EECS 3311
Software Design
Specifications
Good Abstractions

SET[G]
FUN[G, H]
REL[G,H]

Water Park Operator Charged After Boy Decapitated on Slide

(KANSAS CITY, Mo.) — A Kansas waterslide hyped as the world's highest was a "deadly weapon" that had already injured more than a dozen people before a 10-year-old boy was decapitated on it in 2016, according to a grand jury indictment unsealed Friday that charges the water park operator and an executive with involuntary manslaughter.

Time Magazine March 23, 2018

- "was never properly or fully designed"
- "rushed it into use and had no technical or engineering expertise"
- "complied with "few, if any" longstanding safety standards"
- "the . . . death and the rapidly growing list of injuries were foreseeable and expected outcomes"
- "desire to "rush the project" and . . . designer's lack of expertise caused them to "skip fundamental steps in the design process.""
- "not a single engineer was directly involved in . . . engineering or . . . design"

The Guardian
International edition

March 26, 2018

Uber crash shows 'catastrophic failure' of self-driving technology, experts say

Concerns raised about future testing as footage suggests fatal collision in Arizona was failing of system's most basic functions

Sam Levin • Last modified on Mon 26 Mar 2018 09.10 BST

Video released of fatal Uber self-driving crash

Video of the first self-driving car crash that killed a pedestrian suggests a "catastrophic failure" by Uber's technology, according to experts in the field, who said the footage showed the autonomous system erring on one of its most basic functions.

Article on Specifications, Mathmodels & ETF

- Carefully study SVN Readings: *Mathemodels-ETF.pdf*
- **Exercise**: Do the Ehealth Mathmodel Spec yourself.

Eiffel DbC

EHEALTH

patients: SET [PATIENT]
medications: SET [MEDICATION]
prescriptions: REL [PATIENT, MEDICATION]
interactions: SET [INTERACTION] -- dangerous interactions

invariant
∀m1,m2 ∈ medications ∀p ∈ PATIENT:
[m1,m2] ∈ interactions ⇒ ¬([p,m1] ∈ prescriptions ∧ [p,m2] ∈ prescriptions)

ADD_PRESCRIPTION

add_prescription(p: PATIENT; m: MEDICATION)
require [p,m] ∉ prescriptions
∀x ∈ medications: [p,x] ∈ prescriptions ⇒ [x,m] ∉ interactions
ensure prescriptions = **old** prescriptions ∪ {[p,m]}

Precondition must preserve safety invariant

DOI:10.1145/2663342

Viewpoint

Teach Fou… Language …

Industry is ready and waiting for … educated in the principles of prog…

Thomas Ball (tball@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA.

Benjamin Zorn (zorn@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA.

Our recommendations are threefold, ... First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, ... "To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper." (Lesley Lamport)

The methods, tools, and materials for educating students about "formal specs" are ready for prime time. Mechanisms such as "design by contract," now available in mainstream programming languages, should be taught as part of introductory programming, as is done in the introductory programming language sequence at Carnegie Mellon University. … We are failing our computer science majors if we do not teach them about the value of formal specifications.

```
a: ARRAY [G] -- array to be sorted
p: INTEGER   -- pivot

lampsort (i, j: INTEGER)
    -- sort slice of array a[i..j] using pivot p
  require
    (a.lower <= i <= j <= a.upper) and (a.lower <= p <= j)
  local
    not_sorted: SET [INTEGER_INTERVAL]
    picked: INTEGER_INTERVAL
  do
    from
      create not_sorted.make_one (i |..| j)
    invariant
      permutation (a, old_a)
      -- partitions in intervals are disjoint
      -- a[1..n] is sorted iff all partitions in set intervals are sorted
    until
      not_sorted.is_empty -- stop when no more intervals in set
    loop
      not_sorted.choose_item
      picked := not_sorted.item
      not_sorted.remove_item    -- remove interval just picked
      if picked.count > 1 then  -- pre-condition of partition holds
        partition (picked.lower, picked.upper)
        -- insert new [non-empty] intervals into set
        if picked.lower < p then
          not_sorted.extend (picked.lower |..| (p - 1))
        end
        if picked.upper > p then
          not_sorted.extend ((p + 1) |..| picked.upper)
        end
      end
    variant
      sum_of_interval_counts (not_sorted) -- total number of unsorted indices
    end
ensure
    sorted (i, j)
    permutation (old a.deep_twin, a)
end
```

Lamport's Quicksort Algorithm

command-query separation principle

SET[G]
make_empty
choose_item
item: G
remove_item
extend(g:G)
union …

Algorithms vs. Programs

Lamport's Quicksort is specified as an abstract divide and conquer **algorithm** using

- *intervals*: SET[INTEGER_INTERVAL]
- non-deterministic choice of an arbitrary element in the set
- that can be refined to a recursive, iterative or concurrent implementation

SET[G] and loop variants/invariants allow for the specification of the algorithm at a high level rather than an implementation in a programming language with unnecessary and confusing implementation detail.

