Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State

JYOTHI VEDURADA, IIT Madras, India
V. KRISHNA NANDIVADA, IIT Madras, India

Refactoring is a program transformation that restructures existing code without altering its behaviour and is a key practice in popular software design movements, such as Agile. Identification of potential refactoring opportunities is an important step in the refactoring process. In large systems, manual identification of useful refactoring opportunities requires a lot of effort and time. Hence, there is a need for automatic identification of refactoring opportunities. However, this problem has not been addressed well for many non-trivial refactorings. Two such non-trivial, yet popular refactorings are “Replace Type Code with Subclass" (SC) and “Replace Type Code with State" (ST) refactorings. In this paper, we present new approaches to identify SC and ST refactoring opportunities.

Our proposed approach is based around the notion of *control-fields*. A control-field is a field of a class that exposes the different underlying behaviors of the class. Each control-field can lead to a possible SC/ST refactoring of the associated/interacting classes. We first present a formal definition of control-fields and then present algorithms to identify and prune them; each of these pruned control-fields represents a refactoring opportunity. Further, we present a novel flow- and context-sensitive analysis to classify each of these refactoring opportunities into one of the SC and ST opportunities. We have implemented our proposed approach in a tool called Auto-SCST, and demonstrated its effectiveness by evaluating it against eight open-source Java applications.

CCS Concepts: • **Software and its engineering** → **Software reverse engineering**; *Automated static analysis*; *Software maintenance tools*; *Design patterns*; • **Theory of computation** → *Program analysis*;

Additional Key Words and Phrases: Refactoring, Replace Type Code with State, Replace Type Code with Subclass, Replace Conditionals with Polymorphism, Static Program Analysis, Points-to Analysis

ACM Reference Format:
Jyothi Vedurada and V. Krishna Nandivada. 2018. Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 138 (November 2018), 28 pages. https://doi.org/10.1145/3276508

1 INTRODUCTION

Object-oriented programming languages provide mechanisms supporting polymorphism to avoid explicit checking of object properties for executing various actions. Though many developers are familiar with the concepts of polymorphism, it is quite common to find explicit state checking code in large applications. This is because, even though many large projects start with a good design, as the projects evolve over time and new requirements are introduced into the system, developers/maintainers often take the easier route of adding pieces of conditionally executed code to address each newer design requirement. This leads to an increase in the complexity of the code.

Authors' addresses: Jyothi Vedurada, Department of CSE, IIT Madras, Chennai, India, vjyothi@cse.iitm.ac.in; V. Krishna Nandivada, Department of CSE, IIT Madras, Chennai, India, nvk@iitm.ac.in.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
© 2018 Association for Computing Machinery.
2475-1421/2018/11-ART138
https://doi.org/10.1145/3276508

```java
public class XYSeries extends Series {
 private boolean autoSort;     ...
 public XYSeries(..., boolean as, ...)
 { ... this.autoSort = as; ... }
 public void add(XYDataItem item,...)
 { ...
   if (this.autoSort){
    int index = Collections.binarySearch
         (data,...);
    if (index<0)
      {data.add(-index-1, item);}
    else { ...
      /* handle duplicates. */ }
   } else {.../* add unsorted. */ }
   ...
 }
 public int indexOf(Number x) {
 if (this.autoSort){
    return Collections.binarySearch(data
         ,...);
  } else {
    /* do linear search */ ...
    return ...;}
 } }
```

(a) Code snippet from the jfreechart-1.0.14 project.

```java
public class Set {
 private boolean delegating = false; ...
 public int size() {
  if (this.delegating)
     return delegate.size();
  else
     return oneState == null ? 0 :1;
 }
 public boolean add(StateCache.State ns)
 { ...
  if (this.delegating){
          return delegate.add(ns); }
  else {
    if (oneState==null) {
       oneState = ns; return false;
    }
    if (ns == oneState) return true;
    beginDelegation();
    return delegate.add(ns);
  }
 }
 private void beginDelegation(){...
    this.delegating = true;...
 } }
```

(b) Code snippet from the avrora-1.7.106 project.

Fig. 1. Motivating Examples. The boxed entries indicate the control-fields.

The design of such code can be improved through refactoring [Fowler 1999], which restructures existing code without altering its behavior.

A popular and highly recommended way [Kannangara and Wijayanayake 2014] to refactor code with conditional-state-checking statements (like the ones discussed above) is by using the "replace conditional with polymorphism" (RCP) refactoring [Fowler 1999; Opdyke 1992; Vedurada and Nandivada 2017]. The popularity of RCP refactoring can also be seen from the web link [Tsantalis 2018], which shows that programmers often perform polymorphism (RCP) refactoring.

Effective RCP refactoring depends on systematically performing two important [Vedurada and Nandivada 2017] refactorings to build the required class hierarchy: i) replace type code with subclasses (SC), and ii) replace type code with state (ST)[1]. One of the key challenges in doing systematic SC/ST refactoring is that of identification of the refactoring opportunities. This challenge becomes more tedious and time-consuming in the context of refactoring of large applications that consist of hundreds of classes and fields. We illustrate some of these challenges using two motivating examples.

Fig. 1a shows a snippet of code from the class `XYSeries` taken from the jfreechart [Sourceforge 2016] application. The `autoSort` field of the class controls how the collection object `data` stores the

[1] Considering the similarities between the state and strategy refactorings [Fowler 1999], in this paper we do not differentiate between the state and strategy refactoring opportunities. See Section 4 for a discussion on this topic.

elements (sorted or unsorted). Based on the value of this field, the behavior of the XYSeries object varies; see the code of add and indexOf functions, for example. That is, implicitly, an XYSeries object can be in two states: autoSorting and nonAutoSorting, and the If-statements at Lines 7 and 17 are used to control the execution of the state-specific code. Hereafter, we refer to If and Switch statements as conditional-statements.

As a second example, Fig. 1b shows a snippet of code from the class Set taken from the avrora [Blackburn et al. 2006] application. The add method has an If statement switching on field delegating. Seven of the eight methods present in the class have similar code. Thus, the behavior of an object of type Set varies depending on the value of the delegating field. In other words, a Set object may be present in two states: delegating or nonDelegating and the conditional-statements at Lines 4 and 11 are used to control the execution of the state-specific codes.

Codes such as the ones shown in Fig. 1 can be made more readable by performing SC/ST refactoring, followed by RCP refactoring [Fowler 1999]. The first step in this process is to identify the opportunities to perform SC/ST refactoring that lead to RCP refactoring (hereafter, just referred to as SC/ST refactoring opportunities). There are two important challenges in this step as discussed in a recent work [Vedurada and Nandivada 2017]: (i) In large code bases, like jfreechart (513 classes) and avrora (1,746 classes), identifying SC/ST refactoring opportunities, manually or via scripts [Kim et al. 2015], is a non-trivial task; (ii) Further, even after an SC/ST refactoring opportunity is identified, it is still hard to choose the best-suited refactoring between SC and ST. Fowler [1999] suggests that if the observable state of an object changes during its lifetime, we should choose the ST refactoring, else it should be SC. For example, in Fig. 1a, the field autoSort is defined only in the constructor at Line 4 and that single value of the field autoSort is used throughout the life cycle of any object of type XYSeries. That is, the state of the object does not change and this makes it a candidate for SC refactoring. However, in Fig. 1b, different values of the field delegating may be used during the life time of an object Set – updated in beginDelegation method called from the add method. Thus, it is a candidate for ST refactoring. Performing such an analysis manually in large code bases, (for example, avrora with 4836 fields in 1731 classes spread over 100K lines of code), can be a daunting task.

Considering the challenges in manually identifying the opportunities for SC/ST refactoring, there have been attempts towards building tools that automatically identify such opportunities. Tsantalis and Chatzigeorgiou [2010] identify individual conditional-statements that can be replaced by polymorphic calls; consequently, they do identify a few of the SC/ST refactoring opportunities. For example, they identify SC-refactoring opportunities only in conditional-statements that check runtime-type-information, via instanceof checks. Thus, they cannot identify Fig. 1a as an opportunity to perform SC-refactoring. Similarly, they cannot infer ST-refactoring opportunity in either of the examples shown in Fig. 1, as they need static-final fields (named constants) in the conditional-statements to be able to identify ST-refactoring opportunities.

Christopoulou et al. [2012] propose an approach to identify individual conditional-statements that can be refactored using 'replace type code with Strategy' pattern (has similarities to ST refactoring). Naturally, their approach does not identify occurrences of SC-refactoring opportunities (for example, Fig. 1a). But it identifies Fig. 1a for strategy refactoring. It does not identify the opportunity in Fig. 1b – not a strategy refactoring opportunity. Similarly, if the instance field is stored inside a local variable, and the variable is used in a conditional-statement (for example, the code shown in Fig. 4), their approach fails to identify such opportunities. This motivates us to design dedicated schemes to identify SC/ST refactoring opportunities.

In this paper, we present a novel approach to identify and classify SC/ST refactoring opportunities effectively. Our identification phase is based around an innovative scheme to identify the fields (termed as *control-fields*) that expose the different underlying behaviors of the corresponding classes;

see the $\boxed{\text{boxed}}$ entries in Fig. 1, for example. Identifying control-fields from the numerous fields that are declared across a large number of classes, and used in many conditional-statements is a non-trivial challenge.

Further, classifying the identified refactoring opportunities (each corresponding to a unique control-field) into one of the SC and ST categories becomes quite challenging in large code bases, as one has to analyze the whole code base to establish if the observable state of an object changes during its lifetime. Note that, a completely syntactic approach is not sufficient for establishing this property. For example, conservatively considering only final fields for SC-refactoring will miss marking the opportunity in Fig. 1a for SC-refactoring because the field `autoSort` is not declared final. To track the state changes in the life cycle of an object, we need an analysis which uses points-to information and which takes into consideration the paths across different function calls. In addition, using context-insensitive points-to information is not sufficient to establish the above property. For example, if the constructor `XYSeries()` in Fig. 1a is called from different calling-contexts, the objects that the variable 'this' points-to, are not differentiated by a context-insensitive analysis and hence the field `autoSort` of each object is assumed to get different values corresponding to different states, during the lifetime of the object. Consequently, the analysis will miss marking this code for SC-refactoring. We address these challenges, by presenting a novel flow- and context-sensitive algorithm to perform this classification. Similar to other RCP-refactoring related tools [Christopoulou et al. 2012; Tsantalis and Chatzigeorgiou 2010] our proposed approach also requires the whole-program including the client-codes as input.

Our approach can precisely identify that Fig. 1a has an opportunity to apply SC-refactoring and Fig. 1b has an opportunity to apply ST-refactoring. Such a refactoring can lead to improved codes such as the ones shown in Fig. 2 and Fig. 3, where some conditional-statements are replaced with polymorphic calls; see Section 2 for details. To the best of our knowledge, no existing approaches or tools have addressed this problem of refactoring-opportunity identification effectively, for SC/ST refactoring. We have also implemented an RCP refactoring tool (invoked after the identification phase) to perform the actual code transformation. Note that since the main focus of this paper is to precisely identify SC and ST refactoring opportunities, the discussion about the actual transformation for replacing type codes with subclasses or state, along with the various difficulties involved therein are outside the scope of this paper.

Contributions

- We formally introduce the notion of control-field by building on the idea of type code [Fowler 1999]. The control-fields form the basis for detecting SC/ST refactoring opportunities.
- We present a systematic approach to precisely identify control-fields. Each control-field can be seen as a refactoring opportunity for the associated conditional-statements.
- We present a novel and efficient analysis to choose between SC and ST refactoring, and to create a precise inheritance structure for an identified refactoring opportunity. To the best of our knowledge, no such analysis is present in the literature to do such classification.
- We have implemented the proposed analysis in an Eclipse [2017] refactoring plug-in called Auto-SCST. We have demonstrated the effectiveness of Auto-SCST by evaluating it on eight open source Java applications.

2 BACKGROUND

We now briefly explain the SC, ST and RCP refactorings. Interested readers may refer to Fowler [1999] for a detailed discussion.

