Automated refactoring of super-class method invocations to the Template Method design pattern

Vassilis E. Zafeiris\textsuperscript{a}, Sotiris H. Poulias\textsuperscript{a}, N. A. Diamantidis\textsuperscript{a}, E. A. Giakoumakis\textsuperscript{a,*}

\textsuperscript{a}Department of Informatics, Athens University of Economics and Business, 76 Patission Str., Athens 104 34, Greece

Abstract

\textit{Context}: Implementation inheritance, i.e., overriding of concrete method implementations through subtyping, is prone to potential class contract violations. \textit{Call Super} is a code pattern that employs implementation inheritance for extending a method’s behaviour. In \textit{Call Super} the overriding method includes in its body an invocation to the overridden method. \textit{Template Method} is a design pattern that enables extensions to a multi-step procedure without overriding its concrete implementation. Instead, subclasses provide different variants of the template method’s behaviour through implementation of abstract method definitions (interface inheritance).

\textit{Objective}: This work studies the automated refactoring of \textit{Call Super} to \textit{Template Method}, contributing, thus, to replacement of implementation inheritance with interface inheritance.

\textit{Method}: We introduce an algorithm for the discovery of refactoring candidates that is based on an extensive set of refactoring preconditions. Moreover, we specify the source code transformation for refactoring a \textit{Call Super} instance to \textit{Template Method}. An implementation of the proposed approach is evaluated on a set of open source Java projects.

\textit{Results}: The evaluation results highlight (a) the frequent occurrence of the \textit{Call Super} pattern among method overridings, (b) the potential provided by our method for discovery and elimination of several non-trivial \textit{Call Super} instances and (c) the resulting code improvement, as reflected by the \textit{Specialization Index} metric and the alignment of refactored code with the programmer’s intent. The application of all refactorings identified on a set of benchmark projects and the successful execution of their test suites provide empirical evidence on the soundness of the refactoring procedure. Runtime performance results support the scalability of the proposed method.

\textit{Conclusion}: The proposed method automates the replacement of implementation inheritance with interface inheritance through refactoring \textit{Call Super} instances to \textit{Template Method}. The empirical evaluation of the method supports its applicability, soundness and runtime efficiency.

\textit{Keywords}: Call Super, Implementation inheritance, Interface Inheritance, Refactoring, Template Method Design Pattern

\textit{Information and Software Technology, 82(2), 2017, DOI:dx.doi.org/10.1016/j.infsof.2016.09.008}
1. Introduction

Inheritance is a basic mechanism in object-oriented languages for code reuse and polymorphism support. Inheritance can be generally classified into interface inheritance and implementation inheritance [1] depending on whether the type that is extended is abstract or not. Interface inheritance refers to subtyping an interface or providing concrete implementations for abstract methods defined in an abstract class or interface. On the other hand, implementation inheritance characterizes all cases that concrete method implementations are overridden through subtyping. Unlike interface inheritance, implementation inheritance compromises encapsulation [2]. In other words, it creates a strong coupling of the subclasses against the base class that hinders its evolution and maintenance.

The pitfalls of implementation inheritance become clearer under the prism of Liskov’s Substitution Principle [3] and “Design by Contract” requirements on class contract preservation during subclassing [4, 5]. Contract preservation prescribes that a method that overrides a concrete method must preserve class invariants and do not relax the post-conditions of the overridden method [4]. Given the limitations of conventional object-oriented languages on formal specification of class contracts and the inherent difficulty in formulating such specifications, programmers provide them in the form of method documentation that is usually neither formal nor complete. Thus, the potential violation of an overridden method’s contract is contingent on its complexity and the programmer’s familiarity with its purpose.

*Call Super* is a code pattern that employs implementation inheritance for extending a concrete method’s behaviour. In *Call Super* the overriding method includes in its body an invocation to the overridden method through the *super* keyword. Fowler [6] has denominated and classified *Call Super* as an anti-pattern (or code “smell”). However, the issues raised by this code pattern were initially highlighted by Gamma et al. [7] that suggest the use of the Template Method design pattern for controlled extension of a method’s behaviour. The following excerpt is taken from the Template Method section of their popular book on design patterns [7]:

A subclass can extend a parent class operation’s behavior by overriding the operation and calling the parent operation explicitly (code example). Unfortunately, it’s easy to forget to call the inherited operation. We can transform such an operation into a template method to give the parent control over how subclasses extend it.

**Template Method** enables extensions to a multi-step procedure without overriding its concrete implementation. The pattern introduces a method (template method) whose behaviour includes clearly defined extension points in the form of method invocations (steps). These methods are usually abstract and, thus, subclasses can provide different variants of the template method’s behaviour through implementation of abstract method
definitions. In this manner, Template Method favors interface inheritance and contributes, additionally, to easier preservation of method contracts. In fact, preserving the contract of a template method is reduced to preserving the simpler contracts of individual steps. Template Method extension points enable efficient subclassing according to the suggestions of Bloch [1]:

To allow programmers to write efficient subclasses without undue pain, a class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods.

The Call Super code pattern has, also, implications on the coupling of the child and parent classes, as it becomes apparent by the application of the Dependency Inversion Principle (DIP) [5]. According to the DIP, high-level modules should not depend on low-level models, but they should both depend on abstractions. As a guidance for the application of the DIP in the context of class inheritance, Martin [5] suggests that “no method should override an implemented method of any of its base classes”. The negative effect of concrete method overridings on the maintainability and fault-proneness of a class has, also, been empirically validated by Briand et al. [8]. Recent studies, although involving other anti-patterns, support the correlation with change-/fault-proneness for both participants in anti-patterns and their dependencies [9, 10]. The Call Super code pattern violates the DIP, since it involves overriding of a concrete method implementation declared in a parent class. Moreover, the invocation of the overridden method through the super reference introduces a dependency from high-level subclass functionality to base functionality of the parent class. On the other hand, Template Method complies to the DIP, since the parent class functionality (overridden method) depends on abstractions (abstract methods) and is specialized through appropriate implementations provided by subclasses.

This work focuses on the automated refactoring of Call Super to the Template Method design pattern, contributing, thus, to replacement of implementation inheritance with interface inheritance. We specify an algorithm for the discovery of refactoring candidates that is based on an extensive set of refactoring preconditions. These preconditions ensure that the suggested refactorings can be safely applied to the source code. Moreover, we specify the actual source code transformation procedure for the introduction of the Template Method design pattern. An implementation of the proposed approach, integrated in the JDeodorant Eclipse plugin [11], has been evaluated on a set of open source Java projects. The evaluation results highlight (a) the frequent occurrence of the Call Super code pattern among method overridings, (b) the potential provided by our method for discovery and elimination of a satisfactory number of non-trivial Call Super instances and (c) the code improvement due to refactoring as reflected by the Specialization Index class inheritance metric and the alignment of refactored code with the programmer’s intent. The successful execution of the projects’ test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Moreover, runtime performance results support the scalability of our approach.

The rest of this paper is structured as following: Section 2 presents relevant work on refactoring to design patterns with emphasis on the Template Method pattern. Section 3 introduces the Call Super code pattern and specifies our proposal for identification of refactoring candidates and their elimination through the Template Method
design pattern. Section 4 presents an evaluation of this approach on the basis of a prototype implementation integrated in the JDeodorant Eclipse plug-in. Finally, the paper is concluded in Section 5.

2. Related Work

Refactoring automation is an active research area of software evolution. Many approaches have been proposed for automated refactoring of code flaws that can be categorized on the basis of the techniques used for the identification of the refactoring candidates [12]. Metric-based approaches apply and combine software metrics, quantifying a project’s size and design properties (e.g., size and number of class methods, complexity, coupling, cohesion), in order to identify refactoring candidates among low-scoring code fragments [13, 14, 15, 16, 17]. Logic-based approaches operate on an abstract representation of the source code, comprising facts and rules, and detect code flaws through evaluation of appropriate logic rules [18, 19, 20]. Design flaws, in other methods, are specified as patterns and pattern matching techniques are applied for their identification [21, 22]. Search-based approaches map the problem of automated refactoring to a problem of optimizing a fitness function through search in the domain of alternative designs [23, 24, 25, 26]. Probabilistic methods introduce the factor of uncertainty in the refactoring detection process that results in probabilistic rather than strict suggestions of refactoring candidates [27, 28]. Finally, visualization techniques assist the developer to manually identify design flaws by providing appropriate visual representations of the source code that emphasize problematic areas [29, 30, 31, 32]. The reader that requires a more extensive review on the area of software refactoring may also refer to the work of Mens and Tourwe [33].

Automated refactoring to patterns aims at the elimination of design flaws through the introduction of appropriate design patterns. The rest of this section will focus on relevant work in this area with emphasis to refactoring to the Template Method design pattern. Among the earlier approaches is the method proposed by Tokuda and Batory [34]. The method is based on the specification of primitive code transformations that are applied through appropriate tool support. The introduction of a design pattern is then described as a composition of a sequence of parameterized object-oriented transformations. The method is demonstrated through the introduction of the Abstract Factory design pattern in 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 [35]. 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 has been applied to the Abstract Factory, Builder and Singleton patterns.

