Maintainability (Reloaded): guidelines for easily readable Java Source Code

A presentation by Kogias Marios
based on and inspired by
the book "Building Maintainable Software"
by Joost Visser

What is this speech about

- Sharing my inspirations with fellow java coders
- Enjoying reading our own code, many years after writing it!!!
- General guidelines I wish I was told about when I was still in the university!

Maintainability is :

- How easily a system can be modified
- Part of software quality (So is performance!)
- But can it be measured as easily as performance can be?

Why Is Maintainability Important businesswise?

Issue resolution and enhancements are twice as fast in systems with above-average maintainability than in systems with below-average maintainability!

… twice as fast …

Maintainability acts as an enabler for other quality characteristics. When a system has high maintainability, it is easier to make improvements in the other quality areas, such as fixing a security bug.

Quick Review

1. Write methods of no more than **15** LoC

2. When you are about to paste a chunk of code of at least **6** LoC, create a new function and call that function instead

3. Keep method interfaces small with No more than **4** parameters. When there are more, create an Object to pass the parameters.

1 : Write **Simple** Units of Code

“Each problem has smaller problems inside.”

—Martin Fowler

Writing Simple Units of Code :

- **Limit the number of branch points per unit to 4**.
- Do this by **splitting complex units into simpler ones** and avoiding complex units altogether.

