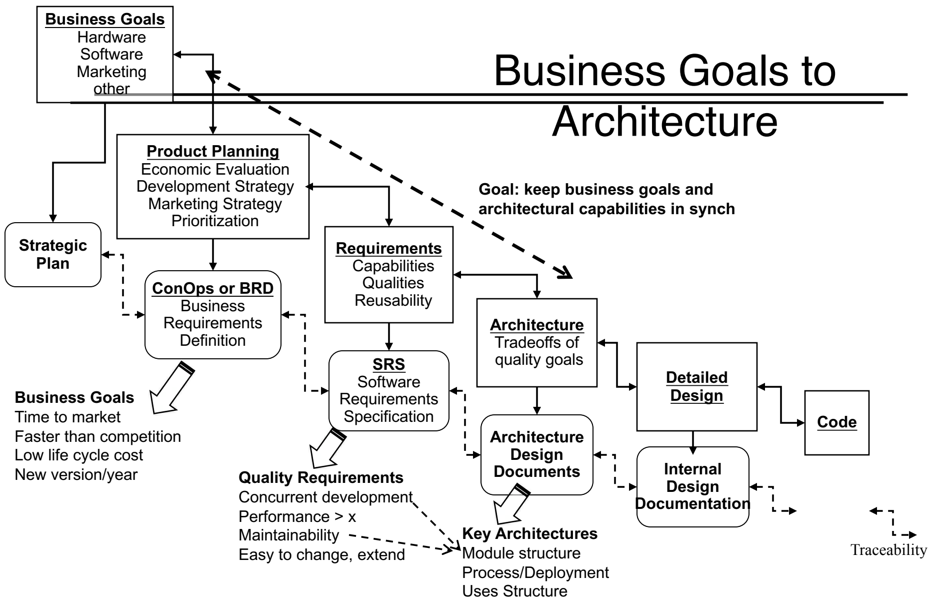
Architectural Design

Why principles work

Documenting architecture

DILBERT
BY SCOTT ADAMS
WHEN WILL YOU HAVE THAT DONE?
TWO WEEKS.
CAN YOU DO IT FASTER?
YES.
ALL I NEED TO DO IS LOWER THE QUALITY.
TELL ME WHAT YOUR MINIMUM ACCEPTABLE QUALITY LEVEL IS AND I'LL TELL YOU WHEN YOU CAN HAVE IT.
I WANT IT IN ONE WEEK.
I CAN DO THAT AT 50% OF PLANNED QUALITY.
WHY DOES IT FEEL AS IF I'M NOT REALLY MANAGING ANYTHING HERE?
MAYBE YOU COULD GO MANAGE SOMEONE ELSE NOW.
I CAN'T TELL IF I'M DOING MY JOB NOW.
IS IT YOUR JOB TO PREVENT ME FROM WORKING?
DilbertCartoonist@gmail.com
www.dilbert.com

Business Goals to Architecture



Architecture Design Process

Breaking design process into a manageable set of steps:

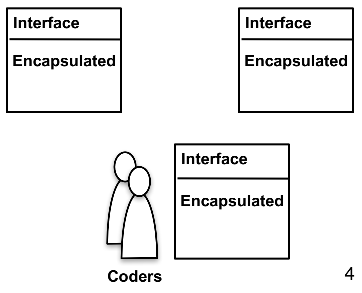
1. Understand the goals for the system
2. Define the quality requirements
3. *Design the architecture*
    1. Views: choose a set of views representing highest priority quality requirements (goals<->architectural structures<->representation)
    2. Design: design to meet quality requirements
    3. Documentation: communicate the design by documenting the views and rationale (see examples)
4. Evaluate the architecture (does the design meet the design goals?)