Finding the right abstractions such as SET[INTERVAL]
to describe the algorithm
is one of the most important parts of software design

Mathematical Model Classes

- PAIR[G, H]
- SET[G]
- SEQ[G]
- BAG[G]
- FUN[G, H]
- REL[G, H]
- ITERABLE_ARITMETIC

- Finding the right abstractions not just for algorithms, but also for system design
- Class invariants capture essential properties

- Balances must be non-negative
- Student time-tables must not conflict etc.
- Patients must not be prescribed dangerous interactions

Example: Course Registrations

- Students take courses
- Students shall not have conflicting course registrations
- Model students taking courses as a relation

registrations: REL[STUDENT, COURSE]

use a class invariant to ensure
that there are no conflicts

Students and Courses

Expression	Value	Type
c1.time	MON, 16:00 -- 17:30	STRING_8
c1.name	EECS3311	STRING_8

```
class STUDENT feature

  name: STRING

end
```

```
class COURSE feature

  name: STRING

  time: STRING

feature {COURSE, REGISTRATION}
  day: 1..7
  start, finish: TIME
  duration: TIME_DURATION

invariant
  finish > start
end
```

Students and Courses

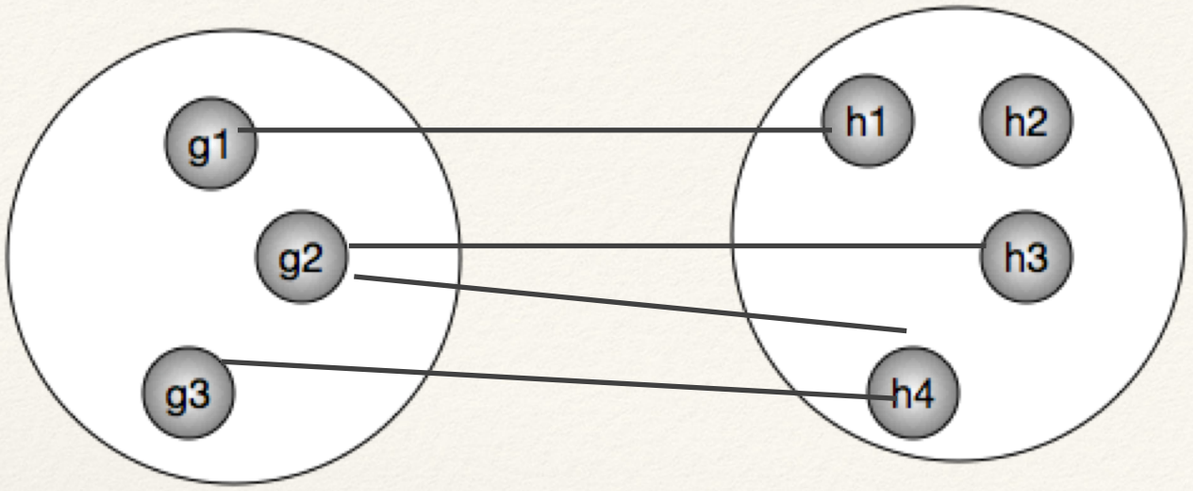
```
create c1.make ("EECS3311", 1, "16:00", "17:30")
create c2.make ("EECS3342", 2, "11:30", "13:00")
create c3.make ("EECS3101", 5, "19:00", "22:00")
create c4.make ("EECS2011", 5, "13:00", "14:30")
```

```
create s1.make ("s1")
create s2.make ("s2")
create s3.make ("s3")
```

Expression	Value	Type
⊞ c1.time	MON, 16:00 -- 17:30	STRING_8
⊞ c1.name	EECS3311	STRING_8

```
class STUDENT feature

  name: STRING

end
```

```
class COURSE feature

  name: STRING

  time: STRING

feature {COURSE, REGISTRATION}
  day: 1..7
  start, finish: TIME
  duration: TIME_DURATION

invariant
  finish > start
end
```

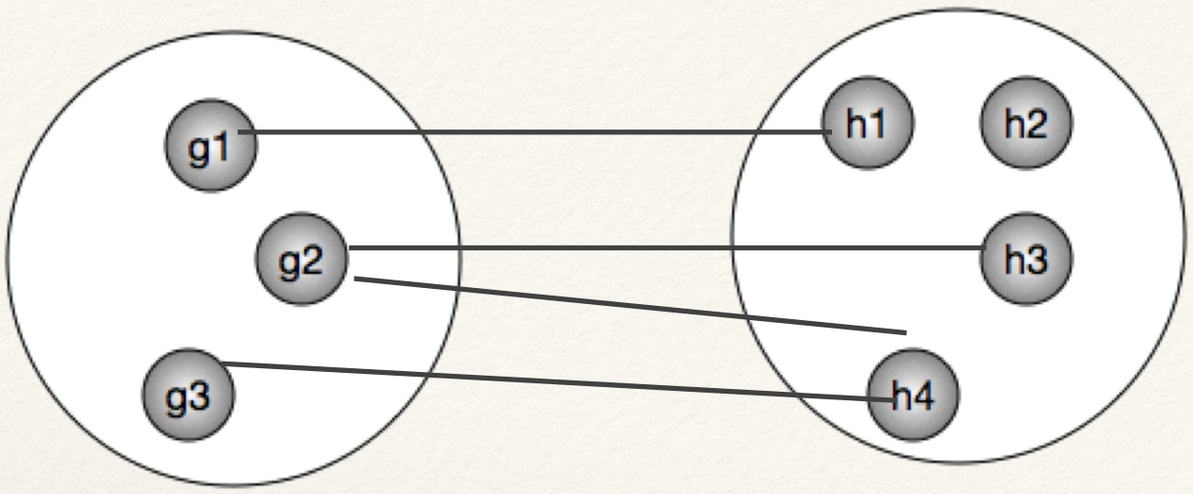


r: REL[G,H]

