

Where we are in the course

Software Lifecycles

1. Waterfall
2. Prototyping

Requirements Engineering

1. The A7E Table Model (formal)
2. Use case analysis (for informal, UI oriented).

⇒ Software Design ⇐

1. Design Principles: Modularity, Information Hiding/Separation of Concerns, Abstraction: (data, control).
2. Architectural Design, Architectural Styles.
3. Object Oriented Design.

Formal Methods (Requirements & Design)

Implementation Techniques

Testing

Software Design

Design Basics.

- Goals of design: Numerous—efficiency, cost, reliability, etc. (But for software engineers): reuse, evolvability, simplicity, etc.
- The impact of good design is *enormous*.
- Design at various levels: architecture, modules, classes, functions, algorithms and datastructures.
- Design is hard—many possible choices, and hard trade-offs. (it's an art). *But!* there are a lot of examples.

What we will study.

- **Design Principles:** (mostly from Prof. David Parnas)
 1. Modularity
 2. Information Hiding/Separation of Concerns
 3. Data Abstraction
 4. Control Abstraction*but watch the trade-offs!*
- **Design by Precedent/archetype** (coming lectures)
 1. Architectural Styles
 2. Design Patterns

Modularity

”Any lexically contiguous sequence of program statement, bounded by boundary elements, having an aggregate identifier” (*Yourdon & Constantine*)

Example: A C Function?

Example: A C++ class?

Example: A CORBA object?

Example: A process in a multi-process system?

Example: A telephone feature (like POTS)?

An alternate definition: A unit of software that maps nicely to an organization: (an individual, supervisor, lab director etc), and can be maintained by that group.

Modules make the system less complex, easier to understand, and easier to maintain.

Modules occur in various ”layers”, and can be of different sizes.

Key question: How to decompose systems into modules.

Answer: Consider coupling, cohesion, information hiding, abstraction, separation of concerns, etc.

Coupling & Cohesion

Coupling: degree or strength of relationship between modules. The lower the degree of coupling, the more independently a module can be evolved, and the less likely that a fault in one module will percolate to another module.

```
static int i; /* a global variable */

int f(int a) {
...
    a = ...i...;
...
}

int g (int b){
...
    i = ...b...;
...
}
```

In C++, many different types of coupling between classes: method–method, method–attribute, with ancestors, descendants, friends, friends⁻¹, etc. Studied by Briand, Devanbu, Melo, El-Emam, etc. Not enough real data yet.

NB: When might you need to have high coupling?

Cohesion

Cohesion Modules whose elements are strongly related to each other have high cohesion. Incohesive modules don't make sense and are hard to maintain.

Information Hiding

Hiding: A module has clients that use it's services. The clients should see just as much about the module as they need to, and no more.

"Secrets help make evolution easier!"

Example Searching Tables

```
/* Two nice little functions that
search for an element in a list */
```

```
int find1(int key) ) {
static int *array;
... search ...
if (found)
return(index)
else return (-1)
}

int find2(int key ) {
static int *array;
... search ...
if (found)
return (1)
else return (-1)
}
```

Questions

What's wrong with the above?

What features of C++ promote information hiding?

What features hinder it?

How about using vs "illegal" values as errors? What's a better way?

Data Abstraction

Creating Abstract Data Types that a) hide their implementation and b) can be used with any type of object.

```
template <class T, int STACKBOUND > class Stack{
    public: push (T obj) {
        if (cursor > STACKBOUND) throw ...
        else stackarr[cursor++] = T;
    };
    T pop () {
        ... check bounds ...
        return(stackarr[--cursor]);
    };
private:
    int cursor;
    T stackarr[STACKBOUND];
};
```

This is statically typed and It hides:

- Memory layout
- algorithms that are used

It abstracts:

- The size of the stack.
- The type of the object on the stack.

But what are the tradeoffs?

Control Abstraction

Creating Abstract algorithms with the same goals:

Example 1 :

```
(defun map (f list) (cons (f (car list)) (map f (cdr list))))
```

Example 2 : Assume tree is a structure with:

```
(root tree) → node  
(left tree) & (right tree) → tree  
(mktree node1 treel treer) → tree  
  
(defun trmap (f tree)  
  (if (null tree) (return nil))  
  (mktree (f (root x)) (trmap f (left tree))  
          (trmap f (right tree))))
```

Can abstract even higher, across Lists, Trees, etc:

But What are the trade-offs?

Abstraction

Example 3:

```

template <class T> void sort (Vector<T>& v,
                             Comparator<T> &cmp) {

    unsigned int n = v.size();

    for (int i = 0; i < n-1; i++)
        for (int j = n -1; i< j; j--)
            if (cmp.lessthan(v[j], v[j-1])) {
                T temp = v[j];
                v[j] = v[j-1];
                v[j-1]= temp;
            }
}

```

Control or Data abstraction?

Is abstract enough?

What are the tradeoffs?

Conclusion

- Software engineering concerns with design are usually **reuse**, *evolution* and *maintenance*.
- Design is a difficult, creative process: but some principles: modularity, information hiding, abstracts (data and control).
- Although the examples are at a fairly low level, same problems apply at every level of modularity.
- And there are always tradeoffs: how do get adequate performance, security, etc while upholding these software engineering design requirements?: *architectural styles and design patterns!*