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 Functional cohesion	(Good)
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.

3. Temporal Cohesion

- A method is of temporal cohesion when *it performs a series of actions related in time*
- **Example**
  - open various files, initialize data structures, read initial data (init() for card games)
  - What we typically do in a constructor.
- **Issues**
  - Actions of the module are weakly related to one another.
  - Unlikely to be reused

4. Procedural Cohesion

- A method is of procedural cohesion if *it performs a series of actions related by a procedure/algorithm to be followed by the product* (we often write such methods to provide logical clarity)
- **Example**
  - read part number and update repair record on master file
  - Read database records and update labels
  - Create panel, set panel layout, set border, add buttons, add listeners.
- **Issues:**
  - difficult to understand without a context
  - Reusability is likely low

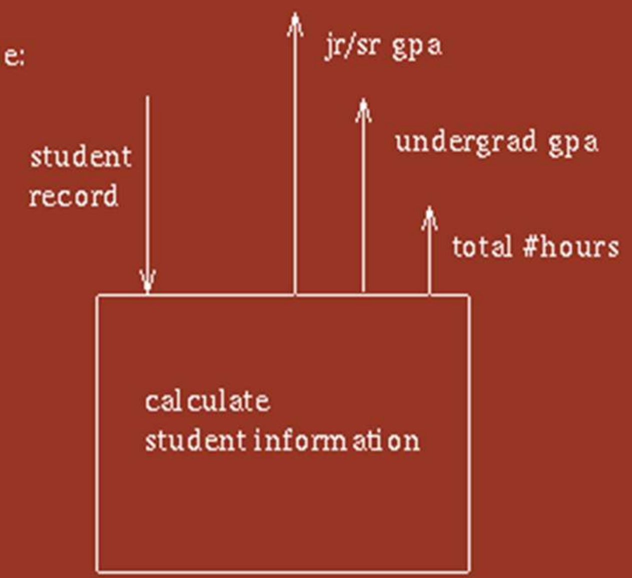
5. Communicational Cohesion

- Module performs a series of actions related by a procedure/algorithm to be followed in a process, but in addition, *all the actions operate on the same data*
- **Examples**
  - *updateAuditTrail*: update record in database and write *it* to audit trail
  - *getCurrentCoordinates*: calculate new coordinates and send *them* to terminal
- **Benefit:**
  - more likely to be reused
  - easy to understand
  - Better stability

communicational

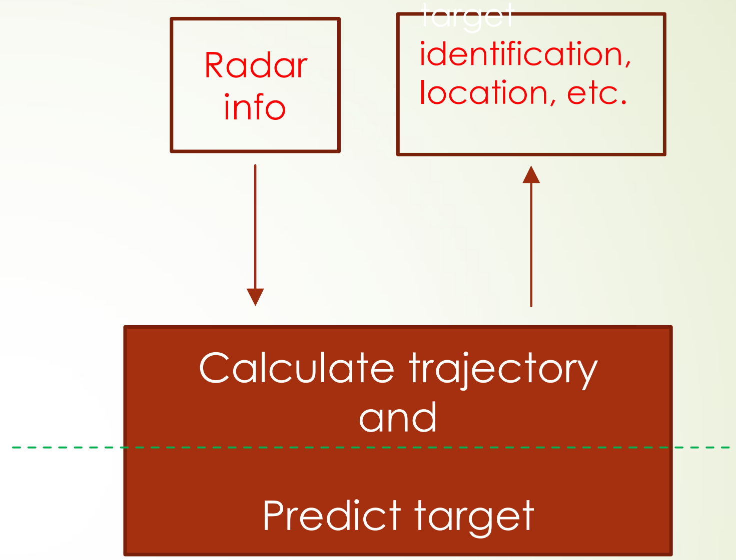
A communicationally cohesive module is one whose elements perform differnt functions, but each function references the same input information or output.

Example:



6. Informational/Sequential Cohesion

- A module has informational/sequential cohesion if it performs several actions
  - *Each has its own entry point with independent code*
  - One action uses the result of another, so, actions are unbreakable
  - All performed on the same data structure
- *Easy to understand and maintain, more likely to be reused*



This method is already in good cohesion, but we might still ask whether further decomposition along the dash line still makes good sense:

- Would trajectory calculation be useful in other situations?
- Can target be predicted with trajectory being calculated in other ways?

7. Functional Cohesion

- Module with functional cohesion *performs exactly one well-defined action*, although there may be many statements.
- **Examples**
  - get temperature of furnace
  - calculate sales commission
  - All data structures' service methods
  - Instance methods are often of this sort of cohesion… many simply change objects' states: *stu.addClass(classCode), enrollment.removeWaitList(classCode)*, …
  - Caution: *enrollment.getWaitlist()* – read-only, *checkout.processPayment(payment info)* – possibly complex

Methods Coupling

- A fact: The less a method is cohesive, the tighter the coupling would be with a calling method
  - Coincidental cohesion (ad-hoc cohesion) – would be coupled with the content of the code around it (content coupling)
  - Logic cohesion – caller controls where to call and what argument to pass (control coupling)
  - Communication cohesion – data in (arguments) and data out (what method returns); method can be treated as a black-box (data coupling)
- Two coupling situations through data sharing
  - Share external data (common coupling), which is more consequential than sharing instance data (sharing instance data is expected, but invariants should be enforced in instance methods)
  - Passing more data than needed (often due to convenience) creates coupling (stamp coupling) that can be easily avoided.