Refactoring to Abstract Factory is, also, handled through logic programming [36]. The method involves inferencing of refactoring opportunities in a Java code base and application of an appropriate strategy for code transformation. Inferne is based on extraction from Java code of the system design and its representation as a set of Prolog-like predicates that are then converted to Prolog facts. Moreover, design pattern inference rules are defined for each target pattern, that are then transformed to Prolog rules. The identification of refactoring opportunities takes place through issuing of Prolog queries.
Logic programming has, also, been applied for detection of refactoring opportunities to the Composite design pattern [20]. The approach involves transformation of the Java project’s Abstract Syntax Tree (AST) into Prolog facts through the JTransformer engine [37]. Problematic code fragments are identified through definition of appropriate Prolog rules.

Besides refactoring to structural design patterns, recent approaches focus on the elimination of code flaws relevant to complex conditional statements through behavioural design patterns. Specifically, Tsantalis and Chatzigeorgiou [38] handle two cases of refactoring for simplification of conditional expressions, as described by Fowler [39]: (a) replace type code with State/Strategy and (b) replace conditional logic with polymorphism. In the first case, a set of criteria is applied on a subset of the variables, defined inside the target class, that participate in conditional expressions. In the second case, code fragments that perform Runtime Type Identification (RTTI) are discovered in order to be replaced with invocation of polymorphic methods. The method has been implemented and integrated in the JDeodorant Eclipse plug-in [11].

The automated introduction of the Strategy design pattern and its discrimination from the State pattern has been proposed by Christopoulou et al. [40]. Suggested refactorings comprise conditional statements that are characterized by analogies to the Strategy design pattern, in terms of the purpose and selection mode of strategies. The approach, also, specifies the procedure for refactoring to 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 [11].

A method for the elimination of null-checking conditionals through refactoring to the Null Object design pattern has been proposed by Gaitani et al. [41]. The method focuses on null-checking conditionals that are associated with optional fields, i.e., class fields that are prone to null dereferences as they are not initialized in all class instances. The approach introduces an algorithm for the automated discovery of refactoring opportunities and specifies the source code transformation procedure for refactoring an optional field and its associated null-checking conditionals to Null Object. An implementation of this method has, also, been integrated in the JDeodorant Eclipse plug-in [11].

Manual application of the Template Method pattern, in appropriate code fragments, has been documented by Fowler [39] and Kerievsky [42] in their textbooks on software refactoring. A first approach toward semi-automated refactoring to Template Method has been described by Juillerat and Hirshbrunner [43]. The method applies to pairs of methods, that are indicated by the programmer, and belong to different classes sharing a common abstract class ancestor. It employs and extends 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 super-class of the refactored classes. Moreover, an additional abstract method is created in the super-class that has the same signature with the extracted methods.

The text-based clone detection method that is applied in [43] fails to identify candidate clones that have trivial syntactic differences, e.g. in the order of statements or the naming of program identifiers. Hotta et al. [44] apply a more advanced clone detection technique, based on Program Dependence Graphs (PDGs), for the detection of
refactoring candidates to Template Method. Their approach is more oriented toward the identification of behavioural clones, through the detection of isomorphic graphs on PDGs. A clone pair is suggested as a refactoring candidate after the evaluation of a set of preconditions to the context that each clone code fragment is declared. 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. In order to assist the programmer in the application of the transformation, the working implementation of this method highlights the common and different code fragments of candidate methods.

3. Refactoring Call Super to Template Method

3.1. Call Super

Call Super is a code pattern that employs implementation inheritance for extending a concrete method’s behaviour. In Call Super the overriding method includes in its body an invocation to the overridden method through the super keyword\(^1\). It is commonly encountered in code bases with dependencies on frameworks that allow integration of user-specific code through inheritance. Such frameworks are designed in a way that application-specific functionality is introduced through (a) subclassing framework classes, and (b) overriding one or more of their non-final methods that are not abstract and their invocation cannot be omitted due to mandatory framework related functionality. Call Super is also employed when inheriting from application classes for code reuse purposes.

Regardless of its context of use, Call Super denotes that the overriding method requires the parent method’s behaviour, as it either implements its core responsibilities or includes essential management functionality (framework-related tasks, initializations etc.). Therefore, it suggests certain programmer responsibilities during method overriding and specifically: (i) introduction of a super method invocation and (ii) checking the order of the super method invocation with respect to the rest of the overriding method’s code, (iii) preservation of the overridden method’s contract. Failing to meet these responsibilities due to poor method documentation or plain negligence often leads to software errors \(^7\). Moreover, setting multiple obligations to the programmer during subclassing is usually a sign of poor API design \(^6\) and, thus, Call Super is often considered as an anti-pattern.

Figure 1 illustrates a class diagram with a typical use of the Call Super code pattern. Let ParentClass be an abstract or concrete class and redefinableMethod a concrete non-final non-static method declared in that class. The redefinableMethod is overridden in DerivedClass that inherits from ParentClass. The base version of the redefinableMethod will be, henceforth, referred to as OverriddenMethod, while the term OverridingMethod will refer to the method that overrides it. The OverridingMethod in Figure 1 declares a single method invocation statement that calls the respective method of the ParentClass. The term SuperInvocation will be used to refer to such statements.

\(^1\)Object-oriented languages that do not support the super keyword (e.g. C++) provide an alternative syntax for invoking the overridden method. Although our method is currently applied to Java source code, it can be also generalized to other object-oriented languages.
In the general case, the OverridingMethod may include more than one SuperInvocation statements.

This work studies a typical implementation of the Call Super code pattern where the OverridingMethod declares a single SuperInvocation statement in its body. In the rest of this paper, we will use the term Call Super to refer to such instances of the code pattern. The OverridingMethod, also, includes statements that either precede, follow or surround the SuperInvocation. For brevity reasons the term BeforeFragment will be used for code that precedes the SuperInvocation, while AfterFragment will refer to the code that follows it. Figure 2 presents a Call Super instance detected in the Jade open source framework for development of agent-based applications [45]. Method activate has the role of the OverriddenMethod and belongs to the JICPPeer class that implements a peer (client and server) for sending and receiving management commands for the agent execution platform. JICPPeer.activate creates the client and server instances for processing management commands and returns the transport address for receiving connections. The functionality of JICPPeer is enhanced by JICPSPeer that enables interaction over a secure connection. The OverridingMethod prepares the SSL context prior to peer activation and logs the activation status. Figure 2 illustrates the BeforeFragment and AfterFragment of JICPSPeer.activate in solid and dashed outline respectively.

3.2. Template Method

Template Method belongs to the behavioural design patterns, as documented and categorized by Gamma et al. [7]. It contributes to easier preservation of a method’s contract, during subclassing, by enforcing controlled extensions to its behaviour. The pattern introduces a method (template method) whose behaviour includes clearly defined extension points in the form of method invocations. The template method’s behaviour cannot be redefined as a whole since it is declared as final. On the other hand, the methods invoked on extension points can be overridden by subclasses, allowing, thus, different variants of the basic behaviour. Each one of these methods may correspond to (a) a concrete method declared in the template method’s class (hook method), (b) an abstract method that is left to be implemented by child classes, or (c) a factory method [7]. The purpose of each extension point has a narrower scope and is easier
Figure 2: Call Super instance in the JICPSPeer class of the Jade project.

to be documented. Thus, preserving the contract of a template method is reduced to preserving the simpler contracts of hook methods.

The pattern is usually applied during software maintenance for the elimination of code duplication among methods of different classes that share a common parent [7]. Specifically, the common behaviour is extracted and moved to the parent class, while differences are left to the original classes as redefinitions of hook methods or abstract method implementations. In this work Template Method is used for refactoring certain cases of implementation inheritance, characterized by the Call Super code pattern, to interface inheritance.

Replacing a Call Super instance with Template Method provides inversion of control with respect to the invocation of the functionality included in the SuperInvocation and the BeforeFragment, AfterFragment. In the case of Call Super, a client class invokes, either directly or polymorphically, the OverridingMethod that calls in its turn the SuperInvocation. Template Method switches control from the derived to the parent class. The message sent from the client class is, now, resolved to an OverriddenMethod invocation. The latter executes its original behaviour and, additionally, invokes abstract methods that are implemented in the derived class with the behaviour of the BeforeFragment, AfterFragment. The effects of such inversion of control is that (a) the invocation of the OverriddenMethod is never accidentally omitted, and (b) the developers are provided with guidance on extending its behaviour through implementation of two abstract methods. Moreover, preservation of the OverriddenMethod’s contract is enabled through the simpler contracts of the abstract methods.

A basic requirement for refactoring Call Super to Template Method is the successful extraction of the BeforeFragment and AfterFragment parts of the Overriding-
Figure 3: Case study refactoring of Call Super to Template Method in Jade.

Method as separate methods. These methods will be, henceforth, referred to as BeforeMethod and AfterMethod respectively. This requirement on method extraction represents one of the preconditions associated with the refactoring that will be further explained in the rest of this section.