Design Goal Implications

Business Goals
- Concurrent development
- Easy to maintain
- Easy to change/evolve

Implication: requires r*elatively independent* work assignments
- Dependencies are few and simple
- Structure is stable
- Likely changes affect one or few modules

Can achieve this if:
- Things that are likely to change are encapsulated
- Things that change together are encapsulated together
- Interfaces are simple and well defined
- Interfaces contain only things unlikely to change



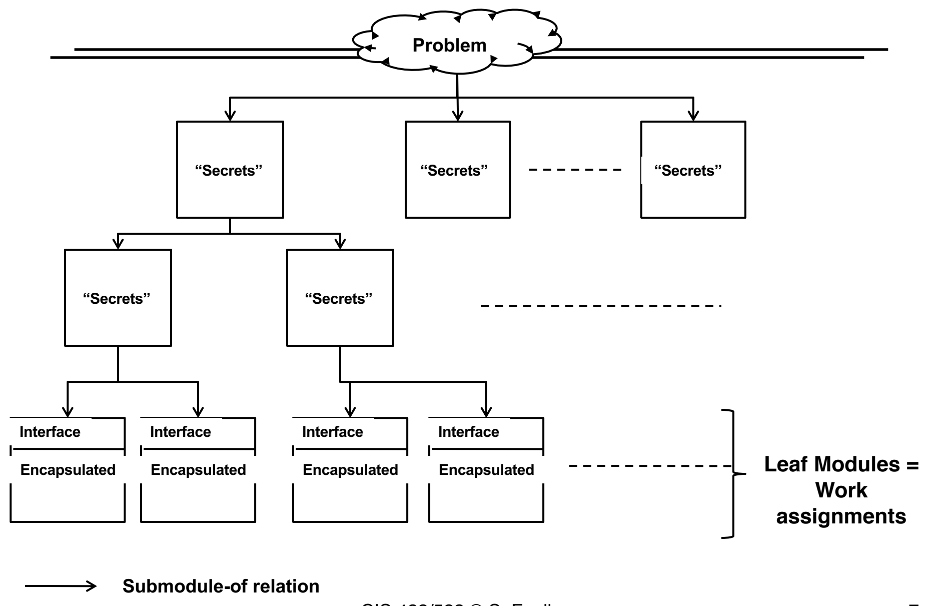
Module Construction

- Detailed design goals
    1. Things that are likely to change are encapsulated
    2. Things that change together are encapsulated together (independently in different modules)
    3. Interfaces are simple and well defined
    4. Interfaces contain only things unlikely to change
- Get the structure right, then get the details right
    - Allocate requirements to modules satisfying 1 & 2
    - Define interfaces to satisfy 3 & 4

Principles vs. Heuristics

- Suggested a set of design principles
    - Most solid first
    - Information hiding
    - Abstraction
- OOD gave us heuristics
    - Underline the nouns
    - Identify causal agents
    - Identify coherent services
    - Identify real-world items
- Why would you prefer one to the other? Which is more effective?

Module Hierarchy



Information Hiding Decompositon

- Decompose recursively
  - If a module holds decisions that are likely to change independently, then decompose it into submodules
  - Decisions that are likely to change together are allocated to the same submodule
  - Decisions that change independently should be allocated to different submodules
- Stopping criteria
  - Each module contains only things likely to change together
  - Each module is simple enough to be understood fully, small enough that it makes sense to throw it away rather than re-do
- Define the Interfaces
  - 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

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

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. Design: how do we decompose the system?
    3. *Documentation: how do we communicate design decisions?*
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

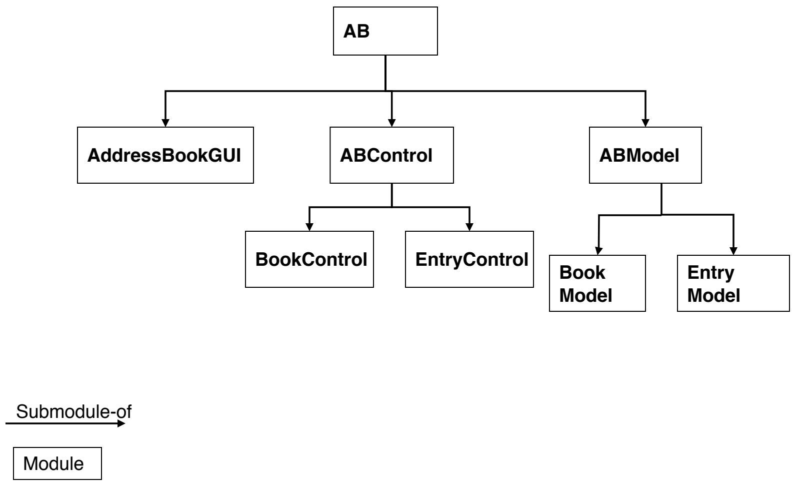
Communicating Architecture

- 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

Address Book Modular Structure



Excerpts from a Module Guide (1)

