Approach of writing NPE-safe code in java applications

Vladislav Matelsky and Natalia Lapitskaya

***Abstract*—** Null pointers and dealing with them. Automated code checks by compiler. Usage of Maybe to reduce amount of null pointer exceptions.

***Keywords*— S**oftware quality, programming technologies, Maybe, compiler-checks, human factor.

I. INTRODUCTION

Complexity of developed systems is growing as well as complexity of goals they are intended to achieve. One of factors that make systems more complex is that the same development tools are used for their creation. Abstraction level of problems to be solved is growing faster than the abstractions that programming languages provide. This difference makes coding process more vulnerable to human factor related mistakes. Therefore the better development tools or modification of development methods are needed.

In some spheres the importance of error may cost the earth. The error may lead to deaths of people or to crashing of highly expensive scientific machines. For example the catastrophe Ariane 5, that happened June 4 1996. According to the prof. Jacques-Louis Lions [1], the reason of the catastrophe was a mistake in software that occurred at boundary values.

To minimize the probability of errors checking of software reliability should be performed for all stages of system creation. If human performs verification, the chance of human-factor related mistake still exists. Instead of humans it's possible to rely on automated tools that check different aspects of systems under development.

This article provides a way to prevent NPE using set of simple rules and agreements during development process.

II. REASONS OF NPE

At first definition of NPE has to be provided. NPE is an abbreviation of "null pointer exception". Where pointers are links to memory that application uses to store own data. These links are commonly known as variables. The variables point to application memory or to a "null area". If a variable points to the "null area" it means that this variable specifies no data application can use. Since this variables contains no data attempts to obtain data from them are exceptions.

If application tries accessing objects by null pointer NPE occurs and this cases may be threated as breach of interaction contract: the null pointer occurs in a place where null pointer shouldn't be. Such contract e.g. is a set of input and output parameter. In programming languages it's a set of typed input/output values. Passing null pointer in function that don't expect it is a breach of interaction contract. Let's take a look at the following function:

```
Complex sum(Complex a, Complex b) {
        return a.add(b);
}
```

V. Matelsky, Faculty of Computer Systems and Networks, BSUIR, Minsk, Belarus (e-mail: vla3089@mail.ru).
N. Lapitskaya, Software for Information Technologies dept., BSUIR, Minsk, Belarus (e-mail: lapan@bsuir.by).

The goal of that function is to sum two complex numbers. Thereby, the function takes two complex values. But it's also possible to pass null pointers instead of certain parameters. It's not contrary to java specification, but it imposes an implicit restriction on the implementation. The implementation must take into account the fact that any of the parameters can be null pointer.

On the other hand the function is responsible for addition of two complex values while NULL is not a complex value, therefore passing it is the breach of interaction contract.

To be sure that NPE is impossible it's needed to analyze the whole code to check whether the variables are initialized correctly and none of computations produces null. And places where null pointers are possible are the places of potential NPE. Lots of condition checks make difficult looking through the code to understand it and to find other mistakes. Furthermore, as it was described above, java compiler allows passing nulls instead of objects everywhere. Such assumption can leads to a situation when changed code produces null pointer and pass it to other one that crash the application with NPE. Compiler says nothing about it despite the fact that it's breach of interaction contract. In other word modules may lose consistency silently. And because NPEs are produced in some use cases only, the investigation of them become more difficult.

III. Way of NPE minimization

As it was described above NPEs occur when the interaction contracts are breached. Compiler does not detect situations of that kind, therefore there is not warranty that contracts in the whole system are consistent. To prove that additional analysis works are needed. The most logical way is to check consistence during the build process. And if such error is found the build process should be stopped.

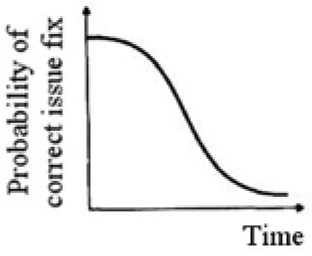

Fig. 1. Dependency of issue fixing cost from the time when the issue is detected

Detection of contract beach during the build step doesn't allow forward movement of product until issues are not fixed.

