AOSD Explained:
ASPECT-ORIENTED SYSTEM DEVELOPMENT

Background & Implications

Professor Emeritus John Florentin
Birkbeck College
AOSD - Background

- AOSD - ‘Aspect-Oriented Software Development’,
- AOSD - ‘Aspect-Oriented System Development’
- AOP - ‘Aspect-Oriented Programming’
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
• 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 data concerns,

• Personnel, pay, are feature concerns

• Each business rule is a business rule concern
• 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.
Requirements Modules

R1  R2  R3  R4

Design Modules

Program Modules
AOSD - Background

• THEMES (Siobhan Clarke)


• 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 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
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 professor gives a student a mark this must be logged in their record
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) Aspects.

• Take R3 - ‘register’ and ‘logging’ Actions. Intuit that ‘logged’ is a behaviour, which crosscuts ‘register.’ ‘register’ is base.
ACTION View

R1 register

R8 give

R3

R9

R2

unregister

logged

flagged

R4

R7

R5

R6
CLIPPED View
THEME View
Shows Entities and Actions
‘register’ is base

Student → R I can for → register → Course
THEME View
Augmented View
additions to make ‘register’ work

student can for

R I can for

addStudent calls

register has

has

Student

Course

has

courseCode

ID

name

THEME - ‘register’
OOP implementation

<<theme>>
register

Course
+ courseCode : int
+ addStudent(Student)

Student
+ name : string
+ ID : string
+ register(Course)

:Student
register(course) → addStudent(self)

:Course

AOSD
John Florentin
Augmented View
‘logged’ - Aspect

Professor

Student

register

unregister

loged

Record

give

Mark

Professor

Wood it must be

a

R6 it must be

Student

then it must be

in their

loged

Record

give

Mark
THEME - ‘:logged’
OOP implementation

<<theme>>
Logger

Collab_LOGGERPattern:
:Logged

Logged

#record: DB

-logRecord()
-log(..)
-#-log(..)
Composition of ‘logged’ and other Themes

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

Concerns

Themes

Theme Program Modules

Woven Code

John Florentin
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
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.’

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


IMPLICATIONS FOR DESIGN OF PROGRAMMING LANGUAGES