Replace Type Code with Subclasses (SC). This refactoring is performed as part of the RCP refactoring when we have an immutable type code which does not change its observable value throughout

```
class XYSeriesAutoSort extends XYSeries
{...
   public boolean getAutoSort ()
     {return true;}
   public int addItem(XYDataItem item){
     int index=Collections.binarySearch(
          data,...);
     if (index<0) {...} else {...} }
}
```

```
class XYSeriesNoAutoSort extends XYSeries
{...
    public boolean getAutoSort ()
      {return false;}
    public int addItem (XYDataItem item) {

         .../*add unsorted*/
      }
}
```

(a) Subclasses created for the class XYSeries.

```
public abstract class XYSeries extends Series{ ...
    public abstract boolean getAutoSort ();
    public abstract void addItem (XYDataItem item);
    public void add(XYDataItem item,...) {
      ... this.addItem(item); ...
    } }
```

(b) Modified original class.

```
    ... XYSeries s1 = new XYSeries(..., true, ...);/*three params*/
    ... XYSeries s2 = new XYSeries(..., false, ...);/*three params*/ ...
```

(c) Client code, before RCP refactoring.

```
    ...  XYSeries s1 = new XYSeriesAutoSort(..., ...); /*two params*/
    ...  XYSeries s2 = new XYSeriesNoAutoSort(..., ...); /*two params*/ ...
```

(d) Client code, after RCP refactoring.

Fig. 2. Example code to illustrate SC- and RCP-refactoring.

the life cycle of an object. The main steps for performing the SC refactoring on the example shown in Fig. 1a are: (i) self encapsulating the type code `autoSort`, by adding an abstract 'get' method, in the `XYSeries` class; and (ii) creating a subclass for each value of the type code, `XYSeriesAutoSort` and `XYSeriesNoAutoSort`, with concrete implementations for the 'get' method; see the 'get' methods in Fig. 2a.

Replace Type Code with State (ST). This refactoring is performed as part of RCP refactoring when the type code is mutable and/or when the class under consideration to refactor already has subclasses. The main steps for performing the ST refactoring on the example shown in Fig. 1b are: (i) self encapsulating the type code `delegating`; (ii) creating an abstract class `SetState` with a declared abstract method to get the value of the type code (see Fig. 3b); (iii) creating two subclasses of `SetState`: `SetDelegating` and `SetNonDelegating` (see Fig. 3a), for each value of the type code, with each class having a concrete implementation for the 'get' method, similar to the code in Fig. 2a (methods not shown in Fig. 3a for brevity); and (iv) replace the type code variable `delegating` with a state variable `state` and adjust the 'set' method (to set the variable `state` to the appropriate instance of the subclass) and 'get' method (to invoke the `state.getDelegating` method); see Fig. 3c.

```
class SetDelegating extends SetState
{  ...
 public int size(Set set)
  { return set.getDelegate().size(); }
}
```

```
class SetNonDelegating extends SetState
{  ...
 public int size(Set set) {
  return set.getOneState()==null?0:1;
 } }
```

(a) Subclasses created for the class `SetState`.

```
public abstract class SetState {...
 public abstract boolean getDelegating();
 public abstract int size(Set set);
}
```

(b) Abstract state class.

```
 1 public class Set { ...
 2  private SetState state; /*state field*/
 3  public void setDelegating(boolean st){
 4    if (st)
 5      this.state = new SetDelegating();
 6    else
 7      this.state = new SetNonDelegating();
 8  }
 9  public boolean getDelegating() {
10    return state.getDelegating();
11  }
12  public int size(){
13    return this.state.size(this);/*rcp*/
14  }
15  private void beginDelegation(){ ...
16   this.setDelegating(true); ...
17  } }
```

(c) Modified original class.

Fig. 3. Example code to illustrate ST- and RCP-refactoring.

Replace Conditionals with Polymorphism (RCP). This refactoring is used to simplify conditional statements. It starts with choosing between SC and ST to create the inheritance structure, followed by a series of refactorings like extract method, move method, and so on. The main steps for performing the RCP refactoring on the method add shown in Fig. 1a are: (i) If `XYSeriesAutoSort` and `XYSeriesNoAutoSort` are the two subclasses created for the class `XYSeries` for performing SC refactoring, then new overridden concrete methods are created in the subclasses by extracting the bodies of each branch of the associated conditional-statements; for example, see `addItem` methods in Fig. 2a. (ii) An `abstract` declaration of `addItem` is inserted in the baseclass `XYSeries`. See Fig. 2b. (iii) Finally, a call to the overridden method replaces the conditional-statement; see Line 5 in Fig. 2b.

In case of RCP-refactoring based on ST-refactoring, there is one main difference in step (ii): the abstract method declaration is added to the abstract class (for example, `SetState`) created as part of the ST-refactoring step. Fig. 3 shows the changes due to RCP refactoring on the method `size` shown in Fig. 1b; see the definitions of the method `size` therein.

Changes in the client code: For both SC and ST refactoring, explicit accesses of type-codes in the client code are replaced with calls to 'get' and 'set' methods (for example, see Line 16 in Fig. 3c). In addition, for SC-refactoring, the instantiations of the base class in the client code need to be changed to the instantiations of specific subclasses. For the code in Fig. 1a, snippets of the client codes before and after SC-refactoring are shown in Fig. 2c and Fig. 2d, respectively. In Fig. 2d, the static types of the variables `s1` and `s2` do not change, but the variables point to instantiations of the subclasses. Note that the changing of instantiations to specific subclasses can also be done via a factory method. Thus, RCP-refactoring may lead to changes in the base-class, as well as the client code and hence requires the whole-program to be available.

If the refactoring steps discussed in this section are performed manually, one needs to compile and test the code after each step, which can be avoided if done by a refactoring tool [Fowler 1999].

A key property of the values of the type codes: Consider an if-statement of the form `if (e1) S1; else if (e2) S2; ... else Sn` and say the expressions `e1`, `e2`, and so on, include a sub-expression involving a type code (say, `f1`). As discussed by Fowler [1999], an object executes different conditional bodies (such as `S1, S2`, and so on) when present in different states; each state corresponds to one or more non-overlapping control-values. Thus, the mapping from the conditional bodies to the values of their associated type codes forms a surjective function. We now use a simple example to show the importance of the *surjective property*.

Consider two conditional-statements '`if (a.x > 4) S1; else if (a.x < 4) S2`' and '`if (a.x > 2) S3`'. There are overlapping ranges of control-values across multiple control-expressions of these conditional-statements. For example, the ranges of control-values corresponding to `a.x < 4` and `a.x > 2` are overlapping and have a common control-value (the number 3). Now, if we create three sub-classes: `C1` for `a.x > 4`, `C2` for `a.x < 4`, and `C3` for `a.x > 2`, then `S2` has to be refactored into `C2` and `C3`, and `S3` has to be refactored into all the three classes – both violations of the surjective property. By the definition of SC/ST refactoring, the conditional body of each extracted branch statement corresponds to a unique state. In the example conditional-statements, the code specific to one branch is getting associated to multiple control-values (or states), which is against the definition of SC/ST refactoring. Hence it is important for the state-checking code to satisfy the surjective property to be fit for SC/ST refactoring.

3 IDENTIFYING SC AND ST REFACTORING CANDIDATES

In this section, we describe our approach to identify potential opportunities for doing SC/ST refactoring. Our approach is based around the definition of *control-fields*. Intuitively, a control-field is a field of some class X, whose different values, termed as *control-values*, help identify the different underlying behaviors of the associated objects of type X. We first present a definition of control-fields and then proceed on to present novel algorithms to identify, prune and classify them.

We define control-fields using the concept of *control-expressions*. A control-expression is an expression used in the guard of any `If/Switch` statement and for the discussion in this paper, we define control-expression using the following grammar:

$$
\begin{array}{lll}
\langle\text{control-expression}\rangle & ::= & [\langle \textit{exp} \rangle \;\&\&]?\ \langle \textit{EQ-exp} \rangle\ [\&\&\ \langle \textit{exp} \rangle]?\ \mid\ !\langle \textit{EQ-exp} \rangle \\
\langle \textit{EQ-exp} \rangle & ::= & \langle \textit{FV} \rangle{==}\langle \textit{CONST} \rangle \mid \langle \textit{FV} \rangle.\text{equals}\langle \textit{CONST} \rangle \\
 & & \mid \langle \textit{FV} \rangle \text{ instanceof } \langle \textit{Type} \rangle \\
\langle \textit{FV} \rangle & ::= & \langle \textit{Field-Dereference} \rangle \mid \langle \textit{Variable} \rangle \\
\langle \textit{CONST} \rangle & ::= & \langle \textit{Constant} \rangle \mid \langle \textit{Final-Field} \rangle
\end{array}
$$

A control-expression can be a logical-AND expression with *EQ-exp* as a sub-expression, or a simple expression derived from *EQ-exp*, or !*EQ-exp*. The non-terminal ⟨*exp*⟩ derives any arbitrary Java expression. *EQ-exp* derives only simple expressions that consist of an equality comparison operation with constants or an `instanceof` comparison operation. Note that since we are interested in SC and ST refactorings, we only focus on the $=$ and $\neq$ operations and skip other arithmetic and logical operations. See Section 4, for a discussion on other possible forms of control-expressions.

Note that, in a control-expression, we do not allow !*EQ-exp* within a logical-AND expression, as otherwise the key property (see Section 2) may be violated. For example, say we want to choose the predicate in the following if-statement as a control-expression: '`if(o1.base != 2 && exp1){S1} else {S2}`', and the target control-field `base` may take the values 2 and 16. Such a choice may lead to `S1` and `S2` both to be associated with the control-value 16 (of `o1.base`). Using a similar argument, we do not allow control-expression to have a logical-OR expression with *EQ-exp* as a sub-expression.

```
public void receiveFrame(Frame frame) {
      int data = frame.value;
      if (mode) { // Instruction mode
          switch (data) {
              case CLEAR_SCREEN: ...
              case SCROLL_LEFT: ...
              case SCROLL_RIGHT: ...
          } ...
      } ...
}
```

Fig. 4. Control-field propagating via copy statements. Code snippet from the avrora-1.7.106 project.

The *CONST* non-terminal in the above grammar refers to literals or `final` fields of the following types: Java primitive types, String or Enum. Similar to the treatment of null-objects in Jdeodorant [Tsantalis and Chatzigeorgiou 2010], we do not include "`null`" as a possible *CONST*. This is because, typically `null` is used to check if a field has been allocated or not, and does not represent the state of the object. Further, the *Introduce Null Object* [Fowler 1999], is an independent refactoring [Gaitani et al. 2015] and is handled differently from RCP refactoring.

Note: we handle simple boolean expressions of the form `(FV)`, `(!FV)`, `(FV.getClass() == Type.getClass())`, and `(FV.getClass() != Type.getClass())`, by treating them like expressions of the form `(FV==true)`, `(FV==false)`, `(FV instanceof Type)`, and `!(FV instanceof Type)`, respectively.

We now use the above-presented grammar of the control-expressions as the basis to formally define control-fields, by building on the concept of type codes [Fowler 1999].

Definition 3.1. A field f of a class with fully-qualified name C is called a *control-field* (denoted as $\langle C, f \rangle$), iff

- CF-Property 1: it is directly present in *EQ-exp* of a control-expression, *or*
- CF-Property 2: its value is copied (by one or more copy statements) to a variable that is present in *EQ-exp* of a control-expression. Such a variable is called a *control-variable*.

We are interested in finding the control-fields because often one of these fields, or their associated control-variables are used by `If` and `Switch` statements to simulate polymorphism. Thus, a control-field $\langle C, f \rangle$ guides the behavior of objects of type C. For example, in Fig. 1a, the field `autoSort` of class `XYSeries` which is used in the control-expression at line 5, is an example of a control-field. Based on this observation, we can create two subclasses of `XYSeries`: `XYSeriesAutoSort` and `XYSeriesNoAutoSort` (see Fig. 2a). Similarly, ⟨`USART.Frame`, `value`⟩ in Fig. 4 (snippet from avrora [Blackburn et al. 2006]) is a control-field because it is copied to a control-variable `data` and `data` is used in the control-expression of `Switch`.