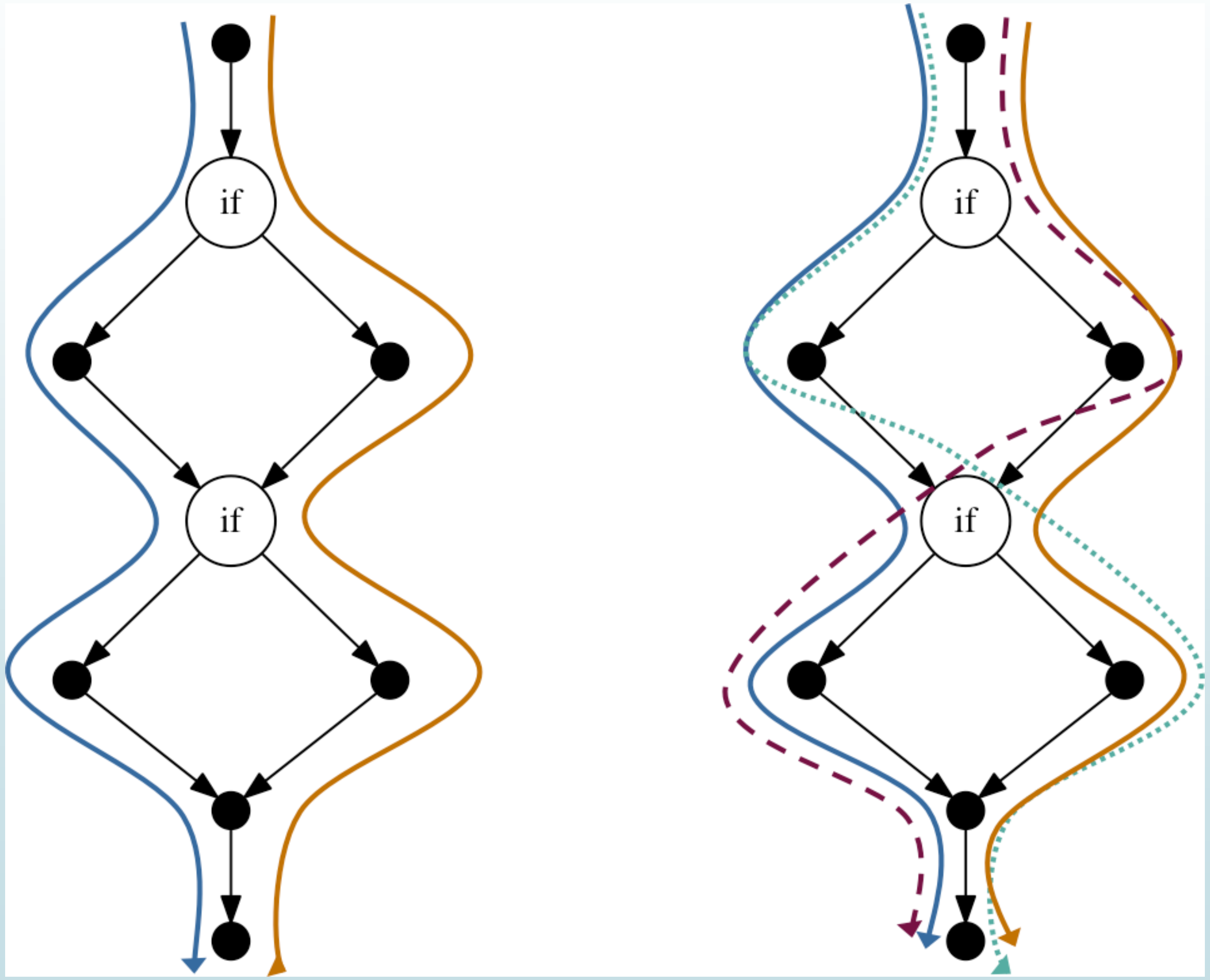
Branch Points

… is the number of possible paths through a piece of code.

A branch point is a statement where execution can take more than one direction depending on a condition (combinatory effects are possible)

The number of execution paths is a maximum, and can be very large due to combinatorial explosion.

Branch Points vs Execution Paths



Motivation

Keeping the number of branch points low **makes units :**

- Easier to Understand
- Easier to Modify
- Easier to Test

Branch point statements & operators

In Java the following statements and operators count as branch points:

- if
- case
- ?
- &&, | |
- while
- for
- catch

How to count Branch points

```java
public boolean intersects( ClusteringComparator comparator, List<ByteBuffer> minClusteringValues,
                           List<ByteBuffer> maxClusteringValues)
{
  if( start.compareTo(comparator, maxClusteringValues) > 0 ||
        end.compareTo(comparator, minClusteringValues) < 0 )
    return false;

  for(int j = 0; minClusteringValues.size() && j < maxClusteringValues.size(); j++) {
    ByteBuffer s = j < start.size() ? start.get(j) : null;
    ByteBuffer f = j < end.size()   ? end.get(j)   : null;

    if (j > 0 && (j < end.size()  && comparator.compareComponents(j, f, minClusteringValues.get(j)) < 0 ||
                  j < start.size()&& comparator.compareComponents(j, s, maxClusteringValues.get(j)) > 0))
      return false;

    if(j >= start.size() || j >= end.size() || comparator.compareComponent(j, s, f) !=0)
      break;
  }
  return true;
}
```

Slice.intersects() method from Apache Cassandra on Jan. 18th 2016

Refactoring Techniques

- Refactoring technique: Extract Method

When you have a code fragment that can be grouped together, then you can extract the fragment into a method whose name explains the purpose of the method.

Extract Method in Action

```
// User user = new User(username, password);
if ( user.exists() ) {
    if( user.lastLoginIsMoreThanAMonth() ){
        return "show_this_months_promotions";
    } else if ( user.hasMessages() ){
        return "show_messages";
    } else if (user.hasOpenOrders() ){
            return "show_open_orders";
    } else {
        return "main_page";
    }
} else {
    return "register_user";
}
```

```
public String getStartPageForExistingUser()
{
  if( user.lastLoginIsMoreThanAMonth() ){
    return "show_this_months_promotions";
  } else if ( user.hasMessages() ){
    return "show_messages";
  } else if (user.hasOpenOrders() ){
    return "show_open_orders";
  } else {
    return "main_page";
  }
}


// User user = new User(username,
password);
if ( user.exists() ) {
    return getStartPageForExistingUser();
} else {
    return "register_user";
}
```

How to rate it?

Number of branch points in methods :

- at most 4
- above 4
- above 9
- above 24

What’s new?

- Keep a method as simple as possible
- That’s common knowledge!
- Understanding that a unit with 4 branch points is a soft limit to method readability is new knowledge!

