

# The starting 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





## Some info about the speaker

- Started as a Java developer (awt, struts/jsf/J2EE, J2ME, android) for 6 years
- Migrated to an Application Server Administrator (Tomcat, GlassFish, OAS, WebLogic, WebSphere), a Database Administrator (Oracle, MySQL) and \*nix Administrator (Linux, AIX, HPUX) for more than 8 years
- And always strive to better myself



# 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

## **Servicing & Repairs**



-  **Brakes**
-  **Tyres**
-  **Exhausts**
-  **Clutches**
-  **Gearboxes**



# 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?



# Types of Software Maintenance

- Bugs (*corrective maintenance*)
- Adapt to changes (*adaptive maintenance*)
- New or changed requirements (*perfective maintenance*)
- Increase quality or prevent future bugs from occurring (*preventive maintenance*)



# 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.

- ONE -

**is bigger  
better?**



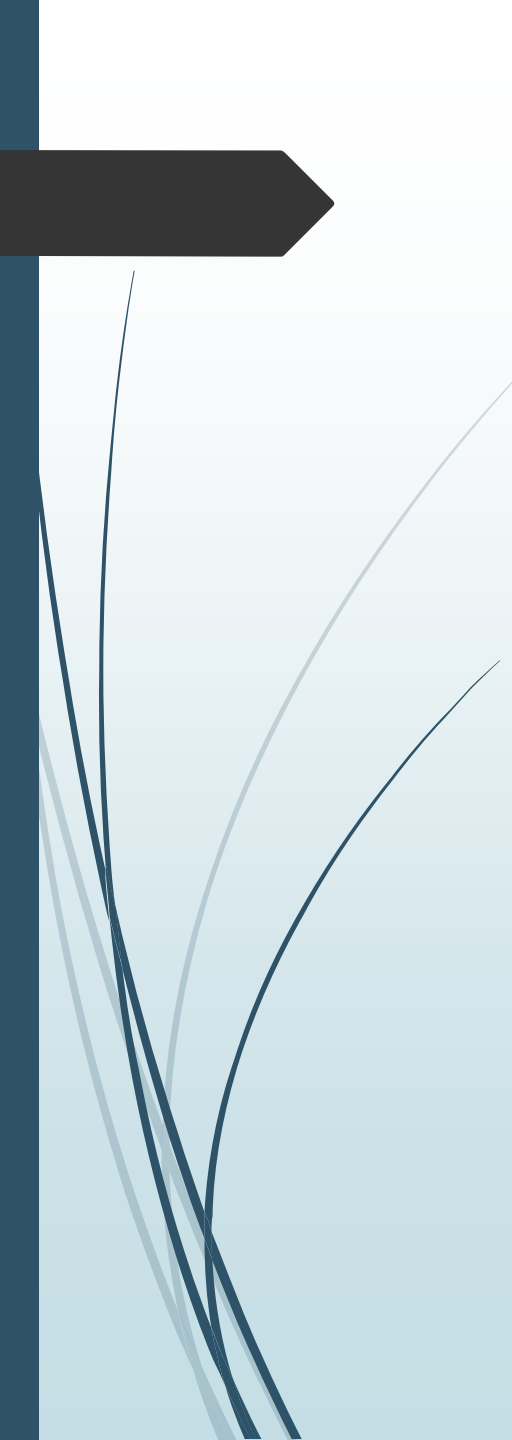


# 1 : Methods should not be bigger than 15 LoC

*“Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand.”*  
—Martin Fowler

Writing Short Units of Code :

- Limit the length of code units to 15 lines of code.
- Do this by not writing units that are longer than 15 lines of code in the first place, or by splitting long units into multiple smaller units until each unit has at most 15 lines of code.



# Defining LoC (=Lines of Code)

A simple look at *lines of code* (LoC) is: any line in source code that ends with an Enter/Return, and neither is empty nor contains only a comment.

A line may have more than one statement.

And a statement can be spread over multiple lines.

