

From Module Decomposition to Interface Specification

Documenting module structure

Specifying module interfaces

Module interface design

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?)

Examples of Key Architectural Structures

- Module Structure
 - Decomposition of the system into work assignments or information hiding modules
 - Most influential design time structure
 - Modifiability, work assignments, maintainability, reusability, understandability, etc.
- Uses Structure
 - Determine which modules may use one another's services
 - Determines subsetability, ease of integration

Modularization

- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
 - Parts can be designed independently
 - Parts can be tested independently
 - Parts can be changed independently
 - Integration goes smoothly

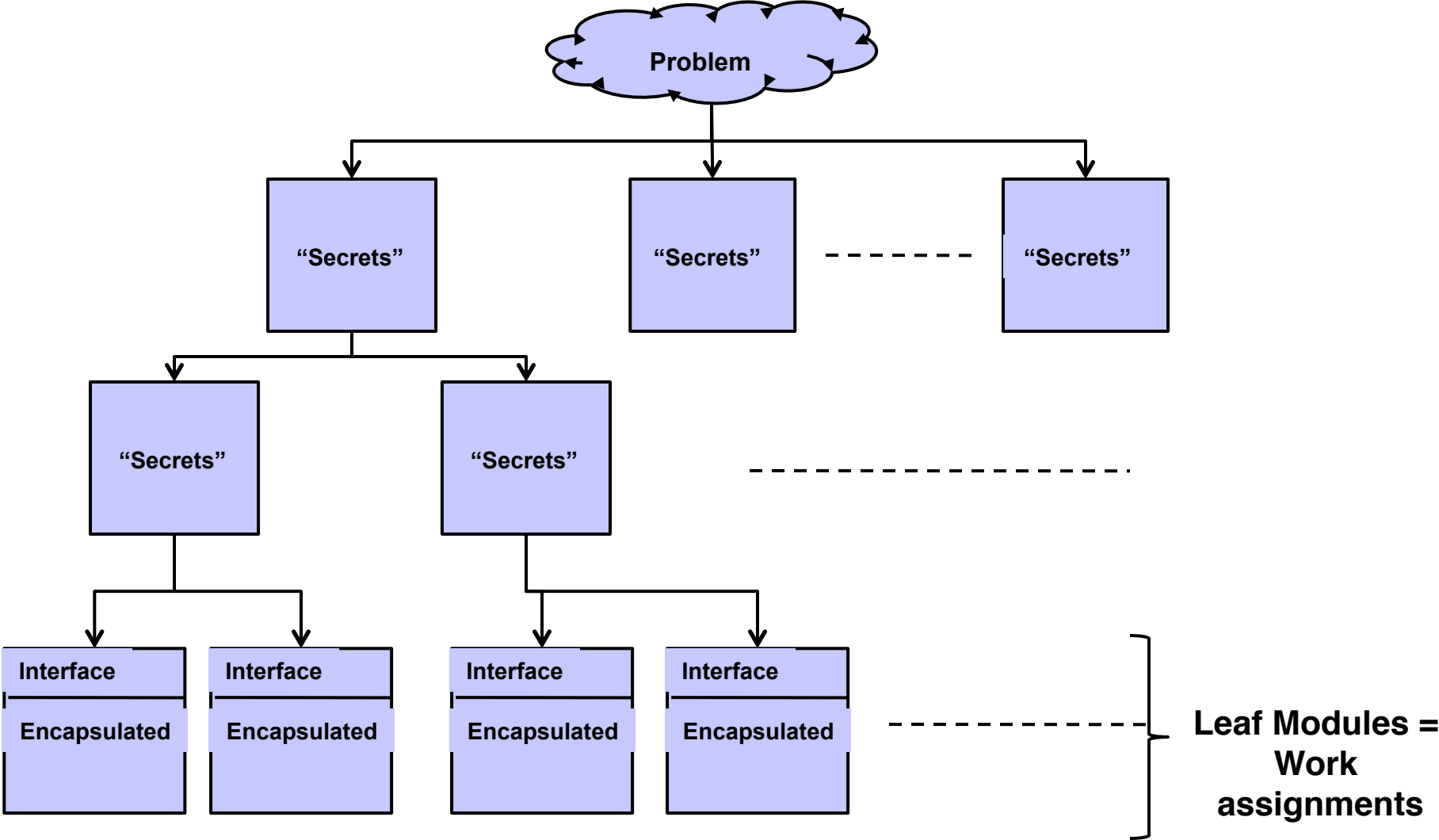
Modular Structure

- Comprises components, relations, and interfaces
- Components
 - Called modules
 - Leaf modules are work assignments
 - Non-leaf modules are the union of their submodules
- Relations (connectors)
 - submodule-of \Rightarrow implements-secrets-of
 - The union of all submodules of a non-terminal module must implement all of the parent module's secrets
 - Constrained to be acyclic tree (hierarchy)
- Interfaces (externally visible component behavior)
 - Defined in terms of access procedures (services or method)
 - Only external (exported) access to internal state