Figure 3 presents the refactoring of two Call Super instances that are identified in the Jade code base [45]. The Call Super instances are declared in JICPSPeer and Maskable-JICPPeer classes that both override the JICPPeer.activate method. The source code of the JICPSPeer.activate is available in Figure 2. The left part of the figure depicts the current class structure, while the right part presents the refactored version. In the refactored version, the overridings of the activate method in JICPSPeer and Maskable-JICPPeer are eliminated and the extensions they provided to the JICPPeer.activate method have been extracted to beforeActivate and afterActivate methods. These methods are concrete implementations of respective hook methods declared in the JICPPeer class and are polymorphically invoked by the JICPPeer.activate that has been transformed to a template method. The formed template method is a three-step procedure comprising one concrete (doActivate) and two abstract steps (beforeActivate, afterActivate). The concrete step includes the original implementation of JICPPeeractivate.
3.3. Identification of candidate refactorings

The identification of Call Super instances that can be automatically refactored to the Template Method design pattern is based on static analysis of the project’s source code. We propose a refactoring identification algorithm that processes the Abstract Syntax Tree (AST) representation of all project classes and analyzes their instance methods that include SuperInvocation statements. The algorithm is based on an extensive set of refactoring preconditions that prevent the introduction of compilation errors and contribute to the preservation of the system’s external behaviour after refactoring. A detailed description of these preconditions is available in Section 3.3.2. A refactoring candidate comprises a triplet \((C_d, m_p, m_d)\) where \(C_d\) is a class that has the role of the Derived-Class in the Call Super instance, while \(m_p, m_d\) correspond to the OverriddenMethod and the OverridingMethod, respectively.

The terminology that is used in this section and is relevant to the Call Super code pattern has already been introduced in section 3.1. Moreover, the AST concepts that are used in the specification of the identification algorithm are depicted in Figure 4. The diagram represents a conceptual model of the AST types, properties and relationships that are relevant to the proposed method. In this model, a program is a single object that has as property \((\text{program.classes})\) the set of all classes that are part of the project’s code base (Class instances).
3.3.1. Refactoring identification algorithm

The refactoring identification algorithm receives as input the project’s code base (program) and generates a set $R$ of Call Super instances that are candidate for refactoring to Template Method. Each refactoring candidate is represented by an instance of the RefactoringCandidate class (Figure 4). The identification of candidate refactorings is a two-stage procedure that is formally described in Algorithm 1.

The first stage of algorithm operation produces an initial set of refactoring opportunities by analyzing each individual class (lines 3–16). The algorithm excludes from further processing any class that violates Class-level preconditions (line 4), since their refactoring may (a) involve changes to library classes (unavailable source code) and/or (b) lead to erroneous code. Section 3.3.2 provides a detailed description of Class-level preconditions. The processing of each eligible class involves the analysis of all methods declared in the class body (lines 5–15). The algorithm searches for instances of the Call Super code pattern among all non-static methods that are not class constructors.

An OverridingMethod $m$ that declares SuperInvocation statements in its body is suggested as refactoring candidate in case that Super-Invocation preconditions and Extract method preconditions are satisfied (lines 8–13). Super-Invocation preconditions evaluate the SuperInvocation statements and the OverriddenMethod with aim to avoid refactorings that result to over-complicated code or are not meaningful to the programmer. A basic constraint that is set requires the declaration of a single SuperInvocation statement in the OverridingMethod (line 9). The reason is that refactoring of multiple SuperInvocation instances in the same method will complicate the formed template method and increase its number of extension points. This property is generally not preferable in template method [7]. On the other hand, Extract method preconditions ensure that the BeforeFragment and the AfterFragment of the OverridingMethod (a) can be successfully extracted as methods of the DerivedClass and (b) can be polymorphically invoked, after refactoring, by the OverriddenMethod without changing the system’s external behaviour. Both types of refactoring preconditions are specified in detail in Section 3.3.2.

The Call Super instance, declared in a method $m$ that satisfies all preconditions, is added to the set $R$ as a new RefactoringCandidate element (line 13). In addition to the OverridingMethod $m$, a RefactoringCandidate triplet comprises the owner class of $m$ ($ctx$) and the OverriddenMethod. The latter corresponds to the declaring method of the single SuperInvocation statement in the OverridingMethod (line 9). The reason is that refactoring of multiple SuperInvocation instances in the same method will complicate the formed template method and increase its number of extension points. This property is generally not preferable in template method [7]. On the other hand, Extract method preconditions ensure that the BeforeFragment and the AfterFragment of the OverridingMethod (a) can be successfully extracted as methods of the DerivedClass and (b) can be polymorphically invoked, after refactoring, by the OverriddenMethod without changing the system’s external behaviour. Both types of refactoring preconditions are specified in detail in Section 3.3.2.

The second stage of processing in Algorithm 1 involves further analysis of refactoring candidates that belong to the same class hierarchy (lines 17–22). The algorithm iterates over all elements of $M_p$ and retrieves for each method $m'$ the set $R_{m'}$ ($R_{m'} \subseteq R$) of refactoring candidates that have $m'$ as the OverriddenMethod. The Call Super instances, that correspond to the elements of a given set $R_{m'}$, are declared in classes that are sibling subclasses of the same ParentClass ($m' .ownerClass$). Each set $R_{m'}$ is checked for potential violation of Sibling preconditions. These preconditions ensure that the template method, introduced in a certain ParentClass, is consistent across all refactored sibling subclasses. A detailed specification of Sibling preconditions is provided in Section 3.3.2. A set $R_{m'}$ that violates these preconditions is excluded from the set of refactoring candidates
Algorithm 1: Algorithm for identification of refactoring candidates to Template Method in a given project.

```
input : Project program
output: Set[RefactoringCandidate] R

1  R ← ∅;
2  M_p ← ∅;
3 foreach Class ctx ∈ program.classes do
4   if violatesClassPreconditions(ctx) = true then continue;
5   foreach Method m ∈ ctx.methods do
6     if m.constructor = true then continue;
7     if static ∈ m.nonAccessModifiers then continue;
8      /* SuperInvocation preconditions */
9     S ← m.superInvocations;
10    if S = ∅ or |S| > 1 then continue;
11       overridden ← S_1.declaringMethod;
12      if evaluateOverriddenMethod(overridden) = false then continue;
13      /* Extract method preconditions */
14      if violatesExtractMethodPreconditions(m) = true then continue;
15      /* Create a new refactoring candidate */
16      R ← R ∪ {RefactoringCandidate(ctx, overridden, m)};
17      M_p ← M_p ∪ {overridden};
18   end
19   /* Exclude candidate refactorings that violate Sibling Preconditions. */
20   foreach Method m' ∈ M_p do
21      R_{m'} ← filterByOverriddenMethod(R, m');
22      if violatesSiblingPreconditions(R_{m'}) = true then
23         R ← R \ R_{m'};
24     end
25 end
26 return R;
```
The elements of $R$ that pass this filtering procedure are returned as algorithm output.

### 3.3.2. Refactoring preconditions

Refactoring preconditions are evaluated by the identification algorithm on each instance of the Call Super code pattern in order to be suggested as candidate for automated refactoring to Template Method. The role of refactoring preconditions is to prevent the application of refactorings that would lead to erroneous refactored code, in the sense that it would either have compilation errors or it would not preserve the external behaviour of the system. Moreover, they reject refactorings that result to over-complicated code or are not meaningful to the programmer. In this section, we specify in detail an extensive set of preconditions that are grouped in four categories on the basis of their scope of application: (a) Class-level preconditions, (b) Super-Invocation preconditions, (c) Extract method preconditions and (d) Sibling preconditions. Class-level, Extract method and Sibling preconditions focus, mainly, on the applicability of the refactoring and the correctness of refactored code. On the other hand, the majority of Super-Invocation preconditions prevent the introduction of trivial Template Method instances that do not match the programmer’s intent.

**Class-level preconditions.** They verify structural properties of the ParentClass and DerivedClass that are required for a correct application of the source code transformation.

1. The ParentClass must belong to the project’s code base. Since the source code transformation introduces changes to that class, it must not be part of an external library or other unmodifiable component.

2. The ParentClass must be a direct ancestor of the DerivedClass. This restriction does not allow any interposed class in the hierarchy to redefine the OverriddenMethod and, thus, cancel the invocation of the template method and the hook method redefinitions in the DerivedClass.

**Super-Invocation preconditions.** The focus of these preconditions is on the SuperInvocation statement, as well as on properties of the OverriddenMethod.

1. There must be a single SuperInvocation in the OverridingMethod.

2. The OverriddenMethod must not be an empty method.

3. The OverriddenMethod must not be one of clone, toString, equals, compareTo, hashCode, finalize. This precondition refers to the Java programming language. A similar precondition can be defined for any other object oriented language.

