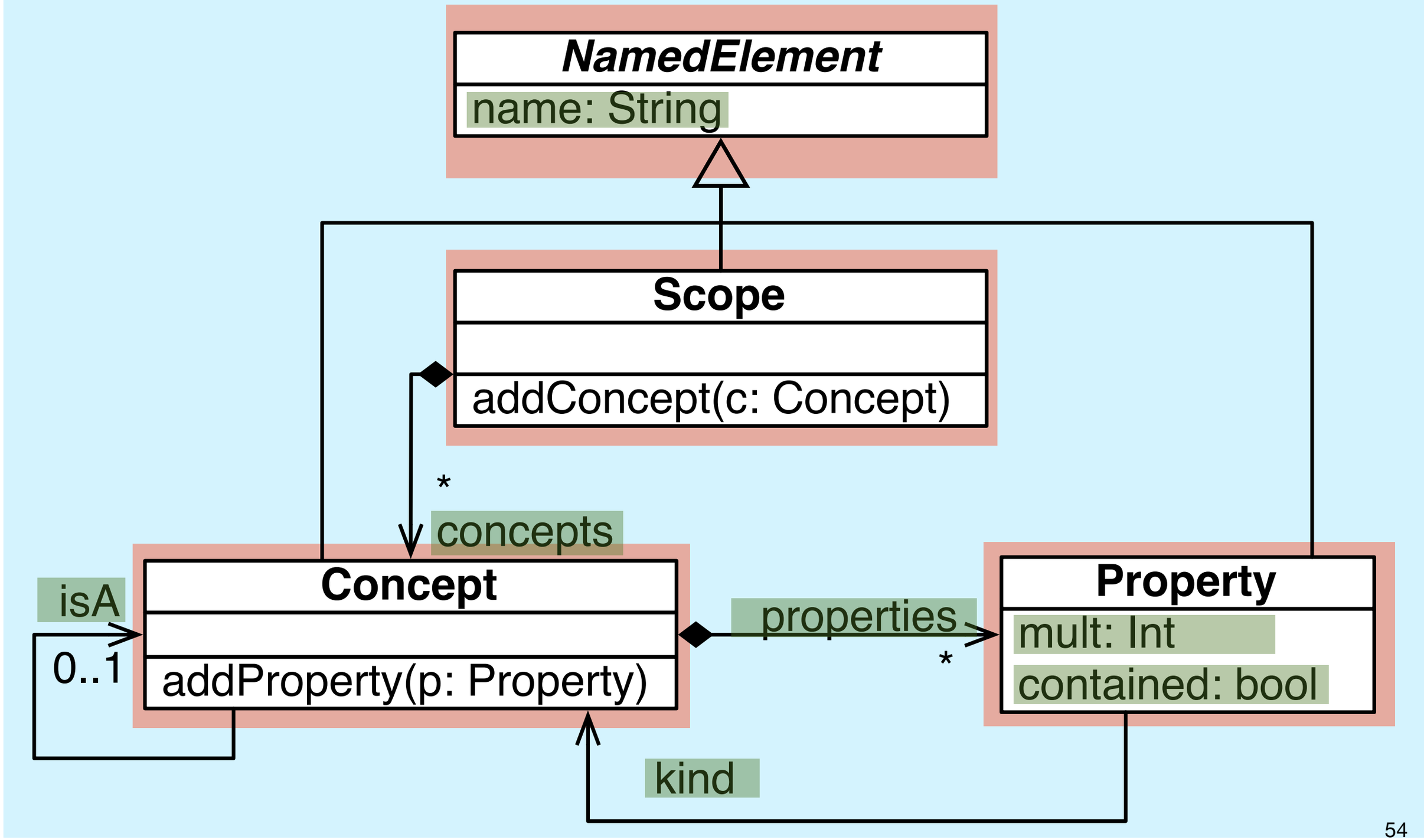


Meta-meta-model **bootstrap**

Scope

Concept

Property



Step #0: Using UML to **bootstrap** (minimal)

Concept
name: String

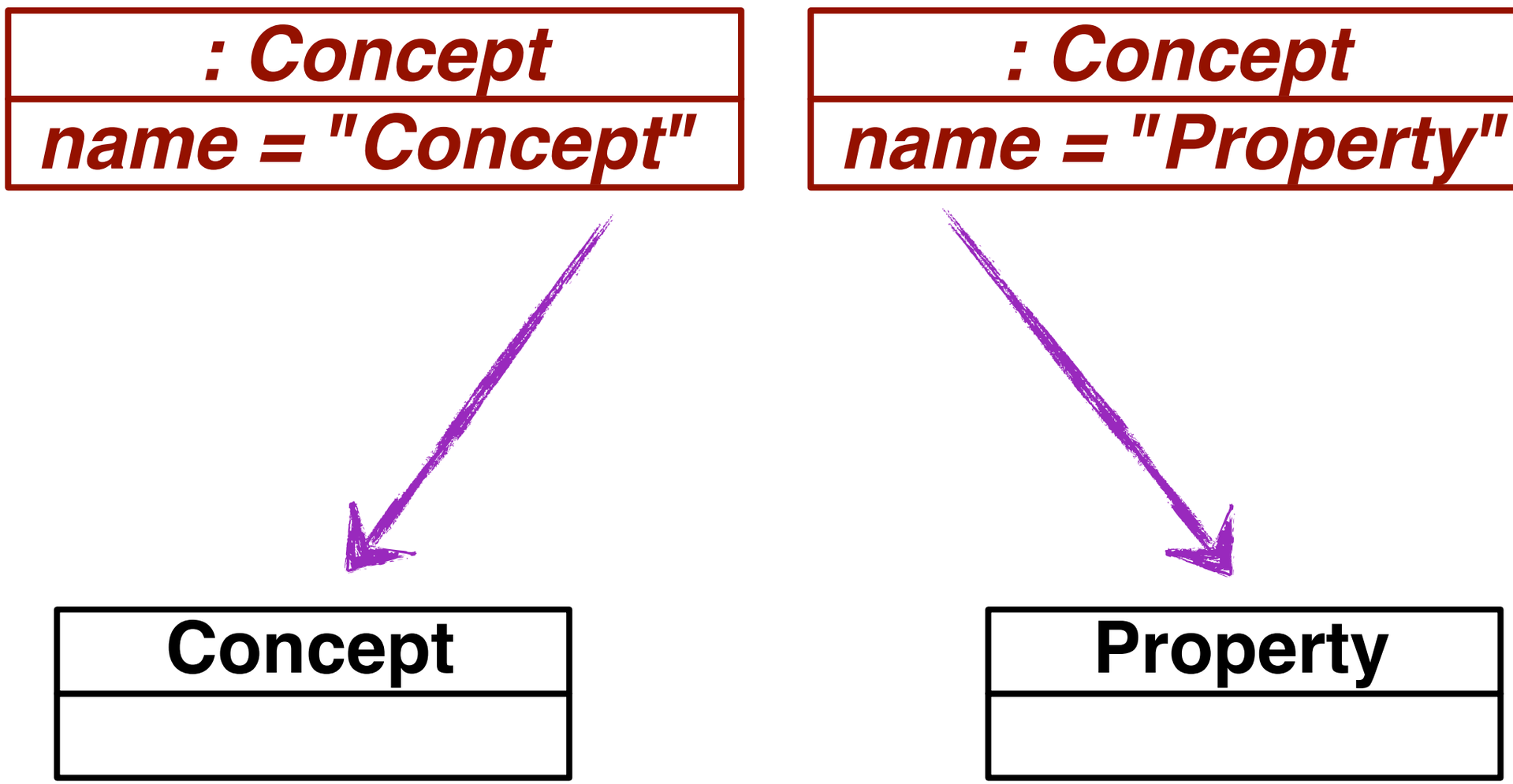
Step #0: Using UML to **bootstrap** (minimal)

Concept
name: String

: Concept
name = "Concept"

: Concept
name = "Property"

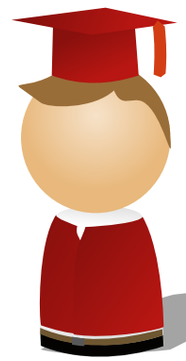
Code Generation



Step #1: Holding **Properties** (UML helper)

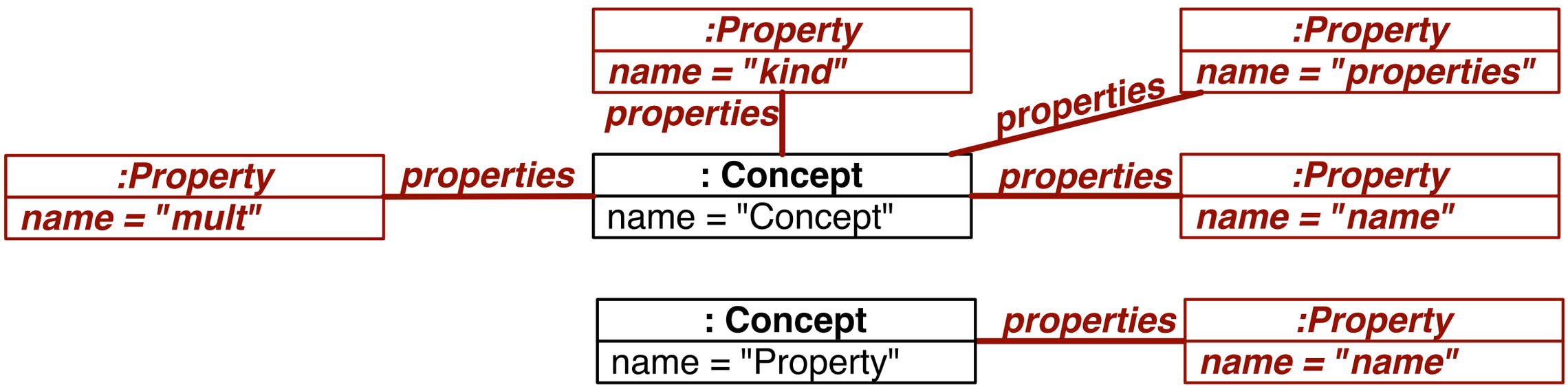
Concept

Property

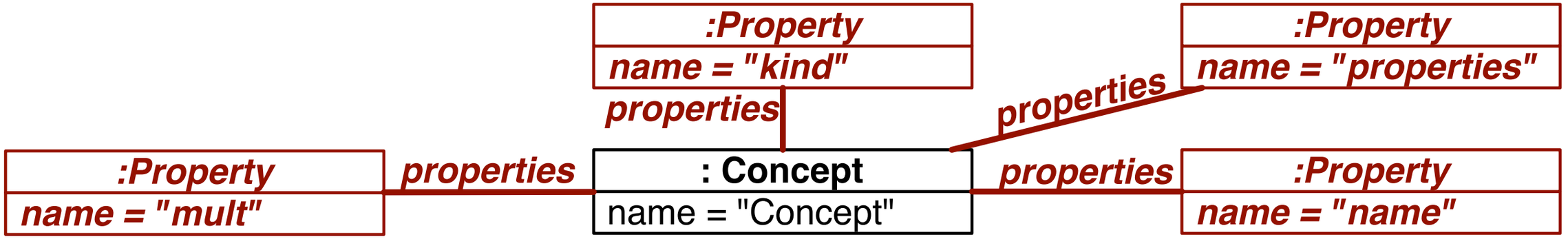


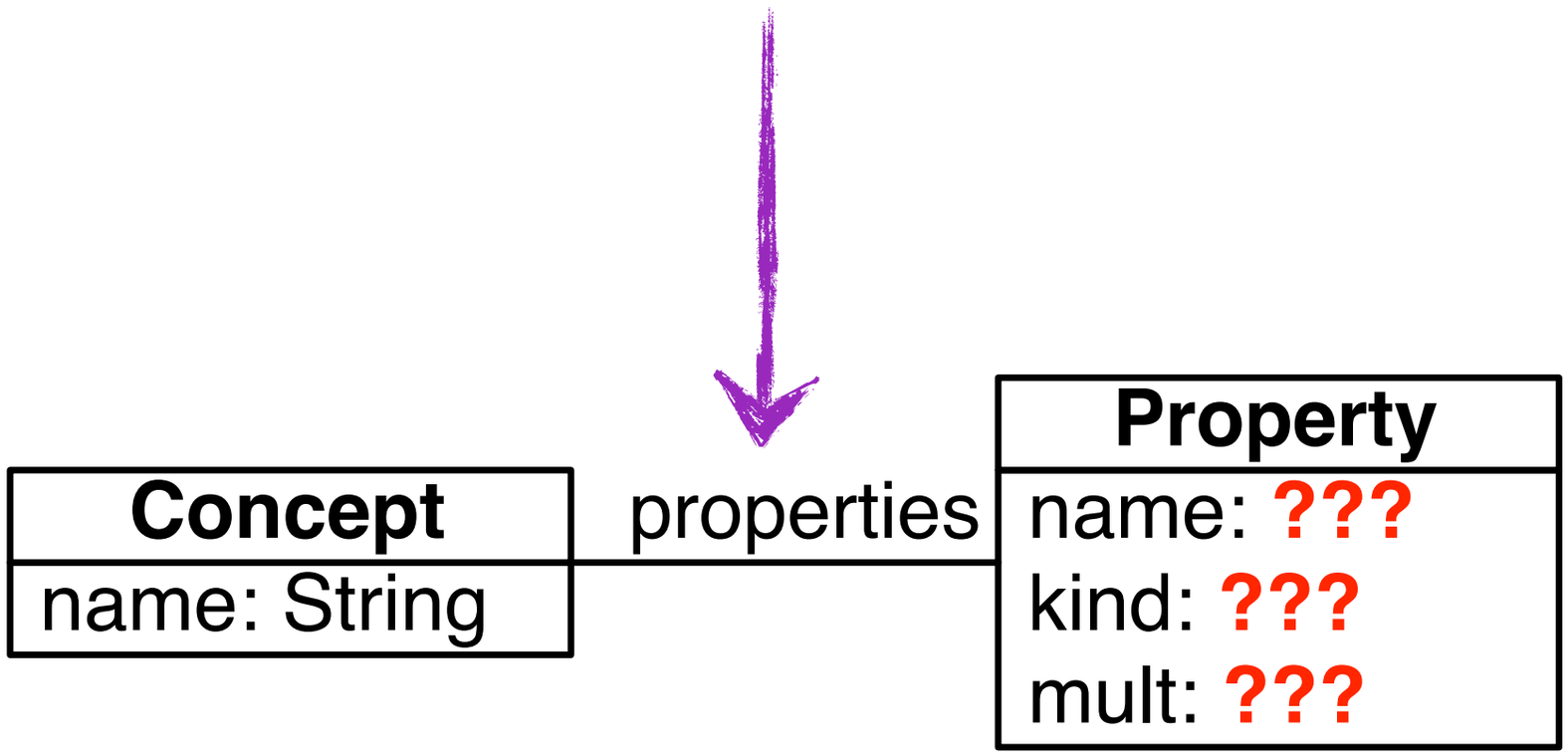
Step #1: Holding **Properties** (**UML** helper)

Step #1: Holding **Properties** (UML helper)