1. AddressBookModel

The ABModel modules include programs that need to be changed if the data model (types of data and relationships among data) is changed. Its secrets include how address books and their associated data are stored and retrieved.

1.1 Book Model

Includes programs that must be changed if the data and relations associated with an address book changes.

Services

Provides the services needed to operate on address books as a whole.

Secret

Algorithms and data structures used create and maintain address books or retrieve information about address books.

1.2 Entry Model

Includes programs that must be changed if the entity model or its implementation are changed.

Excerpts From Module Guide (2)

2. AddressBookControl Modules

The ABControl modules consist of those programs that need to be changed if the operations on address books or address book entries are changed. Its secrets include the how the application implements the set of address book operations specified in the requirements.

2.1. BookControl

The Book Control modules consist of those programs that need to be changed if the operations on address books change. Its secrets include the algorithms used and how the BookControl operations use the ABModel to set or retrieve information about address books.

2.1.1 SortAB

Service

Provides services to sort the entries in an address book by field values.

Secrets

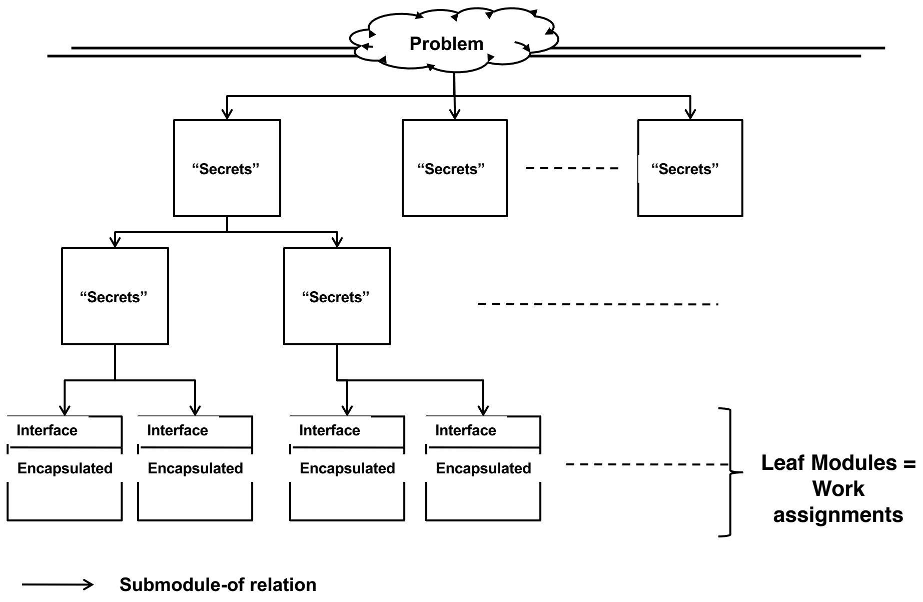
Algorithms used to compare and sort entries. How this module uses the services provided by the ABModel.

