Applying Active Review

CIS 423/523

Overview

• UNC students have a set of artifacts (requirements, architecture)
• Task: UO (and other) students tasked with quality assurance review
• Goals
  – Understand the rationale for this QA approach
  – Understand what must be reviewed
  – Understand how to conduct an effective review
• Focus on application now explanations later
Progress and Schedule

- Requirements
  - Use cases, models
  - Review/validation
- Design
  - Architectural views
  - Interface specification
  - Reviews
- Detailed design
- Implementation
- Unit testing
- Integration testing

What’s the plan?

- Define system requirements
- Decompose the system into work assignments (modules)
- Fully specify each work assignment (Interface Spec.)
- Review the specifications for quality

How do we do this so as to maintain control over a distributed development
Why this approach?

• Recall the role that quality assurance plays in maintaining control of a software development
• Consider: assuming a distributed development, what can we learn better by having a different team conduct a review

Code Structure and DSD

• Problems of coordination and control are affected by the way the code is structured (decomposed into parts and the relationships between the parts)
  – Problem of distributing work: want to be able to have different sites develop code concurrently without increasing communication overhead
  – Problem of incremental development: need to coordinate development so all the pieces are developed in the right order for each increment
  – Problem of run-time dependencies: timing dependencies at run require synchronization
• Focus on systematic approaches to architectural design
Problem of Distributing Work

- What kinds of issues arise when we want to have different sites develop different parts of the system?
  - Need to divide the system into parts
  - Have parts worked on concurrently by distributed teams
  - Have a working system when the parts are put together for testing or integration (each increment)

Implications of Code Structure

- In practice, there are always dependencies between system parts
  - Data dependencies (component $C_1$ uses data produced by $C_2$)
  - Functional dependencies (component $C_1$ performs a function needed by $C_2$)
  - Timing dependencies (C1 must wait for C2)
  - Resource dependencies ($C_1$ and $C_2$ share hardware), etc.
- Coupling
  - We characterize the set of dependencies between components $C_1$ and $C_2$ as their *interface*
  - Components with many dependencies are characterized as *tightly coupled*
- Component interfaces == human interfaces!
  - Where components depend on each other, developers must communicate to build components that interact correctly
  - Higher coupling => more communication to get it right
Software Architecture

The Module Structure

Working Definition

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

From Software Architecture in Practice, Bass, Clements, Kazman
WBS Goals

Goal: distribution of work on components requiring least inter-team communication
Q: Properties should the components have?

In practice this means...

- Want to decompose into components that
  - Are team work assignments
  - Implement all requirements
- Inter-component dependencies are minimized
  - Expose only what is necessary
  - Encapsulates properties likely to change
- Services provided by each component are well defined
Module Structure

- Architectural model: called the Module Structure
- Components
  - Called modules
  - Leaf modules are work assignments
- Relations
  - “submodule-of”
  - The set of submodules of any module X partition X’s functionality
  - Constrained to be acyclic tree (hierarchy)
- Module interfaces
  - Modules at the leaves of the tree provide the methods implementing the system’s functionality
  - Module interfaces specify everything one must know to use the module’s services correctly
  - Modules encapsulate properties other modules should not depend on

Module Hierarchy

- Problem
- Interface Encapsulated
- Interface Encapsulated
- Interface Encapsulated
- Interface Encapsulated

Leaf Modules = Work assignments

Submodule-of relation
Interface Specification

Module Interface Specifications
- Documents all assumptions users can make about the module's externally visible behavior (of leaf modules)
  - Access programs, events, types, undesired events
  - Design issues, assumptions
- Document purpose(s)
  - Provide all the information needed to write a module's programs or use the programs on a module's interface (programmer's guide, user's guide)
  - Specify required behavior by fully specifying behavior of the module's access programs
  - Define any constraints
  - Define any assumptions
  - Record design decisions

Why these properties?

Module Implementer
- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don't change the interface

Module User
- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

Key idea: the abstract interface specification defines a contract between a module's developer and its users that allows each to proceed independently
What is an abstract interface?

