Lecture 3: Life Cycles and Design Methods

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2002

Goals for this Lecture

• Review traditional software engineering life cycles
• Introduce the notion of an object-oriented design method
  – Hint: its another name for “life cycle”

Background

• In Software Engineering:
  “Process is King”
  – We want our activities to be coordinated and planned, e.g. “engineered”
  – The reason? A high quality process should increase our ability to create a high quality product

Use of Process

• Car Assembly
  – An assembly line is a process for producing cars.
  – A significant amount of work goes into not just designing a car but into designing the process used to build that car

• Software Engineering
  – The same principles can be applied to developing a software system
Key Difference

• There is a key difference between software engineering and car assembly, however.
• In car assembly, design time for the car is “short”, the majority of the work lies in manufacturing
  – In software engineering, we face the reverse situation, creating new copies of a software system is trivial, it’s the design that is hard
  – Thus, there will be significant differences in the processes used to develop software

Software Life Cycle

• A series of steps that organizes the development of a software product
• Duration can be from days to years
• Consists of
  – people!
  – overall process
  – intermediate products
  – stages of the process

Phases of a Software Life Cycle

• Standard Phases
  – Requirements Analysis & Specification
  – Design
  – Implementation and Integration
  – Operation and Maintenance
  – Change in Requirements
  – Testing throughout!
• Phases promote manageability and provide organization

Requirements Analysis and Specification

• Problem Definition —> Requirements Specification
  – determine exactly what client wants and identify constraints
  – develop a contract with client
  – Specify the product’s task explicitly
• Difficulties
  – client asks for wrong product
  – client is computer/software illiterate
  – specifications may be ambiguous, inconsistent, incomplete
• Validation
  – extensive reviews to check that requirements satisfy client needs
  – look for ambiguity, consistency, incompleteness
  – check for feasibility, testability
  – develop system/acceptance test plan
Design

- Requirements Specification → Design
  - develop architectural design (system structure)
    - decompose software into modules with module interfaces
  - develop detailed design (module specifications)
    - select algorithms and data structures
    - maintain record of design decisions
- Difficulties
  - miscommunication between module designers
  - design may be inconsistent, incomplete, ambiguous
- Verification
  - extensive design reviews (inspections) to determine that design conforms to requirements
  - check module interactions
  - develop integration test plan

Implementation and Integration

- Design → Implementation
  - implement modules and verify they meet their specifications
  - combine modules according to architectural design
- Difficulties
  - module interaction errors
  - order of integration has a critical influence on product quality
- Verification and Testing
  - code reviews to determine that implementation conforms to requirements and design
  - develop unit/module test plan: focus on individual module functionality
  - develop integration test plan: focus on module interfaces
  - develop system test plan: focus on requirements and determine whether product as a whole functions correctly

Operation and Maintenance

- Operation → Change
  - maintain software after (and during) user operation
  - determine whether product as a whole still functions correctly
- Difficulties
  - design not extensible
  - lack of up-to-date documentation
  - personnel turnover
- Verification and Testing
  - review to determine that change is made correctly and all documentation updated
  - test to determine that change is correctly implemented
  - test to determine that no inadvertent changes were made to compromise system functionality

Code-and-Fix (Not a Life Cycle!)

- Build First Version
- Modify until Client is satisfied
- Operations Mode
- Retirement
Discussion of Code-and-Fix

- Useful for “hacking”
- Problems become apparent in any serious coding effort
  - No process for things like versioning, configuration management, testing, etc.
  - Difficult to coordinate activities of multiple programmers
  - Non-technical users cannot explain how the program should work
  - Programmers do not know or understand user needs

Discussion of Waterfall

- Proposed in early 70s
- Widely used (even today)
- Advantages
  - Measurable Progress
  - Experience applying steps in past projects can be used in estimating duration of steps in future projects
  - Produces software artifacts that can be re-used in other projects

Waterfall Model

- Proposed in early 70s
- Widely used (even today)
- Advantages
  - Measurable Progress
  - Experience applying steps in past projects can be used in estimating duration of steps in future projects
  - Produces software artifacts that can be re-used in other projects

Waterfall, continued

- The original waterfall model had disadvantages because it disallowed iteration
  - Inflexibility
  - Monolithic
  - Estimation is difficult
  - Requirements change over time
  - Maintenance not handled well
- These are problems with other life cycle models as well
- The “waterfall with feedback” model was created in response
  - Our slides show this model
**Incremental**

For each build:
- Perform detailed design, implement, Test, Deliver.