The idea of control-fields forms the basis for identifying the SC/ST refactoring opportunities in our proposed approach. We have implemented our proposed approach as an Eclipse plug-in called Auto-SCST. Fig. 5 outlines the block diagram of Auto-SCST, which consists of two main parts : Auto-SCST-I and Auto-SCST-R. Auto-SCST-I takes as input a Java application and returns a list of SC/ST refactoring opportunities along with a proposed new inheritance structure for each such opportunity. The developer can use this information to pick and perform the actual refactoring (using Auto-SCST-R or manually). We now describe each of the three components of Auto-SCST-I and follow it up with a brief discussion of Auto-SCST-R.

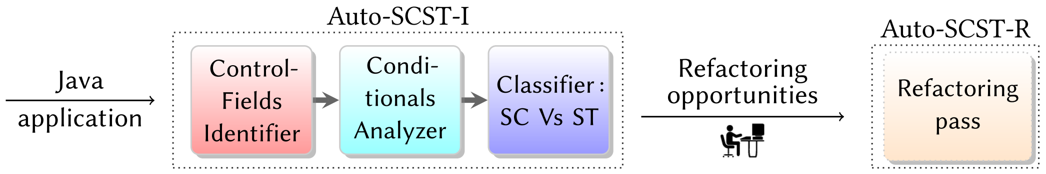


Fig. 5. Block diagram of the working scheme of Auto-SCST.

```
Procedure computePotentialControlFields($m$)
    foreach $s \in$ conditional-statements of $m$ do
        foreach $i \in$ conditional-points of $s$ do
            $CF_i := CF_i \cup$ PFInfo($i$);    // $CF_i$ = potential control-fields in $i$, satisfying CF-Property 1
            foreach $v \in Vars(i)$ do // Local variables
                $CF_{iv}$ := computeRelevantFields($m.G, i, v$);    // $m.G$ is the CFG of $m$
                if $|CF_{iv}| = 1$ then
                    $CF_i := CF_i \cup CF_{iv}$;    // $CF_{iv}$ satisfies CF-Property 2
```

Fig. 6. Algorithm to detect potential control-fields.

3.1 Identifying Control Fields

We now present a systematic approach to identify the *potential* control-fields and their associated `If/Switch` statements that simulate polymorphism. The associated `If/Switch` statements are the potential conditional-statements to get replaced with a polymorphic call. Thus, each of the identified control-fields represents an SC/ST refactoring opportunity. The potential control-fields and the associated conditional-statements found in this step are further pruned in the next step (Section 3.2) to get the final set of candidate control-fields and conditional-statements, which are then classified (in Section 3.3) into SC and ST opportunities, and presented to the developer for performing the actual refactoring.

We now present an algorithm (Fig. 6) to identify the potential control-fields and their associated conditional-statements. We do this by focusing on the two discussed properties of control-fields (see Definition 3.1). We first construct the control flow graph (CFG) for each method; for the ease of presentation, we consider each statement as a basic block node. We define the set of guard expressions of a conditional-statement as *conditional-points*: (1) for an `If` statement of the form '`if(`e_1`)` s_1 `else if(`e_2`)` s_2 `...` `else` s_n', the set of conditional-points is given by $\{e_1, e_2, \ldots\}$; (2) for a `Switch` statement of the form '`switch (e) S`' the set of conditional-points is given by $\{e\}$. To compute the set of potential control-fields, we perform a backward analysis starting from the conditional-points of each conditional-statement.

The procedure `computePotentialControlFields` iterates over the conditional-points of each method. The procedure first adds the potential control-fields (returned by `PFInfo`(i)) that are used in the conditional-point i to the set CF_i (Line 4). These fields potentially satisfy CF-Property 1 (Definition 3.1), as they directly participate in a predicate of a conditional-statement. The function `PFInfo` (code omitted for brevity) takes an expression e as input, and first finds all the sub-expressions connected by logical, arithmetic, or relational operators. For each such sub-expression e_s of e, `PFInfo` adds a control-field $\langle C, f \rangle$ to the list of returned potential control-fields, if e_s is of the form (i) $e_x.f$, where e_x can be any arbitrary expression, and $typeOf(e_x) = C$, or (ii) e_x`.foo(`$\langle args \rangle$`)`, and if the set of all possible return expressions from all possible invocations of e_x.foo is given by the set $S = \{er_1, er_2, \cdots er_n\}$, and $\forall er_i \in S$, PFInfo(er_i) $= \{\langle C, f\rangle\}$. Note that if foo is a recursive method, then we conservatively set the PFInfo of each of its return expressions as the empty set. We have found that the above case (ii) has been effective in analyzing the 'get' methods prevalent in Java classes.

For each variable v used in the conditional-point i, computePotentialControlFields calls computeRelevantFields, by passing the CFG of m, along with i and v. computeRelevantFields (body not shown for brevity) uses an iterative worklist-based backward data-flow algorithm. It computes the potential control-fields that satisfy CF-Property 2 (see Definition 3.1) by checking if their value may flow to the variable v at conditional-point i via one or more 'sCopy' statements. We define an *sCopy* statement to be either a simple copy statement (of the form $x = var$); or an assignment statement $x = e_z$, where e_z is of the form $e_y.f$ or e_y.foo($\langle args\rangle$), and |PFInfo(e_z)| = 1. A brief outline of the function computeRelevantFields can be found in Appendix A.1.

Note that the function computeRelevantFields may return more than one potential control-field. For example, consider the synthetic code snippet: '`if (cond) { x = a.f; } else { x = b.f; } switch (x /*target conditional point*/ ) { ... }`'. Here, for the conditional-point x, the function computeRelevantFields returns a set with potential control-fields corresponding to both a.f and b.f. Here, if the static types of a and b are different, then the procedure computeRelevantFields will return a set with two elements. For the ease of refactoring, we include only those cases where there is exactly one potential control-field corresponding to any variable (Lines 7–8).

Example: By establishing CF-Property 1, the procedure computePotentialControlFields sets: (1) CF_7 and CF_{17} to $\{\langle$XYSeries, autoSort$\rangle\}$ for the code in Fig. 1a, and (2) CF_4 and CF_{11} to $\{\langle$Set, delegating$\rangle\}$ for the code in Fig. 1b.

Similarly, by establishing CF-Property 2, the procedure computePotentialControlFields sets CF_3 to $\{\langle$Frame, value$\rangle\}$ for the code in Fig. 4.

3.2 Analyzing Conditionals

In this step, we compute the final set of *candidate control-fields* and their associated *candidate conditional-statements* (SC/ST refactoring opportunities) by analyzing and pruning the set of potential control-fields and their associated conditional-statements, identified in the previous step. We analyze the If and Switch conditional-statements separately.

If conditional-statement. Consider an If conditional-statement S of the form: if(e_1) s_1; else if(e_2) s_2; ... else s_n;. We consider S to be a *candidate conditional-statement* if the set of conditional-points $E = \{e_1, e_2, \ldots, e_n\}$ has the following *candidate-expr properties*:

- Every expression $e_i \in E$ is a control-expression.
- If $CF_1, CF_2, \ldots, CF_n$ are the sets of control-fields associated with $e_1, e_2, \ldots, e_n$, respectively, then $CF_1 \cap CF_2 \cap \ldots \cap CF_n \neq \emptyset$. This ensures the presence of at least one common control-field in all branches of the conditional-statement.
- No $e_i \in E$ has any side-effects.

We use the potential control-fields computed in Section 3.1 to see if the corresponding conditional-points satisfy the candidate-expr properties. Besides identifying the candidate conditional-statements, we also populate a set CF_{cand} containing the corresponding set of candidate control-fields. In addition to this, during the analysis, we collect the constants against which the control-field $\langle C, f\rangle$ is compared in the guard expressions. We call these constants as *control-values*. The number of distinct control-values indicates the number of subclasses of C that will be created during refactoring.

Name	*Brief Description*
$m.G$	CFG (Control Flow Graph) of a method m.
CF_{cand}	Set of candidate control-fields.
η	For each CFG node n, $\eta(n)$ maps expressions to their points-to information at n.
μ	For each call-site node n in the CFG, $\mu(n)$ maps the temporary (outside) parameter objects of the target method to the actual argument objects at the call site n.
subClassCF	A control-field classified as subClassCF indicates an SC refactoring opportunity.
stateCF	A control-field classified as stateCF indicates an ST refactoring opportunity.
subClassCFSet	Set of control-fields classified for SC refactoring.
stateCFSet	Set of control-fields classified for ST refactoring.
$n.objectFieldDefs$	Set of object-field pairs of the form $\langle(o, f)\rangle$ that are explicitly defined in n.
$n.objectFieldUses$	Set of object-field pairs of the form $\langle(o, f)\rangle$ that are explicitly used in n.

Fig. 7. Variables/Names used in Section 3.3 along with their brief descriptions.

Switch conditional-statement. A `Switch` statement S of the form `switch(`e`) { case` c_1`:`s_1`; case` c_2`:`s_2`; case` c_3`:`s_3`; ... default:`s_n`; }` is considered to be a *candidate conditional-statement* (and added to S_{cand}), if the candidate-expr properties are satisfied for the set $\{e\}$. The control-values with respect to the control-field of S is given by the set of constants $\{c_1, c_2, \ldots, c_n\}$. Similar to the treatment for the `If` conditional-statements, besides identifying the candidate conditional-statements, we populate the set CF_{cand} to include the corresponding candidate control-fields.

Example: For the examples shown in Fig. 1a, Fig. 1b, and Fig. 4, the current step identifies all of the control-fields identified in the previous step as final candidate control-fields. That is, for jfreechart: $\{\langle\texttt{XYSeries}, \texttt{autoSort}\rangle\} \subseteq CF_{cand}$, and for avrora: $\{\langle\texttt{Set}, \texttt{delegating}\rangle, \langle\texttt{USART.Frame}, \texttt{value}\rangle\} \subseteq CF_{cand}$.

3.3 Classification: Subclass vs State Pattern

Once the candidate control-fields (CF_{cand}) are identified, our next step is to classify them into opportunities for performing SC and ST refactoring. We refer to these classes as subClassCF and stateCF, respectively. Based on the classification of a refactoring opportunity (referred to by the corresponding candidate control-field $\langle C, f\rangle$), the actual refactoring steps that are required can be decided using the standard procedures given by Fowler [1999]. For ease of reference, we list the names/variables that are used in this section along with their brief descriptions in Fig. 7.

Note that a naive classifier would classify each of the candidate control-fields as a stateCF and would still lead to a correct refactoring of the classes. But such a scheme would miss the SC refactoring opportunities – which otherwise may further improve the code quality (see Section 2). Inspired by the argument given by Fowler [1999], in this paper, we classify a control-field $\langle C, f\rangle$ as a stateCF, if at least one of the following conditions hold; otherwise, we classify it as a subClassCF.

- Classification-condition I: There exists an object o_1 of type C, such that multiple values of the field f of o_1 are used. It indicates that o_1 is accessed in multiple states, each denoted by a different value of f.
- Classification-condition II: Class C already has one or more subclasses: creating subclasses of the base class in the presence of existing subclasses may lead to a large explosion of subclasses.

Note that to establish Classification-condition I, it is not sufficient to simply establish a path from one definition of a field $o.f$ to another definition of $o.f$. Consider the code snippets shown in Fig. 8. Class `X` has a field `f1` which is defined in its constructor. A naive redefinition check claims that the field `f1` is redefined, because `f1` is defined once at Line 2 and again at Line 4 of class `X`,

```
1 class X {
2         int f1;
3         X() {
4          f1 = _;
5         } ...
6 }
```

```
1 class Y {
2         int f2;
3         m() {
4         switch(f2) { ... }
5         f2 = _; } ...
6 }
```

Fig. 8. Code snippets showing the definitions of control-fields.

and there is an execution path between these two lines. Consequently, this scheme may classify ⟨X, f1⟩ as a stateCF (and not as a subClassCF), but in reality, it is not true. Note that to correctly capture the redefinitions, we cannot ignore the field declarations as definitions even though they are present without any explicit initialization. For example, in class Y, if we do not consider the declaration of field f2 at Line 2 as a definition, then we may wrongly assume that the field f2 is not redefined, even though the default value of f2 is used at Line 4 and f2 is redefined at Line 5. To address this problem and to avoid naively classifying every refactoring opportunity as an ST opportunity, our proposed analysis checks the satisfiability of Classification-condition I by looking for a path from a use of a field to a definition of the same field for any given object (hereafter referred to as *write-after-read* property).