Decomposition Criteria

- Principle: information hiding
 - System details that are likely to change independently should be encapsulated in different modules.
 - The interface of a module reveals only those aspects considered unlikely to change.
- What do I do next?
 - For each module, determine if its secret contains information that is likely to change independently
- Stopping criteria
 - Each module is simple enough to be understood fully
 - Each module is small enough that it makes sense to throw it away rather than re-do.

Module Hierarchy



→ Submodule-of relation

Method of Communication

Module Guide

- Documents the module *structure*:
 - The set of modules
 - The responsibility of each module in terms of the module's secret
 - The “submodule-of relationship”
 - The rationale for design decisions
- Document purpose(s)
 - Guide for finding the module responsible for some aspect of the system behavior
 - Where to find or put information
 - Determine where changes must occur
 - Baseline design document
 - Provides a record of design decisions (rationale)

Method of Communication

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 (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

The FWS Module Structure

An overly simplified example

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


- C 1.** 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 a FWS may vary.
- C5.** The resolution of the *wind speed* sensors may vary.
- C6.** The *wind speed* sensor hardware on a FWS may vary.
- C7.** The transmitter hardware on a 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



**From
Requirements
Specification**

Top-Level Module Decomposition

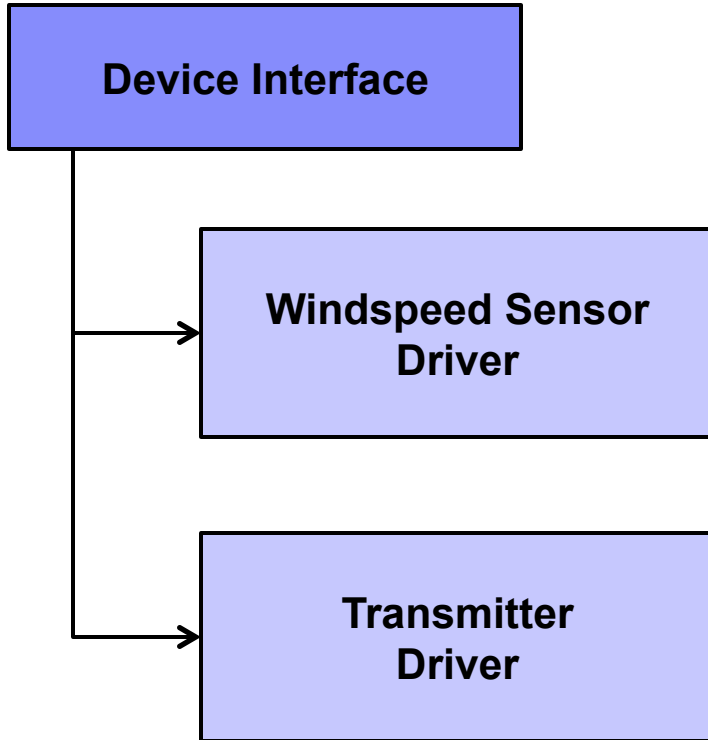
Device Interface

Behavior Hiding

Software Decision

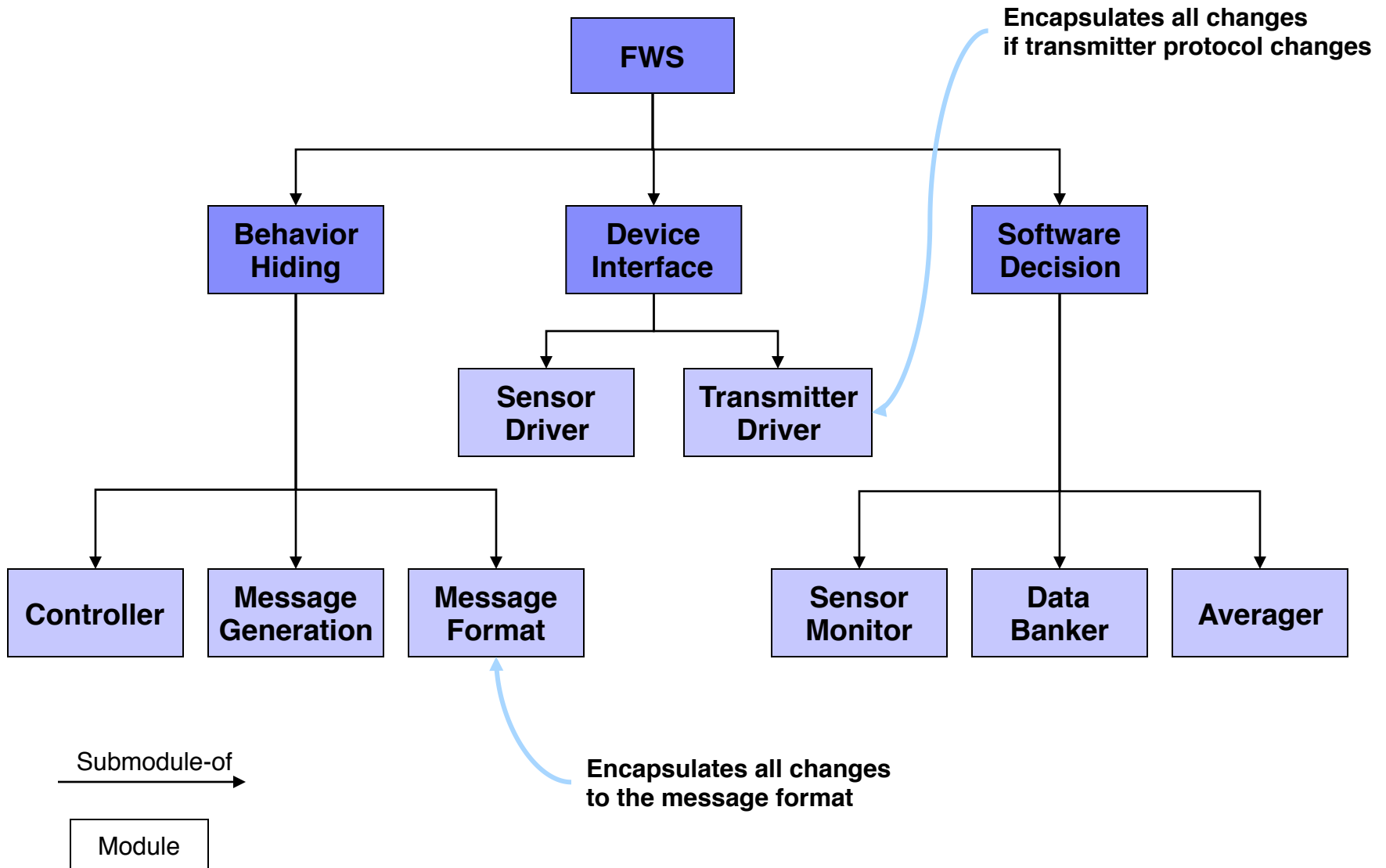
- Device Interface (DI)
 - Secret = properties of physical hardware
 - Encapsulates any hardware changes
- Behavior-Hiding (BH)
 - Secret = algorithms/data addressing requirements
 - Encapsulates requirements changes
- Software Decision (SD)
 - Secret = decisions by designer
 - Encapsulates internal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

FWS Modular Structure



Module Guide

- The module structure is documented in a module guide
- Contents describe:
 - The set of modules
 - The responsibility of each module in terms of the module's secret
 - The “submodule-of relationship”
 - The rationale for design decisions
- Document purposes
 - Orientation for new team members
 - Guide for finding the module responsible for some aspect of the system behavior
 - Where to find or put information
 - Determine where changes must occur
 - Baseline design document
 - Provides a record of design decisions (rationale)

