CSC 408F/CSC2105F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC 408F/CSC2105F in the Fall term 2004/2005 at the University of Toronto.

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

©Kersti Wain-Bantin, 2004
©Yijun Yu, 2004

What is Software Engineering

• The science and art of building LARGE Software Systems
  On time
  On budget
  With Acceptable Performance
  With Correct Operation

• LARGE means:
  Many people, team not individual effort
  Many $s spent on design and implementation
  Over 75,000 lines of source code
  Lifetime measured in years
  Continuing modification and maintenance

• Software costs dominate hardware costs

Survey results

• Q2. Do you remember the LOC of the software?
  A $\leq 1000 : 7 (15\%)
  B $< 10,000 : 25 (54\%)
  C $< 100,000 : 3 (6\%)
  D $> 100,000 : 6 (14\%)
  E Don't know LOC : 5 (11\%)

• Q3. How many people were involved in the dev. team?
  A 1: 12 (26\%)
  B 2-5: 27 (37\%)
  C other: 7 (15\%)
  C no answer: 10 (22\%)

• Q4. How long did it take to complete?
  A 1 day: 0 (0\%)
  B 1 month: 12 (17\%)
  C 1 year: 11 (24\%)
  C other: 23 (50\%)
  D no answer: 4 (9\%)

• Q6. Which programming language was used?
  A C/C++: 20 (43\%)
  B Java: 18 (41\%)
  C other: 10 (21\%)
What Makes Large Software Different?

- **Scale**: Precludes total comprehension
- **Complexity**: Number of functions, modules, paths
- **Team Effort**: Continuously changing body of programmers
- **Communication**: Distribution of specifications and documentation
- **Continuing Change**: During design & implementation
- **Lifetime**: Measured in years or decades
- **Imprecise goals**: Conflicting or ambiguous, changing

Issues in Software Engineering

- Major concern is the construction of large programs.
- Central theme is mastering complexity.
- Software evolves over its lifetime.
- The efficiency of software development is of crucial importance
- Regular cooperation between people is an essential and unavoidable part of large software development.
- Software has to support its users effectively.
- Software Engineering is a field in which members of one culture (designers, programmers) create artifacts on behalf of members of another culture (end users).

The Ideal Goals of Software Engineering

- **Product Quality**: To produce software that is correct and has high quality in terms of user satisfaction.
- **Productivity**
  - To produce software with a minimum of effort.
  - To produce software at the lowest possible cost.
  - To produce software in the least possible time.
- **Profitability**: To maximize the profitability of the software production effort.
- **Maintenability**: To produce software that can be maintained with a minimum of effort.

In practice, none of these ideal goals is ever completely achievable. The challenge of Software Engineering is to see how close we can get to achieving these goals. The art of software engineering is achieving the best balance among these goals for a particular project.

Goodness (Quality) Goals Conflict

- All goodness attributes cost $s to achieve
- Interaction between attributes
  - High efficiency may degrade maintainability, reliability
  - More complex User Interface may degrade efficiency, maintainability, and reliability
  - Better documentation may divert effort from efficiency and reliability
- Software Engineering management has to trade-off satisfying goodness goals
- Software Development is (usually) done with a relatively inelastic upper bound on resources expended.

There Ain’t No Such Thing As A Free Lunch
Need Different Approaches for Developing Large Software

- Need formal management of software production process
- Formal & detailed statement of requirements, specification and design
- Much more attention to modularity and interfaces
  *Must be separable* into manageable pieces
- Need configuration management and version control
- More emphasis of rigorous and thorough testing
- Need to plan for long term maintenance and modification
- Need much more documentation, internal and external

"A typical commercial software project involves creating more than 20 kinds of paper documents on such items as requirements and functional, logic, and data specifications. For civilian projects, at least 100 English words are produced for every source code statement in the software. For military software, about 400 words are produced for every source code statement. Many new software professionals are surprised when they spend more time producing words than code."

*Capers Jones, Gaps in programming education, IEEE Computer, April 1995 v.28 n.4, pg. 71

---

Laws of Software Evolution

1. **Law of Continuing Change** A system that is being used undergoes continuous change until it is judged more cost effective to restructure the system of replace it by a completely new system.

2. **Law of Increasing Complexity** A program that is changed becomes less and less structured (*entropy increases*) and thus becomes more complex. One has to invest extra effort in order to avoid increasing complexity.

3. **Law of Program Evolution** The growth rate of global system attributes may seem locally stochastic, but is in fact self-regulating with statistically determinable trends.

4. **Law of Invariant Work Rate** The global progress in software development projects is statistically invariant.


---

5. **Law of Incremental Growth Limit** A system develops a characteristic growth increment. When this increment is exceeded, problems concerning quality and usage will result.