4. The OverriddenMethod must not be a property accessor method (i.e., “getter”, “setter” method).

5. The access level of the OverridingMethod must not be less restrictive than that of the OverriddenMethod. For instance, if the OverriddenMethod has protected visibility then the OverridingMethod should not be declared as public. The
precondition ensures that the \texttt{OverriddenMethod} can be invoked polymorphically in any call site of the \texttt{OverridingMethod} and, thus, the elimination of the \texttt{OverridingMethod}, after the refactoring, does not introduce compilation errors.

\textit{Extract method preconditions.} These preconditions ensure that the \texttt{BeforeFragment} and \texttt{AfterFragment} can be extracted as separate methods (\texttt{BeforeMethod} and \texttt{AfterMethod}, respectively) and appropriate \textit{hook} or abstract methods can be declared in the \texttt{Parent-Class}. No preconditions are required for the extraction of the \texttt{OverriddenMethod} body to \texttt{doOverriddenMethod} (see Section 3.4).

1. The \texttt{SuperInvocation} must be a top-level statement in the \texttt{OverridingMethod} body or it may be nested in one or more \texttt{try} blocks. This precondition rejects \textit{Call Super} instances where the \texttt{SuperInvocation} is part of blocks like \texttt{while}, \texttt{for}, \texttt{if/else}. In such cases, extracting the \texttt{BeforeMethod} and \texttt{AfterMethod} would require splitting the block among two methods.

2. The \texttt{SuperInvocation} must not be part of a \texttt{catch} clause. A \texttt{SuperInvocation} nested in a \texttt{catch} block would require extracting the block to two different methods during refactoring.

3. In case that the \texttt{SuperInvocation} is nested in a \texttt{try} block ensure that all statements raising the exceptions caught by the \texttt{try/catch} block belong to the \texttt{AfterFragment}. The precondition ensures that the \texttt{BeforeFragment} and the \texttt{SuperInvocation} do not throw an exception and, thus, need not be part of the \texttt{try/catch} block. In this case, the \texttt{try/catch} block starts, practically, after the \texttt{SuperInvocation} and it can be extracted as part of the \texttt{AfterMethod} with the rest of the \texttt{AfterFragment} statements. As concerning the statements belonging to \texttt{BeforeFragment}, they can be extracted to \texttt{BeforeMethod} without being enclosed in a \texttt{try/catch} block.

4. The \texttt{BeforeFragment} must not include any \texttt{return} statements. After extracting \texttt{BeforeFragment} as a separate method, these statements would cause the termination of \texttt{BeforeMethod} and not \texttt{OverridingMethod} that were their initial intent. In such cases, the application of the refactoring leads to different program behaviour.

5. The \texttt{BeforeFragment} may define through assignment the value of at most one local variable or parameter that is used in the \texttt{SuperInvocation} statement or the \texttt{AfterFragment}. Let $D_{v,b}$ be the set of local variables that are defined in the \texttt{BeforeFragment} and used in the \texttt{SuperInvocation} or the \texttt{AfterFragment}. Moreover, let $D_{p,b}$ be the respective set of parameters. It must hold that $|D_{v,b}| + |D_{p,b}| \leq 1$. Let $v_d$ be a local variable or parameter defined in the \texttt{BeforeFragment} and used in the rest of the \texttt{OverridingMethod}. After refactoring, the variable will be returned by the extracted \texttt{BeforeMethod} and will be used in the template method\textsuperscript{1} that is formed in the body of \texttt{OverriddenMethod}. Specifically, the returned value may

\textsuperscript{1}Recall that the template method is a three-step procedure: \texttt{BeforeMethod}, \texttt{doOverriddenMethod} and \texttt{AfterMethod}. 

14
be provided as an actual parameter to the \texttt{doOverriddenMethod}, \texttt{AfterMethod} or both, depending on whether in the \texttt{OverridingMethod} \texttt{v_d} was used in the \texttt{Super Invocation}, \texttt{AfterFragment} or both, respectively.

In case that the \texttt{BeforeFragment} violated this precondition, the extracted \texttt{BeforeMethod} would have to return more than one values that is not allowed in most object oriented languages.

6. The \texttt{SuperInvocation} must receive as actual parameters the corresponding formal parameters of the \texttt{OverridingMethod}; this correspondence may be violated in at most one parameter position. In other words, at most one parameter position of the \texttt{SuperInvocation} may be filled with an expression that is different from the respective parameter of the \texttt{OverridingMethod}. After refactoring, this expression will be returned by the extracted \texttt{BeforeMethod} and the return value will be used in the same parameter position of \texttt{doOverriddenMethod}. Since \texttt{BeforeMethod} can return a single value, the satisfaction of this precondition depends also on the status of precondition 5. Specifically, in case that \(|D_{t,v} + D_{p,b}| = 1\) then all actual parameters of \texttt{SuperInvocation} must match the formal parameters of \texttt{OverridingMethod}. Otherwise, \texttt{BeforeMethod} would have to return more than one values.

The intuition behind this precondition is that since the \texttt{OverridingMethod} is eliminated after refactoring, the \texttt{doOverriddenMethod} parameters must match (a) the \texttt{SuperInvocation} parameters, in case that the \texttt{OverriddenMethod} is invoked polymorphically from a \texttt{DerivedClass} instance, or (b) the \texttt{OverriddenMethod} parameters, in case that the \texttt{OverriddenMethod} is invoked from a \texttt{ParentClass} instance. Thus, only one parameter of \texttt{doOverriddenMethod} can be dynamically fixed on the basis of the invocation target and the return value of \texttt{BeforeMethod}.

7. The sizes of \texttt{BeforeFragment} and \texttt{AfterFragment} must not be less than two statements at the same time. In other words, it is not suggested as refactoring opportunity an \texttt{OverridingMethod} with a very simple method body.

\textit{Sibling preconditions}. These preconditions apply in cases that the \texttt{DerivedClass}, declaring the \texttt{Call Super} instance, is not the only direct subclass of the \texttt{ParentClass}. In other words, the \texttt{DerivedClass} has one or more siblings (\texttt{SiblingClass}) that are direct descendants of the \texttt{ParentClass} and may include \texttt{Call Super} instances referring to the same \texttt{OverriddenMethod}. A violation of a sibling precondition leads to rejection from refactoring candidates of all \texttt{Call Super} instances in the \texttt{DerivedClass} and each \texttt{SiblingClass} that refer to the \texttt{OverriddenMethod}. These preconditions are listed below:

1. The return type of \texttt{BeforeMethod} must be the same across the \texttt{DerivedClass} and each \texttt{SiblingClass}. The precondition ensures the consistency of the \texttt{BeforeMethod} signature across all sibling classes.

2. In case that \texttt{BeforeMethod} returns a result, it must be used as actual parameter for the same formal parameter of \texttt{SuperInvocation} in the \texttt{DerivedClass} and each \texttt{SiblingClass}.

3. In case that a \texttt{SiblingClass} includes an \texttt{OverridingMethod}
• the OverridingMethod must declare a valid Call Super instance,
• the Call Super instance must not violate Super Invocation or Extract Method
preconditions (except for Extract Method precondition 7, as will be explained
below).

4. The OverriddenMethod must not be overridden in subclasses of the DerivedClass
and each SiblingClass. Since the OverriddenMethod is declared as final, after
refactoring to TEMPLATE METHOD, further overridings will lead to compilation
errors.

5. At least one of the Call Super instances declared in the DerivedClass and its
siblings must not violate Extract Method precondition 7. The precondition ensures
that BeforeMethod and AfterMethod do not have empty or single-line implementa-
tions in all subclasses of ParentClass. We relax precondition 7 in the presence
of at least one non-trivial Call Super instance among sibling subclasses, in order
to provide for potential future evolution of the simple cases to more complex func-
tionality.

3.4. Code transformation

The code transformation specifies the required refactoring steps for introducing TEM-
plate METHOD in a successfully identified Call Super refactoring opportunity. It can
be decomposed into the following more primitive refactorings:

1. Apply Extract Method to the OverriddenMethod's body. The new method, doOver-
riddenMethod, has protected visibility and the same argument list and return type
as the OverriddenMethod.

2. Replace the SuperInvocation with a call to doOverriddenMethod. In case that
the SuperInvocation has non-void return type, the result of doOverriddenMethod
execution is assigned to a local variable (superReturnVariable). If the Super-
Invocation result was assigned to a local variable, prior to refactoring, super-
ReturnVariable keeps the same name.

3. Apply Extract Method to the BeforeFragment of the OverridingMethod
   • the extracted method, BeforeMethod, receives the OverridingMethod param-
     eters as method invocation arguments. These parameters are, also, declared
     in the BeforeMethod signature.
   • the result of BeforeMethod execution, if any, is assigned to a local variable of
     appropriate type (beforeReturnVariable) in the OverridingMethod.

4. Apply Extract Method to the AfterFragment of the OverridingMethod
   • the extracted method, AfterMethod, receives as arguments (a) the Over-
     ridingMethod parameters, (b) the superReturnVariable, if doOverridden-
     Method has non-void return type, and (c) the beforeReturnVariable, if
     BeforeMethod has non-void return type. On the basis of these arguments,
     appropriate parameters are declared in the AfterMethod signature.
Figure 5: Application of the refactoring (steps 1–5) to the *Call Super* instance of Figure 2.