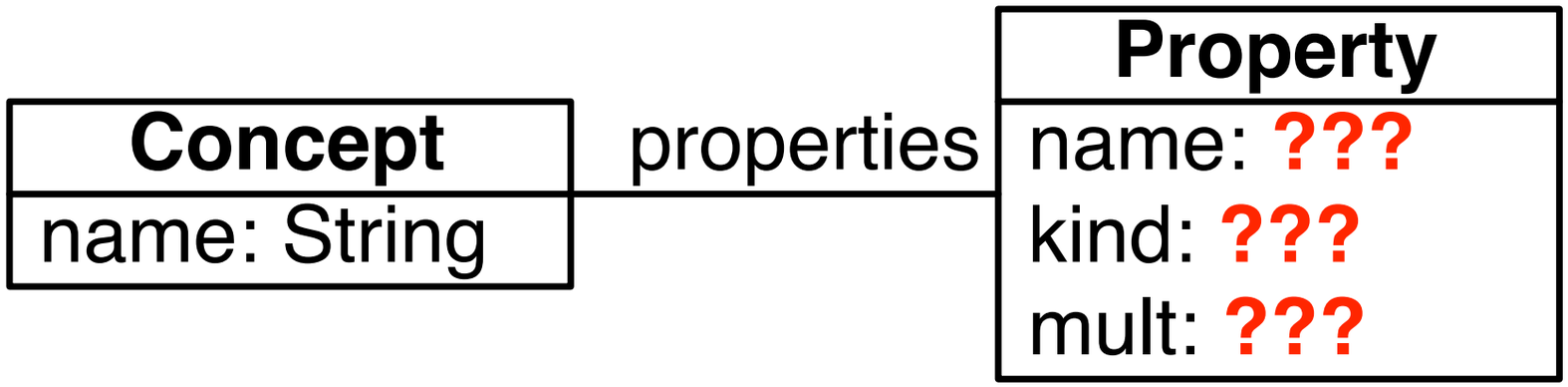


Code Generation

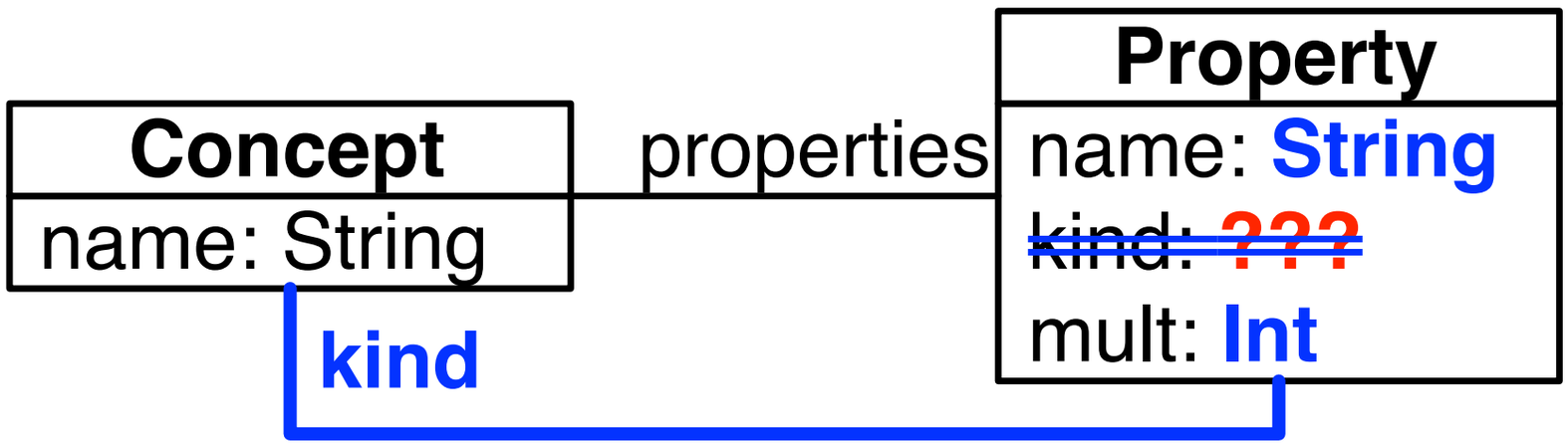




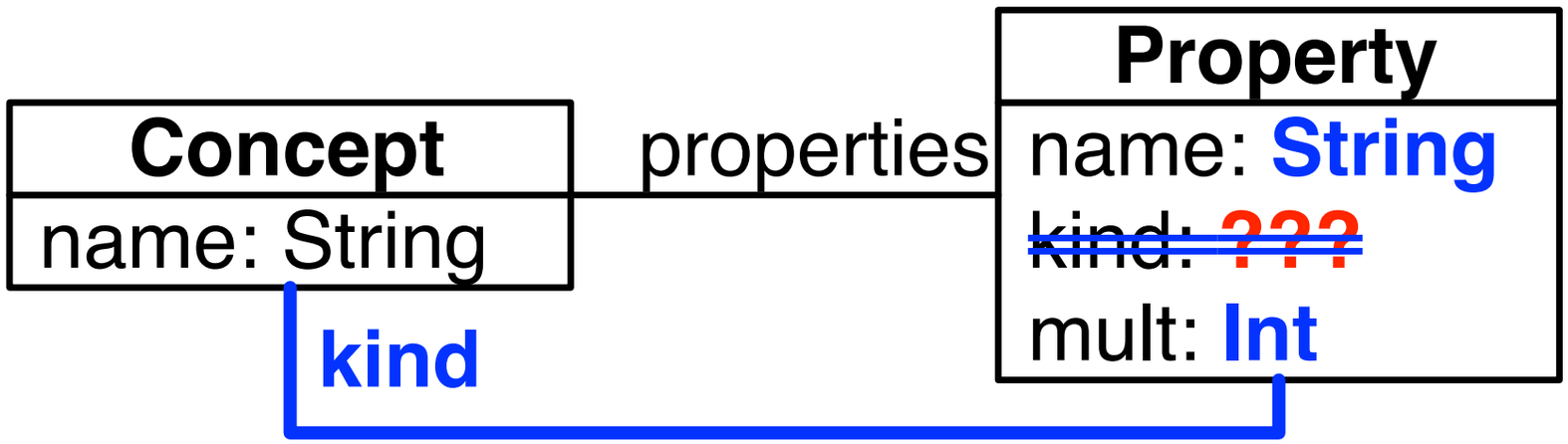
Step #2: Typed **Properties** (UML helper')

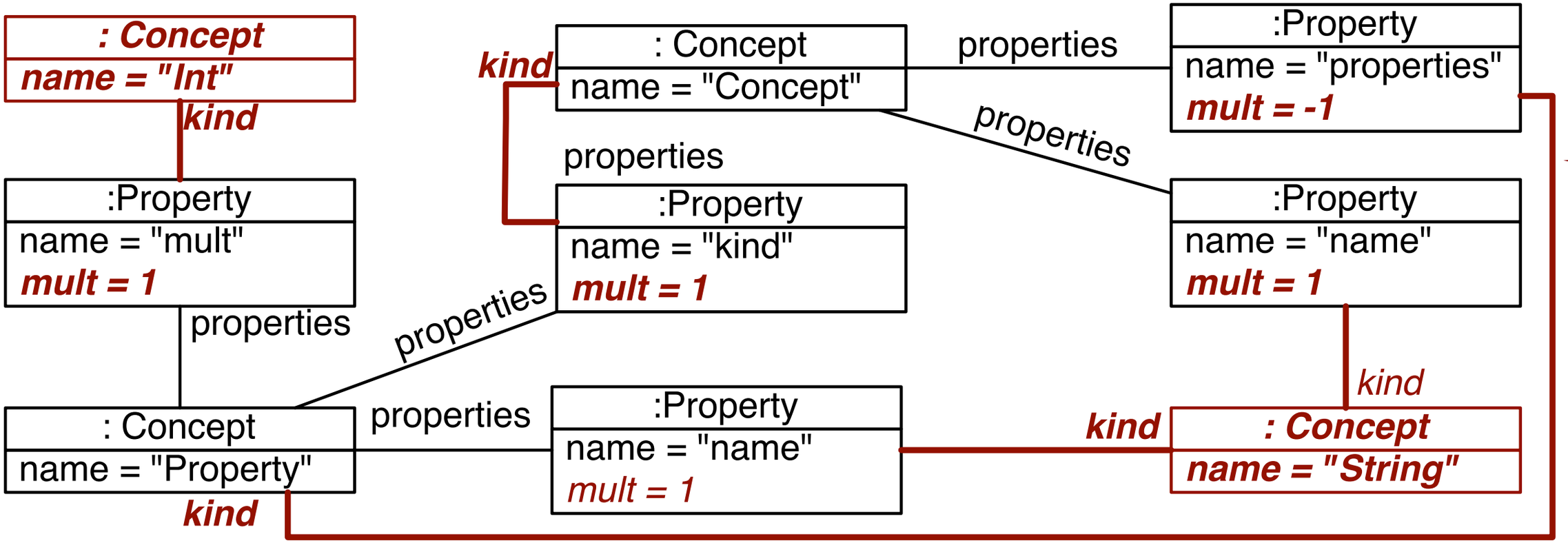


Step #2: Typed **Properties** (UML helper')

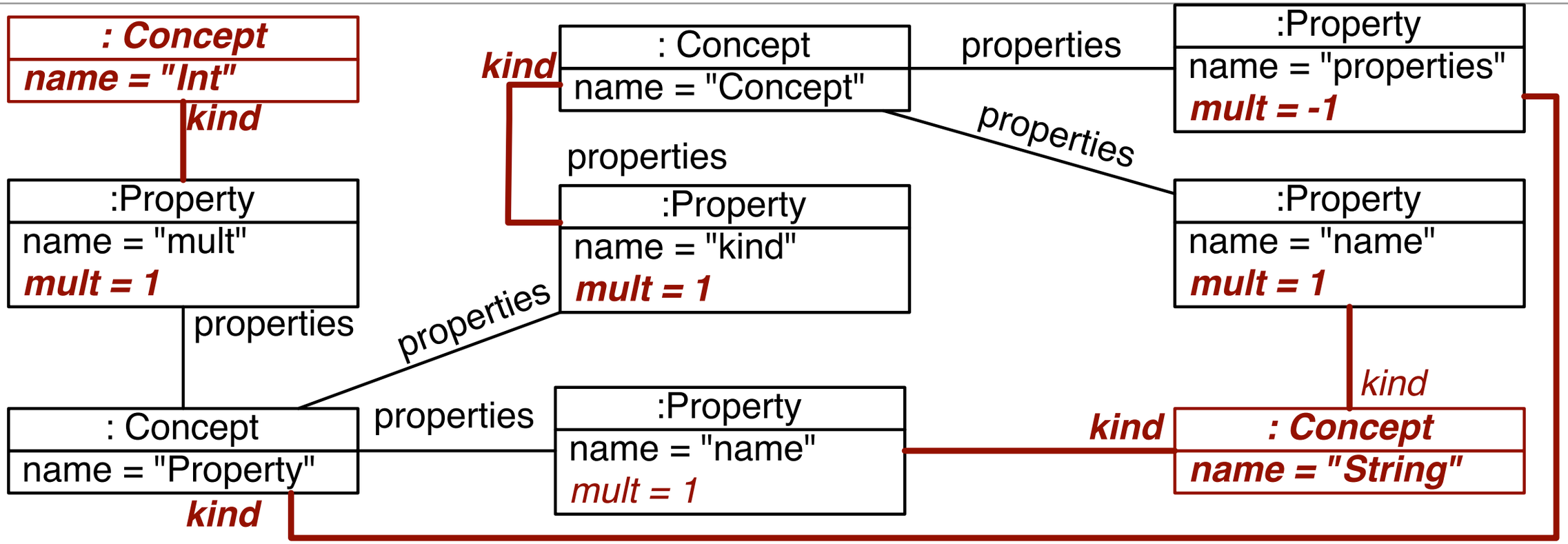


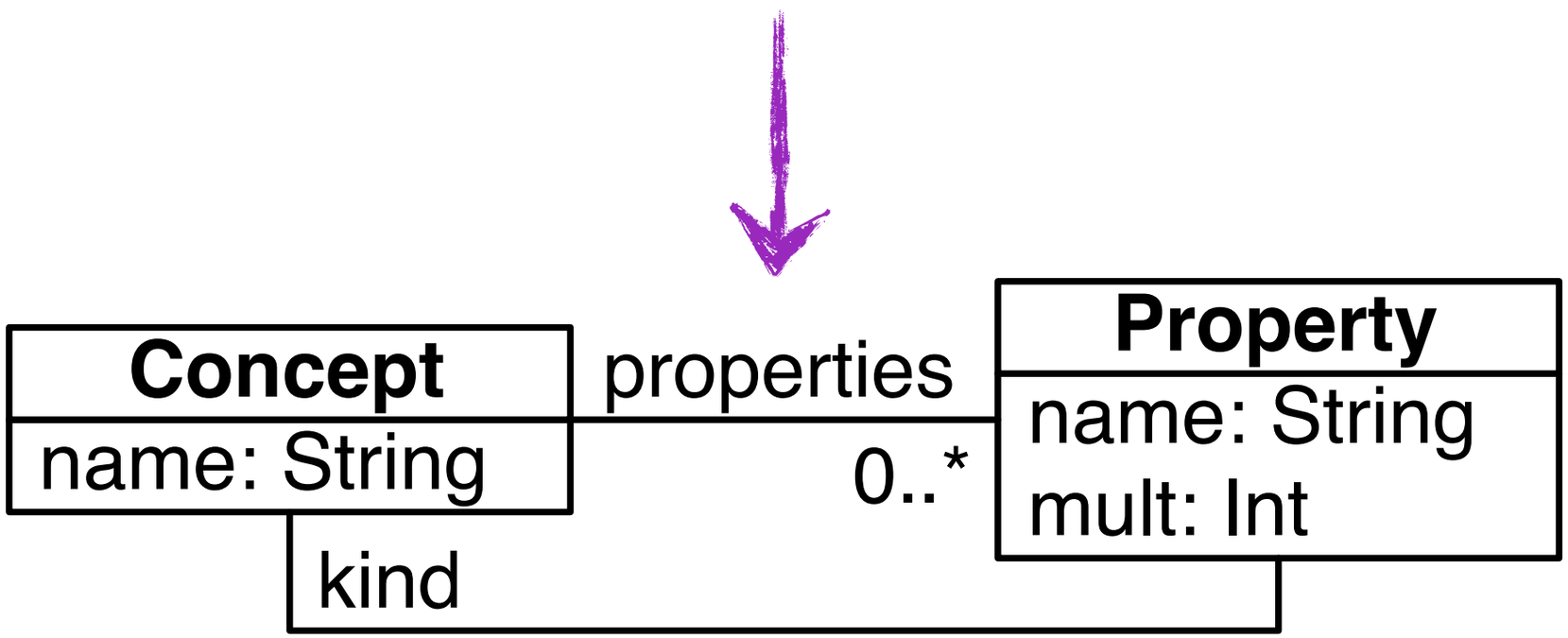
Step #2: Typed **Properties** (UML helper')



