Bachelor Thesis

February 1, 2018

Generating Documentation to Reflect Side Effects of Methods

Sandro Wirth

of Bachs, Switzerland (14-704-308)

supervised by

Prof. Dr. Harald C. Gall

Jürgen Cito & Gerald Schermann

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Generating Documentation to Reflect Side Effects of Methods

Sandro Wirth

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Author: Sandro Wirth, sandro.wirth@uzh.ch

Project period: August 10, 2017 - February 1, 2018

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank all the people who were involved in this thesis and made it possible. First and foremost, I have to thank my advisors Dr. Jürgen Cito and Gerald Schermann for giving me the opportunity to write this thesis and for their assistance and dedicated involvement throughout the process. I would also like to thank Dr. Stefan Hanenberg for providing us with his expertise and helping designing the case study. Huge thanks also to everyone that participated in the study.

I also would like to use this opportunity to thank my team leader and collegues at work for their understanding and for allowing me to work flexibly and take the time I needed to work on this thesis.

Last but not least I want to thank my family and my girlfriend for supporting me everyday throughout this process. I am eternally grateful.

Abstract

Studies have shown that reducing side effects in software projects has a variety of advantages. It has a positive impact on testability, code comprehension, verifiability or performing refactorings. Although there are existing studies about detecting purity and side effects there is no study about

This thesis presents an approach on automatically showing the purity and side effects of Java methods during the workflow of reading and writing code to evaluate a positive influence on code comprehension. It is based on the implementation of a prototype that uses purity information from existing tools and transforms the information into readable Javadoc.

To measure the influence on code comprehension, this thesis presents an evaluation of the implemented prototype by executing a quantitative empirical study with 14 participants. The results show that having the information can improve code comprehension for certain cases but also small details can lead to possible negative influences.

Zusammenfassung

Studien haben gezeigt, dass die Reduktion von Seiteneffekte in Software Projekten eine Vielzahl an Vorteilen mit sich bringt. Es hat einen positiven Einfluss auf beispielsweise die Testbarkeit, das Code-Verständnis, die Verifizierbarkeit oder das Durchführen von Refactorings.

Diese Arbeit präsentiert einen Ansatz, um automatisch das Vorhandensein beziehungsweise das Nicht-Vorhandensein von Seiteneffekten in Java Methoden während dem Lesen und Schreiben von Code anzuzeigen, mit dem Ziel zu prüfen, ob diese Information einen positiven Einfluss auf das Code Verständnis hat. Dazu wurde ein Prototyp entwickelt, der die Informationen über Seiteneffekten von existierenden Tools verwendet, verarbeitet und in lesbare Dokumentation umwandelt in der Form von Javadoc.

Um den Einfluss auf das Code Verständnis zu messen, präsentiert diese Arbeit eine Evaluation des Prototypes anhand einer quantitativen, empirischen Studie mit 14 Teilnehmern. Die Resultate zeigen, dass das Vorhandensein dieser Information das Code Verständnis in gewissen Fällen erhöhen können, wobei kleine Details auch das Gegenteil bewirken können.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Code comprehension is a crucial task for software developers. It describes the process of reading and understanding of existing code which is particularly important when reusing existing or unknown software components. Methods in object-oriented languages often update the objects that they access which is by definition a side effect. Other forms of side effects include the modification of class variables (static and non-static), I/O operations (e.g. file or database access) or raising exceptions. In general, side effects of methods are state changes that can be observed by code that invokes the method [Rou04]. Consider the example shown in Listing 1.1 where the methods modifies the state the given parameter. The presence of such effects can make it more difficult to understand the data flow of a program and they also are potential sources for bugs. Another term for methods that cause side effects is calling them *impure*. The opposite of this are *pure* methods which is shown in Listing 1.2. Such methods do not cause any observable side effect and always evaluate to the same result given the same argument values. Pure methods offer useful advantages such as better understanding and analysis, verification in model checking or better testability. Furthermore, pure methods are also easier to parallelise.

This thesis presents an approach on exposing information about the purity and potential side effects of methods to developers during the task of code comprehension. It is based on the implementation of a prototype that uses information coming from a modified version of an existing tool for purity and side effects analysis and creates documentation that is visible to a developer. Moreover, this thesis presents an empirical study that investigates the impact that such documentation has on the process of code comprehension.

```
public void impure(Rectangle rectangle) {
    rectangle.height = 100;
}
```

Listing 1.1: Impure method

```
public int pure(int x) {
    return x * x;
}
```

Listing 1.2: Pure method

1.1 Problem Statement and Research Question

Names and documentation of methods usually focus on the intention on what these methods are required to do and produce as a result but they lack the information about possible side effects that can occur during the accomplishment of their task [YHHK15]. The absence of such information makes it more difficult to reuse existing components as developers do not know about possible side effects. Existing studies (see Chapter 2) have shown techniques to detect side effects and purity and these ideas have been implemented into different tools. These ideas have mainly been used for optimizing compilers and not to make the information about side effects and purity available for developers. This thesis aims to find a way of providing such information to developers and moreover investigates the following research question:

(i) Does information about the purity and possible side effects of methods improve code comprehension?

Our approach is to use existing techniques and tools for the purity and side effect analysis and generate a documentation based on these information that is available to developers during the process of reading and writing code. A similar thought has been expressed by Rountev [Rou04] where he states that the results of his work could be used to automatically add documentation to the software components.

1.2 Structure

Chapter 2 discusses related work about side effects analysis and empirical studies in software engineering. In Chapter 3 we present our approach on showing information about the purity and side effects directly in the source code. The chapter is divided into two parts where the first part focuses the on actual purity analysis. The second part presents the implementation of our own tool that is responsible for generating the actual documentation out of the results from the purity analysis. Chapter 4 is dedicated to the empirical study we conducted to evaluate the impacts our tool can have on code comprehension of developers. In Chapter 5 a conclusion is made based on the results and the personal experience. To finish, we propose ideas for further research and enhancement of the application.

Chapter 2

Related Work

2.1 Side Effects and Purity Analysis

Studies about techniques to detect side effects and purity in software have been around for many years. One of the first methods was introduced by Banning [Ban79] where he presented a flow-insensitive interprocedural alias analysis.

Based on the ideas of Banning, Cooper and Kennedy [CK88] presented an improved method that solves the problem Banning stated in linear time by employing a new data structure and breaking down the problem into two subproblems. Further improvements of these ideas as well as new algorithms were introduced by Choi et al. [CBC93]. Their work included a new algorithm for flow-sensitive alias analysis which is more precise and efficient. They also extended the traditional flow-insensitive alias analysis to correctly compute side effects in the presence of pointers as this could not be handled by then existing methods for side effects analysis.

One of the first studies about using side effects analysis to optimize Java bytecode was introduced by Clausen [Cla97]. The tool presented in his work implements dead-code elimination and loop-invariant removal. Based on this work Razafimahefa [Raz99] presents an implementation of a loop-invariant removal by using the `Soot` Framework[1]. A different form of side-effects analysis for Java has been introduced by Milanova et al. [MRR02]. Their approach is based on an object-sensitive points-to analysis that is faster than a context-insensitive analysis and also has an improved precision of side-effect analysis.

One of the most cited papers in recent years has been the work of Sălcianu and Rinard [SR04] [SR05] where they introduce a new purity and side effects analysis for Java programs. Their analysis is built on top of a combined pointer and escape analysis, and is able to identify that methods are pure even when they mutate the heap, provided they mutate only new objects. Additionally the analysis can recognize read-only parameters which also has been introduced by Porat et al. [PBKM00]. The analysis is implemented in a tool called `Purity Analysis Kit`[2]. This tool has been evaluated for the usage in this thesis as a potential source for information about the purity and side effects of methods but was not considered to be practical for our research as it has some limitations, e.g. it only supports full program analysis (requires a `main` method) which limits its capabilities. Techniques for identifying side-effect-free methods in partial programs has been the subject of different studies. Rountev [Rou04] presented a different approach than Sălcianu and Rinard described. His approach is based on points-to relationships and calling relationships computed by a fragment class analysis. Another extension to the approach of Sălcianu and Rinard is introduced by Madhavan et. al [MRV12]. They present an analysis that can deal with callbacks and higher-order procedures modularly. An novel purity analysis based on

[1]`https://github.com/Sable/soot`
[2]`http://jppa.sourceforge.net/PURITY-README.html`

a flow-sensitive, interprocedural analysis is presented by Pearce [Pea11]. His work includes the implementation of a tool called `JPure`[3] that employs purity annotations which can be checked modularly. Furthermore, `JPure` can automatically insert annotations into existing code by inferring the purity. `JPure` has also been evaluated for the usage in our case but didn't provide enough information, was difficult to use and the source was not publicly available.

