From Module Decomposition to Interface Specification

Software Architecture Part 2

Outline

- Lecture: Module interface design and specification
- Lab exercise: develop interface specification
- Lecture: Active design reviews
- Lab exercise: review a module specification

DSD Architectural Design Goals

- Limit the necessity for communication by limiting the dependencies (coupling) between components
- Goals: divide the system into work assignments such that
  - Each part can be assigned to a different team and developed independently
  - Parts can be independently verified
  - It is possible to change the implementation details of one module without affecting other modules
  - Only properties of the system that are unlikely to need to be change are used by other modules
  - Role of each part in the overall system is clear (and when together, implement the requirements)
Module Structure Accomplishments

- What have we accomplished in creating the module structure?
- Divided the system into parts (modules) such that
  - Each module is a work assignment for a person or small team
  - Each part can be developed independently
  - Every system function is allocated to some module
- Informally described each module
  - Services: services that the module implements that other modules can use
  - Secrets: implementation decisions that other modules should not depend on
Excerpts From The FWS Module Guide

2. Device Interface Modules
The device interface modules consist of those programs that need to be changed if the input from hardware devices to FWSs or the output to hardware devices from FWSs change. The
secret of the device interface modules is the interfaces between FWSs and the devices that
produce the inputs and that use its output.

2.1. Wind Sensor Device Driver
Service
Provide access to the wind speed sensors. There may be a
submodule
for each sensor

Note
The module hides the boundary between the FWS domain and the sensors domain. The
boundary is formed by an abstract interface that is a standard for all wind speed sensors.
Programs in this module use the abstract interface to read the values from the sensors.

Need for Precise Interface Specifications
- But, informal description is not enough to write the software
- To support independent, distributed development, need a precise interface specification
  - For the implementer: describes the requirements the module must satisfy
  - For other developers: defines everything you need to know to use the module’s services correctly
  - For tester: specifies the range of acceptable behaviors for unit test
- The interface specification defines a contract between the module’s developers and its users

A Simple Stack Module
- A simple integer stack
  - The interface specifies what a programmer needs to know to use the stack correctly, e.g.
    - push push int on stack top
    - pop remove top element
    - peek get value of top element
  - The secrets (encapsulated) any details that might change from one implementation to another
    - Data structures, algorithms
    - Details of class/object structure
- Is this enough to define a contract?
What is an abstract interface?

- An abstract interface defines the set of assumptions that one module can make about another.
- By “abstract” we mean that there is a one-to-many relation.
- 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.

Goals for Module Interface Specifications

- Clearly documents the behavior of the module.
  - Reduces time & knowledge required to adopt module.
- Clearly documents the interfaces used by the module.
  - Aids in creating stubs, mock interfaces and integration test scripts.
- Improves the ability to isolate errors quickly.
- Supports backwards compatibility.
- Defines implementers work assignment.
  - Interface specification is essentially a contract for the developer that specifies the implementer’s task and the assumptions that users can make.
- Enables straight-forward mapping between use case requirements and methods.
  - Reduces effort required for requirements traceability.

Abstraction and Interface Specs

- We strongly prefer an interface spec that is abstract in the sense that it hides implementation details.
  - Allows the implementation to change without affecting other modules.
- BUT … you can’t really design a good interface without thinking about possible implementation(s).
A method for constructing abstract interfaces

- Define services provided and services needed (assumptions)
- Decide on syntax and semantics for accessing services
- In parallel
  - Define access method effects
  - Define terms and local data types
  - Record design decisions
  - Record implementation notes
- Define test cases and use them to verify access methods
  - Cover testing effects, parameters, exceptions
  - Test both positive and error use cases
- Can use Javadoc or similar

An FWS Example: The Data Banker Interface Specification

Define services provided

<table>
<thead>
<tr>
<th>Service</th>
<th>Provided By</th>
<th>Performed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize the set of stored sensor readings</td>
<td>initialize</td>
<td>T,C1,L,C2,TC1,T,C4,TC3</td>
</tr>
<tr>
<td>2. Record a new sensor reading, maintaining only the necessary history, and retrieve the current sensor reading history, keeping reads and writes synchronized</td>
<td>read, write</td>
<td>T,C1,L,C3,TC2,TC3,TC4,TC5</td>
</tr>
</tbody>
</table>