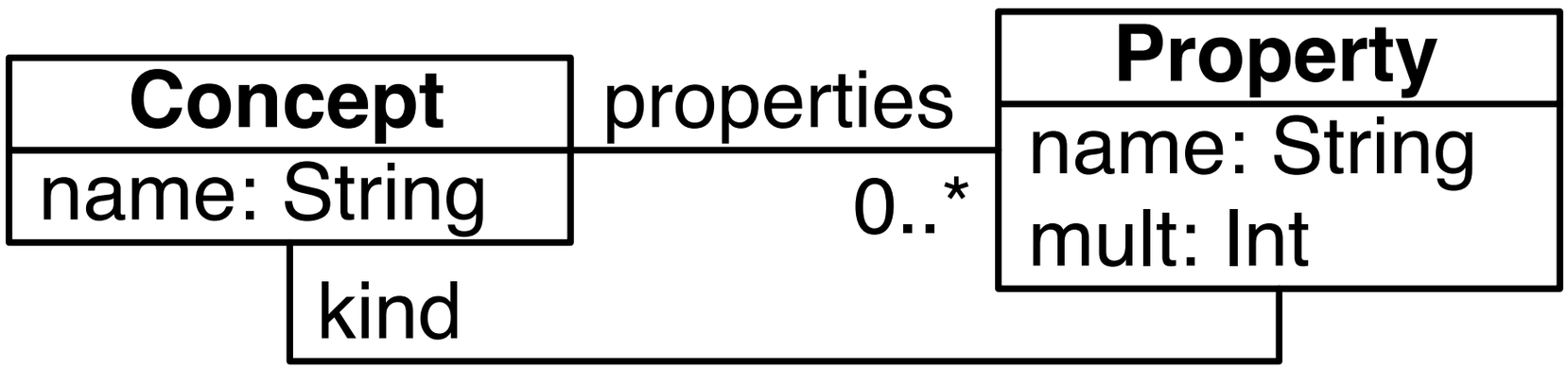


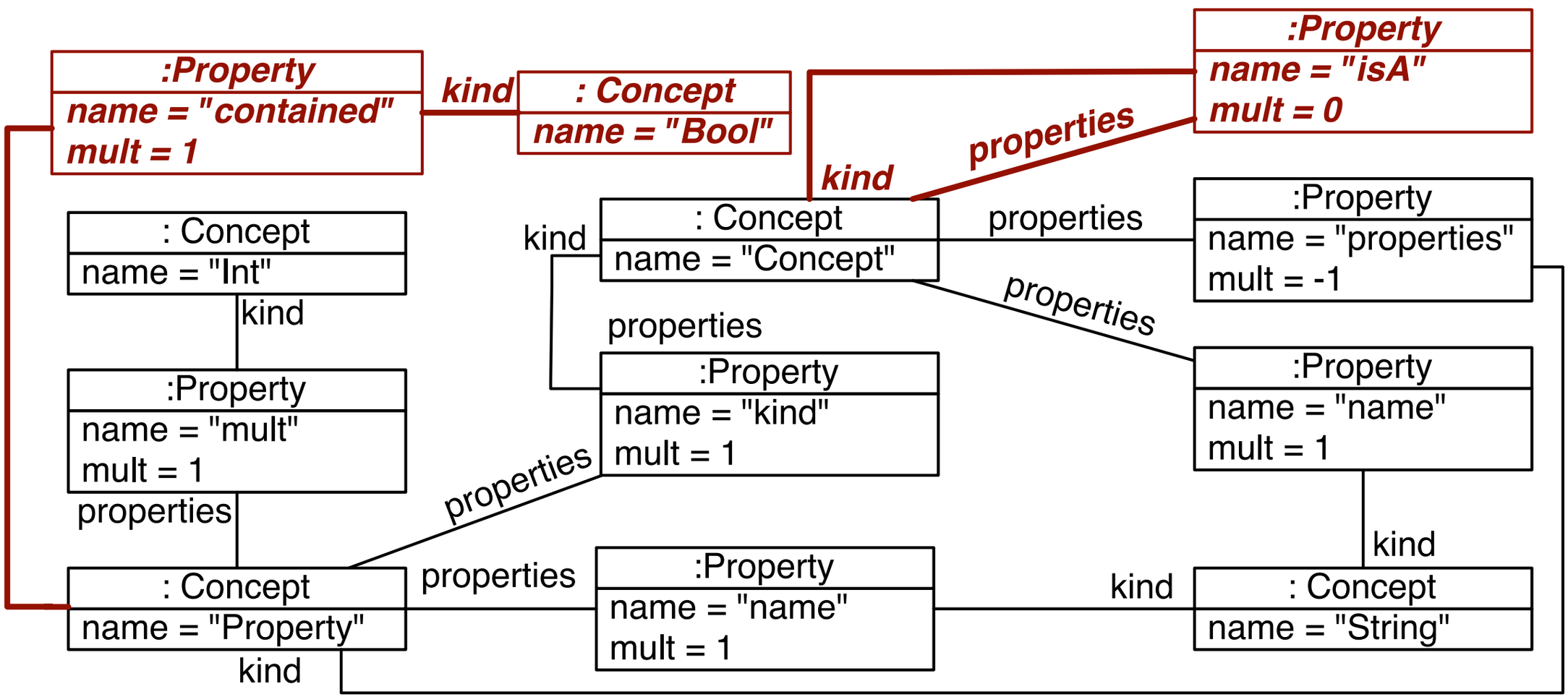
Code Generation



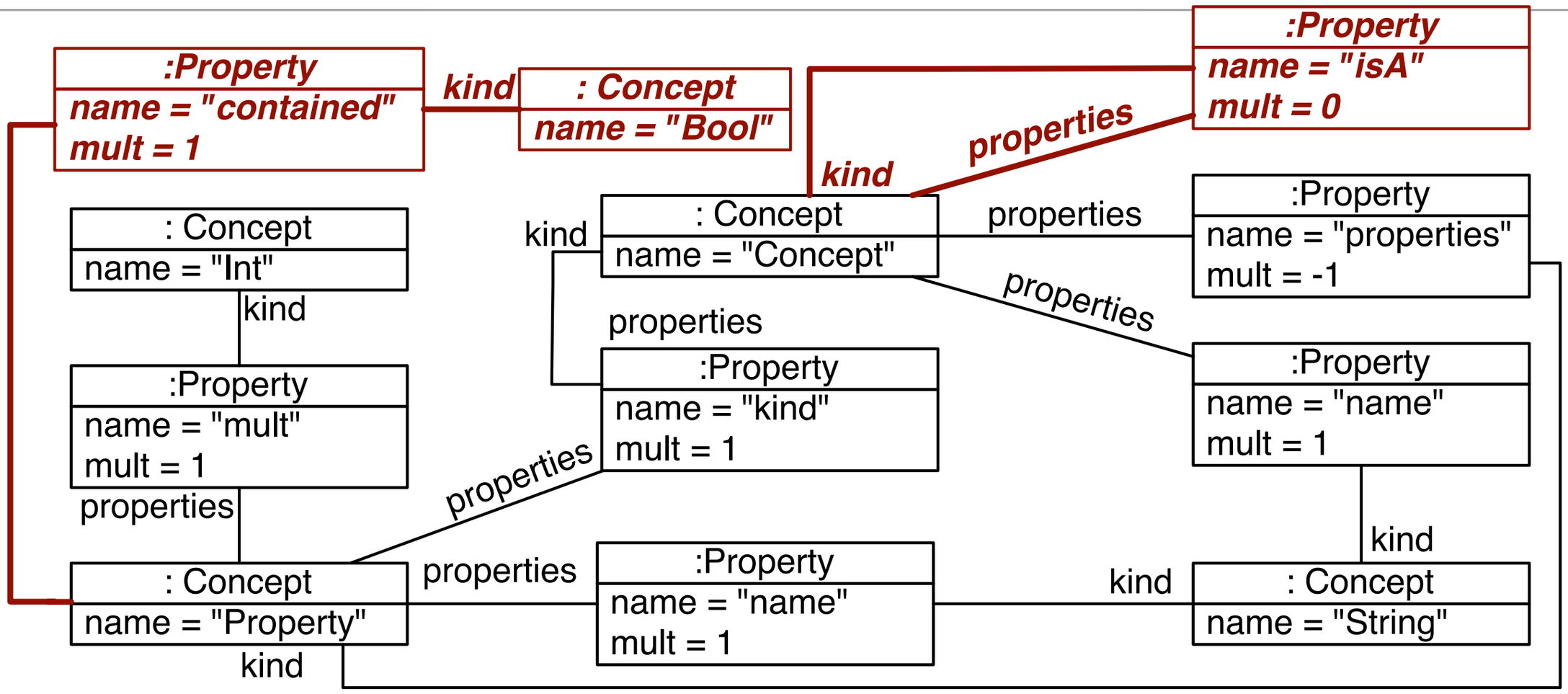


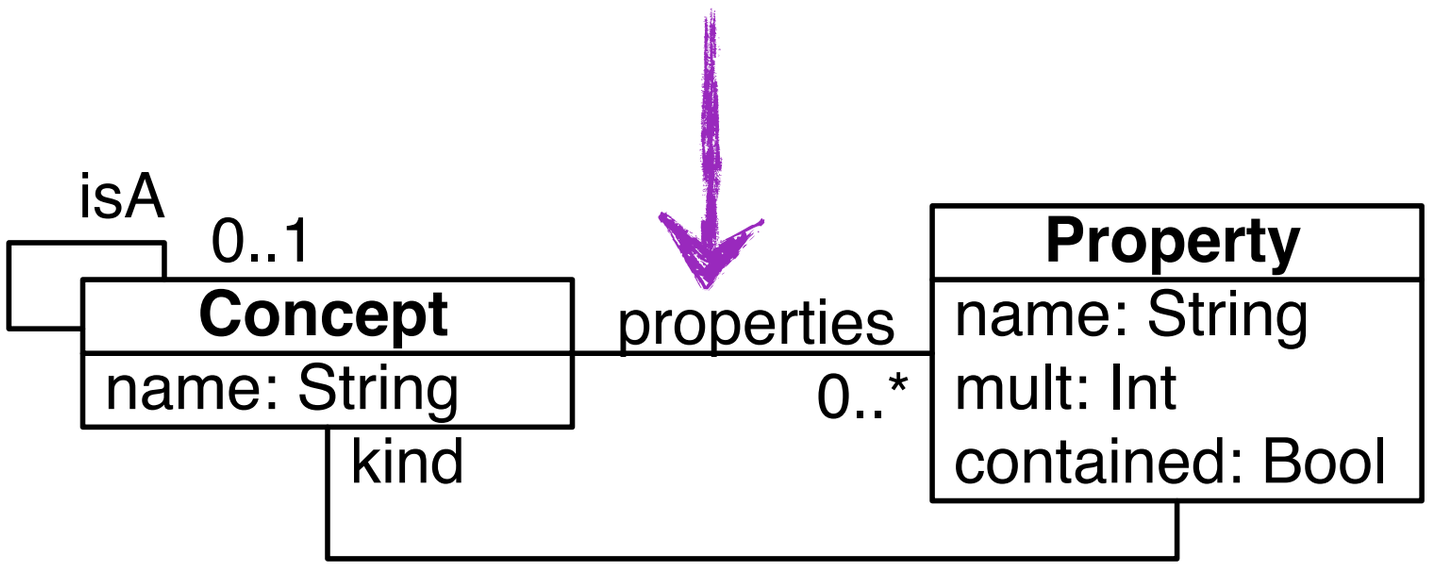
Step #3: Adding **Inheritance** & **Containment**



