Today’s educational objective

- Understand information hiding as a principle for decomposing systems into modules

Background

- “I assume the programmer’s genius matches the difficulty of his problem and assume that he has arrived at a suitable subdivision of the task.” —Dijkstra
- “Usually nothing is said about the criteria to be used in dividing the system into modules.” —Parnas

Information hiding principle

- A fundamental cost in software engineering is accommodating change
- A change that requires modifying multiple modules is more costly than a change that is isolated in a single module
- Therefore
  - Anticipate likely changes
  - Define interfaces that capture the stable aspects and implementations that capture the changeable aspects

Small examples

- double sqrt (int)
  - Can be implemented using bisection methods, factoring methods, Newton’s method
  - The client doesn’t care, and this can change (requiring only relinking)
  - Very low level example, of course

```
#include <math.h>

double sqrt(int x) {
    return sqrtf((float)x);
}
```

- Classic example of data abstraction
  - The representation and algorithms are hidden
  - They can change without affecting clients

Hiding secrets

- These two examples show specific kinds of secrets that modules hide
  - Algorithms
  - Data representations
  - The interfaces capture stable decisions
    - Clients depend on these interfaces
  - The implementations encode the changeable parts
    - Clients do not depend on these
An interface has two parts:
- The signature: the names and type information about the exported functions
- The specification: a precise description of the semantics of the elements in the module

Most commonly, the signature is in a programming language and the specification is in natural language.

But you cannot neglect the specification.

---

Example:

```java
double sqrt (int x) {
    a legitimate different implementation
}
```

```java
double sqrt (int x) {
    return 3.14159;
}
```

```java
bool member (intSet s, int i) {
    return IsOdd(i)
}
```

The contract with the client includes semantic information.

---

Design Level:

- Information hiding is a design principle, not a coding principle
- Obviously, it can be reflected in code that is based on the design

---

Anticipating change:

- It's "easy" to anticipate algorithmic and representational changes
- But you cannot and should not do this and only this
  - By blithely anticipating these changes, you may not think about another kind of change that is more likely and potentially costly
- In general, you cannot build a design that effectively anticipates all changes

---

Data isn't always abstracted:

- Unix byte streams are pervasive
  - imagine trying to change Unix's data model from byte streams to fixed width records
  - good or bad decision?
- Y2K problems arose because a date representation was exposed
  - The USPS, the DJIA and McDonald's have also faced similar problems
- Other examples?
Other kinds of secrets

- An information hiding module can hide other secrets
  - Characteristics of a hardware device
    - Ex: whether an on-line thermometer measures in Fahrenheit or Centigrade
  - Where information is acquired
    - Ex: the Metacrawler (www.metacrawler.com) might hide what other web search engines it uses
  - Other examples?

KWIC: the classic example

Input
- now is the time for all good students to come to the aid of their professors
Output
- aid to come to the aid of their professors
- now is the time for all good students
- for all good students
- now is the time
- of their professors
- their professors of time
- now is the
- to come to the aid
- to the aid to come

The classic decomposition

- Top-down functional decomposition
  - Stepwise refinement
  - Based on the steps the actual computation will take
  - Master Control
    - Input
    - Process Data
    - Output
      - Circular Shift
      - Alphabetize

Cals

Sequence

The data decomposition

- Not based on the actual computation steps
- Hides decisions about data representation
  - Could they be hidden in the previous decomposition?
  - Hides decisions about the granularity of sorting
  - The “sequence” relationship is hazier

Fundamentally different approaches

- These are really different designs
- They are equivalent in terms of the actual user experience
- Indeed, Parnas argued that in principle a compiler could produce identical executables from these two different decompositions

What about performance?