

## Appendix B

# Survey Questions

### Page 1: Introduction to Survey

## Refactoring Effort Survey

Hello!

I am Kirsten Gericke, a GreenIT Research Intern at SIG from UvA's Master Software Engineering.

My research defines the concept "Greenability" as a new Sigrid Software Quality that determines the ability of software to become more sustainable by reducing energy consumption in the future. Greenability focuses on the **process** of making software more sustainable, rather than its current energy consumption. This means we focus on how software quality metrics (maintainability, reliability, etc.) impact **effort** to refactor a code base.

In this survey I will present 6 different common refactoring patterns. For each pattern, I will give a simple example, and ask your opinion based on your professional working experience, on how much **effort** would be required to implement the refactoring, given various Software Quality characteristics.

*Disclaimer:*  
*Your responses to this survey will be kept strictly confidential and anonymous. Personal data is not required and collected answers will be used for the purposes of my thesis. By submitting your responses, you consent to the analysis and reporting of the data derived from your responses.*

Thank you for your time and cooperation!

kirstykromhout7@gmail.com [Switch accounts](#)

 Not shared

Next

Page 1 of 10

[Clear form](#)

Figure B.1: Introduction to Survey

**Page 2: Basic Information about Participant**

**Basic Information**

How familiar are you with SIG's quality models?

	1	2	3	4	5	
Not at all	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Very Familiar

How many years have you been working in IT?

☐ <1 year

☐ 1-5 years

☐ 5-10 years

☐ 10+ years

Do your daily responsibilities include writing, reviewing, and/or analyzing source code?

☐ Yes

☐ No

Back

Next

Page 2 of 10

Clear form

**Figure B.2: Questions about participant**

## Page 3: Task Description

### Task Description

Imagine having a codebase where you must apply a refactoring in **every** relevant instance.

There are 6 refactorings:

1. Convert Local Variable to Field
2. Extract Local Variable
3. Extract Method
4. Introduce Indirection
5. Inline Method
6. Introduce Parameter Object

There are 5 System Qualities:

1. Maintainability
2. Reliability
3. Architecture Quality
4. Freshness
5. Performance Efficiency

You will be asked for each refactoring, whether you think a higher score for a System Quality will make the **process** of implementing the refactoring require more or less **effort**. If you are unsure of an answer, you can leave it blank.

Effort can be interpreted as time, lines of code added/removed/changed, difficulty of understanding the source code, etc.

[Back](#)[Next](#)

Page 3 of 10

[Clear form](#)

Figure B.3: Task Description

## Page 4: Questions for Refactoring 1

### Refactoring 1: Convert Local Variable to Field

#### Convert Local Variable to Field

This refactoring creates a new field by turning a local variable into a field.

**Before**

```
public class MyClass {  
  
    public void myMethod() {  
        int myLocalVariable = 10;  
        System.out.println(myLocalVariable);  
    }  
}
```

**After**

```
public class MyClass {  
    private int myField; // Converted from local variable  
  
    public void myMethod() {  
        myField = 10;  
        System.out.println(myField);  
    }  
}
```

Figure B.4: Refactoring 1: Name, Description and Example

#### Task

You are presented with two systems and their characteristics for various metrics. Imagine applying this refactoring to **every** relevant instance in both systems.

Figure B.5: Task clarification

**Maintainability**

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Volume	10K LOC	50K LOC
Duplication	No duplicated lines	40% of the codebase is duplicated.
Unit Size	Average method size: 15 lines	Average method size: 50 lines
Unit Complexity	Average McCabe Complexity: 3	Average McCabe Complexity: 10
Module Coupling	Few parameters per method	Many parameters per method
Component Independence	Small interface	Many incoming calls to interface
Component Entanglement	Minimal component communication	High component communication

1                  2                  3                  4                  5

Less Effort      ☐      ☐      ☐      ☐      ☐      More Effort

Figure B.6: Maintainability Question and Example

**Reliability**

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Inter-process Communication	Effective design patterns	Inefficient communication
Logic & Data Flow	Correct boolean logic	Contradicting if statements
Memory Allocation & Release	Properly allocated and freed memory	Memory leaks
Concurrency & Synchronization	Well-designed concurrency	Deadlocks, poor synchronization
Pointers	Proper pointer handling	Frequent null pointer exceptions
Resource Management	Scalable under load	Inefficient, poor scalability
Hard-coded Configuration	Flexible configurations used	Hard-coded configurations
Error Handling	Proper use of try-catch	Frequent unhandled exceptions
Arithmetic Operations	Correct precedence	Integer overflows, division by zero
Dead & Irrelevant Code	Clean and relevant codebase	Many dead code sections

