Teaching the Mathematics of Software Design

Emil Sekerinski
McMaster University
August 2006

Experience in teaching

- Software Design 1, 2nd year course
- Software Design 2, 3rd year course

from 99/00 to 05/06

Difficulty of Teaching Software Design

- Rules of software design can be broken
- Design qualities difficult to judge
- Consequences of breaking design rules not experienced
- Poorly working software is "normal"
- Misconception that programming skills are sufficient

How to motivate teaching the mathematics of software design?

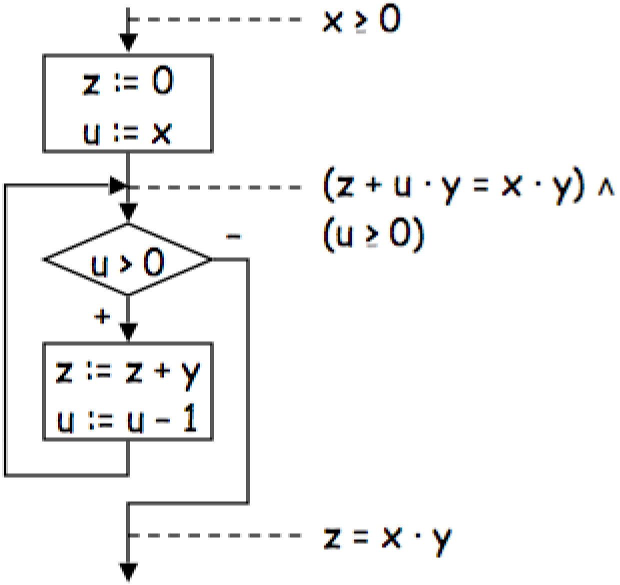
- Uniform design notation and uniform mathematical basis
  - Integrating mathematics, rather than contrasting formal vs informal
  - Uniform textual notation (in which diagrams are explained)
  - Typed logic and equational reasoning
- Middle-out sequencing of topics

Excerpt from 1. Elements of Programming

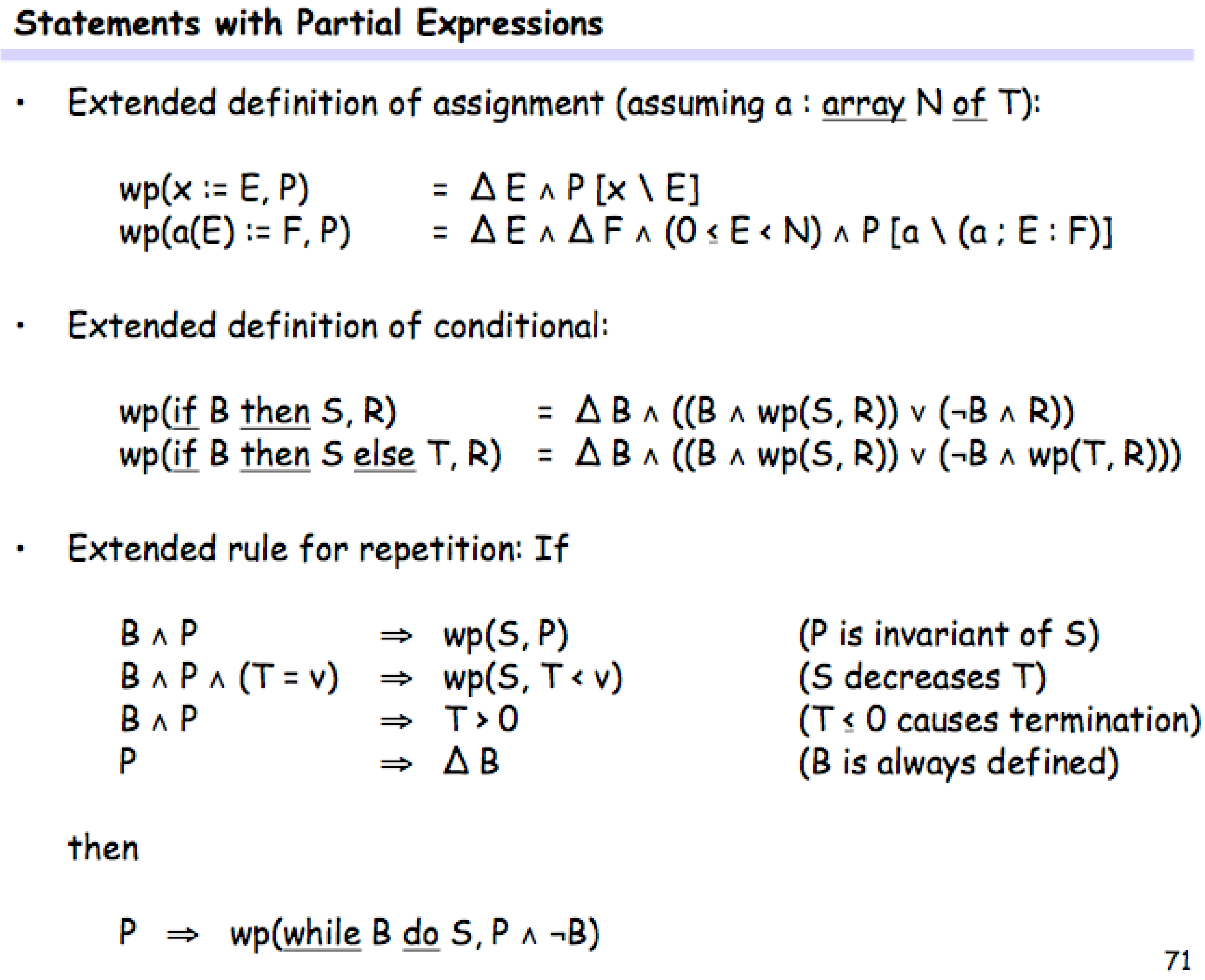
Program Annotations

- We can subdivide the task of checking correctness assertions by adding intermediate annotations:

```
{x ≥ 0}
z, u := 0, x ;
{invariant: (z + u · y = x · y) ∧ (u ≥ 0)}
while u > 0 do
    z, u := z + y, u – 1
{z = x · y}
```



Excerpt from 1. Elements of Programming



Statements with Partial Expressions

- Extended definition of assignment (assuming a : $\underline{\text{array}}$ N $\underline{\text{of}}$ T):

$$\begin{aligned} wp(x := E, P) &= \Delta E \land P[x \setminus E] \\ wp(a(E) := F, P) &= \Delta E \land \Delta F \land (0 \le E < N) \land P[a \setminus (a ; E : F)] \end{aligned}$$