Specifying Abstract Interfaces

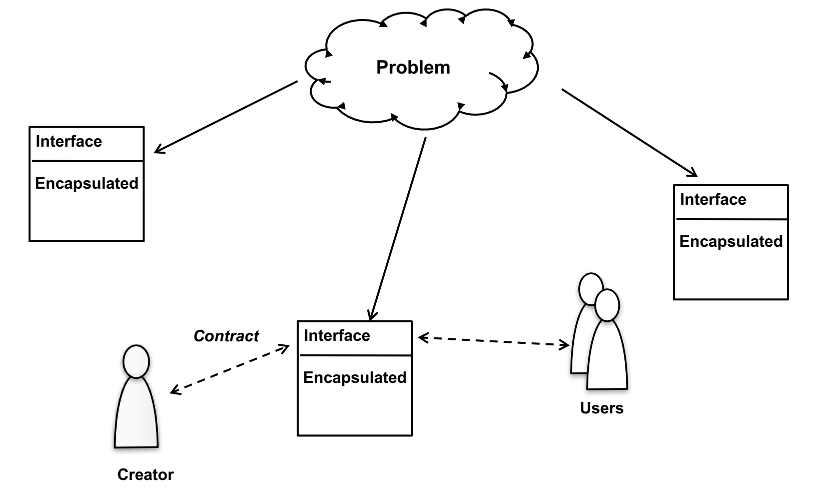
Module Interface Design

- Architectural design: get the structure right, then get the details right
- Module structure: allocated requirements to modules with high-cohesion, loose coupling
  - Those that change together are in the same module
  - Those that change independently are in different modules
- Interface design must follow through
  - Encapsulate likely changes
  - Provide coherent set of services
- Again: must create the design the communicate the design decisions

Module Hierarchy



Notional Modules



Module Interface Design Goals

General goals addressed in module interface design

1. Control dependencies: apply *information hiding*
   - Encapsulate anything other modules should not depend on
   - Hide design decisions and requirements that might change (data structures, algorithms, assumptions)
2. Provide services: apply *abstraction*
   - Provide all the capabilities needed by the module's users
   - Provide no more than is needed (reduce complexity)
   - Provide problem appropriate abstraction (understandability)
   - Provide reusable abstractions

•Specific goals need to be captured (e.g., in the module guide and interface design documents)

Which Principle to Use

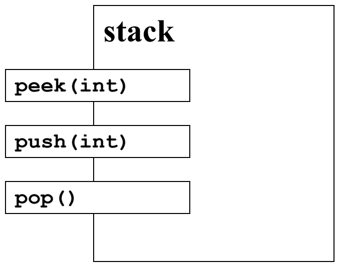
- Use abstraction when the issue is what should be on the interface (form and content)
- Use information hiding when the issue is what information should not be on the interface (visible or accessible)
- AddressBook Model example

Need for Precise Interface Specifications

- Informal description is not enough to write the software
- To support independent 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?

- Preference for an *abstract interface specification*
- 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 (one-to-many)
  - Preserves the module's secrets

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

Data Banker

3.3. Data Banker

Service

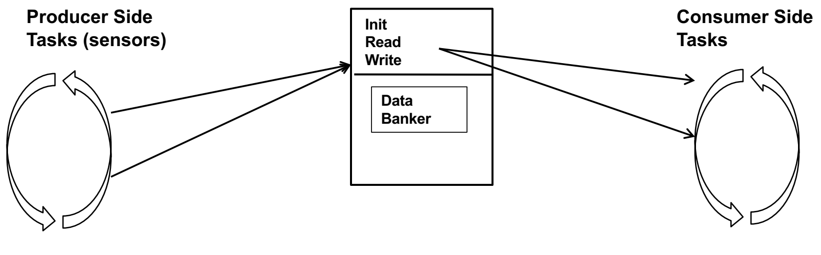
Store wind speed data giving readers and writer concurrent access

