ELEC 372

Introduction
and
Syllabus

Section A: T.C. Nicholas Graham
Section B: Diane Kelly

Hours

- Section A: Nicholas Graham
  - Monday 10:30
  - Wednesday 9:30
  - Friday 8:30
  - JEF 127
- Office Hours:
  - 629 Goodwin
  - Hours:
    - Monday 12:00 to 1:00
    - Wednesday 11:30 to 12:30
    - Friday 12:00 to 1:00
- Tutorial:
  - Thursday 11:30 to 12:30
  - BIO 1102
  - no tutorial first week

For details see **http://www.cs.queensu.ca/~elec372**

Hours

- Section B: Diane Kelly
  - Monday 8:30
  - Tuesday 10:30
  - Thursday 9:30
  - MCL315
- My office:
  - 554 Goodwin
  - Hours:
    - Tuesday 8:30 to 10:00
    - Thursday 8:30 to 9:00
- Tutorial:
  - Tues. 1:30 to 2:30
  - Dupuis Auditorium
  - no tutorial first week

Marking Scheme

- midterm exam
  - week of Feb. 12-16
  - common midterm held outside class
  - 10%
- final exam
  - 50%
- 4 assignments
  - 10% each
  - Java programming
  - OO design
  - testing/inspection
  - design patterns
  - architecture

Textbooks

- Required text:
  - Grady Booch, Object-Oriented Analysis and Design with Applications, 2nd ed, Addison-Wesley, 1994
- Recommended texts:
  - Fowler, Scott and Booch, UML Distilled, 2nd ed: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 1999
  - Gamma, Helm, Johnson and Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
  - Bass, Clements and Kazman, Software Architecture in Practice, SEI Series in Software Engineering, 1998
  - Brooks, The Mythical Man-Month, Anniversary Edition, Addison Wesley, 1995

On reserve in library

Topics

- Introduction
- Object Oriented Programming and Design
- Software Development Processes and Techniques
- Design Patterns
- Software Architecture

What is Software Engineering? (1)

- Reference: Software engineering A practitioner's Approach 4th ed; Roger S. Pressman, McGraw Hill, 1997
- From NATO conference in 1969:
  - "[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

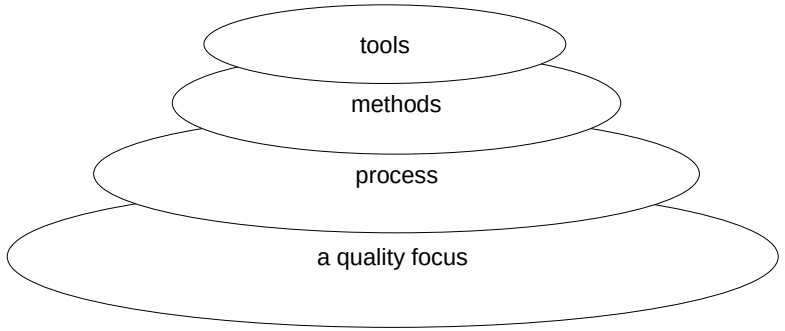
What is Software Engineering? (2)

- evaluation of this definition
  - doesn't address
    - technical aspects of software quality
    - need for customer satisfaction
    - timely product delivery
    - importance of measurement and metrics
    - importance of mature process
  - does provide a baseline
    - does mention
      - engineering principles
      - economy
      - reliability
      - efficiency
      - reality

What is Software Engineering? (3)

- IEEE Standards Collection for Software Engineering, 1993
- "Software engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)."

What is Software Engineering? (4)



Software engineering layers

Software Engineering Topics (1)