r = {[g1,h1], [g2,h3], [g2,h4], g3,h4]}

r.domain = {g1,g2,g3}

r.range = {h1, h3, h4}

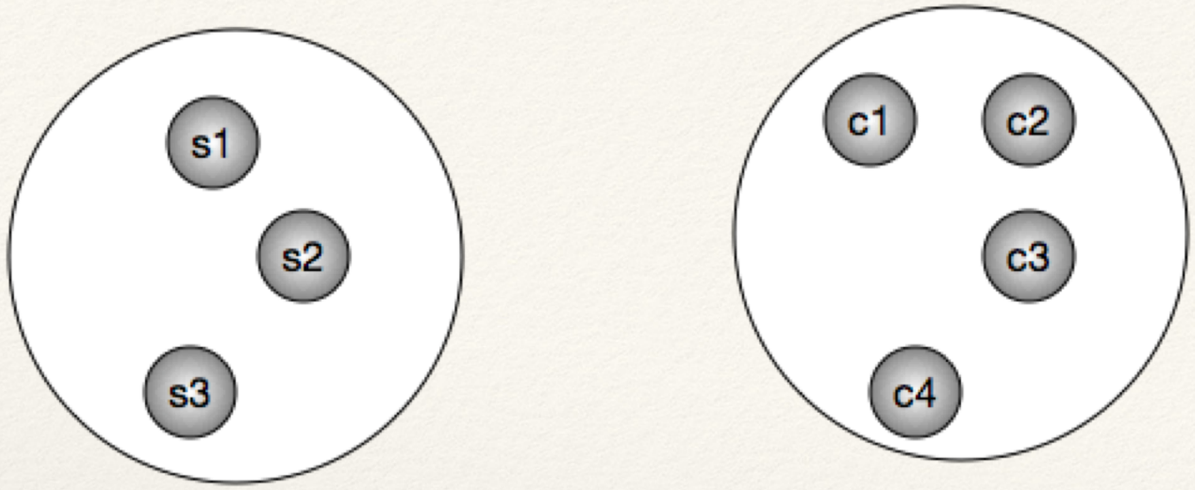


r: REL[G,H]
r = {[g1,h1], [g2,h3], [g2,h4], g3,h4]}
r.domain = {g1,g2,g3}
r.range = {h1, h3, h4}

```
class REL[G,H] inherit SET[PAIR[G,H]] feature

  domain: SET[G]
  range:    SET[H]

  image(g: G): SET[H]

  extend(p: PAIR[G,H]) -- from SET

  extended alias "+" (p: PAIR[G,H]): like Current

  override (r: like Current)
    -- update current relation with r

  overriden_by alias "@<+" (t: TUPLE [g: G; h: H]): like Current

  ...

end
```

```
class PAIR [G, H] feature
  first: G
  second: H
end
```

registrations: REL[STUDENT,COURSE]



class REGISTRATION **feature**

registrations: REL[STUDENT, COURSE]

conflict(c1,c2: COURSE): BOOLEAN

no_conflicts (regs: REL[STUDENT, COURSE]): BOOLEAN

extend_by_array (a: ARRAY[TUPLE[s: STUDENT; c: COURSE]])

invariant

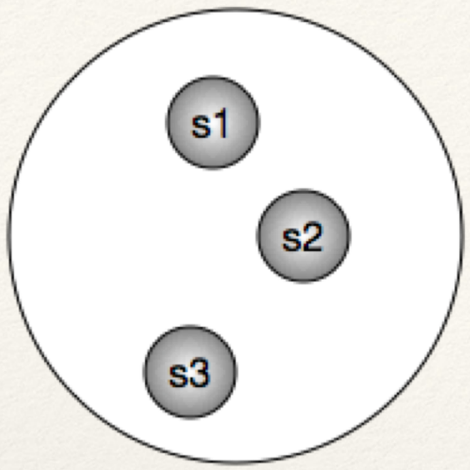
∀ (s1, c1), (s2, c2) ∈ registrations :

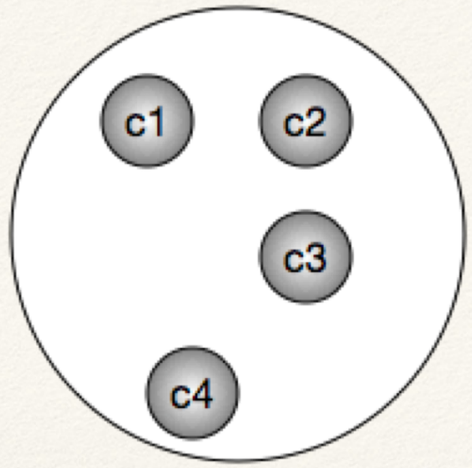
c1 ≠ c2 ∧ s1 = s2 ⇒ ¬(conflict (c1, c2))

end

```
conflict (c1, c2: COURSE): BOOLEAN
        -- Do courses 'c1' and 'c2' conflict with each other?
    ensure
        Result =
            (     (c1.day = c2.day)
             and (    (c1.start ~ c2.start
                   OR (      c1.start < c2.start
                         and c1.start + c1.duration >= c2.start)
                   OR (      c2.start < c1.start
                         and c2.start + c2.duration >= c1.start)) ))
    end
```

registrations: REL[STUDENT,COURSE]

