Reliable Object Construction in Object-oriented Programming

Alexander Prutzkow

***Abstract*—The construction of objects without validating the values of their fields leads to the need to introduce additional checks into the program text. The existing approaches (the Builder pattern and its modification, methods for checking the input parameters of constructors and methods) do not completely solve this problem. We introduce a design pattern for reliable object construction, which consists in adding two subsidiary classes to the original data-class. The first subsidiary class is a subclass of the data-class and has a factory method for creating objects. The parameter of this method is an object of the validator-class. Factory method is the only way to create objects. If the field values are invalid, then a default object (the Null Object pattern) is returned. The second subsidiary class is a class that validate the field values of an object of the original data-class. We demonstrate an example of reliable object construction. The advantages of the design pattern are separation of the object from its construction and parameter validation, reduction of duplication of program fragments, guaranteed object creation, no use of exceptions, null values, and nested classes.**

***Keywords*—Design patterns, object construction, object-oriented programming, reliable programming.**

I. Introduction

Reliable programming is a set of programming techniques that prevent unpredictable program behavior. "The reliability of a programming system is not only determined by the number of errors to be expected, but also by its behavior in error situations" [1].

The set of techniques includes:

– validation of input data;

– checking the result of the call to the external system;

– data protection from accidental changes.

Reliable programming is the foundation for defensive programming. Defensive programming is a set of programming techniques that prevent unauthorized use vulnerabilities in a program and eliminate the effects of malware.

II. Approaches to object construction

There are various approaches to object construction. Some of them can be used to construct objects reliably.

Originally the Builder pattern was as follows [2]. The B object-builder creates an object C, sets its parameters and returns as a result of one of its methods. Parameters mean the values of the C object fields or a set of more objects included in the C composite object.

The advantages of this pattern are:

(b1+) providing an uniform interface for constructing an object;

(b2+) separation of the object and its construction;

(b3+) the ability to adapt the construction of an object for a specific task.

The main disadvantage of this pattern is (b1–) the presence of a rigid connection between the B object-builder and the C object [3], [4] and duplication of program fragments. This leads to the fact that changes made to the C object class often require changes to the B object class.

J. Bloch proposed a modification of the Builder pattern [5] for the Java programming language, which is as follows. A nested class is added to the class whose the C object you want to construct. The object of the nested class is the B object-builder. A feature of nested classes in the Java language is the ability to access static and non-static (after object creation) fields, constructors, and methods of an outer class, including those with the private specifier [6]. The C object class constructors have the private specifier, so the C object is constructed by the B object method. An example of the implementation of this pattern can be found in [7].

The advantages of this modification of the Builder pattern are [5]:

(m1+) the only way to create objects through the method of the nested class;

(m2+) alternative to constructors with various combinations of parameters (telescopic constructors);

(m3+) the ability to use in the object hierarchy.

The disadvantages of the modification are:

(m1–) disadvantage (b1–);

(m2–) nested classes make it harder to understand outer classes.

III. Input validation steps

All data inputting the program is considered unverified.

Input validation includes the following steps [7]:

(1) Checking the data source. Before you start actually checking the data, you need to make sure that the data comes from a trusted source. A trusted source can be identified by an IP address or a pre-issued identification number.

(2) Checking the size of the data. Before doing a deep data check, it is necessary to check the length of the received data. For example, the received data of the phone number can exceed 1 gigabyte, which exceeds all reasonable limits.

Manuscript received January 15, 2022.

A. Prutzkow is with the Utkin Ryazan State Radio Engineering University, 390005, Gagarin str., 59/1, Ryazan, Russia, with Pavlov Ryazan State Medical University, 390026, Vysokovol'tnaja str., 9, Ryazan, Russia, and with Yesenin Ryazan State University, 390000, Svoboda str., 46, Ryazan, Russia (e-mail: mail@prutzkow.com).

(3) Checking the lexical content of the data. If the size of the data is within a certain range, then it is further checked whether the data consists of valid characters. For example, a phone number contains only numbers.

(4) Data syntax checking consists in making sure that the data matches a specific pattern. For example, a telephone number is a sequence of a country code, a region or provider code, and a subscriber number.

(5) Checking the semantics of data consists in checking the existence or validity of such data. For example, is the phone number provided to the subscriber or is it blocked.

Therefore, when creating objects, it is required to check the parameters of their constructors and / or methods.

IV. Approaches to validating constructor and method parameters

Validate class [8] from the org.apache.commons.lang3 package has methods for validating parameters: for a null value, for matching a regular expression, for whether a condition is true (see, for example [7]). If the validation fails, an exception is thrown.