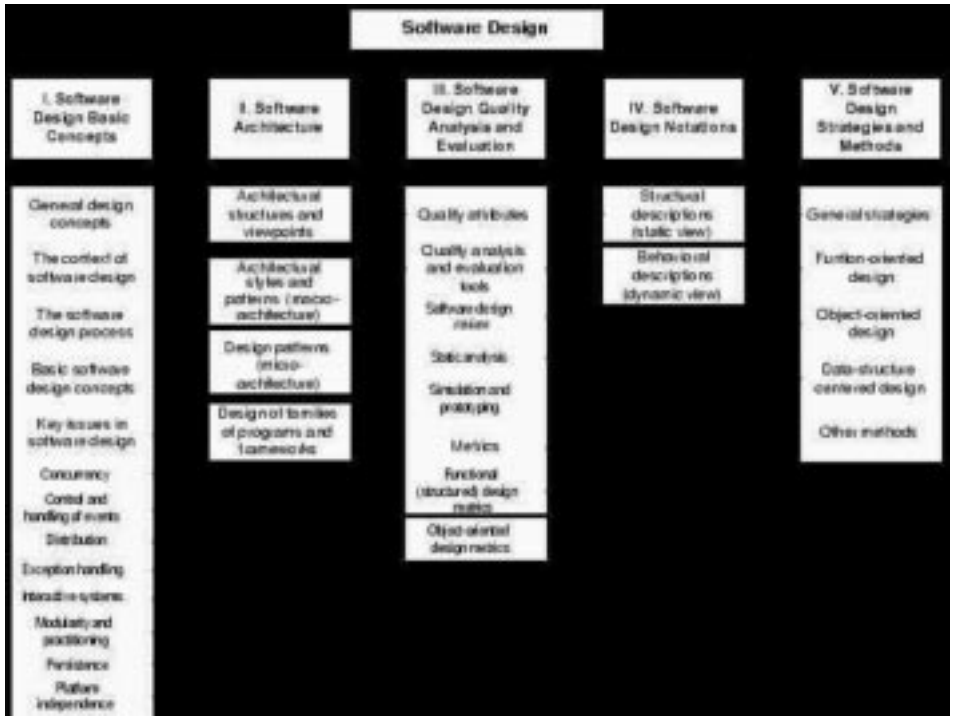
- according to SWEBOK (Université de Québec à Montréal)
  - software requirements
  - software design
    - architecture
    - detailed design
  - software construction
    - languages
    - tools
    - standards
    - unit testing
    - design
  - software testing
  - software maintenance

Software Engineering Topics (2)

- software configuration management
- software engineering management
- software engineering process
- software engineering tools and methods
  - notation, vocabulary, procedures, environments
  - software design tools
  - software construction tools, editors, compilers, debuggers
  - testing, maintenance, process centered tools
  - inspection, static analysis tools
  - configuration management tools
  - project management tools
  - communication tools

Software Engineering Topics (3)

- software quality
  - definition, standards
  - relationship to process
  - quality assurance
  - verification and validation
  - reviews and inspections
  - testing
  - metrics



ELEC 372

Overview
Software Complexity

References

- Measuring Software Design Quality, Card & Glass, Prentice Hall, 1990
- Object Oriented Analysis and Design with Applications 2nd ed, Booch, Addison Wesley, 1994

Views of complexity

- computational
  - machine efficiency
- cognitive
  - human issues of difficulty and understandability
- "The complexity of software is an essential property, not an accidental one." (Brooks)

Modularity

- most widely accepted quality objective for design is modularity
  - produce modules (units) that are as self-contained and independent as possible
  - limit both data and control connections amongst modules
  - Parnas's principle
    - information hiding
    - based on limiting the scope of changes

Module Strength/Cohesion and Coupling

- module strength/cohesion measures "singleness of purpose"
  - module performing only a single function has the greatest strength
  - module performing several unrelated functions has low strength
- module coupling measures "dependence" between modules
  - two modules linked only through data passed in a calling sequence are the weakest coupling
- software quality improved by
  - maximizing module strength
  - minimizing module coupling

Module Size (1)

- simple approach to promoting modularity is to limit module size
  - many texts suggest limiting modules to 50 to 60 source lines of code
  - military standards for module size range from 50 to 200 LOC
  - such standards easy to define and enforce
  - contribution to software quality unclear
- studies have shown
  - program size does not exhibit any strong relation to error rate
  - programmer experience proves to be best predictor of error rate

Module Size (2)

- module cost in terms of hours per executable statement does appear to be related to module size
  - results indicate larger modules cost less (per executable statement) than smaller ones
  - some savings come from distributing interface definition costs over a large number of executable statements
  - part of these savings can be lost due to increased testing requirements

Span of Control

- when a module invokes another module, it establishes a control couple between the modules
  - number of other modules invoked by a given module defines its span of control
- results indicate that modules with larger spans of control tend to cost more and have higher fault rates