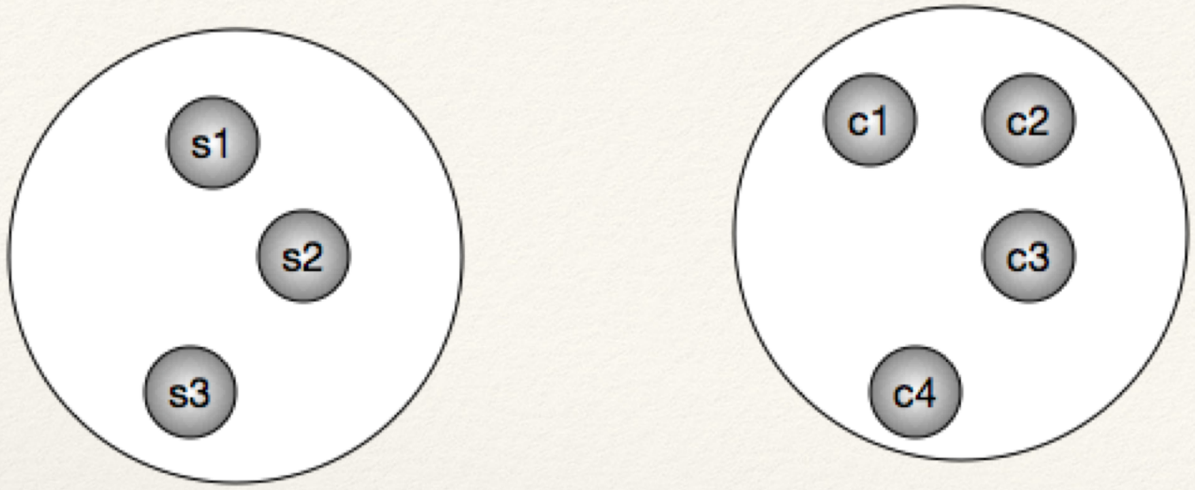





```
no_conflicts (regs: REL[STUDENT, COURSE]): BOOLEAN
    ensure
        -- ∀ (s1, c1), (s2, c2) ∈ regs :
        --       c1 ≠ c2 ∧ s1 = s2 ⇒ ¬(conflict (c1, c2))
```

```
class REGISTRATION feature

  registrations: REL[STUDENT, COURSE]

  conflict(c1,c2: COURSE): BOOLEAN

  no_conflicts (regs: REL[STUDENT, COURSE]): BOOLEAN

  extend_by_array (a: ARRAY[TUPLE[s: STUDENT; c: COURSE]])

invariant
  ∀ (s1, c1), (s2, c2) ∈ registrations :
      c1 ≠ c2 ∧ s1 = s2 ⇒ ¬(conflict (c1, c2))
end
```

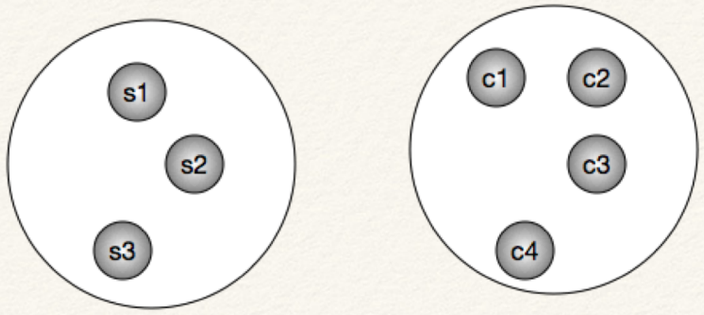
registrations: REL[STUDENT,COURSE]




```
class REGISTRATION feature

  registrations: REL[STUDENT, COURSE]

  conflict(c1,c2: COURSE): BOOLEAN

  no_conflicts (regs: REL[STUDENT, COURSE]): BOOLEAN

  extend_by_array (a: ARRAY[TUPLE[s: STUDENT; c: COURSE]])

invariant
  ∀ (s1, c1), (s2, c2) ∈ registrations :
      c1 ≠ c2 ∧ s1 = s2 ⇒ ¬(conflict (c1, c2))
end
```

```
no_conflicts (regs: REL[STUDENT, COURSE]): BOOLEAN
    ensure
        Result =
            across regs as r1 all
            across regs as r2 all
                      r1.item /~ r2.item
                  and r1.item.first ~ r2.item.first
                IMPLIES
                    NOT (conflict (r1.item.second, r2.item.second))
            end end
end
```

Invariants are precious
No student shall have a conflict in their courses



registrations: REL[STUDENT, COURSE]

```
t1: BOOLEAN
    local
        c1, c2, c3, c4, c5: COURSE
        s1, s2, s3: STUDENT
        a: ARRAY[TUPLE[s:STUDENT; c:COURSE]]
        r : REGISTRATION
        dom: SET[STUDENT]
        ran: SET[COURSE]
        r_conflicted: REL[STUDENT, COURSE]
    do
        comment ("t1: basic features of registrations")

        create c1.make ("EECS3311", 1, "16:00", "17:30")
        create c2.make ("EECS3342", 2, "11:30", "13:00")
        create c3.make ("EECS3101", 5, "19:00", "22:00")
        create c4.make ("EECS2011", 5, "13:00", "14:30")

        create s1.make ("s1")
        create s2.make ("s2")
        create s3.make ("s3")

        a := << [s1, c1],
                [s2, c2],
                [s3, c3],
                [s3, c4]
             >>

        create r
        r.extend_by_array (a)
```

Set-up students and courses

Result := r.registrations.out ~
{
s1 -> EECS3311: MON, 16:00 -- 17:30,
s2 -> EECS3342: TUE, 11:30 -- 13:00,
s3 -> EECS3101: FRI, 19:00 -- 22:00,
s3 -> EECS2011: FRI, 13:00 -- 14:30
}

✓

r	<0x1126497C0>	REGISTRATION
registrations	{ s1 -> EECS3311:MON, 16:00 -- 17:30, s2 -> EECS334...	REL [!STUDENT, !COURSE]

Testing the Model

```
pair (s: STUDENT; c: COURSE): PAIR[STUDENT, COURSE]
    do
        create Result.make (s, c)
    end
```

```
create c1.make ("EECS3311", 1, "16:00", "17:30")
create c2.make ("EECS3342", 2, "11:30", "13:00")
create c3.make ("EECS3101", 5, "19:00", "22:00")
create c4.make ("EECS2011", 5, "13:00", "14:30")
```

```
a := << [s1, c1],
        [s2, c2],
        [s3, c3],
        [s3, c4]
     >>

create r
r.extend_by_array (a)
```

- check for course conflicts
- course c4 and c5 are in conflict
- thus student s3 cannot take both c4 and c5

```
create dom.make_from_array (<< s1, s2, s3 >>)
create ran.make_from_array (<< c1, c2, c3, c4 >>)
Result := r.registrations.domain ~ dom and r.registrations.range ~ ran
check
        Result
end
create c5.make ("EECS2031", 5, "14:00", "15:30")
Result := r.conflict (c4, c5)
check
        Result
end
r_conflicted := r.registrations.deep_twin
r_conflicted.extend (pair (s3, c5))
Result := not r.no_conflicts (r_conflicted)
```

DOI:10.1145/2663342

Viewpoint

Teach Found
Language Pr

Industry is ready and waiting for more g
educated in the principles of programm

Thomas Ball (tball@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA.