• An abstract interface defines the set of assumptions that one module can make about another
• While detailed, an abstract interface specification does not describe the implementation
  – Does not specify algorithms, private data, or data structures
  – Preserves the module’s secrets
• One-to-many: one abstract module specification allows many possible implementations
  – Developer is free to use any implementation that is consistent with the interface
  – Developer is free to change the implementation

A method for specifying abstract interfaces

1. Define services provided and services needed (assumptions)
2. Decide on syntax and semantics for accessing services
3. In parallel
   – Define access method syntax (signatures)
   – Define access method effects
   – Define terms and local data types
   – Define visible module states
   – Record design decisions
   – Record implementation notes
4. Define test cases and use them to verify access methods
   – Cover testing effects, parameters, exceptions
   – Design test cases before implementing module
Quick Example: FWS Data Banker

Exercise: Peer Review

- Team: review the DataBanker interface specification provided (~15 minutes)
- Note any defects found for discussion
Active Reviews

Improved Peer Review Method

Qualities of Effective Review

• Ensures adequate coverage of artifact in breadth and depth
• Reviewers review only issues on which they have expertise
• Individual responsibilities are clear and fulfilling them is evidence of a job well done.
  – Review process is active: i.e., performing the review produces visible output
  – Review process focuses on finding specific kinds of errors.
• Limit meetings to focused groups and purposes requiring common understanding or synergy
  – Permit detailed discussion of issues
  – Expose where assumptions differ
Active Review Process

Goal: Make the reviewer(s) think hard about what they are reviewing
1. Identify several types of reviews targeting different types of errors
2. Identify appropriate classes of reviewers for each type of review
3. Assign reviews to achieve coverage
   – Each applicable type of review is applied to each part of the specification

Active Reviews (2)

4. Design review questionnaires (key difference)
   – Define questions that the review must answer by using the specification
   – Target questions to bring out key issues
   – Phrase questions to require “active” answers (not just “yes”)
5. Review consists using the artifact to fill out questionnaire
6. Review process: overview, review, meet
   – One-on-one or small, group
   – Discuss issues identified in review
   – Track and respond to issues
Examples

- Active review asks a qualified reviewer to check a specific part of a work product for specific kinds of defects by answering specific questions, e.g.,
  - Ask a designer to check the functional completeness by showing the calls sequences sufficient to implement a set of use cases
  - Ask a systems analyst to check the ability to create required subsets by showing which modules would use which
  - For each access program in the module, what exceptions that can occur?
  - Ask a technical writer to check the SRS for grammatical errors
- Can be applied to any kind of artifact from requirements to code

Conventional vs. Active Questions

- Goal: Make the reviewer(s) think hard about what they are reviewing*
  - Define questions that the review must answer by using the specification
  - Target questions to bring out key issues
  - Phrase questions to require “active” answers (not just “yes”)

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional Design Review Questions</td>
<td>Active Design Review Questions*</td>
</tr>
<tr>
<td>Are exceptions defined for every program?</td>
<td>For each access program in the module, what exceptions that can occur?</td>
</tr>
<tr>
<td>Are the right exceptions defined for every program?</td>
<td>What is the the range or set of legal values?</td>
</tr>
<tr>
<td>Are the data types defined?</td>
<td>For each data type, what are • an expression for a literal value of that data type; • a declaration statement to declare a variable for that type; • the greatest and least values in the range of that data type?</td>
</tr>
<tr>
<td>Are the programs sufficient?</td>
<td>Write a short pseudo-code program that uses the design to accomplish (some defined task).</td>
</tr>
</tbody>
</table>
Applying Use Cases

• Use cases or scenarios can be effectively used in active review
• Apply requirements scenarios to verify design against requirements
  – “Show the sequence of program calls that would implement use case C”
  – “Which modules would have to change to add feature F (a likely change)?”
• Conversely, can check properties ask the reviewer to construct scenarios
  – “What sequence of calls would result in an exception E?”

Active Review of UNC Artifacts

• Come up with three or more active review questions and at least two test cases
• For each, briefly explain:
  – What kinds of properties (e.g., functional behavior, exceptional, completeness, etc.) the question seeks to evaluate.
  – What aspects of the specification must be used to answer the questions
• For the test case: an example of specific inputs and expected outputs
Assignments

• Team: Create active review questions and send to me by Wednesday evening
• Contact your counterparts at UNC and Jilin and introduce yourselves
• Collaborate to create a team page on our Assembla site
  – You are not required to use real names or other identifying information
• Metrics: keep track of the tasks done and time spent (will send metrics form)

Questions?