We now present a novel, modular, inter-procedural, flow- and context-sensitive analysis to establish Classification-condition I. We start by adding all the candidate control-fields to the set subClassCFSet that represents the elements of the class subClassCF, and initializing the set stateCFSet, which represents the elements of the class stateCF, to the empty set ($\emptyset$). For each $\langle C, f\rangle \in CF_{cand}$, our proposed algorithm (shown in Fig. 9) checks if Classification-condition I holds for $\langle C, f\rangle$, and if so, it moves $\langle C, f\rangle$ from subClassCFSet to stateCFSet. Establishing Classification-condition II is trivial and is done as a pre-pass, where control-fields satisfying Classification-condition II are moved from subClassCFSet to stateCFSet (details skipped for brevity).

Establishing Classification-condition I: We use an SCC (strongly connected component) call graph [Chatterjee et al. 1999] constructed using Class Hierarchy analysis (CHA) [Dean et al. 1995] for performing our static analysis. The procedure classifySCST (Fig. 9) takes as input a set of candidate control-fields and a list of SCC-nodes *SCC*. The procedure analyzes the input SCC-nodes in reverse topological order of the SCC call graph. The procedure analyzes each method *m* in an SCC-node and computes its summary. The procedure first invokes findPointsToSet function to compute the points-to information and then invokes the procedure performRedefCheck to verify Classification-condition I. In case of recursive methods, the procedure invokes findPointsToSet until fixed-point, followed by performRedefCheck until fixed-point. The fixed-point in both the cases is reached when there is no change in the summaries of the methods (handling the recursive methods is not shown in Fig. 9).

The procedure findPointsToSet (body is not shown) is an implementation of compositional pointer analysis [Salcianu 2006; Whaley and Rinard 1999] to compute the points-to information of each of the accessed variables. The invocation findPointsToSet(m) returns a pair (η, μ). The map η contains the points-to information at each statement in the function m. In the bottom-up points-to analysis for any method m', each parameter (and 'outside' nodes [Whaley and Rinard 1999]) is made to point to a special temporary object and the summary of m' is represented in terms of these temporary objects. In m, at each call-site n that calls m', μ (referred to as the temporary-objects

```
1  Procedure classifySCST($CF_{cand}$, $SCC$)
2      subClassCFSet := $CF_{cand}$; stateCFSet := $\emptyset$;                                  // global vars
3      foreach $m \in$ non-recursive nodes of SCC do  // in reverse topological order of the SCC call graph
4          $(\eta, \mu)$ := findPointsToSet($m$);
5          $m.summary.objectFieldDefs$ := $m.summary.objectFieldUses$ := $\emptyset$;
6          performRedefCheck($m, \mu, \eta$);                   // Establishes write-after-read property.

7  Procedure performRedefCheck($m, \mu, \eta$)
8      foreach $n \in m.G$ do                                                       // each node in CFG
9          updateObjectInfo($n, \mu, \eta$);
10         foreach $(o, f) \in n.objectFieldUses$ do
11             if $\langle o.type, f\rangle \in subClassCFSet$ then
12                 foreach $s \in m.G$ do $s.visit$ := false;
13                 foreach $n' \in succ(n)$ do
14                     if $\neg$dfsDefCheck($n', o, f, \eta, \mu, m.G$) then break;

15     foreach $n \in m.G$ do                                                       // build summary
16         $m.summary.objectFieldDefs \cup= n.objectFieldDefs$;
17         $m.summary.objectFieldUses \cup= n.objectFieldUses$;

18 Function boolean dfsDefCheck($n, o, f, \eta, \mu, G$)
       // Returns false, if further traversal is not required.
19     if $n.visit$ = true then return true;
20     $n.visit$ := false;
21     updateObjectInfo($n, \mu, \eta$);
22     foreach $(o', f') \in n.objectFieldDefs$ do
23         if $o'.id = o.id$ and $f = f'$ then
24             say $t = \langle o.type, f\rangle$;                                      // field $f$ of $o$ is redefined
25             subClassCFSet := subClassCFSet $- \{t\}$;
26             stateCFSet := stateCFSet $\cup \{t\}$;
27             return false;

28     foreach $n' \in succ(n)$ do
29         if $\neg$dfsDefCheck($n', o, f, \eta, \mu, G$) then break;
30     return true;
```

Fig. 9. Algorithm to classify control-fields to subClassCF and stateCF.

map) keeps a mapping of the temporary objects of m' to the actual parameter values (objects) at n. The maps η, μ, along with m, are passed to `performRedefCheck`.

For a control-field $\langle C, f\rangle$, the procedure `performRedefCheck` establishes Classification-condition I by checking the write-after-read property. For each node n in the CFG G of m, the function first invokes the procedure `updateObjectInfo` (Line 9) to get the *object-field* pairs (of the form (o, f), where f is a field of an object o) that are defined and used in n, given by $n.objectFieldDefs$ and $n.objectFieldUses$; we elaborate on the procedure `updateObjectInfo` later in this section. Then, it invokes `dfsDefCheck` for each entry $(o, f) \in n.objectFieldUses$, if $\langle o.type, f\rangle$ is a candidate control-field and is under consideration for being classified as a subClassCF. The function `dfsDefCheck` performs a *depth-first-traversal* on the CFG G starting from the input node n to check if the field

```
1  Procedure updateObjectInfo($n, \mu, \eta$)
2      if $n.isAnalyzed$ = true then return;
3      $n.isAnalyzed$ := true; $\mu_n := \mu(n)$; $\eta_n := \eta(n)$;
4      $n.objectFieldDefs$ := matchingPairs($\eta_n, n.def, CF_{cand}$);      // object-field pairs defined at $n$
5      $n.objectFieldUses$ := matchingPairs($\eta_n, n.use, CF_{cand}$);      // object-field pairs used at $n$
6      foreach $m' \in calleesAt(n)$ do
7          foreach $(o, f) \in m'.summary.objectFieldDefs$ do
8              foreach $o' \in \mu_n(o)$ do                                    // compose summary
9                  $n.objectFieldDefs := n.objectFieldDefs \cup \{(o', f)\}$;
10         foreach $(o, f) \in m'.summary.objectFieldUses$ do
11             foreach $o' \in \mu_n(o)$ do                                    // compose summary
12                 $n.objectFieldUses := n.objectFieldUses \cup \{(o', f)\}$;
```

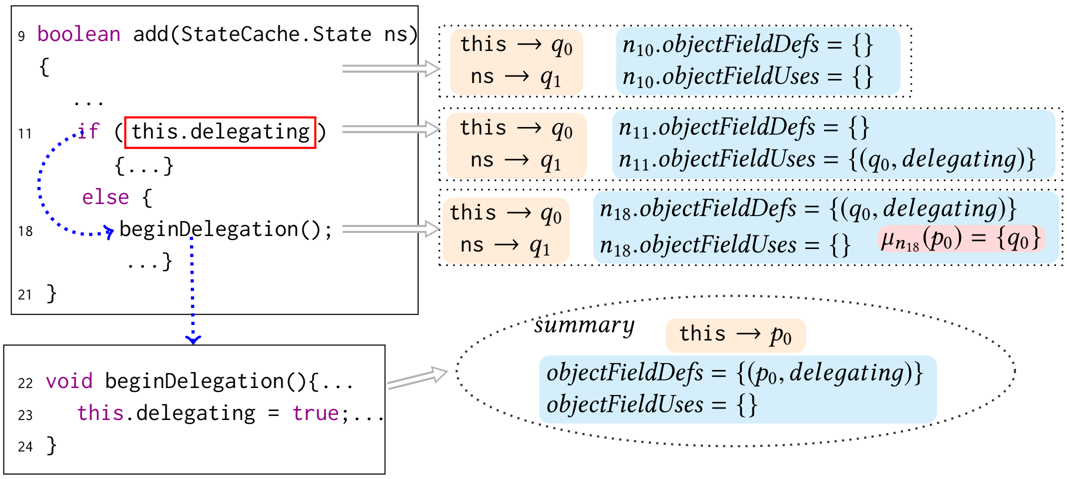
Fig. 10. Algorithm to populate object-field pairs.

f of the object o is defined at any of the successor nodes reachable from n in G. If such a node is found, it indicates that the write-after-read property is satisfied and then we move the control-field corresponding to (o, f) from the set subClassCFSet to the set stateCFSet (Line 25-26). After each node has been processed by `performRedefCheck`, it builds a summary for the current method by taking a union of the use/def information of all the nodes (Lines 15-17).

The procedure `updateObjectInfo` (in Fig. 10) takes as input the node n, the points-to map η, and μ. First, it initializes $n.objectFieldDefs$ and $n.objectFieldUses$ with the object-field pairs that are explicitly defined and used in n (Lines 4, 5) by invoking the function `matchingPairs`. The function `matchingPairs` takes as input η_n, E (set of expressions), CF_{cand} and includes an object-field pair (o, f) in the return value if the following conditions hold: (i) $\exists e.f \in E$, (ii) $o \in \eta_n(e)$, (iii) $typeOf(e) = C$, and (iv) $\exists\langle C, f\rangle \in CF_{cand}$. Second, the procedure `updateObjectInfo` finds the object-field pairs that are defined and used via function calls at n by composing the summaries of the callees. At Lines 6-12, for each procedure that is called at n, the summaries of the callees are composed. As discussed earlier, at each call-site n, the mapping of the temporary objects of a callee to the actual parameter values (objects) at n is maintained during the points-to analysis and is given by μ. During the summary composition, the procedure `updateObjectInfo` uses μ to convert the object-field pairs with temporary objects at callee to the object-field pairs with actual objects in the current context.

An interesting point of our `classifySCST` procedure (Fig. 9) is that calls to the functions `findPointsToSet` and `performRedefCheck` are interleaved. Consequently, after processing each method m, it is enough to keep the summary information about m (containing points-to information at the exit node of m, and information $m.summary.objectFieldDefs$ and $m.summary.objectFieldUses$) and discard the detailed points-to, *objectFieldDefs* and *objectFieldUses* information at each program point. This keeps our analysis scalable.

Example: In Fig. 1b, the opportunity $\langle Set, delegating\rangle$ cannot be an SC refactoring opportunity because there is a path from read/use of the field `delegating` at Line 11 in method add to the write/definition of the same field dereferenced with the same object pointed to by `this`, at Line 23 in method `beginDelegation` via the function call at Line 18. The presence of this path violates the Classification-condition I for $\langle Set, delegating\rangle$ and hence it is an ST refactoring opportunity.



```
9  boolean add(StateCache.State ns)
   {
     ...
11   if (this.delegating)
       {...}
     else {
18     beginDelegation();
       ...}
21 }
```

```
22 void beginDelegation(){...
23   this.delegating = true;...
24 }
```

this $\longrightarrow q_0$
ns $\longrightarrow q_1$
$n_{10}.objectFieldDefs = \{\}$
$n_{10}.objectFieldUses = \{\}$
this $\longrightarrow q_0$
ns $\longrightarrow q_1$
$n_{11}.objectFieldDefs = \{\}$
$n_{11}.objectFieldUses = \{(q_0, delegating)\}$
this $\longrightarrow q_0$
ns $\longrightarrow q_1$
$n_{18}.objectFieldDefs = \{(q_0, delegating)\}$
$n_{18}.objectFieldUses = \{\}$
$\mu_{n_{18}}(p_0) = \{q_0\}$
summary
this $\longrightarrow p_0$
$objectFieldDefs = \{(p_0, delegating)\}$
$objectFieldUses = \{\}$

Fig. 11. Execution of the algorithm in Fig. 9 on the code in Fig. 1b.