Unit Level – Module Level Guidelines

The guidelines presented so far are what we call *unit guidelines*:,

- they address improving maintainability of individual units (methods/constructors) in a system.

With the next guideline, we move up from the unit level to the module (class) level.

This module-level guideline addresses relationships between classes.

2: Separate Concerns in Modules

"In a system that is both complex and tightly coupled, accidents are inevitable."

—Charles Perrow's Normal Accidents theory in one sentence

Separate Concerns in Modules :

- **Avoid large modules in order to achieve loose coupling between them**.
- Do this by **assigning responsibilities to separate modules and hiding implementation details behind interfaces**.

What is tight coupling?

Coupling means that two parts of a system are somehow connected when changes are needed (e.g. direct calls). The code contains too much functionality and also knows implementation details about the code that surrounds it (*tightly coupled)*.

- Even when all the methods in a class comply with all the guidelines already presented, the ***combination*** of the methods in this class is what makes it tightly coupled with the classes that use it.
- Maintainability is affected by the *number of calls* to that class and the *size* of that class. Therefore, the more calls to a particular class that is tightly coupled, the smaller its size should be.

Fan-in/Fan-out

Fan-in and Fan-out relate to interactions between objects.

- High fan-in shows that an object is being used extensively by other objects, and is indicative of re-use.
- High fan-out in object-oriented design is indicated when an object must deal directly with a large number of other objects.

The problem with large modules

- The problem with these classes is that they become a maintenance hotspot.
- All functionalities related (even remotely) to the class are likely to end up in them.
- This is an example of an improper *separation of concerns*.
- Developers will also find this class increasingly more difficult to understand as it becomes large and unmanageable.
- Less experienced developers on the team will find the class intimidating and will hesitate to make changes to it.

Motivation

Small, Loosely Coupled Modules :

- Allow Developers to Work on Isolated Parts of the Codebase
- Ease Navigation Through the Codebase
- Prevent No-Go Areas for New Developers

How to Apply the Guideline

- Split Classes to Separate Concerns
- Hide Specialized Implementations Behind Interfaces
- Replace Custom Code with Third-Party Libraries/Frameworks

How to rate it?

Fan-in per module :

- 1-10
- 11-20
- 21-50
- 51+

Simplicity vs Complexity

“There are two ways of constructing a software design:

- one way is to make it so simple that there are obviously no deficiencies
- the other way is to make it so complicated that there are no obvious deficiencies. ”

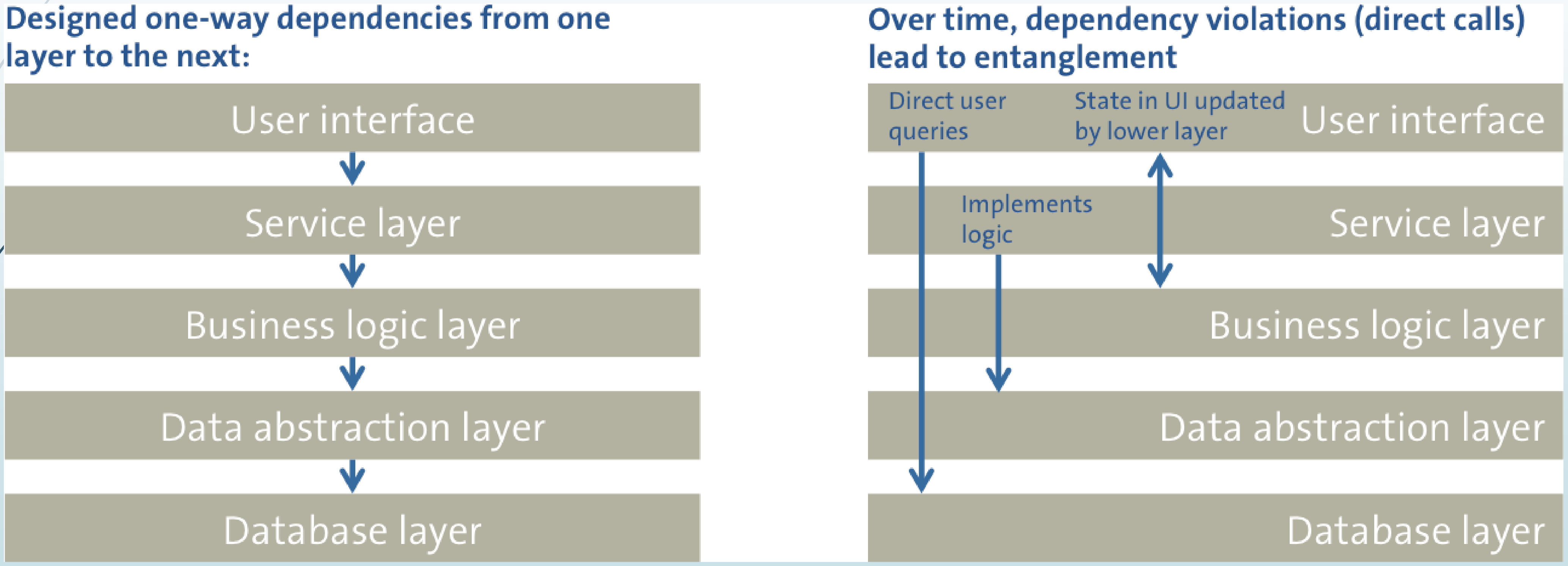
—C.A.R. Hoare.

