Technical report IAI-TR-2012-01, ISSN 0944-8535, Computer Science Department III, University of Bonn, Germany, Feb. 2012, http://sewiki.iai.uni-bonn.de/research/dpd/
Short version to appear in Proceedings of CSMR 2012, IEEE Computer Society 2012.

Joining Forces for Higher Precision and Recall of Design Pattern Detection

Alexander Binun*, Günter Kniesel*

*University of Bonn, Bonn, Germany; Email: gk@iai.uni-bonn.de, binun@iai.uni-bonn.de

***Abstract*—Automated design pattern detection (DPD) is a challenging reengineering task that has been shown to require combination of complex structural and behavioural analyses for good results. Still, the detection quality (precision and recall) of existing tools has so far been insufficient to make DPD integral part of current IDEs and development practices. In this paper we present a novel approach and a related tool, DPJF. For all implemented pattern detectors and all projects used for evaluation, DPJF provides competitive performance while achieving the best recall of all evaluated tools (with a median of 89%) and 100% precision. These results lay the basis for routine application of DPD in program comprehension and software quality assesment. The high detection quality is achieved by a well-balanced combination of structural and behavioural analysis techniques whereas the good performance is achieved by emprically validated simplifications of the individual techniques.**

I. Introduction

Identification of design patterns can deliver important information to designers when it comes to understanding unknown code. Therefore, automated design pattern detection (DPD) is highly desirable. Extending the state of art overview by Dong and Peng [1], Kniesel and Binun [2], presented a practical evaluation of five DPD tools implementing different DPD techniques. Motivated by their findings they proposed a novel DPD approach based on *data fusion*, which combines design pattern candidate sets coming from different tools, and showed that this approach is able (1) to provide correct diagnostics even if the inputs from the evaluated tools were partly wrong and (2) to detect patterns instances not identified by the individual tools.

In this paper, we go one step beyond fusion of information from tool outputs by showing that even better results can be achieved by combining the underlying DPD techniques. The resulting tool is called DPJF (Detect Patterns by Joining Forces) since it jointly applies existing DPD techniques (forces). However, it does not simply implement the sum of known techniques. To improveme precision and recall without compromising detection speed, DPJF *combines limited variations* of many techniques. For example, when DPJF checks whether method M_1 calls method M_2 it follows method invocation relations via at most one intermediate method. Such limitations are not ad hoc but motivated by *systematic empirical analysis* of open source code repositories.

Section II introduces basic concepts and terminology. Sections III, IV and V address the main challenges of our approach: achieving high precision, recall and speed. Section VI describes the implementation of DPJF. Section VII evaluates DPJF in comparison to four other tools (Fujaba [3], Pinot [4], PTIDEJ [5], and SSA [6]). Section VIII explains how DPJF goes beyond traditional DPD. Section IX concludes and sketches possible lines of future work.

II. Basics

The approach presented in this paper is based on a view of designs (or design *motifs*, following the terminology of [7]) as sets of roles and relationships. Relationships capture collaborations between roles [8], [9], [10], [11].

Motif *occurrences* are sets of program elements that play these roles and exhibit the relationships specified for the played roles. Given a program to be analysed, DPD is the problem of finding correct *role assignments* [12], that is, determine which program element plays which role in which design motif. Based on the identified role assignments a DPD tool suggests DP *candidates*, that is, sets of program elements that *might* represent motif occurrences.

DPD potentially suffers from *false negatives* (missed motif occurrences) and *false positives* (suggestion of wrong pattern candidates) [1]. A tool that yields less false negatives on the same input project has better *recall*. One that yields less false positives has better *precision* [13]. See [14], [12] for further basic concepts and terminology compiled by four DPD research groups.

A. Motif Groups

Motifs are characterized by sets of structural and behavioural constraints. In our approach, design motifs that share most of their structural constraints form a *motif group*. Behavioural constraints are used to discriminate the motifs within a group from false positives and from each other. In this paper we focus mainly on two motif groups whose individual motifs are described by Gamma et al. [15]:

- **Supertype Forwarders** (Decorator, Proxy, and Chain of Responsibility[1]) model chains of objects, each linked to a single 'parent' to which it forwards invocations that it does not process itself. The objects in the chain are instances of a common supertype and the forwarding methods invoke methods with the same signature, so that clients cannot tell who processed their message.