---

**Header restructuring project**

![Graph of header restructuring project](image)
**Significance of Lehman & Belady’s Work**

- Unless proactive steps are taken
  - the maintainability of a software system will decrease over time
  - the complexity of a system will increase over time
  - disorder in the system will increase over time
- Change is inevitable, we must learn to accommodate it.
- The amount of maintenance that can be done in a given period of time cannot be (on average) significantly increased by the addition of more resources.
- Attempting to make too large a change in a system will lead to lower quality software.

**Large Software Development**

- The requirements, specification and design of a software system are written by a team of developers.
  A hierarchical approach is often used, with sub-systems being developed by different teams.
- Individual modules (source files) are written, debugged, tested and documented by small teams of programmers.
- The modules in a system exist in some arbitrary client-server configuration. Some form of interface documents the connection between each server and its clients.
- All of the modules in a system evolve over time, due to enhancements and the correction of errors.
- The cost and time to compile an entire system is sufficiently large that separate compilation and independent subsystem development are essential.
- Specific compiled version of individual modules are linked together to form an instance of the complete software system.

**Software Factories (Product-line Families)**

- Many software producing organizations do not produce individual software systems, they produce product families
  - the same product implemented for different hardware platforms.
  - the same product implemented for different operating system environments.
  - similar products with different levels of functionality and price.
  - similar products heavily customized to the environment of each customer.
- Cost minimization considerations imply that the development and implementation of a product family must be integrated
  - Use (and reuse) common software modules
  - Use common development tools and testing
  - Use common documentation
- The coordination and communication required to build product families makes the development of the individual products much more complicated.

**Software Development Process**
Major Software Production Tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Analysis</td>
<td>Analyze software system requirements in detail</td>
</tr>
<tr>
<td>Specification</td>
<td>Develop a detailed specification for the software</td>
</tr>
<tr>
<td>Design</td>
<td>Develop detailed design for the software</td>
</tr>
<tr>
<td>Data structures, software architecture procedural detail, interfaces</td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>Transform design into one or more programming language(s)</td>
</tr>
<tr>
<td>Testing</td>
<td>Test internal operation of the system</td>
</tr>
<tr>
<td></td>
<td>Test externally visible operations &amp; performance</td>
</tr>
<tr>
<td></td>
<td>Perform Software Quality Assurance on the system</td>
</tr>
<tr>
<td>Release</td>
<td>Package and deliver software to users</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Error correction and enhancement after system has been put into use</td>
</tr>
</tbody>
</table>

Software Development Phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Documents</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Planning</td>
<td>Project Plan</td>
<td></td>
</tr>
<tr>
<td>Requirements Analysis</td>
<td>System Requirements</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>System Specification</td>
<td>10%</td>
</tr>
<tr>
<td>Design</td>
<td>System Architecture Description</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>Source Code</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Internal documentation</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Test Plan</td>
<td>45%</td>
</tr>
<tr>
<td></td>
<td>Test case suite, test results log</td>
<td></td>
</tr>
</tbody>
</table>

Allocation of Effort

- A typical project follows the 40-20-40 rule:
  - 40% on requirements, specification, architecture, design
  - 20% on coding, debugging
  - 40% on testing
- Boem’s successful projects followed the 60-15-25 rule:
  - 60% on requirements, specification, architecture, design
  - 15% on coding, debugging
  - 20% on testing
- After a software system is put into use, long term maintenance and modification effort is typically 50% ... 75% of the initial effort.

Requirements Analysis

- A software project will never be successful (i.e. on time, on budget) if the developers do not have a clear idea of what the software should be. The requirements should provide this clear idea.
- Requirements analysis should focus on what the client needs not what the client wants.
- Requirements are one means of communication
  - Between the client and the developer.
  - Between marketing and development.
  - Within the development organization.
- Requirements documents should be organized to accommodate change.
  - Each requirement should be independent.
  - Requirements should be individually modifiable. i.e. without requiring changes in other requirements.
- Changes in requirements should be managed and controlled.
Why Requirements are Important

- Significant percentage of all project failure and/or errors in system are due to errors in requirements analysis.
- The further an error propagates the more it costs to repair.
  - $1 during requirements definition
  - $5 during software design
  - $10 during implementation
  - $20 during unit testing
  - $200 after delivery of the system
- A large percentage of all software project failures can be related to a failure to get the requirements right.