3 : Couple Architecture Components Loosely

Couple Architecture Components Loosely:

- Achieve loose coupling between top-level components.
- Do this by minimizing the relative amount of code within modules that is exposed to (i.e., can receive calls from) modules in other components.

Component dependence



Calls that improve maintainability

***Internal calls* are healthy.**

Modules calling each other are part of the same component, they should implement closely related functionality. Their inner logic is hidden from the outside.

***Outgoing calls* are also healthy.**

As they delegate tasks to other components, they create a dependency outward.

Calls that have a negative impact on maintainability

Incoming calls should be limited

Use small interfaces to shield against direct invocations from other components (improves information hiding).

Modifying code involved in incoming dependencies potentially has a large impact on other components.

Throughput code must be avoided (high risk)

Throughput code both receives incoming calls and delegates to other components (opposite of information hiding)

Motivation

Low Component Dependence :

- Allows for Isolated Maintenance
- Separates Maintenance Responsibilities
- Eases Testing

How to Apply the Guideline

- Limit the size of modules that are the component’s interface.
- Define component interfaces on a high level of abstraction. This limits the types of requests that cross component borders and avoids requests that “know too much” about the implementation details.
- Avoid interface modules that put through calls to other components throughput code.

Design Patterns

- Abstract Factory Design Pattern
- Factory Design Pattern
- Dependency Injection (DI)

Dependency Injection

Classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing.

Dependency Injection (wiring) helps in gluing these classes together and same time keeping them independent.

Dependency Injection in Action

```java
public class Employee {
  private Department dept;

  public Employee() {
    dept = new Department();
  }
}
```

```java
public class Employee {
  private Department dept;

  public Employee (Department dept) {
    this.dept = dept;
  }
}
```

How to rate it

Component independence is measured as :

the percentage of code that is classified as *hidden code*.

Wrap up

Topics for discussion

- Do you already apply any of the presented guidelines?
- If not, would these guidelines prove useful?
- Would you apply these guidelines in your daily routines?
- What are the maintainability guidelines that you follow?

Bibliography

- Visser, Joost. *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. Reading, MA: O'Reilly, February 2016.

- Fowler, Martin. *Refactoring Improving the Design of Existing Code*. Reading, MA: Addison Wesley, 1999.

- Martin, Robert. Clean Code: A Handbook of Agile Software Craftsmanship. Reading, MA: Prentice Hall, August 2008.

- Feathers, Michael. Working Effectively with Legacy Code. Reading, MA: Prentice Hall, October 2004

Thank you

Q&A