Benjamin Zorn (zorn@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA.

Our recommendations are threefold, ... First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, ... "To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper." (Lesley Lamport)

The methods, tools, and materials for educating students about "formal specs" are ready for prime time. Mechanisms such as "design by contract," now available in mainstream programming languages, should be taught as part of introductory programming, as is done in the introductory programming language sequence at Carnegie Mellon University. … We are failing our computer science majors if we do not teach them about the value of formal specifications.

```
feature -- Attributes
    registrations: REL [STUDENT, COURSE]
```

```
extend (s: STUDENT; c: COURSE)
        -- Register student 's' for course 'c'.
    require
        not_registered_yet: -- [c, r] ∉ registrations
            not registrations.has ([s,c])
        no_conflicting_registrations:
            no_conflicts (registrations + [s, c])
    do
        registrations.extend (pair(s, c))
    ensure
        registrations ~ old registrations.deep_twin + [s, c]
        -- registrations = old registrations + [s, c]
    end
```

```
invariant

-- ∀ (s1, c1), (s2, c2) ∈ registrations :
--                    c1 ≠ c2 ∧ s1 = s2 ⇒ ¬(conflict (c1, c2))
    no_conflicting_registrations:
        no_conflicts (registrations)
```

Guaranteeing the class invariant

- REL[STUDENT, COURSE] is the right abstraction to model student registrations
- A class invariant ensures that student are not taking conflicting courses
- command preconditions are required to ensure the class invariant

```
extend (s: STUDENT; c: COURSE)
        -- Register student 's' for course 'c'.
    require
        not_registered_yet: -- [c, r] ∉ registrations
            not registrations.has ([s,c])
        no_conflicting_registrations:
            no_conflicts (registrations + [s, c])
    do
        registrations.extend (pair(s, c))
    ensure
        registrations ~ old registrations.deep_twin + [s, c]
        -- registrations = old registrations + [s, c]
    end
```

