CS 390 – Lecture 2
The Scope of Software Engineering (2)

Requirements, Analysis, and Design Aspects

- Obvious that the earlier we detect and correct a fault, the less it costs to fix
- Question is why is this the case?

Requirements, Analysis, and Design Aspects (2)

- To correct a fault early in the life cycle
  - Usually just a document needs to be changed
- To correct a fault late in the life cycle
  - Change the code and the documentation
  - Test the change itself
  - Perform regression testing
  - Reinstall the product on the client’s computer(s)

Requirements, Analysis, and Design Aspects (3)

- Between 60 and 70% of all faults in large-scale products are requirements, analysis, and design faults
- Example: Jet Propulsion Laboratory inspections
  - 1.9 faults per page of specifications
  - 0.9 per page of design
  - 0.3 per page of code

Requirements, Analysis, and Design Aspects (4)

- Cost of fixing a fault at each phase of the classical life cycle – note the log scale (Figure 1.5)

Problems 1.6, 1.7

- Seven months after delivery, a fault is detected in the software of a product that analyses DNA. The cost of fixing the fault is $16,700. The cause of the fault is an ambiguous sentence in the specification document.
  - Approximately how much would it have cost to have corrected the fault during the analysis phase?
  - During the implementation phase?

Conclusion – Do the work up front!

- It is vital to improve our requirements, analysis, and design techniques
  - To find faults as early as possible
  - To reduce the overall number of faults (i.e., the overall cost)
  - Note this includes reducing postdelivery corrective maintenance, too.
Activities that could be phases

- Planning: must know how to get the project done
- Testing: must check that software does what it is suppose to do
- Documentation: must be able to explain the project

Why planning is not a phase

- We cannot plan at the beginning of the project—we do not yet know exactly what is to be built
- Preliminary planning of the requirements and analysis phases at the start of the project
- The software project management plan (SPMP) is drawn up when the specs have been signed off by the client
- Management needs to monitor the SPMP throughout the rest of the project

Conclusion – Planning

- Planning activities are carried out throughout the life cycle
- Therefore, there is no separate planning phase

Why testing is not a phase

- Verification
  - Testing at the end of each phase is too late to be cost effective
- Validation
  - Testing at the end of the project is far too late to be cost effective

Conclusion – Testing

- Continual testing activities must be carried out throughout the life cycle
- This testing is the responsibility of
  - Every software professional
  - The software quality assurance group
- Therefore, there is no separate testing phase (though client may do separate acceptance testing at the end)

Why writing documentation is not a phase

- Documentation must always be current
  - Key individuals may leave before the documentation is complete
  - We cannot perform a phase without having the documentation of the previous phase
  - We cannot test without documentation
  - We cannot maintain without documentation
Conclusion - Documentation

- It is far too late to document after development and before delivery
- Documentation activities must be performed in parallel with all other development and maintenance activities
- Therefore, there is no separate documentation phase

Team Development Aspects

- Hardware is cheap
  - We can build products that are too large to be written by one person in the available time
- Software is built by teams
  - Interfacing problems between modules
  - Communication problems among team members

Object-Oriented Paradigm

- The structured paradigm was successful initially
  - It started to fail with larger products (> 50,000 LOC)
- Postdelivery maintenance problems (today, 70 to 80% of total effort)
- Reason: Structured methods are
  - Action oriented (e.g., finite state machines, data flow diagrams); or
  - Data oriented (e.g., entity-relationship diagrams, Jackson’s method); But not both

Object-Oriented Paradigm (2)

- In OOP, both data and actions are of equal importance
- Object:
  - A software component that incorporates both data and the actions that are performed on that data
- Example:
  - Bank account
    - Data: account balance
    - Actions: deposit, withdraw, determine balance

Structured versus Object-Oriented Paradigm (Figure 1.7)

- Information hiding
- Responsibility-driven design

Information Hiding

- In the classical version
  - All the modules have details of the implementation of account_balance
- In the object-oriented version
  - The solid line around accountBalance denotes that outside the object there is no knowledge of how accountBalance is implemented
### Strengths of the OOP

- With information hiding, postdelivery maintenance is safer
  - The chances of a regression fault are reduced
- Development is easier
  - Objects generally have physical counterparts
  - This simplifies modeling (a key aspect of the object-oriented paradigm)

### Strengths of the OOP (2)

- Well-designed objects are independent units
  - Everything that relates to the real-world item being modeled is in the corresponding object — *encapsulation*
  - Communication is by sending *messages*

### Strengths of the OOP (3)

- A classical product conceptually consists of a single unit (although it is implemented as a set of modules)
  - The object-oriented paradigm reduces complexity because the product generally consists of independent units
- The object-oriented paradigm promotes reuse
  - Objects are independent entities

### Responsibility-Driven Design

- Also called *design by contract*
- Enhances independence of objects
- Send flowers to your mother in Chicago
  - Call 1-800-flowers
  - Where is 1-800-flowers?
  - Which Chicago florist does the delivery?
  - Information hiding
  - Send a message to a method [action] of an object without knowing the internal structure of the object

### Classical Phases vs Object-Oriented Workflows

<table>
<thead>
<tr>
<th>Classical Paradigm</th>
<th>Object-Oriented Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements phase</td>
<td>1. Requirements workflow</td>
</tr>
<tr>
<td>2. Analysis phase</td>
<td>2’. OO analysis workflow</td>
</tr>
<tr>
<td>3. Design phase</td>
<td>3’. OO design workflow</td>
</tr>
<tr>
<td>4. Implementation phase</td>
<td>4’. OO implementation workflow</td>
</tr>
<tr>
<td>5. Postdelivery maintenance</td>
<td>5. Postdelivery maintenance</td>
</tr>
<tr>
<td>6. Retirement</td>
<td>6. Retirement</td>
</tr>
</tbody>
</table>

### Analysis/Design “Hump”

- Structured paradigm:
  - There is a jolt between analysis (what) and design (how)
- Object-oriented paradigm:
  - Objects enter from the very beginning
  - Extract as part of OO *analysis workflow*
Analysis/Design “Hump” (2)
- In the classical paradigm
  - Classical analysis
    - Determine what has to be done
  - Design
    - Determine how to do it
    - Architectural design — determine the modules
    - Detailed design — design each module

Removing the “Hump”
- In the object-oriented paradigm
  - Object-oriented analysis
    - Determine what has to be done
  - Object-oriented design
    - Determine how to do it
    - Design the objects

Removing the “Hump” (2)
- Modules (objects) are introduced in the OO analysis workflow
  - This ensures a smooth transition from the analysis workflow to the design workflow
- The objects are then coded during the implementation workflow
  - Again, the transition is smooth

The Object-Oriented Paradigm in Perspective
- The object-oriented paradigm must be used correctly
  - All paradigms are easy to misuse
- When used correctly, the object-oriented paradigm can solve some (but not all) of the problems of the classical paradigm

The Object-Oriented Paradigm in Perspective (2)
- The object-oriented paradigm has problems of its own
  - Generally harder to learn and use correctly
- The object-oriented paradigm is the best alternative available today
  - However, it is certain to be superceded by something better in the future

Ethical Issues
- Developers and maintainers need to be
  - Hard working
  - Intelligent
  - Sensible
  - Up to date and, above all,
  - Ethical
- IEEE-CS ACM Software Engineering Code of Ethics and Professional Practice
  http://www.acm.org/about/se-code/