
Communicating Design

Finish up design principles
Design exercise
Design Documentation

CIS 422/522 Winter 2014 1

What are Principles?

- Principle (n): a comprehensive and fundamental rule, doctrine, or assumption
- Design Principles – rules that guide developers in making design decisions consistent with overall design goals and constraints
 - Guide the decision making process of design by helping choose between alternatives
 - Embodied in methods and techniques (e.g., for decompositions)

CIS 422/522 Winter 2014 2

Three Key Design Principles

- Most solid first
- Information hiding
- Abstraction

CIS 422/522 Winter 2014 3

Principle: Most Solid First

- View design as a sequence of decisions
 - Later decisions depend on earlier
 - Early decisions harder to change
- Most solid first: in a sequence of decisions, those that are least likely to change should be made first
- Goal: reduce rework by limiting the impact of changes
- Application: used to order a sequence of design decisions
 - Generally applicable to design decisions
 - Module decomposition – ease of change
 - Developing families – create most commonality

Information Hiding

- Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design principles that (Parnas):
 - System details that are likely to change independently are put in different modules
 - The interface of a module reveals only those aspects considered unlikely to change
 - Details other modules should not depend on are encapsulated

Abstraction

- General: disassociating from specific instances to represent what the instances have in common
 - Abstraction defines a *one-to-many relationship*
E.g., one type, many possible implementations
- Modular decomposition: 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)

CIS 422/522 Winter 2014 7

Exercise: Address Book Data Module

- Design the "model" module
 - What should be hidden?
 - What services should it provide?
 - What else does the user need to know to use the module correctly?

CIS 422/522 Winter 2014 8

Lessons on Patterns

- Patterns are often misused
- Using a pattern correctly requires understanding it
 - "Correctly" – such that the pattern's design goals are realized in your design
 - "Understanding" – you understand what the pattern is supposed to accomplish, how it works, and how to apply it in your context

CIS 422/522 Winter 2014 9

Lessons on Patterns (2)

- A pattern is a three part rule that expresses a relation between [Schmidt]:
 1. A particular problem context
 2. A set of competing forces (goals and constraints) in that context
 3. A software *configuration* that *resolves* the set of forces
 - *Configuration* == objects, interfaces, relations
 - *Resolves* == concurrently addresses the goals and constraints

CIS 422/522 Winter 2014 10

Summary

- Heuristics and patterns are guidelines
 - Do not guarantee qualities
 - Must understand how and why they work to apply effectively
- Principles are more direct – achieve qualities *by construction*
- Good design requires careful thinking
 - Which goals are we trying to achieve
 - How design decisions address those goals

CIS 422/522 Winter 2014 11

Documenting a Module Structure

Communicating Architectural Decisions

CIS 422/522 Winter 2014 12

Architecture Development Process

Building architecture to address business goals:

1. Understand the goals for the system
2. Define the quality requirements
3. Design the architecture
 1. Views: which architectural structures should we use?
 2. Documentation: how do we communicate design decisions?
 3. Design: how do we decompose the system?
4. Evaluate the architecture (is it a good design?)

Purpose and Audience

- To understand what to communicate, consider who will use it and for what purpose
 - Coders/maintainers: defines the build-to spec.
 - Where to put/find specific parts of the system (e.g., where functionality is implemented)
 - Embodies system qualities as design decisions
 - Constrains detailed design and implementation
 - Quality stakeholders
 - How the system satisfies design goals
 - Why specific design decisions were made
 - Testers: which parts should be tested to establish specific qualities

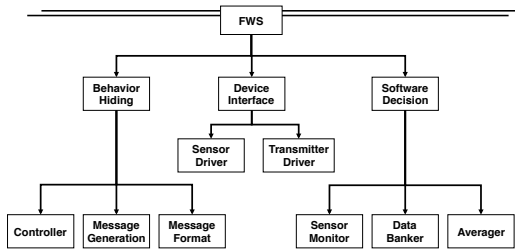
Communicating Architecture

- Provide a set of views addressing key qualities
- For each architectural view deployed
 - Which architectural structures are used (components, relations, and interfaces)
 - Which quality requirements are being addressed in the structure (why)
- Within a given structure
 - How to use/navigate the structure to find specific information
 - What design decisions are made
 - Rationale for important decisions

Example: Module Structure Documentation

- **Module Guide**
 - Documents the module structure:
 - The set of modules and the responsibility of each module in terms of the module's secret
 - The "submodule-of relationship"
 - Document purpose(s)
 - Guide for finding the module responsible for each aspect of the system behavior
 - Provides a record of design decisions (rationale)
- **Module Interface Specifications**
 - Documents all assumptions user's 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

FWS Modular Structure



Submodule-of

Module

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.

A Method for Specifying 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 visible states of the module
 - Record design decisions
- 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

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

For Your Projects

- Develop at least one architectural view
- Include rationale for the overall design
- Include any significant design decisions
- Outcome: should be able to trace from requirements to code objects

CIS 422/522 Winter 2014

22

```

/** The Data Banker provides synchronized storage for sensor readings.
** <ul>
** <li> Services Provided
** </li>
** <li> Initialize the set of stored sensor readings.
** </li>
** <li> Store a new sensor reading, maintaining only the necessary
** history, and retrieve the current sensor reading history, keeping
** reads and writes synchronized.
** </li>
** </ul>
** Synchronization: Supports concurrent access to read/write methods.
** Read or write operations on a vector of sensor readings act as atomic
** operations.
** </ul>
** Exceptions: N/A
** </ul>
** Uses: SensorReading
**/
public class DataBanker
{
    /** HistoryLength is the number of wind speed readings that are retained
    **/
    public static final int HistoryLength = 4;

    /** Initialize the DataBanker for a type of sensor reading.
    ** </ul>
    ** <li> Initializes a vector of elements of type sensorType of length
    ** HistoryLength for each sensor of sensorType with initial values of null.
    ** </li>
    ** @param sensorType The String name of the sensor type
    ** @param numSensors Number of sensors.
    **/
    public static void initialize(String sensorType, int numSensors)
    {
        Vector<SensorReading> v = new Vector<SensorReading>();
        for (int j = 0; j < HistoryLength * numSensors; j++)
            v.addElement(null);
        map.put(sensorType, v);
    }
}

```

23