Keep the rule: One statement per line



# Motivation

- Short Units Are Easy to Understand
- Short Units Are Easy to Analyze
- Short Units Are Easy to Test
- Short Units Are Easy to Reuse

# 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

```
// Car car = new Car();  
openDriversDoor();  
sitInDriversSit();  
closeDriversDoor();  
putKeyInIgnitionSlot();  
startEngine();  
  
goto( ... );  
  
comeToAFullStop();  
pullHandBrake();  
stopEngine();  
getKeyFromEgnitionSlot();  
openDriversDoor();  
getOutOfTheCar();  
closeDriversDoor();
```

```
getInCarAndStartEngine();  
goto( ... );  
stopCarAndGetOff();
```





## How to rate it?

LoC per method

Categorization:






1-15 LoC

16-30 LoC

31-60 LoC

61+ LoC

# Rating Maintainability

Rating	Maintainability
	Top 5% of the systems in the benchmark
	Next 30% of the systems in the benchmark (above-average systems)
	Next 30% of the systems in the benchmark (average systems)
	Next 30% of the systems in the benchmark (below-average systems)
	Bottom 5% least maintainable systems

Star ratings serve as ***predictors*** for actual system maintainability :

- ➡ Issue resolution and enhancements are ***twice as fast*** in systems with 4 stars than in systems with 2 stars.



## Minimum thresholds for a ★★★★★ unit size rating

... more than 60 lines of code: At most 6.9%

... more than 30 lines of code: At most 22.3%

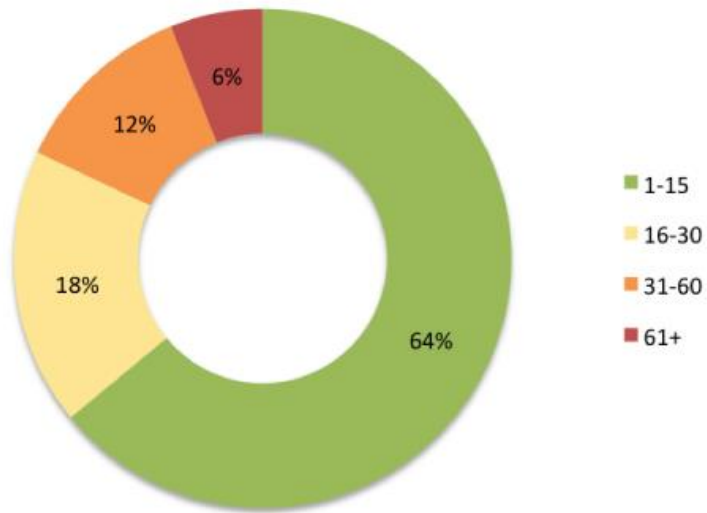
... more than 15 lines of code: At most 43.7%