- Extended definition of conditional:

$$\begin{aligned} wp(\underline{\text{if}}\ B\ \underline{\text{then}}\ S, R) &= \Delta B \land ((B \land wp(S, R)) \lor (\neg B \land R)) \\ wp(\underline{\text{if}}\ B\ \underline{\text{then}}\ S\ \underline{\text{else}}\ T, R) &= \Delta B \land ((B \land wp(S, R)) \lor (\neg B \land wp(T, R))) \end{aligned}$$

- Extended rule for repetition: If

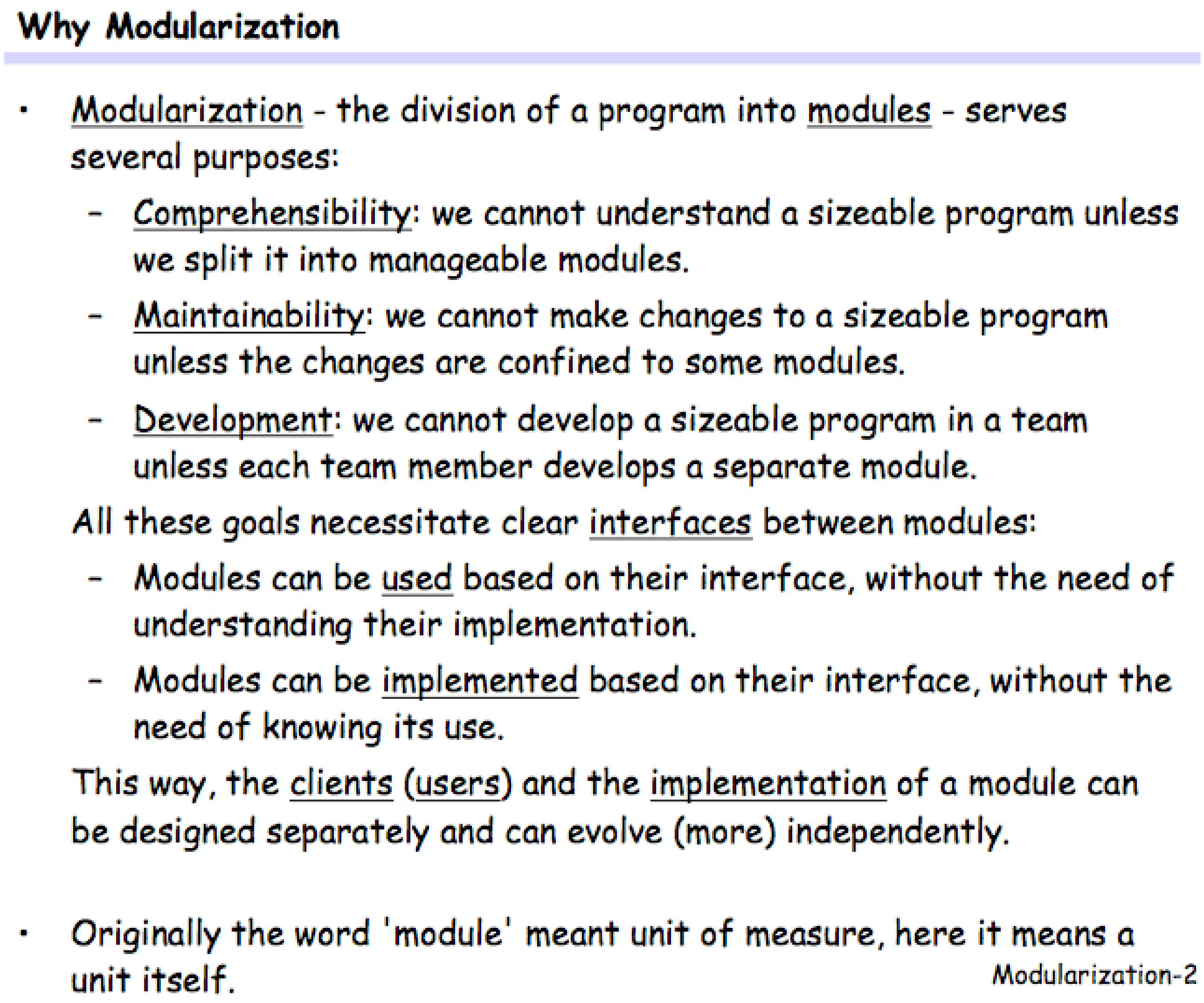
$$\begin{aligned} B \land P &\Rightarrow wp(S, P) && \text{(P is invariant of S)} \\ B \land P \land (T = v) &\Rightarrow wp(S, T < v) && \text{(S decreases T)} \\ B \land P &\Rightarrow T > 0 && \text{(T} \le \text{0 causes termination)} \\ P &\Rightarrow \Delta B && \text{(B is always defined)} \end{aligned}$$

then

$$P \Rightarrow wp(\underline{\text{while}}\ B\ \underline{\text{do}}\ S, P \land \neg B)$$

71

Excerpt from 2. Modularization



Why Modularization

- Modularization - the division of a program into modules - serves several purposes:
  - Comprehensibility: we cannot understand a sizeable program unless we split it into manageable modules.
  - Maintainability: we cannot make changes to a sizeable program unless the changes are confined to some modules.
  - Development: we cannot develop a sizeable program in a team unless each team member develops a separate module.