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

Specifying Abstract Interfaces

Method of Communication

Module Interface Specifications

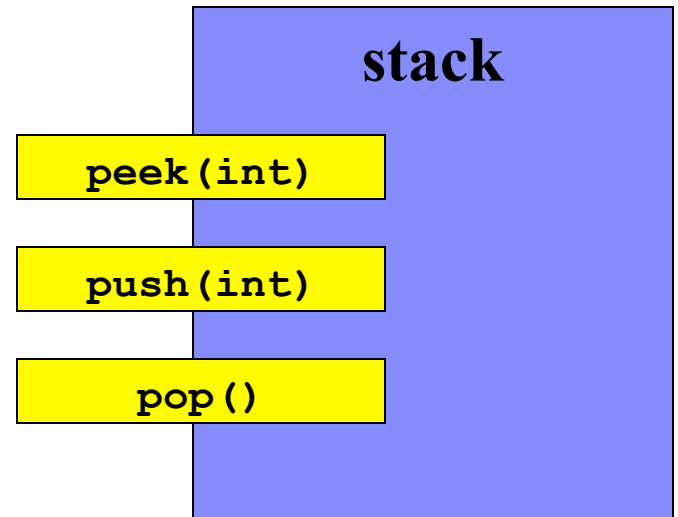
- Documents all assumptions user's can make about the module's externally visible behavior (of leaf modules)
 - Access programs, events, types, undesired events
 - Design issues, assumptions
- Document purpose(s)
 - Provide all the information needed to write a module's programs or use the programs on a module's interface (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

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
- 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
- Defines implementer's 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

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

An FWS Example: The Data Banker Interface Specification

Define services provided

Service	Provided By	Tested By
1. Initialize the set of stored sensor readings.	initialize	TC1, TC2, TC3, TC4, TC5
2. Store a new sensor reading, maintaining only the necessary history, and retrieve the current sensor reading history, keeping reads and writes synchronized.	read, write	TC1, TC2, TC3, TC4, TC5

An FWS Example: The Data Banker Interface Specification

Decide on syntax and semantics for accessing services

Access Methods

Access Method	Parameter name	Parameter type	Description	Exceptions	Map to services
initialize	sensorType	String	Type of sensor.		1
write	sensorType:I r:I	String SensorReading	Type of sensor. Sensor reading value		2
read:O	sensorType:I :O	String Vector<SensorReading>	Type of sensor. Vector of elements of type SensorReading		2

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

Access Method	Description
initialize	Initializes a vector of elements of type <i>sensorType</i> of length <i>HistoryLength</i> for each sensor of <i>sensorType</i> with initial values of null
write	Adds the <i>SensorReading</i> <i>r</i> to the back of the queue and removes the oldest sensor reading value from the front of the queue.
read	Returns the vector of sensor readings of type <i>sensorType</i> . With the most recent values of the sensor readings. The vector is of length (<i>HistoryLength</i> * number of sensors) of that type.

Synchronization: This module supports concurrent access to the *read* and *write* methods. Where any read or write can occur concurrently, the read and write statements act as atomic operators (i.e., the user will see either the sequence *read.write* or the sequence *write.read*).

An FWS Example: The Data Banker Interface Specification

- Decide on syntax and semantics for accessing services
- Local Data Types

Type	Value Space
<i>HistoryLength</i>	The number of sequential, past sensor values kept

- and Types Used

Type	Value Space
SensorReading	A triple (r, v, w) where r is of type SensorReading.resolution, v is of type SensorReading.value, and w 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

Step	Description	Input Type/Value	Expected Results	Service	Preamble
1	Initialize	sensorType	Type of sensor.		1
2	read	sensorType	Returns vector of null values		2

An FWS Example: The Data Banker Interface Specification

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.

Class DataBanker

[java.lang.Object](#)
└ **FWS.DataBanker**

```
public class DataBanker  
extends Object
```

The Data Banker provides synchronized storage for sensor readings.

Services Provided

1. Initialize the set of stored sensor readings.
2. Store a new sensor reading, maintaining only the necessary history, and retrieve the current sensor reading history, keeping reads and writes synchronized.

Synchronization: Supports concurrent access to read/write methods. Read or write operations on a vector of sensor readings act as atomic operations.

Exceptions: N/A

Uses: SensorReading

Field Summary

static int	HistoryLength HistoryLength is the number of wind speed readings that are retained
------------	---

Constructor Summary

DataBanker ()

Method Summary

static void	initialize (String sensorType, int numSensors) Initialize the DataBanker for a type of sensor reading.
static Vector < SensorReading >	read (String sensorType) Retrieve a set of readings for the sensor type