Good Software Design Principles

- Abstraction
- Modularity
  - Keep It Simple Stupid
  - Don’t Repeat Yourself
- Information Hiding
- Functional Independence
  - High Cohesion
  - Low Coupling
- Traceability

Abstraction

- Abstractions are generalizations or simplifications of the design that suppress detail
- Being able to view (think about) the design at different levels of abstraction facilitates understanding
- Pseudo code or graphical techniques are often used to represent an abstract design
- Abstraction is related to the design process refinement
- Develop procedural and data abstractions at different levels in the design
- Need to be careful that high level abstractions don’t preclude consideration of design alternatives at a lower level

Modularity

- Modularity is a key factor in successful design
- Modularity assists in:
  - Understanding
  - Development
  - Testing
  - Portability
  - Maintenance
- Always develop software using the module concept even when implementing in a programming language without a module construct (e.g. C)
- KISS principle should be applied to module interfaces
- Selecting an appropriate modularization for a system is one of the most important tasks in software design
Information Hiding

- Keep knowledge of representation of data as localized as possible. i.e. hide information about the representation of data from all those who do not have an unavoidable need to know.
- Modules in Modula-3, Classes in C++, Packages in Ada are mechanisms for hiding information.
- Strict information hiding takes a lot of effort. Need control every where that a data item might get created or modified.
- Information hiding is essential if modules are to be used as plug-replaceable software components.
- C++ has a very comprehensive set of mechanisms for information hiding and controlled information disclosure.

Functional Independence

- Want modules that implement a single function related to the system requirements.
- Design by successive refinement helps achieve this goal.
- Design to minimize interactions between modules.
- Design to keep module interfaces simple.
  - Minimize number of exported items.
  - Exported functions and procedures should have small number parameters.
- Functional independence helps with:
  - Communication between module developers and users.
  - Maintenance using modules as units of change.
  - Concurrent development and testing of modules.
  - Tracking source code back to specification and requirements.
  - Containing the effects of software errors.

Cohesion

- Cohesion is a measure of the self containedness of a module, i.e. to what degree does it perform a single task.
- High cohesion is desirable as it maximizes separability and functional independence.
- Levels of cohesion for operations exported by a module:
  - Coincidental: together by chance, i.e. unrelated.
  - Logical: perform logically related operations.
  - Temporal: perform operations related by time.
  - Procedural: operations related by processing requirements.
  - Communicational: operations communicate via common data.
  - Sequential: operations must be performed in specified order.
  - Functional: operations implement a single specific function.

Coupling

- Coupling is a measure of the degree of interconnection among modules in a software system.
- Low coupling is desirable as it maximizes separability and functional independence.
- Levels of coupling among modules in a system:
  - None: No direct coupling.
  - Data: Modules communicate via simple data parameters.
  - Stamp: Modules communicate via complex data parameters.
  - Control: Modules communicate via control flags.
  - External: Modules communicate via external media (e.g. files).
  - Common: Modules communicate via shared global data.
  - Content: Module uses data maintained by another module or one module branches into another module.
Advantages of Low Coupling and High Cohesion

- Communication between developers becomes simpler.
- Correctness proofs are easier to derive.
- It is less like that changes in one module will affect other modules making maintenance easier.
- Increases the reusability of modules.
- Increases the comprehensibility of modules.
  This makes maintenance easier and facilitates training new programmers to work on the software.
- Empirically low coupling and high cohesion leads to fewer errors in the software.

Traceability

- An important tool in managing the development of a software system.
- An aid to understanding the how and why in a complicated system design.
- Forward traceability
  Requirements → Specification → Design → Code → Test Cases → Product
- Backward traceability
  Requirements ← Specification ← Design ← Code ← Test Cases ← Product
- Traceability aids in achieving correctness.
  Traceability facilitates long term system maintenance.

Software Maintenance Activities

<table>
<thead>
<tr>
<th>Type</th>
<th>Purpose</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>corrective</td>
<td>Repair of errors</td>
<td>21%</td>
</tr>
<tr>
<td>adaptive</td>
<td>Adapt to environment changes</td>
<td>25%</td>
</tr>
<tr>
<td>perfective</td>
<td>Adapt to changes in requirements</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>Improve performance or user interface</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add new functionality</td>
<td></td>
</tr>
<tr>
<td>preventative</td>
<td>Increase future maintainability</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td>Updating documentation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Improving system modularity/structure</td>
<td></td>
</tr>
</tbody>
</table>

Course Project

- Reengineering a large software system
- Work in teams
- Three Phase Project
  A. Study a large legacy software and specify new requirements, 15%
  B. Partial implementation, 15%
  C. Swap software and complete implementation, 20%
The OmniEditor Project

- Text editors are friends of programmers, they have advanced editing features
  A. C/C++: VIM, Emacs, NEdit, etc.
  B. Java: Eclipse, jEdit, etc.
  C. Commercial: MS Word, Notepad, UltraEdit, etc.

- To edit as a group, we need groupwares, they have limited editing features
  A. Messenger: MSN, Yahoo!, AOL, etc.
  B. Bulletin Board: BBS, firebird, etc.
  C. Bloggers: Blogger, Weblog, etc.

- How to combine the heterogenous editors into one groupware?

Why choose this project for CSC408H?

