Automated Refactoring to the Null Object Design Pattern

Maria Anna G. Gaitani, Vassilis E. Zafeiris, N. A. Diamantidis, E. A. Giakoumakis

Department of Informatics, Athens University of Economics and Business, 76 Patission Str., Athens 104 34, Greece

Abstract

Context: Null-checking conditionals are a straightforward solution against null dereferences. However, their frequent repetition is considered a sign of poor program design, since they introduce source code duplication and complexity that impacts code comprehension and maintenance. The NULL OBJECT design pattern enables the replacement of null-checking conditionals with polymorphic method invocations that are bound, at runtime, to either a real object or a Null Object.

Objective: This work proposes a novel method for automated refactoring to NULL OBJECT that eliminates null-checking conditionals associated with optional class fields, i.e., fields that are not initialized in all class instantiations and, thus, their usage needs to be guarded in order to avoid null dereferences.

Method: We introduce an algorithm for automated discovery of refactoring opportunities to NULL OBJECT. Moreover, we specify the source code transformation procedure and an extensive set of refactoring preconditions for safely refactoring an optional field and its associated null-checking conditionals to the NULL OBJECT design pattern. The method is implemented as an Eclipse plug-in and is evaluated on a set of open source Java projects.

Results: Several refactoring candidates are discovered in the projects used...
in the evaluation and their refactoring lead to improvement of the cyclomatic complexity of the affected classes. The successful execution of the projects’ test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Runtime performance results highlight the potential for applying our method to a wide range of project sizes.

**Conclusion:** Our method automates the elimination of null-checking conditionals through refactoring to the **Null Object** design pattern. It contributes to improvement of the cyclomatic complexity of classes with optional fields. The runtime processing overhead of applying our method is limited and allows its integration to the programmer’s routine code analysis activities.

**Keywords:** refactoring, design patterns, null object, optional fields, null checks

1. **Introduction**

Dereferencing null object references leads to runtime errors and, thus, causes program termination or abnormal operation. In order to avoid such errors, the programmer needs to decide which object references can have a null value and introduce appropriate null-checking conditional statements to avoid null dereferences. Compile-time detection of null dereferences is an effective approach for the discovery of many null-related bugs [1] that is not yet integrated in popular languages such as Java or C#. Static code analysis techniques are, also, applied for the discovery of null dereferences (e.g. [2, 3]) and are gradually integrated in popular code review tools, such as FindBugs for Java [4, 5].

Although null-checking conditionals are a straightforward solution against null dereferences, their frequent repetition is, often, considered a code “smell”, i.e. a sign of poor program design. Fowler [6] and Kerievsky [7] document this code “smell”, in their books on software refactoring, as a source of code duplication that, also, increases code complexity and, thus, impacts its comprehension and maintenance. Fowler [6] focuses on repeated null checks, scattered in the code of a method or class, that refer to a specific object reference. Kerievsky [7] emphasizes on null checks that involve class fields. Both works suggest the elimination of the null-checking conditionals through refactoring to the **Null Object** design pattern [8, 9]. The **Null Object** design pattern...
hides the absence of an object (null value) with a substitutable alternative object, namely the *Null Object*, that has the same interface as the real object, but provides a default “do nothing” behaviour [9]. The term default “do nothing” behaviour denotes that all methods of the *Null Object* class are implemented so as to either have an empty body or return default results. The lifecycle of a *Null Object* ends with the assignment of a non-null value to the object reference.

The *NULL Object* design pattern enables the replacement of null-checking conditionals with polymorphic method invocations that are bound, at runtime, to either a real object or a *Null Object*. The pattern removes duplicate code fragments that are relevant to (a) null-checks on the same object references and (b) repeated “do nothing” behaviour that is executed in the case of a null reference. The latter is extracted to appropriate methods of the *Null Object* class. Besides its contribution to code simplicity, *NULL Object* enables easy and safe program extensions. Specifically, adding method invocations to a potentially null object reference gets simpler and less error prone, since the programmer is not required to remember and introduce relevant null checks. Finally, *NULL Object* increases reusability, as instances of a *Null Object* class can be used in multiple cases of null-checks on the same object type.

This paper deals with the problem of automated refactoring to the *NULL Object* design pattern. It complements the works of Fowler [6] and Kerievsky [7], focusing on the mechanics of the manual application of the refactoring, with a novel method for automated discovery of null-checking conditionals that can be effectively refactored to *NULL Object*. Our analysis focuses on special cases of null-checking conditionals that are encountered in classes with optional collaborators, i.e., with fields that are not always initialized. These conditionals protect optional field dereferences and enclose the behaviour of an “empty” collaborator. Moreover, we specify an extensive set of refactoring preconditions that mark cases that can be safely refactored without changing the external behaviour of the system. The refactoring identification procedure is complemented with a detailed description of the source code transformation for applying the *NULL Object* design pattern to a given optional field and its respective null-checking conditionals. Our method for automated refactoring to *NULL Object* has been implemented as part of the JDeodorant Eclipse plug-in [10], a tool for the automation of complex Java code refactorings. Moreover, it has been experimentally evaluated on a set of open source Java projects. Several refactoring candidates have been
discovered in these projects and their refactoring lead to improvement of the cyclomatic complexity of the affected classes. The successful execution of the projects’ test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Finally, runtime performance results highlight the potential for applying our method to a wide range of project sizes.

The rest of this paper is structured as following: section 2 presents relevant work on the research area of refactoring to design patterns. Section 3 presents the Null Object design pattern, its alternative implementations and introduces appropriate terminology that will be used in this paper. Section 4 specifies our method for automated identification of refactoring candidates and their elimination through the Null Object design pattern. Section 5 presents an evaluation of this work on the basis of a prototype implementation integrated to the JDeodorant Eclipse plug-in [10]. Finally, Section 6 summarizes the conclusions of this work.

2. Related Work

Our work contributes to the research area of automated refactoring to design patterns. Refactoring to patterns aims at the elimination of design flaws through the introduction of appropriate design patterns. The automation of refactoring tasks enables integration of the continuous design improvement practice to the development workflow. This section provides a review on methods for automated refactoring to design patterns. It encompasses approaches relevant to both structural (Abstract Factory, Composite) and behavioural (Decorator, Template Method, State/Strategy) design patterns. As concerning refactoring to Null Object, this work is the first that studies its automation. For an extensive review on the broader research area of software refactoring, the reader may refer to the work of Mens and Tourwe [11].

Abstract Factory. Refactoring to Abstract Factory is among the earlier approaches on refactoring to patterns. Specifically, Tokuda and Batory [12] proposed the introduction of the design pattern as a composition of parameterized object-oriented transformations. The method provides a specification of these primitive transformations and applies them through appropriate tool support. The introduction of the Abstract Factory is demonstrated through a simple case study.
Refactoring to design patterns is also treated as a series of mini-transformations in the methodology proposed by Cinneide and Nixon [13]. A mini-transformation comprises pre-conditions, post-conditions, transformation steps and an argument over how the mini-transformation supports behaviour preservation after its application. The methodology is primarily focused on structure-rich, rather than behavioural patterns, and is applied to refactoring to Abstract Factory.

A logic programming approach to refactoring to patterns has been proposed by Jeon et al. [14]. The method employs logic inferencing for the identification of refactoring opportunities in a Java code base, and the subsequent, selection of an appropriate strategy for source code transformation. Inferencing is based on the extraction of system design from Java code and its representation as a set of Prolog-like predicates that are then converted to Prolog facts. Inferencing rules are, also, specified for each target pattern, that are transformed to Prolog rules. The identification of refactoring opportunities takes place through issuing of Prolog queries. The method applied for the discovery of refactoring candidates to Abstract Factory. The application of the refactoring is based on the mini-transformations approach of Cinneide and Nixon [13].

*Composite.* Jebelean et al. [15] use logic metaprogramming for the detection of incorrect applications of the Composite design pattern. The approach involves transformation of the Java project’s Abstract Syntax Tree (AST) into Prolog facts through the JTransformer engine [16]. Problematic code fragments are identified through the definition of appropriate Prolog rules. Ajouli et al. [17] focus on the automated transformation of a Visitor pattern instance to Composite and vice versa. The transformation is based on a set of refactoring preconditions that ensure its correct application and reversibility. Moreover, the authors present variations of the base transformation for handling special cases of relaxed preconditions.

*Decorator.* Rajesh and Janakiram [18] employ logic programming for the identification of refactoring candidates (or Intent Aspects in the terminology of the paper) to the Decorator design pattern. The method involves construction of the Java project’s AST and generation of Prolog facts that reflect its design. Fact generation is based on Predicate Templates, i.e. predefined facts that are introduced to the facts-base during traversal of the project’s AST. Moreover, the authors specify Prolog rules for the identification of re-
factoring candidates to Decorator. The application of the refactoring is enabled through a third party refactoring tool.

Template Method. A first approach toward semi-automated refactoring to Template Method has been described by Juillerat and Hirsbrunner [19]. It applies text-based clone detection techniques on pairs of methods that are indicated by the programmer. The methods must belong to different classes that share a common abstract class ancestor. The authors employ and extend existing techniques for clone detection in order to identify the common and different statements of the compared methods. The differences are extracted as new methods for each child class while the common parts of the initial methods are moved as a single template method to the common superclass of the refactored classes. Finally, an additional abstract method is created in the superclass that has the same signature with the extracted methods.

Hotta et al. [20] apply a more advanced clone detection technique, based on Program Dependence Graphs (PDGs), for the detection of refactoring candidates to Template Method. This technique is more orientated toward the identification of behavioural clones, through the detection of isomorphic graphs on PDGs. Each identified clone pair is suggested as refactoring candidate, given that a set of refactoring preconditions is not violated. The preconditions are relevant to (a) the methods (candidate methods) that enclose each clone and (b) the potential for applying extract method to the common and different parts of the candidate methods. The refactoring candidates’ detection process has been implemented, but the programmer has to manually apply the suggested refactorings.

State/Strategy. Refactoring to State/Strategy focuses on the elimination of code flaws relevant to complex conditional statements. Tsantalis and Chatzigeorgiou [21] handle two cases of refactoring for simplification of conditional expressions that are specified by Fowler [6]: (a) replace type code with State/Strategy and (b) replace conditional logic with polymorphism. The method employs static analysis in order to identify conditional statements that can be eliminated through the invocation of polymorphic operations. The refactoring identification algorithm and the refactoring procedure have been implemented and integrated in the JDeodorant Eclipse plug-in [10].

The automated introduction of the State/Strategy design pattern and its discrimination from the State pattern has been proposed by Christopoulou et al. [22]. Suggested refactorings comprise conditional statements that are
characterized by analogies to the State/Strategy design pattern, in terms of the purpose and selection mode of strategies. The approach, also, specifies the procedure for refactoring to State/Strategy the identified conditional statements. For special cases of these statements, a technique is proposed for total replacement of conditional logic with method calls of appropriate concrete strategy instances. The identification algorithm and the refactoring procedure are also implemented and integrated in the JDeodorant Eclipse plug-in.

3. The Null Object design pattern

The Null Object pattern in object oriented design has been specified and documented by Woolf [8], initially, and later by Henney [9]. Its purpose is to hide the absence of a class collaborator in cases that it is not required to do anything, i.e., its behaviour must be substituted by a default “do nothing” behaviour [9]. The default “do nothing” behaviour is represented by a substitutable object, the Null Object, that has the same interface as the real object. Null Object methods either have an empty body or return default results.