... at most 15 lines of code : At least 56.3%

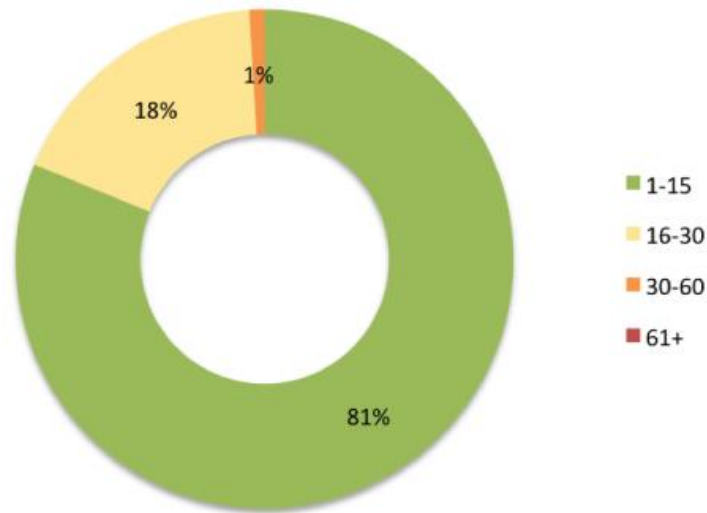


# Three quality profiles for unit size

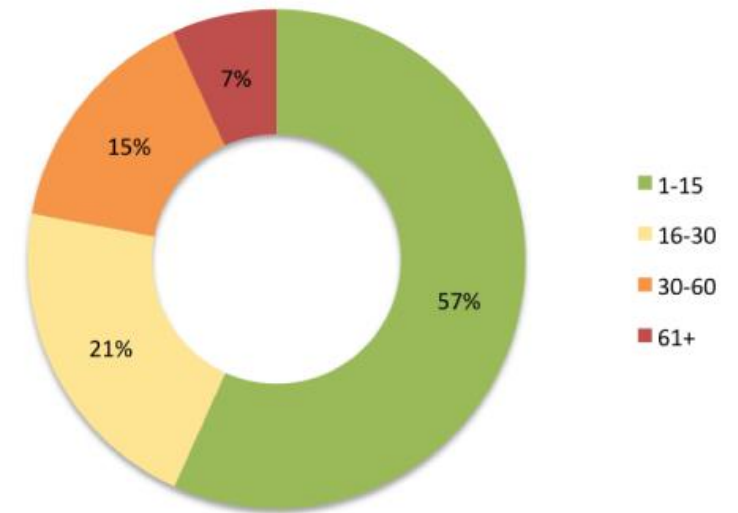
Unit size quality profile of Jenkins



Unit size quality profile of an anonymous ★★★★★ system



Unit size quality profile cut-offs for a ★★★★★ system

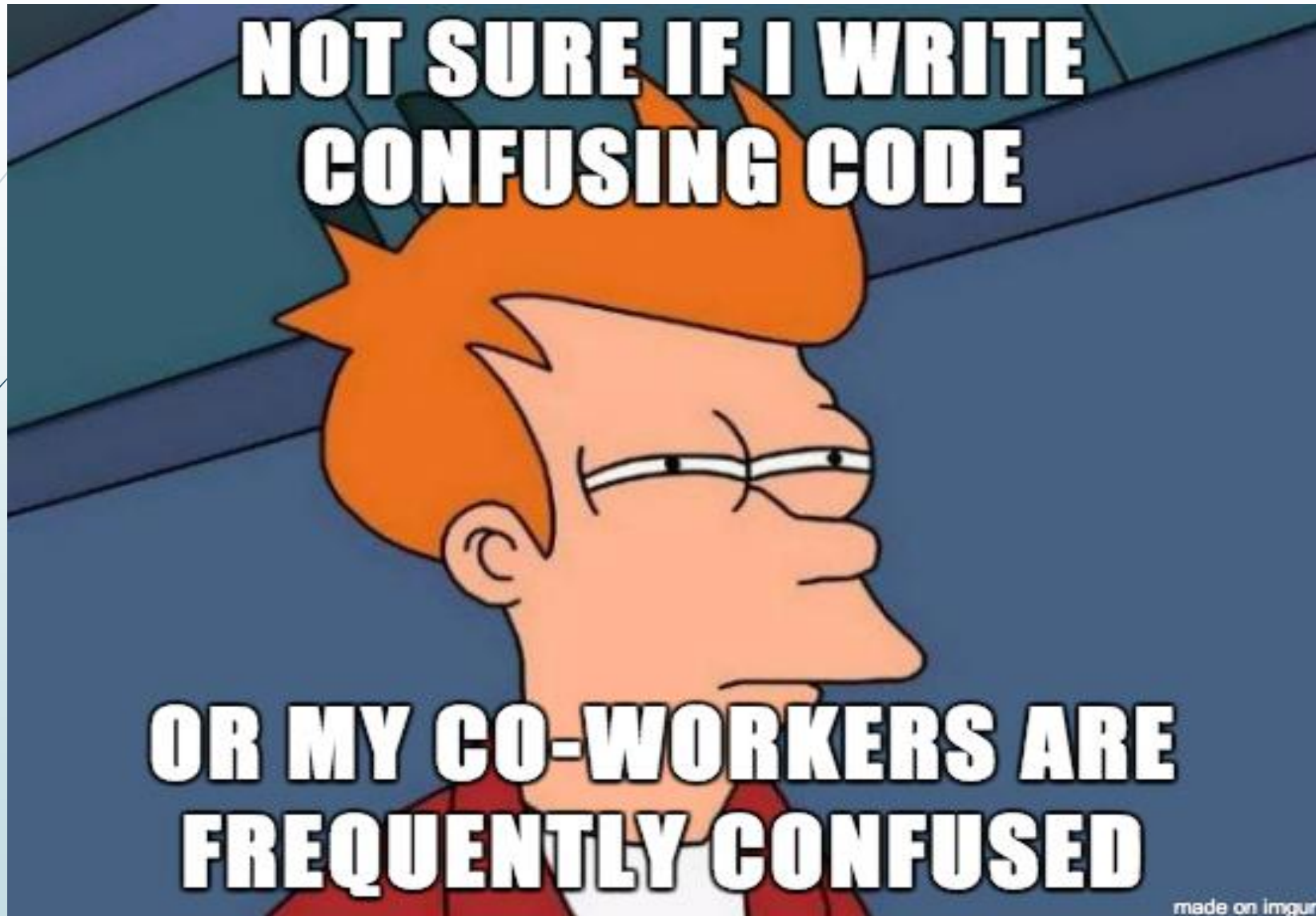




## What's new?

- Keep a method as short as possible
- That's common knowledge!
- Understanding that 15 LoC is a soft limit to method readability is new knowledge!

- TWO -





## 2: Create a new function for each duplicate chunk of code of at least 6 LoC

*“Number one in the stink parade is duplicated code.”*

Writing Code Once :

- Do not copy code.
- Do this by writing reusable, generic code and/or calling existing methods instead.



# Clone Types

## **Type 1 clones:**

*A duplicate or code clone is an identical piece of code at least 6 lines long.*

## **Type 2 clones:**

Two fragments of code that are syntactically the same.



# Motivation

- Duplicated Code Is Harder to Analyze
- Duplicated Code Is Harder to Modify
- ... **when code is copied, bugs need to be fixed at multiple places**, which is inefficient and error-prone.

# How to Apply the Guideline

“**Extract Method**” refactoring technique (Already mentioned)

In case the extracted fragment had to be put in a third class, that class runs the risk of containing a bunch of unrelated methods (a signal of multiple unrelated functionalities within the class).



Then use the “**Extract Superclass**” *Refactoring Technique*

- extracts a fragment of code lines not just to a method, but to a new class that is the superclass of the original class

# Extract SuperClass Mechanic



Create a blank abstract superclass, make the original classes subclasses of this superclass.

Move common elements to the superclass.

Examine the methods left on the subclasses.

After pulling up all the common elements, check each client of the subclasses.

If they use only the common interface you can change the required type to the superclass.



# Extract Superclass in Action

```
public class Dog {  
  
    public void sleep() {  
    }  
  
    public void bark() {  
    }  
}
```



```
public class Animal {  
    public void sleep() {  
    }  
}  
  
public class Dog extends Animal {  
    public void bark() {  
    }  
}
```



## How to rate it?

Split code into two major categories:

