AOSD Explained:
ASPECT-ORIENTED SYSTEM DEVELOPMENT

Background & Implications

Professor Emeritus John Florentin
Birkbeck College

John Florentin

AOSD - Background

- AOSD - ‘Aspect-Oriented Software Development’,
- AOSD - ‘Aspect-Oriented System Development
- AOP - ‘Aspect-Oriented Programming’
  ‘Aspect-Oriented Programming’, G. Kiczales et al, ECOOP 1997

AOSD - Background

Originally developed by Gregor Kiczales to remedy weaknesses in OO programming.

Also, investigations by William H. Harrison, Peri L. Tarr, Harold L. Ossher, (IBM) on ‘Concerns’.

BCS Advanced Programming interested via constructing programs from components. (AOSD = make your own components, + also guidance on correct choice of components.)

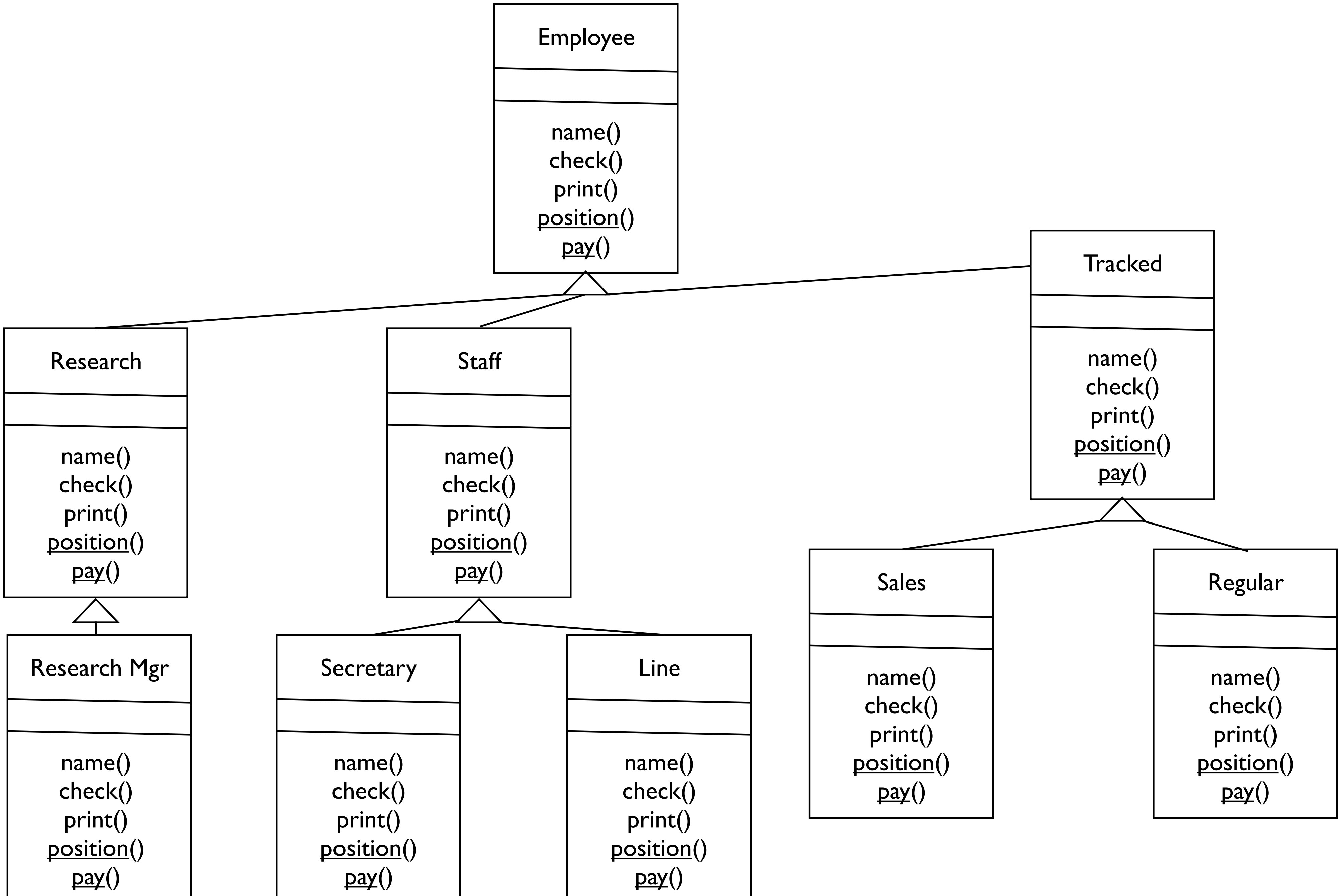
AOSD - Benefits

Addresses Non-Functional Requirements, especially ease of program maintenance.

Comprehendability

Traceability

John Florentin



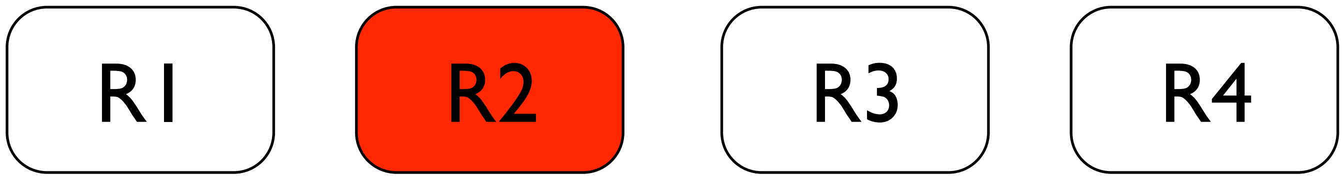
- Personnel feature manages basic information about employees such as name, ID, management chain. Enforces business rules, such as, an employee has 1 to 3 managers
- Payroll feature manages salary and tax information. Enforces further business rules e.g. tax regulations
- Employees are <u>data</u> <u>concerns</u>,
- Personnel, pay, are <u>feature</u> <u>concerns</u>
- Each business rule is a <u>business</u> <u>rule</u> <u>concern</u>