The use of the Null Object is recommended in cases that a client class has a reference to an object whose value may be equal to null. In order to avoid null dereferences, any use of the object is enclosed in appropriate null-checking conditional statements. Figure 1 presents a simple example from [9] that illustrates the use of the Null Object pattern. The Service class in Figure 1a logs its execution status through invocations to a Log instance. The Log field is not always initialized and, thus, write(String) method calls are guarded with null-checking conditional statements. Such repetitive null-checks increase the conditional complexity of the Service class and introduce code duplication. The duplicate code is part of the else branch of the null-checking conditional and may comprise: (a) no functionality (as is the case in this example), (b) default behaviour (e.g. print statements),

---

1Java 8 introduces the Optional<T> type in order to enable the declaration of potentially empty object references and the representation of the absence of an object with a non-null value. An Optional<T> instance is a container for an object of type T or a null value [23]. However, Optional<T> instances are not Null Objects, since they don’t support the interface of T, but, instead, provide utility methods to manage the presence or absence of an object.
Figure 1: Improving null-checking code with **Null Object** [9].

(c) raise of exceptions, (d) variable assignments to a default value, etc. The application of the **Null Object** design pattern moves the default behaviour of null-checking conditionals to the *Null Object* class and replaces null-checks with polymorphic invocations of either the real object or the *Null Object*. Figure 1b presents the *Service* class, refactored to the **Null Object** design pattern. The *Log* field is initialized, by default, with a *NullLog* instance and null-checking conditionals are eliminated.

The simple example of Figure 1 features some key benefits provided by the appropriate application of the **Null Object** design pattern. First of all, the pattern applies the mechanisms of inheritance and polymorphism for the elimination of repetitive null-checks, improving in this way the conditional complexity of the source code. Besides its contribution to code simplicity, **Null Object** contributes to safer extensions and easier maintenance of the affected code. Specifically, the programmer may add further invocations to the *Log* field in the *Service* class without the risk of introducing errors due to neglected null-checks. Finally, the pattern favors reusability, since a *NullLog* instance can be reused in any other context that requires a default “do nothing” behaviour of type *Log*.

As concerning the structure of the **Null Object** design pattern, three
Figure 2: Alternative NULL OBJECT implementations.
different variants are documented in the literature that are presented in Figure 2. Part (a) of Figure 2 illustrates the Null Object implementation that was initially proposed by Fowler [6] (Simple Implementation). The Null Object class (NullComponent) inherits from the real object class (Component) and overrides all its methods with appropriate “do nothing” behaviour. The client class (Context) has a reference of type Component that is initialized with a Component or NullComponent instance. An alternative structure of the pattern is available in part (b) of Figure 2 (Common Ancestor Implementation). In variant (b), both Component and NullComponent inherit from an abstract class (AbstractComponent) that declares abstract operations for all non-private methods of Component [8]. Context class declares an AbstractComponent variable to reference the Component or NullComponent instance. A similar approach, proposed by Henney [9] and depicted in part (c) of Figure 2, uses an interface (IComponent) in place of the AbstractComponent class (Common Interface Implementation). In this case, both Component and NullComponent implement this interface, while Context class references them as IComponent instances.

Refactoring to the Simple Implementation variant does not require modifications to the Component class. Thus, this Null Object variant is appropriate for cases that the source code of Component is unavailable, e.g., it is a library class. A drawback of Simple Implementation is that it may lead to maintainability issues in case of method additions to the Component class. Specifically, should the developer neglect their redefinition with proper do “nothing behaviour” in the NullComponent class, the latter will inherit the Component implementation that may result to undesired behaviour during software execution. Moreover, this variant may lead to violations of the Liskov’s Substitution Principle [24]. Such issues are not present in the other two variants where the NullComponent class does not inherit or override any concrete implementation. A limitation of the Common Ancestor Implementation is that it cannot be applied in cases that Component inherits from another subclass and multiple inheritance is not supported. On the other hand, the application of the Common Interface Implementation may expand, from protected or package to public, the visibility of Component methods that are also declared in the IComponent interface. Such modifications can potentially impact the encapsulation of Component class. Despite these limitations, variants (b) and (c) are a better alternative than Simple Implementation and can be interchangeably applied in many cases. In this work we adopt the Common Ancestor Implementation, since it has less undesired
effects on the refactored code, and study its automated introduction for the elimination of null-checking code fragments.

4. Automated refactoring to the Null Object design pattern

This section presents an algorithm for automated identification of refactoring opportunities to Null Object. Moreover, it specifies appropriate refactoring preconditions and the transformation procedure for applying the design pattern to selected code fragments.

4.1. Identification of candidate refactorings

The identification of refactoring opportunities to Null Object in a given software project is based on static analysis of the source code. We propose an algorithm that processes all project classes by analysing their Abstract Syntax Tree (AST) representation. A refactoring candidate comprises a pair \((C, F)\) where \(C\) is a class that has the role of the Context and \(F\) is a field of \(C\) that is optional and can be initialized with a Null Object instance after the refactoring.

The refactoring identification algorithm focuses on the discovery of fields that may not be initialized in certain instantiations of their respective Context class and, thus, their usage in Context methods is accompanied by repetitive null checks that serve the avoidance of Null Pointer Exceptions (NPEs). These fields will be, henceforth, referred to as Optional Fields. For each Optional Field the algorithm discovers conditional statements that safeguard method invocations to that field. These conditional statements are characterized by a null-checking conditional expression against the optional field and will be referred to as Guarded Field Invocation Conditional (GFI-Conditionals).

Prior to the specification of the refactoring identification algorithm, our analysis will focus on the notions of Optional Field and Guarded Field Invocation Conditional. The definition of these concepts, along with details on their identification in a given Context class, contributes to a more concise and intuitive description of the identification algorithm. The formal parts of these definitions are provided as predicates that qualify program constructs (e.g. fields, methods, statements). We have adopted a rule-based approach, also applied in [25], for the specification of these predicates and the required notation is depicted in the conceptual diagram of Figure 3. The types, properties and relationships, represented in this diagram, form a vocabulary for
Figure 3: Conceptual diagram of the notation used in the proposed method (partially based on [25]).
the specification of the proposed method for identification of candidate refactorings. A program, in this model, is a single object that has as property (program.classTypes) all class types that are part of the project’s code base (ClassType instances).

4.1.1. Optional Field

A field \( f \), declared in a given Context class, is considered as an Optional Field if there exist instantiations of Context that omit its initialization. A basic premise is that \( f \) is either not initialized in the class declaration or it is initialized with a null value. A formal description of the rules that apply for the classification of a field as optional is given by predicate isOptional(Field, Class) (Definition 1). Note that, the helper predicate instanceof is satisfied in case that the type of the first argument equals to or is a subtype of the second argument.

**Definition 1. Predicate isOptional(Field, Class)**

\[
\text{isOptional}(\text{Field } f, \text{Class context}) \equiv \\
\quad f \in \\text{context.fields} \land \\
\quad \text{instanceof}(f.\text{type}, \text{ClassType}) \land f.\text{type} \neq \text{String} \land \\
\quad (f.\text{initializer} = \emptyset \lor \text{instanceof}(f.\text{initializer}, \text{NullLiteral})) \land \\
\quad (\exists c \in \text{context.methods} : c.\text{constructor} = \text{true} \land \\
\quad \quad (\forall s \in c.\text{methodBody.statements} : \neg \text{defines}(s, f)) \lor \\
\quad \quad (\exists m \in \text{context.methods} : \text{isSetter}(m, f)))
\]

Definition 1 states that a field \( f \), that is uninitialized or initialized with a null value and does not have a primitive or String type, can be characterized as optional if either one of the following conditions is satisfied:

- \( f \) is not defined in at least one constructor of the Context class. A constructor defines the value of \( f \) if its body contains an assignment statement for \( f \) or an invocation of a method that defines \( f \) to a non-null value. Predicate defines(Statement, Field) is satisfied by any statement that directly or indirectly assigns a non-null value to the given field.

- Context declares a non-private mutator method (or setter method in Java terminology) for field \( f \). A method is a mutator for a given field if predicate isSetter(Method, Field) is satisfied (refer to Appendix A
for its definition). The presence of a non-private setter for \( f \) enables a client of Context to potentially define the value of this field to a null value. Our method identifies this field as optional, since the contingency of a null value, often, leads programmers to protect access to the field’s members through null-checking conditionals.

The evaluation of \( \text{defines}(s, f) \), in case that statement \( s \) includes a method invocation \( \text{inv} \), involves data flow analysis through construction of the Program Dependence Graph (PDG) for \( \text{inv.declaringMethod} \) (see conceptual model of Figure 3). Given a Context constructor, this analysis is not applied to all method invocations that are declared in the constructor body. Instead, it includes method invocations that can potentially assign a value to field \( f \) and may comprise:

- methods invoked on the current instance of the Context class (this reference in Java). Given a method invocation \( \text{inv} \), the predicate \( \text{defines}(\text{inv}, f) \) is satisfied if \( f \) is among the defined variables of the PDG constructed for the method body of \( \text{inv.declaringMethod} \).

- methods that receive the current Context instance as actual parameter. Let \( \text{inv} \) be a method invocation and \( p \) be the name of the typical parameter of type Context in the \( \text{inv.declaringMethod} \). The predicate \( \text{defines}(\text{inv}, f) \) is satisfied if \( p.f \) is among the defined variables of the PDG for \( \text{inv.declaringMethod} \).

- invocation of another Context constructor (e.g. this() in Java). Each statement \( s \) of the constructor body is recursively analyzed through evaluation of the \( \text{defines}(s, f) \) predicate.

The PDG model used in this work is an extended version of the base PDG specification, initially proposed by Ferrante et al. [26], and enhanced with additions for support of break, continue, try/catch statements. A comprehensive presentation of that PDG model is included in the work of Tsantalis and Chatzigeorgiou [27] that employs a PDG for automation of the extract method refactoring.

4.1.2. Guarded Field Invocation Conditional

A Guarded Field Invocation Conditional (GFI-Conditional) represents a conditional statement with at most two branches (if or if/else) whose
conditional expression is a simple null-checking comparison. The comparison refers to a field of the class that declares the conditional statement (Context class). A GFI-Conditional safeguards access to members (fields or methods) of an Optional Field against Null Pointer Exceptions (NPEs). Our work focuses on the elimination of certain variants of GFI-Conditionals through the NULL OBJECT design pattern.

Let optionalField be an optional field declared in class Context. The proposed refactoring identification algorithm focuses on four variants of GFI-Conditionals for optionalField that are included in Listing 1. A formal description of the rules that determine whether a given conditional statement matches one of these variants is available in Appendix A.

Listing 1: Variants of a GFI-Conditional statement for a given field.

```c
/* Variant 1: Single branch GFI-Conditional, inequality comparison */
if (optionalField != null){
  /* field invocation fragment */
}

/* Variant 2: Two-branch GFI-Conditional, inequality comparison */
if (optionalField != null){
  /* field invocation fragment */
  /* other statements */
} else {
  /* raise of application specific exception */
}

/* Variant 3: Single branch GFI-Conditional, equality comparison */
if (optionalField == null){
  /* raise of application specific exception */
}

/* Variant 4: Two-branch GFI-Conditional, equality comparison */
if (optionalField == null){
  /* raise of application specific exception */
} else {
  /* field invocation fragment */
  /* other statements */
}
```
Figure 4: Case study class with GFI-Conditional statements.
Figure 4 presents an intuitive example of a Context class that includes three variants of GFI-Conditional statements. Specifically, the ShoppingCart class has the role of Context and buyer is an optional field of type Customer. This case study implements a shopping cart that provides base functionality in the absence of a Customer instance (non-authenticated customer), i.e. it supports updating of stored items and their quantities, as well as viewing their total cost. On the other hand, the shopping cart provides enhanced functionality to an authenticated customer (non-null buyer), e.g. discounts based on user profile. The ShoppingCart, also, requires an authenticated customer for placing an order and processing payment information. Thus, the buyer field is optional for part of the ShoppingCart lifecycle and, therefore, its method invocations are protected with GFI-Conditionals.