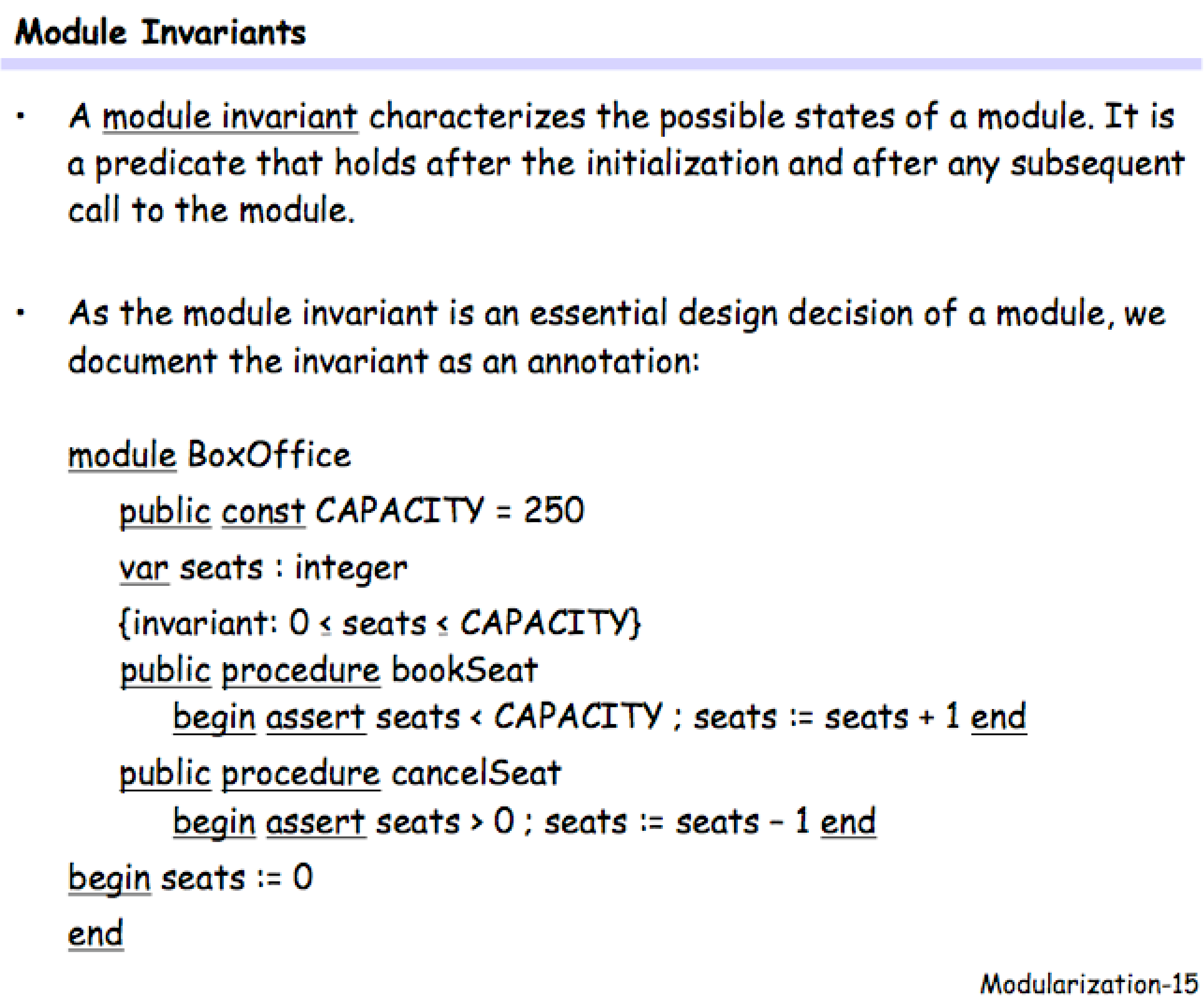
  All these goals necessitate clear interfaces between modules:
  - Modules can be used based on their interface, without the need of understanding their implementation.
  - Modules can be implemented based on their interface, without the need of knowing its use.

  This way, the clients (users) and the implementation of a module can be designed separately and can evolve (more) independently.

- Originally the word 'module' meant unit of measure, here it means a unit itself.

Modularization-2

Excerpt from 2. Modularization

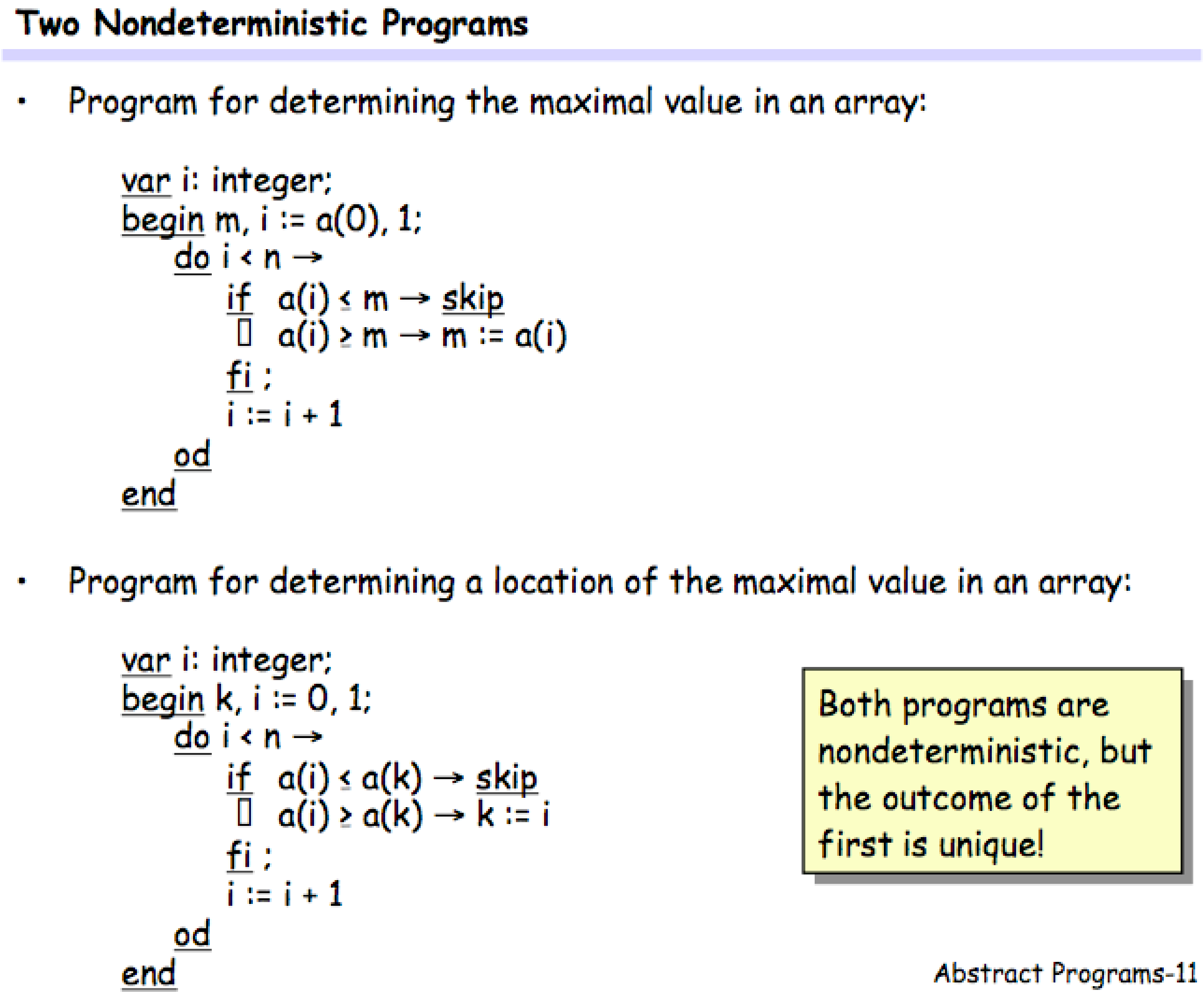


Module Invariants

- A module invariant characterizes the possible states of a module. It is a predicate that holds after the initialization and after any subsequent call to the module.
- As the module invariant is an essential design decision of a module, we document the invariant as an annotation:

```
module BoxOffice
    public const CAPACITY = 250
    var seats : integer
    {invariant: 0 ≤ seats ≤ CAPACITY}
    public procedure bookSeat
        begin assert seats < CAPACITY ; seats := seats + 1 end
    public procedure cancelSeat
        begin assert seats > 0 ; seats := seats – 1 end
begin seats := 0
end
```