Product Engineering with Complexity Criteria

- quality relation needed to be considered by designer: producability
  - developing a design that provides for
    - implementability
    - minimizes cost
    - minimizes error insertion
    - maximizes maintainability
- system design prescribes strategy for implementing requirements
  - difficulty of strategy determines propensity of developers to make errors
  - system design tends to be first cause of errors
- unit design is first cause of poor maintainability

Aspects of Complexity (1)

- requirements are assigned to major components or subsystems
- system design complexity can be viewed as having two components
  - complexity contained within each part defined by the design
  - complexity of relationships among the parts
- components of complexity
  - functional complexity
  - system design complexity
  - structural complexity
  - data complexity
  - procedural complexity

Aspects of Complexity (2)

- functional complexity
  - some applications are inherently more difficult (complex) than others
    - computational flow dynamics
    - analysis of noisy stochastic data
    - communication line handling
    - large data structure handling with hard real time contraints
  - complexity primarily driven by nature of problem
  - control of complexity beyond designer's control
  - must concentrate on minimizing subsequent complexities

Aspects of Complexity (3)

- system design complexity
  - decomposition introduces new opportunities for complexity
  - how work is divided up into modules during system design heavily affects complexity and producability
  - designer can implement some functionality in a module or defer the implementation to another module
    - deferring functionality decreases the local complexity but increases the structural (intermodule) complexity
    - law of "incomplete conservation of complexity"
      - you can push complexity around but you can't get rid of it entirely
  - moreover, actual complexity of a poorly partitioned design may be much greater than the minimum possible complexity corresponding to its functionality

Aspects of Complexity (4)

- structural complexity
  - emerges from the relationships among the modules of a system
  - the most basic relationship is the call of one module to another
  - some studies suggest complexity increases as the square of the connections (fanout)
  - for a fixed total fanout, a system in which invocations are concentrated in a few modules is more complex than one in which invocations are more evenly distributed

Aspects of Complexity (5)

- data complexity
  - internal complexity of a module is a function of the amount of work it must perform
  - workload consists of data items that are input to or output from higher or parallel modules
  - data complexity based roughly on the number of data items being processed
- procedural complexity
  - complexity of the algorithm
  - related to functional complexity
    - designer limited in the level of complexity this can be reduced to
  - most widely accepted measure of procedural complexity is McCabe's cyclomatic complexity
    - simple count of the number of decisions

Maintainability

- a good design should facilitate software maintenance
- maintainability means that changes tend to be confined to localized areas of the system and are easy to make
- natural goal for design is low complexity

Complexity due to Process/People

- problem domain
  - "impedance mismatch" between application domain people and software developers
  - unstated and different assumptions
  - difficulty in capturing software requirements
  - requirements change
    - better understanding
    - impact of the computer on the system itself
    - changing environment
- development process
  - teams of people
  - communication
  - large software systems

Bringing Order to Chaos

- next lecture:
  - divide and rule
  - Parnas's information hiding principle

ELEC 372

Information Hiding

References

- Object Oriented Analysis and Design with Applications, 2nd ed., Booch, Addison Wesley, 1994
- Software Architecture in Practice, Bass, Clements, Kazman, Addison Wesley, 1998
- "On the Criteria to be Used in Decomposing Systems into Modules", D.L. Parnas, Communications of the ACM, Vol.5, No.12, Dec. 1972
- "On the Usability of OO Representations", Agarwal, De, Sinha, Tanniru, Communications of the ACM, Vol.43,No.10, Oct. 2000

- reading:
  - Chapters 1,2 of Booch.