A detailed description of the properties of each GFI-Conditional variant is provided below. The case study of Figure 4 will be used as a reference for examples that clarify the concepts introduced in this description.

**Variant 1 GFI-Conditional.** The first variant of a GFI-Conditional statement is characterized by a conditional expression that is satisfied by a non-null optionalField. The conditional’s body comprises statements that are strictly optionalField method invocations. This type of code fragment will be, henceforth, referred to as FieldInvocationFragment. The rationale of the aforementioned constraint on the GFI-Conditional’s body is twofold: (a) it enables precise identification of conditional statements that incorporate the semantics of a GFI-Conditional and (b) it allows for straightforward elimination of the GFI-Conditional during refactoring to NULL OBJECT through its direct replacement by the FieldInvocationFragment. Thus, after refactoring, the FieldInvocationFragment executes polymorphically either the actual optionalField behaviour or the default behaviour of a Null Object that is assigned to optionalField instead of a null reference.

Let \( m_o \) be an optionalField method that is invoked in the FieldInvocationFragment of a Variant 1 GFI-Conditional and has a void return type. This method is marked by our refactoring identification algorithm as satisfying the predicate emptyOnNull(optionalField, \( m_o \)). All methods of an optionalField that satisfy this predicate will have an empty body implementation in the Null Object. A formal specification of the emptyOnNull(Field, Method) predicate is provided in Definition 2. The ownerClass property of a field or method, in this definition, refers to the class that declares them. Moreover, predicate isGFIv1Conditional(f, b) determines
whether conditional statement $b$ is a Variant 1 GFI-Conditional for field $f$. Predicate $\text{isVoidType}(m.\text{returnType})$ determines whether the return type of $m$ is void, while $\text{invokesFieldMethod}(i, f, m)$ states that method invocation expression $i$ corresponds to an invocation of method $m$ on field $f$. The definitions of these predicates are available in Appendix A.

**Definition 2.** Predicate $\text{emptyOnNull(}Field, Method\text{)}$

$$
\text{emptyOnNull}(\text{Field } f, \text{Method } m) \equiv
f.\text{type} \in \text{program.classTypes} \land
f.\text{type} = m.\text{ownerClass.type} \land
\text{isVoidType}(m.\text{returnType}) \land
(\exists m' \exists b \exists i : m' \in f.\text{ownerClass.methods} \land
b \in m'.\text{methodBody.conditionals} \land
i \in b.\text{thenBlock.methodInvocations} \land
\text{isGFIv1Conditional}(f, b) \land \text{invokesFieldMethod}(i, f, m))
$$

Let $m_r$ be an optionalField method that is invoked in the FieldInvocationFragment and returns a primitive or string value that is assigned to a local variable $v$. Moreover, let $d_i$ be an expression that is assigned to $v$ in the execution path $i$ by the last assignment statement that precedes $m_r$ invocation. Since the $m_r$ invocation may be part of multiple execution paths, variable $v$ may be defined to different expressions $d_i$. In case that all expressions $d_i$ correspond to the same constant literal $c$, method $m_r$ is marked by the refactoring identification algorithm as satisfying the predicate $\text{returnConstantOnNull(optionalField, } m_r, c\text{)}$. The methods of an optionalField that satisfy this predicate will have a “do nothing” implementation in the Null Object comprising a single statement that returns constant $c$. Definition 3 provides a formal specification of the predicate. The helper predicate $\text{isLiteralType}(m.\text{returnType})$ states that method $m$ has a primitive or String return type. The function $\text{getLastDefinition(}\text{VariableAccess, Statement}\text{)}$ returns the set of all literal expressions $d_i$ that are assigned to a variable (first argument) just before the execution of the statement provided as second argument in all execution paths.

The evaluation of $\text{getLastDefinition(}\text{VariableAccess, Statement}\text{)}$ involves the construction of the Program Dependence Graph (PDG) for the method that declares the Variant 1 GFI-Conditional. In a next step, the PDG node
that corresponds to the assignment of the \( m_r \) invocation to variable \( v \), is analyzed in terms of its incoming data dependencies for variable \( v \). On the basis of the data dependence edge definition [26], an incoming data dependence for variable \( v \) starts from a PDG node \( N_{s,i} \) that changes the value of \( v \). Moreover, no intervening definition of \( v \) exists in the control flow path from statement \( N_{s,i} \) to \( N_t \). Since \( v \) is of primitive or string type, its value can be modified by an assignment statement. Thus, \( N_{s,i} \) corresponds to the assignment \( v \leftarrow e_i \) where \( e_i \) is an expression. The analysis of all incoming data dependence edges to node \( N_t \) for variable \( v \) results to the set \( E_t = \cup \{e_i\} \) of all possible expressions \( e_i \) that may be assigned to \( v \) across all control flow paths to \( N_t \). The function \( \text{getLastDefinition}(\text{VariableAccess, Statement}) \) returns a subset of \( E_t \) where each \( e_i \) is a literal expression.

**Definition 3.** Predicate \( \text{returnConstantOnNull(Field, Method, Literal)} \)

\[
\text{returnConstantOnNull}(\text{Field } f, \text{Method } m, \text{Literal } c) \equiv \\
f.type \in \text{program.classTypes} \land \\
f.type = m.ownerClass.type \land \text{isLiteralType}(m.returnType) \land \\
(\exists m' \exists b \exists a : m' \in f.ownerClass.methods \land \\
b \in m'.methodBody.conditionals \land \\
a \in b.thenBlock.assignments \land \\
\text{isGFIv1Conditional}(f, b) \land \\
\text{instanceof}(a.leftHandSide, \text{VariableAccess}) \land \\
a.leftHandSide \in b.localVariableAccesses \land \\
\text{invokesFieldMethod}(a.rightHandSide, f, m) \land \\
\text{getLastDefinition}(a.leftHandSide, a) = \{c\})
\]

An example of a Variant 1 GFI-Conditional statement is illustrated in Figure 4. The statement is declared in the \( \text{getDiscount()} \) method and updates the \textit{discount} local variable to a non-zero value if \textit{buyer} is not null. The FieldInvocationFragment of the GFI-Conditional comprises a single method invocation statement that satisfies the \textit{returnConstantOnNull(buyer, getDiscount, 0)} predicate. The reason is that the \textit{discount} local variable, receiving its return value, is initialized to a constant value (zero) before the method invocation. This constant value will be returned as a default result in the \textit{getDiscount()} method implementation in the \textit{Null Object}.  

19
Variant 2 GFI-Conditional. The Variant 2 GFI-Conditional includes an else branch that throws an application specific exception in the case of a null optionalField. For brevity reasons, this exception will be referred to as NullFieldException. The body of the if branch must begin with a FieldInvocationFragment and may be followed by any statement. A Variant 4 GFI-Conditional is equivalent to a Variant 2 conditional with its if and else branches reordered and a complementary null-checking condition.

A Variant 2 GFI-Conditional is replaced by its if branch body during refactoring to Null Object, as is the case in Variant 1. The FieldInvocationFragment, that was unreachable in the case of a null optionalField, now executes polymorphically the methods of a Null Object and raises a NullFieldException exception for behaviour preservation. The exception is thrown by the fragment’s first statement and the respective optionalField method \( m_1 \) is marked by the refactoring identification algorithm as satisfying the predicate \( \text{nullFieldExceptionOnNull}(\text{optionalField}, m_1) \). The predicate states that \( m_1 \) must have a single throw NullFieldException statement as implementation in the Null Object. Definition 4 provides a formal specification for the nullFieldExceptionOnNull(Field, Method) predicate. Predicates isGFIv2Conditional(f, b) and isGFIv4Conditional(f, b) decide whether a conditional statement \( b \) is, respectively, a Variant 2 or Variant 4 GFI-Conditional for field \( f \). The definitions are available in Appendix A.

**Definition 4.** Predicate nullFieldExceptionOnNull(Field, Method)

\[
\text{nullFieldExceptionOnNull}(\text{Field } f, \text{Method } m) \equiv \\
\text{nullFieldExceptionOnNull}(\text{Field } f, \text{Method } m) \equiv \\
f.type \in \text{program.classTypes} \land \\
f.type = m.\text{ownerClass}.\text{type} \land \\
(\exists m' \exists b : m' \in f.\text{ownerClass}.\text{methods} \land \\
b \in m'.\text{methodBody}.\text{conditionals} \land \\
((\text{isGFIv2Conditional}(f, b) \land \\
\text{invokesFieldMethod}(b.\text{thenBlock}.\text{methodInvocations}[1], f, m)) \\
\lor (\text{isGFIv4Conditional}(f, b) \land \\
\text{invokesFieldMethod}(b.\text{elseBlock}.\text{methodInvocations}[1], f, m))))
\]

In the example of Figure 4, method updatePaymentInfo(String, Date) declares a Variant 2 GFI-Conditional. Its FieldInvocationFragment includes both method invocation statements of the if branch. The first me-
method invoked in the FieldInvocationFragment is associated with the predicate `nullFieldExceptionOnNull(buyer, setCreditCardNumber)`, while the second method is not associated with any predicate. `CustomerNotFoundException` has the role of the `NullFieldException` and will be thrown in the body of the `setCreditCardNumber(String)` method in the `Null Object` implementation.

**Variant 3 GFI-Conditional.** A Variant 3 GFI-Conditional is a single-branch conditional statement that checks for `optionalField` equality with null in its conditional expression. Its body comprises a throw `NullFieldException` statement. In Figure 4, a Variant 3 GFI-Conditional is declared in the start of `placeOrder()` method.

**Unguarded Optional Field Invocations.** Methods $m_n$ of an `optionalField` that are invoked outside GFI-Conditionals are qualified by the refactoring identification algorithm as satisfying the predicate `nullPointerExceptionOnNull(optionalField, $m_n$)`. The predicate states that their behaviour in the `Null Object` involves raising a `NullPointerException`, reflecting the behaviour prior to refactoring when invoked on a null `optionalField` reference. Definition 5 provides a formal specification for the `nullPointerExceptionOnNull(Field, Method)` predicate.

**Definition 5.** Predicate `nullPointerExceptionOnNull(Field, Method)`

\[
\text{nullPointerExceptionOnNull}(\text{Field } f, \text{Method } m) \equiv \\
f.type \in \text{program.classTypes} \land \\
f.type = m.ownerClass.type \land \\
(\exists m' \exists s : m' \in f.ownerClass.methods \land \\
s \in m'.methodBody.methodInvocations \land \\
\text{invokesFieldMethod}(s.invokeExpression, f, m) \land \\
\neg \text{isGuardedInvocation}(f, s))
\]

The helper predicate `isGuardedInvocation(f, s)` states that a specific method invocation $s$ is nested in a conditional statement that protects it against null dereferences on field $f$. The proposed methodology evaluates the `isGuardedInvocation(Field, MethodInvocation)` predicate through analyzing the Abstract Syntax Tree (AST) of the method that declares the $s$ statement. Specifically, the predicate is satisfied on the basis of the following rules:
• statement $s$ is nested recursively in the $\textit{if}$ branch of a conditional statement with the following properties: (a) its conditional expression is a conjunction $e_1 \land e_2 \ldots \land e_n$ of boolean expressions $e_i$, and (b) at least one sub-expression $e_i$ is of the form $f \neq \textit{null}$.