- the return type of *AfterMethod* matches that of the *OverridingMethod*, since the *AfterFragment* declares all return statements of the method body.

5. Create the BeforeMethod and AfterMethod abstract or hook methods in the Parent-Class on the basis of the respective methods declared in the DerivedClass.

6. Pull up the OverridingMethod implementation in the inheritance hierarchy and replace OverriddenMethod with it.

7. Change the visibility of doOverriddenMethod to private and make the OverriddenMethod final in order to enforce controlled extensions to its behaviour through hook method overriding.

The Extract Method refactoring extracts part of a method’s body into a new method that is invoked in place of the extracted code fragment. The details of applying this refactoring have been thoroughly documented by Fowler [39] and will not be further
elaborated in this paper. However, the Extract Method refactoring applied to the Before-Fragment (Step 3) involves some variations related to the return value of BeforeMethod. According to the basic transformation, BeforeMethod returns the value of a local variable or parameter of the OverridingMethod that is defined in the BeforeFragment and used in the rest of the method body. The returned value is assigned to the beforeReturnValue that is named after the defined local variable or parameter.

The pull up of the OverridingMethod to the ParentClass (Step 6) introduces further requirements for the return value of BeforeMethod. Specifically, moving the Super-Invocation (after the transformation of Step 2) to the ParentClass may lead to compilation errors in case that the method invocation statement includes, in its parameter list, (a) references to fields that are defined in the DerivedClass, (b) invocations of methods that are declared in the DerivedClass. In order to overcome this issue the extracted BeforeMethod may return a field value or a method invocation expression, provided that Extract method preconditions 5 and 6 are not violated.

Figure 5 presents the first five steps of the refactoring procedure. Each step, identified by its number, is associated through arrows with the code fragments that introduces.
The first four steps extract the doActivate, beforeActivate and afterActivate methods and eliminate the SuperInvocation. Step 5 introduces hook methods for beforeActivate and afterActivate in the parent class. Note that hook method creation applies in the case of a concrete parent class. If JICPPeer were an abstract class, the aforementioned methods would have been declared as abstract. Figure 6 presents the remaining steps of the refactoring procedure. Specifically, Step 6 involves replacement of JICPPeer.activate with the respective method of the derived class. The latter is eliminated from JICPSPeer. In Step 7, the visibility of doActivate is changed to private and JICPPeer.activate is set as final.

4. Experimental Evaluation

We have evaluated the proposed method for automated refactoring of Call Super instances to the Template Method design pattern through an experimental study. The goal of this study is to analyze our method for the purpose of evaluating (a) its applicability, (b) its effectiveness in improving the design of a software project and (c) its practicality for being integrated in tools that assist developers in their routine code maintenance tasks. The study is conducted from the perspective of a researcher that seeks to evaluate the automation of a complex refactoring in the context of a set of open source projects.

4.1. Context selection

The context of the experimental evaluation comprises 12 open source projects (benchmark projects). The selection of benchmark projects is based on the following requirements: (a) their source code must be publicly available for study reproducibility reasons, (b) they must be written in Java, since our analysis is based on software with Java parsing capabilities, (c) they must range in size and make moderate to high use of class inheritance. The use of inheritance is estimated on the basis of the number of subclasses that are present in each code base. Table 1 presents the projects along with relevant source code metrics\(^1\). Columns 2–4 present the total lines of code without blank lines and comments (SLOC), the number of classes and the total number of class methods for each project. Columns 5–6 include the total number of subclasses (direct subclasses of java.lang.Object are excluded), their ratio over all project classes (enclosed in parentheses) and the total number of “Overridings”, i.e., methods that override a super-class method. Column 7 presents the projects’ test coverage, i.e., the ratio of code instructions covered by the execution of each project’s test suite over the total source code instructions of the project\(^2\). Since jade 4.4 does not provide a publicly available test suite, its code coverage is estimated on the basis of tests implemented for the needs of this study\(^3\). Finally, projects are sorted by ascending size (SLOC number) and the same pattern is followed in all tables that are included in Section 4.

The analysis of benchmark projects for the discovery of refactoring opportunities to Template Method is based on a Java implementation of the proposed refactoring

---

\(^1\)Project size and inheritance metrics were estimated with the Metrics v.1.3.6 plugin for Eclipse.

\(^2\)Estimated with the use of the JaCoCo v.0.7.2 code coverage library for Java.

\(^3\)Source code is available at https://github.com/bzafiris/jade-test-suite
Table 1: Benchmark projects and their source code attributes.

<table>
<thead>
<tr>
<th>Project name</th>
<th>SLOC</th>
<th>Classes</th>
<th>Methods</th>
<th>Subclasses</th>
<th>Overridings</th>
<th>Code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j 2.0.0rc1</td>
<td>17,863</td>
<td>153</td>
<td>2,500</td>
<td>66 (43%)</td>
<td>353</td>
<td>46%</td>
</tr>
<tr>
<td>xstream 1.4.8</td>
<td>19,373</td>
<td>334</td>
<td>2,141</td>
<td>174 (52%)</td>
<td>242</td>
<td>78%</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>90,125</td>
<td>806</td>
<td>7,744</td>
<td>445 (49%)</td>
<td>749</td>
<td>61%</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>78,457</td>
<td>736</td>
<td>5,516</td>
<td>189 (26%)</td>
<td>747</td>
<td>44%</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>94,970</td>
<td>916</td>
<td>7,744</td>
<td>445 (49%)</td>
<td>749</td>
<td>58%</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>93,460</td>
<td>510</td>
<td>7,796</td>
<td>258 (33%)</td>
<td>332</td>
<td>54%</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>101,679</td>
<td>1,009</td>
<td>9,320</td>
<td>692 (65%)</td>
<td>919</td>
<td>46%</td>
</tr>
<tr>
<td>jade 4.4</td>
<td>110,827</td>
<td>1,174</td>
<td>7,357</td>
<td>408 (35%)</td>
<td>698</td>
<td>21%</td>
</tr>
<tr>
<td>jena 2.1.0</td>
<td>112,520</td>
<td>718</td>
<td>7,811</td>
<td>465 (51%)</td>
<td>1,107</td>
<td>24%</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>161,020</td>
<td>1,004</td>
<td>8,666</td>
<td>521 (52%)</td>
<td>1,802</td>
<td>15%</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>179,322</td>
<td>1,941</td>
<td>13,170</td>
<td>988 (51%)</td>
<td>1,575</td>
<td>54%</td>
</tr>
<tr>
<td>top 1.1</td>
<td>179,928</td>
<td>1,913</td>
<td>14,472</td>
<td>1,137 (59%)</td>
<td>2,290</td>
<td>67%</td>
</tr>
</tbody>
</table>

A screen capture of the Call Super plugin, after its execution on the Jade project, is presented in Figure 7. The identified refactoring candidates to Template Method are listed in the table with title “Call Super”, positioned on the lower part of the Eclipse workbench. Each list item corresponds to an OverriddenMethod and groups the identified Call Super instances for that method. The expanded sublist includes more than one elements in case of a SuperInvocation encountered in two or more sibling subclasses. The application of the refactoring on a selected refactoring candidate is activated by an appropriate button in the top right part of the table.
4.2. Research questions

The experimental evaluation of the proposed method aims at addressing the following research questions:

RQ1 How common is the usage of the Call Super code pattern? This research question seeks to study the presence of the Call Super code pattern in software projects. Since Call Super instances constitute the search domain for refactoring candidates, the results of this study will provide insight into the applicability of the proposed method.

RQ2 How effective is the proposed method in identifying refactoring opportunities to Template Method? This research question aims at evaluating the effectiveness of the proposed method through estimation of the number of refactoring candidates on benchmark projects. The results of this study will highlight the potential of the proposed refactoring identification algorithm for discovering several refactoring candidates on benchmark projects.

RQ3 Does the proposed source code transformation preserve the correctness of refactored code? This research question investigates correctness, a basic requirement for practically applying the proposed complex refactoring during programmers’ routine maintenance tasks. The results of this study will confirm that the application of suggested refactorings in benchmark projects does not introduce compilation errors and preserves the external behaviour of refactored code.

RQ4 How practical is the proposed method for the developer? The research question focuses on practical aspects of the proposed method that promote its adoption by software developers. The results of this question will quantify the code improvement due to refactoring through an object oriented metric (Specialization Index class inheritance metric [46]) and highlight the alignment of the introduced Template Method with the programmer’s intent. Finally, the results will confirm the runtime efficiency of the refactoring identification algorithm.