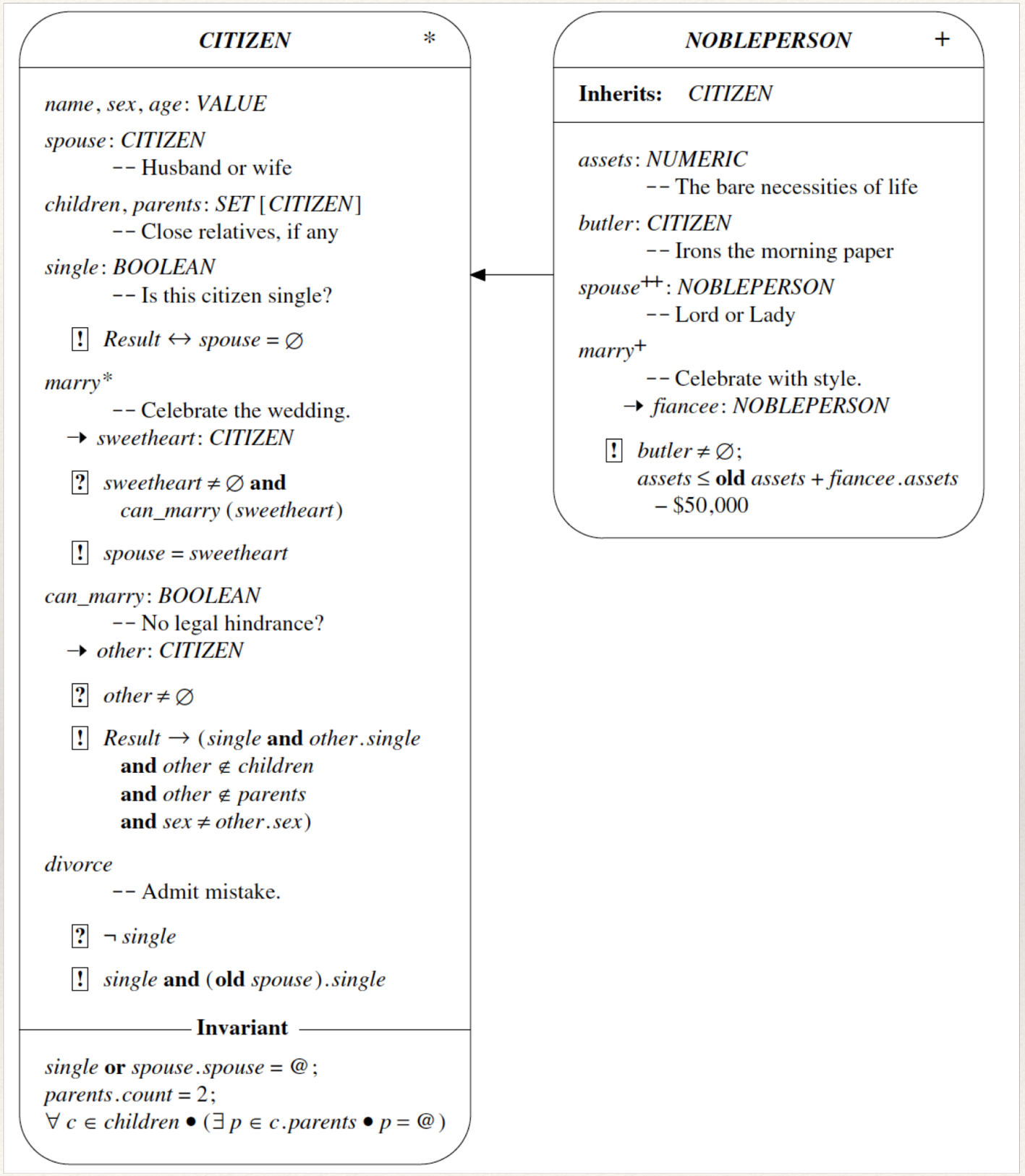
Finding the right design abstractions

- REL[STUDENT, COURSE] is the right abstraction to model student registrations free of implementation detail
- **A class invariant ensures that student are not taking conflicting courses**
- Command routines must have preconditions that guarantee the class invariant
- The **model specification** is executable but may have to be refined to a better **implementation** for efficiency
  - Queries of REL may be used to check correctness of the implementation

Another Example

- Marriage Registry
- Challenge: specify marry

```
PERSON
---------------------------------------
name: NAME
spouse: detachable PERSON

marry(p:PERSON)
   -- contracts  to satisfy
   -- marriage regulations?
```



CITIZEN *

name, *sex*, *age*: *VALUE*

spouse: *CITIZEN*
-- Husband or wife

children, *parents*: *SET* [*CITIZEN*]
-- Close relatives, if any

single: *BOOLEAN*
-- Is this citizen single?

! *Result* ↔ *spouse* = ∅

*marry**
-- Celebrate the wedding.
→ *sweetheart*: *CITIZEN*

? *sweetheart* ≠ ∅ **and**
can_marry (*sweetheart*)

! *spouse* = *sweetheart*

can_marry: *BOOLEAN*
-- No legal hindrance?
→ *other*: *CITIZEN*

? *other* ≠ ∅

! *Result* → (*single* **and** *other*.*single*
and *other* ∉ *children*
and *other* ∉ *parents*
and *sex* ≠ *other*.*sex*)

divorce
-- Admit mistake.

? ¬ *single*

! *single* **and** (**old** *spouse*).*single*

Invariant

single **or** *spouse*.*spouse* = @;
parents.*count* = 2;
∀ *c* ∈ *children* • (∃ *p* ∈ *c*.*parents* • *p* = @)

NOBLEPERSON +

Inherits: *CITIZEN*

assets: *NUMERIC*
-- The bare necessities of life

butler: *CITIZEN*
-- Irons the morning paper

spouse^{++}: *NOBLEPERSON*
-- Lord or Lady

marry^{+}
-- Celebrate with style.
→ *fiancee*: *NOBLEPERSON*

! *butler* ≠ ∅;
assets ≤ **old** *assets* + *fiancee*.*assets*
– $50,000

But we need a registry of citizens, not just a citizen!

Challenge: specify marry

REGISTRY

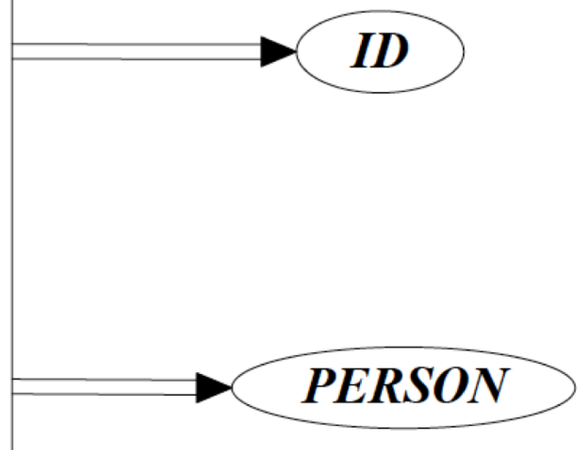
person:FUN[ID, PERSON]
spouse:FUN[ID, ID]

spouse_of(id:ID): **detachable** ID
  ensure:
    id ∈ spouse.domain ⇒ Result = spouse[id]
    id ∉ spouse.domain ⇒ Result = void

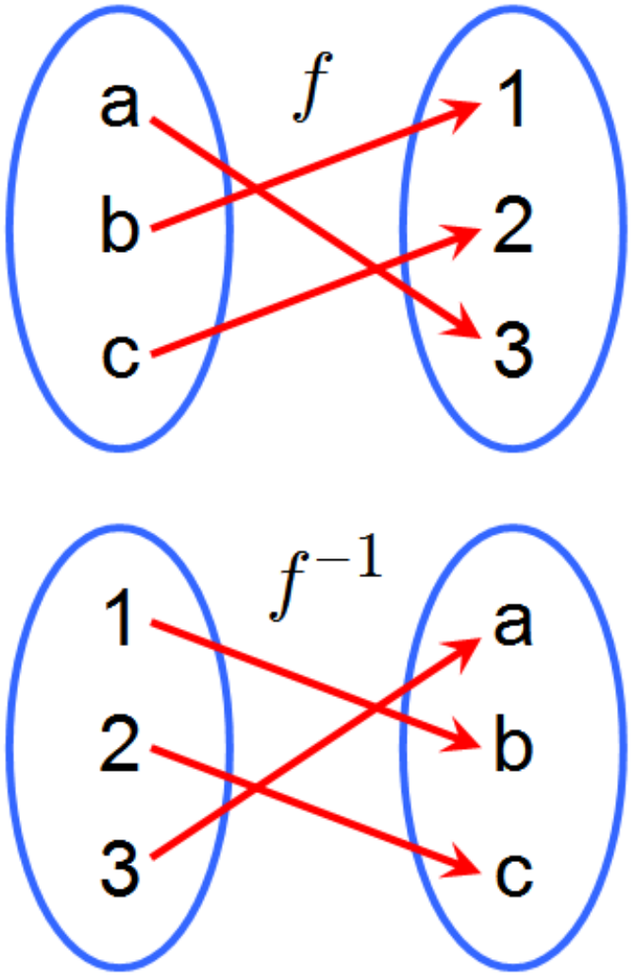
marry(id1,id2:ID)
  require id1 ≠ id2
      id1 ∉ spouse.domain ∧ id2 ∉ spouse.domain
  ensure person ~ **old** person
      spouse ~ **old** spouse ∪ {id1 ↦ id2, id2 ↦ id1}