- It tackles a real problem: until now, there is not much work on bridging the editors. One notable solution is VIM/NetBeans or VIM/VisioStudio combinations. However, the VIM/Eclipse combination is rather weak. Can you have a general and strong solution?
- It involves large software systems: nowadays text editors are no longer toy systems. They are tools that have been engineered for years.
- It requires new technologies: Web services, middleware
- It is tractable within our course: by divide and conquer

Size of editors

Table 1: LOC of popular open-source editors

<table>
<thead>
<tr>
<th>editor</th>
<th>version</th>
<th>lines of code</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIM</td>
<td>6.3</td>
<td>260,623</td>
<td>C</td>
</tr>
<tr>
<td>Emacs</td>
<td>21.3</td>
<td>306,746</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>263,178</td>
<td>Lisp</td>
</tr>
<tr>
<td>XEmacs</td>
<td>21.5.4</td>
<td>353,045</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>140,830</td>
<td>Lisp</td>
</tr>
<tr>
<td>jEdit</td>
<td>4.1</td>
<td>114,117</td>
<td>Java</td>
</tr>
<tr>
<td>Eclipse</td>
<td>3.0</td>
<td>1,585,873</td>
<td>Java</td>
</tr>
</tbody>
</table>

Phase A. Understanding legacy software and Requirements Analysis

- Pick 4 partners to form a project team
- Describe the people in the team, allocation of tasks
- Pick one editor (VIM or Eclipse) to study, document the discussions why and how you choose the editor
- Discover why we shall use Web Services to bridge heterogeneous editors
- Describe the architecture of the OmniEditor project
- Plan the project as an iterative process and proposes a test plan
Phase A. Project Requirements

- You must have a reasonable solution: Finally at least any two different editors can synchronize the contents with each other through a pair of editing operations.
- You don’t reinvent the wheel: try to reuse existing editor as much as possible
- You don’t implement everything: spirit of divide and conquer between teams
- You don’t all implement the simplest features
- You must be responsible for the product quality

\[
\text{phaseMark} = \text{projectMark} \times \left( 1 + \max \left\{ 0, 0.02 \times n\text{Selectors} - 0.001 \times n\text{NetBugs} \right\} \right)
\]

Deliverables after Phase A by Sept. 30

- Documents
  - What is the architecture of the legacy editor and Why
  - What is the architecture of the OmniEditor and Why
  - What is the project plan of the OmniEditor and Why
  - What is the risk of project failure and How to prevent
  - What is your test plan
- Project team
  - Who are the team members and Who is the team leader
  - Skills and Preferences
  - Tasks and allocations
  - Team synchronizations and coordinations

Phase B. Develop the OmniEditor Web Service

- Web service is good for interoperability and reuse
- Understand the three elements of Web services
  - SOAP – Simple Object Access Protocol
  - WSDL – Web Service Description Language
  - UDDI – Universal Description, Discovering and Integration
- Define the WSDL description of your web service
- Implement the operations as a web service
- Install a web server for web services
  - Java: Apache/Tomcat/Axis or webMethods glue
  - C/C++: Apache/Axis-C or gSOAP
- Deploy the web service on the web server

Deliverables for Phase B by Nov 4

- Source code
  - WSDL and server URL
  - Web service implementation, optional
  - Unit Test cases
- Documentation
  - User’s Guide
  - Installation and Deployment
  - Traceability: Requirements, Design and Implementation
  - Bug Report and Maintenance plan

*The instructor will assign a unique port number to the web service*
Phase C. Integrate the OmniEditorWS with an Editor

- Invoke the web service when it is required by the editor to perform cooperative tasks
- Discover bugs in the web service and communicate with the developers
- Help them to fix the bug and also fix the bug that others found in your deliverable in Phase B

Deliverables for Phase C by Dec 2

- Source code
  - An OmniEditor editor client implementation
  - Integration test scenarios
- Documentation
  - User’s Guide
  - Installation and Deployment
  - Bug Report and Fixed Bugs
  - Measured software metrics during software evolution, optional
- Note that the Instructor and Tutors may conduct a system test to find outstanding bugs in your deliverables. So try not to be too ambitious in number of features to keep bugs in bay.

Tutorials

1. Web Services: What is web service, Examples, HowTo’s
2. VIM editor: Architecture, Features, How to integrate VIM with IDE?
3. Eclipse: Plugin Architecture, Plugin development, CVS, Text Editor
4. Unit Tests: JUnit, CppUnit, How to test a web service
5. Measure Software Quality: Complexities, Reliability
6. Team Presentation 1 (by end of Phase B)
7. Web services registry (UDDI): Publish/Subscribe, Repository, Advertise/Evaluation
8. Quality of Service: Customer Satisfaction and Defects Fixing
9. Team Presentation 2 (by end of Phase C)
10. Team Presentation 2 (by end of Phase C)
11. Q/A