Excerpt from 3. Abstract Programs



Two Nondeterministic Programs

- Program for determining the maximal value in an array:

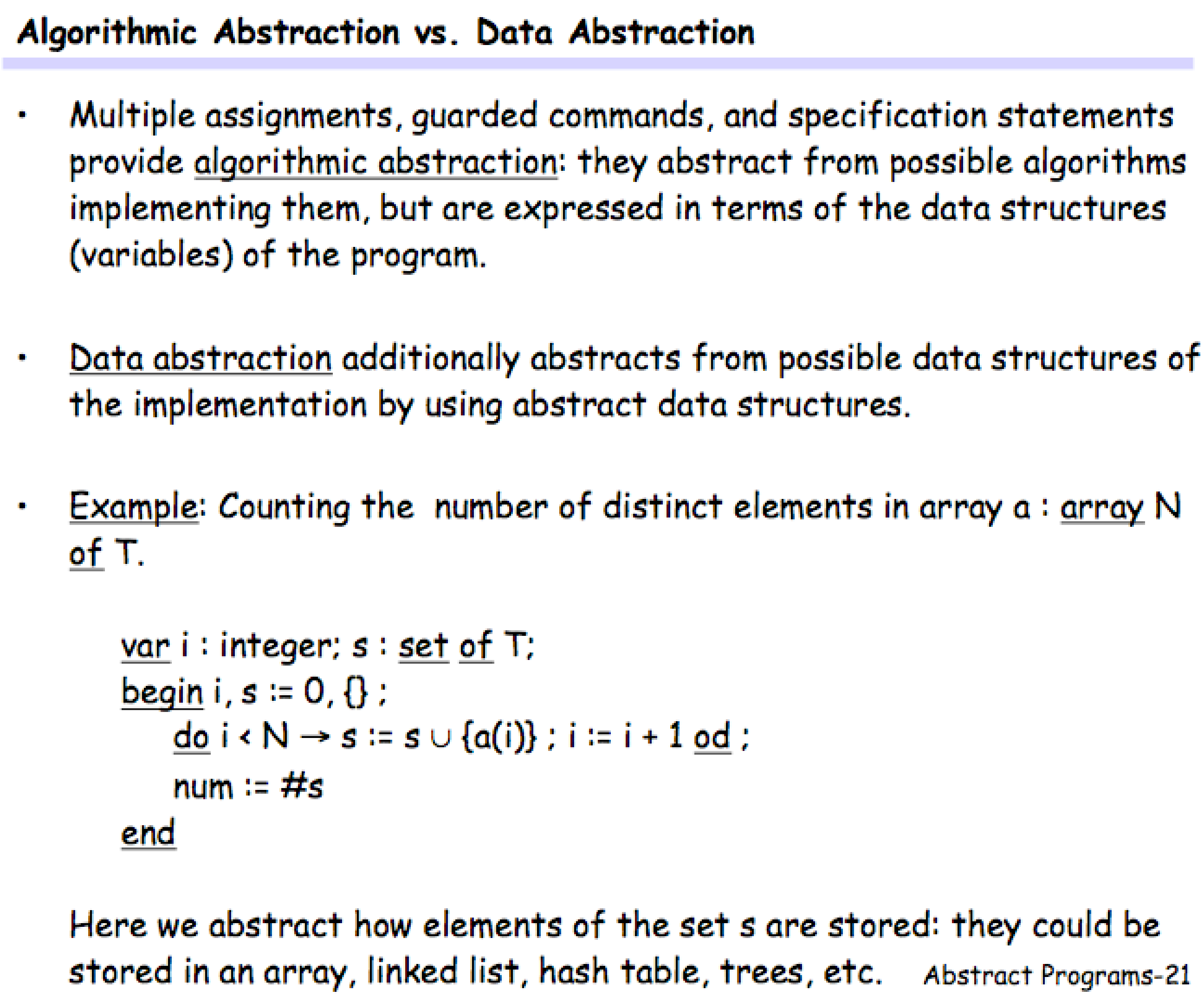
```
var i: integer;
begin m, i := a(0), 1;
    do i < n →
        if  a(i) ≤ m → skip
         ▯  a(i) ≥ m → m := a(i)
        fi ;
        i := i + 1
    od
end
```

- Program for determining a location of the maximal value in an array:

```
var i: integer;
begin k, i := 0, 1;
    do i < n →
        if  a(i) ≤ a(k) → skip
         ▯  a(i) ≥ a(k) → k := i
        fi ;
        i := i + 1
    od
end
```

Both programs are nondeterministic, but the outcome of the first is unique!

Excerpt from 3. Abstract Programs



Algorithmic Abstraction vs. Data Abstraction

- Multiple assignments, guarded commands, and specification statements provide algorithmic abstraction: they abstract from possible algorithms implementing them, but are expressed in terms of the data structures (variables) of the program.

- Data abstraction additionally abstracts from possible data structures of the implementation by using abstract data structures.

