CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 9.1: Design Quality

In this Session

- Cohesion
- Coupling
- Law of Demeter
- Mindless Classes
- God Classes
- Fragile Super Class

Terminology — Cohesion

Cohesion:

- How well do the parts of the class (state and methods) fit together?
- Do they all contribute to a single, clear purpose?

e.g. A `Car` class contains:

- Fuel
- Steering
- Speed
- Route planner
- Public holiday calculator

High cohesion is preferable.
(Low cohesion usually indicates that a class should be split.)

Terminology — Cohesion

Cohesion:

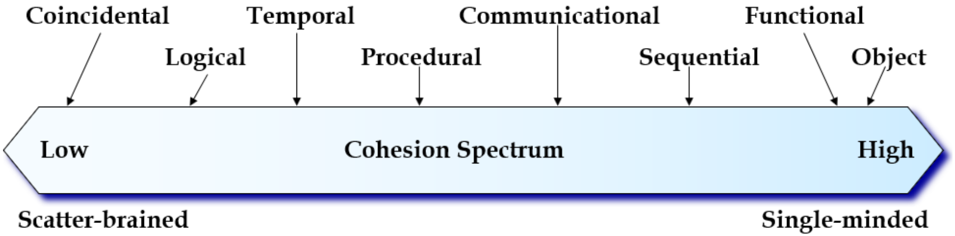
- How well do the parts of the class (state and methods) fit together?
- Do they all contribute to a single, clear purpose?

e.g. A `Car` class contains:

- Fuel $\leftarrow$ ok
- Steering $\leftarrow$ok
- Speed $\leftarrow$ok
- Route planner $\leftarrow$probably should not be here
- Public holiday calculator $\leftarrow$definitely should not be here

High cohesion is preferable.
(Low cohesion usually indicates that a class should be split.)

Cohesion



High Cohesion

- Class should be easier to understand
    - fewer extraneous ideas to think about
- Testing is easier
- Modification is easier

Cohesion — Type Overview

see McConnell, S. (2004). Code complete. pp. 445-452

Coincidental

- Grouped (e.g. in a class or method) arbitrarily and do not have a clear relationship to each other.

Logical

- Grouped together because they do the same thing.

Temporal

- Grouped together because of when they are executed during program runtime.

Procedural

- Grouped together because the operations are always done in the same specified order.

Communicational

- Grouped together because they operate on the same data.

Sequential

- Grouped together because operations are performed in a specific order, and operate on the same data.

Functional

- Grouped together because they all contribute to a single well defined objective or task.

Cohesion — Rule of Thumb

Write a sentence describing the purpose of the class

- Short, clear and unambiguous description implies good cohesion
- Compound sentence, contains a comma, or more than one verb — probably represents more than one concept
- Contains time related words — probably sequential or temporal cohesive
- Words like Initialise, clean-up, etc. — imply temporal cohesion
- Predicate doesn't contain a single specific object following the verb — probably logically cohesive

Terminology — Coupling

Coupling:

- To what extent does this class depend on other classes?
- How many methods are called on how many other classes?
- How much internal data is passed as parameters?
- How much internal data is returned by “getter” methods?
- Are references to internal data returned from methods?
- Can another object influence the flow of control in this object?

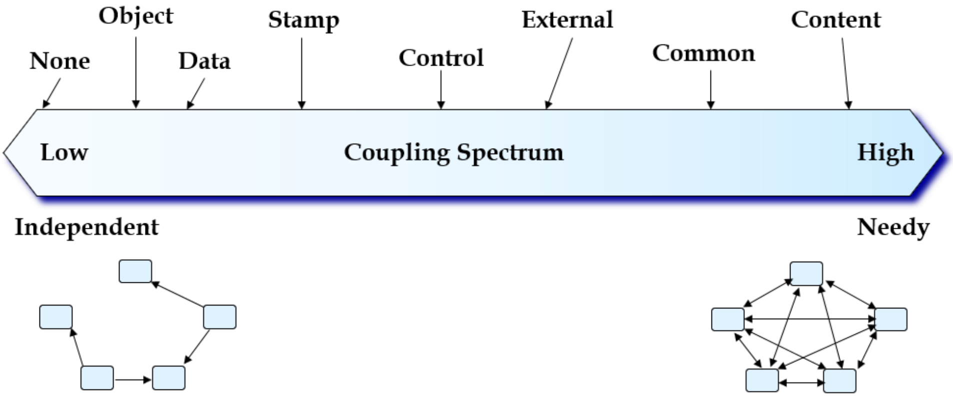
Consider the degree of complexity of the interface between classes.

Low coupling is preferable.

- Classes coupled to lots of other classes are harder to write and test in isolation.
- High coupling may indicate that a class has been split when it shouldn’t have been.

Coupling

Concerned with how dependent classes are on others.
Single biggest influence on system's understandability.



Note: both coupling and cohesion are relative terms.

Coupling — Type Overview

Data

- Sharing of data through means such as parameter passing in methods.

Stamp

- Sharing a data structure, but each class only needs access to a select part of it.

Control

- One class is controlling what happens in another class, by passing it information/instructions.

External

- Sharing something imposed by an external source. E.g. a data format.

Common

- Sharing access to the same global data/variables.

Content

- One class directly manipluates data another class (e.g. variables set public, not private).

Types of Coupling

Class X has an attribute of class Y

X has a method that references an instance of Y

- parameter or local variable of type Y
- object of type Y returned to it from a message

A method in X sends a message to Y

X is a direct or indirect subclass of Y

Y is an interface and X implements that interface

Y is a global variable and is accessed by X

Law of Demeter[1] (principle of least knowledge)

Target of a message (invoke methods) can only be one of the following objects

- The method's object (`this`)
- Object that was passed as a parameter
- Object referred to by an attribute of the object
    - Weak form of law allows sending messages to objects within a collection
- Object created by the method
- Object referred to by a global variable

Intent is to reduce coupling

- An object should avoid invoke methods of an object returned by a another method

[1] https://www2.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf

Law of Demeter Rephrased[2]

A method can call other methods in its own class
A method can call methods on its class' data members

- but not on the data member's, members

A method can call methods on its parameters
If a method creates an object, it can call methods on that object

Avoid chained messages `a.getB().getC().doSomething()`

[2]Peter Van Rooijen

Mindless Classes

A class should decide its own destiny

- restrict other classes from accessing its state
    - data members are private
    - limit accessor methods

A class with many accessor methods risks becoming *mindless*

Mindless classes tend to have low cohesion and promote high coupling

God Classes

Do everything

- within their context

Highly coupled to other classes in their context

Have low cohesion

- do many things

Cohesion & Coupling Heuristics

A class should only depend on the public interface of another class.

Attributes and their related methods, should belong to one class

- Frequently violated by classes that have many public accessor methods (`get`, `set`)

A class should represent a *single* logical concept

- Unrelated information should be factored out to another class
    - e.g. When a subset of methods operate on a proper subset of attributes

System logic should be distributed as uniformly as possible

- Classes share work fairly uniformly

Fragile Super Class

Does changing a super class affect its subclasses?

- `private` – *shouldn't*
- `protected` data – if used directly
- `public` or `protected` methods – if specification changes
  - used in subclass
    - consider `super()` as well as direct method calls
  - overridden in subclass
  - concept super class represents changes
    - no longer suitable to be super class

Another impact of coupling.

Insanity is inherited; you get it from your children.

Up & Down Calls

See `Client.java`

Summary

Classes should be highly cohesive

- single, easily understood concept

Classes should have low dependency (coupling) on each other

- reduce system complexity

Design is king

- if you don't design well, you'll end up looking like a jester