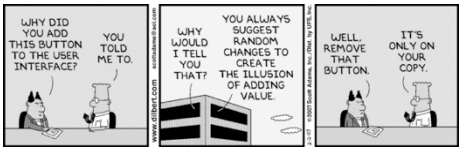
From Module Decomposition to Interface Specification

Designing a module structure

FWS Example



Elements of Architectural Design

- Design goals
  - What are we trying to accomplish in the decomposition?
- Relevant Structure
  - How to we capture and communicate design decisions?
  - What are the components, relations, interfaces?
- Decomposition principles
  - How do we distinguish good design decisions?
  - What decomposition (design) principles support the objectives?
- Evaluation criteria
  - How do I tell a good design from a bad one?

2

Architecture Design 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? (goals<->architectural structures<->representation)
   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 (called *modules*)
  - Most influential design time structure
    - Modifiability, work assignments, concurrent development, maintainability, reusability, understandability, etc.
- Uses Structure
  - Determine which modules may use one another's services
  - Determines subsetability, ease of integration

Designing the Module Structure

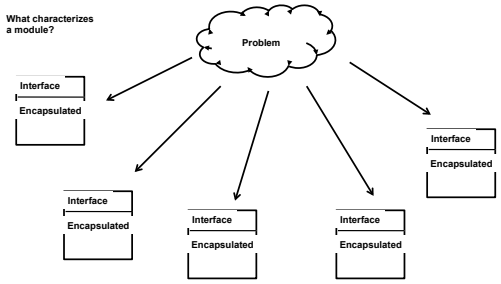
Modularization

- For any large, complex system, must divide the coding 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

Relation to Stakeholder Goals

- Reduce complexity, improve manageability
- Coding
  - Can write modules with little knowledge of other modules
  - Replace modules without reassembling the whole system
- Managerial
  - Allows concurrent development
  - Avoids "Mythical Man Month" effect ("adding people to a late software project makes it later")
- Flexibility/Maintainability
  - Anticipated changes affect only a small number of modules (preferably one)
  - Can calculate the impact and cost of change
- Review/communicate
  - Can understand or review the system one module at a time

Notional Modules



What is a module?

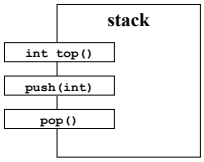
- Concept due to David Parnas (conceptual basis for objects)
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design and implementation decisions that other parts of the system *should not depend on*
- Modules are abstract, design-time entities
  - Modules are "black boxes" – specifies the visible properties but not the implementation
  - May, or may not, directly correspond to programming components like classes/objects
    - E.g., one module may be implemented by several objects

Floor Plan Analogy

- Meaning of "design-time abstraction"
- Represents a particular set of components and relationships
- Removes unnecessary detail
- But, is precise in what it represents
  - Does not mean "vague"
  - Correspondence to implementation is verifiable

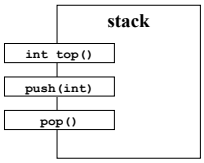
A Simple Module

- A simple integer stack
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *top*: get value of top element
- What information is on the interface?
- What are the secrets?
- What information is missing?
- Why is this an abstraction?



A Simple 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
  - *top*: 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
- A module spec is *abstract*: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)



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**

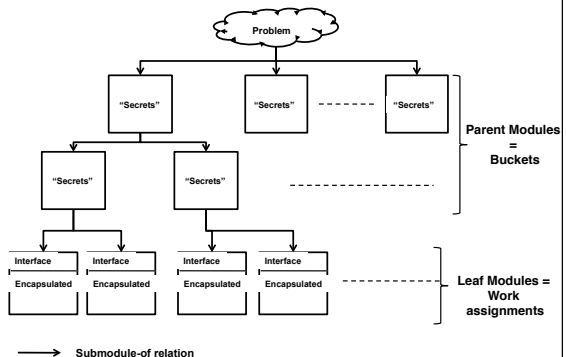
Is a module a class/object?