- ➡ Non-redundant code
- ➡ Redundant code



## Minimum thresholds for a duplication rating

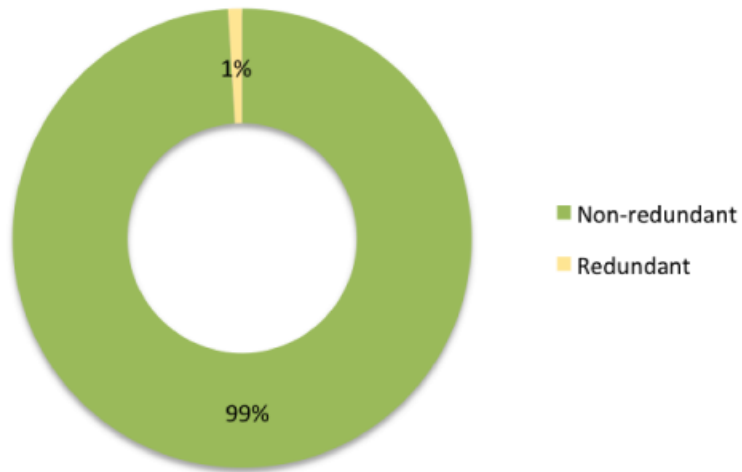


... non-redundant At least 95.4%

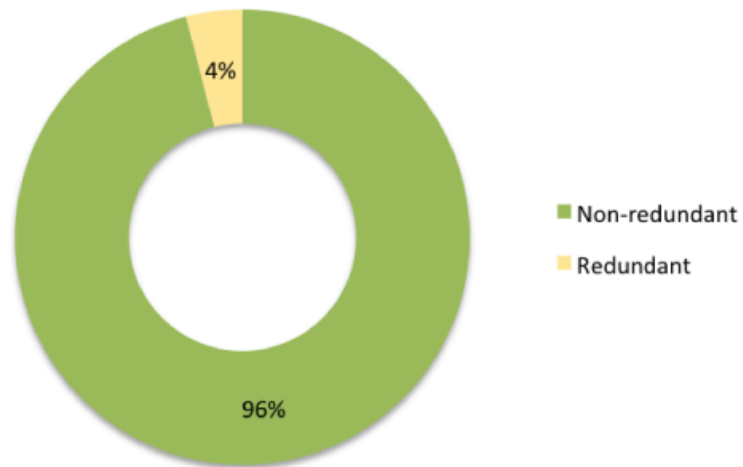
... redundant At most 4.6%

# Three code duplication quality profiles

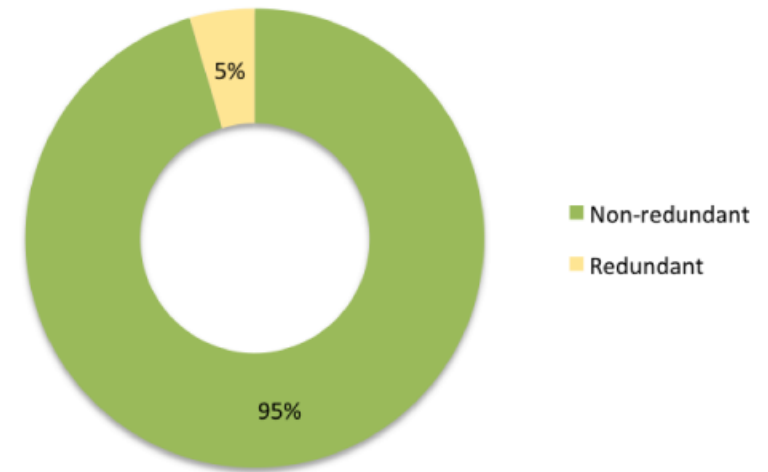
Code duplication quality profile in Jenkins



Code duplication quality profile of an anonymous ★★★★★ system



Code duplication quality profile cut-offs for a ★★★★★ system





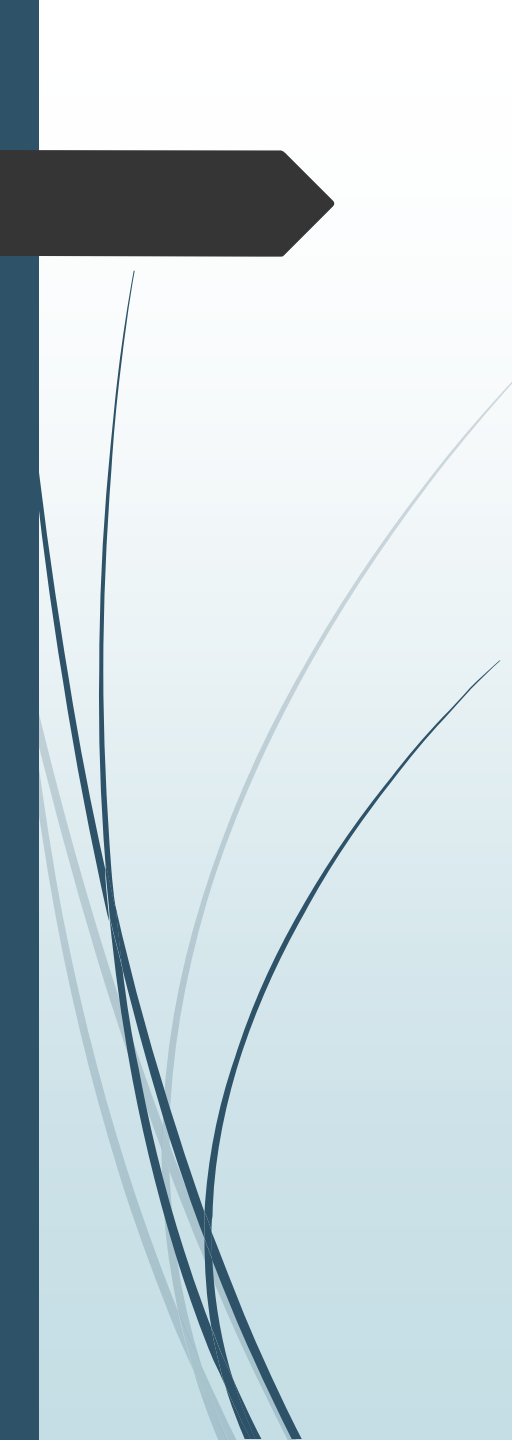
# What's new?