4.3. Evaluation results

4.3.1. RQ1. How common is the usage of the Call Super code pattern?

We have studied the presence of the Call Super code pattern in the benchmark projects with the help of JDeodorant. The study involved an analysis of how frequent is the occurrence of Call Super instances among (a) methods that include at least one method invocation through the super reference and (b) methods that override a superclass method. For brevity reasons, we use the term “super-invocation” to refer to a method invocation through the super reference, as opposed to the SuperInvocation that refers to an invocation to the overridden Method through the super reference. Our analysis is based on the following sets of methods that are characterized by the presence of super-invocation statements in their body:

- \( M_s \): set of methods that (a) are not class constructors and (b) include in their method body at least one super-invocation. A method \( m \in M_s \) may not necessarily override a ParentClass method. Moreover, in case that it overrides a method it may not include a SuperInvocation in its method body, i.e., the method referenced
through the super keyword may not be the OverriddenMethod. The set \( M_s \) is partitioned into three subsets \( M_{1cs}^s, M_{ncs}^s, M_{*s}^s \), i.e., \( M_s = M_{1cs}^s \cup M_{ncs}^s \cup M_{*s}^s \).

- \( M_{1cs}^s \): set of methods that include a single SuperInvocation in their method body. These methods correspond to typical instances of the Call Super code pattern. This set is analyzed by the proposed refactoring identification algorithm for the discovery of refactoring candidates to Template Method. It holds that \( M_{1cs}^s \subseteq M_s \).

- \( M_{ncs}^s \): set of methods that include two or more SuperInvocation statements in their method body. It holds that \( M_{ncs}^s \subseteq M_s \) and \( M_{ncs}^s \cap M_{1cs}^s = \emptyset \).

- \( M_{*s}^s \): set of methods that declare in their method body at least one super-invocation that is not a SuperInvocation statement (invocation of the OverriddenMethod). It holds that \( M_{*s}^s \subseteq M_s \), \( M_{*s}^s \cap M_{1cs}^s = \emptyset \) and \( M_{*s}^s \cap M_{ncs}^s = \emptyset \).

The results of benchmark projects’ analysis are summarized in Table 2. The frequency of the Call Super code pattern among methods with super-invocations is analyzed in columns 2–5. Column 2 presents the number of methods of each project that belong to the \( M_{1cs}^s \) set and their respective ratio over the total number of methods that declare super-invocations in their body (\( M_s \)). Specifically, methods that include instances of the Call Super code pattern (\( M_{1cs}^s \)) range from 74.2% to 96.7% (86.7% on average) of methods with super-invocations (\( M_s \)). The presence of more than one SuperInvocation statements in an OverridingMethod is not frequent, since methods belonging to \( M_{ncs}^s \) correspond to 1.9% of \( M_s \) members, on average. The rest of the methods with super-invocations in their body (\( M_{*s}^s \)) represent, also, a small part of the \( M_s \) set (11.4% on average). These results suggest that the \( M_s \) methods in each benchmark project are dominated by instances of the Call Super code pattern.

| Project name | Methods with super invocations | \( M_{1cs}^s \) | \( M_{ncs}^s \) | \( M_{*s}^s \) | \( |M_{1cs}^s| / \text{Overridings} \) |
|--------------|-------------------------------|---------------|---------------|---------------|----------------------------------|
| dom4j 2.0.0rc1 | 87 (90.6%) | 5 (5.2%) | 4 (4.2%) | 96 | 87/353 (24.6%) |
| xstream 1.4.8 | 73 (76.8%) | 5 (5.3%) | 17 (17.9%) | 95 | 73/242 (30.2%) |
| myfaces-impl 2.1.9 | 123 (93.9%) | 2 (1.5%) | 6 (4.6%) | 131 | 123/747 (16.5%) |
| jackrabbit-core 2.9 | 132 (86.3%) | 2 (1.3%) | 19 (12.4%) | 153 | 132/405 (32.8%) |
| meter 2.9 | 299 (84.2%) | 3 (0.8%) | 53 (14.9%) | 355 | 299/749 (39.9%) |
| freetext 1.0.14 | 446 (96.7%) | 3 (0.7%) | 12 (2.6%) | 460 | 446/582 (76.5%) |
| apache ant 1.8.2 | 311 (74.2%) | 10 (2.4%) | 98 (23.4%) | 419 | 311/791 (39.8%) |
| jade 4.4 | 193 (81.4%) | 5 (2.1%) | 39 (16.5%) | 267 | 193/698 (27.7%) |
| severes 2.11.0 | 154 (86.2%) | 0 (0.0%) | 31 (18.8%) | 225 | 154/1107 (14.0%) |
| xalan 2.7 | 434 (82.2%) | 6 (1.1%) | 88 (16.7%) | 530 | 434/1862 (23.3%) |
| batik 1.7 | 390 (85.0%) | 22 (5.0%) | 31 (7.0%) | 443 | 390/1575 (24.8%) |
| fop 1.1 | 579 (93.4%) | 7 (1.1%) | 34 (6.0%) | 620 | 579/2290 (24.8%) |
| AVERAGE (%) | 86.7% | 1.9% | 11.4% | - | 28.3% |

Call Super is, also, a common code pattern among methods that override a superclass method. Column 6 of Table 2 (“\(|M_{1cs}^s|/\text{Overridings} \)” presents the ratio of methods
that include a Call Super instance over total overridings in each benchmark project. This ratio ranges from 16.5% to 76.5% (28.3% on average) and highlights the broad applicability of our method among project subclasses.

4.3.2. RQ2. How effective is the proposed method in identifying refactoring opportunities to Template Method?

We have analyzed all benchmark projects with the refactoring identification algorithm that is implemented as a JDeodorant module. The refactoring candidates, identified in each project, are analyzed in Table 3. The table juxtaposes the number of refactoring candidates (Column 2) with the number of Call Super instances (Column 3) for each project. Refactoring candidates are, also, expressed as a ratio of Call Super instances in Column 4. This ratio ranges from 1.6% to 20.5% and has an average value of 8.0%. Thus, our method contributes to elimination of implementation inheritance through the Template Method pattern in several cases of concrete method overridings.

Table 3: Identified refactoring candidates in benchmark projects.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Refactoring Candidates (M_{1c})</th>
<th>Call Super instances (M_{1cs})</th>
<th>( \frac{M_{1cs}}{M_{1c}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j 2.0.0rc1</td>
<td>2</td>
<td>87</td>
<td>2.4%</td>
</tr>
<tr>
<td>xstream 1.4.8</td>
<td>4</td>
<td>73</td>
<td>5.5%</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>8</td>
<td>123</td>
<td>6.5%</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>27</td>
<td>132</td>
<td>20.5%</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>16</td>
<td>299</td>
<td>5.4%</td>
</tr>
<tr>
<td>jtracechart 1.0.14</td>
<td>7</td>
<td>445</td>
<td>6.6%</td>
</tr>
<tr>
<td>apacice ant 1.8.2</td>
<td>49</td>
<td>341</td>
<td>7.3%</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>22</td>
<td>194</td>
<td>11.3%</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>24</td>
<td>434</td>
<td>16.8%</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>27</td>
<td>390</td>
<td>12.3%</td>
</tr>
<tr>
<td>fop 1.1</td>
<td>11</td>
<td>579</td>
<td>14.5%</td>
</tr>
<tr>
<td>AVERAGE (%)</td>
<td>-</td>
<td>-</td>
<td>8.0%</td>
</tr>
</tbody>
</table>

Table 4: Rejected Call Super instances per precondition category.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Rejections % per precondition category</th>
<th>Class-level</th>
<th>Super-Invocation</th>
<th>Extract method</th>
<th>Sibling</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j 2.0.0rc1</td>
<td>63.6</td>
<td>9.1</td>
<td>9.7</td>
<td>21.6</td>
<td></td>
</tr>
<tr>
<td>xstream 1.4.8</td>
<td>27.1</td>
<td>1.7</td>
<td>27.2</td>
<td>30.6</td>
<td></td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>95.0</td>
<td>0.8</td>
<td>3.4</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>41.1</td>
<td>26.3</td>
<td>15.8</td>
<td>16.8</td>
<td></td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>74.7</td>
<td>10.2</td>
<td>4.9</td>
<td>10.2</td>
<td></td>
</tr>
<tr>
<td>jtracechart 1.0.14</td>
<td>18.7</td>
<td>74.9</td>
<td>8.7</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>apacice ant 1.8.2</td>
<td>26.4</td>
<td>32.8</td>
<td>22.1</td>
<td>18.6</td>
<td></td>
</tr>
<tr>
<td>jade 4.4</td>
<td>40.6</td>
<td>13.9</td>
<td>9.1</td>
<td>36.4</td>
<td></td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>17.7</td>
<td>39.6</td>
<td>27.5</td>
<td>15.2</td>
<td></td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>43.6</td>
<td>29.7</td>
<td>17.4</td>
<td>19.3</td>
<td></td>
</tr>
<tr>
<td>batik 1.7</td>
<td>38.3</td>
<td>18.9</td>
<td>16.0</td>
<td>26.8</td>
<td></td>
</tr>
<tr>
<td>fop 1.1</td>
<td>46.2</td>
<td>13.1</td>
<td>6.8</td>
<td>33.9</td>
<td></td>
</tr>
<tr>
<td>AVERAGE (%)</td>
<td>40.6</td>
<td>27.1</td>
<td>12.3</td>
<td>20.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 analyzes the refactoring candidates that were rejected due to precondition violations (members of the \( M_{1bs} \setminus \bar{M_{1cs}} \) set). Specifically, it presents for each project the fraction of rejections that correspond to each precondition category. Most rejections are due to Class-level preconditions (40.6% on average) and Super-Invocation preconditions (27.1% on average). The cases rejected by Super-Invocation preconditions are mainly
characterized by an OverriddenMethod that is either empty or is one of toString, get/set, clone, equals, compareTo etc. Extract method preconditions handled 12.3% of the rejections on average and most of them are characterized by a SuperInvocation nested in control statements. Finally, Sibling preconditions contribute to the rejection of several refactoring candidates (20.0% on average) that are, mainly, characterized by (a) simple implementations of the OverridingMethod in most sibling subclasses (precondition 5) and (b) overridings of the OverriddenMethod in non-children descendants of the ParentClass (precondition 4).