**Arch. Design**
- Verify

**Operations**
- Retirement

**Discussion of Incremental Model**

- Used by Microsoft
  - Programs are built everyday by the build manager
  - If a programmer checks in code that “breaks the build” they become the new build manager!
  - Iterations are classified according to features
    - e.g. features 1 and 2 are being worked on in this iteration, features 3 and 4 are next

**Summary**

- Life cycles make software development
  - predictable
  - repeatable
  - measurable
  - efficient

- High-quality processes should lead to high-quality products
  - at least it improves the odds of producing good software

**Survey of OOA&D Methods**

- Generalization
  - Taken from "SE: A Practitioner’s approach, 4th ed." by Roger S. Pressman, McGraw-Hill, 1997

- The Booch Method
- The Jacobson Method
- The Rambaugh Method
- The Unified Software Process
OO Methods In general...

- Obtain customer requirements for the OO System
  - Identify scenarios or use cases
  - Build a requirements model
- Select classes and objects using basic requirements
- Identify attributes and operations for each object
- Define structures and hierarchies that organize classes
- Build an object-relationship model
- Build an object-behavior model
- Review the OO analysis model against use cases

Detailed comparisons

- What follows is a barebones description of each method, detailed comparisons can be found in:
  - For related links: <http://www.ultranet.com/~lebrun/Steven/Computer/Programming/Object-Oriented.html>

Background on OO Methods

- An OO Method should cover and include
  - requirements and business process modeling
  - a lightweight, customizable process framework
  - project management
  - component architecture
  - system specification
    - use cases, UML, architecture, etc.
  - component design and decomposition
  - testing throughout the life cycle
  - QA and configuration management
  - Process Patterns

Process Patterns

- A pattern in the form of
  - Whenever your goal is A and your current situation is B then try doing C
    - (but be aware of prerequisite P, risk R, side-effect S, time-scale T, etc.)
**The Booch Method**

- Identify classes and objects
  - Propose candidate objects
  - Conduct behavior analysis
  - Identify relevant scenarios
  - Define attributes and operations for each class
- Identify the semantics of classes and objects
  - Select scenarios and analyze
  - Assign responsibility to achieve desired behavior
  - Partition responsibilities to balance behavior
  - Select an object and enumerate its roles and responsibilities
  - Define operations to satisfy the responsibilities

**Booch, continued**

- Identify relationships among classes and objects
  - Define dependencies that exist between objects
  - Describe the role of each participating object
  - Validate by walking through scenarios
- Conduct a series of refinements
  - Produce appropriate diagrams for the work conducted above
  - Define class hierarchies as appropriate
  - Perform clustering based on class commonality
- Implement classes and objects
  - In analysis and design, this means specify everything!

**The Jacobson Method**

- Object-Oriented Software Engineering
  - Primarily distinguished by the use-case
  - Simplified model of Objectory
    - Objectory evolved into the Rational Unified Software Development Process
  - For more information on this Objectory precursor, see

**Jacobson, continued**

- Identify the users of the system and their overall responsibilities
- Build a requirements model
  - Define the actors and their responsibilities
  - Identify use cases for each actor
  - Prepare initial view of system objects and relationships
  - Review model using use cases as scenarios to determine validity
- Continued on next slide
Jacobson, continued

• Build analysis model
  – Identify interface objects using actor-interaction information
  – Create structural views of interface objects
  – Represent object behavior
  – Isolate subsystems and models for each
  – Review the model using use cases as scenarios to determine validity

The Rambaugh Method

• Object Modeling Technique (OMT)
• Analysis activity creates three models
  – Object model
    • Objects, classes, hierarchies, and relationships
  – Dynamic model
    • object and system behavior
  – Functional model
    • High-level Data-Flow Diagram