Cohesion/Coupling Examples

- Logic cohesion/control coupling
  - void doThisOrDoThat(boolean flag){

    if (flag){ …twenty lines of code to do this…}

    else {…twenty lines of code to do that… }   }
- Better?

  void doThisOrDoThat(boolean flag){

    if (flag) doThis(); else …twenty lines of code to do that… }
- Better?
  void doThisOrDoThat(boolean flag){

    if (flag) doThis(); else doThat() }

  - More analysis needed ….

```
Danger of method side effect! Common
coupling effect…

int x = 10;
int  getVal(int a){ return a + x++; }

This "apparent" equality becomes
false!
    getVal(3) == getVal(3) ?
```

```
Good methods potentially:
displayTimeOfArrival (flightNumber);
computeGrossPay (hoursWorked, payRate);
jobQueue.getJobWithHighestPriority();
```

```
methodX(){…
boolean isOk = processData(dataInfo,
"update"); …
if( !isOk ){ … }
…. }

auditTranscript(Collection students, String
studentId){ … }

void method(int arg){
  while(instanceVar == 0){
   if(arg > 25) methodX(); else mathodY();
}
```

Design Inclusion, Exclusion, and Tradeoffs

- Specificity vs. Generality: description of a method should be sufficiently specific to exclude implementations that are unacceptable but sufficiently general to allow all implementations that are acceptable.
  - pre/post conditions must serve the needs for specificity and generality
- Seeking tradeoffs: readability vs. complexity vs. efficiency
-
```
for( j = 0, j < arr.length; j++ ){
   for(k = 0; k < arr[j].length; k++)  sum += grades[j, k]; }
```

OR:
```
for(j = 0, j < courses.length; j++)  sum += courses[j].getGradesSum();
```

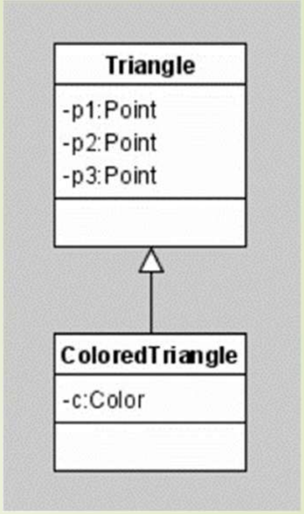
-
```
for(j = 0, j < stores.length; j++)
        change =  stores[j].getMonthlySales(month).getPercentageChange();
```

OR:
```
for(j = 0, j < stores.length; j++)
   change= stores[j].getMonthlySalesPercentageChange(month);
```

Design Robustness & Reliability

- Robustness
  - the ability of a method to weather any input data, some of which might be invalid – *Essentially, **it refers to the method's robustness and error-handling capabilities when dealing with different types of inputs***
- Reliability
  - *always producing correct results in all operational conditions*
- Robustness and reliability require precise pre and post conditions to ensure correct behavior of a method
  - Preconditions: a) operational condition (when/where/why is the method called), b) validity of the data method uses
  - Postconditions: a) operational condition (if any) when method call ends, b) What information has been altered or created after the method call ends
- How to improve robustness and reliability?
  - Mindful on the correct ranges of instance variables (class invariant)
  - Use "assert" statements to ensure preconditions (and class invariants)
  - *Good code needs fewer comments; comment only what the code doesn't tell*
  - *Deal with exceptions as opposed to throwing them*

Case Study – Override: **equals(Object)**



- Java API states: "The `equals` method implements an equivalence relation:
  - It's *reflexive*: for any `x`, `x.equals(x)` returns true.
  - It's *symmetric*: for any `x` and `y`, if `x.equals(y)` returns true, so does `y.equals(x)`.
  - It's *transitive*: for any `x`, `y`, and `z`, if `x.equals(y)` and `y.equals(z)` are both true, so is `x.equals(z)`.

- **Consider the Triangle class – two triangles are equal if their vertices are equal**
  - public boolean equals(Object obj) {

```
    if( obj == null ) return false;
    if( obj == this ) return true;
    if(obj.getClass() !=  this.getClass())
            return false;
    if(!super.equals(obj)) return false;
    ColoredTriangle otherTriangle =
        (ColoredTriangle) obj;
    return
     this.color.equals(otherTriangle.color); }
```

Behavior of "equals" in supperclass may not be maintained:

```
Triangle t1 = new Triangle(p1, p2, p3);
Triangle t2 = new ColoredTriangle(p1, p2, p3, "red");
```

`t1.equals(t2)` returns *true*, but
`t2.equals(t1)` returns *false*,
→ symmetry is violated.

Solution: 1) Override Object's equals method only in a super class, or 2) allow multiple classes in an inheritance chain to override the "equals" method; but accept the fact that the **Liskov Substitution** Principle will be violated.

Code Refactoring

- "Refactoring" is a *process of modifying working code to make it more readable, sustainable, or elegant without changing its external behavior*.
  - Most high-end IDEs have built-in support for refactoring.
- Refactoring Activities:
  - Rename a method or a variable
  - Introduce new types/interface when opportunities for polymorphism identified
  - Replace a variable with a query method
  - Extract methods into a superclass
  - Delegation of responsibilities
  - Change of visibility (of variables or methods)
- *Code refactoring can't rescue a poor design, and it can't help with structural alteration*.