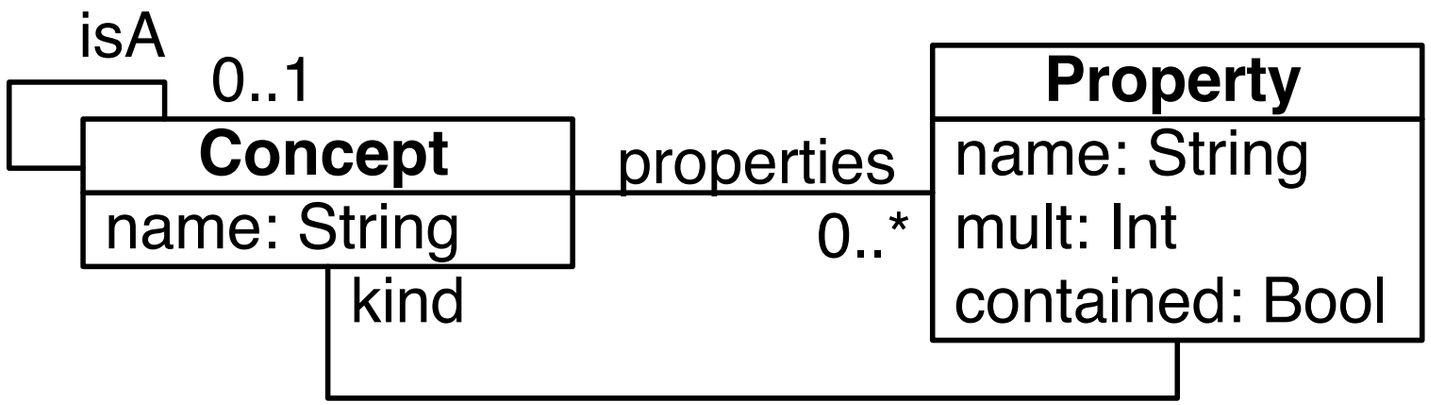


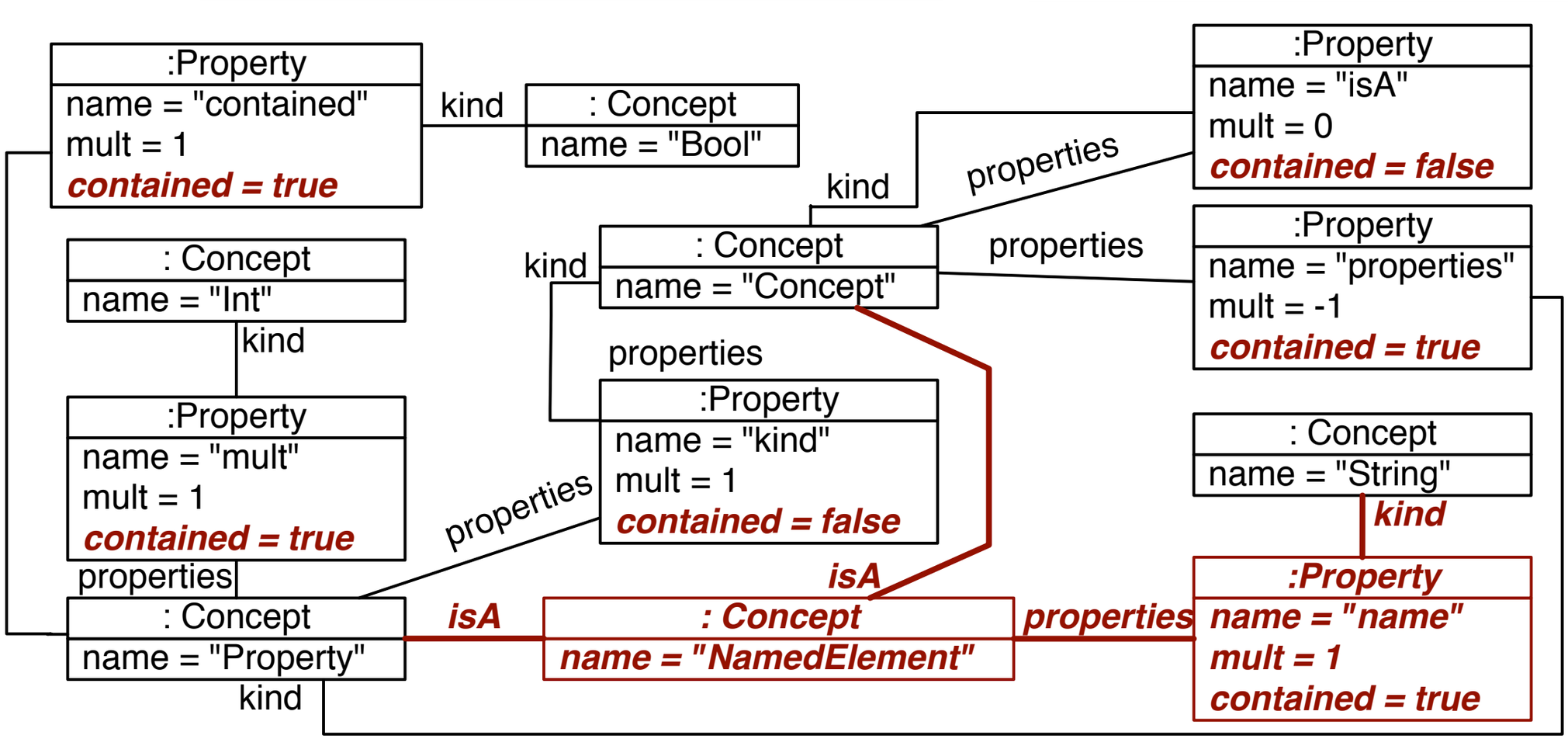
Code Generation



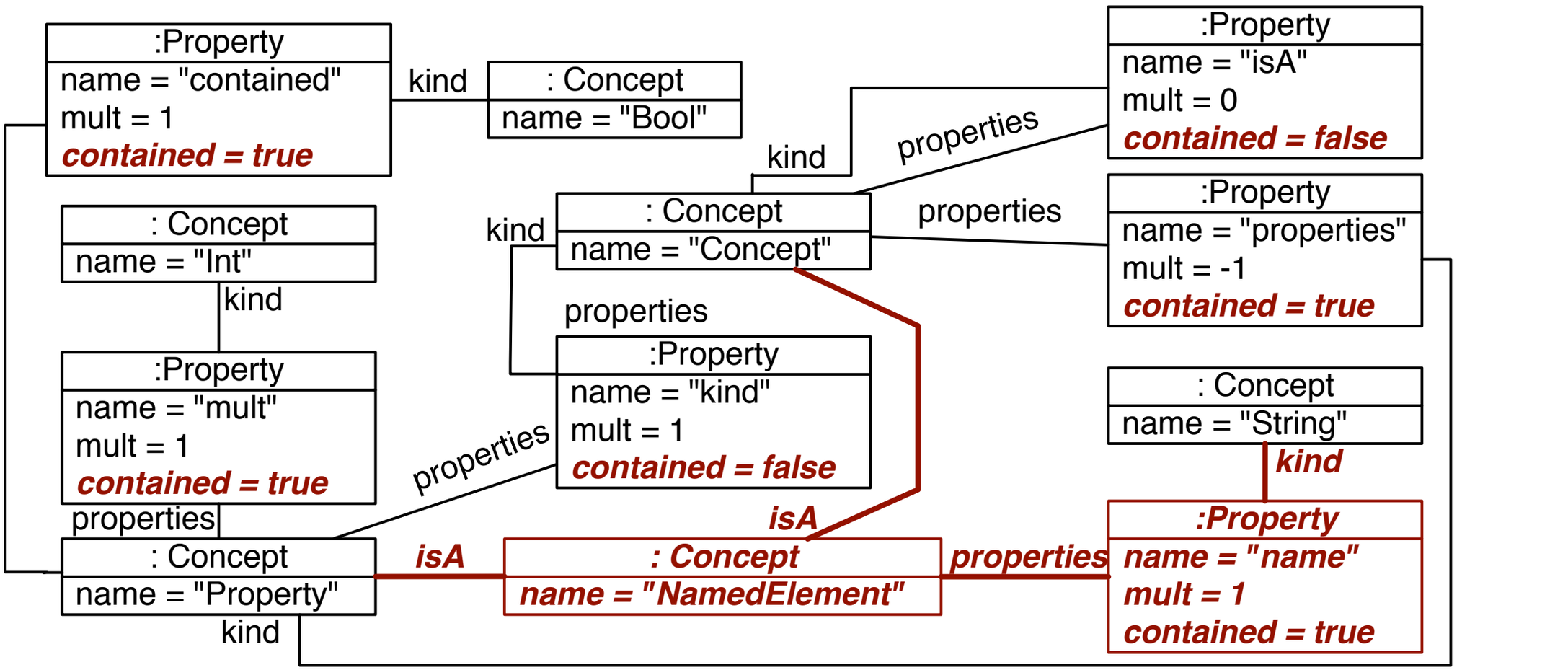


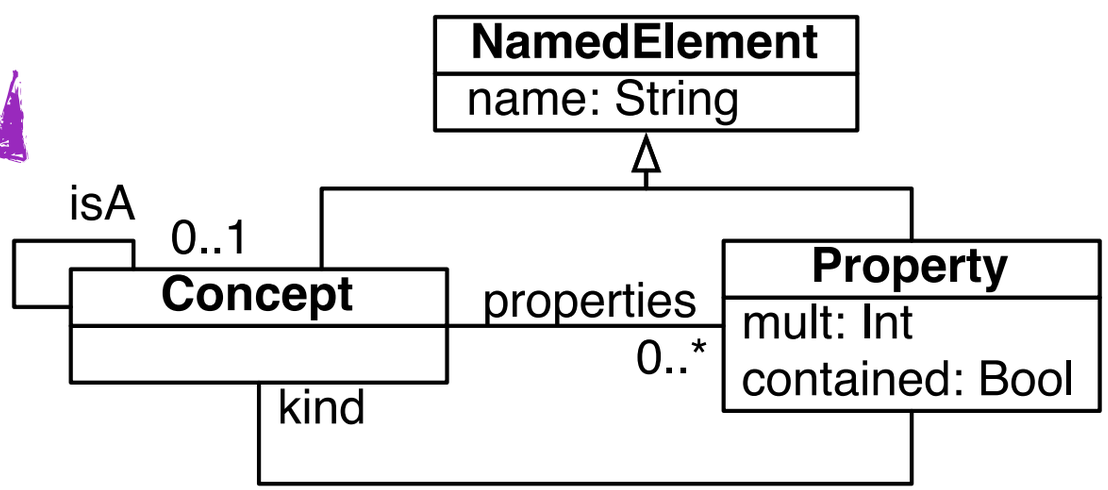
Step #4: Expressive enough to **refactor**





Code Generation





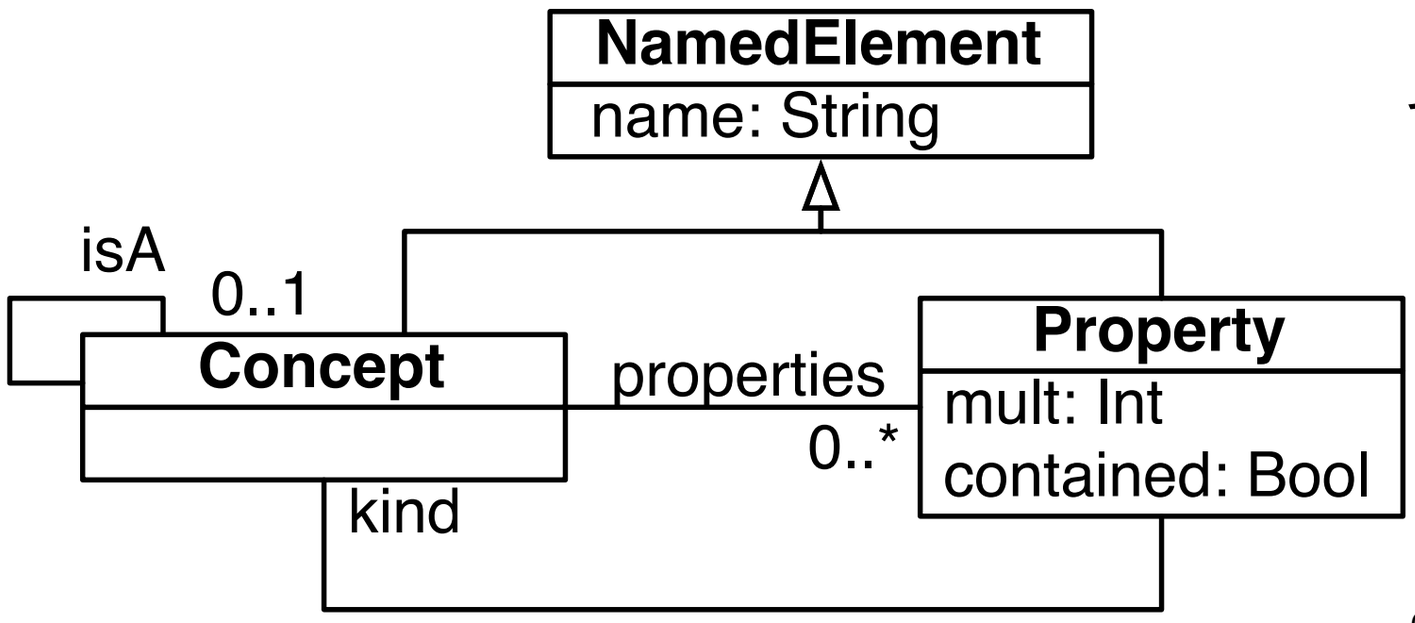
We just **bootstrapped**

a modeling language!

conforms to

Concept
name: String

conforms to



Time for a break!

From models to domain-specific languages

Secret Compartments

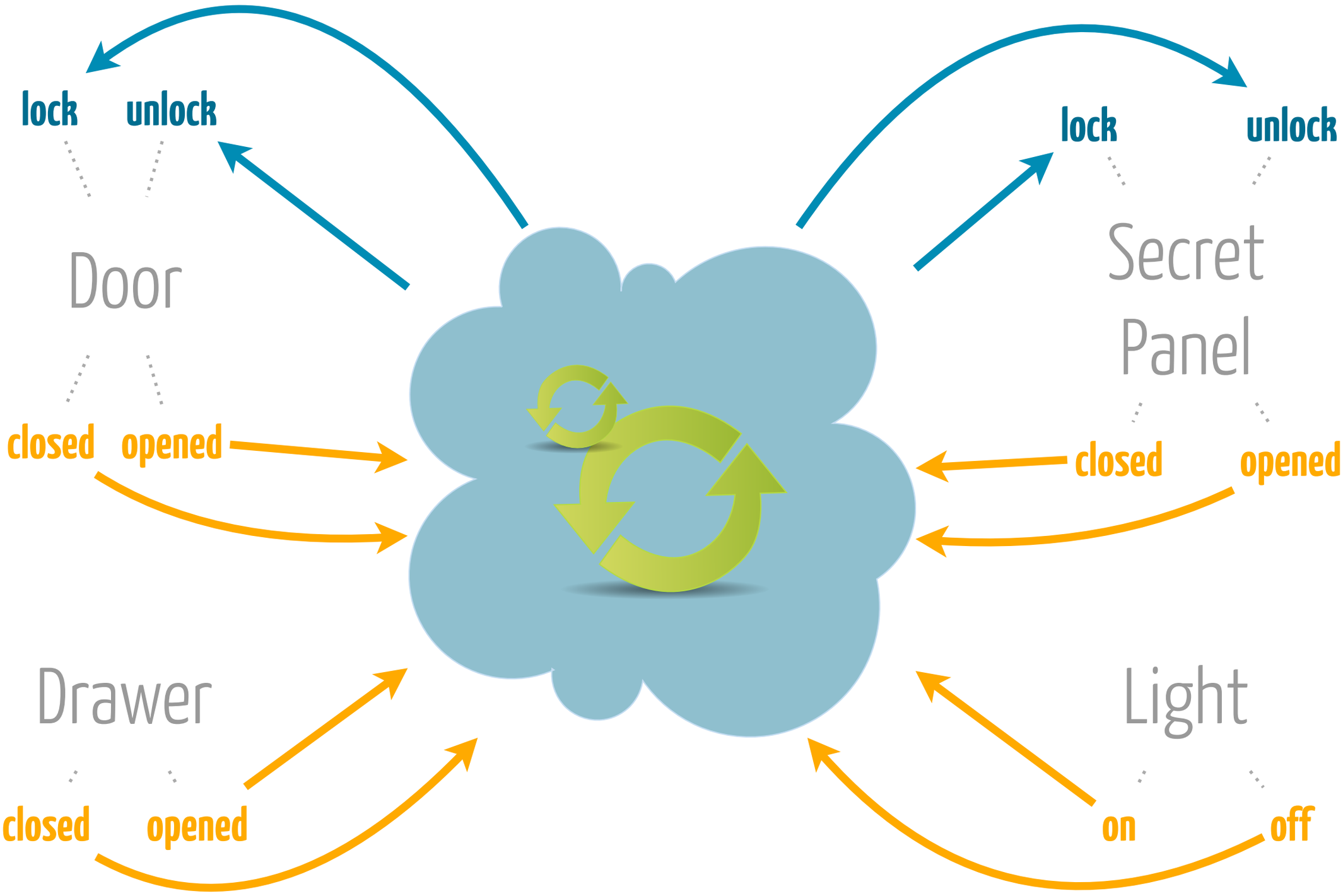
- To open the **Diagon Alley** wall:
  - «From the trash can,
  - Tap three bricks up,
  - and two across.»
- To open the **secret door**:
  - Pull the left candlestick,
  - Push the 3rd brick from the right,
  - Pull Vol. 7 of the Encyclopedia.

Miss Grant's Secret Drawer

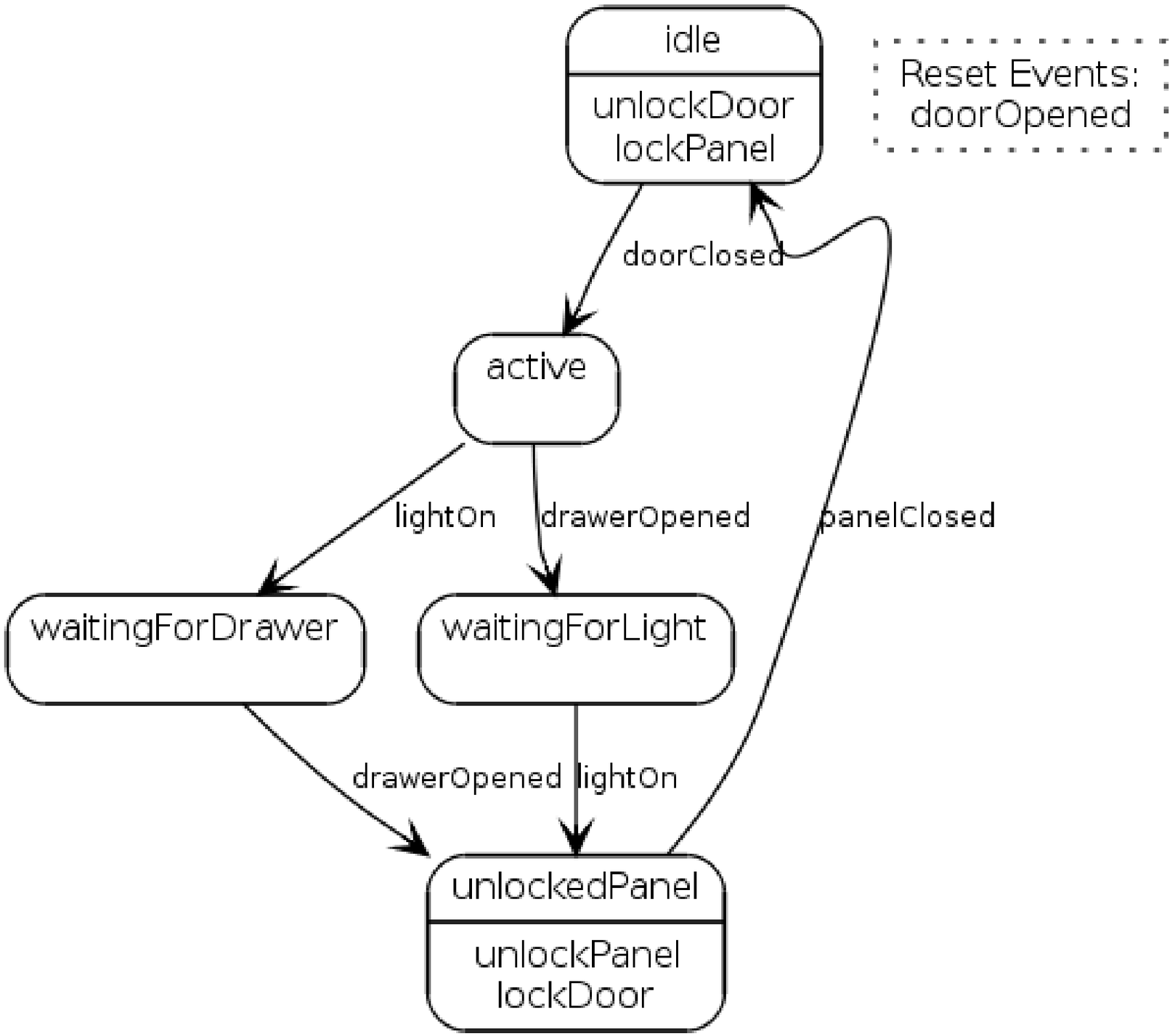
«To **open** it, she has to **close the door**, then **open the second drawer** in her chest and **turn her bedside light on**—in either order. Once these are done, **the secret panel is unlocked** for her to open.»