Yang et al. [YHHK15] presented an approach to automatically infer purity and side effects by using a lexical state accessor analysis. They specifically focus on exposing effects information of methods. In their work they introduce a slightly different definition of pure methods by dividing them into *stateless* and *stateful* methods. *Stateless* refers to the traditional definition of pure methods where the return value only depends on the state of its arguments and the method does not cause any observable side effects. The definition of *stateful* also includes methods whose return value depends on the state of member fields as pure methods. They implemented their analysis in a tool called `Purano`[4] which targets Java bytecode and allows the analysis of partial programs. Due the fact that `Purano` exposes both detailed information about pure methods as well as potential side effects of methods, and also is publicly available, it is chosen for performing the purity and side effects analysis in our work. Section 3.2 gives a more in-depth description about the functionalities of `Purano` as well as how it has been adapted for our case.

2.2 Empirical Studies

The design and execution of the empirical study presented in this thesis is heavily influenced by the work of Stefan Hanenberg [FKL$^+$12] [HH13] [HKJW09]. Another related work includes guidelines for empirical research in software engineering that are introduced by Kitchenham et al. [KPP$^+$02a]. They provide guidelines for experimental context and design, conduct of the experiment and data collection as well as analysis, presentation and interpretation of results.

Dolado et al. [DHOH03] presented an empirical study on the impacts of side effects upon program comprehension. Their study is based on comprehension of short code snippets that include very simple side effects which is different from our approach where we investigate the impacts on more complex form of side effects, such as interprocedural side effects. Their results showed that already very simple side effects can have strong effects on comprehension.

[3] `http://homepages.ecs.vuw.ac.nz/~djp/jpure/`
[4] `https://github.com/farseerfc/purano`

Chapter 3

Implementation

This chapter focuses on the technical details of the implementation of the whole project. The first section will give a general overview about how everything works together on a high-level view. The second section is about `Purano`[1], the tool for detecting side effects and purity in Java bytecode. It will explain why `Purano` was chosen, what changes have been made and why. The third and last section describes how the information coming from `Purano` is used by a self-written tool called `SideEffectsDocumenter`[2] to generate Javadoc in existing source files.

3.1 Overview

The whole project of generating documentation to reflect side effects of methods can be divided into two parts. The first part is based on a program called `Purano` and focuses on detecting the purity and side effects of methods where the second part takes care of writing the actual documentation based on the results from the purity analysis. The purity analysis itself can be further divided into three parts: (1) The actual purity analysis provided by `Purano`, (2) conversion of the information into an easier understandable and better exportable data structure (3) export to JSON. Section 3.2 will provide further detail about the decision to choose `Purano` for the analysis and the data conversion.

The second tool called `SideEffectsDocumenter` uses the generated JSON and extends the corresponding Java source files from the analysed project with human readable information about the purity and possible side effects of methods in the form of Javadoc. The main steps are: (1) Import of the JSON, (2) parse Java source files, (3) match the purity information to the corresponding source and (4) extend the source file with the information. A more in depth explanation of the tool is given in section 3.3.

[1]`https://github.com/sawirth/purano`
[2]`https://github.com/sawirth/SideEffectsDocumenter`

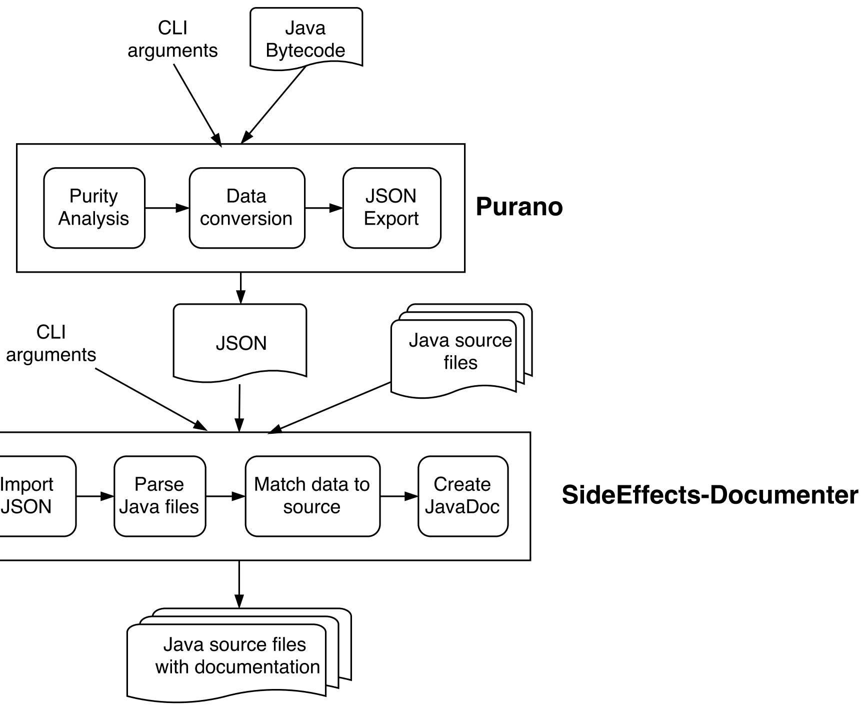


Figure 3.1: Approach overview showing the different steps from analyzing the compiled source to generating the documentation

3.2 Purity Analysis

Chapter 2 already gave an overview about what tools exists to detect the purity and potential side effects of methods. This section will cover some details about the evaluation of the suitable tool, explanation on how it works, what additions in functionality have been made to it and how the output looks like.

At the beginning of this thesis a lot of work went into the evaluation of these tools and testing which is suitable. In order to be an applicable tool for our research the following requirements need to be satisfied:

(i) Able to analyse both programs with or without a `main` method

(ii) Provide information about purity and side effects on method level

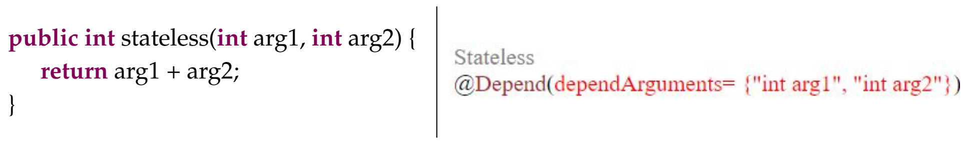
This turned out to be rather challenging because all of the found tools are research prototypes with little to no information on how to use them or get them to run. Also not all tools mentioned in the literature could be found online. Another problem faced was that some of the tools required a `main` method to start their analysis which is a significant limitation to analyse a wide range of different programs which we wanted to be possible especially that nowadays a lot of software projects are using third-party libraries to perform all kind of operations. Usually these libraries do not have a `main` method which would make them unable to be analysed. Limitations were also given by some some tools that only focus on whether a method is pure or not and not what side effects they have. One of the only programs that satisfies all needs is `Purano` hence it is chosen for performing the purity analysis.

3.2.1 Functionality

Section 2 already gave an overview of the basic concepts that Yang et. al [YHHK15] introduced and implemented in `Purano`. This section focuses more about the actual functionality the tool offers, especially what forms of purity and side effects it detects and how the corresponding output looks like.

Pure Methods

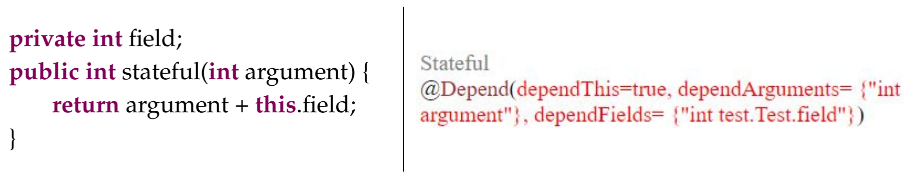
Traditionally a function is defined as pure if it always evaluates to the same result given the same arguments and is not dependent on any hidden state that may change during the execution of a program and does not cause any side effect. In object oriented languages (e.g. Java, C#) often states are encapsulated within objects that are passed to methods where as in pure functional languages (e.g. Haskell) states are passed through function arguments [YHHK15]. Based on the traditional definition all methods in object oriented languages that depend on the state of an object but do not modify it or cause any other side effect would be considered impure. Due to the fact that the object oriented paradigm is defined by methods depending on the states of object, `Purano` does not consider such methods as impure. It calls these methods *Stateful* (see Figure 3.2) where as methods without any dependency on the state of an object are called *Stateless* (see Figure 3.3).



```
public int stateless(int arg1, int arg2) {
    return arg1 + arg2;
}
```

```
Stateless
@Depend(dependArguments= {"int arg1", "int arg2"})
```