- Concerns - personnel, payroll, business rules - not well separated in the program
- ‘Scattering’, - same code repeated in different classes
- ‘Tangling’ - Different concerns mixed-up in code
- ‘Crosscutting’
- Outcome - costly evolution, complicated integration, brittle software
- ‘Tyranny of the dominant decomposition’

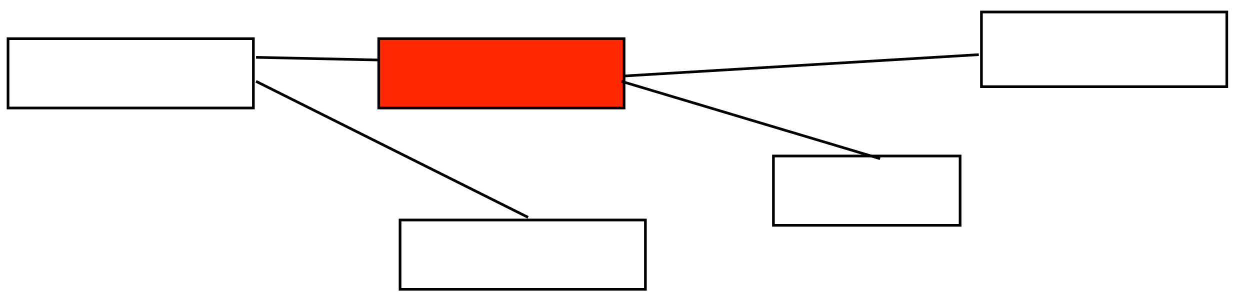
- ASPECTS
  - A concern whose behaviour is triggered by other concerns, usually in multiple situations
- Design, and write code, for each Concern separately. Followed by Weaving (= compose automatically) Concern codes together.

John Florentin

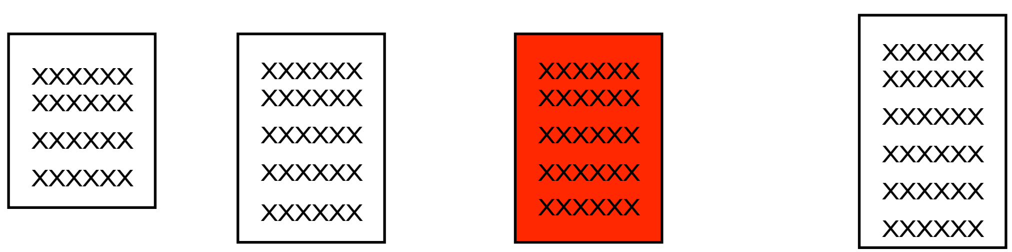
Requirements Modules



Design Modules



Program Modules



AOSD - Background

- THEMES (Siobhan Clarke)
- ‘Aspect-Oriented Analysis and Design: The THEME Approach’ by Siobhan Clarke and Elisa Baniassad, AddisonWesley, ISBN 0-321-24674-8
    - A Theme is an element of design; a collection of structures and behaviours that represent one feature.
- Overlap and interaction: base Themes and crosscutting Themes (= Aspects).

REQUIREMENTS ANALYSIS

- Narrative statement of user Requirements. User terminology.
- Extract Concerns - e.g. 'Security'
- Propose Actions/Functions/Themes modules to realise each Concern, user terminology - e.g. 'Check password'.
- Decide on Base - Aspect relationship between Actions/ Functions/Themes

EXAMPLE - Course Management System Requirements

R1. Students can register for courses

R2. Students can unregister for courses

R3. When a student registers then it must be logged in their record

R4. When a student unregisters it must also be logged

R5. Professors can unregister students

EXAMPLE - Course Management System Requirements Continued

R6. When a professor unregisters a student it must be logged

R7. When a professor unregisters a student it must be flagged as special

R8. Professors can give marks for courses

R9. When a professors gives a mark this must be logged in the record

R1. Students can <u>register</u> for courses

R2. Students can <u>unregister</u> for courses

R3. When a student registers then it must be <u>logged</u> in their record

R4. When a student unregisters it must also be logged

R5. Professors can unregister students

R6. When a professor unregisters a student it must be logged

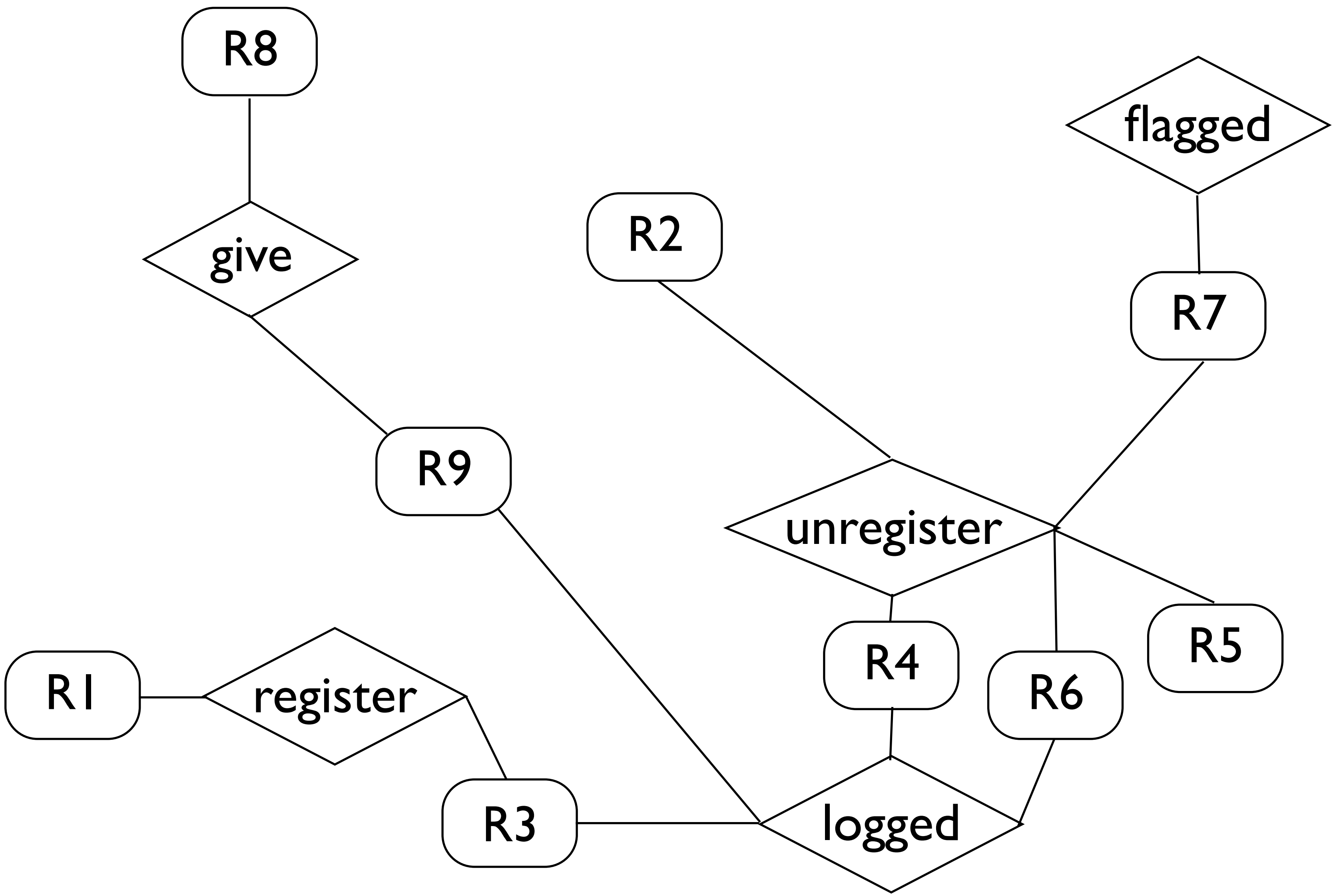
R7. When a professor unregisters a student it must be <u>flagged</u> as special