Secret

The algorithm and data structure used to store and retrieve data

Associated Changes

None



DB Example

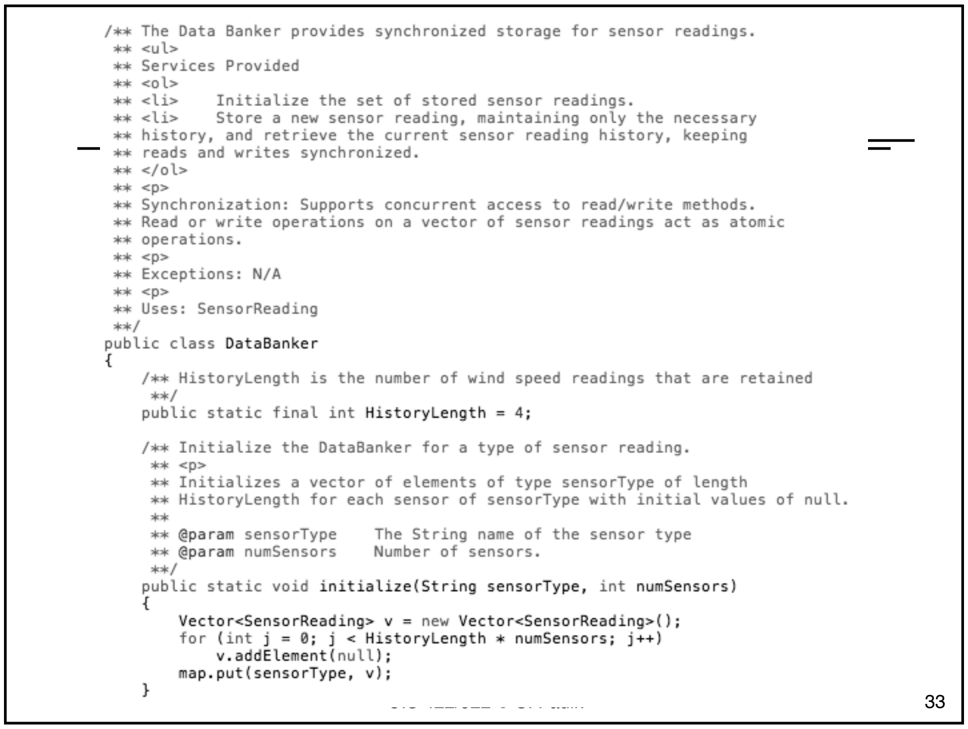
Benefits Good Module Specs

- 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

- Try to provide two views including a module decomposition (if appropriate)
- Include rationale for the overall design
- Include any significant design decisions
- Expected outcomes:
  - Should be able to trace from requirements to code objects
  - Should be able to understand how your design addresses your design goals (quality requirements, developmental goals)

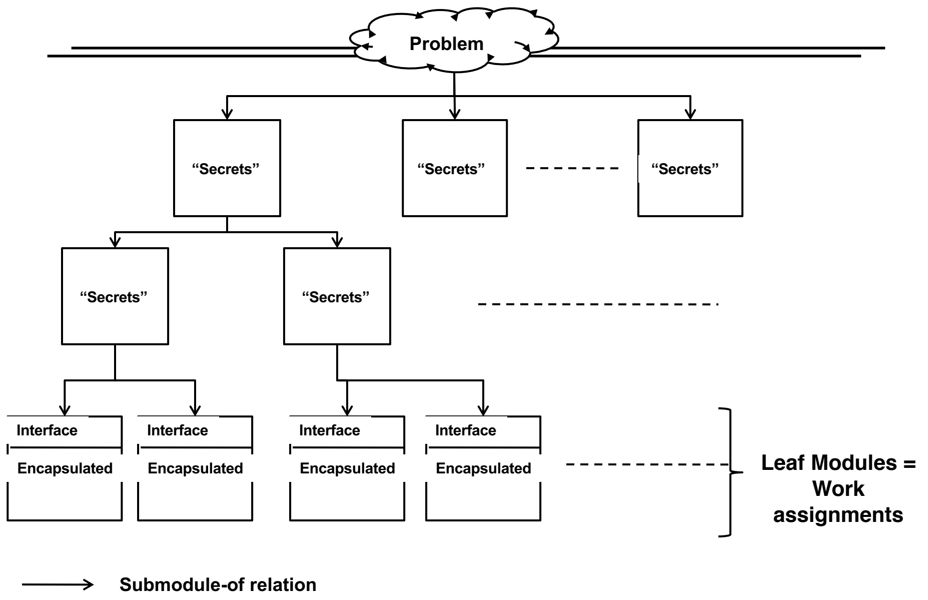
Questions?



```
/** 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);
    }
```

Module Hierarchy


Problem
"Secrets"
"Secrets"
"Secrets"
"Secrets"
"Secrets"
Interface
Encapsulated
Interface
Encapsulated
Interface
Encapsulated
Interface
Encapsulated
Leaf Modules = Work assignments
Submodule-of relation