[1]Abreviated in this article as 'CoR'.

- **Decouplers** (Observer, Composite, Bridge, State, Strategy) model a 'master object' (subject, composite, etc) that manages a variable set of dependent object(s) with whom it interacts. An action initiated by the master is propagated to all dependents. The master may have the same interface as the dependents (in Composite) but the interfaces may also differ (in Observer).

B. Programs and Program Analyses

Because dynamic analysis [16], [3], [17], [18], [19] depends on hard to achieve complete test sets for the analysed program, DPJF uses static analysis.

1) Program Abstraction: The supported analyses are defined on *types* (classes or interfaces), *variables* (fields, local variables, parameters, collection elements and method return values), class, instance and constructor *methods*, a selection of *DPD-relevant statements* (method calls, null checks, loop entries and dummy statements for the entry and exit points of a method) and *objects*. In our static analysis, objects are either identified with their creation statements or are anonymous. The *anonymous object* represents objects created in sections of a program that are not available for analysis (e.g. in extensions of a framework by unknown clients).

2) Structural analyses: The main structural analyses are subtyping and ownership (containment). A type owns shared ('static') fields and methods, an object owns instance fields and methods, a method owns local variables and parameters. In addition, every variable is associated with its static type, every method with its visibility, its type signature and a flag indicating whether it is a constructor. For every method call we distinguish its receiver variable, its parameters and the set of potentially called methods. Called methods are computed based on the types of objects pointed to by the receiver variable, following Tip et al. [20] .

3) Behavioral analyses: Program behaviour is captured in terms of control and data flow.

Control flow: The control flow graph (CFG) contains only nodes for DPD-relevant statements (see II-B1). An edge $n_1 \rightarrow n_2$ exists if and only if execution of node n_1 can be followed by execution of node n_2 either immediately or by passing only through DPD-irrelevant statements. In the latter case, the arc abstracts a sequence of irrelevant statements beetween n_1 and n_2. We say that a method M invokes a method call mc if there is a path from M's entry point to mc. The depth of the invocation is the number of method entry nodes minus the number of method exit nodes on the path. An invocation of depth = 0 is *direct*, one of depth > 0 is *indir*ect. Figure1 shows an indirect invocation of mc by M that has depth 3.

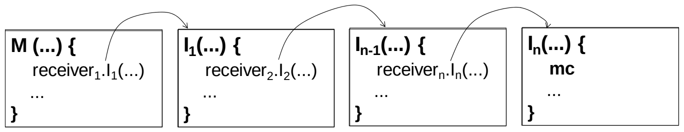


Figure 1: Indirect invocation

Data flow: The data flow graph contains nodes for variables and objects (see II-B1). An edge $o \rightarrowtail v$ exists if the object o is assigned to the variable v. An edge $v_1 \rightarrowtail v_2$ exists if the variable v_1 is assigned to the variable v_2. Parameter and result passing are treated like assignments. The *points-to set* of a variable v are all objects (including the anonymous one) from which v is reachable in the data flow graph.

C. Approach In a Nutshell

The next three sections elaborate the main principles of our approach:

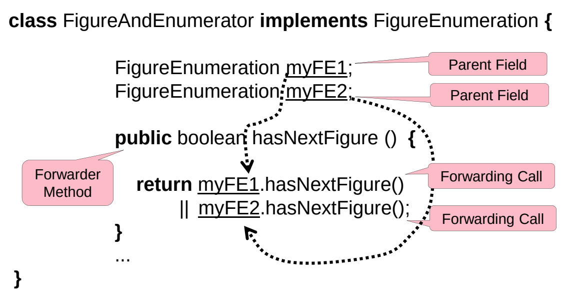
1) To achieve high precision, we combine various structural and behavioural constraints (and the static analysis techniques required to verify them).
2) To achive high recall, we relax overly restrictive constraints imposed by other tools for precision.
3) To achieve high speed we limit the depth of the analysis on the basis of empirically validated heuristics.