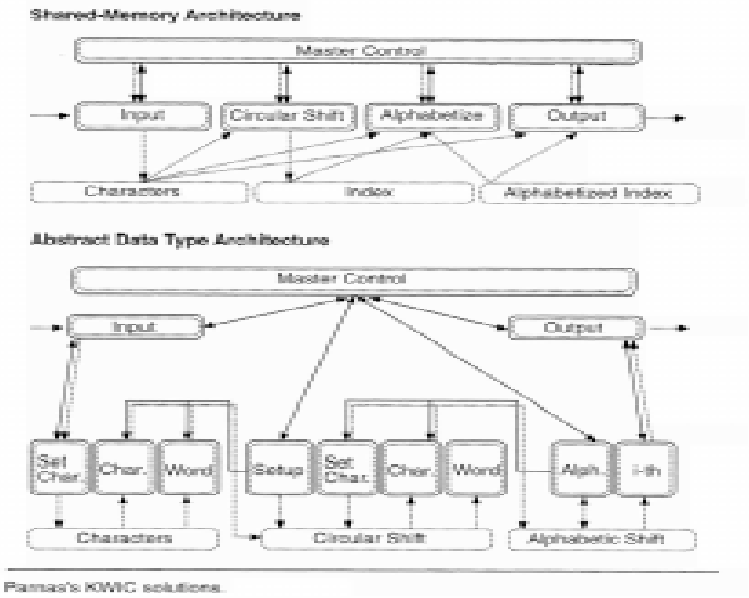
Decomposition

- divide and rule
- essential to divide software system into smaller and smaller parts
- constraint on human cognition
  - software code is there to tell a story to the human, not to the machine
- Parnas:
  - intelligent decomposition directly addresses the inherent complexity of software by forcing a division of a system's space

Example - Description

- KWIK system
  - takes a set of text lines as input
  - produces all circular shifts of words in line
  - alphabetizes results
- used to create index that is quickly searchable

- example:
  - input:
    - An Introduction to Software Architecture
    - Key Word in Context
- output
  - An Introduction to Software Architecture
  - Architecture an Introduction to Software
  - Context Key Word in
  - in Context Key Word
  - Introduction to Software Architecture an
  - Key Word in Context
  - Software Architecture an Introduction to
  - to Software Architecture an Introduction
  - Word in Context Key



Algorithmic versus Object Oriented Decomposition (1)

- "In the functional [algorithmic] paradigm, function and procedure are primary, data are only secondary. Functions and related data are either conceived of as independent, or data are associated with or attached to the functional components. ---- The antithesis of this viewpoint is the object-oriented paradigm. In this paradigm, data are considered primary and procedures are secondary; functions are associated with related data."

Algorithmic versus Object Oriented Decomposition (2)

- both views are important
  - algorithmic view highlights ordering of events
  - OO view highlights agents that cause action or subjects that are acted upon
- cannot construct a complex system in both ways simultaneously
  - decompose either by algorithm or by objects
- OO proponents
  - find great value in building models that are focused on things we find in the problem space
  - identify hierarchies (patterns of behaviour)
  - use of abstraction (also in algorithmic design)
  - use of encapsulation (also in algorithmic design)
- which decomposition you use is very dependent on the problem being solved

Algorithmic versus Object Oriented Decomposition (3)

- experiment comparing OO and process (algorithmic) models of a payroll system and a vehicle registration system
  - used object diagrams and method descriptions for OO design model
  - used data flow diagrams and data dictionaries for process oriented (PO) model
  - subjects were undergraduate students with some PO background and little or no industry experience
  - trained in OO modelling concepts before experiment
  - PO modelling concepts reviewed
  - each student was given either a PO model or an OO model for one system
    - had to answer 8 questions
    - some on structure, some on behaviour and some on both

Algorithmic versus Object Oriented Decomposition (4)

- conclusions from experiment
  - OO representations do not appear to be universally more usable or less usable than PO models
  - experienced PO modellers do not perceive usability of OO models to be any different than usability of PO models
  - inexperienced developers find it easier to apply OO representations to OO type tasks than to apply OO representation to PO type tasks
- Are OO representations better for all types of tasks?
  - serious consideration should be given to the nature of tasks in the domain
- UML may provide needed flexibility to capture multiple perspectives