We now explain in steps, how our algorithm in Fig. 9 finds such a path in Fig. 1b. For the ease of reference, we have shown in Fig. 11, pieces of code from Fig. 1b that are of interest to the current example. In Fig. 11, next to each node n_i, we show the points-to graph at n_i (this is same as $\eta(n_i)$) and the sets of object-field pairs that are defined and used at n_i. In the `beginDelegation` method (in Fig. 1b), assuming p_0 is the temporary object used for the receiver (or `this` pointer), our analysis will update its *summary.objectFieldDefs* to include the pair (p_0, `delegating`). Similarly, in the `add` method, say q_0 and q_1 are the parameter nodes for the receiver and the field `ns`, respectively. When the method summary is composed at Line 18, (q_0, `delegating`) will be added to the set $n_{18}.objectFieldDefs$, because at n_{18}, $\mu_{n_{18}}(p_0) = \{q_0\}$. Since, ($q_0$, `delegating`) $\in n_{11}.objectFieldUses$ and there is a path (shown with dotted arrows in Fig. 11) from a read at n_{11} to a write at n_{18}, the algorithm classifies ⟨`Set`, `delegating`⟩ as a stateCF. However, for the examples shown in Fig. 1a and Fig. 4, the function `classifySCST()` does not find any such path from the reads of the control-fields to the writes (and hence classifies them as SC opportunities).

3.4 Automatic refactoring using Auto-SCST-R

The developer may choose to pass one or more of the SC/ST opportunities identified by Auto-SCST-I to Auto-SCST-R (see Fig. 5) for performing the actual refactoring. For each such opportunity, Auto-SCST-R invokes the refactoring procedure described by Fowler [1999] to perform the actual code transformation.

Inspired by the conditions used by JDeodorant [Tsantalis and Chatzigeorgiou 2010]) to identify refactoring opportunities suitable for automatic refactoring, and considering the complexities in performing the actual transformation, Auto-SCST-R does not consider an opportunity for automatic refactoring, if the branch bodies of the corresponding conditional-statements contain (not-so-frequent) patterns such as: (C1) assignments to more than one local variable, where the variables are live after the conditional-statement; (C2) branching statements such as `break` and `continue` associated with a loop outside the conditional-statement; (C3) super method invocations (with `super` keyword); (C4) `return` statement(s) only in some (not all) branches of the conditional-statement.

We now briefly describe the complexities involved in refactoring the code with patterns C1-C4. C1 requires to return more than one value from the extracted methods. C2 requires to change the control-flow in the outer-loop based on the events in the extracted methods. C3 requires the extracted method to invoke the '`super`' method of the original method. C4 requires the caller method to return based on the code of the invoked actual extracted method.

4 DISCUSSION

We now briefly elaborate some salient points of Auto-SCST.

Scalability. Inspired by the heuristics specified by Salcianu [2006] (to keep the bottom-up analysis scalable in the presence of large strongly-connected components) we have parameterized our implementation with the maximum size of the SCC ($Kmax$). If the size of an SCC exceeds $Kmax$, we abort the fixed-point computation for that SCC and analyze the intra-SCC methods in a standalone manner. For each such call, we conservatively assume that (i) the fields of the actual arguments may points-to the 'global-node' and (ii) each control-field associated with the arguments may be both used and defined in the call. In Section 5, we present our evaluation by setting $Kmax = 1$. We have carefully analyzed the results and interestingly found no loss of precision due to this heuristic, for any of the chosen applications.

Design decisions I/III. As discussed by Fowler [1999] the biggest gains in RCP refactoring occur when the same set of conditions occur at many places in the program. Hence, we only report the refactoring opportunities that have more than one associated conditional-statements.

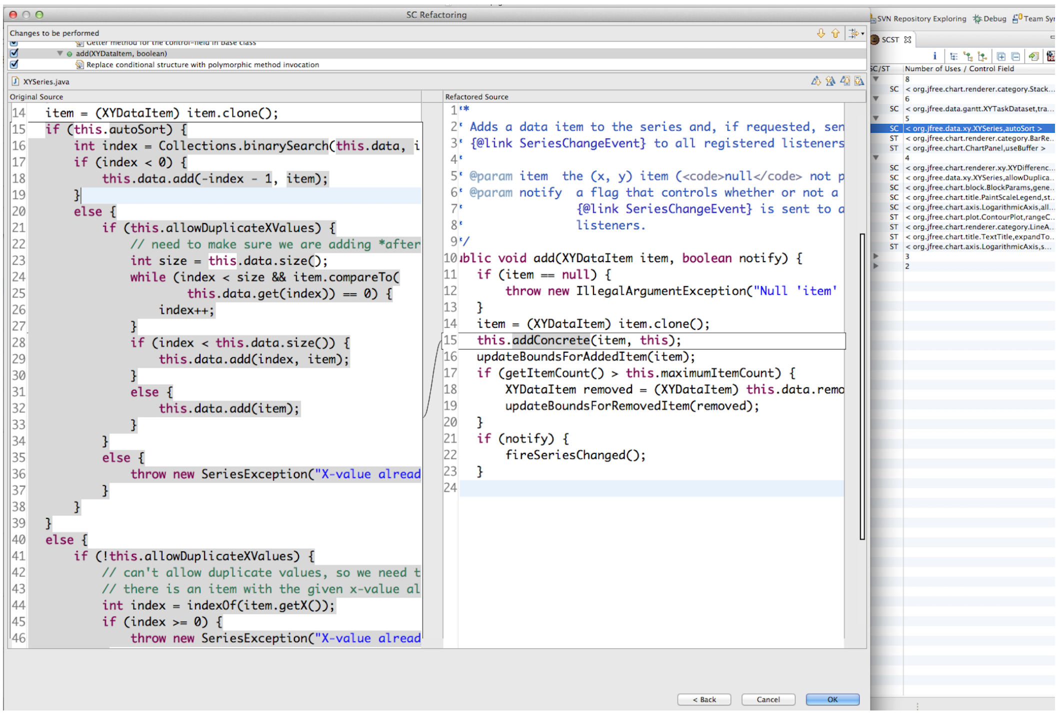
Design decisions II/III. Any `static` control-field that gets classified into the subClassCF, we change it to stateCF (as a post pass). Since the static fields may be modified anywhere in the program, refactoring the class, based on such a control-field, using SC seems unintuitive.

Design decisions III/III. In this paper, we do not handle expressions where the control-field is compared using operators like > and <. In such codes, each 'state' may correspond to one or more control-values. When the ranges of control-values of two states overlap, it violates the surjective property; see Section 2. In cases where the ranges do not overlap—checking requires elaborate range-analysis [Harrison 1977]—we can introduce a new type code in the class such that the type code has unique values for each range of the control-values and then we can invoke our tool on the modified class. We call such a two-step refactoring as *indirect* RCP refactoring; different from our focus of *direct* RCP refactoring.

SC/ST classification. Our classification algorithm shown in Figure 9 models both the call graph information and the points-to information in a conservative way, and concludes that Classification-condition I is not satisfied by a candidate control-field only after the procedure `performRedefCheck` has traversed all the conservatively reachable nodes in the CFG. Hence, the algorithm is conservative and would not miss any path which satisfies write-after-read property. In other words, a control-field of class stateCF is never incorrectly classified as subClassCF.

Improving the recall of Auto-SCST. We have found that the recall of Auto-SCST may get affected because of the following two reasons: (i) the control-field is compared against a field (constant nevertheless) from an unknown source (e.g., some libraries). (ii) the control-field is compared against another non-final field/variable whose value does not change. Providing Auto-SCST with the source code of the whole program, and using a helper tool to perform constant propagation will further improve the recall.

State Vs Strategy pattern. State pattern and Strategy pattern are very similar in structure although there are differences in the design problems they solve. State pattern is used when an object changes its state and exhibits different behaviours corresponding to each state. In contrast, strategy pattern is used when the object performs a task using different variants of an algorithm/approaches, depending on some condition. Despite these subtle differences, the overall refactoring approach for both the cases is the same [Fowler 1999]. Consequently, in our paper, we do not differentiate between the state and strategy pattern, towards the identification of RCP refactoring opportunities. Further, automatically differentiating strategy pattern from state pattern, in a precise manner



```
item = (XYDataItem) item.clone();
if (this.autoSort) {
    int index = Collections.binarySearch(this.data, i
    if (index < 0) {
        this.data.add(-index - 1, item);
    }
    else {
        if (this.allowDuplicateXValues) {
            // need to make sure we are adding *after
            int size = this.data.size();
            while (index < size && item.compareTo(
                    this.data.get(index)) == 0) {
                index++;
            }
            if (index < this.data.size()) {
                this.data.add(index, item);
            }
            else {
                this.data.add(item);
            }
        }
        else {
            throw new SeriesException("X-value alread
        }
    }
}
else {
    if (!this.allowDuplicateXValues) {
        // can't allow duplicate values, so we need t
        // there is an item with the given x-value al
        int index = indexOf(item.getX());
        if (index >= 0) {
            throw new SeriesException("X-value alread
```

```
**
* Adds a data item to the series and, if requested, sen
* {@link SeriesChangeEvent} to all registered listeners
*
* @param item  the (x, y) item (<code>null</code> not p
* @param notify  a flag that controls whether or not a
*                {@link SeriesChangeEvent} is sent to a
*                listeners.
*/
ıblic void add(XYDataItem item, boolean notify) {
   if (item == null) {
       throw new IllegalArgumentException("Null 'item'
   }
   item = (XYDataItem) item.clone();
   this.addConcrete(item, this);
   updateBoundsForAddedItem(item);
   if (getItemCount() > this.maximumItemCount) {
       XYDataItem removed = (XYDataItem) this.data.remo
       updateBoundsForRemovedItem(removed);
   }
   if (notify) {
       fireSeriesChanged();
   }
```

SC/ST	Number of Uses / Control Field
▼	8
SC	< org.jfree.chart.renderer.category.Stack...
▼	6
SC	< org.jfree.data.gantt.XYTaskDataset,tra...
▼	5
SC	< org.jfree.data.xy.XYSeries,autoSort >
ST	< org.jfree.chart.renderer.category.BarRe..
ST	< org.jfree.chart.ChartPanel,useBuffer >
▼	4
SC	< org.jfree.chart.renderer.xy.XYDifferenc...
SC	< org.jfree.data.xy.XYSeries,allowDuplica..
SC	< org.jfree.chart.block.BlockParams,gene..
SC	< org.jfree.chart.title.PaintScaleLegend,st..
SC	< org.jfree.chart.axis.LogarithmicAxis,all...
ST	< org.jfree.chart.plot.ContourPlot,rangeC..
ST	< org.jfree.chart.renderer.category.LineA...
ST	< org.jfree.chart.title.TextTitle,expandTo..
ST	< org.jfree.chart.axis.LogarithmicAxis,s...
▶	3
▶	2

Fig. 12. Screenshot of Auto-SCST plug-in in action.

is a very hard problem, as it would require checking for code equivalence (to identify if we are demarcating different behaviors of an object or different algorithms for performing a task).

5 IMPLEMENTATION AND EVALUATION

We have implemented Auto-SCST as an Eclipse source-to-source refactoring plug-in. It has two components: Auto-SCST-I (for identifying the refactoring opportunities using the techniques presented in Section 3) and Auto-SCST-R (a prototype for performing the actual RCP refactoring using the mechanisms discussed in Section 2). Auto-SCST works in the following manner: After the developer has invoked it on a chosen project (or a package, or a file), Auto-SCST first presents the set of identified refactoring opportunities to the developer. Once the developer selects a particular refactoring, the developer is given an option to change the names of the new classes to be generated. After that, a preview of the refactored code is shown to the developers and based on their input the actual refactoring is performed. Fig. 12 shows a sample screenshot of Auto-SCST in execution, invoked to refactor the jfreechart application; the screenshot shows the preview-screen while refactoring the `XYSeries` class (discussed Fig. 1a), for the control-field `autoSort`.