Rambaugh, continued

• Develop a statement of scope for the problem
• Build an object model
  – Identify classes that are relevant for the problem
  – Define attributes and associations
  – Define object links
  – Organize object classes using inheritance
• Develop a dynamic model
  – Prepare scenarios
  – Define events and develop an event trace for each scenario
  – Construct an event flow diagram and a state diagram
  – Review behavior for consistency and completeness

Rambaugh, continued

• Construct a functional model for the system
  – Identify inputs and outputs
  – Use data flow diagrams to represent flow transformations
  – Develop a processing specification for each process in the DFD
  – Specify constraints and optimization criteria
• Iterate!
Rational Unified Process: Overview

Inception
• High-level planning for the project
• Determine the project’s scope
• If necessary
  – Determine business case for the project
    • Estimate cost and projected revenue

Elaboration
• Develop requirements and initial design
• Develop Plan for Construction phase
• Risk-driven approach
  – Requirements Risks
  – Technological Risks
  – Skills Risks
  – Political Risks

Construction
1 2 3

Transition

Taken from UML Distilled, Chapter 2
See also, page 482-485 of Graham’s OO Methods book

Requirements Risks
• Is the project technically feasible?
• Is the budget sufficient?
• Is the timeline sufficient?
• Has the user really specified the desired system?
• Do the developers understand the domain well enough?
Dealing with Requirements Risks

• Construct models to record Domain and/or Design knowledge
  – Domain model (vocabulary)
  – Use Cases (discussed next week)
  – Design model
    • Class diagrams
    • Activity diagrams
• Prototype construction

Dealing with Requirements Risks, continued.

• Begin by learning about the domain
  – Record and define jargon
  – Talk with domain experts
    • Oftentimes end-users!
• Next construct Use cases
  – What are the required external functions of the system?
  – Iterative process; Use Cases can be added as they are discovered

Dealing with Requirements Risks, continued.

• Finally, construct Design model
  – Class diagrams identify key domain concepts and their high-level relationships
  – Activity diagrams highlight the domain’s work practices
    • A major task here is identifying parallelism that can be exploited later
• Be sure to consolidate iterations into a final consistent model

Dealing with Requirements Risks, continued.

• Build prototypes
  – Used only to help understand requirements
  – Throw them all out!
    • Do not be tied to an implementation too early
    • Make use of rapid prototyping tools
      – 4th Generation Programming Languages
      – Scripting and/or Interpreted environments
      – UI Builders
• Be prepared to educate the client as to the purpose of the prototype
Technology Risks

- Are you tied to a particular technology?
- Do you “own” that technology?
- Do you understand how different technologies interact?
- Techniques
  - Prototypes!
  - Class diagrams, package diagrams

Skill Risks

- Do the members of the project team have the necessary skills and background to tackle the project?
- If not
  - Training, Consulting, Mentoring and Hiring new people are available options!

Political Risks

- How well does the proposed project mesh with corporate culture?
  - Consider the attempt to use Lotus Notes at Arthur Anderson
    - Lotus Notes attempts to promote collaboration
    - Arthur Anderson consultants compete with each other!
  - Consider e-mail: any employee can ignore the org chart and mail the CEO!

Political Risks, continued

- Will the project directly compete with another business unit?
- Will it be at odds with some higher level manager’s business plan?
- Any of these can kill a project…
- Examples from students?
Reference

• Lotus Notes vs. Arthur Anderson

• If you are interested you can borrow my copy of the CSCW’92 proceedings to make a copy

Ending Elaboration

• Baseline architecture Constructed
  – List of Use cases (with estimates)
  – Domain Model
  – Technology Platform

• AND
  – Risks identified
  – Plan constructed
    • Use cases assigned to iterations

Construction

• Each iteration produces a software product that implements the assigned Use cases
  – Additional analysis and design may be necessary as the implementation details get addressed for the first time

• Extensive testing should be performed and the product should be released to (some subset of) the client for early feedback

Transition

• Final phase before release 1.0
• Optimizations can now be performed
  – Optimizing too early may result in the wrong part of the system being optimized
  – Largest boosts in performance come from replacing non-scalable algorithms or mitigating bottlenecks