[Fowler]

Miss Grant's Environment



Formalising Miss Grant's Controller

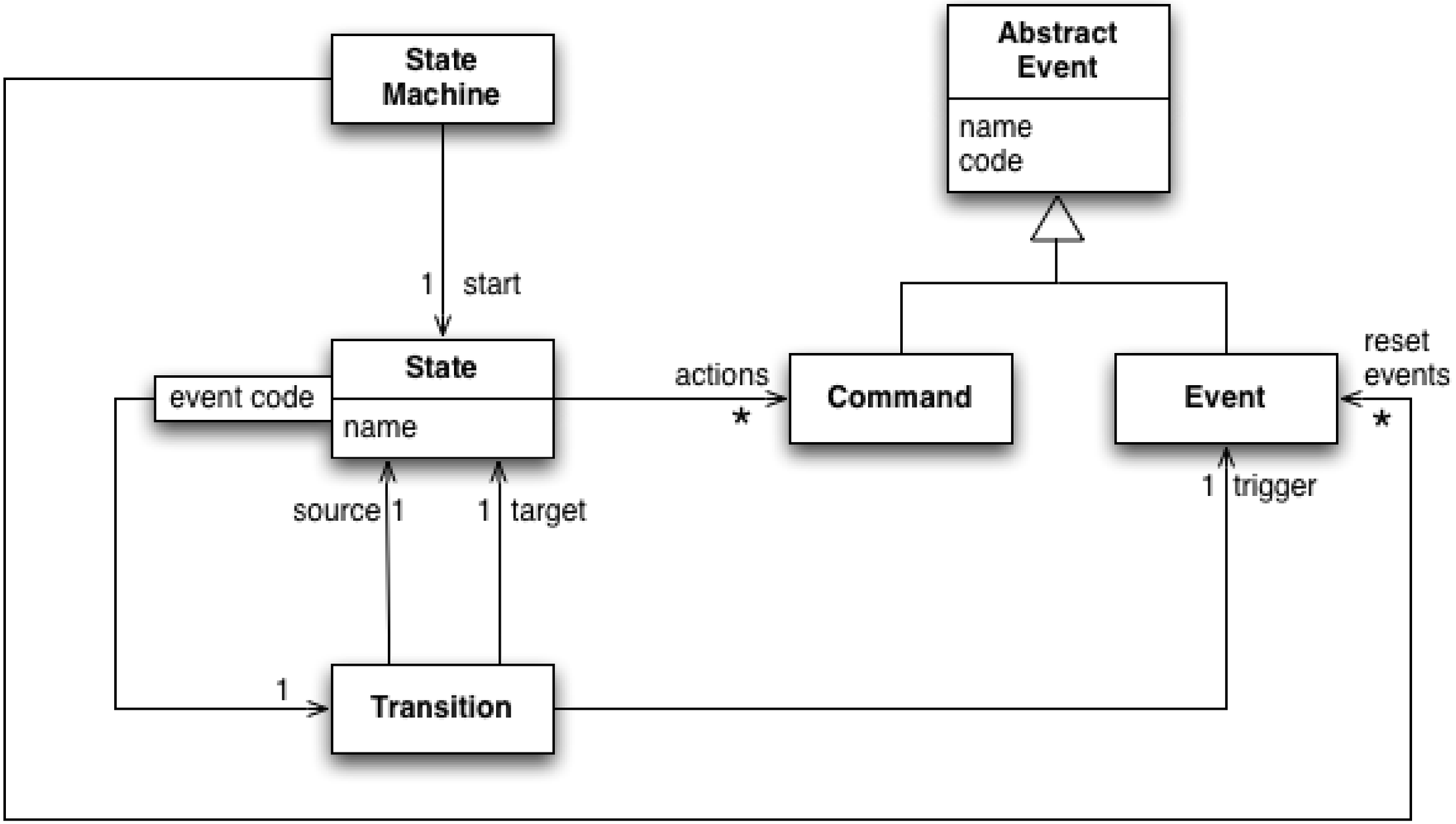


[Fowler]

Modeling Controllers

- **Events**:
  - A code sent **by** the environment
  - Inform about context changes
- **Commands**:
  - A code sent **to** the environment
  - Change the context (e.g., unlock)
- **States**
  - Controller's current situation
  - Might trigger Commands
  - Reacts to events with Transitions
- **Transitions**
  - Associate an event to a next State

Associated Structural Concepts



[Fowler]

Programming the Controller [1/2]

```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");


State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");

State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);
```

[Fowler]

Programming the Controller [2/2]

```
idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd); idle.addAction(lockPanelCmd);


activeState.addTransition(drawerOpened, waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);
```

```
waitingForLightState.addTransition(lightOn, unlockedPanelState);
waitingForDrawerState.addTransition(drawerOpened,
unlockedPanelState);


unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);


machine.addResetEvents(doorOpened);
```

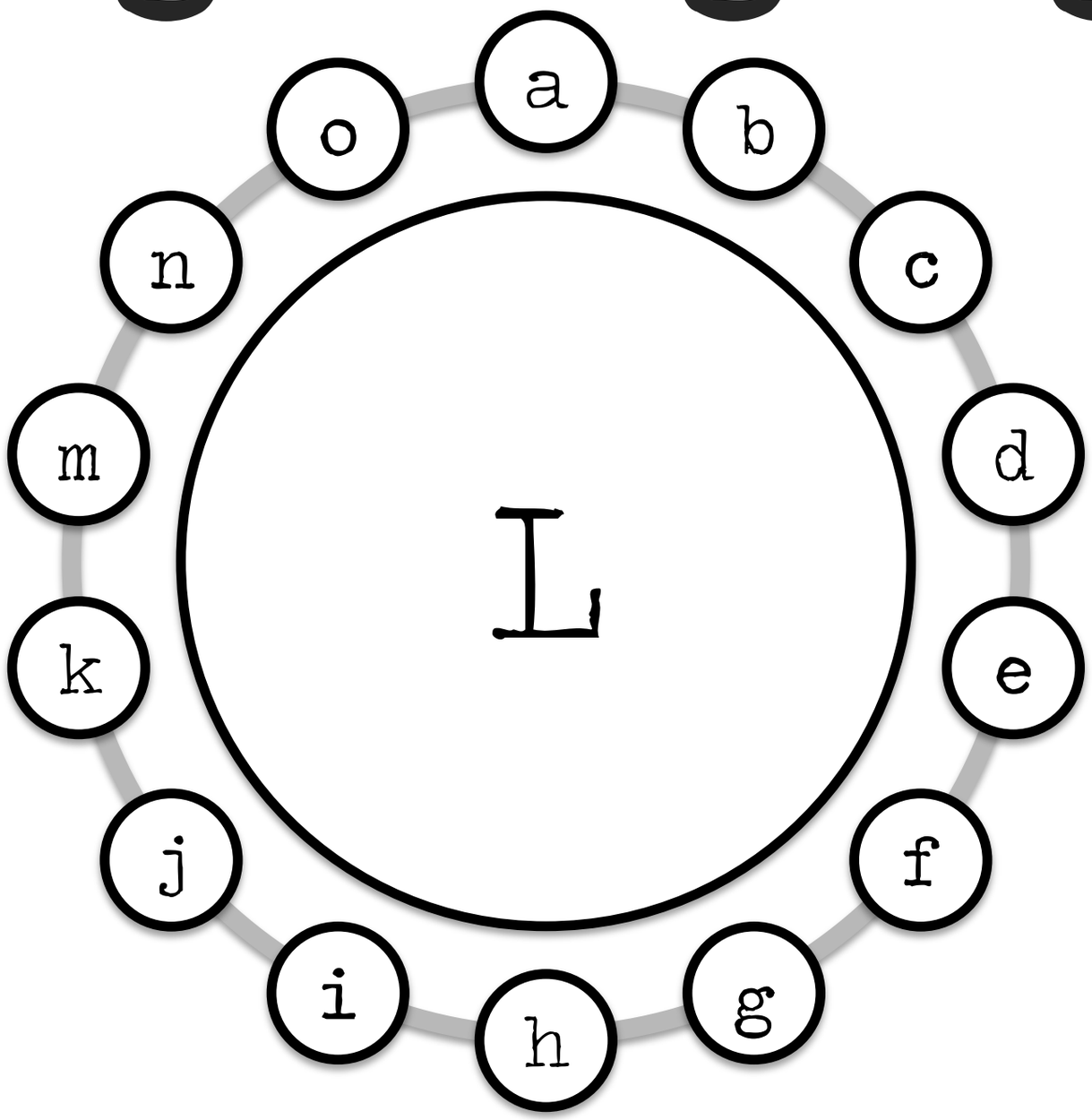
[Fowler]

Wait!

Where the **hell**

is my **domain**?

Big Language



with many first class concepts!

[Voelter]

What the hell is my domain?

Controller

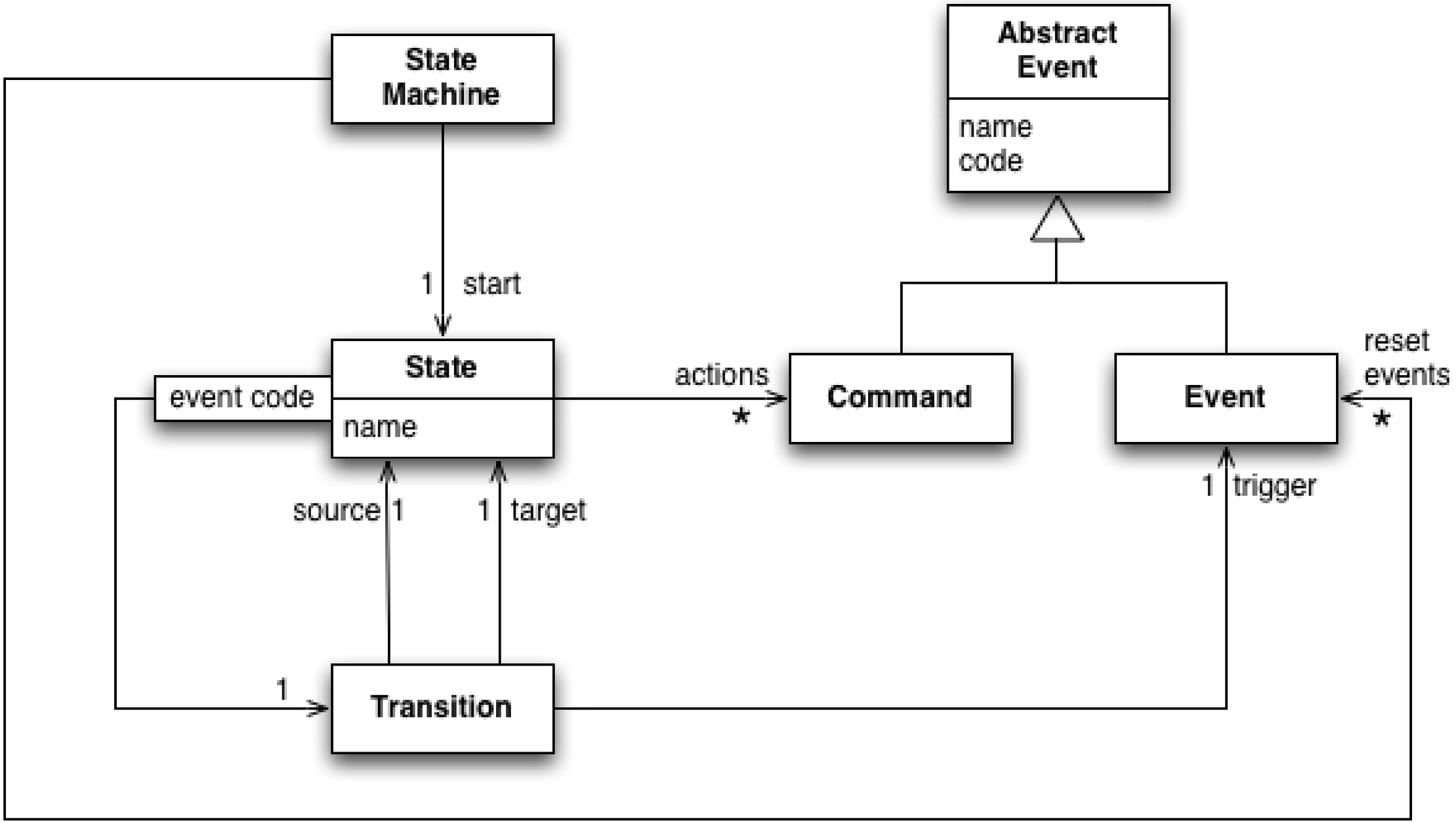
From «**Programming** the **model** of the **Controller**» ...

