From Module Decomposition to Interface Specification

Software Architecture Part 2

Outline

• Module decomposition with example
• Module interface 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)

Information-Hiding Structure

• Architectural model: called the “information hiding” 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
  – The set of methods and their behavior define the module interfaces
    • The interface methods provide the only access to a module’s internal state
    – Information encapsulated (internal) by the module are called its “secrets”
Submodule-of Relation

- To define the structure, need the relation and the rule for constructing the relation
- Relation: sub-module-of
- Rules
  - If a module consists of parts that can change independently, then decompose it into submodules
  - Don’t stop until each module contains only things likely to change together
  - Anything that other modules should not depend on become secrets of the module (e.g., implementation details)
  - If the module has an interface, only things not likely to change can be part of the interface

Module Hierarchy

An Example: The FWS Module Structure

Floating Weather Stations (FWS)

Floating weather stations (FWS) are buoys that float at sea and that are equipped with sensors to monitor weather conditions. Each FWS has an on-board computer that maintains a history of recent weather data. At regular intervals the buoy transmits the weather data using a radio transmitter.

The initial prototype for the buoy will measure the wind speed in knots. The buoys will use four small wind speed sensors (anemometers): two high-resolution sensors and two, less expensive, low-resolution sensors.

Accuracy is software enhanced by computing a weighted-average of the sensor readings over time. Each sensor is read once every second with the readings averaged over four readings before being transmitted. The calculated wind speed is transmitted every two seconds.

Over the course of development and in coming versions, we anticipate that the hardware and software will be routinely upgraded including adding additional types of sensors (e.g. wave height, water temperature, wind direction, air temperature). A system that can be rapidly revised to accommodate new features is required.
FWS Likely Changes

Likely changes

**Behavior**

C1. The formula used for computing wind speed from the sensor readings may vary. In particular, the weights used for the high resolution and low resolution sensors may vary, and the number of readings of each sensor used (the history of the sensor) may vary.

C2. The format of the messages that an FWS sends may vary.

C3. The transmission period of messages from the FWS may vary.

C4. The rate at which sensors are scanned may vary.

**Devices**

C4. The number and types of wind speed sensors on an FWS may vary.

C5. The resolution of the wind speed sensors may vary.

C6. The wind speed sensor hardware on an FWS may vary.

C7. The transmitter hardware on an FWS may vary.

C8. The method used by sensors to indicate their reliability may vary.

Classifying Changes

- **Three classes of change**
  - **Hardware**
    - new devices
    - new computer
  - **Required behavior**
    - new functions
    - new rules of computing values
    - new timing constraints
  - **Software decisions**
    - new ways to represent data types
    - different algorithms or data structures

Top-Level Module Decomposition

- **Device Interface (DI)**
  - Secret = properties of physical hardware
  - Encapsulates any hardware changes
- **Behavior-Hiding (BH)**
  - Secret = algorithms/data addressing requirements
  - Encapsulates requirements changes
- **Software Decision (SD)**
  - Secret = decisions by designer
  - Encapsulates internal design decisions

DI Submodules

- **Windspeed Sensor Driver**
  - Service: provides access wind speed values
  - Secrets: Anything that would change if the current wind speed sensor were replaced with another. For example, the details of data formats and how to communicate with the sensor

- **Transmitter Driver**
  - Service: transmit given data on request
  - Secrets: details of transmitter hardware
Excerpts From The FWS Module Guide (1)

1. Behavior Hiding Modules
   The behavior hiding modules include programs that need to be changed if the required outputs from a FWS and the conditions under which they are produced are changed. Its secret is when (under what conditions) to produce which outputs. Programs in the behavior hiding module use programs in the Device Interface module to produce outputs and to read inputs.

   1.1 Controller
   Service
   Provide the main program that initializes a FWS.
   Secret
   How to use services provided by other modules to start and maintain the proper operation of a FWS.

Excerpts From The FWS Module Guide (2)

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 its 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 type.
   Secret
   How to communicate with, e.g., read values from, the sensor hardware.
   Note
   This 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.
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

Abstract Interface Specifications

Defining the Contract Between Module Developers and Users

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 integer 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

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
  - Define states of the module
  - 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
  - Support automation
  - Design test cases before implementing module
- Can use Javadoc or similar

Data Banker Module

- DB sits between input sensor side and output side. Stores sensor values for later use
- Sensor devices & input processing
- Actuator devices & output processing
- Data Banker
- Write(s,r) / Read(s,v)
An FWS Example: The Data Banker Interface Specification

Define services provided

<table>
<thead>
<tr>
<th>Service</th>
<th>Provided By</th>
<th>Tested By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize the set of stored sensor readings.</td>
<td>initialize</td>
<td>TC1, TC2, TC3, TC4, TC5</td>
</tr>
<tr>
<td>2. Store 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>TC1, TC2, TC3, TC4, TC5</td>
</tr>
</tbody>
</table>

Decide on syntax and semantics for accessing services

**Access Methods**

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>sensorType1</td>
<td>String</td>
<td>Type of sensor.</td>
<td>1</td>
</tr>
<tr>
<td>write</td>
<td>sensorType1</td>
<td>SensorReading</td>
<td>Type of sensor. Sensor reading value</td>
<td>2</td>
</tr>
<tr>
<td>read0</td>
<td>sensorType1</td>
<td>Vector&lt;sensorType1&gt;</td>
<td>Type of sensor, Vector of elements of type SensorReading</td>
<td>2</td>
</tr>
</tbody>
</table>

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>

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SensorReading</td>
<td>A triple (t, v, w) where t is of type SensorReading.resolution, v is of type SensorReading.value, and w is of type SensorReading.weight</td>
</tr>
</tbody>
</table>
An FWS Example: The Data Banker Interface Specification

Define test cases and use them to verify access method

Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Input Type/Value</th>
<th>Expected Results</th>
<th>Service</th>
<th>Preamble</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize</td>
<td>sensorType</td>
<td>Type of sensor.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>read</td>
<td>sensorType</td>
<td>Returns vector of null values</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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 to save space and CPU cycles.

Using Javadoc

```java
public class DataBanker {

    private int HistoryLength;
    private float[] readings;

    public DataBanker() {
        // Default constructor
    }

    // Public methods
    public void initialize(String sensorType, int numSensors) {
        // Initialize the DataBanker for a type of sensor reading.
    }
    public void addReading(int index, float reading) {
        // Add a reading to the history.
    }
}
```

```java
// The Data Banker provides synchronized storage for sensor readings.

// Services Provided
  // Initialize
  // Read

// HistoryLength is the number of wind speed readings that are retained
public static final int HistoryLength = 4;
```
Should you use UML?

+ UML is widely used and understood
+ UML can capture signatures and some relations reasonably well
  - UML doesn’t help much with semantics
  - UML doesn’t help much with judging completeness, consistency, abstractness, or almost anything else important
+ So go ahead if you want, but don’t confuse “nice UML” with “well-designed interface spec”

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
  - Improves testability
- 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

Summary

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

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
1. Control dependencies
   - Encapsulate anything other modules should not depend on
   - Hide design decisions and requirements that might change (data structures, algorithms, assumptions)
2. 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?

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throttle: double</td>
</tr>
<tr>
<td>Mass: int</td>
</tr>
<tr>
<td>Friction: int</td>
</tr>
<tr>
<td>Speed: double</td>
</tr>
<tr>
<td>GetThrottle()</td>
</tr>
<tr>
<td>Euler()</td>
</tr>
<tr>
<td>EvaluateModel()</td>
</tr>
<tr>
<td>GetSpeed()</td>
</tr>
</tbody>
</table>

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

End