Information Systems Concepts

Refining the Requirements Model

Roman Kontchakov

Birkbeck, University of London

Outline

- Software and Specification Reuse
  - Section 20.2 (pp. 585 – 586)
  - Section 8.2 (pp. 234 – 237)
  - Section 8.4 (pp. 246 – 247)
  - Section 8.5.1 – 8.5.2 (pp. 252 – 253)

- Adding Further Structure (to Class Diagrams)
  - Section 8.3.1 – 8.3.3 (pp. 237 – 244)
  - Section 14.4.4 (pp. 409 – 410)
Why Reuse?

- The arguments for reuse are
  - partly economic
    - saving time and effort in software development including software testing and quality assurance
  - partly concerned with quality
    - fewer defects
  - partly about business flexibility
    - faster time to market
How O-O Contributes to Reuse

- Inheritance and Encapsulation
  - Two main forms of *abstraction* that O-O relies on to achieve reuse
- Components
- Patterns
Reuse: Encapsulation

- allows one class or component to be replaced by another with different internal details, as long as they adhere to the same external interface
  - thus classes or components can be used in systems for which they were not originally designed
- a group of classes can be encapsulated through aggregation or composition to become a reusable subassembly

*Universal Serial Bus (USB)*


*Plug and Play*
Reuse: Inheritance

- encourages identifying those aspects of a design or specification that has general application to a variety of situations or problems
- allows the creation of new specialised classes when needed, with little effort

“Do not reinvent the wheel!”
http://en.wikipedia.org/wiki/Wheel
http://images.google.co.uk/images?q=wheel&hl=en
same circular form and central shaft
Reuse: Components

- For example, a house (bricks, tiles, doors, windows, pipes, etc.), a home theatre (a big screen TV, a DVD player, a decoder, an amplifier, speakers, etc.), ...

- Software development has concentrated on inventing new solutions. Recently, the emphasis has shifted. Much software is now assembled from components that already exist.
Reuse: Patterns

- Next year, Information Systems Management
A generalization structure can be added when two classes are similar in most respects, but differ in some details such as:

- behaviour (operations or methods)
- data (attributes)
- associations with other classes
Adding Generalization Structure

Superclass associations are inherited by subclasses

**Superclass**

- StaffMember (abstract)
  - staffName
  - staffNo
  - staffStartDate
  - calculateBonus()
  - assignNewStaffGrade()
  - getStaffDetails()

**Two subclasses**

- AdminStaff
  - gradeName
  - calculateBonus()

- CreativeStaff
  - qualification
  - calculateBonus()
  - assignStaffContact()
Liskov Substitution Principle

- In object interactions, it should be possible to treat a derived object as if it were a base object without integrity problems.
  - If the principle is not applied, then it may be possible to violate the integrity of the derived object.

Prof Barbara Liskov

2009 A. M. Turing Award Winner
Disinheritance of `debit()` means that the left-hand hierarchy is not Liskov compliant.
Aggregation and Composition

- Two special types of association
  - Aggregation represents a **whole-part** relationship between classes
  - Composition expresses a similar relationship but differs in showing a **stronger** form of ownership by the whole
    - Each part may belong to only one whole at a time.
    - When the whole is destroyed, so are all its parts.
Notation: Aggregation

- A student could be in a number of modules
- If a module is cancelled, students are not destroyed

unfilled diamond denotes aggregation
Notation: Composition

- An ingredient is in only one meal at a time
- If you drop your meal on the floor, you probably lose the ingredients too

filled diamond denotes composition
Take Home Messages

- Software and Specification Reuse
  - Why Reuse
  - How O-O Contributes to Reuse
- Adding Further Structure (to Class Diagrams)
  - Generalization/Specialization
    - Liskov Substitution Principle
  - Aggregation and Composition