4.3.3. RQ3. Does the proposed source code transformation preserve the correctness of refactored code?

We have empirically evaluated the soundness of the proposed source code transformation in terms of: (a) the syntactic correctness of refactored code and (b) the preservation of refactored projects’ external behaviour. The evaluation involved the application of all refactorings to TEMPLATE METHOD that were automatically identified in the benchmark projects by the implemented JDeodorant module. The application of each suggested refactoring was, also, supported by our tool and did not affect the successful compilation of the respective project’s source code. The latter provides empirical evidence on the syntactic correctness of the proposed transformation.

The behaviour preservation property of the source code transformation has been empirically validated through the execution of each project’s test suite on the refactored source code. The results of test suites’ execution on the refactored projects were identical to the results of execution on their original version. Thus, the external behaviour of benchmark projects is preserved after the application of the refactorings, at least to the extent that can be verified by the execution of their test suites.

4.3.4. RQ4. How practical is the proposed method for the developer?

We have studied the practicality of the proposed method in terms of (a) the code improvement after refactoring as evaluated through the Specialization Index (SIX) class inheritance metric [46], (b) the alignment of refactored code with the programmer’s intent on the original version of the code, and (c) the runtime efficiency of the refactoring identification procedure.

Software metrics provide a framework for an objective assessment of source code quality with the use of automated tools. However, the selection of metrics that best reflect the desired quality properties of source code is not an easy task. In this work, the focus is on source code quality improvement through replacement of interface inheritance with implementation inheritance. Simple inheritance metrics such as depth of inheritance tree or number of inherited/overridden methods cannot individually assess the impact of this transformation. In this study, we have adopted a composite inheritance metric, the Specialization Index (SIX) [46], in order to evaluate refactored subclasses. The SIX metric expresses the proportion of concrete overridings among all the methods of a given class, multiplied with the depth of the class in the inheritance hierarchy. Its definition is given by Equation 1 and is based on the following metrics:

- DIT (Depth of Inheritance Tree): the length of the longest path from a class to the root of its inheritance hierarchy,
• **NMO (Number of Methods Overridden):** the number of methods of a class that override a concrete super-class method,

• **NMA (Number of Methods Added):** the number of new methods that are declared in the class,

• **NMI (Number of Inherited Methods):** the number of methods that are inherited from the ancestors of the class.

\[
\text{SIX} = \frac{\text{DIT} \times \text{NMO}}{\text{NMO} + \text{NMA} + \text{NMI}}
\]

Table 5: Impact of refactoring on the Specialization Index metric.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>SIX__p</th>
<th>SIX__a</th>
<th>Decrease (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j 2.0.0rc1</td>
<td>0.734 (0.890)</td>
<td>0.620 (0.876)</td>
<td>15.53</td>
</tr>
<tr>
<td>xstream 1.4.8</td>
<td>0.724 (0.328)</td>
<td>0.389 (0.201)</td>
<td>47.01</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>0.667 (0.392)</td>
<td>0.469 (0.277)</td>
<td>29.08</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>0.253 (0.251)</td>
<td>0.142 (0.182)</td>
<td>43.43</td>
</tr>
<tr>
<td>jmeter 2.9</td>
<td>0.116 (0.066)</td>
<td>0.076 (0.066)</td>
<td>34.48</td>
</tr>
<tr>
<td>jfreechart 1.0.14</td>
<td>0.113 (0.095)</td>
<td>0.064 (0.067)</td>
<td>43.36</td>
</tr>
<tr>
<td>apache ant 1.3.2</td>
<td>0.239 (0.118)</td>
<td>0.113 (0.115)</td>
<td>52.72</td>
</tr>
<tr>
<td>jade 4.4</td>
<td>0.306 (0.180)</td>
<td>0.143 (0.158)</td>
<td>53.27</td>
</tr>
<tr>
<td>xerces 2.11.0</td>
<td>0.380 (0.176)</td>
<td>0.141 (0.133)</td>
<td>53.00</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>0.403 (0.263)</td>
<td>0.239 (0.212)</td>
<td>44.09</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>0.428 (0.429)</td>
<td>0.251 (0.321)</td>
<td>41.35</td>
</tr>
<tr>
<td>top 1.3</td>
<td>0.403 (0.386)</td>
<td>0.259 (0.286)</td>
<td>35.41</td>
</tr>
<tr>
<td>AVERAGE (%)</td>
<td>0.382 (0.298)</td>
<td>0.240 (0.258)</td>
<td>39.49</td>
</tr>
</tbody>
</table>

The refactoring of Call Super instances reduces the concrete overridings of the Derived-Class (NMO metric decrease) and, thus, has a positive effect on its maintainability, as discussed in Section 1. The SIX metric depends on NMO and its value, obviously, decreases with the elimination of concrete overridings. The average decrease of the SIX value in the refactored subclasses of each project is presented in Table 5. The table includes measurements of the SIX metric for each benchmark project\(^3\), prior and after refactoring the Call Super instances. Column SIX_\_p provides for each project the average metric value, prior to refactoring, for all classes that include refactoring candidates. Column SIX_\_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 decrease of the SIX metric after refactoring that ranges from 15.53% to 53.27% (39.49% on average). This fluctuation depends mainly on the DIT of the refactored classes in each project and the fraction of their concrete overridings that are eliminated.

A weakness of the SIX metric is that its value does not provide a scaled indicator on the appropriate use of inheritance and method overriding on a given subclass. In other words, two classes cannot be compared reliably, in terms of their safe use of inheritance, on the basis of their SIX value. In this study, we do not give prominence to the absolute SIX value of a refactored class, but instead we employ the percent reduction of SIX as

---

\(^3\)Estimated with the use of the Eclipse Metrics plugin that can be downloaded from [https://github.com/bzafiris/eclipse-metrics-plugin](https://github.com/bzafiris/eclipse-metrics-plugin).
a hint for the identification of higher impact refactorings. Specifically, a high percent
decrease of the SIX metric in a refactored class, denotes either (a) a high DIT, (b) the
elimination of several of its concrete method overridings or (c) a combination of them.
Projects of Table 5 that are characterized by a high average decrease of the SIX metric
include, proportionally, more high impact refactorings among their refactored classes. For
instance, in jade project, 9 out of the 26 refactored classes have all their concrete method
overridings eliminated, while 4 of them are characterized by DIT≥ 4. As concerning the
ant project, 8 out of 14 refactored classes have DIT≥ 4, while 4 of them are characterized
by total elimination of their concrete method overridings.

Despite the positive effect of a refactoring on software metrics, its final approval and
application to the source code is up to the programmer’s judgment. For this reason,
the proposed method has been implemented as an interactive tool that allows the pro-
grammer to preview and approve or reject each individual refactoring. The alignment
of refactored code with the programmer’s intent at the implementation of the Over-
ridingMethod contributes to easier approval of a refactoring suggestion. Such alignment
is possible if the extracted BeforeMethod and AfterMethod can be renamed to appropri-
ate intent revealing names that reflect the semantics of the problem domain. For instance,
Figure 8 presents a refactoring candidate in method DerbyPersistenceManager.init
(jackrabbit-core project) where BeforeFragment and AfterFragment provide targeted
extensions to the super-class behaviour. Notice that source code comments highlight the
purpose of the statements surrounding the SuperInvocation. It is clear for the program-

Figure 8: Suggested refactoring to Template Method in DerbyPersistenceManager class of the
jackrabbit-core 2.9 project.

```java
BundleDbPersistenceManager
+ void init(PCMcontext context) throws Exception ...

OraclePersistenceManager
+ void init(PCMcontext context) throws Exception ...

PostgreSQLPersistenceManager
+ void init(PCMcontext context) throws Exception ...

MySqlPersistenceManager
+ void init(PCMcontext context) throws Exception ...

H2PersistenceManager
+ void init(PCMcontext context) throws Exception ...

DerbyPersistenceManager
@Override
public void init(PCMcontext context) throws Exception {
  // init default values
  if (getDriver() == null) {
    setDriver(DERBY_EMBEDDED_DRIVER);
  }
  if (getDatabaseType() == null) {
    setDatabaseType("derby");
  }
  if (getUti) == null {
    setUti("jdbc:derby:
      + context.getHomeDir()+getPath()
      + "db/itemState:create=true");
  }
  if (getSchemaObjectPrefix() == null) {
    setSchemaObjectPrefix("/");
  }
  super.init(context);
  // ... four more conHelper.exec(.,.) statements
}
```

26
mer that the proposed refactoring can be accepted, since the new methods corresponding to BeforeFragment and AfterFragment can have intention revealing names. The application of the rename method refactoring on these methods and their assignment of intention revealing names, such as configureJdbcDriver for the BeforeFragment and setJdbcProperties for the AfterFragment, show the existence of appropriate hooks for effective subclassing of the BundleDbPersistenceManager class.