Invariant

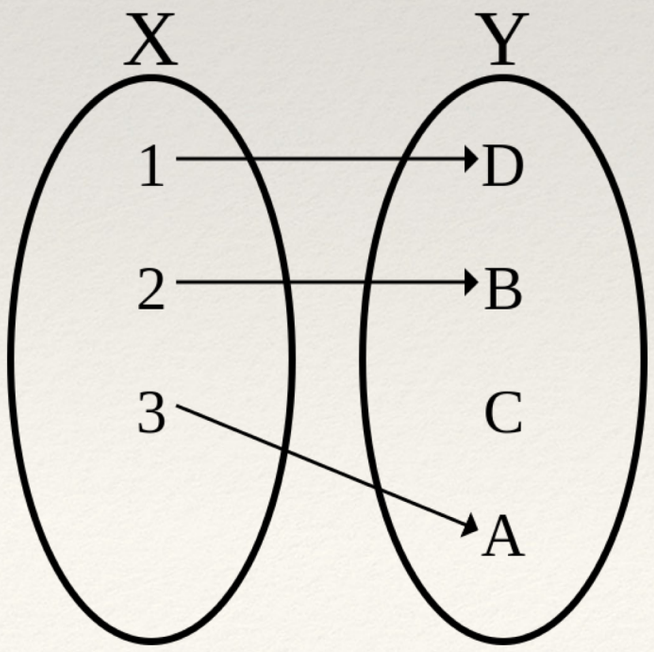
spouse.domain ⊆ person.domain
spouse.is_injection
spouse.has_inverse



- Remove spouse from person
- Marriage does not change person
- Use FUN from mathmodels



$$\{y \mapsto x \mid x \mapsto y \in r\}.$$



REGISTRY

person:FUN[ID, PERSON]
spouse:FUN[ID, ID]

spouse_of(id:ID): **detachable** ID
ensure:
id ∈ spouse.domain ⇒ Result = spouse[id]
id ∉ spouse.domain ⇒ Result = void

marry(id1,id2:ID)
require id1 ≠ id2
id1 ∉ spouse.domain ∧ id2 ∉ spouse.domain
ensure person ~ **old** person
spouse ~ **old** spouse ∪ {id1 ↦ id2, id2 ↦ id1}

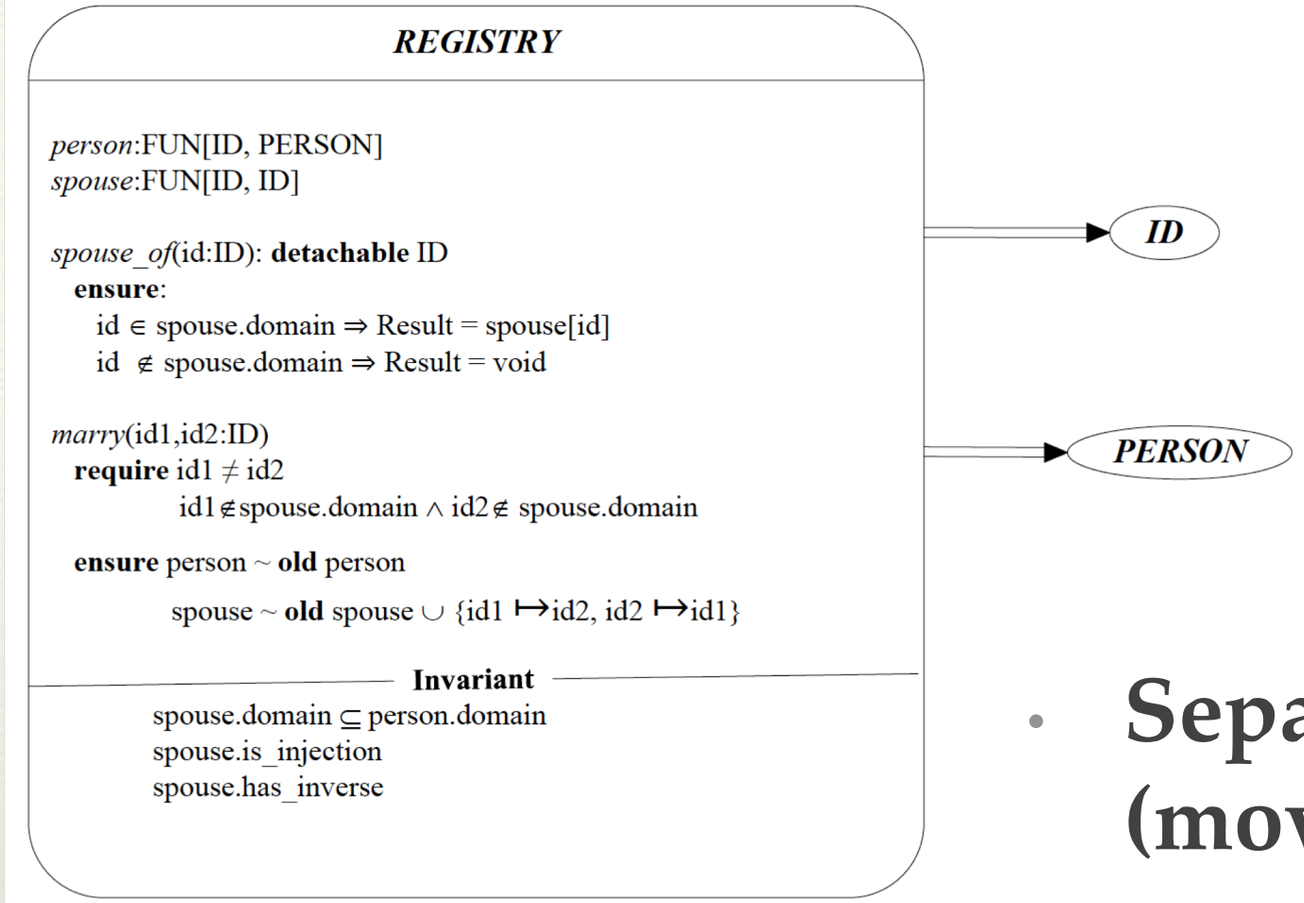
Invariant
spouse.domain ⊆ person.domain
spouse.is_injection
spouse.has_inverse