… to

«Modeling

the Controller»

Associated Structural Concepts



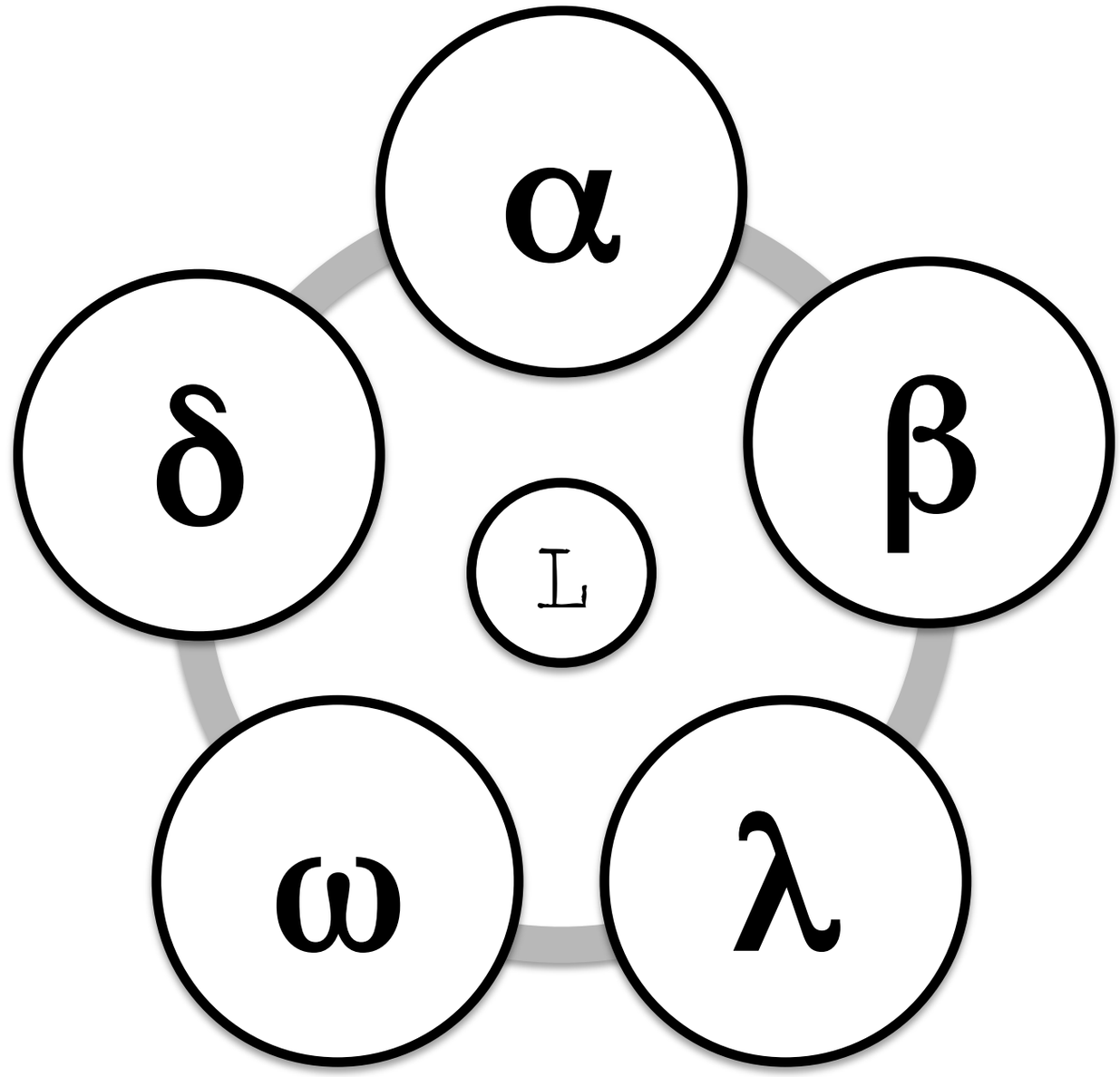
[Fowler]

```
events
  doorClosed      D1CL
  drawerOpened    D2OP
  lightOn         L1ON
  doorOpened      D1OP
  panelClosed     PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end
```

```
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

[Fowler]

Small Language



with a few, orthogonal and poweful concepts

[Voelter]

```
events
  doorClosed  D1CL
end

state idle
  doorClosed => active
end
```

input

populates

:State
name = "idle"

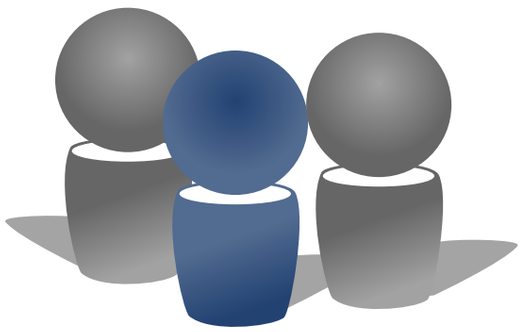
:Transition

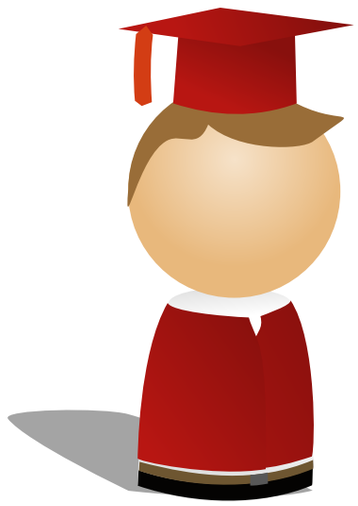
:Event
name = "doorClosed"
code = "D1CL"

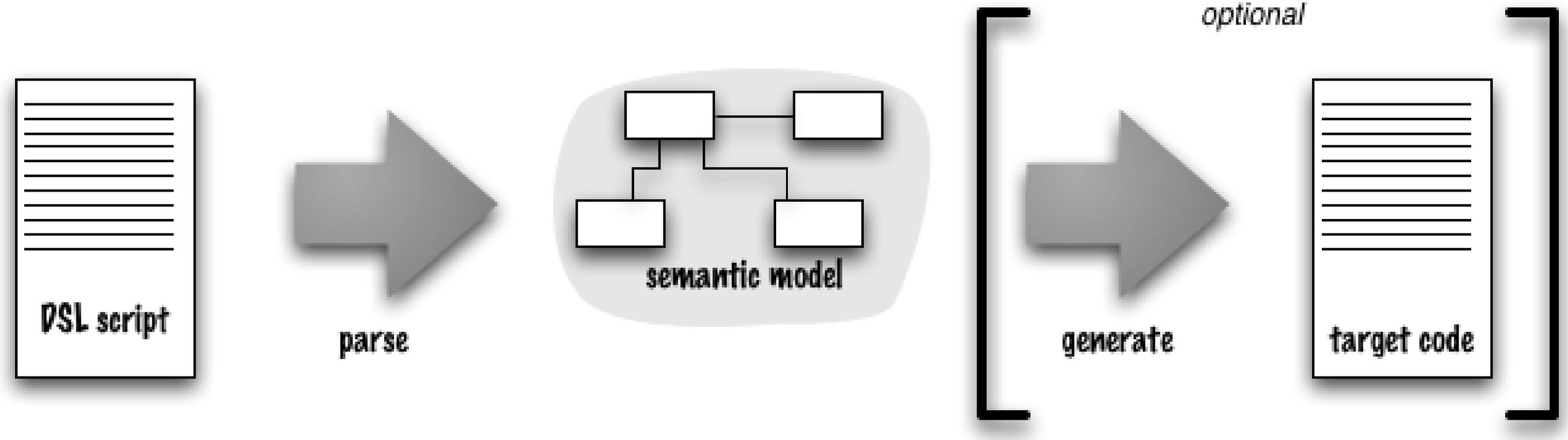
:State
name = "active"

semantic model

[Fowler]





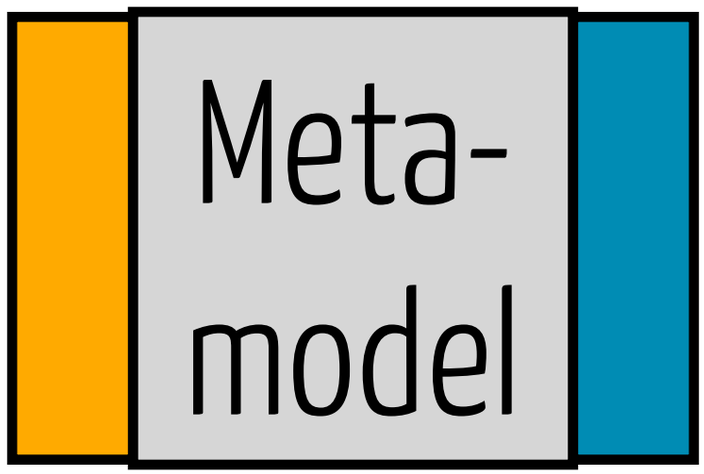


[Domain-Specific Languages]

Semantic Model ≡ Meta-model

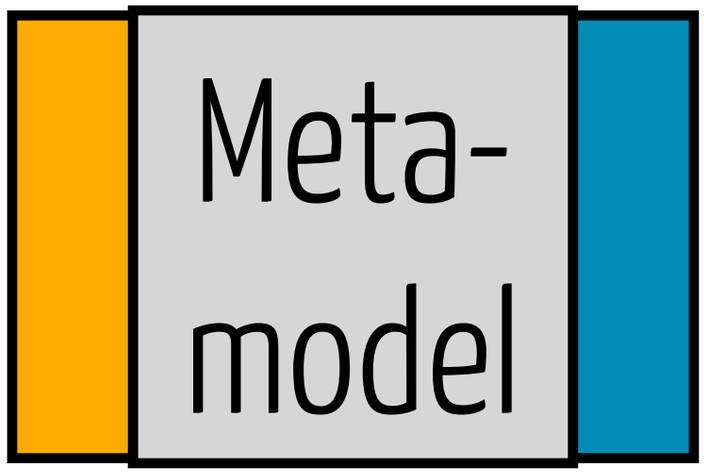
- The «**essence**» of a domain, according to a **given viewpoint**
- Should be **usable with or without a DSL**
- Interfaces to **exploit** the meta-model

Population Interface

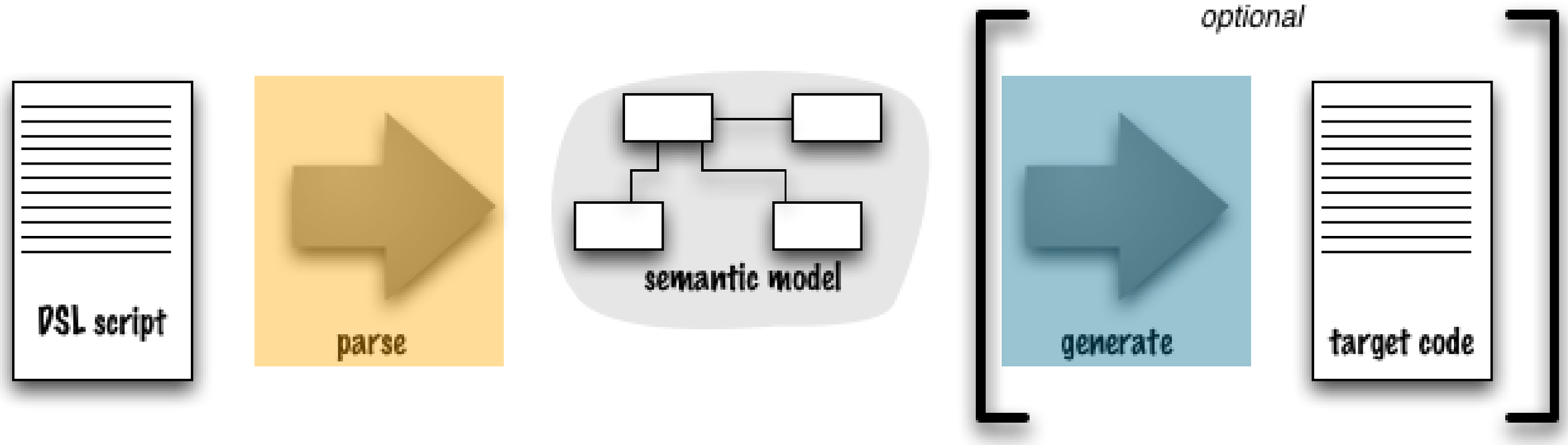


Operational Interface

Population
Interface



Operational
Interface



[Domain Specific Languages]

DSLs

versus

GPLs

This is not a compilation course!

	more in GPLs	more in DSL
Domain Size	large and complex	
Designed by	guru or committee	
Language Size	large	
Turing-completeness	almost always	
User Community	large, anonymous and widespread	
In-language abstraction	sophisticated	
Lifespan	years to decades	
Evolution	slow, often standardized	
Incompatible Changes	almost impossible	

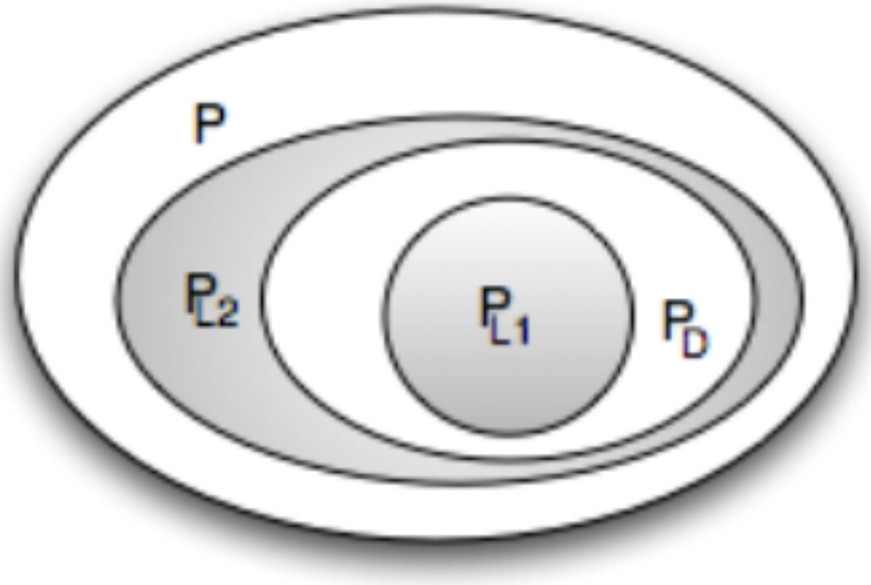
?

[Voelter]

	more in GPLs	more in DSL
Domain Size	large and complex	smaller and well-defined
Designed by	guru or committee	a few engineers and domain experts
Language Size	large	small
Turing-completeness	almost always	often not
User Community	large, anonymous and widespread	small, accessible and local
In-language abstraction	sophisticated	limited
Lifespan	years to decades	months to years (driven by context)
Evolution	slow, often standardized	fast-paced
Incompatible Changes	almost impossible	feasible

[Voelter]


Your
DSL
You
#ABCWorldNews



A DSL L_D **for** D **is a** language that is **specialized** to encoding P_D programs.

more efficient
smaller

[Voelter]

Shorter Programs

More Accessible Semantics

[Voelter]

For a limited Domain!

Domain Knowledge encapsulated in language

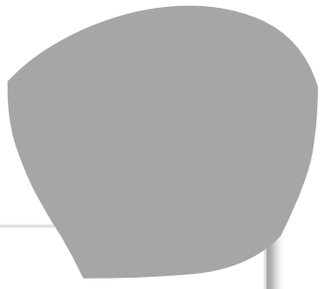
[Voelter]

"Using patterns in a language is a symptom of lack of specialization

[Voelter]

Overspecification!
Requires Semantic Analysis!

```
int[] arr = ...
for (int i=0; i<arr.size(); i++) {
    sum += arr[i];
}
```

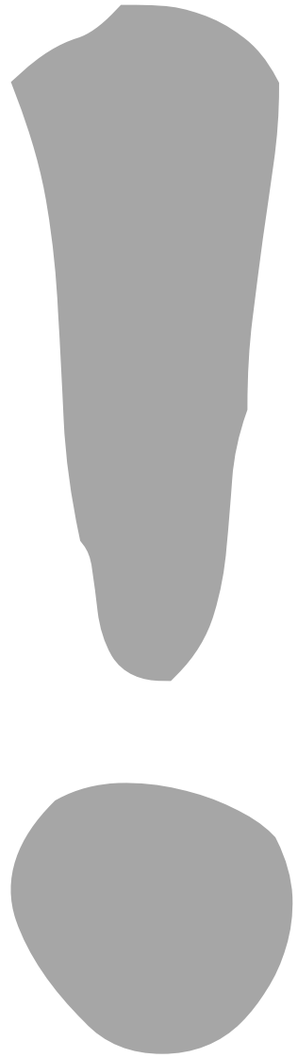


```
int[] arr = ...
List<int> l = ...
for (int i=0;  i<arr.size(); i++) {
    l.add( arr[i] );
}
```

[Voelter]

Linguistic Abstraction

```
for (int i in arr) {
    sum += i;
}
```



Declarative!
Directly represents Semantics.