Which means that this kind of issues will be fixed during testing process, because testing process goes afterwards. Therefore tests will be performed on consistent modules.

Such way allows detecting issue earlier, and the earlier issue is detected the less money fix costs. This relationship is shown in fig. 1.

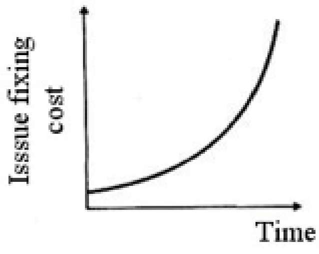

Fig. 2. Probability of applying correct issue fix during the application lifetime

Additional checks of contract consistency allows to detect issue earlier, therefore fixing of it will cost less money. And from the right graph follows that the probability of detection becomes higher.

The most reliable way of preventing issue is to prevent the possibility of occurring issues of that kind. For instance, there are some languages like Haskell where null pointers absent as a class. Instead of nulls programmers use special type called Maybe. Maybe can take one of two states: Nothing of “Just Value”. Nothing is an equivalent for null.

```
data Maybe a = Just a | Nothing deriving (Eq, Ord)
```

Variable of Maybe type clearly specifies that it may obtain no value. So, to get a value programmer have to check whether variable have it. It's semantic approach that helps to differ whether it’s value or maybe it's value. In the second case the check follows from the type.

One by one other languages implements similar solution. For instance, Scala has Option. Or Kotlin's compiler allows to assign null to variable only if “?” is followed after the type of variable. Let's take a look at example:

```
var a: Int;  // assigning null is forbidden
var b: Int? //  possible to assign null
```

These types are not equals therefore compiler checks all assignment operations between them:

```
b = a; // correct assignment
a = b; // type mismatch
```

Such way of defining type of variables allows reducing count of NPEs mostly because of types semantic: the type presupposes that value may be absent and it force programmer to think whether null-available type is needed in certain case. In additional such code is friendlier for verification: it's possible to detect where value-containing check is missing. Because if value-check is not needed it means that type should be strict and doesn't allow to assign nulls.

IV. Implementing type Maybe for Java

Common java library doesn't support semantic difference between null-possible and null-safe types, therefore NPEs occurs. There are several languages that successfully prevents from this kind of issue, so it may be useful to implement similar approach on java. One of the possible implementations is shown in the listing 1.

Listing 1. Implementation of NPE-safe class on Java

```
public abstract class Maybe<T> {

    public static <T> Maybe<T> just(T value) {
        return new JustValue<T>(value);
    }

    public static <T> Maybe<T> nothing() {
        return new Nothing<>();
    }

    static <T> Maybe<T> nullIsNothing(T value) {
        if (value == null) {
            return nothing();
        } else {
            return just(value);
        }
      }

    public abstract boolean isValue();
    public abstract T value();
    public abstract T or(T defaultValue);

    public static class JustValue<T> extends Maybe<T> {
        private final T value;

        public JustValue(T value) {
            this.value = value;
        }

        @Override
        public boolean isValue() {
            return true;
        }

        @Override
        public T value() {
            return value;
        }

        @Override
        public T or(T defaultValue) {
            return value;
        }
    }

    private static class Nothing<T> extends Maybe<T> {

        @Override
        public boolean isValue() {
            return false;
        }

        @Override
        public T value() {
            throw new RuntimeException("Trying to obtain value from nothing");
        }

        @Override
        public T or(T defaultValue) {
            return defaultValue;
        }
    }
}
```

Class Maybe<T> encapsulates null check in a way to make variable creation code linear. The example is provided in listing 2.

Listing 2. Example of usage of Maybe<T>

```
  String defaultString = "Default string";
  Maybe<String> withValue = Maybe.nullIsNothing("String with value");
  Maybe<String> nothing = Maybe.nothing();
  System.out.printf("with value \"%s\"\n", withValue.value());
  // System.out.printf("with value \"%s\"\n", nothing.value()); // throws an
exception if uncomment
  System.out.printf("or()     for     string     with     value     \"%s\"\n",
withValue.or(defaultString));
  System.out.printf("or() for nothing. Should return default string: \"%s\"",
nothing.or(defaultString));
```

As it's shown in listing creation code is linear, and if it's needed to get value if it exists or default value otherwise it's possible to use method or(). And to obtain certain value programmer have to check whether value exists using method isValue().