We have evaluated Auto-SCST on an Intel i5 2.9GHz machine with 8GB RAM, running macOS 10.13, using eight Java applications (see Fig. 13) taken from multiple sources, covering both stable and alpha/pre-alpha releases. Recently, Vedurada and Nandivada [2017] show that these applications span over varied domains and have multiple opportunities for RCP refactoring. The alpha and pre-alpha releases include javaGeom-0.10.2 (a geometrical computations library), jfreechart-1.0.14 (a Java chart library), jOcular-0.039 (an optical design software for simulating systems of lenses, prisms, and so on), RackJ-1.05 (a tool to analyze RNA-sequence data), Unicode-Rewriter-1.0 (UR, in

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Application	LoC	# classes			IF	CF	%CF	class		# Uses				MCF	%MCF	Atime
		IPC	QC	%QC				SC	ST	>15	11-15	6-10	2-5			(sec)
javaGeom	27K	105	5	4.8	199	5	2.5	4	1	0	0	2	3	4	80.0	5
jfreechart	204K	510	58	11.4	2845	100	3.5	34	66	2	2	5	91	88	88.0	26
jOcular	31K	219	16	7.3	791	17	2.1	13	4	0	0	2	15	13	76.5	36
RackJ	23K	102	14	13.7	694	17	2.5	5	12	0	0	2	15	14	82.4	71
UR	11K	41	5	12.2	294	8	2.7	3	5	0	0	1	7	7	87.5	1
avrora	100K	1731	66	3.8	4836	112	2.3	20	92	9	2	8	93	91	81.2	337
fop	162K	1125	91	8.1	4120	120	2.9	46	74	2	0	5	113	97	80.8	65
sunflow	25K	185	12	6.5	905	16	1.8	6	10	1	0	0	15	8	50.0	13

Fig. 13. Refactoring opportunities in different applications. Abbreviations used: Atime = analysis time; IPC = input classes; QC = classes qualified for refactoring, IF = input fields; CF = identified control-fields; MCF = marked control-fields.

short – a tool to convert ID3 tags to Unicode) from SourceForge [Sourceforge 2016]. Stable projects include avrora-1.7.106 (a simulator for running programs on a grid of micro-controllers), fop-0.95 (a print formatter), and sunflow-0.07.2 (ray-tracing based image rendering software) – all from the DaCapo [Blackburn et al. 2006] benchmark suite. Fig. 13 (columns 2-3, and 6) shows some static characteristics of these applications: 11K-204K lines of code (LoC), 41-1731 classes (IPC), and 199-4836 fields (IF).

For each of the applications, Fig. 13 (columns 4-5, 7-8) lists the number of classes qualified for refactoring (QC), %QC (= QC*100/IPC), number of identified control-fields (CF), and %CF (= CF*100/IF). As it can be seen, Auto-SCST is able to identify a number of SC/ST refactoring opportunities. Further, the set of refactoring opportunities identified by Vedurada and Nandivada [2017] is a subset of the ones identified by Auto-SCST. On an average, the refactoring opportunities, identified by Auto-SCST, cover 8.47% of the classes and 2.54% of the fields. Columns 9-10 show the statistics of classification into SC (between 3-46) and ST (between 2-92). Among the identified refactoring opportunities, the presence of up to 80% of SC opportunities confirms the importance of our proposed classification algorithm, in the absence of which those would have been classified to ST.

5.1 Effectiveness of Auto-SCST

We now present our evaluation of the effectiveness of Auto-SCST by answering four research questions.

5.1.1 What is the precision and recall of Auto-SCST? To reason about the precision of the identified opportunities empirically, we use the standard equation, $P = \frac{100 \times TP}{TP+FP}$, where TP (number of true positives) represents refactorable opportunities that are identified by Auto-SCST and FP (number of false positives) represents non-refactorable opportunities that are identified by Auto-SCST. We invoked Auto-SCST on each of the identified refactoring opportunities (counts are listed in column 7, Fig. 13) and we found that all of the opportunities identified by Auto-SCST are refactorable. That is, the number of refactorable opportunities identified by Auto-SCST (TP) is equal to the total number of opportunities identified by Auto-SCST ($TP + FP$); precision is 100%. We believe that the high value of precision is due to the properties of cond-statements (Section 3.2) and the grammar for control-expressions (Section 3).

We calculate the recall using the standard equation, $R = \frac{100 \times TP}{TP+FN}$, where TP (number of true positives) represents refactorable opportunities that are identified by Auto-SCST and FN (number of false negatives) represents the refactorable opportunities that are not identified by Auto-SCST.

That is, $TP+FN$ represents the total number of refactorable opportunities that are present in a given application. As discussed in Section 1, manual identification of SC/ST refactoring opportunities (to compute #total refactorable opportunities), in large (23K-204K LOC) unfamiliar applications is extremely challenging. Note that measuring the recall using a separate set of smaller applications (with a couple of thousands of lines) is not interesting because such small applications may not have many refactoring opportunities. Consequently, we believe that the conclusions drawn on these applications may not be applicable to the real-world large applications. Hence, we created a sample subset of 400 'potential' refactoring opportunities (to satisfy a margin of error of ±5% and a confidence interval of 95%) to evaluate the recall of the heuristics used by our identification approach[2].

We say that $\langle C, f \rangle$ is a 'potential' refactoring opportunity if the field f of an object of type C is used (either explicitly or via return statement(s) of a function call) in the guard expressions of at least two conditional-statements. Our definition of 'potential' opportunities for computing recall, is inspired by Fowler [1999], who states that the ground truth to identify an SC/ST refactoring opportunity is the presence of `If`/`Switch`-statements that switch on type codes (fields). Note that the ground truth from Fowler does not take into consideration 'refactorability' issues. As our recall computation is based on 'refactorable' opportunities, we first identify the potential (= possibly refactorable) opportunities, and then we manually analyze them for refactorability. We randomly selected 400 of the total 980 such potential opportunities (across all the chosen applications), to evaluate the recall. Among the 400 potential opportunities, we found that TP = 167. To compute the FN, we manually analyzed the remaining 223 potential opportunities to check if they can be refactored. We found that only 9 of those could be refactored; that is, FN = 9. Hence, recall = (100*167/ (167+9)) = 94.89%.

Summary: The high values of precision (100%) and recall (94.89%) attest to the relevance and effectiveness of Auto-SCST.

Discussion 1: We made the following two observations during the above analysis: (1) Using the extensions discussed in Section 4, the recall percentage can be further improved to 98.29% (by additionally identifying 6 of the above 9 false-negatives). The rest (three) fall into the category where the guard expressions in the conditional-statements refer to the control-fields as part of return expressions of functions (involving local variables/constants), deep inside nested calls. (2) The 214 opportunities cannot be refactored (even manually) due to the presence of one of the following: (i) complex expressions involving operators such as `||`, or with operators that result in the overlapping of the sub-ranges of the control-values (see Section 4), (ii) expressions with side-effects, (iii) expressions with library calls that take the candidate field as an argument, or (iv) the candidate field is not present in all the predicates of a conditional-statement.

Discussion 2: The definition of "potential-opportunity" is not complete; for example, it does not consider opportunities where the value of control-field is flowing via some local/array-element. However, to include such opportunities as well in the set of 'potential' opportunities, we may have to consider all the conditionals (on fields, locals, and array-elements, totaling 253,167) as expressions containing potential-opportunities and this may result in an unreliable recall-value.

5.1.2 How many of the opportunities identified by Auto-SCST are acceptable to experts? To understand which of the opportunities identified by Auto-SCST are accepted by the experts, we performed manual analysis (as done in other prior works [Christopoulou et al. 2012; Tsantalis and Chatzigeorgiou 2010]). We selected two experts, each having nearly 25 years of experience in large software development, to participate in the experiment. Both the experts had prior understanding

[2] Thanks to the anonymous reviewers of ICSE on a previous version of this manuscript for this suggestion on computing the recall.

Application	C_b	C_a	%Impr	Application	C_b	C_a	%Impr
javaGeom	3.62	1.95	46.05%	UR	9.0	5.89	34.57%
jfreechart	10.8	8.57	20.66%	avrora	11.12	5.02	54.88%
jOcular	6.55	4.42	32.47%	fop	9.01	6.88	23.66%
Rackj	17.08	14.83	13.18%	sunflow	7.76	5.24	32.49%

Fig. 14. Cyclomatic complexity before (C_b) and after (C_a) SC/ST refactoring, and the resulting improvements.

of object-oriented design and design patterns. The experts volunteered their services on our request, because of their passion for software engineering and their desire to learn about Auto-SCST, so that they can use it in their own projects.

Each of the experts was given independent time slots, spanning across multiple days. Before starting the experiment, we made a presentation of Auto-SCST to them individually.

Considering the complexities in understanding very large pieces of code, we requested the experts to only analyze applications (from Fig. 13) that have < 100K lines of code, namely jOcular, javaGeom, Rackj, UR and sunflow. Further, compared to the very large applications, the chosen applications had a fewer number of refactoring opportunities, thereby making it easier for the experts to reason about the opportunities. Each of the experts was given access to Auto-SCST, the chosen applications, and a pdf file containing the explanation and details (#uses, #subclasses after refactoring, SC/ST) of all the 63 identified opportunities. For each of these opportunities, the experts were asked the following two key questions: KQ_A: "Is it a candidate SC/ST refactoring opportunity?" KQ_B: "If the answer to KQ_A is 'Yes', then will you be open to performing the refactoring?"

Feedback from the experts E1 and E2. For KQ_A: we found that E1 and E2 have both marked 'Yes' for 57 opportunities each. Further, we found that of 52 opportunities (83%) were marked 'Yes' by both, 10 were marked 'Yes' by only one, and 1 was marked 'No' by both.

For KQ_B: we found that E1 and E2 have marked 'Yes' for 40 and 49 opportunities, respectively. Analyzing further, we found that of the 63 opportunities 33 (52%) were marked 'Yes' by both, 24 were marked 'Yes' by only one expert, and 6 were marked 'No' by both.

Summary: 52% of the identified opportunities were accepted by both the experts for refactoring and 91% by at least one. This shows the effectiveness of Auto-SCST.

Discussion: We now list some of the reasons provided by the experts for their negative answers to the key questions. (i) The identified opportunity involves debugging code, controlled by a 'debug' flag, leading to a subclass with methods containing only debug statements. KQ_B: (i) The identified opportunity has few (two/three) associated conditional-statements and those conditional-statements have a very small average body size; (ii) Most of the conditional-statements associated with the identified opportunity perform state-checking for only one state (subclasses in the refactored code may contain methods with no statements or with only a return statement); (iii) More than one control-field in a class are identified as refactoring opportunities (note: the expert chose only one of them to refactor, to avoid class explosion); (iv) Too many control-values and hence refactoring would lead to a large number of sub-classes. (v) The identified refactoring was found in obsolete/auto-generated part of the code.

5.1.3 How is the software quality affected by applying the identified refactoring opportunities?

To show the impact of Auto-SCST on improving the code quality, we first show the number of conditional-statements associated with the identified opportunities. Fig. 13 lists the number of uses (# associated conditional-statements, columns 11-14) of the identified control-fields, which gives an indication of the number of the conditional codes that will be replaced. It can be seen that Auto-SCST identifies refactoring opportunities with $\geq$ 2 uses in all the applications under consideration and even $\geq$ 15 uses in some.

Next, we show the improvement of software quality in terms of McCabe's cyclomatic complexity [McCabe 1976] metric (CCM, in short). CCM is an appropriate measure as RCP-refactoring removes the branches at the outer-level in a conditional-statement. In contrast, other possible code-quality metrics such as lines-of-code (LOC), number of new classes added, and change in inheritance depth are not much interesting, in our context: (1) Change in LOC: this metric is not interesting because LOC reduction due to the removal of conditional-statements is almost the same as LOC increase due to new methods; (2) New classes added: this metric is straightforward and it matches the #control-values (Section 3.2); (3) Change in depth of inheritance hierarchy of existing classes: this metric is trivial/uninteresting because in case of SC-refactoring, it always increases by 1 and in case of ST-refactoring, there is no change. Hence we use CCM.

We compute CCM for each of the methods that contained conditional-statements associated with the identified refactoring opportunities, before and after refactoring (SC/ST followed by RCP refactoring)[3]. To make a fair comparison with the input code, while computing the CCM of the refactored code, we do not ignore the CCM of the branch-bodies, as they remain unchanged in the new methods. CCM of a method m (to be refactored) is calculated as: (1) for the original code: CCM_o = 1+#decision points in m; (2) for the code to be obtained after refactoring = CCM_o−#decision points in the outer-levels of conditional-statements associated with the control-fields. Fig. 14 lists these metrics as an average over each application. It can be seen that SC/ST refactoring (coupled with RCP refactoring) led to improving the CCM of the associated methods by 13.18% to 54.88%.