ID

PERSON

The function is **injective** (**one-to-one**) if every element of the codomain is mapped to by *at most* one element of the domain. An injective function is an **injection**. Notationally:

$$\forall x, x' \in X, f(x) = f(x') \Rightarrow x = x'.$$



- **Separation of concerns (move spouse out of person)**
- Information hiding
spouse function can always be moved elsewhere if it needs to be optimized, and spouse_of is the interface)

Building Maintainable Software (Java Edition)

Write short units of code (Chapter 2)

Shorter units (that is, methods and constructors) are easier to analyze, test, and reuse.

Write simple units of code (Chapter 3)

Units with fewer decision points are easier to analyze and test.

Write code once (Chapter 4)

Duplication of source code should be avoided at all times, since changes will need to be made in each copy. Duplication is also a source of regression bugs.

Keep unit interfaces small (Chapter 5)

Units (methods and constructors) with fewer parameters are easier to test and reuse.

Separate concerns in modules (Chapter 6)

Modules (classes) that are loosely coupled are easier to modify and lead to a more modular system.

Separate concerns in modules (Chapter 6)

Modules (classes) that are loosely coupled are easier to modify and lead to a more modular system.

Couple architecture components loosely (Chapter 7)

Top-level components of a system that are more loosely coupled are easier to modify and lead to a more modular system.

Keep architecture components balanced (Chapter 8)

A well-balanced architecture, with not too many and not too few components, of uniform size, is the most modular and enables easy modification through separation of concerns.

Keep your codebase small (Chapter 9)

A large system is difficult to maintain, because more code needs to be analyzed, changed, and tested. Also, maintenance productivity per line of code is lower in a large system than in a small system.

Automate development pipeline and tests (Chapter 10)

Automated tests (that is, tests that can be executed without manual intervention) enable near-instantaneous feedback on the effectiveness of modifications. Manual tests do not scale.

Write clean code (Chapter 11)

Having irrelevant artifacts such as TODOs and dead code in your codebase makes it more difficult for new team members to become productive. Therefore, it makes maintenance less efficient

Table P-1. A generic grouping of concepts and their representation in Java.

Generic name	Generic definition	In Java
Unit	Smallest grouping of lines that can be executed independently	Method or constructor
Module	Smallest grouping of units	Top-level class, interface, or enum
Component	Top-level division of a system as defined by its software architecture	(Not defined by the language)
System	The entire codebase under study	(Not defined by the language)

For the Exam, with SET, SEQ, FUN, REL

```
class SET [G] create
  make_empty
feature -- Iteration
  new_cursor: ITERATION_CURSOR [G]
    -- Fresh cursor associated with current structure
feature -- Commands with efficient implementation
  extend (g: G)
    -- Extend the current set by `g'.
  union (other: like Current)
    -- Union of the current set and `other'.
  subtract (g: G)
   -- Subtract the current set by `g'.
  difference (other: like Current)
    -- Difference of the current set and `other'.
feature -- Queries
  count alias "#": INTEGER
    -- Return the cardinality of the set.
  is_empty: BOOLEAN
    -- Is the set empty?
  has (g: G): BOOLEAN
    -- Does the set contain `g'?
  extended alias "+" (g: G): like Current
    -- Return a new set representing the addition of `g' to
        Current
  unioned alias "|\/|" (other: like Current): like Current
    -- Return a new set representing the union of Current
        and `other'
  subtracted alias "-" (g: G): like Current
    -- Return a new set representing the subtraction of `g'
        from Current
  differenced alias "|\" (other: like Current): like Current
    -- Return a new set representing the difference between
        Current and `other'.
  ...
end
```

```
class PAIR [G, H] create
  make, make_from_tuple
convert
  make_from_tuple
    --allows the use of TUPLE[G,H]
    -- wherever PAIR[G, H] is expected
feature -- Constructor
  make (g: G; h: H)
    -- Intialize a new pair (g, h).
  make_from_tuple (t: TUPLE[g: G; h: H])
    -- Initialize a new pair [g, h]
feature -- Queries
  first: G
  second: H
end
```

```
class REL [G, H] inherit
  SET[PAIR[G, H]]
create
  make_empty
feature -- Queries
  domain: SET [G]
    -- Return the domain set of relation.
  range: SET [H]
    -- Return the range set of relation.
  image alias [] (g: G): SET [H]
    -- Retrieve set of range items
   -- for domain element g
   extended alias "+" (p:PAIR[G, H]):
         like Current
      -- return a new relation with addition of t
   overriden_by (p: PAIR[G, H]): like Current
    -- Return a new relation the same as Current,
    -- except p.first now maps to p.second
    -- alias "@<+"
end
```