1                  2                  3                  4                  5

Less Effort      ☐      ☐      ☐      ☐      ☐      More Effort

Figure B.7: Reliability Question and Example

**Architecture Quality**

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Code Breakdown	Classes < 200 lines Methods < 20 lines	Classes > 1000 lines Methods > 100 lines
Component Coupling	Few component dependencies	Many component dependencies
Technology Prevalence	Uses Java 11, Spring Boot 3.0	Uses Java 6, outdated frameworks
Component Cohesion	Follows single responsibility principle	Many responsibilities per component
Code Reuse	No duplication across modules	Duplicate code across modules
Communication Centralization	80% internal centralized code	30% internal code, scattered
Data Coupling	Components use their own databases	Components share a single database
Bounded Evolution	10% co-evolution, changes localized	One change impacts entire code base
Knowledge Distribution	Knowledge spread across 5 people	Concentrated in 1-2 people
Component Freshness	Updated monthly with latest patches	Updated irregularly, every 2-3 years

1                  2                  3                  4                  5

Less Effort      ☐      ☐      ☐      ☐      ☐      More Effort

Figure B.8: Architecture Quality Question and Example

**Freshness**

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Freshness Risk	Dependencies updated within the last 6 months	Dependencies not updated for over 2 years
Component Freshness	Components updated within the last month	Components not updated for over a year
Technology Prevalence	Uses the latest stable versions of technologies	Uses outdated versions of technologies

1                  2                  3                  4                  5

Less Effort      ☐      ☐      ☐      ☐      ☐      More Effort

Figure B.9: Freshness Question and Example

**Performance Efficiency**

Would implementing this refactoring in **System A** require more or less effort than in **System B**?

Metrics	System A	System B
Time Behavior	Response time < 100ms Handles 1000 requests per second	Response time > 500ms Handles 100 requests per second
Capacity	Can add 1000 more users without affecting performance	Adding 100 more users causes significant slowdowns
Resource Utilization	CPU usage < 50% RAM usage < 4GB under load	CPU usage > 90% RAM usage > 8GB under load

  

1

Less Effort

2

☐

3

☐

4

☐

5

☐

More Effort

Figure B.10: Performance Efficiency Question and Example

## Page 5: Questions for Refactoring 2

From this point on, the property questions and examples questions were identical for all refactorings, and are not repeated to reduce unnecessary space in this report being used.

### Refactoring 2: Extract Local Variable

#### Extract Local Variable

Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.

##### Before

```
public class MyClass {
    public void myMethod() {
        int result = 2 * (3 + 5);
        System.out.println(result);
    }
}
```

##### After

```
public class MyClass {
    public void myMethod() {
        int sum = 3 + 5; // Extracted local variable
        int result = 2 * sum;
        System.out.println(result);
    }
}
```

Figure B.11: Refactoring 2: Name, Description and Example

## Page 6: Questions for Refactoring 3

### Refactoring 3: Extract Method

#### Extract Method

Creates a new method containing the currently selected statement or expression and replaces the selection with a reference to the new method.

**Before**

```
public class MyClass {  
    public void myMethod() {  
        int result = 2 * (3 + 5);  
        System.out.println(result);  
    }  
}
```

**After**

```
public class MyClass {  
    public void myMethod() {  
        int result = calculateResult(); // Extracted method  
        System.out.println(result);  
    }  
  
    private int calculateResult() {  
        return 2 * (3 + 5);  
    }  
}
```

Figure B.12: Refactoring 3: Name, Description and Example



## Page 7: Questions for Refactoring 4

### Refactoring 4: Introduce Indirection

#### Introduce Indirection

Creates a static method that can be used to indirectly delegate to the selected method.

**Before**

```
public class MyClass {
    public void myMethod() {
        originalMethod();
    }

    public void originalMethod() {
        System.out.println("Original method");
    }
}
```

**After**

```
public class MyClass {
    public void myMethod() {
        indirectMethod(); // Introduced indirection
    }

    public static void indirectMethod() {
        MyClass myClass = new MyClass();
        myClass.originalMethod();
    }

    public void originalMethod() {
        System.out.println("Original method");
    }
}
```

Figure B.13: Refactoring 4: Name, Description and Example

## Page 8: Questions for Refactoring 5

### Refactoring 5: Inline Method

#### Inline Method

Copies the body of a callee method into the body of a caller method.

**Before**

```
public class MyClass {
    public void myMethod() {
        printMessage();
    }

    public void printMessage() {
        System.out.println("Hello, World!");
    }
}
```

**After**

```
public class MyClass {
    public void myMethod() {
        System.out.println("Hello, World!"); // Inlined method
    }
}
```

Figure B.14: Refactoring 5: Name, Description and Example

## Page 9: Questions for Refactoring 6

### Refactoring 6: Introduce Parameter Object

#### Introduce Parameter Object

Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduced parameter.

**Before**

```
public class MyClass {  
    public void myMethod(String firstName, String lastName) {  
        System.out.println(firstName + " " + lastName);  
    }  
}
```

**After**

```
public class MyClass {  
    public void myMethod(Person person) { // Introduced parameter object  
        System.out.println(person.getFirstName() + " " + person.getLastName());  
    }  
}  
  
class Person {  
    private String firstName;  
    private String lastName;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
}
```

Figure B.15: Refactoring 6: Name, Description and Example

## Page 10: Thank you and Additional Questions

### Thank you for your participation!

Do you have any additional comments?

Your answer

Back

Submit

Page 10 of 10

Clear form

Figure B.16: Final Thanks and Additional Questions