- Copy/paste code is bad.
- That's common knowledge!
- Knowing that a chunk of duplicated code of at least 6 LoC is the soft limit to refactoring is new knowledge!

- THREE -



**Less is more**



## 3 : Methods should have NO more than **4** parameters

*“Bunches of data that hang around together really ought to be made into their own object.”*

Keeping Unit Interfaces Small :

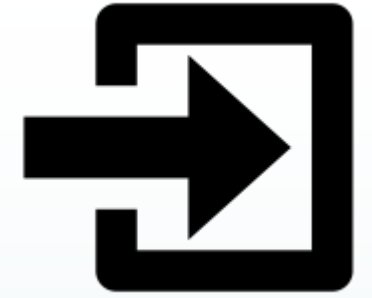
- Limit the number of parameters per unit to at most 4.
- Do this by extracting parameters into objects.



# Motivation

- Small Interfaces Are Easier to Understand and Reuse
- Methods with Small Interfaces Are Easier to Modify





# Data Transfer Objects

## Parameter Objects

Objects that group parameters. Usually represent meaningful concepts from the domain.

A point, a width, and a height is a

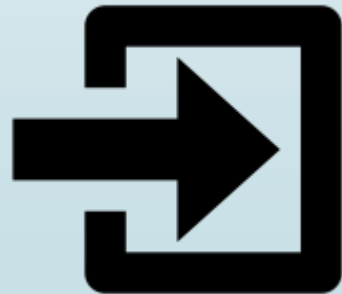
- class Rectangle

A first name, a last name, and an address is a :

- class MailAddress

# Introduce Parameter Object in Action

```
public void printEnvelope(  
    String name,  
    String postalCode,  
    String fullAddress,  
    String town,  
    String country ){  
    //Use parameters to print  
}
```



```
public class Address(){  
    String name;  
    String postalCode;  
    String fullAddress;  
    String town;  
    String country;  
    //Getters - Setters  
}
```


And our function declaration becomes :

```
public void printEnvelope(Address address){  
    //Use Parameter Object to print  
}
```



# How to rate it

Categorize by Number of parameters per method :