• statement $s$ is nested recursively in the $\textit{else}$ or $\textit{else / if}$ branch of a conditional statement with the following properties: (a) at least one conditional expression of the previous branches is a disjunction $e_1 \lor e_2 \ldots \lor e_n$ of boolean expressions $e_i$, and (b) at least one sub-expression $e_i$ of that conditional expression is of the form $f == \textit{null}$.

• statement $s$ is invoked in a sub-expression $e_i$ of a conjunctive ($e_1 \land e_2 \ldots \land e_n$) or disjunctive ($e_1 \lor e_2 \ldots \lor e_n$) conditional expression that has a sub-expression $e_j : j < i$ of the form $f \neq \textit{null}$ or $f == \textit{null}$, respectively.

In the example of Figure 4, the $\text{Customer.setShipmentAddress(String)}$ method is associated with the predicate $\text{nullPointerExceptionOnNull(buyer, getShipmentAddress)}$. The reason is that its invocation in the $\text{updateShipmentInfo()}$ method is unprotected from null dereferences on the $\text{buyer}$ field.

Table 1 summarizes all predicates that are assigned to $\text{optionalField}$ methods during the identification of $\text{Null Object}$ refactoring opportunities. These predicates will be referred to as $\text{Null Object Predicates}$.

Table 1: $\text{Null Object Predicates}$ that characterize the methods of an optional field $f$.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Behaviour of $m_i$ in Null Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{emptyOnNull}(f, m_i)$</td>
<td>Empty implementation</td>
</tr>
<tr>
<td>$\text{returnConstantOnNull}(f, m_i, c)$</td>
<td>Returns constant value $c$</td>
</tr>
<tr>
<td>$\text{nullFieldExceptionOnNull}(f, m_i)$</td>
<td>Throws $\text{NullFieldException}$</td>
</tr>
<tr>
<td>$\text{nullPointerExceptionOnNull}(f, m_i)$</td>
<td>Throws $\text{NullPointerException}$</td>
</tr>
</tbody>
</table>

4.1.3. Refactoring identification algorithm

This section specifies the algorithm for identification of refactoring opportunities to $\text{Null Object}$. The algorithm processes sequentially all classes of the system's code base. The processing of each individual class $ctx$, that has the role of the $\text{Context}$, is described in Algorithm 1. The algorithm receives
as input a system class and produces as output a set of refactoring opportunities
\( R = \bigcup \{(f, G, P_{\text{empty}}, P_{\text{const}}, P_{\text{nfe}}, P_{\text{npe}})\} \), where: (i) \( f \) is an optional field,
(ii) \( G \) is the set of GFI-Conditionals on \( f \) that can be eliminated, (iii) \( P_{\text{empty}}, P_{\text{const}}, P_{\text{nfe}}, P_{\text{npe}} \) are sets that correspond to the Null Object Predicates of Table 1 and contain methods of class \( f \) that satisfy them. Especially \( P_{\text{const}} \) contains pairs \((m, c)\), where \( c \) is the default constant value that method \( m \) returns in the Null Object. For brevity reasons, we will, often, refer to the refactoring opportunities, identified for a given Context class, through the set of Optional Fields corresponding to these opportunities.

Algorithm 1 iterates over the fields of class \( \text{ctx} \) and analyzes those that:
(a) satisfy the \( \text{isOptional}(\text{Field}, \text{Class}) \) predicate (line 3) and (b) do not violate refactoring preconditions (line 4). The violation of refactoring preconditions leads to code that either has compilation errors or does not preserve the external behaviour of the system. A detailed specification of the preconditions that apply to an optional field is provided in the next section. The algorithm represents as \( G_f \), the set of GFI-Conditionals that can be eliminated for the current field \( f \). Moreover, sets \( P_{\text{empty}, f}, P_{\text{const}, f}, P_{\text{nfe}, f}, P_{\text{npe}, f} \) correspond to the methods of \( f \) that satisfy the Null Object Predicates. These sets are initialized in line 5 and are filled with elements during the processing of each individual method of the \( \text{ctx} \) class (lines 6-43).

The algorithm processes each method \( m \) of class \( \text{ctx} \) and analyzes the method invocations and conditional statements that are declared in the method body. Initially, it checks all method invocations declared in the body of \( m \) for unguarded invocations on field \( f \) (lines 8-12). The invoked methods are appended as elements to the set \( P_{\text{npe}, f} \) that contains violations of the \( \text{nullPointerExceptionOnNull}(\text{Field}, \text{Method}) \) predicate.

In a next stage, the algorithm iterates over the conditional statements, declared in the body of \( m \), in search of Variant 1-4 GFI-Conditionals (lines 13-42). In case that the currently processed conditional statement, \( \text{cond} \), conforms to the structure of any variant, it is temporarily marked by setting the \( \text{isGFI} \) variable to a \( \text{true} \) value. Moreover, the algorithm evaluates the method invocations that are declared in the FieldInvocationFragment of \( \text{cond} \) and associates them with appropriate Null Object Predicates (lines 16-37). If \( \text{cond} \) is a Variant 1 GFI-Conditional, the algorithm analyzes all the statements of the FieldInvocationFragment\(^1\), that are either method in-

\(^1\)Refer to the definition of the \( \text{isFieldInvocationFragment}(\text{Set[Statement]}, \text{Field}) \) predi-
Algorithm 1: Algorithm for identification of refactoring candidates to NULL OBJECT in a given class.

\[
\text{Algorithm 1: Algorithm for identification of refactoring candidates} \\
\text{to NULL OBJECT in a given class.}
\]

\[
\text{input : Class } \text{ctx} \\
\text{output: } R = \{\text{field, } G, \text{P}_{\text{empty}}, \text{P}_{\text{const}}, \text{P}_{\text{fct}}, \text{P}_{\text{npe}}\}
\]

1. \(R \leftarrow \emptyset\)

2. \begin{align*}
&\text{foreach Field } f \in \text{ctx.fields do} \\
&\quad \text{if isOptional}(f, \text{ctx}) = \text{false} \text{ then continue}
\end{align*}

3. \begin{align*}
&\text{if violatesOptionalFieldPreconditions}(f) = \text{true} \text{ then continue} \\
&\text{/* Set of GFI-Conditionals and sets of methods that satisfy Null Object}\n\end{align*}

4. \begin{align*}
&\text{Predicates for field } f. \\
&G_f \leftarrow \emptyset; P_{\text{empty}, f} \leftarrow \emptyset; P_{\text{const}, f} \leftarrow \emptyset; P_{\text{fct}, f} \leftarrow \emptyset; \text{P}_{\text{npe}, f} \leftarrow \emptyset;
\end{align*}

5. \text{foreach Method } m \in \text{ctx.methods do}

6. \begin{align*}
&\text{body } \leftarrow m.\text{methodBody}; \\
&\text{/* Assign the nullPointerExceptionOnNull predicate.} \\
&\text{/* Identify and process GFI-Conditionals} \\
&\text{/* Temporary sets for } cond \text{ processing.} \\
&\text{isGFI } \leftarrow \text{false}; \\
&\text{P}_{\text{empty}, t} \leftarrow \emptyset; P_{\text{const}, t} \leftarrow \emptyset; P_{\text{fct}, t} \leftarrow \emptyset;
\end{align*}

7. \begin{align*}
&\text{if isGFIv1Conditional}(f, cond) = \text{true} \text{ then} \\
&\text{isGFI } \leftarrow \text{true}; \text{cond.thenBlock.statements } \leftarrow \text{cond.thenBlock.statements}; \\
&\text{foreach ExpressionStatement stmt } \in \text{cond.thenBlock.statements do} \\
&\text{expr } \leftarrow \text{stmt.expression}; \\
&\text{if instanceof}(\text{expr}, \text{MethodInvocation}) = \text{true} \text{ then} \\
&\text{P}_{\text{empty}, t} \leftarrow \text{P}_{\text{empty}, t} \cup \{\text{expr.declaringMethod}\}
\end{align*}

8. \begin{align*}
&\text{else if instanceof}(\text{expr}, \text{Assignment}) = \text{true} \text{ then} \\
&\text{inv } \leftarrow \text{expr.rightHandSide}; \text{decl } \leftarrow \text{inv.declaringMethod}; \\
&\text{if isLiteralType(decl.returnType) } = \text{true} \text{ then} \\
&\text{C } \leftarrow \text{getFieldDefinition(expr.leftHandSide, expr.rightHandSide)}; \\
&\text{if } |C| = 1 \text{ then } P_{\text{const}, t} \leftarrow P_{\text{const}, t} \cup \{(\text{decl, } C_1)\}
\end{align*}

9. \begin{align*}
&\text{end}
\end{align*}

10. \begin{align*}
&\text{else if isGFIv2Conditional}(f, cond) = \text{true} \text{ then} \\
&\text{isGFI } \leftarrow \text{true}; s_1 \leftarrow \text{cond.thenBlock.methodInvocations}[1]; \\
&\text{P}_{\text{fct}, t} \leftarrow \text{P}_{\text{fct}, t} \cup \{s_1.\text{declaringMethod}\};
\end{align*}

11. \begin{align*}
&\text{else if isGFIv3Conditional}(f, cond) = \text{true} \text{ then} \text{isGFI } \leftarrow \text{true}; \\
&\text{else if isGFIv4Conditional}(f, cond) = \text{true} \text{ then} \text{isGFI } \leftarrow \text{true}; \\
&\text{end}
\end{align*}

12. \begin{align*}
&\text{if isGFI } = \text{true} \text{ and violatesGFIPreconditions}(\text{cond} ) = \text{false} \text{ then} \\
&\text{G_f } \leftarrow G_f \cup \{\text{cond}\}; P_{\text{empty}, f} \leftarrow \text{P}_{\text{empty}, f} \cup \text{empty}; \\
&\text{P}_{\text{const}, f} \leftarrow \text{P}_{\text{const}, f} \cup \text{P}_{\text{const}, t}; P_{\text{fct}, f} \leftarrow \text{P}_{\text{fct}, f} \cup \text{P}_{\text{fct}, t};
\end{align*}

13. \begin{align*}
&\text{end}
\end{align*}

14. \begin{align*}
&\text{if } G_f \neq \emptyset \text{ then } R \leftarrow R \cup \{f, G_f, P_{\text{empty}, f}, P_{\text{const}, f}, P_{\text{fct}, f}, P_{\text{npe}, f}\}
\end{align*}

15. \text{return } R;
vocation or assignment statements (lines 18-29). Method invocations with a
void return type are associated with the emptyOnNull(Field, Method) predicate
(lines 20-22). As concerning assignments, the algorithm processes those
with a right hand side corresponding to the invocation of a method with
primitive or String return type (lines 22-28). Specifically, it evaluates the get-
LastDefinition(VariableAccess, Statement) function on the assigned variable
to retrieve its potential values in the current statement (see Section 4.1.2).
If those values correspond to a single constant the invoked method is associ-
ated with the returnConstantOnNull(Field, Method, Literal) predicate (lines
25-26). Processing of Variant 2, 4 GFI-Conditionals involves only their first
method invocation that is associated with the predicate nullFieldException-
OnNull(Field, Method).

Each method invocation that satisfies a Null Object Predicate is added
to one of the “temporary” sets $P_{empty,t}$, $P_{const,t}$, $P_{nfe,t}$ (lines 21, 26, 32, 36).
The algorithm makes a temporary assignment, since each GFI-Conditional
is checked for violation of refactoring preconditions prior to being suggested
as candidate for elimination. Refactoring preconditions that apply to a GFI-
Conditional are elaborated in Section 4.1.4. The processing of a GFI-Con-
ditional that satisfies refactoring preconditions involves (a) its addition to
the $G_f$ set and (b) the expansion of $P_{empty,f}$, $P_{const,f}$, $P_{nfe,f}$ sets with the
contents of $P_{empty,t}$, $P_{const,t}$, $P_{nfe,t}$ respectively (lines 39-40).