Figure 3.2: Stateless method with the corresponding output from Purano



```
private int field;
public int stateful(int argument) {
    return argument + this.field;
}
```

```
Stateful
@Depend(dependThis=true, dependArguments= {"int
argument"}, dependFields= {"int test.Test.field"})
```

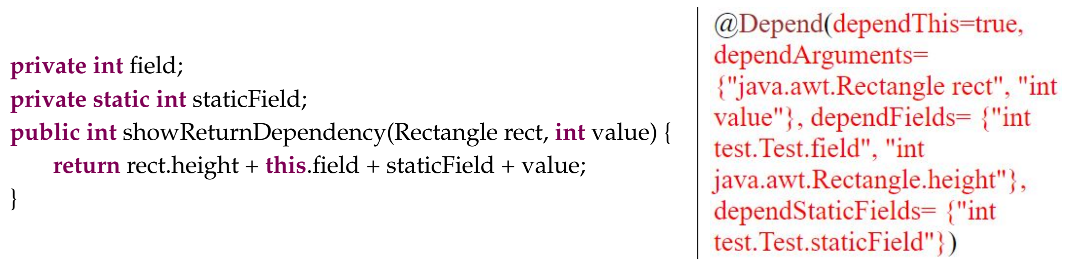
Figure 3.3: Stateful method with the corresponding output from Purano

Return Dependency

As seen in Figure 3.2 and Figure 3.3 `Purano` provides information about the dependency of the return value. It can show the following dependencies:

- Argument
- Field of an object (both argument and `this`)
- Static field

An example is shown in Figure 3.4.



```
private int field;
private static int staticField;
public int showReturnDependency(Rectangle rect, int value) {
    return rect.height + this.field + staticField + value;
}
```

```
@Depend(dependThis=true,
dependArguments=
{"java.awt.Rectangle rect", "int
value"}, dependFields= {"int
test.Test.field", "int
java.awt.Rectangle.height"},
dependStaticFields= {"int
test.Test.staticField"})
```

Figure 3.4: Example of a method with all three types of dependencies

Detecting Side Effects

`Purano` can detect four different types of side effects:

Effect type	Description	Figure
ArgumentModifier	The method modifies the state of an object passed as an argument	3.5
FieldModifier	The method modifies the state of a member variable	3.6
StaticFieldModifier	The method modifies the state of static variable	3.7
Native	The method calls system routines	3.8

Table 3.1: Side effects detected by Purano

As seen in Figure 3.5 `Purano` also detects that the modification of the object is not directly happening inside the analysed method but in a method that is called inside of it. This is also called an *interprocedural* side effect. The same information is also given for the other three types of side effects. Section 3.3 will show how this information is being used for creating the appropriate documentation.

Static and Dynamic Effects

`Purano` distinguishes between static and dynamic effects of a method. An effect is static if it is caused by a static invocation of the method that causes the effect where as a dynamic effect is inferred from a dynamic call of such methods, e.g. a call to an overridden method in a subclass. For example a class A has a method A that changes the state of a passed argument and two subclasses B and C, each overriding method A and having a different side effect. Now method A has both a static effect and two dynamic effects. The default output of `Purano` does not show this difference but it can be extracted from the underlying data structure by comparing the set of static effects with the set of dynamic effects. Figure 3.9 shows an example of a method that both has a static effect (`@Static`) and two different dynamic effects (`@Native` and `@Argument`) coming from the overridden methods in the subclasses.

```
public void argumentModifier(
        Rectangle rect,
        int height) {
    privateArgumentModifier(rect, height);
}

private void privateArgumentModifier(
        Rectangle rect,
        int height) {
    rect.height = height;
}
```

```
ArgumentModifier
@Argument(name="rect", dependArguments=
{"java.awt.Rectangle rect"}, from = "void
test.Test#privateArgumentModifier
(java.awt.Rectangle, int)")
```

Figure 3.5: Method that modifies the state of a passed argument with the corresponding output from Purano

```
private int fieldToModify;
public void fieldModifier(int arg) {
    this.fieldToModify = arg;
}
```

```
FieldModifier
@Field(type=int.class,
owner=test.Test.class,
name="fieldToModify",
dependArguments= {"int arg"})
```

Figure 3.6: Method that modifies the state of its own object

```
public static int staticField;
public void staticFieldModifier(int arg) {
    staticField = arg;
}
```

```
StaticModifier
@Static(type=int.class,
owner=test.Test.class,
name="staticField",
dependArguments= {"int arg"})
```

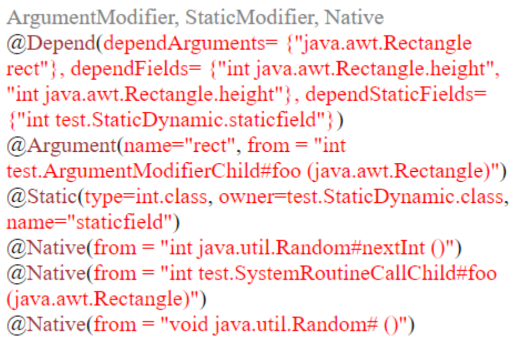
Figure 3.7: Method that modifies the state of a static variable

```
public void callSystemRoutine() {
    systemRoutine();

    Random r = new Random();
    r.nextBoolean();
}

private native void systemRoutine();
```

```
Native
@Native(from = "boolean java.util.Random#nextBoolean
()")
@Native(from = "void java.util.Random# ()")
@Native(from = "void test.Test#systemRoutine ()")
```

Figure 3.8: Method that calls system routines

```
public abstract class StaticDynamic {
    private static int staticfield;

    public int foo(Rectangle rect) {
        Random random = new Random();
        staticfield = random.nextInt();
        return rect.height * staticfield;
    }
}
```



```
ArgumentModifier, StaticModifier, Native
@Depend(dependArguments= {"java.awt.Rectangle
rect"}, dependFields= {"int java.awt.Rectangle.height",
"int java.awt.Rectangle.height"}, dependStaticFields=
{"int test.StaticDynamic.staticfield"})
@Argument(name="rect", from = "int
test.ArgumentModifierChild#foo (java.awt.Rectangle)")
@Static(type=int.class, owner=test.StaticDynamic.class,
name="staticfield")
@Native(from = "int java.util.Random#nextInt ()")
@Native(from = "int test.SystemRoutineCallChild#foo
(java.awt.Rectangle)")
@Native(from = "void java.util.Random# ()")
```

(a) Output from Purano

```
public class ArgumentModifierChild
    extends StaticDynamic{
        @Override
        public int foo(Rectangle rectangle) {
            rectangle.height = 100;
            return rectangle.height;
        }

}

public class SystemRoutineCallChild
    extends StaticDynamic {
        @Override
        public int foo(Rectangle rectangle) {
            Random random = new Random();
            return random.nextInt();
        }
}
```

Figure 3.9: Method that has a static effect as well as two dynamic effects coming from two different overridden methods

3.2.2 Data Structure and Export

By default `Purano` outputs the results of the analysis as a HTML document as seen in the figures 3.2 and 3.5. Given that a HTML document is not suitable for a further usage by importing and parsing the information, `Purano` has been extended to support an export to JSON. Due to the underlying data structure being deeply nested and containing information that is not needed for our further usage another data structure is introduced that is very similar to the concepts of `Purano` but is much less nested and reduced to the relevant information (see figure 3.10). The idea is to combine all necessary information about one effect in one class and using primitive data types to avoid traversing through a lot of different classes once the data is used for generating the documentation (see section 3.3). Another important aspect when designing this data structure was to make sure each `MethodRepresentation` can be clearly identified especially when multiple methods with the same name are present in a project (e.g. when overloading methods inside a class or using the same names for classes in different packages). Once the data structure has been fully constructed it is exported to JSON by using `google-gson`[3] allowing it to be further used in the process of generating the documentation.

