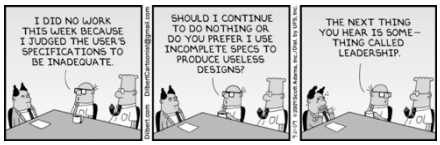
Modularization and Interface Specification

Designing a module structure

Communicating design decisions



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?
   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?)

Module Structure Design Goals

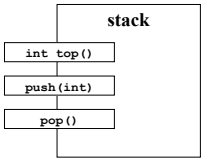
- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a "module."
- Properties of a "good" module structure
  - Components can be designed independently
  - Components can be understood independently
  - Components can be tested independently
  - Components can be changed independently
  - It is clear where to put or find specific information
  - Integration goes smoothly

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

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

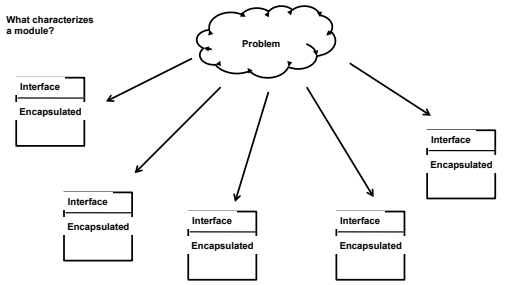
- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don't change the interface

Module User

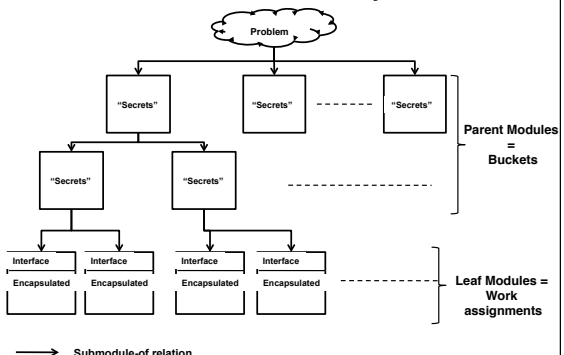
- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

***Key idea*: the abstract interface specification defines a contract between a module's developer and its users that allows each to proceed independently**

Notional Modules



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

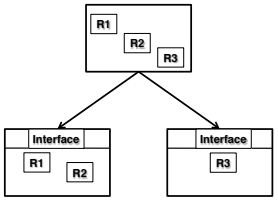
- How do we develop this structure so that *we know* the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Pipes and Filters: each module is a step in a chain of processing
  - Transactional: data transforming components
  - 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 holds decisions 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 called the *information hiding principle*
- 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

Design Principles

Three Key Design Principles

- Address the basic issue: which constructs are essential to the problem solution vs. which can change
  - "Fundamental assumptions"
  - "Likely changes"
- Most solid first
- Information hiding
- Abstraction

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

- 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 encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

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

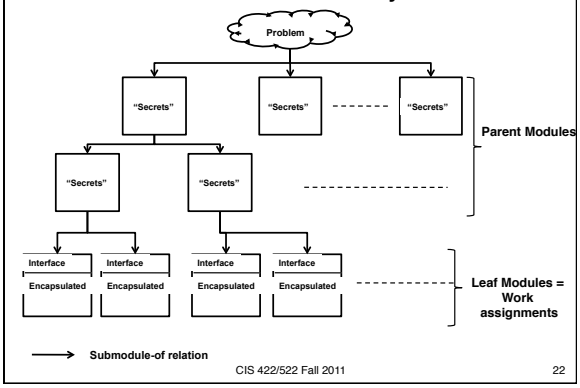
Abstraction

- 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)

Example: Simple Library Model

- What are the abstractions?
- What information is hidden?

Module Hierarchy



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