III. Improving Precision

False positives typically emerge from defining the characteristics of a design motif by insufficiently strong constraints. This leads to candidates that are either no instances of any sought pattern or no instances of the reported pattern but of a *superpattern* characterized by weaker constraints [2]. The ability to check stronger constraints eliminates false positives and enables distinguishing between motifs whose differences are quite subtle (e.g. implementation variants such as the Statically Typed and Dynamically Typed Proxy – see Sec. III-D). However, too strong constraints are counterproductive since they will discard proper instances reducing recall (Sec. IV). So the challenge is to find suitable constraints. From our empirical evaluation we found that false positives typically arise from insufficiently strong *behavioral* constraints, as independently observed also by Fülöp et al. [21]. In the remainder of this section we discuss new high-level behavioral constraints that have helped us boost the precision achieved for DP candidates from the motif groups introduced in Sec. II-A.

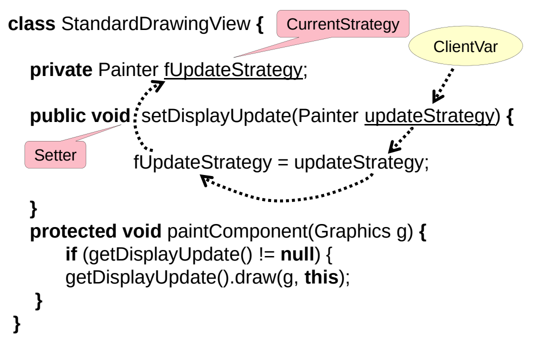
A. Forward to a single object

In occurrences of motifs from the Forwarder group (Decorator, Proxy or CoR), the forwarding method of an object O calls the corresponding method of a *single* 'parent' object. If O forwards through different fields to different parents we have no Decorator. Figure 2 shows a class from JHotDraw 6.0, $FigureAndEnumerator$, whose instances forward to *two* receiver objects referenced by the fields *myFE1* and *myFE2*. All reviewed tools that are able to recognize the Decorator pattern and identify role players at the method level (PINOT and SSA) report two Decorator candidates: One where *myFE1* plays the 'Parent' role and *myFE1.nextFigure*() plays the 'Forwarding Call' role and one where *myFE2* and *myFE2.nextFigure*() play these roles. However, none is a proper Decorator because forwarding is done to multiple parents. Because the parents are organized in a fixed tree-like structure we call this motif a *Tree Forwarder*.



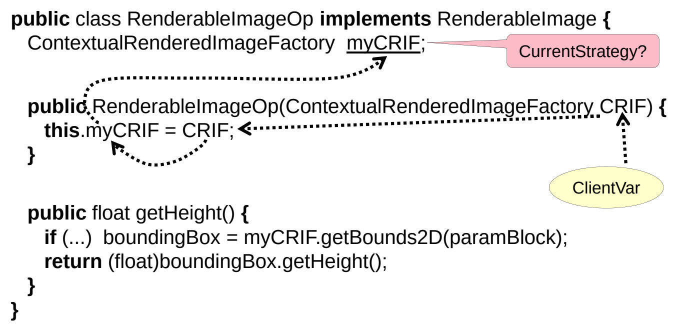
```
class FigureAndEnumerator implements FigureEnumeration {

    FigureEnumeration myFE1;
    FigureEnumeration myFE2;

    public boolean hasNextFigure () {

        return myFE1.hasNextFigure()
            || myFE2.hasNextFigure();
    }
    ...
}
```

Figure 2: Forward to multiple objects (JHotDraw 6.0). In all diagrams dotted lines represent data flow and dashed lines represent control flow.



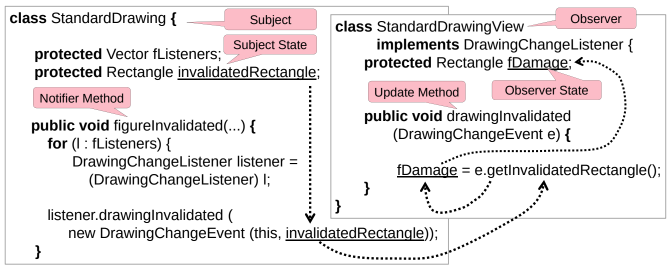
```
class StandardDrawingView {

    private Painter fUpdateStrategy;

    public void setDisplayUpdate(Painter updateStrategy) {

        fUpdateStrategy = updateStrategy;

    }
    protected void paintComponent(Graphics g) {
        if (getDisplayUpdate() != null) {
        getDisplayUpdate().draw(g, this);
    }
}
```