- The programming language concepts of classes and objects are based on Parnas' concept of modules
- To separate design-time concerns from coding issues, however, *they are not the same thing*
  - A module must be a work assignment at design time, does not dictate run-time structures
  - Coder free to implement with a different class structure as long as the interface capabilities are provided
  - Coder free to make changes as long as the interface does not change
- In simple cases, we will often implement each module as a class/object

Module Hierarchy

- For large systems, the set of modules need to be organized such that
  - We can check that all of the functional requirements have been allocated to some module of the system
  - Developers can easily find the module that provides any given capability
  - When a change is required, it is easy to determine which modules must be changed
- The *submodule-of* relation provides this architectural view (parent/child)

Module Hierarchy



Modular Structure

- Architecture = 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 => 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 behavior)
  - Defined in terms of access procedures (services or methods)
  - Only access to internal state

A Decomposition Approach

Decomposition Strategies Differ

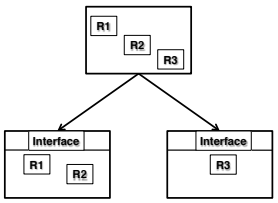
- How do we develop this structure so that *we know* the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Steps in processing: each module is a step in a chain of processing
  - Data: data transforming components
  - Client/server
  - Use-case driven development
- But, these result in different kinds of dependencies (strong coupling)

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 are likely to 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

Effects of Changes

- Consider what happens to communication among module developers
- Suppose we have groups of requirements R1 – R3:
  - R1 and R3 are related and likely to change together
  - R2 is likely to change independently
- Suppose we put R1 and R2 in the same module and assign to different teams
  - What happens when R1 changes?
  - R2?
- Suppose R1 and R3 are put in the same module?



Applied Information Hiding

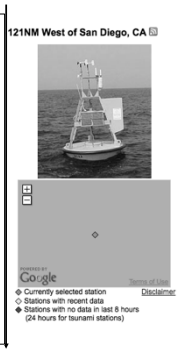
- The rule we just described is calls *the information hiding principle*
- Information hiding (or encapsulation): Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design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

The FWS Module Structure

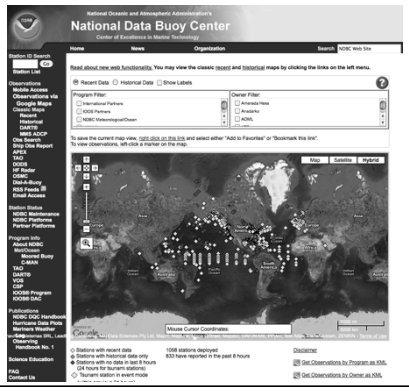
An overly simplified example

A Floating Weather Station

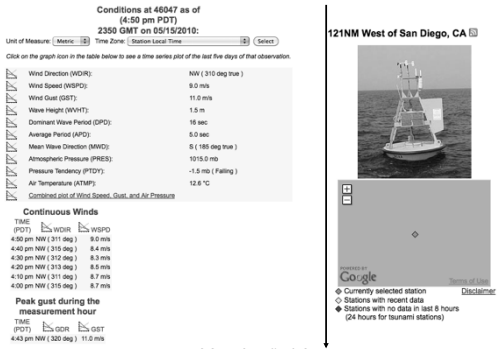
Floating weather stations (FWS) are buoys that float at sea and that are equipped with sensors to monitor weather conditions. Each FWS has an on-board computer that maintains a history of recent weather data. At regular intervals the buoy transmits the weather data using a radio transmitter.



http://www.ndbc.noaa.gov/

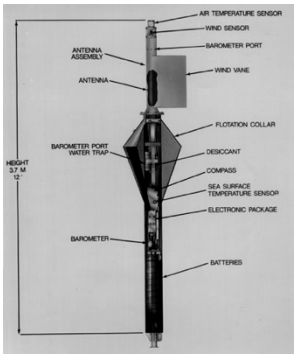


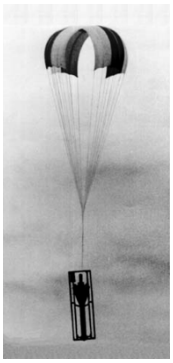
25



26

Drifting FWS





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 w*ind 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.