After analyzing all methods of class $ctx$ with respect to field $f$, the algo-
thesis suggests $f$ as a refactoring opportunity if the set $G_f$ is not empty, i.e.,
there exists at least one GFI-Conditional on $f$ that can be eliminated. In
this case the tuple $(f, G_f, P_{empty,f}, P_{const,f}, P_{nfe,f}, P_{npe,f})$ is appended to the
set $R$ of refactoring candidates for class $ctx$.

4.1.4. Refactoring preconditions

The refactoring identification algorithm evaluates a set of preconditions
prior to suggesting an optional field or a GFI-Conditional for automated
refactoring to NULL OBJECT. The role of refactoring preconditions is to
prevent the application of refactorings that would lead to erroneous refac-
tored code, in the sense that it would either have compilation errors or it
would not preserve the external behaviour of the system. In this section,
we specify in detail an extensive set of preconditions that are relevant to

cate in Appendix A.
the source code transformation for the introduction of the **NULL OBJECT** pattern. Preconditions are grouped in two categories, on the basis of their scope of application: (a) Optional field preconditions, (b) GFI-Conditional preconditions.

**Optional Field Preconditions.** These preconditions are relevant to properties of the optional field’s type (**Component** class), as well as to the declaration of the optional field in the **Context** class. Optional field preconditions require that:

1. **Component** must belong to the analyzed project’s code base, since refactoring to **NULL OBJECT** introduces modifications to its source code,

2. **Component** must not be an interface, since the code transformation would change its contract through the addition of extra methods (see Section 4.2). Any change to the contract of that interface would introduce compilation errors to all other classes that implement it.

3. **Component** must not inherit from another class, as it is required to become a subclass of **AbstractComponent**. This precondition is mandatory for Java, as well as for other programming languages that do not support multiple inheritance.

4. **Component** fields must not be directly accessed in the **Context** class through the **optionalField** reference. In the opposite case, it would be required to “pull up” these fields to the **AbstractComponent** class, so that they can be accessed through an **AbstractComponent** reference in the refactored code. Such a refactoring would introduce extensive modifications to the **Component** class and would severely impacts its encapsulation.

5. the optional field declaration in **Context** class must not be visible to subclasses of **Context**. This precondition averts cascading modifications to **Context** subclasses due to the optional field’s type change from **Component** to **AbstractComponent**.

6. in case that **Component** is declared as an inner class of **Context**, the latter must not invoke private **Component** methods through the **optionalField** reference. Such invocations would require to break the encapsulation of **Component**, through moving these methods to **AbstractComponent** and, possibly, changing their visibility to non-private.
GFI-Conditional Preconditions. These preconditions ensure that a GFI-Conditional can be safely eliminated during refactoring to NULL OBJECT. Specifically, any of the GFI-Conditional variants that were specified in Section 4.1.2 must satisfy the following preconditions:

1. a NullFieldException that is thrown in a GFI-Conditional must belong to the analyzed project’s code base,

2. the constructor arguments of the NullFieldException, invoked in the GFI-Conditional, must be known at compilation time. Moreover, they must have a primitive or string type. The restriction on argument types reduces the complexity of evaluating the precondition.

3. methods invoked in the FieldInvocationFragment part of a GFI-Conditional must not be static. In the opposite case, static methods invoked through the optionalField reference, would need to be moved to AbstractComponent class.

4. methods invoked in the FieldInvocationFragment of a GFI-Conditional must not have conflicting Null Object Predicates associated with them. In other words, each method must be characterized by at most one predicate.

5. all methods invoked in the FieldInvocationFragment of a Variant 1 GFI-Conditional must be associated with one of the predicates emptyOnNull(Field, Method) or returnConstantOnNull(Field, Method, Literal).

4.2. Application of the refactoring

This section specifies the procedure for transforming a refactoring candidate (Context, optionalField) to the NULL OBJECT design pattern. The source code transformation involves the creation of extra classes and the modification of existing ones. It can be decomposed into a series of more elementary transformations, each one focusing on an individual class participant of the design pattern:

1. Creation of the AbstractComponent class.

2. Refactoring of Component to an AbstractComponent subclass.
3. Creation of the `NullComponent` class.


The description of each elementary transformation is provided below:

**Creation of AbstractComponent class.** The `AbstractComponent` class is introduced as an abstract super-class of `Component`, it has the same class modifiers (except for the `final` modifier) and implements all interfaces that `Component` implements. Furthermore, all non-private, non-static methods of `Component` class are declared in the `AbstractComponent` class as `abstract` methods with the same signature. As concerning `Component` methods $m_i$ that satisfy the `nullFieldExceptionOnNull(optionalField, m_i)` predicate, they are declared in `AbstractComponent` as throwing a `NullFieldException`, since their implementation in `NullComponent` throws the respective exception. Recall that `Component` methods that are invoked in GFI-Conditional statements have already been qualified with certain predicates by the refactoring identification algorithm (Section 4.1.2).

Refactoring to NULL OBJECT changes the `optionalField` type from `Component` to `AbstractComponent` and it, henceforth, disallows its assignment to a null value in the bounds of `Context` class. Thus, the interface of `AbstractComponent` needs to be expanded so as to (a) handle the absence of a null `optionalField` reference within `Context`, e.g. express null checking comparisons against the `optionalField` in GFI-Conditional statements that cannot be eliminated, and (b) make the use of the Null Object transparent to `Context` clients. The methods that are introduced to `AbstractComponent` and its subclasses in order to handle the absence of a null `optionalField` reference will be referred to as Null Object Utility Methods. The signatures of these methods are available in Listing 2:

<table>
<thead>
<tr>
<th>Method Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public abstract boolean isNull();</code></td>
<td>Enables the replacement of <code>optionalField</code> null-checking comparisons that cannot not be eliminated during refactoring. The method is implemented by <code>AbstractComponent</code> subclasses and returns <code>true</code></td>
</tr>
<tr>
<td><code>public abstract Component getReference();</code></td>
<td></td>
</tr>
<tr>
<td><code>public abstract void assertNotNull() throws NullFieldException;</code></td>
<td></td>
</tr>
</tbody>
</table>
when declared in a *Null Object*. As concerning the `getReference()` method, its purpose is to prevent the use of the *Null Object* instance, referenced by `optionalField`, outside the bounds of the *Context* class. The objective is to isolate the use of the *Null Object* within *Context* and do not affect the external behaviour provided to *Context* clients. The `getReference()` method returns null when implemented in a *Null Object* class. In any other case it returns a reference to the current object (e.g. through the `this` keyword in Java).

Unlike `isNull()`, the `assertNotNull()` method is not used in conditional expressions. Its purpose is to replace *Variant 3* GFI-Conditionals that are identified in *Context* class. Recall that this type of GFI-Conditional has a single branch and throws an application specific exception (*NullField-Exception*) in case that the `optionalField` is null. The method’s implementation in a *Null Object* class comprises a single statement that throws `NullFieldException`. On the other hand, the *Component* class provides an empty implementation for `assertNotNull()`. Note that in case that *Context* includes *Variant 3* GFI-Conditionals for `optionalField` that throw two or more different types of application specific exceptions, a different version of `assertNotNull()` is needed for each exception type. Each version has its own name and throws the respective exception type. For the sake of simplicity we will assume a single `assertNotNull()` for *AbstractComponent* for the rest of this section.

Figure 5 illustrates the *AbstractCustomer* class that is introduced during refactoring the shopping cart case study to *Null Object*. The class includes abstract declarations of *Customer* methods, as well as of *Null Object Utility Methods*. Note that the method `setCreditCardNumber(String)` throws `CustomerNotFoundException` (it has the role of the `NullFieldException`), since it has been marked with the `nullFieldExceptionOnNull(optionalField, m)` predicate during the refactoring identification phase.

**Creation of NullComponent class.** The *NullComponent* class is, also, introduced during refactoring and represents the implementation of the *Null Object*. The class is declared as a subclass of *AbstractComponent* and provides a default implementation for *Null Object Utility Methods* that is available in Listing 3.


```java
public boolean isNull() {
    return true;
}
```
Figure 5: Creation of AbstractComponent and its subclasses in the case study.
public Component getReference (){
  return null;
}

public void assertNotNull () throws NullFieldException {
  throw new NullFieldException ( );
}

As concerning the rest of the methods declared in AbstractComponent, the NullComponent class provides a default implementation that depends on the predicates that were associated with them during the identification of refactoring opportunities to NULL OBJECT. This default implementation comprises either (a) an empty method body, (b) return of a constant value (String or primitive type), (c) throwing a NullPointerException, or (d) throwing a NullFieldException. The mapping of Null Object Predicates to these implementations has been presented in Table 1. In case that no predicate is associated with an AbstractComponent method, its default implementation is to throw an UnsupportedOperationException.

Figure 5 presents the NullCustomer implementation for the shopping cart case study. Method getDiscount() returns a constant value, as it satisfies the predicate returnConstantOnNull(buyer, getDiscount, 0). Methods setCreditCardNumber(String) and setShipmentAddress(String), marked with predicates nullFieldExceptionOnNull(buyer, setCreditCardNumber) and nullPointerExceptionOnNull(buyer, setShipmentAddress) respectively, throw relevant exceptions. Finally, the method setCreditCardIssueDate(Date) throws UnsupportedOperationException, since it is not associated with any predicate, and, thus, is not expected to be invoked on a NullCustomer instance. Raising an exception, in this case, enables behaviour preservation, since an empty “do nothing” functionality is not necessarily behaviour preserving. On the other hand, the raised exception forces the programmer to decide on an appropriate default behaviour for the setCreditCardIssueDate(Date) method.

Refactoring of Component class. Component class represents the optional-field type before refactoring. Refactoring to NULL OBJECT turns Component to an AbstractComponent subclass and removes all interface implementation declarations from the class. The implemented interfaces of Component have already been declared in its parent class, AbstractComponent. Finally,
a default implementation is provided for the *Null Object Utility Methods* that is available in Listing 4. The implementation of these methods for the shopping cart case study is presented in Figure 5.


```java
class Component {
    public boolean isNull() {
        return false;
    }

    public Component getReference() {
        return this;
    }

    public void assertNotNull() throws NullFieldException {
    }
}
```

Refactoring of Context and elimination of GFI Conditionals. The refactoring of *Context* class focuses, primarily, on the elimination of all null-checking expressions on `optionalField`. Null-checking expressions that appear as conditions in GFI-Conditionals are removed along with the respective conditional statement. The rest of them are replaced with an `isNull()` invocation on `optionalField`, or its negation.

The proper introduction of the *Null Object* design pattern requires that `optionalField` is never defined to a *null* value inside *Context* class. Thus, the code transformation modifies the `optionalField` declaration by changing its type to `AbstractComponent` and initializing it with a `NullComponent` instance. However, `optionalField` may be defined to a null value in any assignment statement that is declared in *Context* class. The proposed refactoring safeguards against such contingencies with the help of a utility method, `assignToOptionalField(AbstractComponent)`, that filters the value expressions assigned to `optionalField`. Listing 5 presents the implementation of `assignToOptionalField(AbstractComponent)`. The method returns a new `NullComponent` instance, if the supplied parameter is null, or the parameter value in any other case.