Access Methods

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Parameter</th>
<th>Parameter Type</th>
<th>Description</th>
<th>Exception</th>
<th>Map to services</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>sensor</td>
<td>String</td>
<td>Type of sensor</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>startG</td>
<td>sensor</td>
<td>String</td>
<td>Type of sensor</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>sensor</td>
<td>String</td>
<td>Type of sensor</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>readG</td>
<td>sensor</td>
<td>String</td>
<td>Type of sensor</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>record</td>
<td>sensor</td>
<td>String</td>
<td>Sensor reading value</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>recordG</td>
<td>sensor</td>
<td>String</td>
<td>Sensor reading value</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>recordData</td>
<td>sensor</td>
<td>String</td>
<td>Vector of elements of type SensorReading</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
An FWS Example: The Data Banker Interface Specification

Decide on syntax and semantics for accessing services

**Access Method Semantics**
- Values returned
- State changes
- Legal call sequences
- Synchronization and other call interactions

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initn</td>
<td>Initiates a queue of elements of type sensorType of length HistoryLength. For each element of sensorType with twice values of nil.</td>
</tr>
<tr>
<td>write</td>
<td>Sums the sensorReading to the back of the queue and removes the oldest sensor reading value from the front of the queue.</td>
</tr>
<tr>
<td>read</td>
<td>Returns the vector of sensor readings of type sensorType. With the most recent values of the sensor readings. The vector is of length (HistoryLength * number of sensors) of that type.</td>
</tr>
</tbody>
</table>

**Synchronization:** This module supports concurrent access to the read and write methods. Where any read or write call occur concurrently, the read and write statements are atomic operations (i.e., the read will not occur after the sequence read write or the sequence write read).

---

An FWS Example: The Data Banker Interface Specification

**Local Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>HistoryLength</td>
<td>The number of sequential, past sensor values kept</td>
</tr>
</tbody>
</table>

**SensorReading**

A triple \((r, v, w)\), where \(r\) is of type SensorReading resolution, \(v\) is of type SensorReading value, and \(w\) is of type SensorReading weight.

---

An FWS Example: The Data Banker Interface Specification

Define test cases and use them to verify access method

**Example**

1.1.1 T1

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>InputType/Value</th>
<th>ExpectedResults</th>
<th>Service</th>
<th>Preamble</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>initn</td>
<td>sensorType</td>
<td>Type of sensor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>read</td>
<td>sensorType</td>
<td>Return vector of null values</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
An FWS Example: The Data Banker

Record design decisions

Interface Design Issues

1. Should we let the user read an empty vector of sensor readings after initialization, or just throw an exception?
   A1. Yes. An empty vector should be treated just as any other.
   A2. No. There are no valid values in an empty vector that can be averaged, so we should let the user know that the vector is empty by throwing the exception.
   Resolution: Yes. We will check values during testing during testing to save space and CPU cycles.

Using Javadoc

```java
public class DataBanker {
    private class DataBanker
    private static String INTERFACE_SPECIFICATION

    The Data Banker provides synchronized storage for sensor readings.

    Interface Specification
    1. Indicate the set of stored sensor readings.
    2. Store a new sensor reading, maintaining only the necessary.
    3. Reduce to filtered values for a specified sensor: reading history, keeping
    4. Calculate.
    5. Sort.

    Synchroic/Access: Synchronized access to readwrite methods.

    The SensorReading is the type of sensor reading.

    Field Summary
    Duration: String
    Description: The number of wind speed readings that are stored.

    Constructor Summary
    Constructor 1

    Method Summary
    // The Data Banker provides synchronized storage for sensor readings.
    void iifc(){}
    Use: Start
    Description: First

    void initialize()
    Use: Start
    Description: Initialize the set of stored sensor readings.

    void add(String)
    Use: Add
    Description: Store a new sensor reading, maintaining only the necessary.

    void reduce()
    Use: Filter
    Description: Reduce to filtered values for a specified sensor: reading history, keeping

    void calculate()
    Use: Calculate
    Description: Calculate.

    void sort()
    Use: Sort
    Description: Sort.

    // Synchronization/Access: Synchronized access to readwrite methods.
    void readwrite()
    Use: Readwrite

    // The SensorReading is the type of sensor reading.

    public static final String SENSORREADER = "SensorReading";
    public class DataBanker {
        public static final int SENSORREADER_LENGTH = 4;

        void initialize(String sensorType, int sensorIndex)
            SENSORREADER_LENGTH = sensorIndex;
            SENSORREADER = SENSORREADER + sensorType + SENSORREADER_LENGTH;

        DataBanker() {
            SENSORREADER_LENGTH = 4;
        }
    }
```
Benefits Good Module Specs

- Enables development of complex projects:
  - Support partitioning system into separable modules
  - Complements incremental development approaches
- Improves quality of software deliverables:
  - Clearly defines what will be implemented
  - Errors are found earlier
  - Error Detection is easier
- Defines clear acceptance criteria
- Defines expected behavior of module
- Clarifies what will be easy to change, what will be hard to change
- Clearly identifies work assignments