R8. Professors can <u>give</u> marks for courses

R9. When a professor gives a student a mark this must be logged in their record

John Florentin

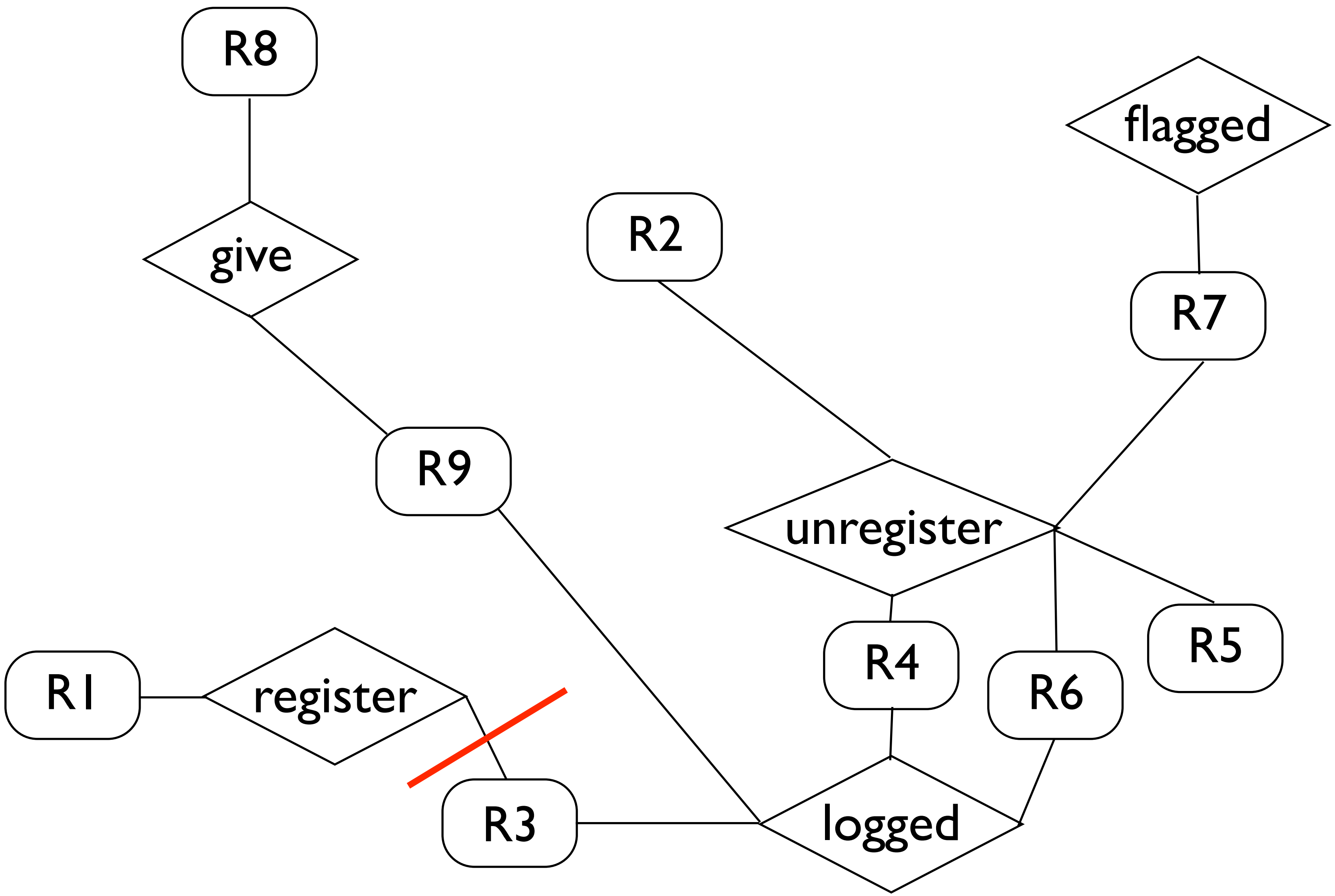
ACTION View



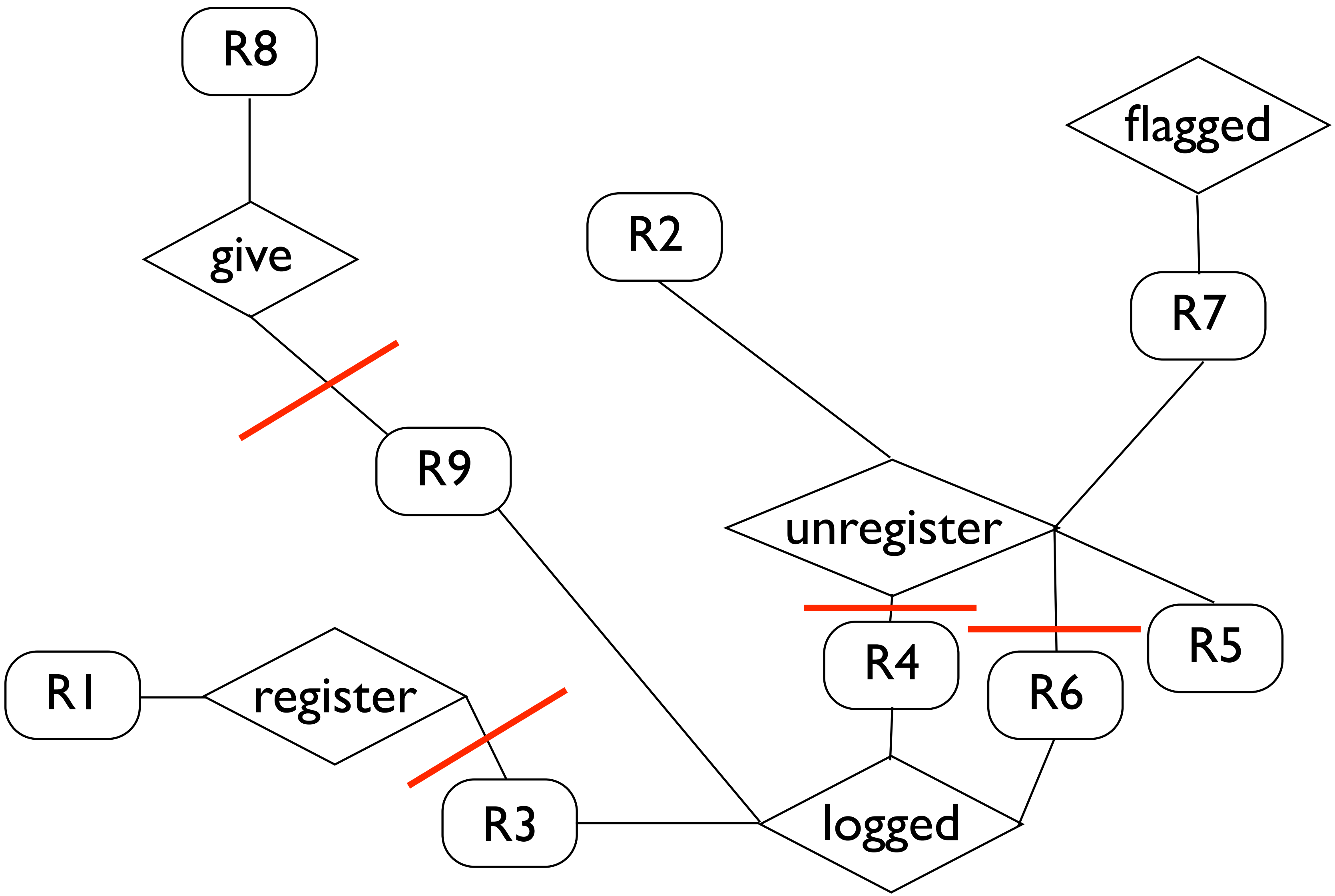
Identifying Aspects

- Want to associate each Requirement with just one Action. If there is more than one Action, decide on one main Action, and the other Actions are (crosscutting) <u>Aspects</u>
- Take R3 - ‘register’ and ‘logging’ Actions. Intuit that ‘logged’ is a behaviour, which crosscuts ‘register.’ ‘register’ is base