- 1-2 parameters
  - 3-4 parameters
  - 5-6 parameters
  - 7+ parameters
- 

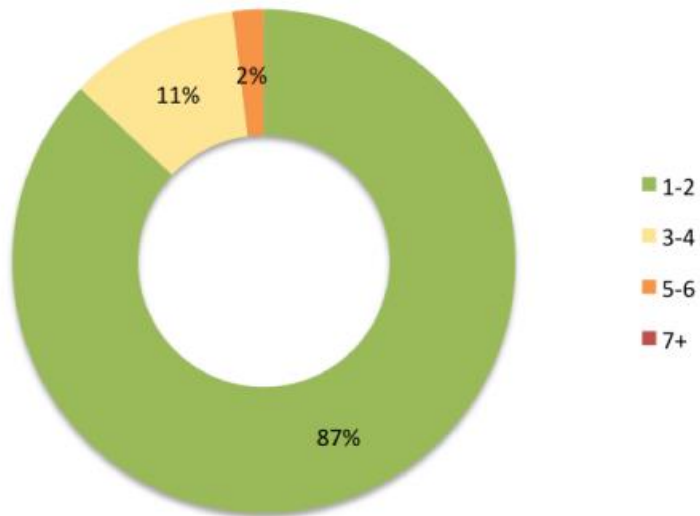


## Minimum thresholds for a ★★★★★ unit size rating

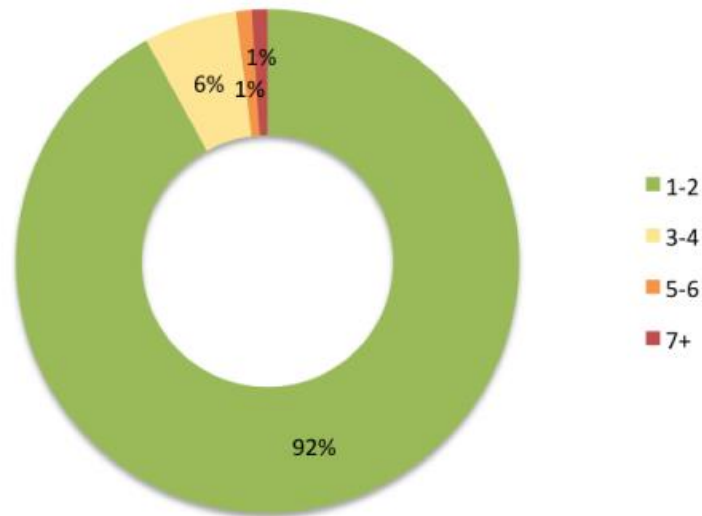
... more than 7 parameters	At most 0.7%
... 5 or more parameters	At most 2.7%
... 3 or more parameters	At most 13.8%
... at most 2 parameters	At least 86.2%

# Three quality profiles for unit interfacing

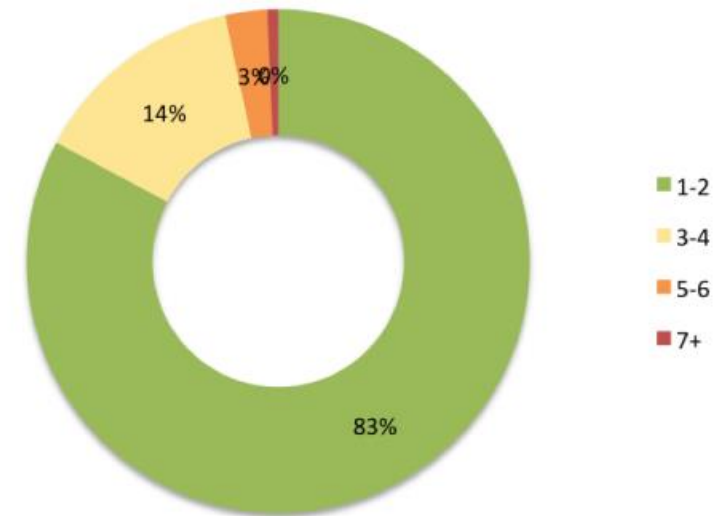
Unit interfacing quality profile for Jenkins



Unit interfacing quality profile of an anonymous ★★★★★ system



Unit interfacing quality profile cut-offs for a ★★★★★ system





## What's new?

- Having a large number of parameters passed to a method makes the function illegible!
- That's common knowledge
- Knowing that 4 parameters is the soft limit before starting thinking about refactoring, is new knowledge!





## There is more to maintainability...

- There are 10 guidelines in total to be learned and many more refactoring techniques to be learned
- There is a certificate :*“Quality Software Developer Foundation Certificate in Maintainability”*
- *There are software assessments measuring the maintainability of a java software project objective metrics (e.g. BetterCodeHub)*



# Wrap up



# Easy as One-Two-Three

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.



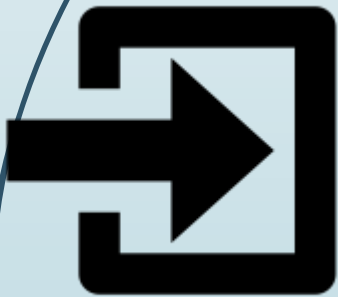
# Easy as One-Two-Three (refactoring version!)



1. Extract Method



2. Extract Super Class



3. Introduce Parameter Object



## Some tips

- There is usually not the luxury to start applying the guidelines of maintainability to an existing project
- One can apply the guidelines, whenever a need for modifying the codebase occurs.
- So, the code can be sanitized, little by little.



## More simple tips

- Don't put more than one statement on a line
- Use meaningful names that convey the purpose of the variable
- Be consistent about it

*[From <http://introcs.cs.princeton.edu/java/11style/>]*



## Tools : BCH (Beta)

► BetterCodeHub (BCH): currently in beta  
*<https://bettercodehub.com/> (github plugin)*

Compliance to guidelines is derived from the Software Improvement Group's industry benchmark, which consists of over 8 billion lines of code in more than 170 different technologies.



## Tools: PMD (eclipse plugin)

Scans Java source code for potential problems :

- Possible bugs
- Dead code
- Suboptimal code
- Overcomplicated expressions
- Duplicate code



# Q&A







# 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

## Books

- 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.

Thank  
you