Listing 5: Implementation of `assignToOptionalField()`.

```java
private AbstractComponent assignToOptionalField(AbstractComponent value) {
    if (value == null) {
        return new NullComponent();
    } else { return value; }
}
```
A basic requirement for the proposed refactoring algorithm is to make the use of the Null Object design pattern transparent to the clients and dependencies of the Context class. The Null Object pattern allows for representing the absence of an optionalField value with a NullComponent instance (instead of a null value). However, this situation must be signalled to clients and dependencies of Context with an ordinary null value. This is achieved through converting the NullComponent instance to a null reference in cases that the optionalField value is provided as a return value or parameter to other classes. Such conversion is enabled by the getReference() utility method of optionalField that returns null when invoked from a NullComponent instance or the current object’s reference (through the this keyword in Java) when invoked from Component.

The refactoring procedure for Context class can be summarized to the following transformation steps:

1. Change the type of the optionalField declaration from Component to AbstractComponent.
   - Initialize the optionalField with a new NullComponent instance.

2. Create an appropriate assignToOptionalField(AbstractComponent) method in the Context class.

3. Eliminate null-checking expressions on optionalField as following:
   - replace Variant 1,2 GFI-Conditionals with their if branch body,
   - replace Variant 3 GFI-Conditionals with an assertNotNull() invocation on optionalField,
   - replace Variant 4 GFI-Conditionals with their else branch body,
   - replace remaining optionalField equality comparisons against null with optionalField.isNull(),
   - replace remaining optionalField inequality comparisons against null with !optionalField.isNull().

4. Isolate the use of the NullComponent within the bounds of the Context class by replacing the optionalField with optionalField.getReference() invocations in the following cases:
• return statements with optionalField as return value,
• occurrences of optionalField as actual parameter in non Context method invocations.

5. Ensure that optionalField is never assigned a null value:

• replace any expression expr that appears as right-hand operand in optionalField assignment statements with an invocation of assignToOptionalField(expr).

Figure 6 illustrates the application of the refactoring steps in the shopping cart case study. The left and right frames present the ShoppingCart code before and after refactoring, respectively. The code fragments that are affected by the code transformation are outlined and associated with an appropriate refactoring step. Specifically, step 1 changes the type of buyer field to AbstractCustomer and initializes it with a NullCustomer instance. Step 2 introduces the assignToOptionalField(AbstractCustomer) method to the refactored ShoppingCart and step 3 eliminates three GFI-Conditional statements. Step 4 modifies the buyer assignments in the class constructor and the respective “setter” method by filtering the assigned value through the assignToOptionalField() method. Finally, step 5 replaces all occurrences of optionalField as return value or actual parameter to other classes’ method invocations with an invocation of optionalField.getReference().

5. Experimental Evaluation

5.1. Implementation details

The proposed method for refactoring to NULL OBJECT has been implemented as part of the JDeodorant plug-in for the Eclipse IDE [10]. The syntactic analysis of Java source files, performed during execution of the refactoring identification algorithm, is realized through (a) AST parsing capabilities provided by the Eclipse Java Development Tools (JDT) Core infrastructure and (b) utility classes included in the JDeodorant project. The identification of refactoring candidates requires, in certain cases, control and data flow analysis that is applied in methods containing GFI-Conditionals. Such analysis is based on Control Flow Graph and Program Dependence Graph representations of these methods that are constructed through relevant infrastructure of JDeodorant [27]. These graphs are generated and processed
Figure 6: Refactoring of `Context` class during refactoring.
in the following cases: (a) evaluation of the \textit{defines}(Statement, Field) and \textit{getLastDefinition}(VariableAccess, Statement) predicates during analysis of GFI-Conditional statements and, (b) evaluation of the second \textit{GFI-Conditional Precondition} (see Section 4.1.4). Finally, the required transformations for applying the refactoring have been implemented with functionality provided by JDT and the Eclipse Language Toolkit (LTK).

Figure 7 presents a screen-shot of the tool after its execution on the Apache Ant project. The table with title “Null Checks”, positioned on the lower part of the figure, includes the identified refactoring opportunities to \textit{Null Object}. Each table row corresponds to a GFI-Conditional statement, that protects method invocations on an optional field satisfying all \textit{Optional Field Preconditions}. The respective \texttt{optionalField} and \texttt{Context} class are available in columns “Field Declaration” and “Context Class”. GFI-Conditionals that, also, satisfy \textit{GFI-Conditional Preconditions} are characterized by a \texttt{true} value in column “Conditional can be removed”. These statements can be automatically eliminated through refactoring to \textit{Null Object}. On the other hand, GFI-Conditionals characterized by a \texttt{false} value cannot be automatically refactored due to violation of preconditions. The introduction of the \textit{Null Object} in these cases depends on the developer’s judgment and may be applied manually.

The actual application of the refactoring on a selected row can be activated by an appropriate button in the top right part of the table. Refactoring a specific GFI-Conditional involves, also, the elimination of the rest of the GFI-Conditionals that correspond to the same \texttt{optionalField} and satisfy all refactoring preconditions.

5.2. Evaluation Results

The proposed methodology has been experimentally evaluated with respect to its soundness, effectiveness and practicality. Soundness has been empirically evaluated through application of our method to refactoring candidates identified in a set of benchmark projects and execution of the projects’ test suites. The effectiveness and practicality have been evaluated in terms of (a) the number of identified refactoring candidates on a set of benchmark projects and (b) the impact of the applied refactorings to the cyclomatic complexity of the methods with eliminated null-checks. The projects used in our evaluation were selected on the basis of the following requirements:

- the projects’ source code must be publicly available in order to be used
as input in our refactoring tool and to enable the reproducibility and confirmation of evaluation results by third-parties,

- projects must be implemented in the Java programming language, since the proposed methodology has been implemented on an infrastructure for Java code analysis,

- projects must vary in terms of size and complexity in order to study their impact on the applicability of this approach.

Table 2 provides information on the size and structural properties of the selected software projects, as well as on the code coverage of their respective test suites. Specifically, each project is characterized by four metrics that correspond to respective table columns: (a) total lines of code without blank lines and comment lines (SLOC), (b) total number of classes, (c) total number of class methods and (d) ratio of covered instructions by the execution of the
project’s test suite over the total source code instructions of the project\textsuperscript{2}.

Table 2: Size characteristics of the examined software projects.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>SLOC</th>
<th>Classes</th>
<th>Methods</th>
<th>Code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml commons ext. 1.4.1</td>
<td>12,856</td>
<td>153</td>
<td>3,168</td>
<td>-</td>
</tr>
<tr>
<td>violet 1.0</td>
<td>19,965</td>
<td>370</td>
<td>1,741</td>
<td>-</td>
</tr>
<tr>
<td>nutch 1.1</td>
<td>26,500</td>
<td>356</td>
<td>1,741</td>
<td>30%</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>78,457</td>
<td>792</td>
<td>5,438</td>
<td>44%</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>90,125</td>
<td>978</td>
<td>7,249</td>
<td>61%</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>91,979</td>
<td>1,063</td>
<td>7,932</td>
<td>58%</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>103,148</td>
<td>1,167</td>
<td>9,430</td>
<td>46%</td>
</tr>
<tr>
<td>jade 4.1</td>
<td>109,036</td>
<td>1,569</td>
<td>8,582</td>
<td>-</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>112,511</td>
<td>791</td>
<td>8,458</td>
<td>24%</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>143,104</td>
<td>936</td>
<td>10,644</td>
<td>54%</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>171,492</td>
<td>1,158</td>
<td>9,430</td>
<td>49%</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>179,122</td>
<td>2,288</td>
<td>14,021</td>
<td>-</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>179,924</td>
<td>2,157</td>
<td>14,965</td>
<td>67%</td>
</tr>
</tbody>
</table>

5.2.1. Soundness of the approach

The soundness of the proposed method has been empirically evaluated on the basis of (a) the syntactic correctness and (b) the preservation of the external behaviour of refactored projects. At first, we have applied to the benchmark projects all the refactorings that were suggested by our refactoring identification algorithm. Both the identification and the application of the respective source code transformation to refactoring candidates were performed through the automation provided by our Eclipse plugin implementation. No errors resulted from the compilation of refactored projects which supports the syntactic validity of the applied transformations.

In a next step we have executed the test suite of each project on the refactored version of its source code. We have excluded from this part of the evaluation: (a) jade 4.1, as it does not have a publicly available test suite, (b) batik 1.7, due to many erroneous test executions on the original code (problems related with missing required graphics files for evaluation of test results) and (c) projects with no refactoring candidates. The test suite execution results of the refactored projects were identical to the respective results of their original version. Thus, the applied refactorings preserve the projects’ external behaviour, at least to the extend that can be verified by the execution of their test suites.

\textsuperscript{1}Project size results were estimated with CodePro Analytix v.7.1.0 plugin for Eclipse.

\textsuperscript{2}Estimated with the use of the JaCoCo v.0.7.2 code coverage library for Java.
5.2.2. Effectiveness and practicality

Since the Optional Fields of a software project form the search domain for refactoring opportunities to NULL OBJECT, we provide an analysis of their presence in benchmark projects in Table 3. The total number of optional fields for each project is available in column “Optional Fields”. The rest of the columns of Table 3 present the part of them that are rejected by Optional Field Preconditions. Specifically, columns $P_1$ to $P_6$ correspond to the six Optional Field Preconditions of Section 4.1.4 and include for each project: (a) the absolute number of rejections that result when applying the respective precondition on all optional fields, (b) the ratio of rejected optional fields over total (inside parentheses). Preconditions 1 and 3 have the higher selectivity and they reject, respectively, 54.7% and 63.4%, on average, of optional fields. The values of $P_1$ and $P_3$ increase with the number of optional fields that are external project dependencies or are parts of a class hierarchy. Precondition 2 has the third higher selectivity (31.4% on average), while the rest of them reject a smaller share of optional fields (< 4% on average). A high value of $P_2$, for a specific project, denotes an accordingly high abstractness among optional fields, as rejected fields have an interface type declaration.