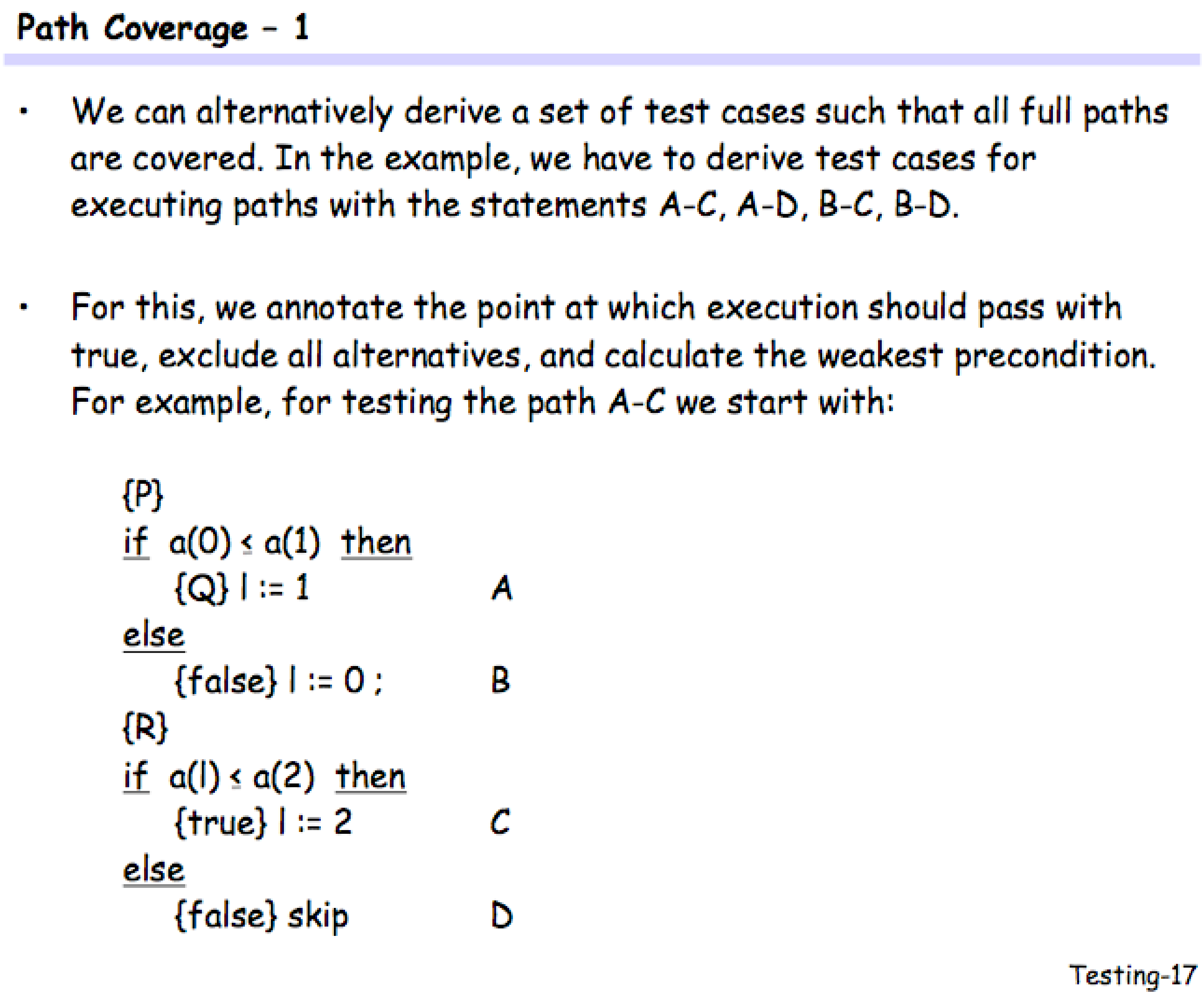
- Example: Counting the number of distinct elements in array a : array N of T.

```
var i : integer; s : set of T;
begin i, s := 0, {} ;
    do i < N → s := s ∪ {a(i)} ; i := i + 1 od ;
    num := #s
end
```

Here we abstract how elements of the set s are stored: they could be stored in an array, linked list, hash table, trees, etc.

Abstract Programs-21

Excerpt from 4. Testing



Path Coverage - 1

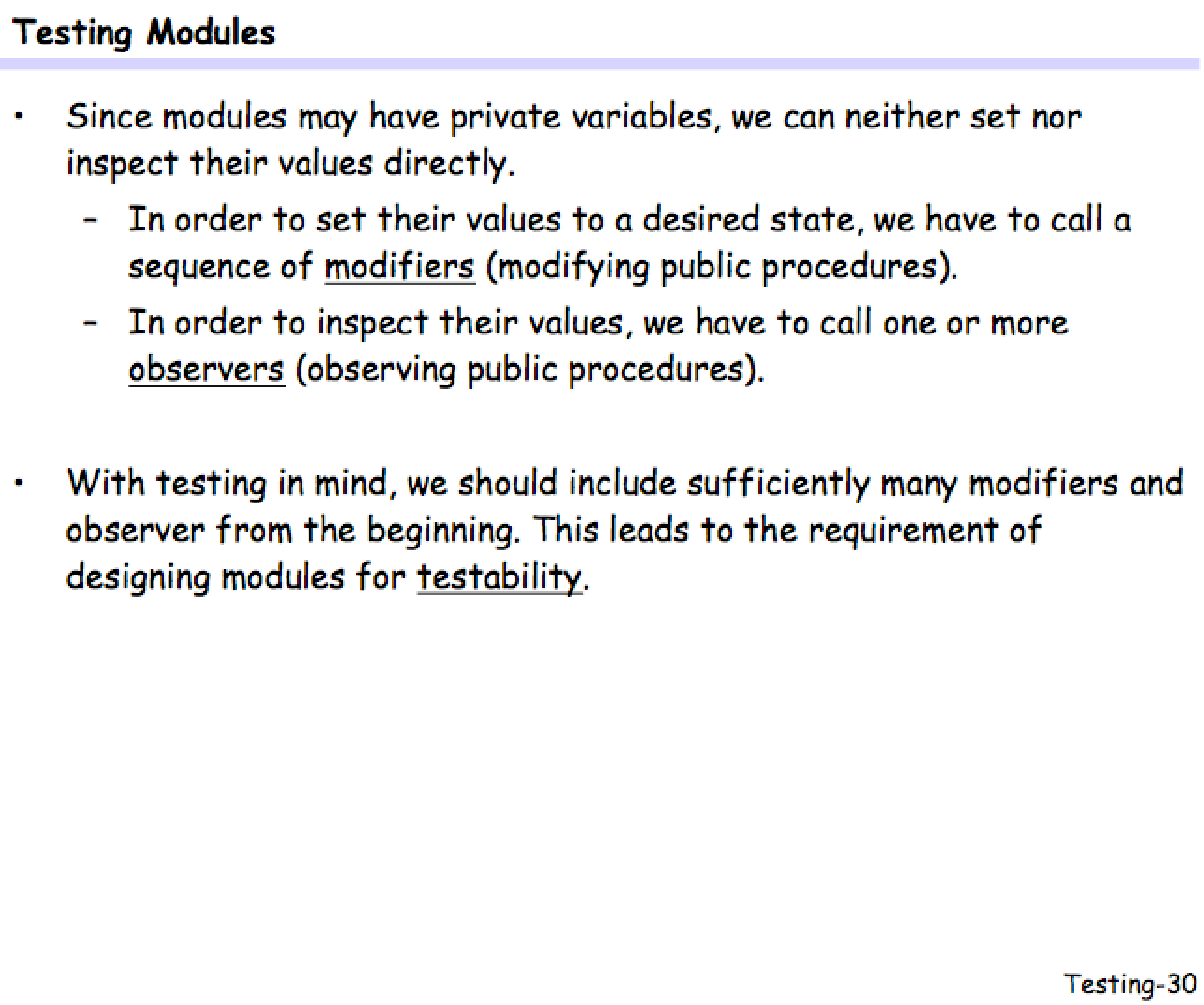
- We can alternatively derive a set of test cases such that all full paths are covered. In the example, we have to derive test cases for executing paths with the statements A-C, A-D, B-C, B-D.