Information Hiding

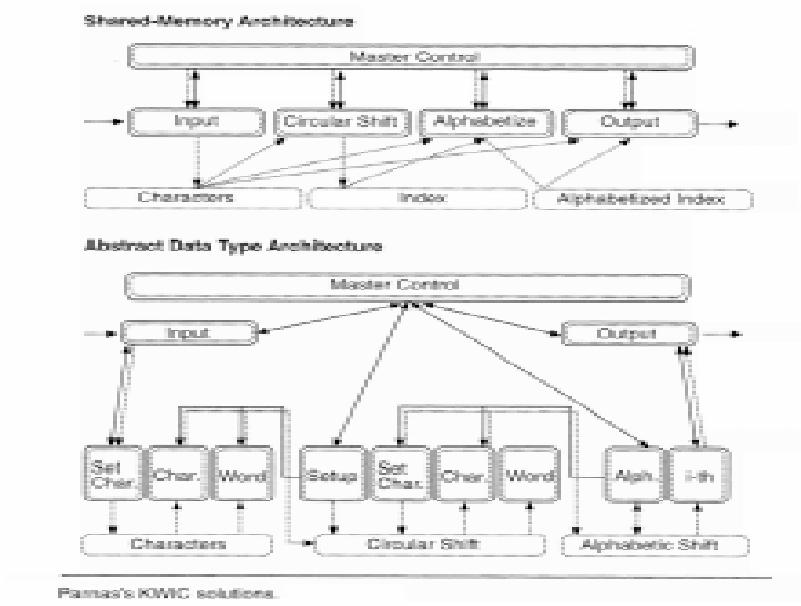
- system modularity as defined by 1970
  - each task forms a separate, distinct program module
  - inputs and outputs (interface) well defined
  - integrity of module can be tested independently
  - scope of detailed error searching can be limited
  - system maintained in modular fashion
- nothing said about the criteria to be used in dividing system into modules

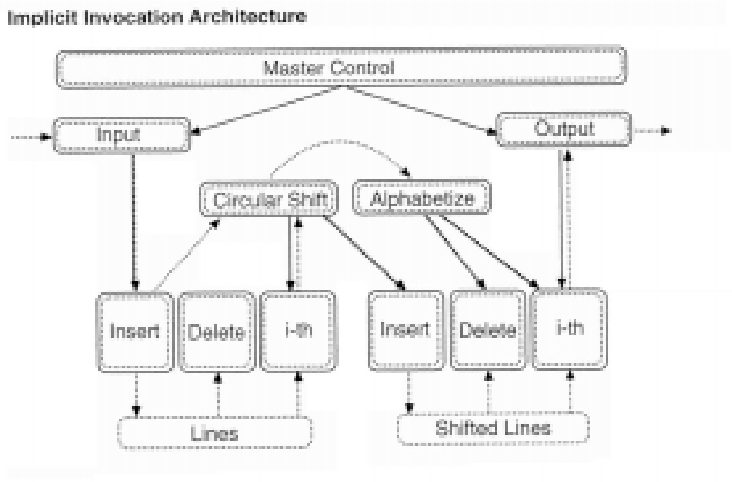
- Parnas:
  - consider difficult design decisions or decisions likely to change
  - design module to hide such decisions from other modules
  - modules do not correspond to steps in processing

Example - Description

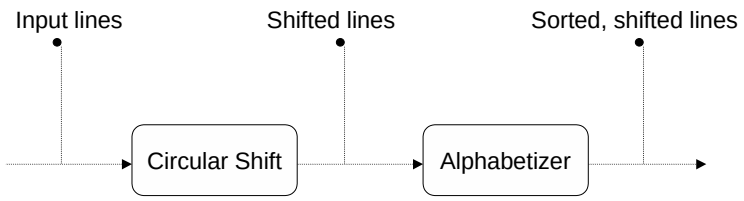
- KWIK system
  - takes a set of text lines as input
  - produces all circular shifts of words in line
  - alphabetizes results
- used to create index that is quickly searchable

- example:
  - input:
    - An Introduction to Software Architecture
    - Key Word in Context
  - output
    - An Introduction to Software Architecture
    - Architecture an Introduction to Software
    - Context Key Word in
    - in Context Key Word
    - Introduction to Software Architecture an
    - Key Word in Context
    - Software Architecture an Introduction to
    - to Software Architecture an Introduction
    - Word in Context Key





Pipe-and Filter Architecture



Rankings of KWIK Architectures

	Shared Data	Abstract Datatype	Implicit Invoc.	Pipe and Filter
Change in algorithm	-	-	+	+
Change in data structure	-	+	-	-
change in function	-	-	+	+
performance	+	+	-	-
reuse	-	+	-	+