Table 3: Optional fields and precondition violations for each project.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>Optional Fields</th>
<th>$P_1$ (%)</th>
<th>$P_2$ (%)</th>
<th>$P_3$ (%)</th>
<th>$P_4$ (%)</th>
<th>$P_5$ (%)</th>
<th>$P_6$ (%)</th>
<th>None (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml commons ext. 1.4.1</td>
<td>54</td>
<td>20 (37.0)</td>
<td>30 (55.6)</td>
<td>36 (66.7)</td>
<td>1 (1.9)</td>
<td>2 (3.7)</td>
<td>0 (0.0)</td>
<td>5 (9.3)</td>
</tr>
<tr>
<td>violet 1.0</td>
<td>217</td>
<td>132 (60.8)</td>
<td>33 (15.2)</td>
<td>119 (54.8)</td>
<td>1 (0.5)</td>
<td>1 (0.5)</td>
<td>0 (0.0)</td>
<td>5 (13.4)</td>
</tr>
<tr>
<td>nutch 1.1</td>
<td>131</td>
<td>81 (61.8)</td>
<td>33 (25.2)</td>
<td>67 (51.1)</td>
<td>4 (3.1)</td>
<td>2 (1.5)</td>
<td>0 (0.0)</td>
<td>36 (27.5)</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>1,065</td>
<td>1,005 (94.4)</td>
<td>64 (6.0)</td>
<td>932 (87.5)</td>
<td>0 (0.0)</td>
<td>5 (0.5)</td>
<td>1 (0.1)</td>
<td>40 (3.8)</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>276</td>
<td>146 (52.9)</td>
<td>130 (47.1)</td>
<td>179 (64.9)</td>
<td>0 (0.0)</td>
<td>13 (4.7)</td>
<td>1 (0.4)</td>
<td>55 (19.9)</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>388</td>
<td>253 (65.2)</td>
<td>142 (36.6)</td>
<td>277 (71.4)</td>
<td>0 (0.0)</td>
<td>6 (1.5)</td>
<td>1 (0.3)</td>
<td>43 (11.1)</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>675</td>
<td>418 (61.9)</td>
<td>66 (9.8)</td>
<td>302 (44.7)</td>
<td>1 (0.1)</td>
<td>21 (3.1)</td>
<td>5 (0.7)</td>
<td>73 (10.8)</td>
</tr>
<tr>
<td>jade 4.1</td>
<td>775</td>
<td>313 (40.4)</td>
<td>125 (16.1)</td>
<td>386 (49.8)</td>
<td>9 (1.2)</td>
<td>17 (2.2)</td>
<td>0 (0.0)</td>
<td>262 (34.8)</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>731</td>
<td>251 (34.3)</td>
<td>324 (44.3)</td>
<td>481 (65.8)</td>
<td>20 (2.7)</td>
<td>49 (6.7)</td>
<td>0 (0.0)</td>
<td>170 (23.3)</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>706</td>
<td>486 (68.8)</td>
<td>381 (54.0)</td>
<td>492 (69.7)</td>
<td>0 (0.0)</td>
<td>3 (0.4)</td>
<td>0 (0.0)</td>
<td>37 (5.2)</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>574</td>
<td>305 (53.1)</td>
<td>187 (32.6)</td>
<td>389 (67.8)</td>
<td>0 (1.0)</td>
<td>37 (6.4)</td>
<td>0 (0.0)</td>
<td>115 (20.0)</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>1,021</td>
<td>428 (41.9)</td>
<td>322 (31.5)</td>
<td>690 (65.6)</td>
<td>14 (1.4)</td>
<td>84 (8.2)</td>
<td>0 (0.0)</td>
<td>171 (16.7)</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>1,193</td>
<td>455 (38.1)</td>
<td>410 (34.4)</td>
<td>760 (62.9)</td>
<td>20 (1.7)</td>
<td>36 (3.1)</td>
<td>0 (0.0)</td>
<td>256 (21.5)</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>-</td>
<td>54.7%</td>
<td>31.4%</td>
<td>63.4%</td>
<td>1.6%</td>
<td>3.4%</td>
<td>0.1%</td>
<td>16.6%</td>
</tr>
</tbody>
</table>

As concerning the last column of Table 3, it includes the number and the respective ratio of optional fields that satisfy all six preconditions. These fields are the source of refactoring opportunities to NULL OBJECT and represent the 16.6% of optional fields, averaged on all projects. Let $F_{opt}$ be the set of optional fields for a specific project and $F_{opt}^a \subseteq F_{opt}$ its subset that
satisfies all Optional Field Preconditions. The evaluation of the effectiveness and practicality of the proposed method will be based on the following sets of optional fields:

- **Automatically refactorable fields** $F^a_{opt}$. Subset of $F_{opt}$, i.e., $F^a_{opt} \subseteq F_{opt}$, with optional fields that have at least one GFI-Conditional that can be eliminated, i.e. it satisfies all GFI-Conditional Preconditions. The set $F^a_{opt}$ for a given project includes all optional fields that are candidates for automated refactoring to *Null Object*.

- **Manually refactorable fields** $F^1_{opt}$. Let $F^1_{opt}$ be the subset of $F_{opt}$ with optional fields that their class belongs to the analyzed project’s code base (satisfy precondition 1), but violate at least one of the other Optional Field Preconditions. Manually refactorable fields $F^1_{opt}$ represent a subset of $F^1_{opt}$ where each optional field has, also, at least one GFI-Conditional that can be eliminated, i.e. it satisfies all GFI-Conditional Preconditions. It holds that $F^1_{opt} \subseteq F^1_{opt} \subseteq F_{opt}$, $F^1_{opt} \cap F^a_{opt} = \emptyset$ and, consequently, $F^1_{opt} \cap F^a_{opt} = \emptyset$.

Refactoring a manually refactorable field to *Null Object* cannot be fully automated as it will lead to compilation errors or changes to the system’s external behaviour. However, we take into account manually refactorable fields in the evaluation of our method, as they represent a potential for further application of the pattern. Depending on his/her judgment and the number of GFI-Conditionals that can be potentially eliminated for a given field $f \in F^1_{opt}$, a programmer may apply appropriate code modifications in order to lift precondition violations and enable its automated refactoring.

- **Refactorable fields** $F^*_{opt}$. They represent the union of automatically and manually refactorable fields for a given project, i.e., $F^*_{opt} = F^a_{opt} \cup F^1_{opt}$.

- **Potentially candidate fields** $F^1_{opt}$. Subset of $F_{opt}$ with fields that their class belongs to the analyzed project’s code base (satisfy at least precondition 1) and their respective Context class contains at least one null-checking conditional on them (not necessarily conforming to one of the four GFI-Conditional variants). $F^1_{opt}$ represents a broader set than refactorable fields, i.e., $F^*_{opt} \subseteq F^1_{opt}$, and will be used in our analysis as
a basis for evaluating the effectiveness of the refactoring identification algorithm. Our assumption is that any optional field \( f \notin \hat{F}_{opt} \) is out of the programmer’s scope for manual or automatic refactoring to Null Object, as: (a) its class does not belong to the project’s code base and, thus, the programmer cannot make code changes to it, or (b) the respective Context class does not declare any null-checking conditionals on the field and, therefore, refactoring to Null Object does not lead to any code improvement.

The number of refactoring opportunities that were discovered in the benchmark projects by our refactoring identification algorithm are available in Table 4. Recall that refactoring candidates correspond to automatically refactorable fields \( \hat{F}_{opt} \) and their number for each project is presented in the third column of the table. Each refactoring opportunity, also, comprises the optional field’s Context class and the GFI-Conditionals that can be automatically eliminated after refactoring to Null Object. The number of GFI-Conditionals that correspond to the \( \hat{F}_{opt} \) set for each project are included in Table 5.

Table 4: Automatically and manually refactorable fields against potentially candidate fields.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>Potentially candidate fields</th>
<th>Refactorable Fields</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Automatically</td>
<td>Manually</td>
<td>Total</td>
</tr>
<tr>
<td>xml commons ext. 1.4.1</td>
<td>15</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>violet 1.0</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nutch 1.1</td>
<td>10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>22</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>55</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>42</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>181</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>jade 4.1</td>
<td>95</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>172</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>64</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>124</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>138</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>207</td>
<td>5</td>
<td>38</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The effectiveness of the refactoring identification algorithm is indicated by the “Applicability” column that represents for each project the ratio \( |\hat{F}_{opt}|/|\hat{F}_{opt}| \), i.e., number of refactorable fields over potentially candidate fields. On the basis of Table 4 results, the Applicability of Null Object on
the benchmark projects ranges from 4.55% to 33.33%. These figures reflect the frequency of optional fields and, thus, the potential for applying NULL OBJECT in each project. Applicability is project specific and depends on the role and semantics of individual class fields. Moreover, it is independent of the project size or the number of potentially candidate fields, as we can figure out by the juxtaposition of Table 2 and Table 4 results. As concerning automatically refactorable fields, they are generally outnumbered by manually refactorable fields, since they must satisfy both the properties and preconditions of optional fields that can be quite limiting, as illustrated in Table 3. In any case, our method manages to identify enough refactoring candidates to NULL OBJECT across all benchmark projects.

An analysis of the GFI-Conditionals that can be eliminated, by refactoring the identified refactorable fields to NULL OBJECT, is available in Table 5. The second and third columns focus on the automatically refactorable fields $F_{opt}^a$ of each project and present respectively: (a) the total number of null-checking conditionals on these fields, (b) the number of them (and the respective ratio inside parentheses) that satisfy the GFI-Conditional properties and preconditions, and, thus, can be eliminated after refactoring. The next two columns present respective results that are relevant to the manually refactorable fields $F_{opt}^m$ of each project. Finally, the last column includes the ratio of eliminated GFI-Conditionals over total null-checking conditionals across all refactorable fields $F_{opt}^*$. The results show that, on average, 65.7% of null-checking conditionals on refactorable fields can be eliminated by our method.

A limitation of the proposed method is that it does not handle conditional checking statements of arbitrary complexity, e.g., conditional statements with more than two branches or with condition expressions. Instead, it focuses strictly on statements that conform to the structure of Variant 1-4 GFI-Conditionals. A basic premise is that the elimination of GFI-Conditionals is straightforward, while their structure provides a strong hint for the identification of optional class collaborators. In order to estimate the extent of missing refactoring opportunities, as well as the potential for further extensions of this approach, we have conducted an analysis of the frequency of GFI-Conditionals against total null-checking conditional statements on the same fields. The results of this analysis, averaged on all benchmark projects, show that GFI-Conditionals represent the 65.2% of total null-checking conditionals on refactorable fields. On the other hand, in a broader set of optional fields comprising the union of $F_{opt}^a$ and $F_{opt}^m$, GFI-Conditionals correspond to
Table 5: Eliminated GFI-Conditionals on automatically and manually refactorable fields.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>Null-checking conditionals on fields $F^c_{null}$</th>
<th>Null-checking conditionals on fields $F^c_{opt}$</th>
<th>Null-checking conditionals on fields $F^c_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Eliminated</td>
<td>Total</td>
</tr>
<tr>
<td>xml commons ext. 1.4.1</td>
<td>0</td>
<td>0 ( - )</td>
<td>30</td>
</tr>
<tr>
<td>violet 1.0</td>
<td>0</td>
<td>0 ( - )</td>
<td>2</td>
</tr>
<tr>
<td>nutch 1.1</td>
<td>2</td>
<td>2 (100 %)</td>
<td>0</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>8</td>
<td>6 (75.0%)</td>
<td>0</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>6</td>
<td>4 (66.7%)</td>
<td>16</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>9</td>
<td>7 (77.8%)</td>
<td>14</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>15</td>
<td>10 (66.7%)</td>
<td>13</td>
</tr>
<tr>
<td>jade 4.1</td>
<td>7</td>
<td>4 (53.8%)</td>
<td>12</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>2</td>
<td>1 (50.0%)</td>
<td>116</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>0</td>
<td>0 ( - )</td>
<td>53</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>35</td>
<td>25 (71.4%)</td>
<td>33</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>2</td>
<td>2 (100 %)</td>
<td>30</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>7</td>
<td>5 (71.4%)</td>
<td>65</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>73.3%</td>
<td>-</td>
<td>59.7%</td>
</tr>
</tbody>
</table>

19.3% of null-checking conditionals on these fields.

The evaluation of our method involves, also, a study on how software quality is affected by the application of all identified refactorings to NULL OBJECT. The focus is on assessing the improvement to McCabe’s Cyclomatic Complexity metric [28] due to the applied refactorings. The computation of McCabe’s Cyclomatic Complexity for a given class method is based on the program’s Control Flow Graph (CFG) and its value is calculated by counting the number of alternative control flows within the method. Specifically, whenever a control flow branch appears in the CFG (corresponding to either an if, for, while, do/while, switch or catch statement, the ternary operator ?: or the logical expression operators && and ||), the value of the McCabe’s Cyclomatic Complexity metric is increased by one.

Table 6 presents the measurements of McCabe’s Cyclomatic Complexity for each benchmark project, prior and after refactoring the automatically refactorable fields. Column $C_p$ provides for each project the average metric value, prior to refactoring, for all methods that declare the GFI-Conditionals that were identified for the specific project. Column $C_a$ displays the aforementioned metric after refactoring. The standard deviation corresponding to each average value is provided inside parentheses. Finally, the last column provides the percent improvement of the metric (percent reduction of cyclomatic complexity) after refactoring that ranges from 5% to 66.67%. These
figures highlight the contribution of the proposed method to complexity reduction in several classes of each code base.