Interface Design

Considerations in Interface Design

Design Principles

Role of Information Hiding and Abstraction

Module Interface Design Goals

- General design goals addressed by module interface design
  - Carry forward architectural goals: Independent work assignments, maintainability, understandability, testability, etc.
  - Addressed by two module interface design goals
- Control dependencies
  - Encapsulate anything other modules should not depend on
  - Hide design decisions and requirements that might change (data structures, algorithms, assumptions)
- Provide services
  - Provide all the capabilities needed by the module’s users
  - Provide only what is needed (complexity)
  - Provide problem appropriate abstraction (useful services and states)
  - Provide reusable abstractions
- Specific goals need to be captured (e.g., in the module guide and interface design documents)
1. Controlling Dependencies

- Addressed using the principle of information hiding
- IH: design principle of limiting dependencies between components by hiding information other components should not depend on
- When thinking about what to put on the interface
  - Design the module interface to reveal only those design decisions considered unlikely to change
  - Required functionality allocated to the module and considered likely to change must be encapsulated
  - Each data structure is used in only one module
  - Data structures may be accessed by programs within the module but not by those outside the module
  - Any other program must access internal data by calling access programs on the interface
- Consistent with good OOD principles

2. Provide Services

- Interface provides the capabilities of the module to other modules in the system, addressed by:
- Abstraction: interface design principle of providing only essential information and suppressing unnecessary detail

Abstraction

- Two primary uses
- Reduce Complexity
  - Goal: manage complexity by reducing the amount of information that must be considered at one time
  - Approach: Separate information important to the problem at hand from that which is not
  - Abstraction suppresses or hides “irrelevant detail”
  - Examples: stacks, queues, abstract device
- Model the problem domain
  - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
  - Approach: Provide components that make it easier to model a class of problems
    - May be quite general (e.g., type real, type float)
    - May be very problem specific (e.g., class automobile, book object)
Example: Car Object

- What are the abstractions?
  - Purpose of each?
- What information is hidden?

---

Which Principle to Use

- Use abstraction when the issue is what should be on the interface (form and content)
- Use information hiding when the issue is what information should not be on the interface (visible or accessible)

---

Summary

- Every module has an abstract interface that provides a way for other modules to use its secret without knowing how the secret is implemented
- An interface is the set of assumptions that the users of a module can make about the module
- The interface specification for a module is a contract between the users of the module and the implementers of a module
- An abstract interface specification specifies both syntax and semantics for the interface
- There is a systematic process for developing interface specifications
Exercise

FWS Modules Specifications
- Create module specifications for new FWS modules added to the Module Guide
- Decide which part of the team will be responsible for each of the new modules
  - Advise: divide along the lines of producer/consumer
- Each team should write at least one module specification
  - Should includes services/data that is used by the other team if possible
  - Put on a wiki page
  - Will be reviewed by remote teammates
- Use simplified template

Examples
- Examples/InterfaceSpecsSimplifiedFWS.docx
Quality Assurance Using Reviews

Peer Reviews
Inefficiencies in Peer Review
Active Reviews

Importance of Early Defect Detection

1. The majority of software errors are introduced early in software development
2. The later that software errors are detected, the more costly they are to correct

<table>
<thead>
<tr>
<th>Phase in which error detected</th>
<th>$1 error</th>
<th>$100 error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acceptance testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architectural analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detailed design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quality is Cumulative

- Are the requirements valid?
- Complete? Consistent? Implementable?
- Testable?
- Does the design satisfy requirements?
- Are all functional capabilities included?
- Are qualities addressed (performance, maintainability, usability, etc.)?
- Do the modules work together to implement all the functionality?
- Are likely changes encapsulated?
- Is every module well defined?
- Implement the required functionality?
- Race conditions? Memory leaks? Buffer overflow?
Peer Reviews

- Peer Review: a process by which a software product is examined by peers of the product’s authors with the goal of finding defects
- Why do we do peer reviews?
  - Review is often the only available verification method before code exists
  - Formal peer reviews (inspections) instill some discipline in the review process
- Particularly important for distributed teams
  - Supports communication and visibility
  - Provides feedback on both quality and understanding
  - A good review shows communication is working!