Classes

The `ClassRepresentation` is the top most class that holds a set of `MethodRepresentations` and is identified by its full name (package plus class name). It is mainly used for matching (see Section 3.3.2) the data from Purano to the actual source. `MethodRepresentation` is the

[3]`https://github.com/google/gson`

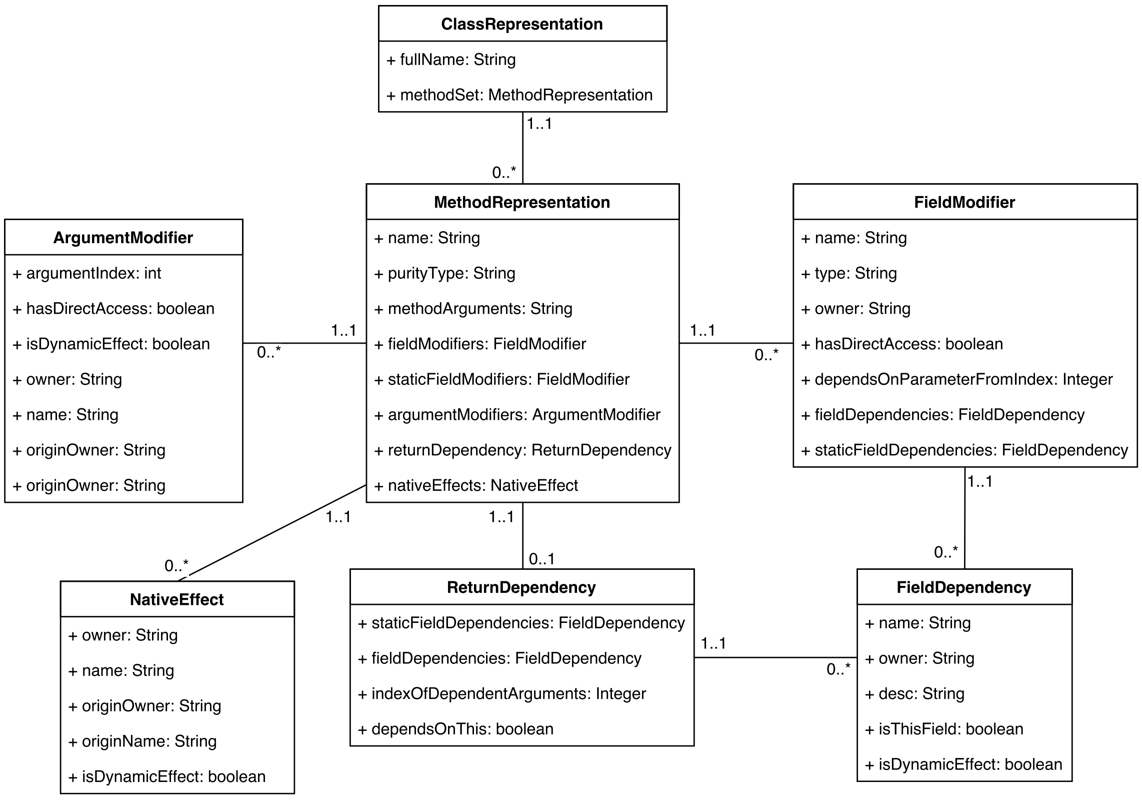

ClassRepresentation
+ fullName: String
+ methodSet: MethodRepresentation
1..1
0..*
MethodRepresentation
+ name: String
+ purityType: String
+ methodArguments: String
+ fieldModifiers: FieldModifier
+ staticFieldModifiers: FieldModifier
+ argumentModifiers: ArgumentModifier
+ returnDependency: ReturnDependency
+ nativeEffects: NativeEffect
ArgumentModifier
+ argumentIndex: int
+ hasDirectAccess: boolean
+ isDynamicEffect: boolean
+ owner: String
+ name: String
+ originOwner: String
+ originOwner: String
0..*
1..1
FieldModifier
+ name: String
+ type: String
+ owner: String
+ hasDirectAccess: boolean
+ dependsOnParameterFromIndex: Integer
+ fieldDependencies: FieldDependency
+ staticFieldDependencies: FieldDependency
1..1
0..*
1..1
0..*
NativeEffect
+ owner: String
+ name: String
+ originOwner: String
+ originName: String
+ isDynamicEffect: boolean
1..1
0..1
ReturnDependency
+ staticFieldDependencies: FieldDependency
+ fieldDependencies: FieldDependency
+ indexOfDependentArguments: Integer
+ dependsOnThis: boolean
1..1
0..*
FieldDependency
+ name: String
+ owner: String
+ desc: String
+ isThisField: boolean
+ isDynamicEffect: boolean
1..1
0..*

Figure 3.10: Class diagram of the data structure that is used for exporting to JSON

central class holding all information about the purity and side effects of a single method. It holds a list of each of the different types of side effects (see Table 3.1) as well as a list of argument names and the return dependency. Each of the classes that represent a side effect contain both a field for the owner and the name of a method called inside the method (represented by `MethodRepresentation`) that is responsible for the causing the side effect. If the side effects happens directly in a method without calling other methods these fields are empty. Additionally `NativeEffect` and `ArgumentModifier` contain fields to store information about the origin of an effect which can be seen in Figure 3.11.

The field modifier effects (both static and non-static) are represented by the class `Field-Modifier` that holds information on what the change depends on. This can either be from a method parameter, a field or a static field. Dependencies on fields (both static and non-static) are stored in `FieldDependencies` which includes information about from which class it is, what its name is and which data type it has. Furthermore, it has a boolean to indicate if the effect is a dynamic effect (see Section 3.2.1). This information is also stored for `ArgumentModifier` and `NativeEffect`. Information stored in `ReturnDependency` is very similar to the dependencies of a `FieldModifier`, the only addition is that it holds a boolean whether the return value depends on a field from the class of the method or not.

```java
//The JSON is from this method
public void first(Rectangle rect) {
    second(rect);
}

private void second(Rectangle rect) {
    third(rect);
}

private void third(Rectangle rect) {
    rect.height = 100;
}
```

```json
"argumentModifiers": [
{
    "argumentIndex": 0,
    "hasDirectAccess": false,
    "isDynamicEffect": false,
    "owner": "test.ArgumentModifier",
    "name": "second",
    "originOwner": "test.ArgumentModifier",
    "originName": "third"
}
```

Figure 3.11: JSON result of a method where the side effect does not happen directly inside of it (i.e. an interprocedural side effect)

3.3 Generating Documentation

This section covers the details about the tool `SideEffectsDocumenter` which is responsible for converting the information coming from `Purano` into readable Javadoc and mapping it to the correct method to make it available to the person reading the source code of the analysed project.

3.3.1 Import JSON and Parsing of Source Files

The import of the JSON coming from `Purano` is done by using the same data structure as shown in figure 3.10 and `google-gson`.

```java
public Set<ClassRepresentation> deserializePuranoResult(String puranoJson) {
    Type collectionType = new TypeToken<HashSet<ClassRepresentation>>(){}
                                    .getType();
    return gson.fromJson(puranoJson, collectionType);
}
```

Listing 3.1: Deserializing the JSON

The parsing of the Java source files is done by using the library `JavaParser`[4] that offers a wide variety of functions to analyse and manipulate source files. Especially it offers an easy to use API to add or change the Javadoc of a method (see 3.3.3). Listing 3.2 shows how the parsing of java files works.

3.3.2 Matching

Once both the information from `Purano` and source files have been imported they have to be matched together and stored in a new object. Each `ClassRepresentation` is matched with the corresponding object from `JavaParser` and the same is done for all `MethodRepresentations`. In both cases this is done by taking each object from `JavaParser` and then filtering all the objects from `Purano` based on the name and package declaration to find the correct object.

[4]`https://github.com/javaparser/javaparser`

```
public Set<JavaParserResult> parseFilesFromPath(String rootPath) {
    Set<Path> filePaths = getAllFilePaths(rootPath);

    Set<JavaParserResult> javaParserResults = new HashSet<>();
    filePaths.forEach(path −> {
        CompilationUnit cu = null;
        try {
            cu = JavaParser.parse(path);
        } catch (IOException e) {
            e.printStackTrace();
        }

        if (cu != null) {
            javaParserResults.add(new JavaParserResult(path, cu));
        }
    });

    return javaParserResults;
}
```

Listing 3.2: Parsing of source files

For matching the data of methods together also the number and data types of the arguments is used to find the correct `MethodRepresentation` if filtering by name and package was not sufficient. Algorithm 1 illustrates this approach.

Algorithm 1: Find matching MethodRepresentation from Purano

```
Data: parsed method from JavaParser, list of methods from Purano
Result: Matching method from Purano
list:= filterByName(list of methods);
if list has one element then
    return element;
end
list:= filterByNumberOfArguments(list of methods);
if list has one element then
    return element;
end
list:= filterByDataTypesOfArguments(list of methods);
if list has one element then
    return element;
end
return null;
```

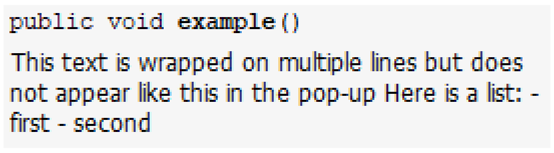
3.3.3 Create Documentation

This section covers some of the ideas behind how the actual documentation is created as well as some code examples showing the result. The goal was to create a documentation that provides useful information to the developer and is easy to read in both the source itself as well as the pop-up most modern IDE[5] (e.g. IntelliJ IDEA[6]) offer to show the Javadoc when hovering over a method with the mouse cursor.

