SOFTWARE DESIGN

Monika Gupta

Researcher, IBM Research

Architecture

Design

Stages of Design

Problem understanding

Identify one or more solutions

Describe solution abstractions

Repeat process for each identified abstraction until the design is expressed in primitive terms.

Software Design takes abstract requirements and creates detail ready to be developed

Decide classes, methods, data types to be used in the solution

Separate Architecture from Design (or don't)

Provide detail which is implementation-ready…

…but doesn't include implementation detail/constrictions.

SOFTWARE DESIGN: MODULARITY

Coupling

Cohesion

Information Hiding

Data Encapsulation

DEFINITION AND PRINCIPLES OF MODULARITY

Complex systems must be broken down into smaller parts

Three Primary Goals:

- Decomposability
- Composability
- Ease of Understanding

INFORMATION HIDING

- Hide complexity in "Black Box" – what is does, NOT How?

Examples – Class, Macros, Libraries, Functions

```
void sortAscending (int *array, int length)
```

Don't know which sort is uses

Don't really need to

Know how to use it

DATA ENCAPSULATION

- Encapsulate the data
- Helps find where problems are
- Makes design more robust

SUMMARY

Modularity is about the breakdown and reassembly of components

Many aspects at play

Coupling and cohesion

Information Hiding

Data Encapsulation

Coupling

Measuring the strength of connections between (sub-)system components

Loose coupling allow for changes to be unlikely to propogate across components

Shared variables and control information lead to tight coupling

Loose coupling achieved by state decentralization and message passing

COUPLING LEVELS - TIGHT

- Content Coupling – Module A directly relies on the local data members of module B rather than relying on some access or a method

- Common Coupling – Module A and module B both rely on some global data or global variable.

- External Coupling - External coupling is a reliance on an externally imposed format, protocol, or interface.

COUPLING LEVELS - MEDIUM

- Control Coupling – Control coupling happens when a module can control the logical flow of another by passing in information on what to do or the order in which to do it, a what-to-do flag.

- Data structure Coupling - When two modules rely on the same composite data structure, especially if the parts the modules rely on are distinct.

COUPLING LEVELS - LOOSE

- Data Coupling – When only parameters are shared. This includes elementary pieces of data like when you pass an integer to a function to compute the square root

- Message Coupling - It's primarily achieved through state decentralization, and component communication is only accomplished either through parameters or message passing

- No Coupling

SOFTWARE DESIGN: COHESION

Measures how well a module's components 'fit together'

Implements a single logical entity or function

Represents a desirable design attribute

Divides into various levels of strength

COHESION LEVEL - WEAK

- Coincidental Cohesion – Just because in the same file
- Temporal Cohesion - Temporal cohesion means that the code is activated at the same time
- Procedural Cohesion - one comes after the other doesn't really tie them together, not necessarily
- Logical Association - components which perform similar functions are grouped

COHESION LEVELS - MEDIUM

- Communicational Cohesion - all elements of the component operate on the same input or produce the same output

- Sequential Cohesion - when one part of the component is the input to another part of the component. It's a direct handoff and a cohesive identity.

COHESION LEVEL - STRONG

- Object Cohesion - each operation in a module is provided to allow the object attributes to be modified or inspected

- Functional Cohesion - Beyond sequential cohesion to assure that every part of the component is necessary for the execution of a single well-defined function or behavior. So, it's not just input to output, it's everything together is functionally cohesive.

INHERITANCE?

- By paying attention to the different types of coupling and cohesion, you can build better systems, better designs, and better solutions.

SOFTWARE DEVELOPMENT: IMPLEMENTATION TIPS

- Program when you are alert

- Write code for people, not for the computer

```
// if account flag is zero (HOW)
if ( accountFlag == 0 ) ...

// if establishing a new account (WHY)
if ( accountFlag == 0 ) ...

// establish a new account (COMMENT FOR SECTION HEADER, CODE TELLS WHY)
if ( accountType == AccountType.NewAccount ) {
   ...
}
```

Methods focused on a single task, preferably without side effects

Writing code for a second time? Extract into a method

Make each class represent a single, intuitive concept

Break long methods into manageable pieces

Write your comments, tests, and exception handling BEFORE you write functional code

Optimize only when you're sure it's necessary