5.1.4 How fast does Auto-SCST run? Fig. 13 (last column) shows the time taken by Auto-SCST to identify the refactoring opportunities. As it can be seen, the analysis is quite scalable: Auto-SCST took 1-337s (geomean 23.5s) to analyze the full applications. We found that the analysis time depends not only on the application size but also on the size of the SCCs (Section 4), precision of the static call resolution (too many 'compose' operations), length of call chains, and the method summaries sizes. Example: the higher timing for avrora is due to the last 3 reasons. However, considering the large number of classes present in avrora (#classes=1,731), we believe that the analysis time is reasonable. In addition, for the convenience of the developers, we also have a command line version of the tool (runs faster than the GUI version - geomean 19.6s) that can be invoked offline to obtain the list of refactoring opportunities; which can be fed to the Eclipse plugin of Auto-SCST to actually perform the refactorings.

Further, in practice, the developer may invoke Auto-SCST at sub-application level granularity (for example, package level, class level, and so on). This would lead to much more faster execution times (order of few seconds). Also, note that the actual refactoring (code-modification) is quite-fast; independent of the identification process (our focus).

5.1.5 Study of the points-to analysis as a parameter. We have found that the most time-consuming part in Auto-SCST is the points-to analysis component, which is invoked by our classification algorithm; the rest of the computation takes a small fraction (5.98%) of the total time. Since points-to analysis can be seen as a parameter to Auto-SCST, it is natural to ask if instead of the used context-sensitive whole-program analysis, an equally precise demand driven points-to analysis, or a less precise context-insensitive whole-program analysis can further speed up the tool, without compromising on the precision.

[3]Only performing SC/ST refactoring alone does not lead to any reduction in CCM. And RCP refactoring cannot be performed without doing a pre-pass of SC/ST refactoring. But together they lead to improvement in CCM.

Application	#points-to queries	Boomerang (sec)	Auto-SCST (sec)	Spark (sec)
javaGeom	91	5.8	3.9	1.6
jfreechart	707	409.4	25.2	15.5
jOcular	172	39.5	32.3	3.3
RackJ	73	31.4	70	1.4
UR	58	0.8	0.5	0.5
avrora	1788	1068.7	336	14.2
fop	882	636.5	54.2	19
sunflow	84	63.1	11.4	2

Fig. 15. Points-to analysis used in Auto-SCST vs Boomerang and Spark.

We have compared Boomerang [Späth et al. 2016], the only known demand-driven flow-sensitive, context-sensitive analysis for Java, and Spark [Lhoták and Hendren 2003], the inbuilt context-insensitive analysis of Soot, against our used points-to analysis. Fig. 15 presents brief comparison of the times taken by Boomerang, Spark, and Auto-SCST to compute the points-to information followed by the classification of SC/ST refactoring opportunities for the set of applications discussed in Section 5. We found that Auto-SCST generates a large number of points-to queries (up to 1788) in each application, and the performance of Boomerang was much lower than that of the used points-to analysis (average 418.3% slower). Note that our used points-to analysis computes only the specific information required by Auto-SCST. In contrast, a generic tool like Boomerang computes all-alias information for each variable in the query; which we believe is the main reason for slow-down. Extending the ideas of Boomerang to design an efficient demand-driven points-to analysis to suit the needs of Auto-SCST would be a challenging exercise and is left as an interesting future work.

In contrast, Spark ran much faster (average 66% faster). However, the resulting imprecision led a number of SC refactoring opportunities being classified as ST. For example, over the application used by the experts to evaluate Auto-SCST (UR, Sunflow, Jocular, JavaGeom, and RackJ), we found that compared to Spark, our used context-sensitive analysis led to 50.2% (geomean) more SC refactoring opportunities.

We conclude that using a precise whole program points-to analysis is more efficient than the demand-driven analysis for classifying SC/ST opportunities. And while a context-insensitive analysis is faster, it leads to a situation where a number of SC refactoring opportunities are missed. We argue that for the precision obtained, the time taken by our used whole-program points-to analysis is reasonable.

Overall: Auto-SCST successfully identifies a number of precise refactoring opportunities across varied real-world applications, and is scalable. 52% of the identified opportunities were accepted by both the experts and the refactoring improves the code quality.

5.2 Comparison with related work

To the best of our knowledge, we are not aware of any other tool whose focus is to identify SC/ST refactoring opportunities. The tools that come closest are the ones by Tsantalis and Chatzigeorgiou [2010] and Christopoulou et al. [2012]; we refer to these works as JDEO and STRP, respectively. The goal of JDEO is to identify and refactor some individual conditional-statements that can be replaced by polymorphic calls (but not necessarily controlled by control-fields that lead to SC/ST

Application	JDEO				STRP			
	|A−R|	|R−A|	|A∩R|(SC)	#Ref	|A−R|	|R−A|	|A∩R|(SC)	#Ref
javaGeom	5	0	0 (0)	0	2	4	3 (3)	3
jfreechart	99	0	1 (0)	1	89	18	11 (5)	18
jOcular	13	0	4 (4)	4	13	0	4 (2)	4
Rackj	17	0	0 (0)	0	15	2	2 (1)	2
UR	7	0	1 (0)	1	8	0	0 (0)	0
avrora	109	0	3 (0)	3	109	4	3 (0)	4
fop	102	2	18 (8)	20	×	×	×	×
sunflow	16	0	0 (0)	0	15	1	1 (1)	1

Fig. 16. Auto-SCST vs JDEO and STRP. A and R denote the set of comparable opportunities identified by Auto-SCST, JDEO/STRP, respectively. #Ref indicates the number of SC/ST refactorable opportunities identified.

refactoring). Similarly, the goal of STRP is to identify some individual conditional-statements that can be refactored using 'replace type code with Strategy' pattern (has similarities to ST refactoring; see Section 4). Consequently, a subset of the opportunities identified by JDEO/STRP does lead to SC/ST-refactoring. Hence, considering the similarities (and keeping in mind the differences in their goals), we present a brief comparison of Auto-SCST with JDEO and STRP.

We have studied all the *comparable* opportunities reported by JDEO and STRP. We consider any of their reported opportunities to be "comparable" in our context if (i) it involves fields of classes (so as to apply RCP refactoring), (ii) it has >1 associated cond-stmts (see Section 4), and (iii) the predicates in the associated cond-stmts do not compare the control-field against null (see Section 3). For each of the applications, Fig. 16, lists the unique and common comparable opportunities identified by JDEO vs Auto-SCST (columns 2-4) and STRP vs Auto-SCST (columns 6-8). Note: STRP crashed while analyzing fop.

Comparison against JDEO: We found that if we exclude fop, the set of comparable opportunities identified by JDEO forms a strict subset of the opportunities identified by Auto-SCST. In the case of fop, we studied the two additional reported opportunities and found that these opportunities cannot be refactored (note: the JDEO tool does perform refactoring in these cases, but generates incorrect code).

Comparison against STRP: We found that for the set of chosen applications, Auto-SCST identifies many more RCP opportunities than STRP. However, there were 29 comparable opportunities that were identified by STRP, but not by Auto-SCST, as shown in column 7 (Fig. 16). However, of these only eight were refactorable. We found that using the extensions discussed in Section 4 we can handle seven of them. (three of them by analyzing the full program and four by doing whole program constant propagation). The last case falls into the category where the conditional-statement uses a guard in which the control-field is accessed in a return expression of a function (involving other local variables/constants), deep inside a nested call.

For each of the applications, columns 4 and 8 show (in brackets) the possible SC opportunities that would be marked as ST/Strategy opportunity if we use JDEO or STRP for identifying RCP refactoring. Among the RCP opportunities identified by JDEO and STRP, Auto-SCST is able to identify 24 (22 unique) opportunities for SC refactoring. This shows the importance of Auto-SCST in comparison to JDEO and STRP in precisely classifying the refactoring opportunities (SC or ST).

Thus it can be seen that Auto-SCST is able to identify more SC/ST refactoring opportunities than JDEO and STRP; mainly because of our proposed analysis to identify control-fields. Also, the novel classification algorithm of Auto-SCST helps identify many SC opportunities.

Tool	Precision	Recall	Expert Acceptance		CCM impr (avg)	Time (avg)
			both	at least one		
Auto-SCST	100%	94.89%	52%	91%	32.5%	23.5
JDEO	93.5%	8.52%	20%	80%	21.7%	12.4
STRP	57.1%	6.25%	50%	80%	×	17.2

Fig. 17. Comparison of Auto-SCST with JDEO and STRP.

Fig. 17 gives a summary of the behavior of Auto-SCST, JDEO and STRP, in terms of (i) precision (column 2), (ii) recall (column 3), (iii) acceptance by the experts (columns 4, 5), (iv) average improvement to CCM (column 6), and (v) average execution time (column 7). We could not obtain the CCM for STRP as the tool does not have the option to perform the actual refactorings.

The comparison of precision and recall clearly states the insufficiency of existing tools such as JDEO and STRP and the relevance (and importance) of Auto-SCST for performing SC/ST refactoring. Note that this finding is only in the context of the SC/ST refactoring opportunities (the focus of our paper) and does not in anyway comment on the general effectiveness of JDEO and STRP, in the context of their respective goals (different from that of Auto-SCST).

For the five applications used by the experts to evaluate Auto-SCST (Section 5.1.2), the sets of identified 'comparable' opportunities by both JDEO and STRP are proper subsets of that by Auto-SCST. The columns 4-5 reveal that compared to JDEO and STRP, the percentage acceptance for Auto-SCST was better (though marginally).

In terms of percentage improvement to the CCM, we see that Auto-SCST leads to more improvement in CCM than JDEO. This is consistent with the more number of SC/ST refactoring opportunities identified by Auto-SCST than JDEO.

In terms of time taken, we see that Auto-SCST takes more time than JDEO and STRP. However, we believe that considering the higher precision and recall, the additional time taken is a reasonable trade-off.

Summary: Looking at the low impact of JDEO and STRP in identifying SC/ST refactoring opportunities, we can easily see the need and importance for a dedicated (and effective) tool like Auto-SCST, for identifying (and refactoring) SC/ST opportunities.

5.3 Threats To Validity

A usual threat to external validity is that the conclusions drawn from a limited set of Java applications might not be generalized to wider classes of applications. We tried to mitigate this threat by using open-source applications with varied domains, sizes and implementation stages (pre-alpha, alpha and matured).

Threats to internal validity: (1) The relevance of the identified refactoring opportunities is analyzed manually. To mitigate this threat, we have taken two steps: (a) requested experts with vast experience in software engineering to analyze and used their inputs; (b) complete list of identified opportunities and the tool are made available [Vedurada and Nandivada 2018] for cross-checking and experiment repeating. (2) The real developers were not involved and therefore whether developers use SC/ST refactoring in practice is not concluded. In general, due to their busy schedule, developers of unknown open-source projects do not participate in such experiments - a known issue. For example, only the developers of jOcular responded; they have opened a ticket to refactor. To mitigate this threat, as done by the prior researchers [Christopoulou et al. 2012; Tsantalis and Chatzigeorgiou 2010], we used independent experts as a substitute. (3) The experts had insufficient knowledge of the applications when analyzing the identified refactoring opportunities, and consequently may have erred in their judgment. We tried to mitigate this threat by using the feedback from two experts (instead of just one) and studied the inter-rater agreement. (4) The experts were only used to verify the identified refactoring opportunities, but not the missed opportunities. Such a threat-to-validity is unavoidable, as otherwise, it would require manual analysis of large unfamiliar applications (11K-31K LOC); nearly impractical for independent experts. (5) For computing recall, a statistically significant subset of 400 random samples was chosen from a limited subset of 980 "potential" candidates instead of from considering all the 253,167 conditional-statements (on fields, locals, and array-elements) as expressions containing potential-opportunities. Such a threat-to-validity is unavoidable: randomly choosing 400 samples from all the conditional-statements in the program will lead to a situation where most of them are not real opportunities (very small #total-refactorable-opportunities); this would have led to an unreliable recall-value.