Program options

`SideEffectsDocumenter` offers some options to tweak how the final documentation looks like. One option tries to make Javadoc links out of method references[7]. Another options tackles the problem that Javadoc that contains lists with line breaks and indentation may be easily read inside the source file itself but not in the pop-up provided by the IDE. The solution for this are HTML tags for line breaks (`<br>`) and lists (`<ul>` and `<li>`) that are inserted into the documentation to make it better readable inside the Javadoc pop-up (see Figure 3.12).

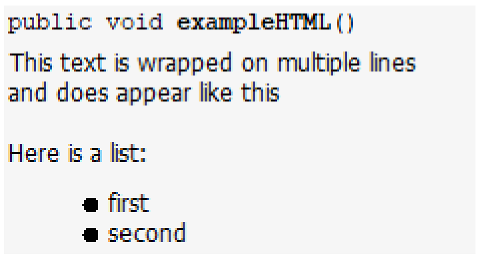
```
/**
* This text is wrapped on multiple lines
* but does not appear like this in the pop-up
*
* Here is a list:
* - first
* - second
*/
public void example() {/*empty*/}
```



```
public void example()
```

This text is wrapped on multiple lines but does not appear like this in the pop-up Here is a list: - first - second

```
/**
* This text is wrapped on multiple lines <br>
* and does appear like this <br><br>
*
* Here is a list:
* <ul>
* <li>first</li>
* <li>second</li>
* </ul>
*/
public void exampleHTML() {/*empty*/}
```



```
public void exampleHTML()
```

This text is wrapped on multiple lines
and does appear like this

Here is a list:

- first
- second

Figure 3.12: Difference between Javadoc with and without HTML tags

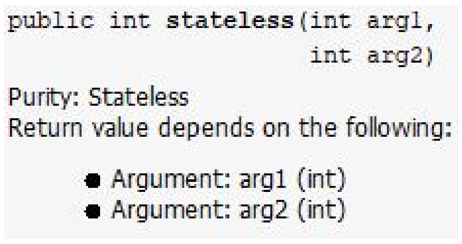
A detailed description on how the options are being used and how to run the program can be found in Appendix A.2.

[5]Integrated Development Environment
[6]`https://www.jetbrains.com/idea/`
[7]`https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html#CHDDIECH`

Pure methods

For *stateless* methods the only information besides the actual purity itself that can be shown is the `ReturnDependency` which is described in Section 3.3.3. An example of how the documentation looks like as code and in the IDE pop-up is shown in Figure 3.13. In case the return value depends on either a field or a static field the purity of that method is *stateful* and the `ReturnDependency` cointains the name of the field. Figure 3.14 shows an example of such a method.

```
/**
 * Purity: Stateless <br>
 *
 * Return value depends on the following:
 * <ul>
 * <li> Argument: arg1 (int) </li>
 * <li> Argument: arg2 (int) </li>
 * </ul>
 */
public int stateless(int arg1, int arg2) {
    return arg1 + arg2;
}
```



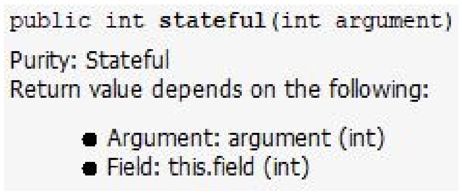
```
public int stateless(int arg1,
                     int arg2)
```

Purity: Stateless
Return value depends on the following:

- Argument: arg1 (int)
- Argument: arg2 (int)

Figure 3.13: Documentation of a stateless method with pop-up inside the IDE

```
private int field;

/**
 * Purity: Stateful <br>
 *
 * Return value depends on the following:
 * <ul>
 * <li> Argument: argument (int) </li>
 * <li> Field: this.field (int) </li>
 * </ul>
 */
public int stateful(int argument) {
    return argument + this.field;
}
```



```
public int stateful(int argument)
```

Purity: Stateful
Return value depends on the following:

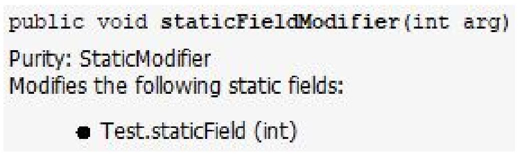
- Argument: argument (int)
- Field: this.field (int)

Figure 3.14: Documentation of a stateful method with pop-up inside the IDE

FieldModifier

If the method modifies an internal field of a class no additional information besides the purity type (`FieldModifier`) is shown. The reason for this is that a method calling such a method doesn't need to know about what internal fields are changing because this is an implementation detail which should be hidden in an object oriented software design. `StaticModifier` is different, here the information which static field is changing is contained in the documentation. An example is shown in Figure 3.15.

```
public static int staticField;

/**
 * Purity: StaticModifier <br>
 *
 * Modifies the following static fields:
 * <ul>
 * <li> Test.staticField (int) </li>
 * </ul>
 *
 */
public void staticFieldModifier(int arg) {
    staticField = arg;
}
```


public void staticFieldModifier(int arg)

Purity: StaticModifier
Modifies the following static fields:

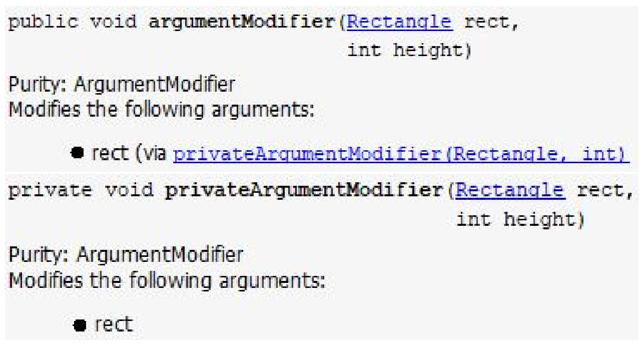
- Test.staticField (int)

Figure 3.15: Documentation of a method that modifies a static field

ArgumentModifier

The documentation for `ArgumentModifier` includes the name of the argument that is being changed as well as which method call inside the body of the method is responsible for it if the effect does not directly happen inside the method itself. Figure 3.16 shows an example of method where the effect does happen in another method (`privateArgumentModifier()`) and that information is visible in the documentation of `argumentModifier()`.

```
/**
 * Purity: ArgumentModifier <br>
 *
 * Modifies the following arguments:
 * <ul>
 * <li> rect (via Test.privateArgumentModifier) </li>
 * </ul>
 */
public void argumentModifier(Rectangle rect, int height) {
    privateArgumentModifier(rect, height);
}

/**
 * Purity: ArgumentModifier <br>
 *
 * Modifies the following arguments:
 * <ul>
 * <li> rect </li>
 * </ul>
 */
private void privateArgumentModifier(Rectangle rect, int height) {
    rect.height = height;
}
```



```
public void argumentModifier(Rectangle rect,
                             int height)
```

Purity: ArgumentModifier
Modifies the following arguments:

- rect (via privateArgumentModifier(Rectangle, int)

```
private void privateArgumentModifier(Rectangle rect,
                                     int height)
```

Purity: ArgumentModifier
Modifies the following arguments:

- rect

Figure 3.16: Documentation and IDE pop-up of method that modifies the state of an argument

Native Effects

For native effects it was important to distinguish between regular system calls and I/O-operations such as printing something to the console or reading from a file. To support this, `SideEffects-Documenter` compares both the actual method of the effect as well as the origin method with a blacklist. The blacklist is a text file that can contain packages, classes or single methods that do I/O-operations. The path to the file is then passed to the program as CLI argument. A sample blacklist could look like this:

```
# Packages
java.io
java.nio
org.apache.commons.io
java.sql

# Classes
test.utils.FileWriterUtils

# Methods
test.NativeImpl.nativeCalculation
task1.Test.IOCall
```

If an effect is caused by a method that is blacklisted the documentation contains a hint that the method might cause I/O operations (see Figure 3.17).

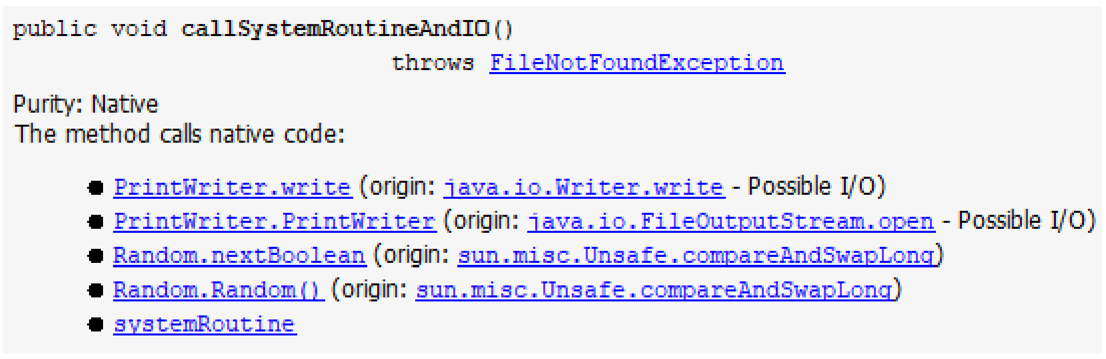
ReturnDependency

The documentation of the `ReturnDependency` includes information on which fields (static and non-static) and arguments the return value depends on. Each of the values is written as an element of a list to enable easy reading (see Figure 3.18).