```
seqfor (int i in arr) {
    l.add( arr[i] );
}
```

[Voelter]

Def: DSL

A DSL is a **language** at D that provides **linguistic abstractions** for **common patterns and idioms** of a language at D-1 when used within the domain D.

[Voelter]

Def: DSL cont'd

A good DSL does **not** require the use of patterns and idioms to express **semantically interesting** concepts in D.

Processing tools do not have to do "semantic recovery" on D programs.

Declarative!

[Voelter]

Computation Models

Decision Table

Premium Customer	X	X	Y	Y	N	N
Priority Order	Y	N	Y	N	Y	N
International Order	Y	Y	N	N	N	N
Fee	150	100	70	50	80	60
Alert Rep	Y	Y	Y	N	N	N

conditions

consequences

[Fowler]

Example: FIT

result.htm - Microsoft Word

File Edit View Insert Format Tools Table Window Documents To Go Help

77% Recount

Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR E

http://fit.c2.com/wiki.cgi?IntroductionToFit

Business Rules

```
if
 passenger.frequentFlier
then
 passenger.priorityHandling = true;

if
   mileage > 25000
then
 passenger.frequentFlier = true;
```

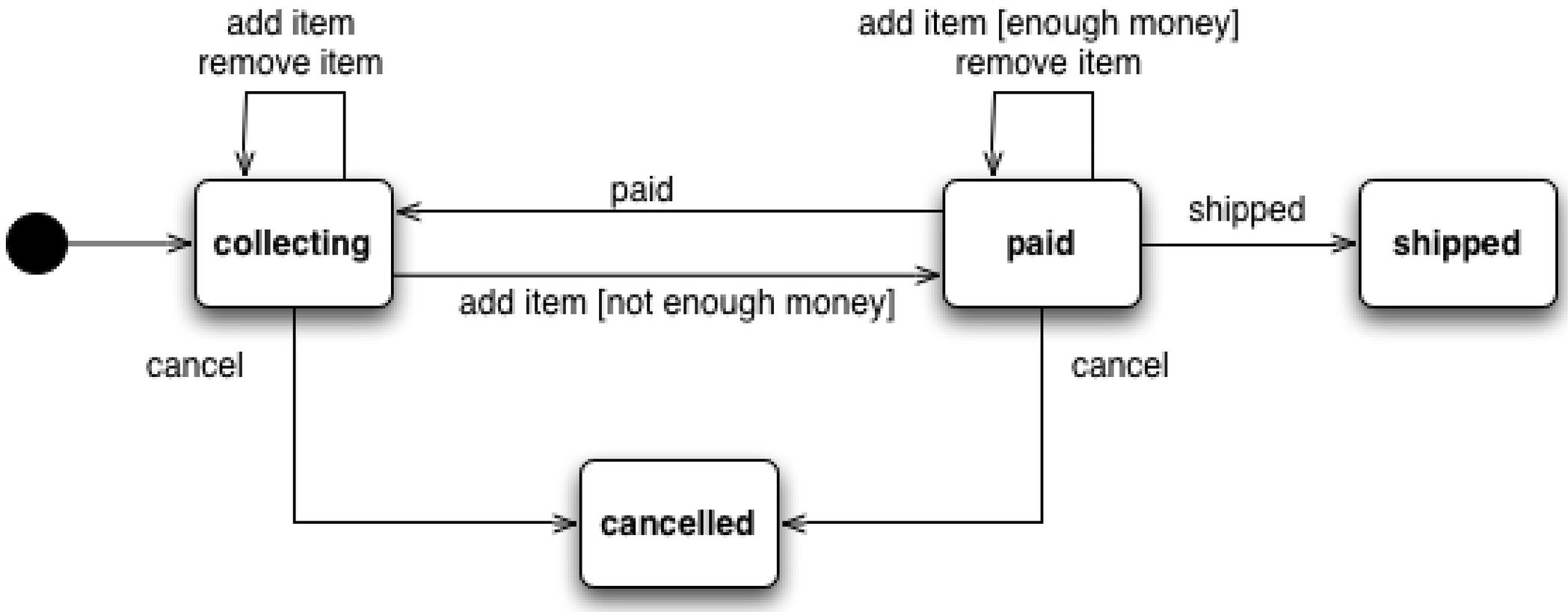
[Fowler]

Example: DRools

```
package com.comp.drools;

import function com.comp.drools.DroolsHelper.error;
import function com.comp.drools.DroolsHelper.ok;


rule "Account balance is less than 100"
    no-loop
 when
  $account : Account( balance < 100 )
 then
  error(kcontext, $account); // consequence
end

rule "Account balance is More than 100"
    no-loop
 when
   $account :Account( balance > 100 )
 then
  ok(kcontext, $account);
end
```

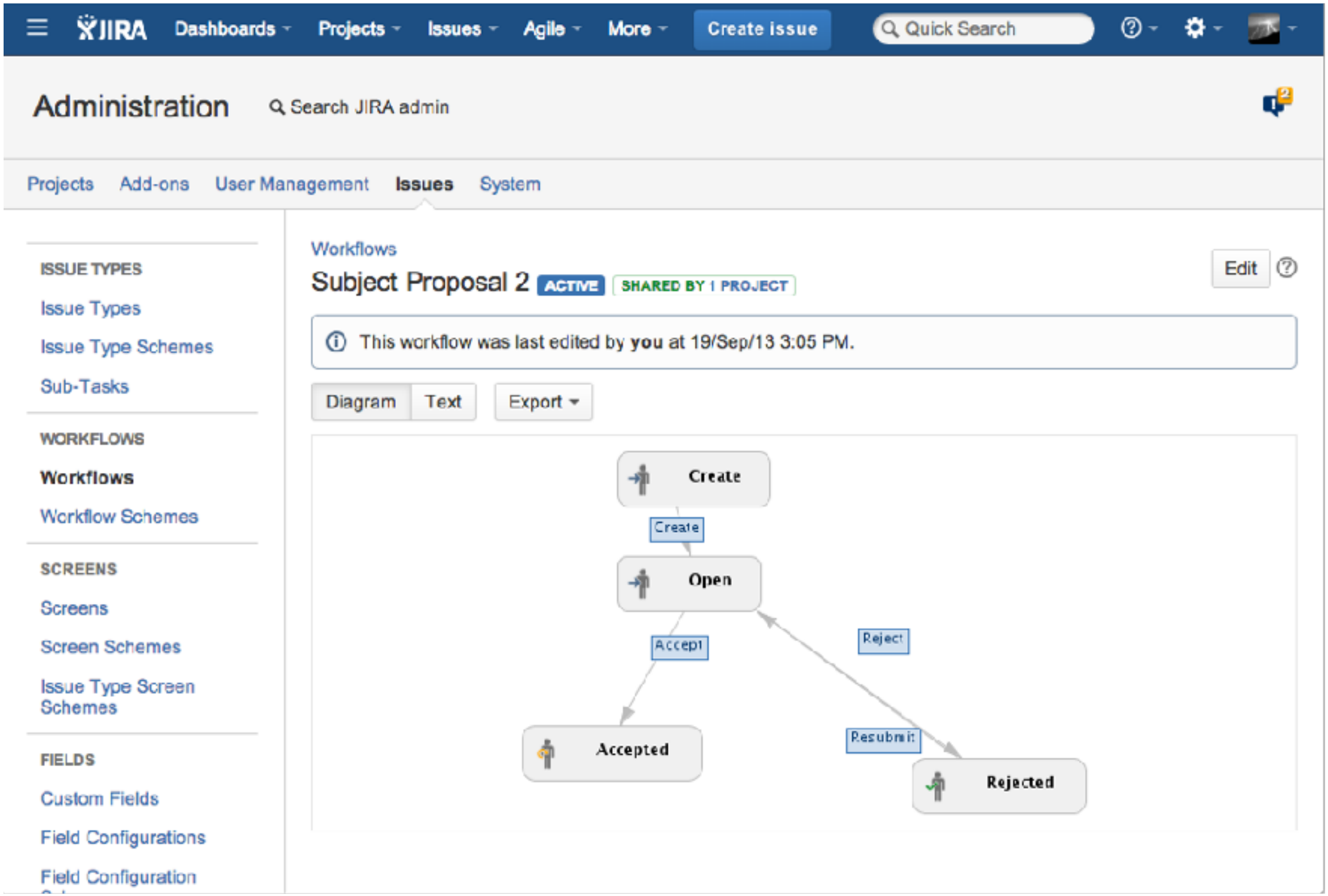

http://www.commonj.com/blogs/2010/11/13/drools-tutorial-by-examples/

State Machine

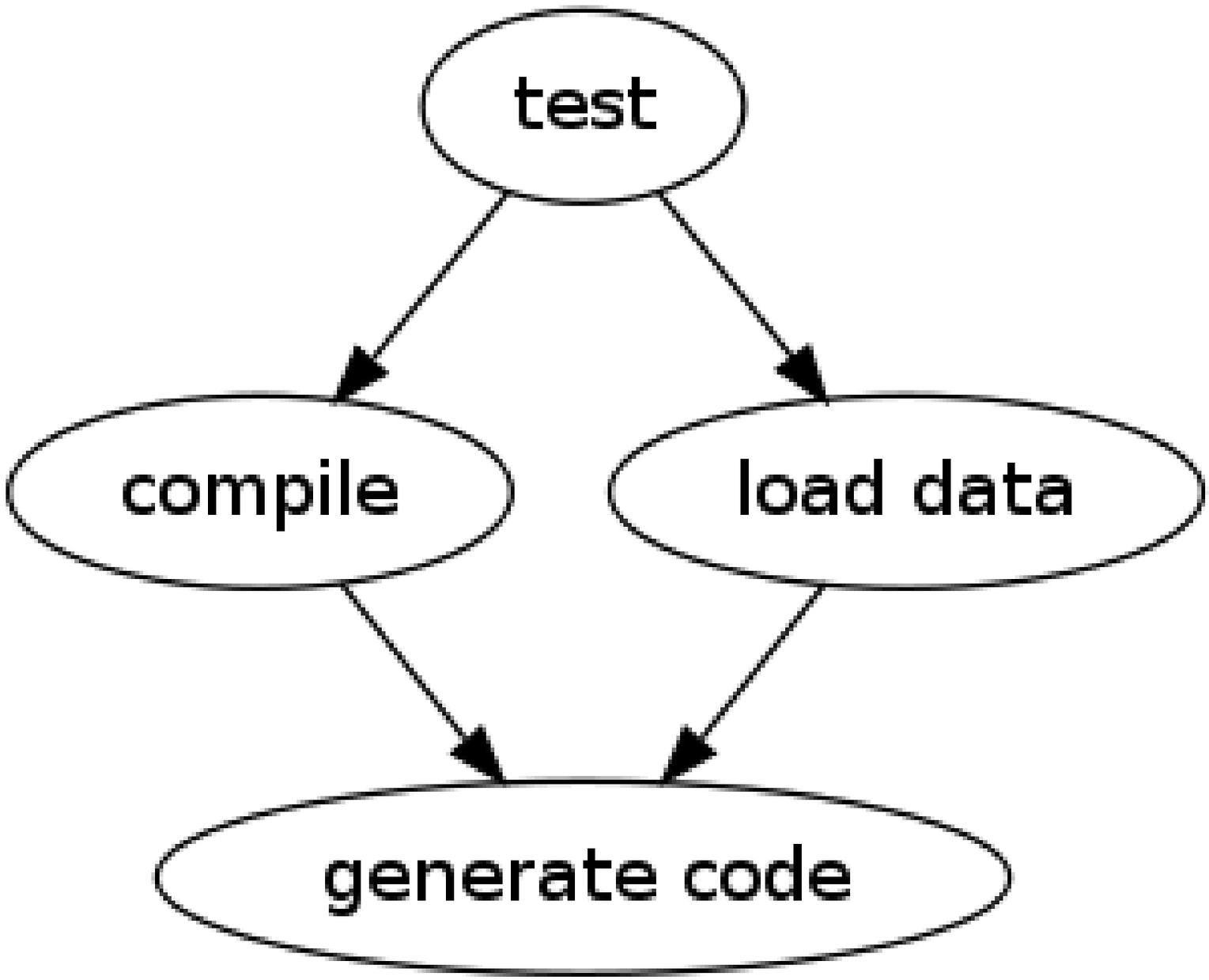


[Fowler]

Example: JIRA Workflows



Dependency Network



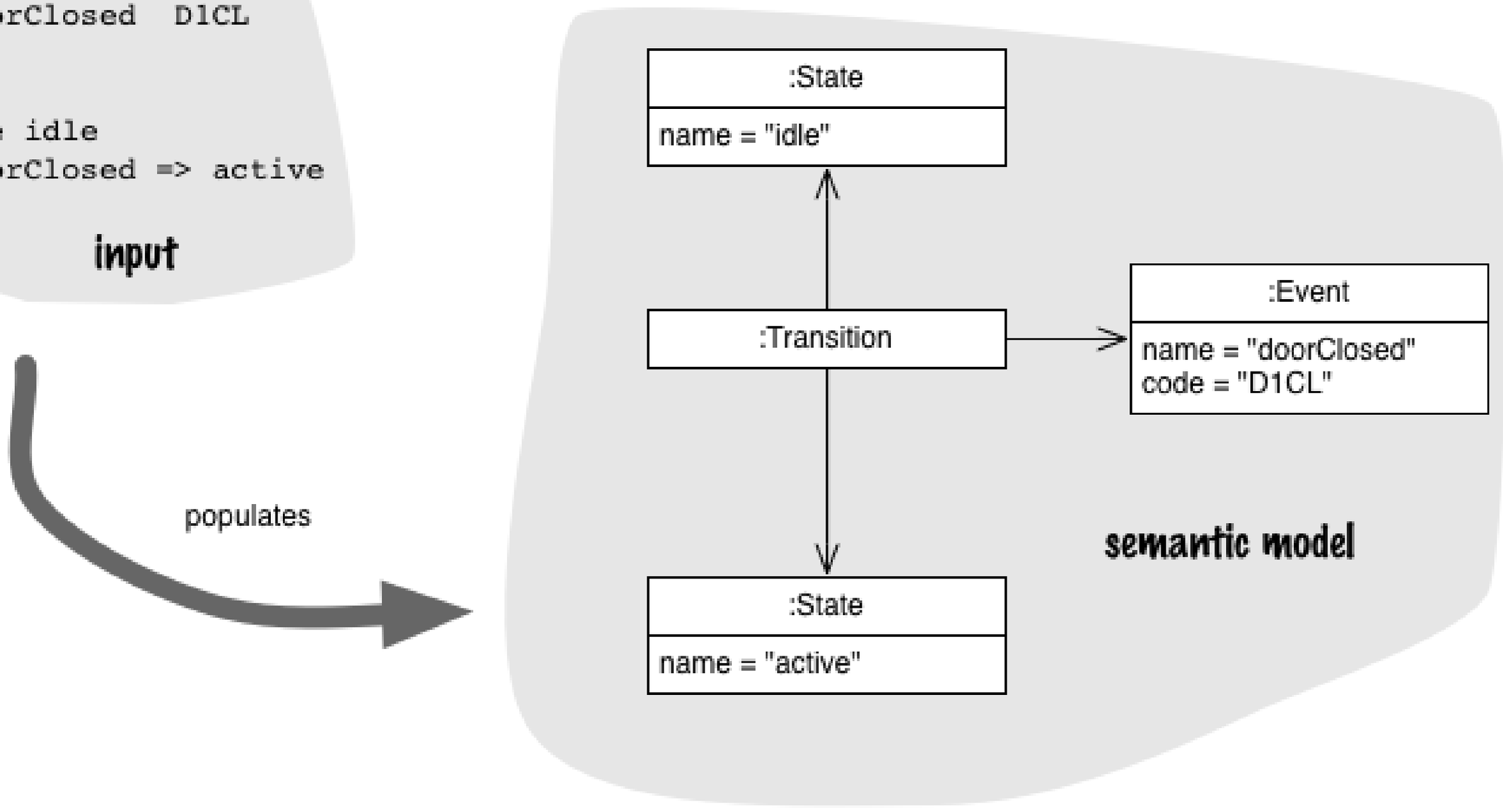
Example: Build Automation