V. Comparison with alternatives

Java provides different ways of handling null variables. For example Listing 3 provides comparison with null implementation and listing 4 shows how to handle NPE using try\catch.

Listing 3. Using comparison with null to check if variable is null

```
void printIfValue(String str) {
  if (str != null) {
    System.out.printf("value: %s\n", str);
  } else {
    System.out.printf("no value defined\n");
  }
}
Listing 4. Using try\catch to handle NPE if variable is null
void printIfValue(String str) {
  try {
    System.out.printf("value: %s\n", str.toString());
  } catch(NullPointerException) {
    System.out.printf("no value defined\n");
  }
}
```

The main feature of comparison with null is that programmers have to remember that variable may be null everywhere they are trying to access it. These approaches don't separate null-possible and variable cannot be null, therefore they presuppose accessing each variable after null check. So, programmers prefer the following compromise: use null check only if variable may be null. But it's necessary to remember that code is changing, therefore variable may become null after some time and nobody will warns about possible NPEs in this case.

There is one more approach based on using annotations [3]. The approach is provided in listing 5.

Listing 5. Using annotations to specify that function expects not null parameter

```
void printIfValue(@NotNull String str) {
    System.out.printf("value: %s\n", str.toString());
}
```

If anybody passes null to such function compiler will generate warning. Such solution is suitable if team is looking through warnings list and fixing them. It allows to pass null as parameter, compile code and perform it.

The example of function that process Maybe variable is shown in listing 6.

Code described in Listing 6 provides following output:

– value: String with value

– no value defined

During code review places where Maybe variable is not checked is clearly visible that allows to detect possible error earlier.

Listing 6. Example of processing Maybe-parameter in function

```
printIfValue(withValue);
printIfValue(nothing);

public static void printIfValue(Maybe<String> maybeString) {
        if (maybeString.isValue()) {
            System.out.printf("value: %s\n", maybeString.value());
        } else {
            System.out.printf("no value defined\n");
        }
    }
```

Moreover if the whole team follows the rule "use no nulls in own code", approach with Maybe introduces compiler check on passing wring parameter. For example, if function expects to get Maybe, programmer can pass Nothing in function. On the other hand if function expects certain type and passing null is forbidden, programmer should change the signature of the function. Changing of signature produce a set of compiler errors that indicates breach of interaction contract that should be fixed as well to restore consistency.

VI. Conclusion

This article describes the reason why NPE occurs and a way to struggle against it in different programming languages.

For instance, some programming languages don't use null pointers at all. Other languages require direct specifying that null can be assigned to variable. For other languages like java was introduced similar way of preventing NPE.

The approach that was described in current article had helped to reduce count of NPE close to 0% during development process one of business applications.

Ought to remember that java still allows assigning null pointer to variable of Maybe type, but it should be treated as incorrect usage of development methods. Particularly Maybe is intended to change null pointers inside application code under development, therefore null pointers should be changed with Maybe.nothing() as quickly as possible in places of obtaining data places that are out of developer's control like remote sources.

References

[1] Flight 501 Failure. / Report by the Inquiry Board; Prof. J.L. Lions (on-line). [Accessed on May, 31]. Available from: https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[2] Software reliability. (on-line). [Accessed on May, 31]. Available from: http://www.tehprog.ru/index.php_page=lecture13.html

[3] @Nullable and @NotNull Annotations. (on-line). [Accessed on August, 30]. Available from: https://www.jetbrains.com/idea/help/nullable-and-notnull-annotations.html

This publication is the result of the project implementation:
TEMPUS CERES: Centers of Excellence for young RESearchers.
Reg.no.544137-TEMPUS-1-2013-1-SK-TEMPUS-JPHES