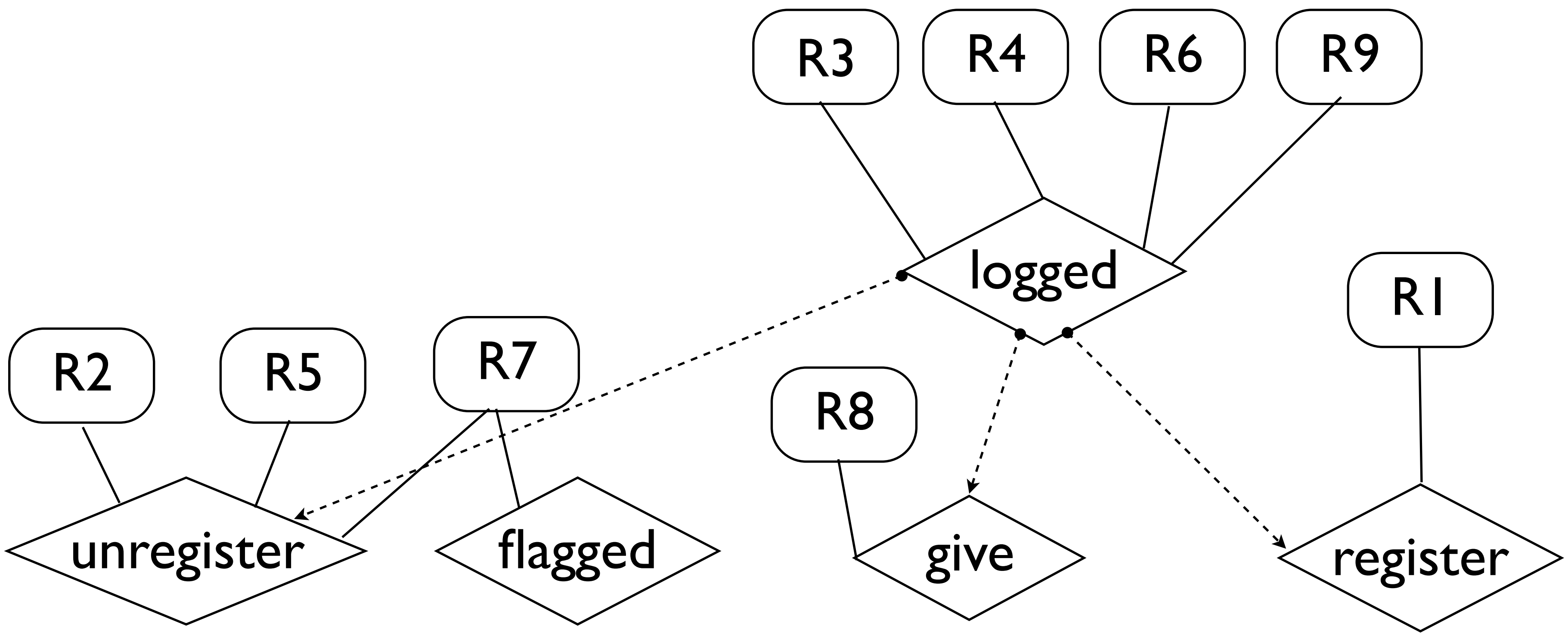
ACTION View


R8
give
R2
flagged
R7
R9
unregister
R4
R6
R5
R1
register
R3
logged

ACTION View

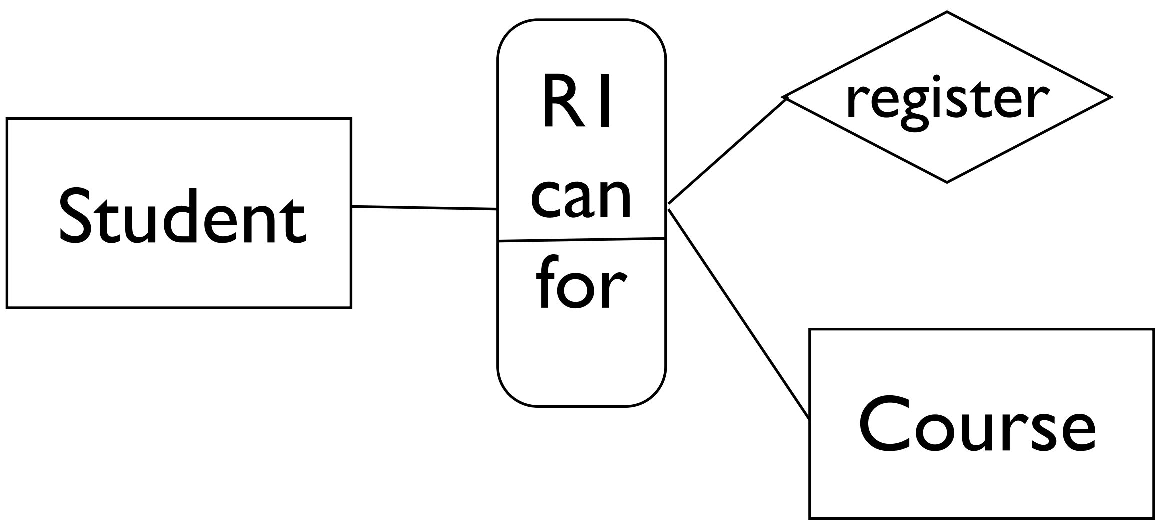

R8
give
R9
R2
flagged
R7
unregister
R4
R6
R5
R1
register
R3