Table 6: Refactoring candidates with overridden method having object lifecycle semantics.

<table>
<thead>
<tr>
<th>Project name</th>
<th>Refactoring Candidates</th>
<th>Object Lifecycle Semantics</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom4j 2.0.0rc1</td>
<td>2</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>xstream 1.4.8</td>
<td>4</td>
<td>2</td>
<td>50.0%</td>
</tr>
<tr>
<td>myfaces-impl 2.1.9</td>
<td>8</td>
<td>5</td>
<td>62.5%</td>
</tr>
<tr>
<td>jackrabbit-core 2.9</td>
<td>27</td>
<td>20</td>
<td>74.1%</td>
</tr>
<tr>
<td>maven 2.9</td>
<td>16</td>
<td>6</td>
<td>37.5%</td>
</tr>
<tr>
<td>freenode 1.0.14</td>
<td>7</td>
<td>4</td>
<td>57.1%</td>
</tr>
<tr>
<td>apache ant 1.8.2</td>
<td>15</td>
<td>8</td>
<td>53.3%</td>
</tr>
<tr>
<td>code 4.4</td>
<td>37</td>
<td>20</td>
<td>54.1%</td>
</tr>
<tr>
<td>xerces 11.0</td>
<td>22</td>
<td>9</td>
<td>40.9%</td>
</tr>
<tr>
<td>xalan 2.7</td>
<td>24</td>
<td>5</td>
<td>20.8%</td>
</tr>
<tr>
<td>batik 1.7</td>
<td>27</td>
<td>8</td>
<td>29.6%</td>
</tr>
<tr>
<td>opj 1.1</td>
<td>71</td>
<td>18</td>
<td>25.4%</td>
</tr>
<tr>
<td>AVERAGE (%)</td>
<td>-</td>
<td>-</td>
<td>40.4%</td>
</tr>
</tbody>
</table>

A further inspection of suggested refactorings across benchmark projects revealed that the purpose of the respective overridden methods is, often, relevant to object initialization/cleanup and configuration. Pre- and post-processing hooks associated with such methods have clear semantics for the developer, relevant to an object’s lifecycle, and can be easily renamed to intention revealing names. Thus, the potential for approval of the respective refactorings by the developer is increased. Table 6 presents the results of the manual evaluation of suggested refactorings through code inspection. Column 2 presents the number of refactoring candidates for each project, while Columns 3–4, provide the fraction of them (count and respective percentage) having an overridden method with semantics relevant to object lifecycle (initialization/cleanup, configuration). The results provide for each project an approximation of the minimum number of refactorings that can be approved by the developer due to alignment of the refactored code with his/her intent on the original version. These refactorings range from 0% to 74.1% (40.4% on average) of refactoring candidates. Notice that relevance to the developer of the rest of refactoring candidates, not related with object lifecycle operations, required high familiarity with the projects’ source code in order to be reliably evaluated.

As concerning the runtime efficiency of the identification algorithm implementation, it has been evaluated by measuring its execution time on the benchmark projects. The measurements have been collected through running the implemented Eclipse plugin on a workstation equipped with a quad core 3.1GHz processor and 8 GB of RAM. Runtime performance results support the scalability of the approach, since the execution of the identification algorithm requires 7–25s for small to medium size projects (<100 KLOC) and does not exceed 1min for large ones (>100 KLOC).
4.4. Threats to validity

This section discusses potential threats to the validity of empirical evaluation results. Threats to validity in applied research are categorized and prioritized (in descending order) as follows: internal, external, construct and conclusion [47].

Threats to internal validity refer to factors that may influence the results of the empirical evaluation and are, either, ignored or cannot be controlled. A possible threat is relevant to the quality of the benchmark projects’ test suites that may affect the conclusion of RQ3, i.e. that the proposed refactoring procedure preserves the external behaviour of refactored code. The availability of test suites that thoroughly test the refactored code strengthen the validity of this conclusion. Since the empirical evaluation involved relatively large code bases (17KLOC-180KLOC), non familiar to the authors, the extent of testing of the refactored code could, practically, be assessed only through code coverage information. The benchmark projects’ test suites covered 64% of the refactorings included in Table 3 after being applied by the JDeodorant plugin. The correctness of non-covered refactorings was verified through code inspection.

External validity threats affect the potential for generalization of empirical evaluation results. Given that the results of the empirical study depend on the analysis of 12 open source projects written in Java, it cannot be safely asserted that they can be generalized for any other project, potentially written in a different programming language. As part of our future work, we intend to study the validity of our conclusions in a broader scope through analyzing projects written in other programming languages that support inheritance and super method invocation statements.

Threats to construct validity concern the correspondence between theory and the observation of the empirical evaluation. Possible threats to construct validity are relevant to the measurement of RQ1 and RQ2 results. In the case of RQ1, potential bugs in our extension of JDeodorant may impact the estimation of Call Super usage. However, small deviations from the actual Call Super instances on benchmark projects do not alter the conclusions of RQ1. On the other hand, potential bugs in the implementation of the refactoring identification algorithm will lead to underestimation of the effectiveness of our method, without affecting the conclusions of RQ2. The reason is that currently suggested refactoring candidates are validated through code inspection and evaluated as part of the study relevant to RQ3.

Finally, conclusion validity threats refer to the statistical significance of conclusions relevant to cause-effect relations studied in an experiment. In an exploratory study, where experimental measurements and observations shape the findings of the study rather than confirm the validity of certain hypotheses, reliability validity is the counterpart of conclusion validity [47]. Threats to reliability concern the reproducibility of the study. Our empirical evaluation represents an exploratory study and all details for its replication are publicly available.

5. Conclusions

We have proposed a method for automated refactoring to the Template Method design pattern of certain design flaws related to concrete method overriding. The focus is on a common code pattern encountered in class hierarchies, where an overriding method includes in its body an invocation to the overridden method through the super
keyword (super-invocation). This pattern is known as Call Super and usually denotes that overriding methods in sub-classes require the parent method’s behaviour, as it either implements core responsibilities or includes essential management functionality. Thus, negligence of the programmer to include the super invocation in overriding methods often leads to incorrect use of the parent class’s API.

Our approach applies the Template Method design pattern for the elimination of appropriate Call Super instances from a code base. We have introduced an algorithm for the discovery of refactoring opportunities that is based on an extensive set of refactoring preconditions. These preconditions ensure that the suggested refactorings can be safely applied to the source code. Moreover, we have specified the source code transformation procedure for refactoring a Call Super instance to the Template Method design pattern. An implementation of the proposed approach has been evaluated on a set of open source Java projects.

The evaluation results revealed the frequent occurrence of the Call Super code pattern that was present in 16.5%–76.5% of method overridings across the benchmark projects. The proposed refactoring identification algorithm suggested up to 20.5% of Call Super instances (depending on the project) as refactoring candidates (8.0% on average). Given that a great part of rejected refactorings correspond to trivial or semantically irrelevant cases, our method manages to eliminate a satisfactory number of Call Super instances. The application of the refactorings contributed to a decrease of the Specialization Index metric (SIX) in the affected subclasses that ranges from 15.53% to 53.27% (39.49% on average). Moreover, a manual evaluation of suggested refactorings revealed that 40.4% of them are characterized by an overridden method that serves a purpose relevant to object lifecycle (initialization/cleanup and configuration). Pre- and post-processing hooks associated with such methods have clear semantics for the developer and can be easily renamed to intention revealing names, contributing, thus, to easier approval of suggested refactorings. The successful execution of the benchmark projects’ test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Finally, runtime performance results support the scalability of the approach, since algorithm execution requires less 7–25s for small to medium size projects and does not exceed 1 min for large ones.

Our future work will extend the scope of this study to other programming languages that support inheritance and super invocation statements. Moreover, we will focus on the design of a software metric that overcomes the shortcomings of SIX in assessing the appropriate use of inheritance and the implications of method overriding. Finally, we plan to further study the correlation of Call Super and, generally, implementation inheritance with software bugs.

Acknowledgments

The authors would like to thank Prof. Tsantalis, Prof. Chatzigeorgiou, as well as the rest of the JDeodorant development team members for providing us access to the project’s source code. Last but not least, we would like to thank the anonymous reviewers for their useful comments that improved the quality of this work.
References


30