Using Javadoc

```

/** The Data Banker provides synchronized storage for sensor readings.
** <ul>
** Services Provided
** <ol>
** <li> Initialize the set of stored sensor readings.
** <li> Store a new sensor reading, maintaining only the necessary
** history, and retrieve the current sensor reading history, keeping
** reads and writes synchronized.
** </ol>
** <p>
** Synchronization: Supports concurrent access to read/write methods.
** Read or write operations on a vector of sensor readings act as atomic
** operations.
** <p>
** Exceptions: N/A
** <p>
** 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.
    ** <p>
    ** Initializes a vector of elements of type sensorType of length
    ** HistoryLength for each sensor of sensorType with initial values of null.
    **
    ** @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);
    }
}

```

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

Interface Design

Considerations in interface design

Design principles

Role of information hiding and abstraction

Module Interface Design Goals

General goals addressed by module interface design

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
 - Any other program must access internal data by calling access programs on the interface
- Consistent with good OOD principles

2. Provide Services

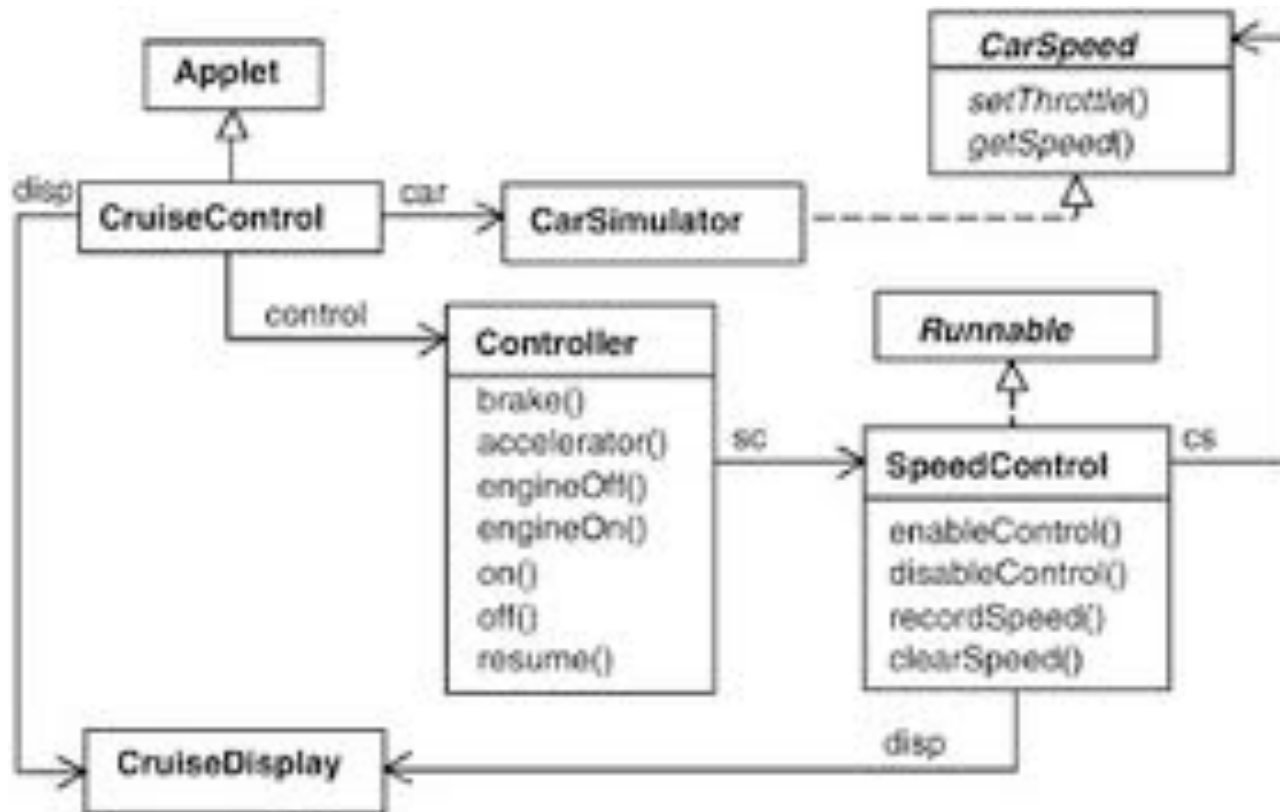
- Interface provides the capabilities of the module to other modules in the system, addressed by:
- *Abstraction*: interface design principle of providing only essential information and suppressing unnecessary detail

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

Questions

Assignment

- For Thursday
 - Standup report in class (no slides): status of major deliverables
 - Schedule project status meeting with instructor
 - To do: make sure I have a link to your current assembly site