Starting with Java EE version 6, it introduces the Bean Validation technology [9]. Constraints are described by special annotations of classes, fields, methods and their parameters. After creating an object of a class with annotations, it is passed as a parameter to the validation method. The check results in many constraint violations.

V. Purpose of the study

The existing approaches to constructing objects and validating parameters have disadvantages. So purpose of the study is to develop a reliable object construction pattern with parameter validation of constructors and methods that eliminates or mitigates these disadvantages.

VI. Reliable Object Construction Pattern

We propose the following design pattern for reliable object construction.

Let there be a C data-class (Listing 1). It needs to add subsidiary classes without modifying the C class so that all of these classes together provide reliable C object construction. We use the approach proposed by J. Bloch, with a constructor whose parameter is an object to construct an object of the class.

Listing 1. Original data-class example

```
1  public class DataClass {
2      private int var1;
3      private String var2;
4
5      public DataClass() { }
6
7      public DataClass(int var1, String var2) {
8          this.var1 = var1;
9          this.var2 = var2;
10     }
11
12     public final int getVar1() {
13         return this.var1;
14     }
15
16     public final String getVar2() {
17         return this.var2;
18     }
19
20     @Override
21     public String toString() {
22         String className = this.getClass().getSimpleName();
23         return String.format("%s [%d, %s]", className, this.var1, this.var2);
24     }
25 }
↳
```

Two subsidiary classes are needed to construct objects reliably:

F: data-class with a factory method – subclass of the C class (Listing 2);

V: class for validating field values of the C class object (Listing 3).

The F data-class must contain:

(f1) Constant field with a Null Object [10] (Listing 2, line 2).

(f2) Overridden superclass constructors with private specifier (lines 4, 6-8).

(f3) Constructor creating an object using a superclass object (lines 10-12).

(f4) A static method that creates an object (factory method) with the V validator-class (lines 14-24).

Listing 2 . The F data-class example with factory method

```
1  public class VerifiedDataClass extends DataClass {
2      public static final VerifiedDataClass DEFAULT = new VerifiedDataClass();
3
4      private VerifiedDataClass() { }
5
6      private VerifiedDataClass(int var1, String var2) {
7          super(var1, var2);
8      }
9
10     private VerifiedDataClass(DataClass dataClass) {
11         super(dataClass.getVar1(), dataClass.getVar2());
12     }
13
14     public static VerifiedDataClass createReliably(DataClassValidator dataClassValidator) {
15         boolean isValid = dataClassValidator.isValid();
16         if (isValid) {
17             DataClass dataClass = dataClassValidator.getDataClass();
18             VerifiedDataClass verifiedDataClass =
19                     new VerifiedDataClass(dataClass);
20             return verifiedDataClass;
21         } else {
22             return VerifiedDataClass.DEFAULT;
23         }
24     }
25 }
↳
```

The V validator-class must include:

(v1) The C class field and other fields required for validation (Listing 3, line 2).

(v2) Constructor with the validated object of the C class as a parameter and, if necessary, with subsidiary parameters (lines 6-8).

(v3) Method for validating class constructor parameters (lines 10-16).

(v4) Method to get the value of the C class field (lines 26-28).

Consider the purpose of the parts of the F data-class.

(f1) Used in place of the null value or throwing an exception. Is of the same type as the F data-class.

(f2) Hide superclass constructors.

(f3) Copies the field values of the C superclass object to the fields of the F subclass object. Has the private specifier so that only methods of the class can call this constructor.

(f4) The only way to create an object of the F class from the outside, since the rest of the methods are hidden (f2, f3). Object construction is impossible without validating the parameters of its constructor. If the validation fails, then the Null Object (f1) is returned.

The purpose of the parts of the V validator-class of object field values is as follows.

(v1) Stores the values to be validated and the values needed for the validation, such as the limits of a range of valid values.

(v2) Initializes the fields of the object of the V validator-class.

(v3) The main method of the V validator-class. Used when creating an object of the F class.

(v4) Needed to retrieve the values of the fields of the C class object in the subsequent to construct the F class object.

The considered classes are used as follows (Listing 4, lines 1-4). Line 6 will print the false value, because the values of the class constructor parameters are valid. Line 7 will print the true value, because the object is an instance of the VerifiedDataClass class (the F class). The truth of the last condition confirms that the object was created after validating the parameters of its constructor.

VII. Conclusions

The features of the proposed design pattern for reliable object construction are as follows.

