Software Engineering and OO Issues

CS3: SEOC1

Note 2

Software Engineering: How to build good systems

- Define and control the process of development
- Have well defined requirements
- Incorporate V and V into the development process
- Use/Reuse relevant knowledge, architectures and components
- Make sensible use of tools

What is a “Good” System?

- Useful and usable
- Reliable
- Flexible
- Affordable
- Available

Objects and Components: Good Systems Have...

- Encapsulation
- Abstraction
- Architecture
- Components

Encapsulation: Low Coupling

The process of hiding all the details of an object that do not contribute to its essential characteristics; typically the structure of an object is hidden, as well as the implementation of its methods.

- Coupling should be low in the system
- Based on dependencies – ensures manageability
- Interfaces, what a module provides, what context it needs, control use of modules

Abstraction: High Cohesion

The essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer; the process of focusing upon the essential characteristics of an object.

- Module should have high cohesion
- High cohesion indicates good abstraction
- Encapsulation of internal details hides structure

Abstraction: You don't need to know
Encapsulation: and I'm not going to tell you.

Architecture and components

- Object-based approach not, usually, leading to desired levels of reuse.
- Component is the unit of reuse and replacement
- Architecture controls interaction
- Architecture encourages reuse, may be reused itself

Does this help make better Systems?

- Classes – high cohesion, low coupling
- "natural" modelling:
  - ease requirements capture
  - track changes more easily
  - natural interactions
- Reuse
  - high cohesion and low coupling supports reuse
  - but... classes are often too small
  - components (can be) cohesive collections of classes
- OOPLs support these things

(Some) Software Development Phases

- Requirements
- Design
  - specification
  - formal specification
- Validation
  - ... and verification
  - testing
  - prototyping
- Implementation and integration
- Maintenance and evolution

(Some) Software Development Lifecycles

- Waterfall model

  Pros: highly visible, easy tracking

  Cons: linear, hence vulnerable to change

- Evolutionary model

  Pros: very robust

  Cons: tracking difficult, visibility compromised

- Spiral model

  Pros: very robust, highly visible

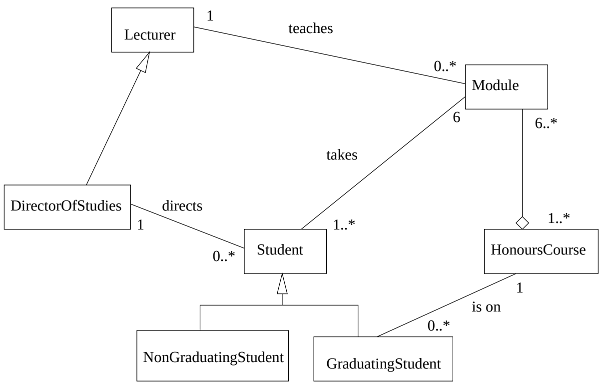
  Cons: dependent on high quality management

- Others
  - SSADM, DSDM, Rational Unified process, Extreme Programming (XP), ...

Development Process Methodologies

- Fault avoidance
- Metrics and measurement
- Software management
  - project planning
  - process management
  - risk management
- OO design
  - advantages: easy to understand, maintenance, reuse
  - disadvantages: OO difficult, not always applicable
- COTS (Commercial, Off The Shelf software) and trusted components

Static system structure: OO flavour *Using UML: Figure 15.2*



In a traditional (non-OO) programming paradigm, the module structure would be a tree; how could you make the above conceptual model fit that structure?

Unified Modelling Language

The industry standard modelling language **UML** provides a language in which to talk about designs. It doesn't say anything about how to get the designs.

- **Genesis of OO** SIMULA (1962˜1967); Smalltalk (1972˜1980); C++ (1983˜1985); Java (1991˜1995); ...

  1989–1994 OO "method wars"

  1994–1995 three Amigos (Booch, Jacobson, Rumbaugh) and birth of UML

- The UML language(s):

  Requirements: Use Cases

  Statics: Class Diagrams

  Dynamics: Collaboration, Sequence, State, Activity Diagrams

  Implementation ...

Object Oriented Analysis with UML

- Requirements
  - use cases
- Static Model
  - class diagrams
- Dynamic Model
  - interactions diagrams
  - statecharts
  - activity diagrams
- Validation
  - CRC cards
  - interaction diagrams
- Attributes and operations
  - class diagrams

(In)-Applicability of OO Model

- Metaphor of real world being "objects which pass messages" may be:

  inappropriate: basing system structure on real world objects may not lead to best, or even good, design

  inaccurate: eg, two people colliding are not sending "bump" message
- OO (arguably) best understood as "particular approach for relating data and processing in software systems"
  - traditional procedural systems separate data and processing functions (data stored in one place)
  - OO systems decompose data and localise/integrate with relevant operations

Conceptual Pitfalls

- Believing the hype:
  - Going OO for the wrong reasons
  - Thinking objects come for free
  - Thinking objects will solve all problems
- Mistaking style for substance:
  - confusing buzzwords with concepts
  - confusing tools with principles
  - confusing presentation with methodology
  - confusing training with skill
  - confusing prototypes with finished products