Table 6: Impact of refactoring on cyclomatic complexity.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>$C_p$</th>
<th>$C_a$</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml commons ext. 1.4.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>violet 1.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nutch 1.1</td>
<td>3.00 (0.00)</td>
<td>1.00 (0.00)</td>
<td>66.67</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>2.00 (0.00)</td>
<td>1.00 (0.00)</td>
<td>50.00</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>2.75 (0.96)</td>
<td>1.75 (0.96)</td>
<td>36.36</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>3.00 (2.05)</td>
<td>2.00 (2.05)</td>
<td>33.33</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>6.67 (8.31)</td>
<td>5.56 (8.32)</td>
<td>16.67</td>
</tr>
<tr>
<td>jadex 4.1</td>
<td>3.29 (1.50)</td>
<td>2.29 (1.50)</td>
<td>30.43</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>20.00 (0.00)</td>
<td>19.00 (0.00)</td>
<td>5.00</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>4.30 (3.09)</td>
<td>1.80 (1.48)</td>
<td>58.14</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>8.00 (0.00)</td>
<td>6.00 (0.00)</td>
<td>25.00</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>2.00 (0.00)</td>
<td>1.00 (0.00)</td>
<td>50</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>5.50 (1.65)</td>
<td>4.14 (1.49)</td>
<td>37.16</td>
</tr>
</tbody>
</table>

The reduction of the cyclomatic complexity metric contributes to more concise and understandable code in the Context class that has positive impact to all maintenance tasks on this class. Moreover, the structure of the Null Object pattern streamlines, after refactoring, the maintenance of the affected code through:

- reuse of the default behaviour represented by the NullComponent class across different optional fields in the same or different Context classes. For instance, among the refactoring candidates identified in the xalan 2.7 project, we have discovered two remarkable cases of such reuse. In the first case, a Null Object class (NullXPath for fields of XPath type) is reused in 7 fields belonging to 4 different Context classes, while in the second case another Null Object class (NullAVT for fields of AVT type) is reused in 4 different fields declared in the same Context class.

- targeted introduction of changes to the default behaviour executed due to the absence of an optional class collaborator. NULL OBJECT enables the declaration of consistent default behaviour that can be easily modified by directly editing the NullComponent class. In any other case the developer has to inspect the optional field invocations and introduce the required changes to the body of the respective null-checking conditionals.
safer extension of Context class behaviour that requires additional invocations on the optional field. Prior to refactoring, the developer has to be aware of the field optionality and introduce appropriate null-checking code in order to guard its invocations. The presence of the Null Object pattern provides a strong hint on field optionality, while allowing the developer to avoid the null-checking code on field invocations.

Finally, we have evaluated the runtime performance of the identification algorithm implementation by measuring its execution time on the software projects of Table 2. The measurements, collected with a 2.5 GHz quad core processor and 4 GB of RAM hardware configuration, show that the execution time ranges from 30s to 4.5min. Thus, the runtime processing overhead of applying our method is limited and allows its integration to the programmer’s routine code analysis activities.

6. Conclusions and Future Work

We have proposed a novel method for automated refactoring to the Null Object design pattern. Null Object provides a design solution against code duplication and conditional complexity that is introduced by repetitive null-checks on potentially null object references. A source of repetitive null-checks in the code of a given class are optional fields, i.e., fields that are not initialized in all class instantiations and, thus, their usage needs to be guarded by appropriate null-checking conditionals that serve the avoidance of null dereferences.

Our approach focuses on the elimination of null-checking conditionals, associated with optional fields, through refactoring to Null Object. We have provided a detailed specification of the Optional Field concept and identified cases of conditional statements that can be automatically eliminated through refactoring to Null Object. Based on the Optional Field concept, we have designed an algorithm for automated discovery of refactoring opportunities. Moreover, we have specified the source code transformation procedure and an extensive set of refactoring preconditions for safely refactoring an Optional Field and its associated null-checking conditionals to the Null Object design pattern. The proposed method has been implemented as an Eclipse plug-in and has been evaluated on a set of open source Java projects. Several refactoring candidates have been discovered in these projects and their
refactoring lead to improvement of the cyclomatic complexity of the affected classes. The successful execution of the projects’ test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Finally, the runtime processing overhead of the refactoring identification algorithm is limited, as it ranges from 30s to 4.5min across all benchmark projects, and allows the integration of the proposed method to the programmer’s routine code analysis activities.

Our future work will focus on enhancements to the proposed method with emphasis on the refactoring and simplification of complex null-checking conditional statements. Moreover, we will examine appropriate adaptations to the existing Null Object implementations in order to broaden the applicability of our method to optional fields whose class inherits from another class or its source code does not belong to the project’s code base (library class). Finally, we will investigate the potential for effective application of the Null Object design pattern to other cases of optional class collaborators, e.g., optional method parameters.

Appendix A.

The appendix provides the specification of auxiliary predicates that are applied in the formal definition of the Null Object Predicates of Section 4.1.2. The following definitions are also based on the notation introduced in Figure 3.

Definition 1. Predicate isSetter(Method, Field)

\[
\text{isSetter}(\text{Method } m, \text{Field } f) \equiv \\
|\text{m.parameters}| \geq 1 \land m.\text{accessModifier} \neq \text{private} \land \\
(\exists s \in m.\text{methodBody}.\text{assignments} : \\
\text{instanceof}(s.\text{leftHandSide}, \text{VariableAccess}) \land \\
s.\text{leftHandSide}.\text{declaration} = f \land \\
\text{instanceof}(s.\text{rightHandSide}, \text{VariableAccess}) \land \\
s.\text{rightHandSide}.\text{declaration} \in m.\text{parameters})
\]
Definition 2. Predicate isGFIv1Conditional(Field, Statement)

\[
isGFIv1Conditional(\text{Field } f, \text{Statement } s) \equiv \\
\text{instanceof}(s, \text{IfStatement}) \land \\
\text{checksNullInequality}(s.\text{expression}, f) \land \\
s.\text{elseBlock} = \emptyset \land \\
isFieldInvocationFragment(s.\text{thenBlock}, f)
\]

Definition 3. Predicate isGFIv2Conditional(Field, Statement)

\[
isGFIv2Conditional(\text{Field } f, \text{Statement } s) \equiv \\
\text{instanceof}(s, \text{IfStatement}) \land \\
\text{checksNullInequality}(s.\text{expression}, f) \land \\
\text{instanceof}(s.\text{elseBlock}, \text{ThrowStatement}) \land \\
( \exists n \in \mathbb{N}^* : n \leq |s.\text{thenBlock.statements}| \land \\
isFieldInvocationFragment(\cup_{i \leq n} \{s.\text{thenBlock.statements}[i]\}, f))
\]

Definition 4. Predicate isGFIv3Conditional(Field, Statement)

\[
isGFIv3Conditional(\text{Field } f, \text{Statement } s) \equiv \\
\text{instanceof}(s, \text{IfStatement}) \land \\
\text{checksNullEquality}(s.\text{expression}, f) \land \\
\text{instanceof}(s.\text{thenBlock}, \text{ThrowStatement}) \land s.\text{elseBlock} = \emptyset
\]

Definition 5. Predicate isGFIv4Conditional(Field, Statement)

\[
isGFIv4Conditional(\text{Field } f, \text{Statement } s) \equiv \\
\text{instanceof}(s, \text{IfStatement}) \land \\
\text{checksNullEquality}(s.\text{expression}, f) \land \\
\text{instanceof}(s.\text{thenBlock}, \text{ThrowStatement}) \land \\
( \exists n \in \mathbb{N}^* : n \leq |s.\text{elseBlock.statements}| \land \\
isFieldInvocationFragment(\cup_{i \leq n} \{s.\text{elseBlock.statements}[i]\}, f))
\]
Definition 6. Predicate isFieldInvocationFragment(Set[Statement], Field)

\[\text{isFieldInvocationFragment}(\text{Set[Statement]} \ S, \text{Field } f) \equiv \]
\[
\forall i \in S : \text{instanceof}(i, \text{ExpressionStatement}) \land
\quad (\text{instanceof}(i.\text{expression}, \text{MethodInvocation}) \land
\quad \text{isFieldInvocation}(i.\text{expression}.\text{invokeExpression}, f)) \lor
\quad (\text{instanceof}(i.\text{expression}, \text{Assignment}) \land
\quad \text{isFieldInvocation}(i.\text{expression}.\text{rightHandSide}.\text{invokeExpression}, f))
\]

Definition 7. Predicate isFieldInvocation(Expression, Field)

\[\text{isFieldInvocation}(\text{Expression } e, \text{Field } f) \equiv \]
\[
\text{instanceof}(e, \text{MethodInvocation}) \land
\quad \text{instanceof}(e.\text{invokeExpression}, \text{VariableAccess}) \land
\quad e.\text{invokeExpression}.\text{declaration} = f
\]

Definition 8. Predicate invokesFieldMethod(Expression, Field, Method)

\[\text{invokesFieldMethod}(\text{Expression } e, \text{Field } f, \text{Method } m) \equiv \]
\[
\text{isFieldInvocation}(e, f) \land e.\text{declaringMethod} = m
\]

Definition 9. Predicate isVoidType(Type)

\[\text{isVoidType}(\text{Type } \text{type}) \equiv \]
\[
\text{instanceof}(\text{type}, \text{PrimitiveType}) \land \text{type.code} = \text{void}
\]

Definition 10. Predicate isLiteralType(Type)

\[\text{isLiteralType}(\text{Type } \text{type}) \equiv \]
\[
(\text{instanceof}(\text{type}, \text{PrimitiveType}) \land \text{type.code} \neq \text{void}) \lor \text{instanceof}(\text{type}, \text{String})
\]
Definition 11. Predicate $\text{checksNullEquality}(Expression, Field)$

$$\text{checksNullEquality}(Expression \, e, \, Field \, f) \equiv \ \text{\texttt{instanceof}}(e, \, \text{InfixExpression}) \land \ e.\text{operator} = "==" \land (\ (\text{\texttt{instanceof}}(e.\text{rightHandSide}, \, \text{NullLiteral}) \land \text{\texttt{instanceof}}(e.\text{leftHandSide}, \, \text{VariableAccess}) \land \ e.\text{leftHandSide}.\text{declaration} = f) \lor (\text{\texttt{instanceof}}(e.\text{leftHandSide}, \, \text{NullLiteral}) \land \text{\texttt{instanceof}}(e.\text{rightHandSide}, \, \text{VariableAccess}) \land \ e.\text{rightHandSide}.\text{declaration} = f))$$

Definition 12. Predicate $\text{checksNullInequality}(Expression, Field)$

$$\text{checksNullInequality}(Expression \, e, \, Field \, f) \equiv \ \text{\texttt{instanceof}}(e, \, \text{InfixExpression}) \land \ e.\text{operator} = "!=" \land (\ (\text{\texttt{instanceof}}(e.\text{rightHandSide}, \, \text{NullLiteral}) \land \text{\texttt{instanceof}}(e.\text{leftHandSide}, \, \text{VariableAccess}) \land \ e.\text{leftHandSide}.\text{declaration} = f) \lor (\text{\texttt{instanceof}}(e.\text{leftHandSide}, \, \text{NullLiteral}) \land \text{\texttt{instanceof}}(e.\text{rightHandSide}, \, \text{VariableAccess}) \land \ e.\text{rightHandSide}.\text{declaration} = f))$$

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful comments that improved the quality of this work. Moreover, we would like to thank Dr. Tsantalis, Prof. Chatzigeorgiou, as well as the rest of the JDeodorant development team members for providing us access to the project’s source code.

References