(1) The object is constructed:

– the only way is by the factory method;

– with the obligatory participation of a validator-class object;

– not using exceptions and null values;

– guaranteed; if errors are detected for parameters, a Null Object will be returned.

(2) A simple condition for validating the construction of an object without errors: it is necessary to compare the created object with a Null Object.

(3) The original data-class does not need to be modified.

(4) The object is separated from its construction.

(5) The original data-class may be immutable. Its getter methods are not used.

(6) The original data-class can be a superclass or a subclass.

(7) The F subclass of an object is a confirmation that the values of its fields have been validated.

Listing 3. The *V* class example for validating field values of the *C* class object

```
1  public class DataClassValidator {
2      private DataClass dataClass;
3
4      private DataClassValidator() { }
5
6      public DataClassValidator(DataClass dataClass) {
7          this.dataClass = dataClass;
8      }
9
10     public boolean isValid() {
11         boolean isDataClassValid = true;
12         isDataClassValid = isDataClassValid //
13                 && this.isVar1Valid() //
14                 && this.isVar2Valid();
15         return isDataClassValid;
16     }
17
18     private boolean isVar1Valid() {
19         return this.dataClass.getVar1() != 0;
20     }
21
22     private boolean isVar2Valid() {
23         return this.dataClass.getVar2() != null;
24     }
25
26     public final DataClass getDataClass() {
27         return this.dataClass;
28     }
29 }
```

↳

Listing 4. An example of using the reliable object construction pattern

```
1  DataClass dataClassObject = new DataClass(1, "Text");
2  DataClassValidator dataClassValidator = new DataClassValidator(dataClassObject);
3  VerifiedDataClass verifiedDataClassObject = VerifiedDataClass
4          .createReliably(dataClassValidator);
5
6  System.out.println(verifiedDataClassObject == VerifiedDataClass.DEFAULT);
7  System.out.println(verifiedDataClassObject instanceof VerifiedDataClass);
```

↳

Degree of elimination of the disadvantages of existing approaches:

(b1–) Not completely eliminated, but significantly. Subsidiary classes *F* and *V* depend on the *C* class weakly. There is no duplication of class fragments. When changing the fields of the *C* class, you need to change the constructor in the *F* class (Listing 2, lines 10-12) and the parameter validation method in the *V* class (Listing 3, lines 10-16).

(m2–) Nested classes are not used.

The disadvantages of the proposed pattern are determined by the lack of the "Factory Method" pattern [5]: factory methods are difficult to distinguish from other static methods (mitigating this disadvantage through naming is also found in [5]).

The software project with the classes demonstrated in the paper is available for free download from the Internet resource of the author [11].

REFERENCES

[1] H. Gerstmann, H. Diel, and W. Witzel, “The reliability of programming systems”. In C.E. Hackl (ed.) *Programming Methodology. IBM 1974. Lecture Notes in Computer Science*, 1975, no. 23. DOI: 10.1007/3-540-07131-8_23.

[2] E. Gamma, R. Helm, R. Johson, J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Pearson Education, 1994.

[3] S. Stelting, O. Maassen, *Applied java patterns*. Sun Microsystems Press, 2002.

[4] J. Hunt, *Scala Design Patterns. Patterns for practical reuse and design*. Springer, 2013.

[5] J. Bloch, *Effective java*, 3rd ed. Addison-Wesley, 2018.

[6] H. Schildt, *Java. The complete reference*, 11th ed. McGrow-Hill Publisher, 2019.

[7] D. Johnsson, D. Deogun, and D. Sawano, *Secure by design*. Manning, 2019.

[8] Validate (Apache Commons Lang 3.12.0 API) Available: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html

[9] E. Jendrock, R. Cervera-Navarro, I. Evans, D. Gollapudi, K. Haase, W. Markito, Ch. Srivathsa, *The Java EE 7 tutorial, release 7 for Java EE platform.* Oracle, 2013.

[10] B. Woolf, “Null Object”. In R. Martin, D. Riehle, F. Buschmann (eds.) *Pattern Languages of Program Design*, 1998, no. 3, pp. 5-18.

[11] A. Prutzkow, “Internet-resurs dlja razmeshchenija rezul'tatov nauchnoj i obrazovatel'noj dejatel'nosti” [Internet-resource for Scientific and Educational Work Result Publishing]. In *Vestnik Rjazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*, 2018, no.63, pp. 84-89. DOI: 10.21667/1995-4565-2018-63-1-84-89. [In Rus]