- For this, we annotate the point at which execution should pass with true, exclude all alternatives, and calculate the weakest precondition. For example, for testing the path A-C we start with:

```
{P}
if  a(0) ≤ a(1)  then
    {Q} l := 1              A
else
    {false} l := 0 ;        B
{R}
if  a(l) ≤ a(2)  then
    {true} l := 2           C
else
    {false} skip            D
```

Testing-17

Excerpt from 4. Testing



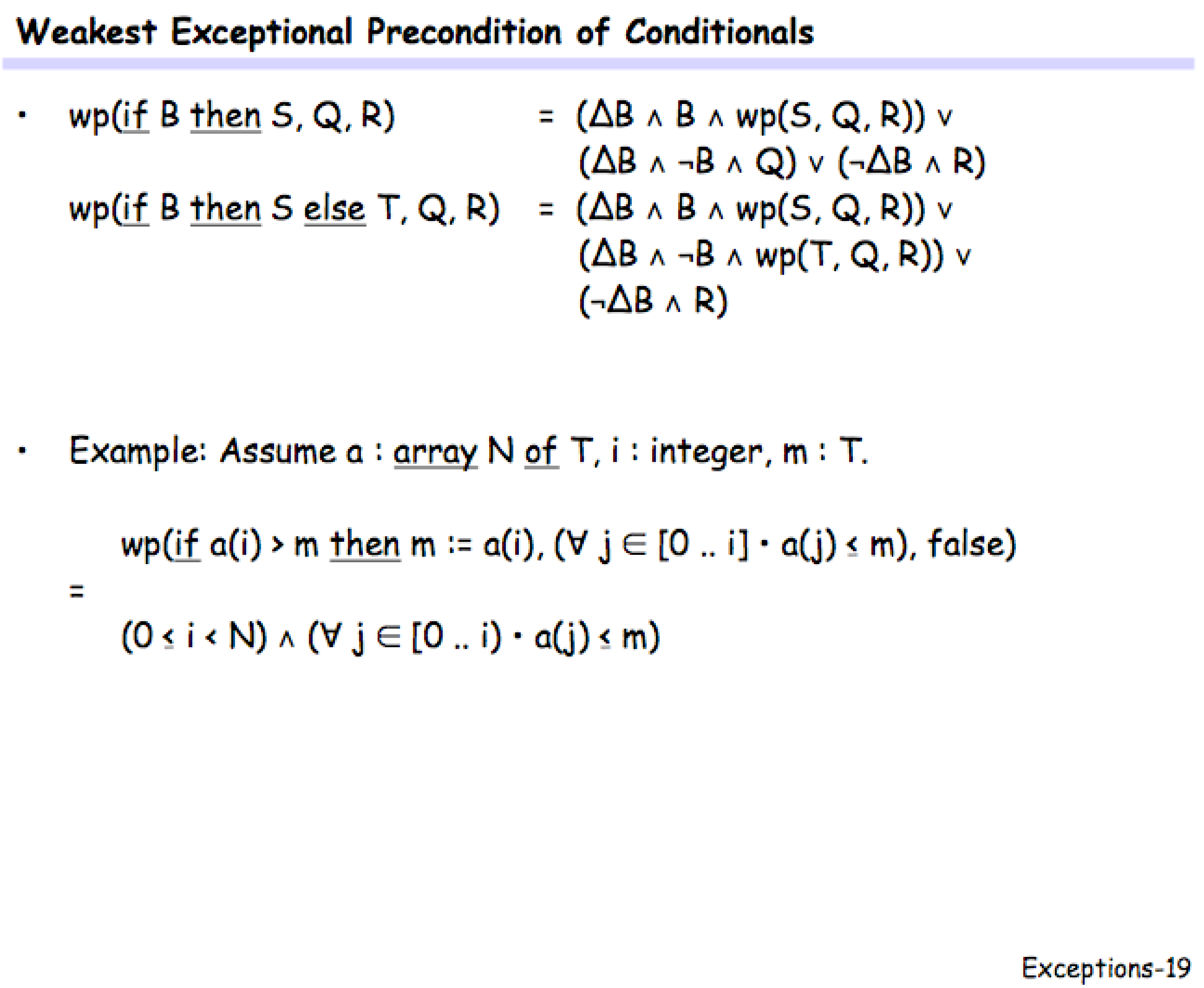
Testing Modules

- Since modules may have private variables, we can neither set nor inspect their values directly.
    - In order to set their values to a desired state, we have to call a sequence of modifiers (modifying public procedures).
    - In order to inspect their values, we have to call one or more observers (observing public procedures).

- With testing in mind, we should include sufficiently many modifiers and observer from the beginning. This leads to the requirement of designing modules for testability.

Testing-30

Excerpt from 5. Exception Handling



Weakest Exceptional Precondition of Conditionals

- wp(if B then S, Q, R) = $(\Delta B \wedge B \wedge wp(S, Q, R)) \vee (\Delta B \wedge \neg B \wedge Q) \vee (\neg\Delta B \wedge R)$

  wp(if B then S else T, Q, R) = $(\Delta B \wedge B \wedge wp(S, Q, R)) \vee (\Delta B \wedge \neg B \wedge wp(T, Q, R)) \vee (\neg\Delta B \wedge R)$

- Example: Assume a : array N of T, i : integer, m : T.