Static and Dynamic Effects

Section 3.2.1 already explained how `Purano` detects both static and dynamic effects. If a method has both dynamic and static effects of the same type, the dynamic effects will be shown separately from the static effects as seen in Figure 3.9. Does a method only have static effects of one type these are not particulary marked as static effects.

```
/**
 * Purity: Native <br>
 *
 * The method calls native code:
 * <ul>
 * <li> {@link PrintWriter#write} (origin: {@link Writer#write} – Possible I/O) </li>
 * <li> {@link PrintWriter#PrintWriter} (origin: {@link FileOutputStream#open} – Possible I/O) </li>
 * <li> {@link Random#nextBoolean} (origin: {@link Unsafe#compareAndSwapLong}) </li>
 * <li> {@link Random#Random()} (origin: {@link Unsafe#compareAndSwapLong}) </li>
 * <li> {@link Test#systemRoutine} </li>
 * </ul>
 */
public void callSystemRoutineAndIO() throws FileNotFoundException {
    systemRoutine();

          Random r = new Random();

          r.nextBoolean();
          PrintWriter pw = new PrintWriter("filename.txt");
          pw.write("hello");
      }

/**
* Purity: Native <br>
*/
private native void systemRoutine();
```



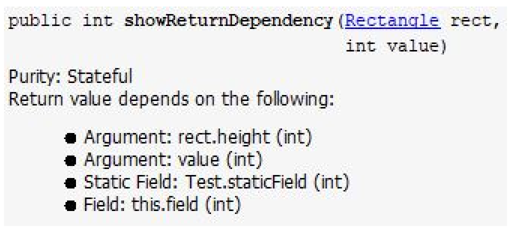
```
public void callSystemRoutineAndIO()
                              throws FileNotFoundException
```

Purity: Native
The method calls native code:

- PrintWriter.write (origin: java.io.Writer.write - Possible I/O)
- PrintWriter.PrintWriter (origin: java.io.FileOutputStream.open - Possible I/O)
- Random.nextBoolean (origin: sun.misc.Unsafe.compareAndSwapLong)
- Random.Random() (origin: sun.misc.Unsafe.compareAndSwapLong)
- systemRoutine

Figure 3.17: Documentation of a method that calls system routines and how it looks inside an IDE

```
/**
 * Purity: Stateful <br>
 *
 * Return value depends on the following:
 * <ul>
 * <li> Argument: rect.height (int) </li>
 * <li> Argument: value (int) </li>
 * <li> Static Field: Test.staticField (int) </li>
 * <li> Field: this.field (int) </li>
 * </ul>
 */
public int showReturnDependency
(Rectangle rect, int value) {
    return rect.height
            + this.field
            + staticField
            + value;
}
```



```
public int showReturnDependency(Rectangle rect,
                                int value)
```

Purity: Stateful
Return value depends on the following:

- Argument: rect.height (int)
- Argument: value (int)
- Static Field: Test.staticField (int)
- Field: this.field (int)

Figure 3.18: Example of the return dependency documentation

```
public abstract class StaticDynamic {
    private static int staticfield;

    /**
    * Purity: ArgumentModifier, StaticModifier, Native
    *
    * Modifies the following arguments:
    * Dynamic effects (i.e. from subclasses)
    * rect (via ArgumentModifierChild.foo)
    *
    * Modifies the following static fields:
    * StaticDynamic.staticfield (int)
    *
    * Return value depends on the following:
    * Argument: rect (Rectangle)
    * Static Field: StaticDynamic.staticfield (int)
    * Field of subclass: Rectangle.height (int)
    *
    * The method calls native code:
    * Static effects
    * Random.nextInt
    * Random.Random()
    * Dynamic effects (i.e. from subclasses)
    * SystemRoutineCallChild.foo
    */
    public int foo(Rectangle rect) {
        Random random = new Random();
        staticfield = random.nextInt();
        return rect.height * staticfield;
    }
}
```

Figure 3.19: Method with both static and dynamic effects from the same type. Same method as in Figure 3.9 but with documentation

Chapter 4

Evaluation

We conducted an empirical study to measure if having prior knowledge about the purity and possible side effects of methods can have a positive influence on code comprehension of developers. The participants are divided into two groups where one group has access to the information coming from `SideEffectsDocumenter` (see Section 3.3) and the other group has no documentation.

4.1 Methods

In order to make quantitative conclusions about the impact our tool has on code comprehension of developers we designed two tasks that the participants had to solve in time. The tasks aimed to reflect real world problems caused by methods having side effects that had to be found by navigating through the code inside an IDE.

Both tasks have been built on top of a separate project[1] specifically designed for this study that tries to hide these effects in nested class hierarchies so the participants are not able to find the problem by just opening each class and look through each method. The project has then being analysed by our tool to generate the documentation which one group had access to.

To avoid further distortion of the measurements the participants only had to found the root of the problem and delete the associated lines of code and not perform any kind of refactoring task. This also enables people with experience in another object oriented language than Java to participate in the study. Also some of the participants began with the second task and later performed the first task to limit influence of learning effects on the result. Moreover, the project was designed to avoid code that might be difficult to understand due to unknown constructs (e.g. lambda expressions, usage of design patterns or unknown libraries).

4.1.1 Task 1 - Unintended I/O operation

The scenario in the first task is that different steps in a process are being executed in a task and somehow this modifies the productive database. An access to the database means that an I/O operation is happening. The location of this operation has to be found by the participants. Each of the different steps in this process contains multiple method calls with various depths of call hierarchies.

[1]`https://github.com/sawirth/SideEffectsDocumenter-Experiment`

4.1.2 Task 2 - Modifying an argument

The second task imitates a process of verifying a list of prices of products. At the end a check is made that none of the prices has changed during this process which is not the case so the check fails. This means during the process somehow the list of prices got modified which is a side effect (*ArgumentModifier*). The participants had to found the location of this modification.

4.1.3 Participants

For our study we recruited 14 participants, nine of them are currently full-time developers and five are computer science students. Each of the participants has work experience as a software developer ranging from 1 up to 13 years with an average of a little over 5 years. Eight started with task one and the other seven with task two. A detailed overview of each participant can be seen in Figure 4.1.

Documentation	**Id**	**Starting task**	**Age**	**Experience (yrs.)**	**Role**
With	1	1	26	4	Master Student
	2		31	7	Software Engineer
	3		37	13	Software Engineer
	4		26	5	Master Student
	5	2	27	5	Software Engineer
	6		27	1	Junior Software Engineer
	7		29	1	Junior Software Engineer
Without	8	1	29	5	PhD Student
	9		35	10	Software Engineer
	10		28	4	Master Student
	11		24	2	Master Student
	12	2	28	6	Software Engineer
	13		28	5	Domain Architect
	14		29	7	Software Engineer
Average			**28.9**	**5.4**	

Table 4.1: Participants of the study

4.1.4 Procedure

Each of the 14 participant was first given an introduction about what side effects and pure methods are with examples. After that they received an explanation of the two tasks and what they had to do. The group of participants that had access to the documentation also received an introduction on what information the tool generates and how it looks inside the IDE by presenting examples of each of the different purities (see Figure 4.1). All of the participants also received information about basic shortcuts for navigating through code inside the IDE. Besides the measurement of time that the participants required for solving the tasks, everything is recorded by using screen and webcam capturing as well as sound recording and tracking the events inside the IDE by using a plugin called `Activity Tracker`[2].

```java
private int field;
private static int staticField;

/**
* Purity: Stateless
*/
public int stateless(int a, int b) {
    return a + b;
}

/**
* Purity: Stateful <br>
*
* Return value depends on the following:
* <ul>
* <li> Argument: a (int) </li>
* </ul>
*/
public int oo_stateless(int a) {
    return a + this.field;
}

/**
* Purity: FieldModifier
*/
public void fieldModifier(int a) {
    this.field = a;
}

/**
* Purity: StaticModifier <br>
*
* Modifies the following static fields:
* <ul>
* <li> {@link PurityExplanation#staticField} </li>
* </ul>
*/
public void staticModifier(int a) {
    staticField = a;
}
```

```java
/**
* Purity: ArgumentModifier <br>
*
* Modifies the following arguments:
* <ul>
* <li> rectangle </li>
* </ul>
*/
public void argumentModifier(Rectangle
        rectangle) {
    rectangle.height = 100;
}

/**
* Purity: Native <br>
*
* The method calls native code:
* <ul>
* <li> {@link PurityExplanation#nativeMethod
        ()} </li>
* <li> {@link FileWriter#write(String)} (Possible
        I/O) </li>
* </ul>
*/
public void nativeCall() {
    nativeMethod();

    try {
        FileWriter writer = new FileWriter("file.txt
                ");
        writer.write("text in file");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private native void nativeMethod();
```

Figure 4.1: Explanation that the participants with access to the documentation received

[2] `https://github.com/dkandalov/activity-tracker`

4.2 Results

In this section we present the evaluation results of both the quantitative measurements as well as observations on how the participants interacted with the documentation based on the collected video material.