logged

CLIPPED View



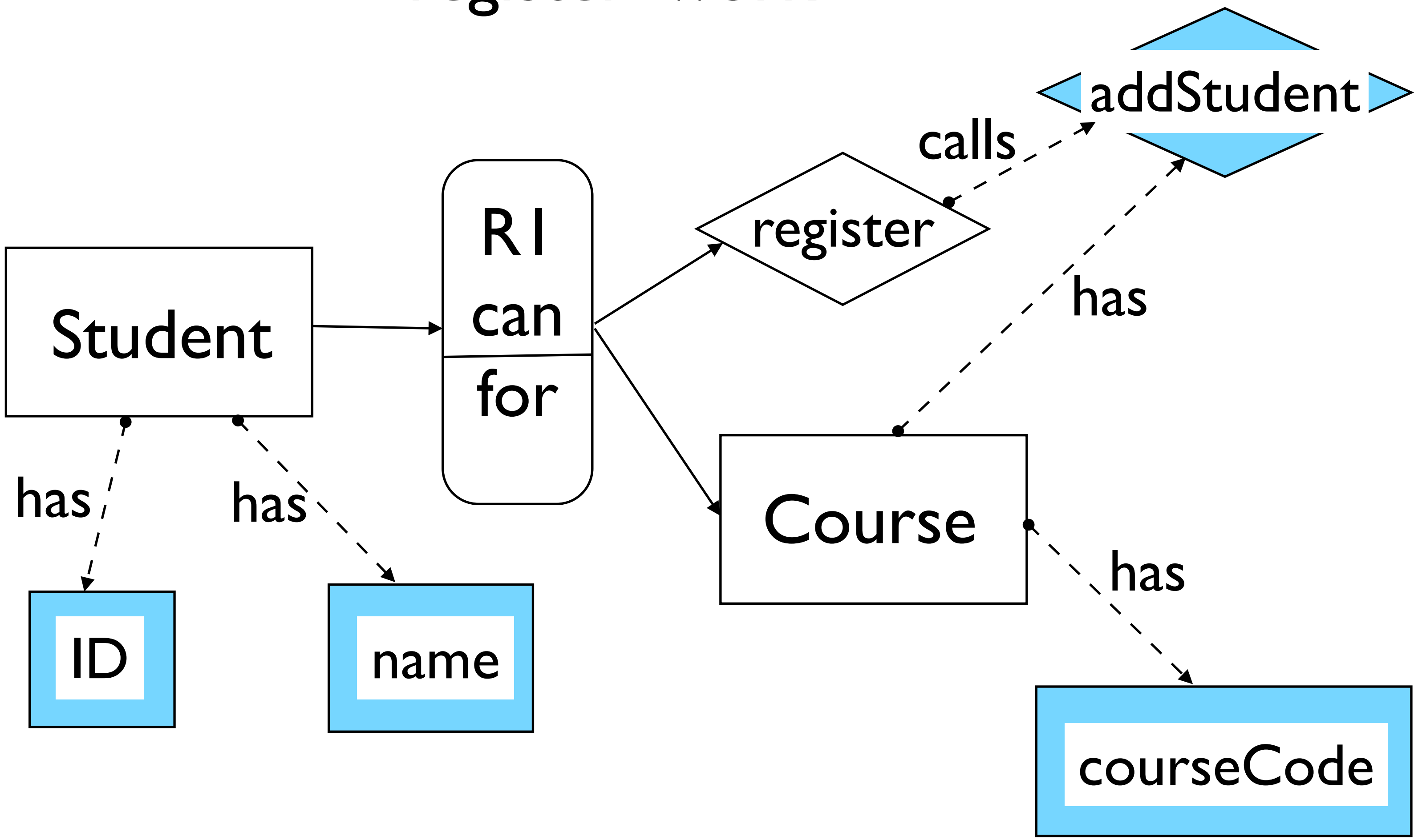
John Florentin

THEME View
Shows Entities and Actions
‘register’ is base



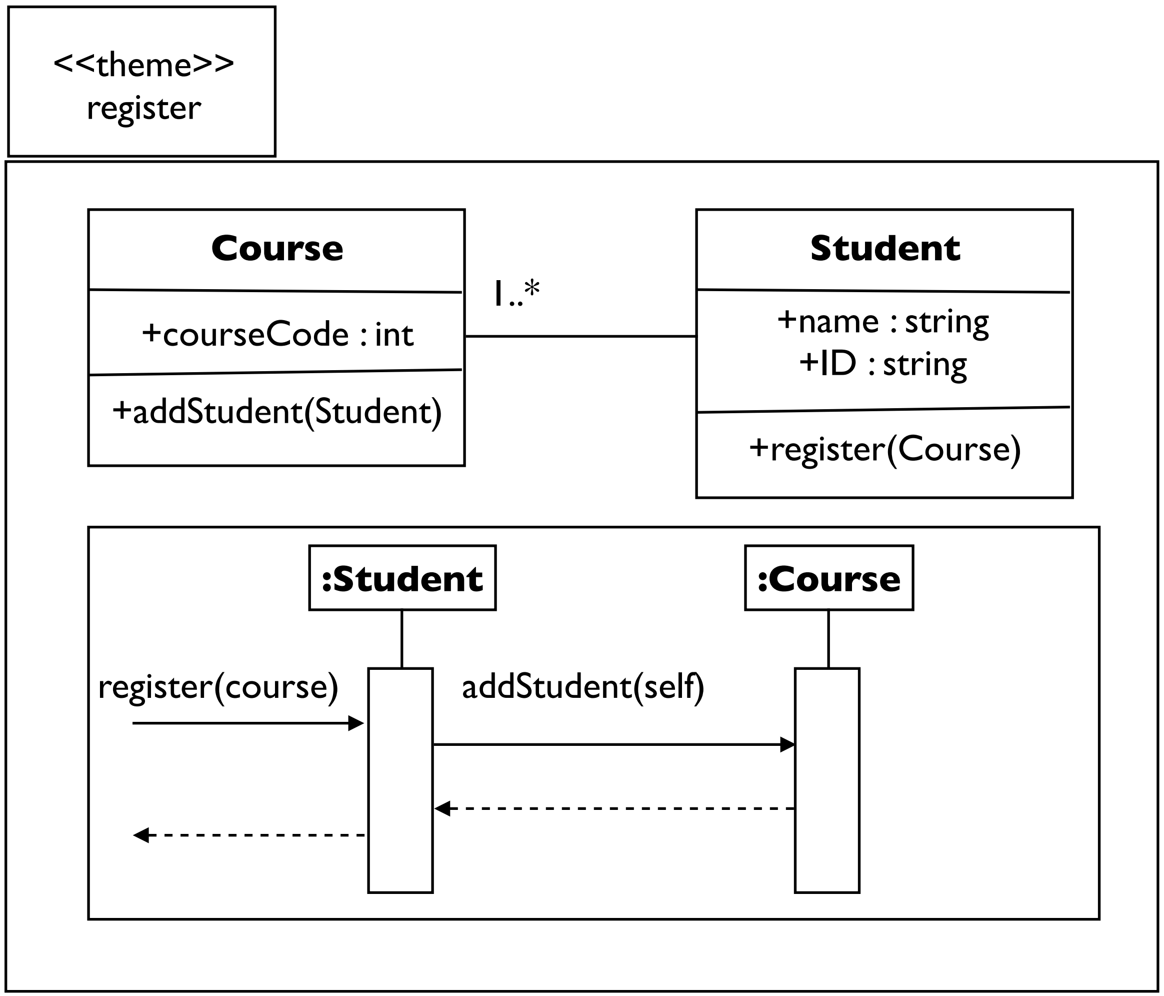
THEME View
Augmented View additions to make 'register' work