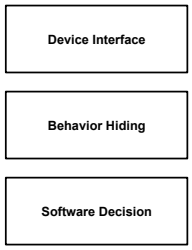
Devices
C4. The number and types of *wind speed* sensors on a FWS may vary.
C5. The resolution of the win*d speed* sensors may vary.
C6. The win*d speed* sensor hardware on a FWS may vary.
C7. The transmitter hardware on a FWS may vary.
C8. The method used by sensors to indicate their reliability may 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

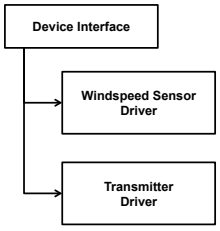
From Requirements Specification

Top-Level Module Decomposition



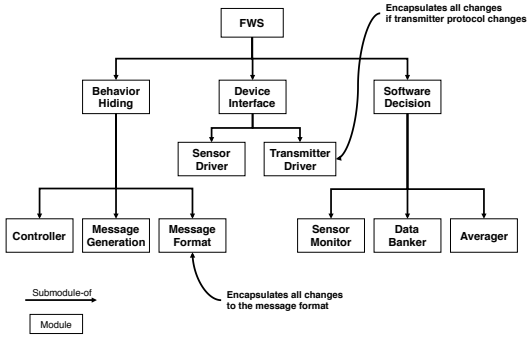
- Device Interface (DI)
  - Secret = properties of physical hardware
  - Encapsulates any hardware changes
- Behavior-Hiding (BH)
  - Secret = algorithms/data addressing requirements
  - Encapsulates requirements changes
- Software Decision (SD)
  - Secret = decisions by designer
  - Encapsulates internal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



Key Properties of Module Structure

- Keep the purpose of the structure in mind
- Goals:
  - Robust relative to change
  - Can determine where to *put* or *find* specific information quickly and easily
- Partition: each kind of information goes in exactly one place
  - Structure guides the reader to location of specific information
  - Decisions about structure chose to be unlikely to change
- Information hiding:
  - Information likely to change encapsulated in modules
  - Information that changes independently put in diff. modules

Documenting a Module Structure

Communicating Architectural Decisions

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?)

Architectural Specification

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)

Architectural Specification

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

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

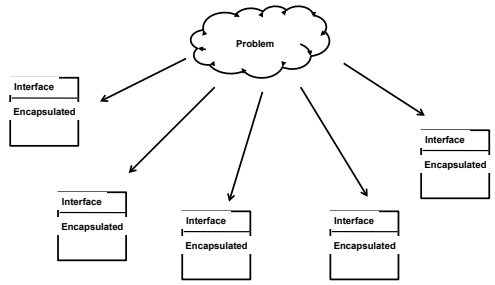
- What have we accomplished in creating the module structure?
- Divided the system into parts (modules) such that
  - Each module is a work assignment for a person or small team
  - Each part can be developed independently
  - Every system function is allocated to some module
- Informally described each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

Questions

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
  - Components can be designed independently
  - Components can be understood independently
  - Components can be tested independently
  - Components can be changed independently
  - Integration goes smoothly

Notional Modules

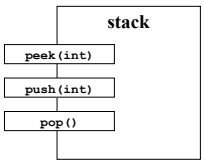


What is a module?

- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system should not depend on
- Modules are abstract, design-time entities
  - Modules are "black boxes" – specifies the visible properties but not the implementation
  - May or may not directly correspond to programming components like classes/objects

A Simple Module

- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
- The *secrets* (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- A module spec is *abstract*: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)



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**