Design of Methods

Modularity, cohesion, coupling, instance methods design, tradeoffs

What's a 'good' method?

- Attributes of "a good method":
  - Broadly, a method should be easy to understand, reuse, and maintain.
  - Specifically, 1) clear intent, 2) precise pre and post conditions, 3) cohesive, 4) loosely coupled, and 5) reusable, 6) unit-testable
- Cohesion – degree of interaction within a method
  - Does the method do one thing or multiple things?
  - If the method does multiple things, how tightly are they related?
- Coupling – degree of dependency on external *info/knowledge/capability*
  - Two methods are coupled if one calls the other, or share an external variable
  - If a method uses parameters, it is coupled with external operational environment. The more parameters are used, the stronger the coupling.
  - A complex method may necessarily use more parameters and invoke more external methods to help mitigate the complexity (so coupling is not avoidable)
  - Loose coupling means use of less parameters and invoking fewer other methods
  - Key is to manage the balance between cohesion, coupling, and complexity while maintaining the method's functional significance.
- URL link: https://www.youtube.com/watch?v=Df0WVO-c3Sw&t=54s

Functional Decomposition

- Decompose a system/task into smaller systems/tasks, which are further decomposed into even smaller system/task units.
- Methods are action oriented, so should objects – ask: "what can this object do?" (not what attributes we need for this object)
- Advantages of functional decomposition to allow:
  - better readability if detail is abstracted away
  - thinking at a higher and more abstract level
  - more reusability of code (by eliminating code duplication)
  - changes to be isolated
  - self-documentation
    - public static double nthRoot(double value, int n)
    - public static Set intersect(Set s1, Set s2)
    - public static int[ ] Sort(int[ ] array, Comparator comp)
- Functional decomposition also provides opportunities for discovering polymorphic functional units when tasks become parallel or scenario-dependent, or branching out

Good Methods Start with Variable Names

- Intention-revealing method names
  - Typically, method names are verbs or verb phrases, such as *sort printStudentRecord*, or *getSize, getList*.
  - Sometimes, method names can be nouns if they refer to properties of an object, like: *size, length, firstElmt* or sound like questions like *isVisible, isOnTime* if Boolean values are returned.
- Same criteria apply to variable names:
  - *nT* is too short for "number of threads"
  - *numberOfThreadsInThisProgram* is too long
  - *numberOfThreads* or even *numThreads* is acceptable
- What if you don't seem to figure out a good name easily?
  - Is the method doing too much?
  - Is the method just a product of ad-hoc practices?

Different Levels of Cohesion

- **Levels/Categories of Cohesion on a Non-linear Scale:**

7.	Informational cohesion	(Good)
	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

1. Coincidental 2. Logical Cohesion

- A method has coincidental cohesion if it performs multiple, unrelated actions
- **Issues**
  - *not likely reusable, not maintainable*
  - Unpredictable impact going forward
  - Bad for unit tests
- **Easy to address**
  - *Break it into separate methods, integrate the pieces into other methods*, or avoid in the first place

- A method has logical cohesion when *it performs a series of actions, but only one is selected at a time by the calling module* (conditionals are present), such as:
  - runApp(userCmd)
  - draw(shapeName)
  - calculate(algorithm, input)
- **Issues**
  - Little clarity on what method does exactly
  - Tightly coupled with contextual code (do I call the method at the right place using correct arguments? – less freedom for code modifiability)
  - *Reusability is low*
  - Factory methods are of this kind, but we have less concerns because of the predictability of such methods.