John Florentin



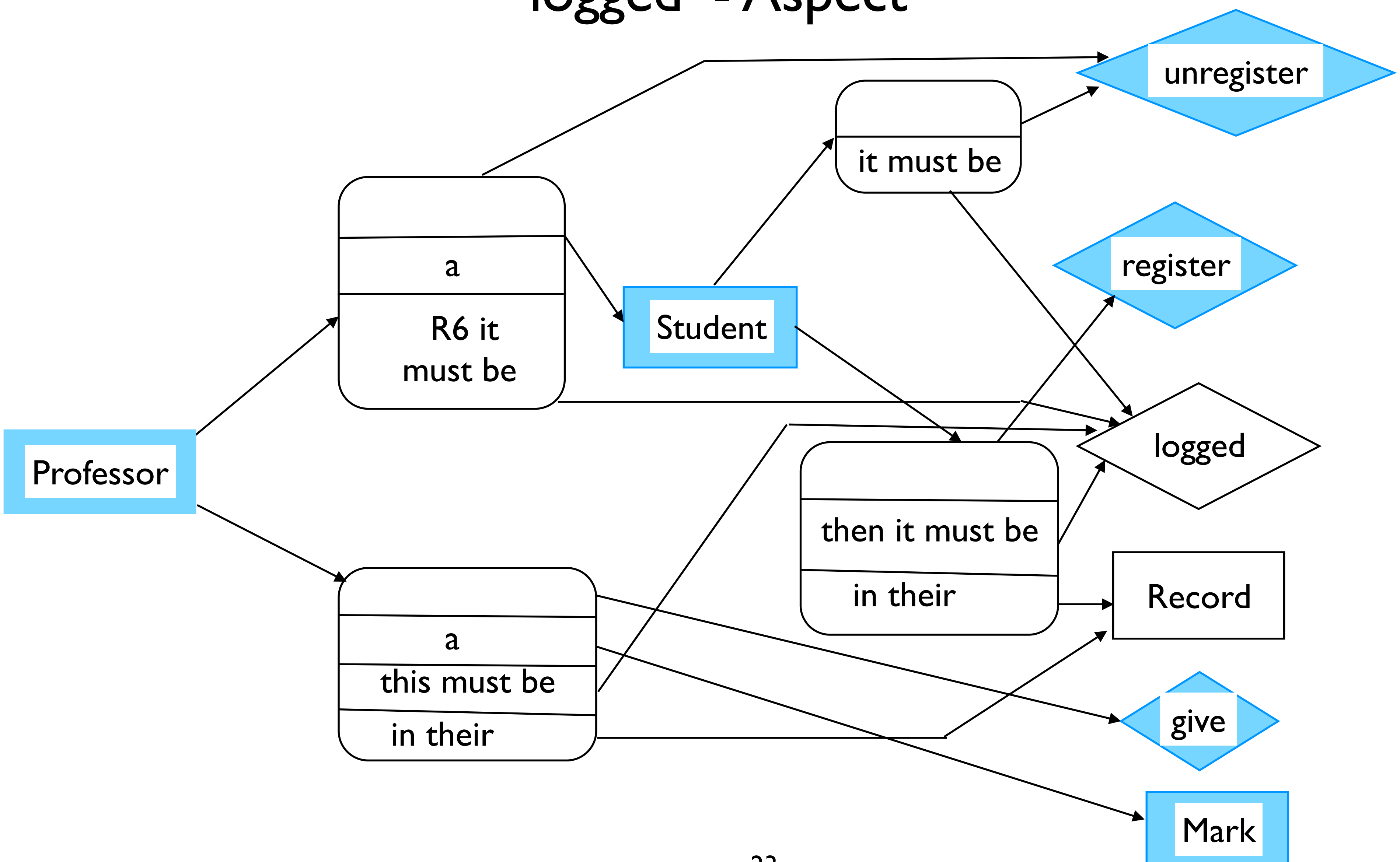
John Florentin

THEME - ‘register’ OOP implementation



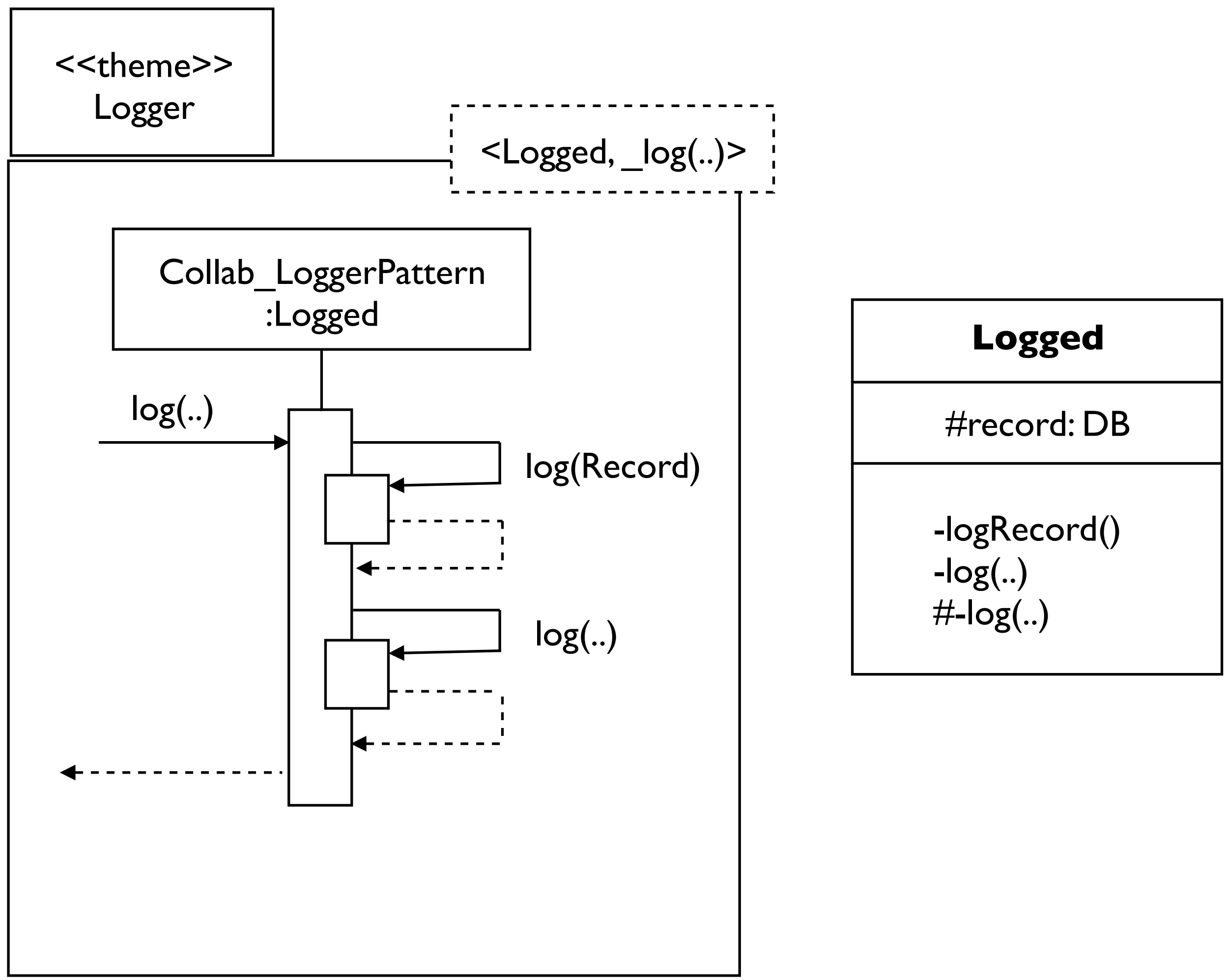
John Florentin

Augmented View
‘logged’ - Aspect

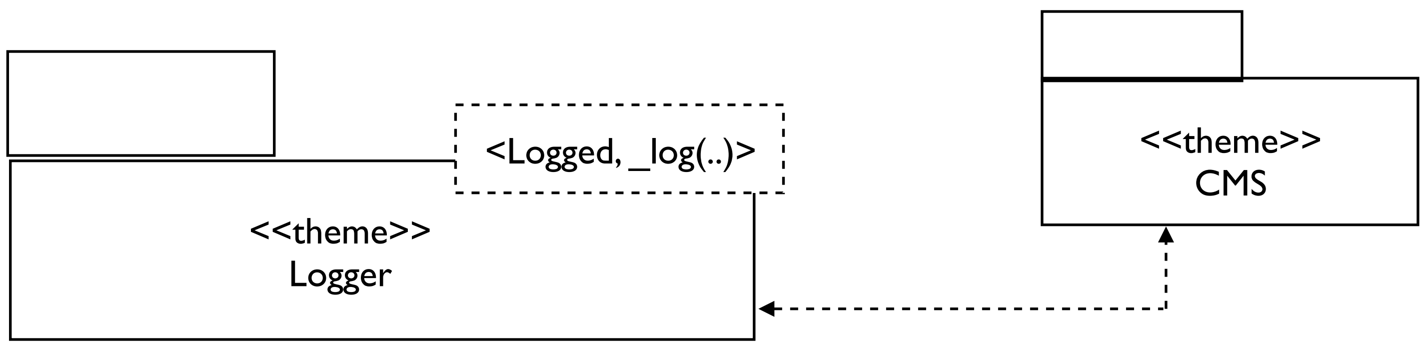


John Florentin

THEME - ':logged' OOP implementation



Composition of ‘logged’ and other Themes



bind[<{Person, Student, Professor}, {Student.register(), Person.unregister(), Professor.giveMark()}>]

John Florentin

Concerns

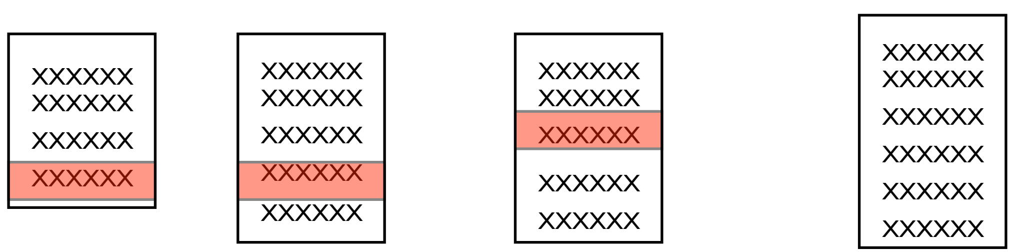
C1 C2

Themes

T1 T2 T3 T4

Theme Program Modules

Woven Code



IMPLICATIONS FOR SYSTEM DEVELOPMENT

- The REAL Task
  - Given a set of narrative Requirements written in user terminology, plus a proposed hardware configuration.
  - Produce a Code (+ Object?) design and implementation which fully covers the Requirements, and as little as possible extra

John Florentin

IMPLICATIONS FOR SYSTEM DEVELOPMENT

- Structure of code must incorporate the structure of Requirements. Irreducible code complexity results from complexity within Requirements;
- Refine and re-factor Requirements using notion of Concerns and possibly Themes
- Program design based directly on the refined Requirements structure
- Traceability - Requirements ⟷ code

IMPLICATIONS FOR DESIGN OF PROGRAMMING LANGUAGES

'Software design processes and programming languages exist in a mutually supporting relationship. Design processes break a system down into smaller and smaller units. Programming languages provide mechanisms that allow the programmer to