5.4 A brief note on Auto-SCST-R

In Fig. 13, column 15 lists the counts of the control-fields (MCFs) that can be refactored by Auto-SCST-R; these MCFs are identified using the procedure discussed in Section 3.4. Column 16 shows that on (geo) average 77.3% of the CFs are automatically refactorable by the current version of Auto-SCST-R. On analysis of the remaining opportunities, we found that among the conditions (C1-C4) discussed in Section 3.4, 17.8% of them were not marked because of C1, 34.3% due to C2, 2.7% due to C3, 37% due to C4, and 8.2% due to multiple conditions (C1-C4) affecting together. We are working on improving the scope of Auto-SCST-R to handle the remaining opportunities.

6 RELATED WORK

There have been many works that try to (semi-) automatically identify different types of refactoring opportunities: abstract factory refactoring [Jeon et al. 2002], composite pattern refactoring [Jebelean et al. 2010], move method refactoring [Tsantalis and Chatzigeorgiou 2009], extract superclass refactoring [Opdyke and Johnson 1993], strategy pattern refactoring [Christopoulou et al. 2012], subclass/state pattern refactoring [Tsantalis and Chatzigeorgiou 2010], introduce null object refactoring [Gaitani et al. 2015], and so on. We present an approach to automatically identify opportunities for both subclass (SC) and state (ST) pattern, as part of RCP refactoring.

Opdyke [1992] introduced RCP refactoring as "refactoring to specialize: subclassing, and simplifying conditionals" in his largely cited thesis. Further explanation and mechanisms for performing RCP-refactoring are given by later works [Demeyer et al. 2002; Fowler 1999; Kerievsky 2005]. Our work is inspired by these refactoring mechanisms, and mainly focuses on the identification of SC/ST refactoring opportunities.

The popular Eclipse plug-in JDeodorant [Tsantalis and Chatzigeorgiou 2010] identifies individual conditional-statements that can be replaced by polymorphic calls; these include a few of the SC/ST refactoring opportunities. They identify SC opportunities only in conditional-statements involving RTTI (RunTime Type Identification). In case of ST pattern, they propose grouping of refactoring opportunities based on the values of the named-constants (can lead to grouping of seemingly unrelated pieces of code). In contrast, we identify SC and ST patterns precisely based on the identification of control-fields. Further, JDeodorant focuses mainly on the methods that contain the conditional-statements (intra-procedural) and misses the possible flow of information from other classes and methods via the heap.

Christopoulou et al. [2012] also identify the importance of class instance fields (instead of simply named constants) and give steps to identify opportunities to refactor using 'replace type code with Strategy' pattern (has similarities to ST refactoring). However, their proposed scheme is restrictive for our purpose: it does not handle the flow of data via local variables or 'get' methods, updating of the state fields in the class, or possible data-flow via heap updates in other parts of the application.

In contrast, we formally define a control-field and present an efficient algorithm (based on data flow analysis) to identify the precise control-fields. Further, after identifying the opportunities for SC/ST-refactoring, we use a novel data flow analysis to precisely classify them as SC- or ST opportunity.

Kataoka et al. [2001] use a dynamic invariant detection tool called Daikon to automatically detect refactoring opportunities for several refactorings such as eliminate useless return value, encapsulate downcast, separate query from modifier, and so on. Streckenbach and Snelting [2004] present a tool (KABA) that automates the refactoring of Java class hierarchies. KABA is based on dynamic as well as static analysis. In contrast, ours is a static analysis based tool and it aims at the automatic identification of SC/ST opportunities.

Prior works have used notions similar to that of control-fields. Komondoor et al. [2012] identify tag fields (that directly control independent pieces of code) using the techniques presented by Komondoor and Ramalingam [2007] to identify services in Cobol programs. Komondoor and Ramalingam recover data models from weakly typed languages, using guarded dependencies to identify the flow of dependencies from input statements and based on that, identify the conditionally executed independent pieces of code and data to generate class hierarchies. Control fields are more general in nature and we use them to identify opportunities for SC/ST-refactoring.

Many refactoring tools have been developed for popular OO languages [Bavota et al. 2014; Tokuda and Batory 2001; Tsantalis and Chatzigeorgiou 2009, 2010]. However, none of these focus on SC- and ST-refactorings. In contrast, Auto-SCST helps perform RCP refactoring, by automatically identifying SC and ST refactoring opportunities.

7 CONCLUSION

RCP refactoring is a popular way to improve code with conditional-state-checking statements. Effective RCP refactoring depends on systematically performing “Replace Type Code with Subclass" (SC) and “Replace Type Code with State" (ST) refactorings. In this paper, we present an approach to identify SC and ST refactoring opportunities. We define control-fields, and present a static analysis to identify the same. The control-fields form the basis of our refactoring-opportunity identification steps. We present a novel flow-sensitive context-sensitive analysis to classify these opportunities into SC and ST. We have implemented our proposed approach as an Eclipse refactoring plug-in (Auto-SCST) to identify SC/ST refactoring opportunities and perform refactoring. We have used Auto-SCST to evaluate our approach on eight open-source Java applications. Based on the results, we conclude that Auto-SCST successfully identifies relevant and refactorable refactoring opportunities from large code-bases (otherwise, a non-trivial task to do manually). We believe that the precision and scalability of Auto-SCST makes it a practical and valuable asset in the context of maintaining large legacy code.

A APPENDIX

A.1 Details of the function `computeRelavantFields`

We detail the working of the function `computeRelevantFields` (Fig. 18) here. It uses a set CF_{iv} (initialized to the empty set) to hold the potential control-fields for the variable v, at conditional-point i. It also maintains a *relevant-variable-set* R for each node in the CFG; all initialized to the empty set. The relevant-variable-set of a node represents the variables that are relevant to input variable v, or in other words, the variables whose values may flow to v. The procedure is an iterative worklist-based backward data-flow algorithm; WL is initialized to the predecessor set of i.

For each CFG node n in the WL, `computeRelevantFields` first adds all the relevant variables of the successors to the relevant-variable-set of n. We process the node n only if the variable defined

```
1  Function computeRelevantFields($G, i, v$) // Returns control-fields that may reach $v$ at node $i$.
2      $CF_{iv} := \emptyset$;                                                    // local variable
3      foreach $n' \in G$ do $R(n') := \emptyset$;
4      $R(i) := \{v\}$; $WL := pred(i)$;
5      while $WL \neq \emptyset$ do
6          $n := remove(WL)$; $R_{old}(n) := R(n)$;
7          foreach $m \in succ(n)$ do $R(n) := R(n) \cup R(m)$;
8          if $(def(n) \cap R(n)) \neq \emptyset$ then
9              $R(n) := R(n) - def(n)$;                                           // remove def
10             if $\neg sCopyStmt(n)$ then return $\emptyset$;
11             $x := RHS(n)$;                                                     // Right Hand Side
12             if $isVar(x)$ then $R(n) := R(n) \cup \{x\}$;                      // add use
13             else $CF_{iv} := CF_{iv} \cup$ PFInfo($x$);                        // possible field deref
14         if $R_{old}(n) \neq R(n)$ then $WL := WL \cup pred(n)$;
15     return $CF_{iv}$;
```

Fig. 18. Algorithm to detect potential control-fields that satisfy CF-Property 2

in n is part of the current relevant-variable-set, and n is a sCopy statement. If $RHS(n)$ is a local variable then it is added to $R(n)$ (Line 12). Otherwise, we add the potential control-fields returned by `PFInfo(x)` to CF_{iv} (Line 13). Whenever the relevant-variable-set of a node changes, the procedure adds all the corresponding predecessor nodes to the worklist (Line 14).

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their insightful comments and suggestions. We thank Nikolaos Tsantalis and Danny Dig for their valuable feedback and suggestions that helped in improving the evaluation of the paper. We thank the two experts for their participation in the experimental evaluation and also for providing insightful comments. Jyothi Vedurada is supported by the TCS Research Scholarship Program for her doctoral studies.

REFERENCES

Gabriele Bavota, Andrea Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Automating Extract Class Refactoring: An Improved Method and Its Evaluation. *Empirical Softw. Engg.* 19, 6 (Dec. 2014), 1617–1664. https://doi.org/10.1007/s10664-013-9256-x

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant Context Inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/292540.292554

Aikaterini Christopoulou, E. A. Giakoumakis, Vassilis E. Zafeiris, and Vasiliki Soukara. 2012. Automated Refactoring to the Strategy Design Pattern. *Inf. Softw. Technol.* 54, 11 (Nov. 2012), 1202–1214. https://doi.org/10.1016/j.infsof.2012.05.004

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, London, UK, UK, 77–101. http://dl.acm.org/citation.cfm?id=646153.679523

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2002. *Object-oriented reengineering patterns.* Elsevier.

Eclipse. 2017. Eclipse: A Java Integrated Development Environment (IDE). https://eclipse.org/

M. Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Maria Anna G. Gaitani, Vassilis E. Zafeiris, N.A. Diamantidis, and E.A. Giakoumakis. 2015. Automated Refactoring to the Null Object Design Pattern. *Inf. Softw. Technol.* 59, C (March 2015), 33–52. https://doi.org/10.1016/j.infsof.2014.10.010

William H. Harrison. 1977. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering* SE-13, 3 (May 1977).

C. Jebelean, C.B. Chirila, and V Cretu. 2010. A logic based approach to locate composite refactoring opportunities in object-oriented code. In *Automation Quality and Testing Robotics*. IEEE, 1–6.

Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. 2002. An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs. In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSEC '02)*. IEEE Computer Society, Washington, DC, USA, 337–. http://dl.acm.org/citation.cfm?id=785409.785835

Sandeepa Harshanganie Kannangara and Janaka Wijayanayake. 2014. An Empirical Exploration of Refactoring effect on Software Quality using External Quality Factors. *The International Journal on Advances in ICT for Emerging Regions (ICTer)* 7, 2 (2014).

Yoshio Kataoka, David Notkin, Michael D Ernst, and William G Griswold. 2001. Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 736.

Joshua Kerievsky. 2005. *Refactoring to patterns*. Pearson Deutschland GmbH.

Jongwook Kim, Don Batory, and Danny Dig. 2015. Scripting Parametric Refactorings in Java to Retrofit Design Patterns. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 211–220. https://doi.org/10.1109/ICSM.2015.7332467

Raghavan Komondoor, V. Krishna Nandivada, Saurabh Sinha, and John Field. 2012. Identifying Services from Legacy Batch Applications. In *Proceedings of the 5th India Software Engineering Conference (ISEC '12)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2134254.2134257

Raghavan Komondoor and G. Ramalingam. 2007. Recovering Data Models via Guarded Dependences. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)*. IEEE Computer Society, Washington, DC, USA, 110–119. https://doi.org/10.1109/WCRE.2007.40

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA. UMI Order No. GAX93-05645.

William F. Opdyke and Ralph E. Johnson. 1993. Creating Abstract Superclasses by Refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science (CSC '93)*. ACM, New York, NY, USA, 66–73. https://doi.org/10.1145/170791.170804

Alexandru D. Salcianu. 2006. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. Ph.D. Dissertation. Cambridge, MA, USA. Advisor(s) Rinard, Martin C. AAI0818179.

Sourceforge. 2016. SoureForge. https://sourceforge.net/

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Mirko Streckenbach and Gregor Snelting. 2004. Refactoring Class Hierarchies with KABA. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 315–330. https://doi.org/10.1145/1028976.1029003

Lance Tokuda and Don Batory. 2001. Evolving Object-Oriented Designs with Refactorings. *Automated Software Engg.* 8, 1 (Jan. 2001), 89–120. https://doi.org/10.1023/A:1008715808855

Nikolaos Tsantalis. 2018. https://users.encs.concordia.ca/~nikolaos/stats.html.

Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2010. Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software* 83, 3 (2010), 391–404.

Jyothi Vedurada and V Krishna Nandivada. 2017. Refactoring Opportunities for Replacing Type Code with State and Subclass. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 305–307. https://doi.org/10.1109/ICSE-C.2017.97

Jyothi Vedurada and V. Krishna Nandivada. 2018. Supplementary Material. https://github.com/anony-user/Auto-SCST.

John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400