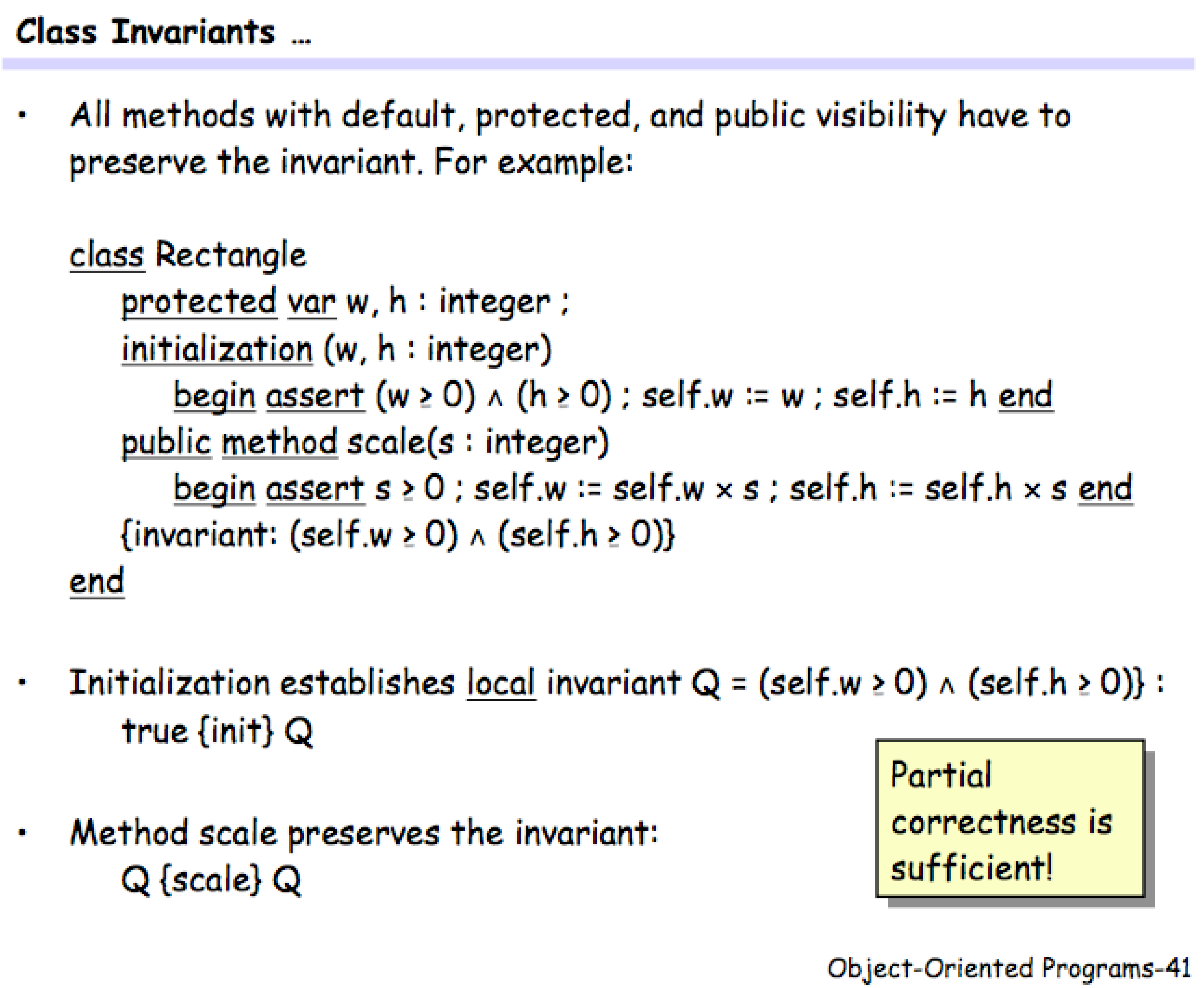
  wp(if $a(i) > m$ then $m := a(i)$, $(\forall\, j \in [0 \,..\, i] \cdot a(j) \le m)$, false)

  =

  $(0 \le i < N) \wedge (\forall\, j \in [0 \,..\, i) \cdot a(j) \le m)$

Exceptions-19

Excerpt from 7. Object-Oriented Programming



Class Invariants …

- All methods with default, protected, and public visibility have to preserve the invariant. For example:

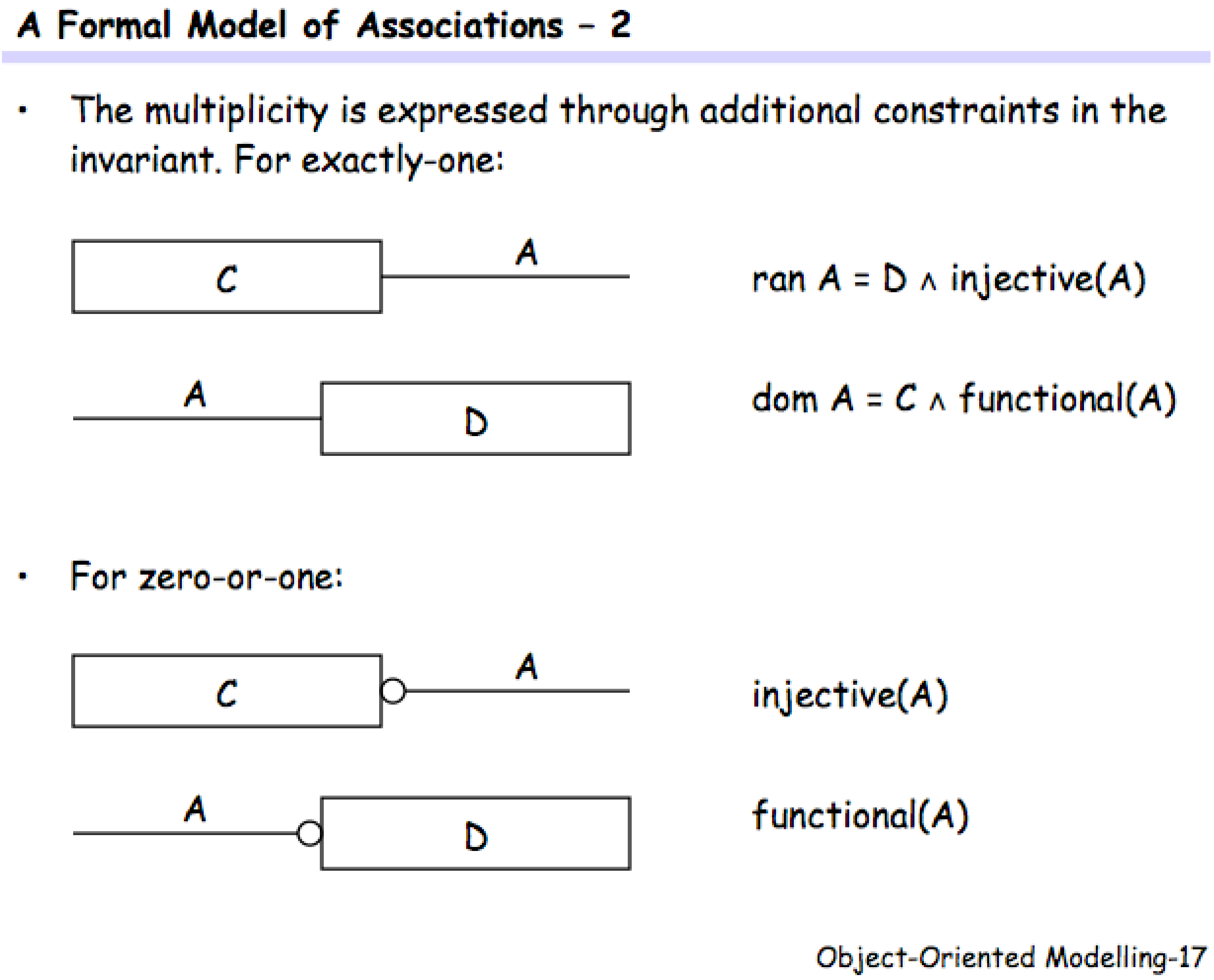
```
class Rectangle
    protected var w, h : integer ;
    initialization (w, h : integer)
        begin assert (w > 0) ∧ (h > 0) ; self.w := w ; self.h := h end
    public method scale(s : integer)
        begin assert s > 0 ; self.w := self.w × s ; self.h := self.h × s end
    {invariant: (self.w > 0) ∧ (self.h > 0)}
end
```

- Initialization establishes local invariant Q = (self.w > 0) ∧ (self.h > 0)} :
  true {init} Q

- Method scale preserves the invariant:
  Q {scale} Q

Partial correctness is sufficient!