Effectiveness of Peer Reviews

- Generally considered most effective manual technique for detecting defects
  - Analysis of 12,000 development projects showed defect detection rate of 60-65% for formal inspection 30% for testing
  - Bell-Northern found 1 hour code inspecting saves 2 to 4 hours code testing
  - Effect is magnified in earlier inspections (e.g., 30 times for requirements in one study)
- Means that you should be doing peer reviews, but...
  - Doesn’t mean that manual inspections cannot be improved
  - Doesn’t mean that manual inspections are the best way to check for every properties (e.g., completeness)
  - Should be one component of the overall V&V process
Peer Review Problems

- Tendency for reviews to be incomplete and shallow
- Reviewers typically swamped with information, much of it irrelevant to the review purpose
- Reviewers lack clear individual responsibility
- Effectiveness depends on reviewers to initiate actions
  - Review process requires reviewers to speak out
  - Keeping quiet gives lowest personal risk
  - Rewards of finding errors are unclear at best

Peer Review Problems (2)

- Large meeting size hampers effectiveness, increases cost
  - Makes detailed discussion difficult
  - Few present reviewers have interest/expertise on any one issue
  - Wastes everyone else's time and energy
- No way to cross-check unstated assumptions

Qualities of Effective Review

- Ensures adequate coverage of artifact in breadth and depth
- Reviewers review only issues on which they have expertise
- Review process is active: i.e., performing the review produces visible output (risk in in doing nothing)
- Individual responsibilities are clear and fulfilling them is evidence of a job well done.
Qualities of Effective Review (2)

- 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 Reviews

- Goal: Make the reviewer(s) think hard about what they are reviewing
  1) Identify several types of review each targeting a different type of error (e.g., UI behavior, consistency between safety assertions and functions).
  2) Identify appropriate classes of reviewers for each type of review (specialists, potential users, methodology experts)
  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 of filling out questionnaires defining
   - Section to be reviewed
   - Properties the review should check
   - Questions the reviewer must answer
6) Review process: overview, review, meet
   - One-on-one or small, similar group
   - Focus on discussion of issues identified in review
   - Purpose of discussion is understanding of the issue (not necessarily agreement)
Examples

- In practice: an 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
  - As a developer to check the data validity of a module's specification by showing what the output would be for in-range and out-of-range values
  - Ask a technical writer to check the SRS for grammatical errors
- Can be applied to any kind of artifact from requirements to code as long as the reviewer is forced to use the document

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>Conventional Design Review Questions</th>
<th>Active Design Review Questions*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are exceptions defined for every program?</td>
<td>For each access program in the module, what exceptions can occur?</td>
</tr>
<tr>
<td>Are the right exceptions defined for every program?</td>
<td>What is 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>

Individual Review Process

- Role: [Specialist] Reviewer
  - Where [Specialist] denotes area expertise like problem domain requirements, architectural design, module design, etc.
- Artifacts
  - Input: artifact under review (e.g., SRS, module spec), prepared review questions
  - Output: answers to questions, defects, issues
- Activities
  - Overview of artifact
  - Individual review – use artifact to answer questions
  - Discussion of results
    - One-on-one or small group, for DSD can be on-line
    - Focus on discussion of issues identified in review
    - Purpose of discussion is understanding of the issue (not necessarily agreement)
Role of 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?”

Why Active Reviews Work

- Focuses reviewer’s skills and energies where they have skills and where those skills are needed
  - Questionnaire allows reviewers to concentrate on one concern at a time
  - No one wastes time on parts of the document where there is little possibility of return.
- Largest part of review process (filling out questionnaires) is conducted independently and in parallel
- Reviewers must participate actively but need not risk speaking out in large meetings
- Downside: much more work for V&V (but can be productively pursued in parallel with document creation)

Simple Integer Stack

```java
//Module Interface Spec
void push(int newItem); //push an integer on the stack
int pop(); //Remove the top int from the stack
boolean isEmpty(); //Returns true if the stack is empty
```

- What kinds of questions would one ask?
- Suppose we are handling order numbers LIFO and different subsystems handle different groups of order numbers in parallel?
Summary

- Need to do reviews to find defects
- Critical for distributed teams
  - Provides another communication pathway
  - Makes level of understanding visible
- Active reviews are more efficient and effective but may take more effort

Assignment
- Choose one artifact (e.g., a module spec)
- Review within your own team
- Request a review from PKU side (I'll ask them to do the same)

Exercise

Active Review of Module Spec

- Write a set of active review questions that will require your remote team to use your module specification
- Exchange specification and review questions
- Answer the questions:
  - Was the review effective?
    - Did the reviewer have to show understanding of the specification?
    - Was the reviewer able to answer the questions?
  - Was the module specification complete and sufficient?
    - Did it provide all the information needed by the reviewer?
    - Were any problems detected?
End