4.2.1 Quantitative Results

Based on the measurements from the study shown in Table 4.2 the group with access to the documentation solved the tasks 22% faster. However there is a big difference between task 1 and task 2. On average there is only a 12% improvement in task 1 which is equal to not even a minute based on an average solving time of 472 seconds. Differences are bigger for task 2 where the group with documentation solved the task 44% faster which is measured in absolute time is 159 seconds based on the average solving time of 361 seconds.

Based on the computed descriptive statistics seen in Table 4.4 it is also interesting to see that the fastest participants from each group needed almost the same time for solving task 1 (187s to 197s) where as for task 2 the differences are much more significant (113s to 228s). The table also shows that the standard deviation for the group with documentation is lower than for the other group for both tasks which means the results are less scattered.

These differences between the two tasks can also be seen in the boxplots in Figure 4.2 and Figure 4.3. Less scattered values result in a narrower inner box (= lower interquartile range).

	Id	**Task 1 (s)**	**Task 2 (s)**
With documentation	1	360	300
	2	474	113
	3	720	180
	4	528	294
	5	480	240
	6	187	416
	7	341	268
Without documentation	8	562	507
	9	295	459
	10	197	228
	11	960	538
	12	234	660
	13	497	426
	14	780	420
Average		**473**	**361**

Table 4.2: Absolute measurements of all participants for both tasks in seconds

	Without documentation	With documentation	Difference
Task 1	504	441	12%
Task 2	463	259	44%
Overall	966	700	28%

Table 4.3: Comparison of the averages (in seconds) for both tasks with and without documentation

Task	Documentation	min(s)	max(s)	mean(s)	median(s)	std. dev.
1	with	187	720	441	474	155.42
	without	197	960	504	497	266.82
2	with	113	413	259	268	89.09
	without	228	660	463	459	122.37

Table 4.4: Descriptive statistics for both tasks

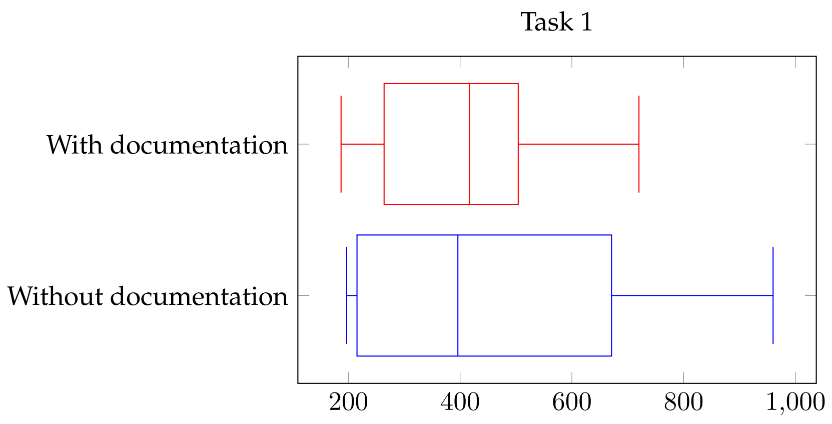


Figure 4.2: Boxplot of the results from task 1

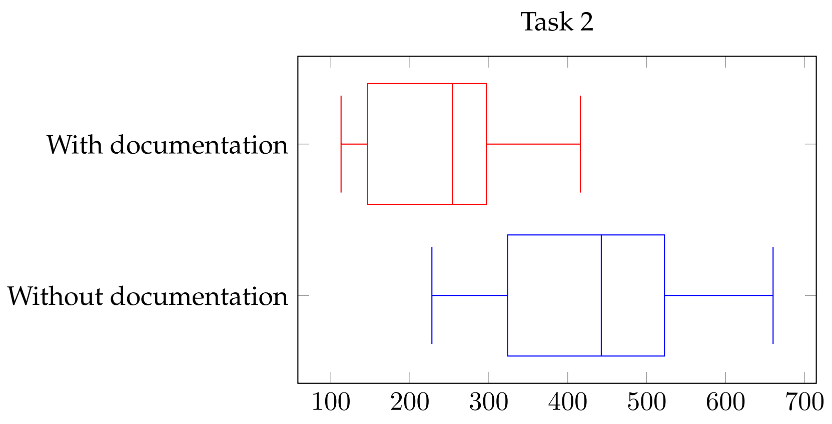


Figure 4.3: Boxplot of the results from task 2

4.2.2 Observations

During the study we also collected video material to see how the participants interacted with the documentation and how they tried to solve the tasks. Although all of the participant that were in the group with the documentation received the same explanation on how the documentation looks like some barely noticed it or were confused by it and jumped right into the code without even looking at the documentation. They stated that they are not familiar with the usage of Javadoc. Another source of irritation especially in the first task were the native effects. In the experiment code both native effects with and without an info about possible I/O operations are present. This confused some of the participants because they began to deeply investigate also the methods without the info about possible I/O and therefore lost a lot of time.

Another noticeable observation was that some of the participants from both groups jumped right into the first method call on the upper most level without even looking on what other method calls are on the same level. This behaviour had in most cases a worse impact on the outcome if they didn't had the documentation because they were not able to exclude particular methods calls from further investigation.

4.3 Threats to Validity

This section discusses several points which threaten the validity of the results from this study which is necessary according to known guidelines for empirical research in software engineering (see [KPP+02b] and [KAKB+06]).

4.3.1 Internal Validity

Experiment Design It is unclear to what extent a programming task can be considered as representative which does not permit generalisation to an arbitrary set of programming tasks [HH13]. Both tasks and the experimental code were designed to favour the benefit of having our documentation. There was no task that tries to measure if the documentation could also have a negative influence on code comprehension.

To reduce the threat of learning effects the two task scenarios are different from each other and do not rely on the exact same methods. Also not all the participants began with the same task first to reduce the same effect. Nonetheless, it is possible that this aim was not completely achieved by the design of the experiment.

Another slight problem is that each of the participants performed the experiment separately which means that not all of them received the explanation in the exact same way although it was the intention to do so.

Participants The selection of the participants is a possible threat as they are not chosen randomly and might be biased. It is also often stated in the literature (see Hanenberg [Han10]) that students are not a valid sample for developers and therefore can be a threat to validity. However, only the minority of the participants were students and all of them have experience in professional software engineering. The participants also had different levels of programming experience and knowledge about Java and the used IDE as well as what side effects and pure methods are. The definition and explanation of the tasks may also not be understood equally well by all the participants.

Novelty Effects All participants completed the task on the same machine (which was a Toshiba Laptop). Most certainly this required some time to get used to this setup especially for people that are not familiar with developing on laptops. This is potential threat to validity as some people can adapt quicker to new and unknown environments than others.

4.3.2 External Validity

Used Systems/Scenarios The solely for this study written project may be considered as a threat because it does not reach the size and complexity of large systems used in the industry, in terms of number of classes, lines of code or hierarchy.

Questions Furthermore, the representativeness of the questions is a potential threat because the questions may not reflect real problems. Other kinds of tasks might matter more than the ones we considered.

Participants The small number of 14 participants is another threat to validity that limits the generalisation of our results.

Programming Language Due to the limitation of choosing Java as the programming language for the evaluation the results may not be directly inferred to other languages.

Chapter 5

Conclusions

5.1 Conclusions

The work of this thesis shows an approach of exposing information about the purity and possible side effects of methods to developers. Based on the results presented in Section 4.2 having such information can help increase code comprehension. They also showed that there are situations where the documentation is more useful and situations where it can cause irritation which has a negative impact on code comprehension. For example, a lot of problems that are caused by unexpected side effects (such as a modification of a parameter) can be located by using a variety of functions that are included in modern IDEs rather then by conducting our documentation. This also implies that the documentation might not be very useful during the task of locating an error but rather be useful when writing new code that reuses unknown software modules.

However, the study also revealed that little details can decide whether the documentation can help increasing code comprehension or creates confusion. One of the main irritations were caused by the presence of native effects. Most of the participants did not quite understand what they are and how the information could help them.