define abstractions of system sub-units, and then compose those abstractions in different ways to produce the overall system. A design process and a programming language work well together when the programming language provides abstractions and composition mechanisms that cleanly support the kinds of units the design process breaks the system into.'

from 'Aspect-Oriented Programming', G. Kiczales et al, ECOOP 1997

IMPLICATIONS FOR DESIGN OF PROGRAMMING LANGUAGES

- High-level programming languages based on conceptual paradigms, ('- - *abstractions and composition mechanisms - -'*), e.g. functional, modular, OO
- Paradigms help match code to application elements, also make code structures more comprehensible; e.g. entities to objects

IMPLICATIONS FOR DESIGN OF PROGRAMMING LANGUAGES

- ‘Object-Oriented Programming in Oberon-2’, Hanspeter Mössenbock, Springer-Verlag 1993, ISBN 3-540-56411-X or, 3-387-56411-X
- ‘Modula-3’, Samuel P. Harbison, Prentice Hall 1992, ISBN 0-13-596396-6,
- ‘Aspect-Oriented Analysis and Design: The THEME Approach’ by Siobhan Clarke and Elisa Baniassad, AddisonWesley 2005, ISBN 0-321-24674-8

IMPLICATIONS FOR DESIGN OF PROGRAMMING LANGUAGES

- 'Aspect-Oriented Programming', G. Kiczales et al, Proceedings ECOOP 1997
- 'Aspect-Oriented Software Development With Use Cases', Ivar Jacobson and Pan-Wei Ng, Addison-Wesley 2005, ISBN 0-321-26888-1
- 'Aspect-Oriented Software Development', Robert E. Filman et al, Addison-Wesley 2005, ISBN 0-321-21976-1, note chapter 24 'Aspect-Oriented Dependency Management'