Excerpt from 8. Object-Oriented Modeling



A Formal Model of Associations – 2

- The multiplicity is expressed through additional constraints in the invariant. For exactly-one:

$\text{ran } A = D \wedge \text{injective}(A)$

$\text{dom } A = C \wedge \text{functional}(A)$

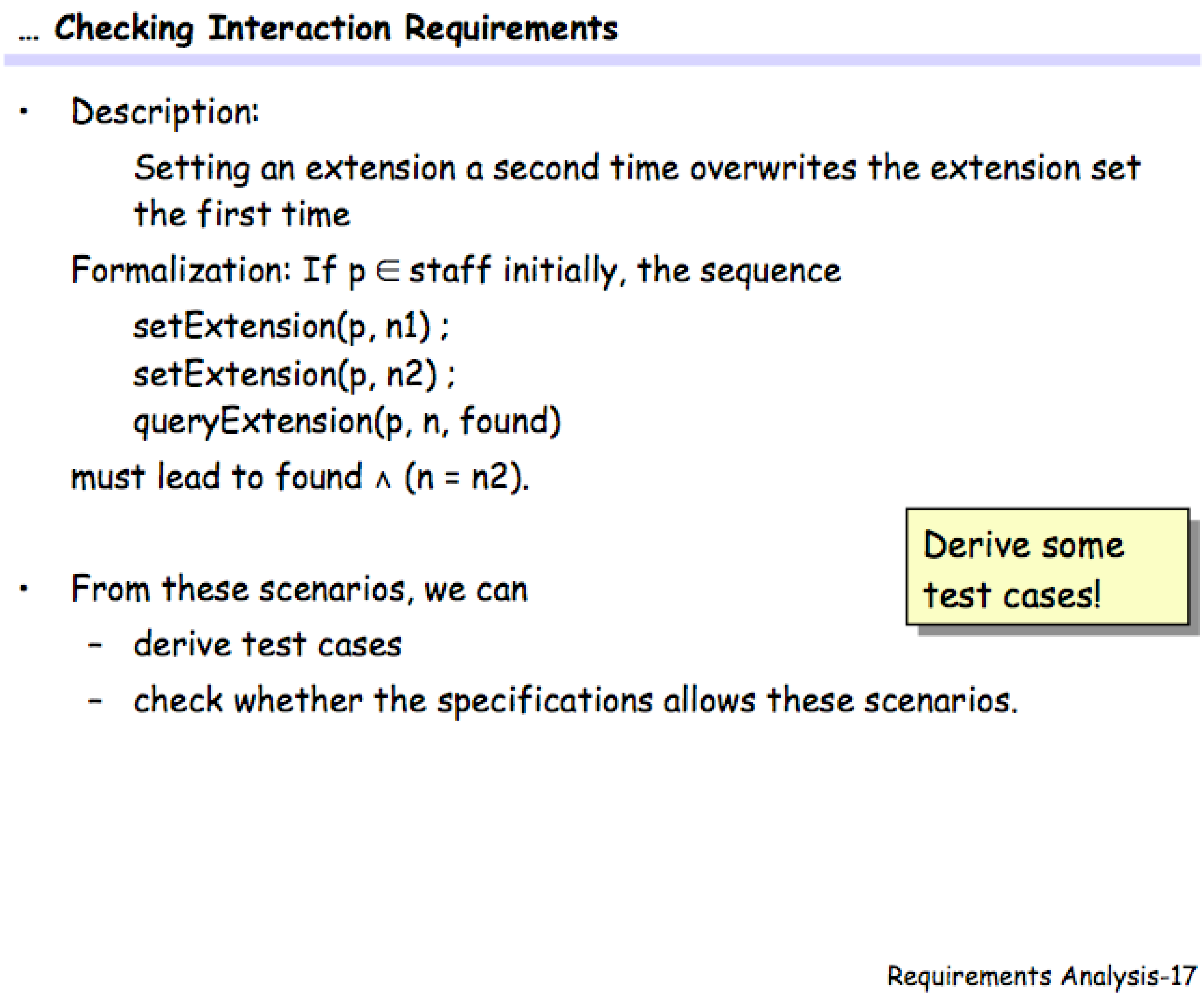
- For zero-or-one:

$\text{injective}(A)$

$\text{functional}(A)$

Object-Oriented Modelling-17

Excerpt from 9. Requirements Analysis



… Checking Interaction Requirements

- Description:

  Setting an extension a second time overwrites the extension set the first time

  Formalization: If $p \in staff$ initially, the sequence

  ```
  setExtension(p, n1) ;
  setExtension(p, n2) ;
  queryExtension(p, n, found)
  ```

  must lead to $found \wedge (n = n2)$.

- From these scenarios, we can
  - derive test cases
  - check whether the specifications allows these scenarios.

Derive some test cases!

Requirements Analysis-17

Middle-out Sequencing of Topics

1. Elements of Programming
2. Program Modularization
3. Abstract Programs
4. Testing
5. Exception Handling
6. Functional Specifications
7. Object-Oriented Programs
8. Object-Oriented Modeling
9. Requirements Analysis
10. Object-Oriented Design
11. Reactive Systems
12. Software Design Process

\+ Configuration Management

- **Tools:**
  - Pascal
  - Java
  - jUnit
  - iState

Evaluation

- No sense of students being math-phobic:
  - At the end of SD2 students take logic for granted
  - (Topic with most difficulties was Object-Oriented Modeling)
- Requiring Logic & Discrete Math prerequisite had moderate effect:
  - Too many differences in notation
  - Large part of SD1 "wasted" on logic & data types
- Evaluation at end of 4th year Software Design Project:
  - SD1 & SD2 among top 3 most useful courses
  - Project rarely show systematic application of concepts
  - Reason: material not repeated in other courses, eg. Of 5 algorithms books with 550-770 pages, 1 has 8 pages on correctness
- Course evaluation:
  - 30%-65% report 81%-100% of material valuable
  - 35%-50% report 61%-80% of material valuable
  - Critical judgment high, course delivery mixed
  - No complaints of overly mathematical contents

Discussion

- **710 pages of lecture notes plus selected articles**
- **Mixture of mathematical and less mathematical topics give confidence that use of mathematics is justified.**
- **No formal tools, no "light" method**
- **Not possible with 1 semester course.**
- **Likely influenced students' way of understanding programs, but little their programming practice**
- **Dijkstra:**

  > ... providing symbolic calculation as an alternative to human reasoning ... is sometimes met with opposition from all sorts of directions: ... 6. the educational business that feels that if it has to teach formal mathematics to CS students, it may as well close its schools.