Furthermore, the study definitely showed that Javadoc might not the best suited way to show information about purity and side effects. The majority of the participants were not familiar with using Javadoc as a source of code comprehension. Javadoc is too inconspicuous to recognize that a method causes potential side effects or that it is pure. Finally, the usage of Javadoc to document purity is questionable particularly when it is combined with the usual content of Javadoc. This would make the purity information even less recognizable.

The success of a tool that provides information about purity and side effects also strongly depends on its precision in detecting these effects correctly. Once developers notice that the information might not be correct for several methods they lose confident in the tool and abandon it. A major concern also is that the side effects a method could cause based on its documentation might never arise because it depends on one or more conditions to be satisfied. This implies that is better for such methods to not contain these information directly in their description or at least not showing them in the same way as effects are shown that always occur.

In conclusion, this thesis presents an early work on how information about the purity and potential side effects of methods can be included in the workflow of developers and how it can affect code comprehension.

5.2 Summary of Contributions

This section covers the main contributions of this thesis sorted from most to least important.

(i) Implementation of a prototype to automatically generate information about the purity and potential side effects of methods based

(ii) Empirical study to evaluate an increase in the productivity of developers if they have access to information about purity and side effects of methods

(iii) Evaluation and modification of an existing tool for purity and side effects analysis for further usage

5.3 Future Work

This section discusses possible ideas for further research and improvements of the prototype that is introduced in this thesis. The ideas are based on our own experiences and opinions from participants of the study.

The prototype we present in this thesis could be used for further studies about how developers can benefit from the documentation of purity and side effects. One idea is to investigate if the information actually can help during the process of writing new code, e.g. when reusing software modules. Another approach could be to investigate about potential negative influences the documentation can have. Considering the following example: Methods that might cause trouble due to side effects should probably be refactored to avoid problems in the future. Now if there exists a documentation that shows that such methods have side effects, maybe developers tend to avoid a refactoring because they know about the side effects and therefore might implement workarounds so that the side effect does not cause any problems in their case.

Improvements to the prototype can be made in a variety of possible ways. Based on the fact that Javadoc might not be the best way to provide information about the purity of methods a simple way for improvement would be to introduce annotations similar to what Pearce [Pea11] has shown. A plugin could use this information and flag calls that potentially cause side effects. Another way of improvement could be to integrate our tool into the build process so every time the build process runs also a side effect analysis is performed and documentation is created.

Currently the `SideEffectsDocumenter` works only on the modified export coming from `Purano`. This could be extended so that the tool supports other sources of information about the purity and side effects of methods, i.e. coming from other analysing tools. An even further improvement would be to compare different sources and create the documentation accordingly. For instance if one source reports that a method has no side effects and other one detects side effects, the documentation should consider this.

During the study we collected video and sound material as well as tracking data from inside the IDE that could be used for further research. For example, this material can give insights on how many different source files each of the participants opened until they solved the task. Conducting the video and sound material in detail could also help find ideas to further improve the prototype.

Appendices

Appendix A

Run instructions

A.1 Purano

To be able to run the modified version of `Purano` the following prerequisites must be met:

(i) Java Runtime Environment (JRE) 1.7 installed

(ii) .jar of `Purano`

(iii) Compiled version of the project to be analysed (either as .class files or as .jar)

The program can then be run with the following command:

```
java -cp [path to purano jar;path to project]
jp.ac.osakau.farseerfc.purano.reflect.ClassFinder
-p [name of packages to be analysed]
-o [output path]
```

To analyse an entire project containing different packages you either have to enter the name of the uppermost package (usually the top-level domain of a country) or enter multiple package names separated by a whitespace. This produces the JSON that is used for the next step.

A.2 SideEffectsDocumenter

To be able to run `SideEffectsDocumenter` the following prerequisites must be met:

(i) JRE 1.7 installed

(ii) .jar of `SideEffectsDocumenter`

(iii) Java source files from same project that has been analysed by `Purano`

(iv) JSON Output from `Purano`

(v) .txt file for the I/O blacklist

The I/O blacklist is needed to mark specific methods as possible I/O methods such as methods from `java.io`. The basic run instruction is the following:

```
java -jar [path to jar of SideEffectsDocumenter]
   [root path of source files]
   -p [path to JSON]
   -io [path to I/O blacklist]
```

The tools offers additional CLI arguments to further tweak the output:

Option	**Description**	**CLI argument**
Extended documentation	If enabled the documentation shows all possible informations and not just the purity (e.g. Stateless, Stateful etc.)	-e
Output file path	The modified version of the source file will be saved here. Default is the location the file already is	-o [path]
Replace files	If enabled the tool will override existing source files	-r
Create links	This will try to make Javadoc hyperlinks (`@link{}`) for methods	-l
Create HTML lists	The Javadoc will be formatted with `<ul>` and `<li>` elements which enables better reading inside the Javadoc pop-up of the IDE	-t

Table A.1: Optional CLI arguments for `SideEffectsDocumenter`

Appendix B

Used tools

Programming tools and libraries

IntelliJ IDEA IDE for all programming tasks as well as the execution of the study

JavaParser Library that is used for parsing and modifying Java source files

Gson Library that is used for both the export and import of JSON files

Guice Dependency Injection framework used in `SideEffectsDocumenter`

Activity Tracker Plugin for IntelliJ IDEA to track user activity inside the IDE

JCommander Framework for trivial parsing of command line parameters

JSAP Java Simple Argument Parser

Other tools

TeX Live TeX Distribution that is used to build this thesis

Atom Editor that is used to write this thesis

Draw.io Online diagram software

Zotero Used to manage bibliographic data and related research materials

ActivePresenter Used for screen, webcam and audio recording

Appendix C

Content on the CD-ROM

Abstract.txt Unformatted abstract of this thesis in English

Zusfsg.txt Translation of Abstract.txt to German

Bachelorarbeit.pdf Complete and final version of this thesis

purano.zip Source code of the modified version of `Purano` (also available on GitHub: `https://github.com/sawirth/purano`)

SideEffectsDocumenter.zip Source code of `SideEffectsDocumenter` (also available on GitHub: `https://github.com/sawirth/SideEffectsDocumenter`)

SideEffectsDocumenter-Experiment.zip Source code that is used in the empirical study (also available on GitHub: `https://github.com/sawirth/SideEffectsDocumenter-Experiment`)

Toolchain Folder that contains all necessary files and executables to run both the purity analysis as well as the generation of the documentation. Please refer to Appendix A for further details.

Bibliography

[Ban79] John P. Banning. An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 29–41, New York, NY, USA, 1979. ACM.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[CK88] K. D. Cooper and K. Kennedy. Interprocedural Side-effect Analysis in Linear Time. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 57–66, New York, NY, USA, 1988. ACM.

[Cla97] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, November 1997.

[DHOH03] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, July 2003.

[FKL$^+$12] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 73–82, June 2012.

[Han10] Stefan Hanenberg. An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[HH13] Michael Hoppe and Stefan Hanenberg. Do Developers Benefit from Generic Types?: An Empirical Comparison of Generic and Raw Types in Java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.

[HKJW09] Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. Does Aspect-oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proceedings of the 2009 3rd International Symposium* *on Empirical Software Engineering and Measurement*, ESEM '09, pages 156–167, Washington, DC, USA, 2009. IEEE Computer Society.

[KAKB$^+$06] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammad Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating Guidelines for Empirical Software Engineering Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 38–47, New York, NY, USA, 2006. ACM.

[KPP$^+$02a] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[KPP$^+$02b] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 1–11, New York, NY, USA, 2002. ACM.

[MRV12] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular Heap Analysis for Higher-order Programs. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 370–387, Berlin, Heidelberg, 2012. Springer-Verlag.

[PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic Detection of Immutable Fields in Java. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '00, pages 10–, Mississauga, Ontario, Canada, 2000. IBM Press.

[Pea11] David J. Pearce. JPure: A Modular Purity System for Java. In *Compiler Construction*, Lecture Notes in Computer Science, pages 104–123. Springer, Berlin, Heidelberg, March 2011.

[Raz99] Chrislain Razafimahefa. A Study Of Side-Effect Analyses For Java. Technical report, 1999.

[Rou04] A. Rountev. Precise identification of side-effect-free methods in Java. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 82–91, September 2004.

[SR04] Alexandru Salcianu and Martin Rinard. A Combined Pointer and Purity Analysis for Java Programs. May 2004.

[SR05] Alexandru Sălcianu and Martin Rinard. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 199–215. Springer, Berlin, Heidelberg, January 2005.

[YHHK15] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Revealing Purity and Side Effects on Functions for Reusing Java Libraries. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, Lecture Notes in Computer Science, pages 314–329. Springer, Cham, January 2015.