```
COMPILER = gcc
FLAGS    = -O -Wall
PROGRAM  = myProg

$(PROGRAM): main.o mathlib.o
   $(COMPILER) $(FLAGS) -o \
           $(PROGRAM) main.o mathlib.o

main.o: main.c mathlib.h
   $(COMPILER) $(FLAGS) -c main.c

mathlib.o: mathlib.c mathlib.h
   $(COMPILER) $(FLAGS) -c mathlib.c
```

DSL implementation overview

```
events
  doorClosed  D1CL
end

state idle
  doorClosed => active
end
```

input



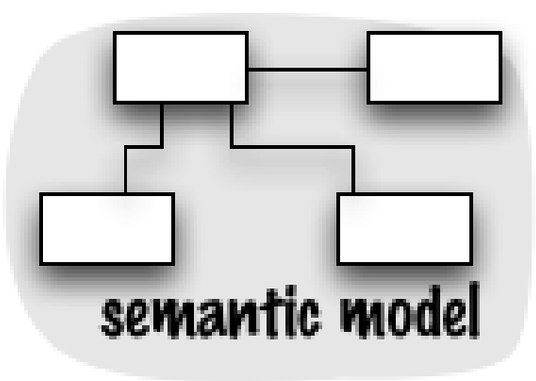
Delimiter-directed Translation

```
state unlockedPanel
  actions unlockPanel lockDoor
  panelClosed => idle
end
```

```
for each line in input
```

```
parse (line)
```

```
state unlockedPanel
actions unlockPanel lockDoor
panelClosed => idle
end
```

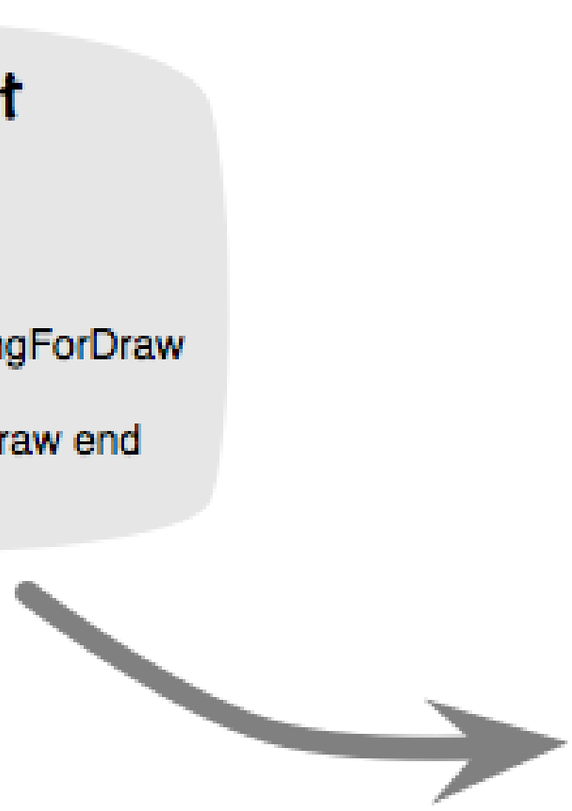


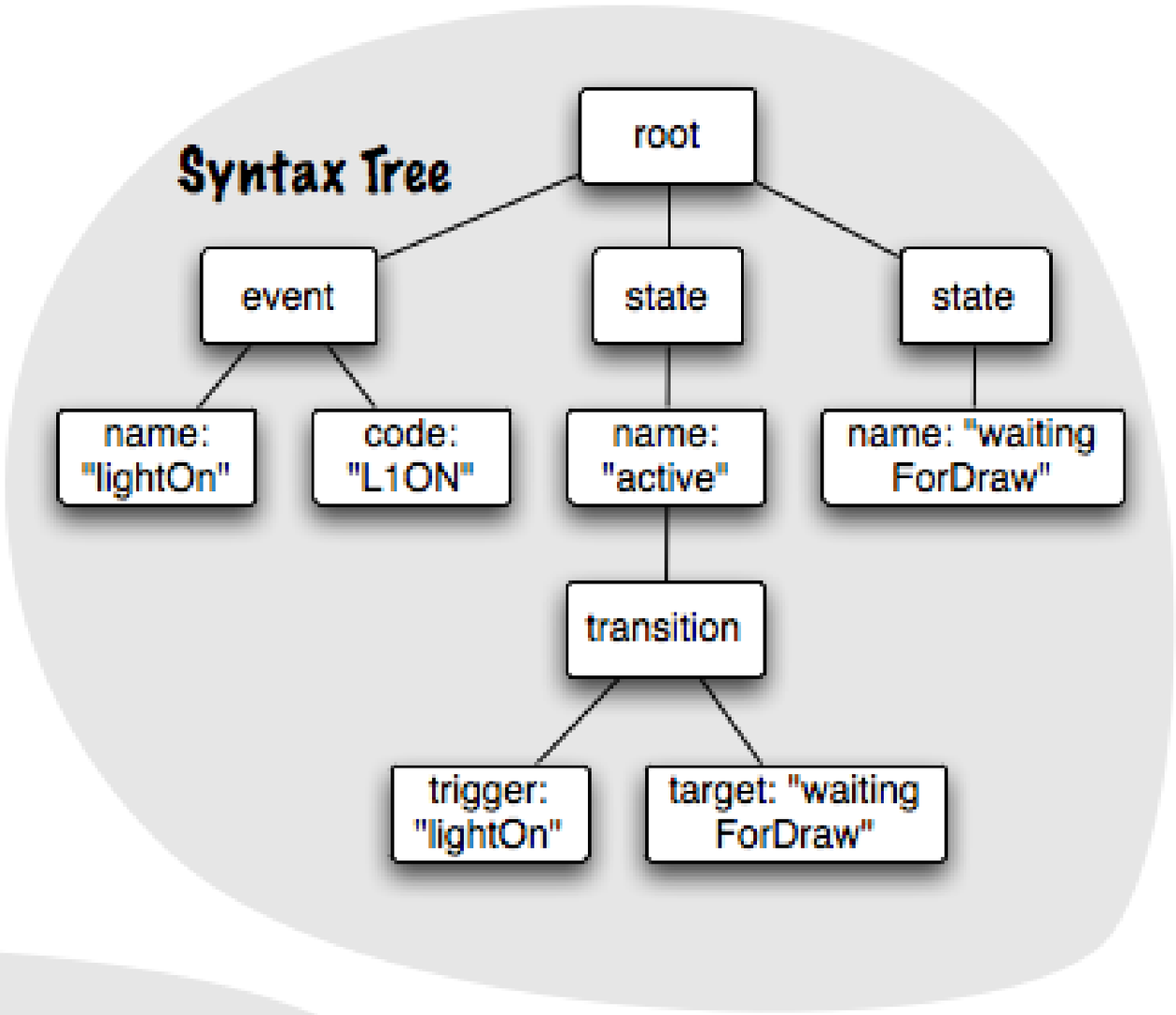
Easy

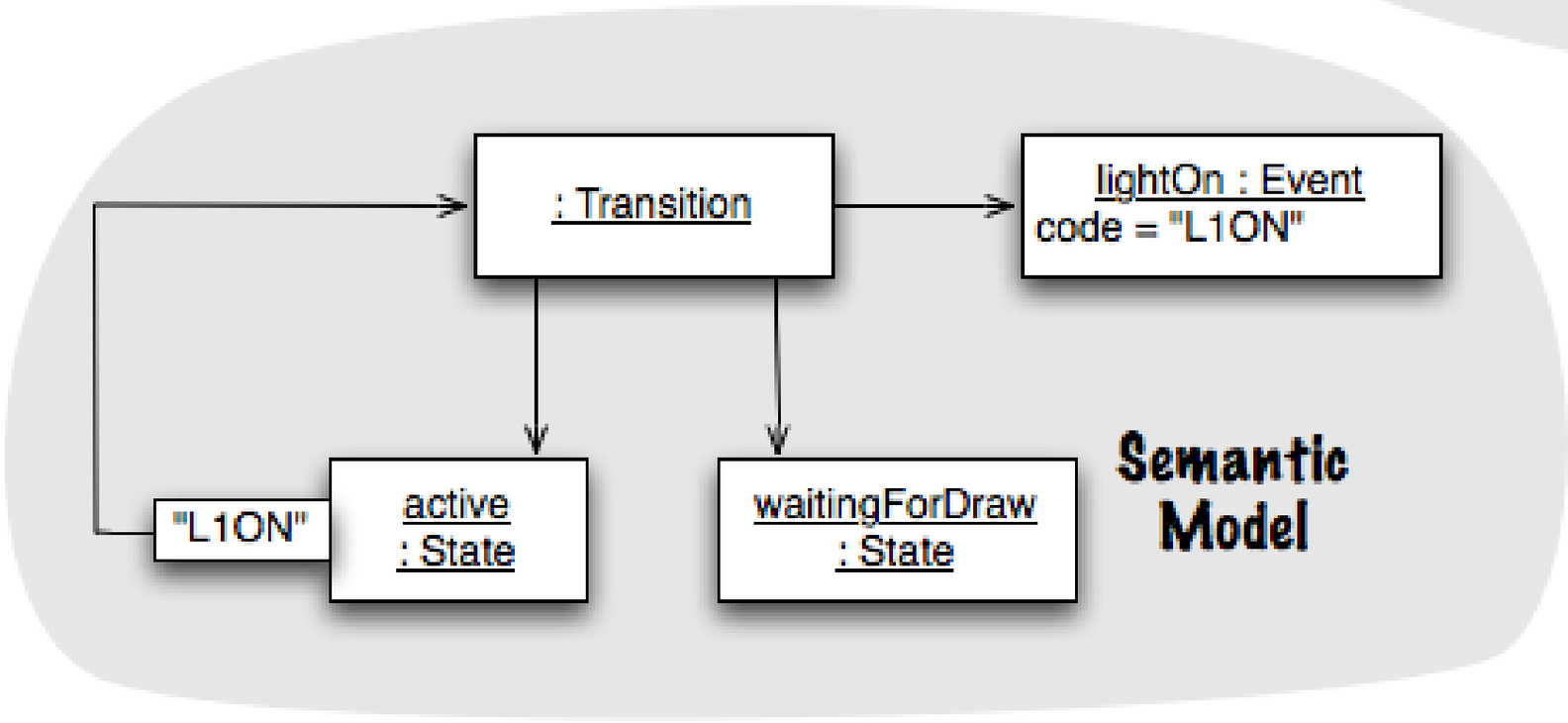
Limited

DSL Text

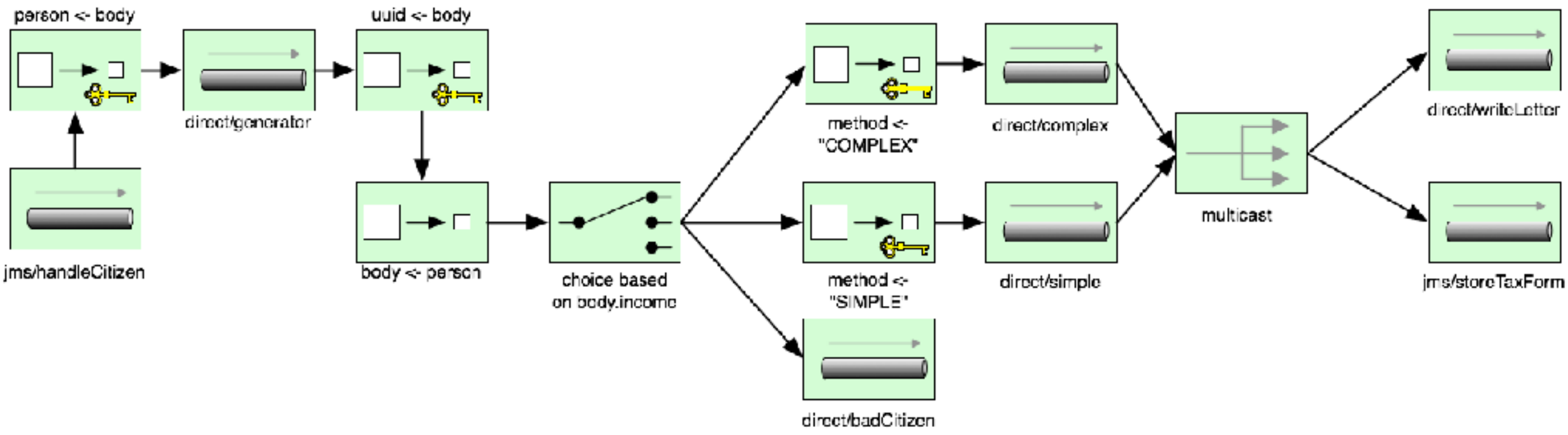
```
events
  lightOn L1ON
end
state active
  lightOn => waitingForDraw
end
state waitingForDraw end
```







Graphical DSL



e.g., enterprise integration patterns

```
// Route to handle a given Person
from(HANDLE_CITIZEN)
                .log("    Routing ${body.lastName} according to income ${body.income}")
                .log("      Storing the Person as an exchange property")
                .setProperty("person", body())
                .log("      Calling an existing generator")
                .to("direct:generator")
                .setProperty("p_uuid", body())
                .setBody(simple("${property.person}"))
                .choice()
                        .when(simple("${body.income} >= 42000"))
                                .setProperty("tax_computation_method", constant("COMPLEX"))
                                .to("direct:complexTaxMethod")
                        .when(simple("${body.income} >= 0 && ${body.income} < 42000"))
                                .setProperty("tax_computation_method", constant("SIMPLE"))
                                .to("direct:simpleTaxMethod")
                        .otherwise()
                                .to("direct:badCitizen").stop() // stopping the route for bad citizens
                .end() // End of the content-based-router
                .setHeader("person_uid", simple("${property.person.uid}"))
                .multicast()
                        .parallelProcessing()
                        .to("direct:generateLetter")
                        .to(STORE_TAX_FORM)
;
```

Embedded DSL

```
events
  doorClosed     D1CL
  drawerOpened   D2OP
  lightOn        L1ON
  doorOpened     D1OP
  panelClosed    PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end
```

```
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

```
events
  doorClosed    D1CL
end
```

```
commands
  lockPanel   PNLK
  unlockDoor  D1UL
end
```

```
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end
```

```
 state active
   ...
 end
```

```
events
  doorClosed    D1CL
end
```

```
event :doorClosed, "D1CL"
```

```
commands
  lockPanel   PNLK
  unlockDoor  D1UL
end
```

```
command :lockPanel, "PNLK"
command :unlockDoor, "D1UL"
```

```
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end
```

```
state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end
```

```
 state active
   ...
 end
```

```
state :active do
  ...
end
```

```
event :doorClosed, "D1CL"
```

```
command :lockPanel, "PNLK"
command :unlockDoor, "D1UL"
```

```
state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end
```

```
state :active do
  ...
end
```

```haskell
main :: IO ()
main = either print print . fmap generate $ buildApp "example" $ do
  addSensor button $ onPin 9
  addActuator light $ onPin 12
  defineStates [offline, online]
  actionsWhen offline `execute` [ set light `to` off ]
  actionsWhen online  `execute` [ set light `to` on ]
  transitionsFrom online  `are` [ when button `is` pressed $ goto offline ]
  transitionsFrom offline `are` [ when button `is` pressed $ goto online ]
  start offline
```