Figure 3: Maintaining a field (JHotDRaw 6.0)



```
public class RenderableImageOp implements RenderableImage {
    ContextualRenderedImageFactory  myCRIF;

    public RenderableImageOp(ContextualRenderedImageFactory CRIF) {
        this.myCRIF = CRIF;
    }

    public float getHeight() {
        if (...)  boundingBox = myCRIF.getBounds2D(paramBlock);
        return (float)boundingBox.getHeight();
    }
}
```

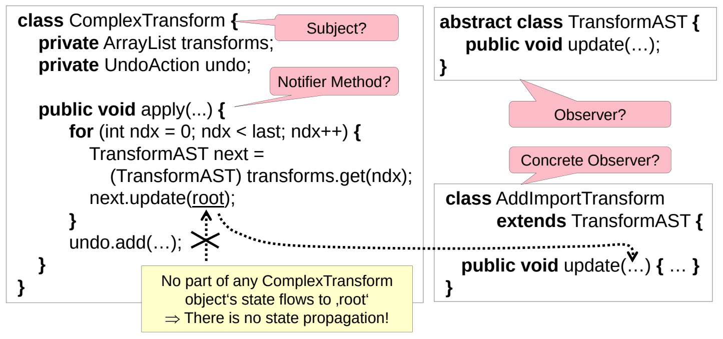
Figure 4: No field maintenance – myCRIF is set only in the constructor (AWT 1.14)



```
class StandardDrawing {

    protected Vector fListeners;
    protected Rectangle invalidatedRectangle;

    public void figureInvalidated(...) {
        for (l : fListeners) {
            DrawingChangeListener listener =
                (DrawingChangeListener) l;

        listener.drawingInvalidated (
            new DrawingChangeEvent (this, invalidatedRectangle));
    }
```

```
class StandardDrawingView
    implements DrawingChangeListener {
    protected Rectangle fDamage;

    public void drawingInvalidated
        (DrawingChangeEvent e) {

        fDamage = e.getInvalidatedRectangle();
    }
}
```

Figure 5: State propagation in a proper Observer (JHotDRaw 5.1)



```
class ComplexTransform {
    private ArrayList transforms;
    private UndoAction undo;

    public void apply(...) {
        for (int ndx = 0; ndx < last; ndx++) {
            TransformAST next =
                (TransformAST) transforms.get(ndx);
            next.update(root);
        }
        undo.add(…);
    }
}
```

```
abstract class TransformAST {
    public void update(…);
}
```

```
class AddImportTransform
        extends TransformAST {

    public void update(…) { … }
}
```

Subject?
Notifier Method?
Observer?
Concrete Observer?
No part of any ComplexTransform object‘s state flows to ‚root‘
⇒ There is no state propagation!

Figure 6: No state propagation in a service loop (JRefactory 2.6)

B. *Field maintenance*

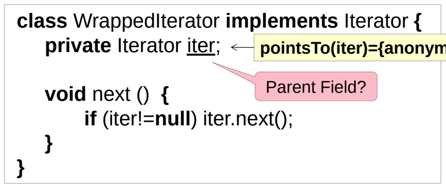
Certain collaborational design patterns (e.g. Observer, Composite, State, Strategy, Bridge) facilitate decoupling of master objects from their dependent objects. The common structural characteristic is that the master object O contains a field F that points to the dependent objects, no matter whether F is a collection (in Observer and Composite) or a scalar (in State, Strategy and Bridge). The common behavioural characteristic is that *the contents of* F *can change after* O *is constructed*. That is, the contents of F (or of the elements of the collection referenced by F) is updated outside the constructor of O.

Figure 3 shows a proper Strategy motif occurrence from JHotDraw 6.0 in which the method *setDisplayUpdateStrategy* updates the field *fUpdateStrategy* after an instance of the *StandardDrawingView* is created. In contrast, Fig. 4 shows a false Strategy candidate from AWT 1.14 accepted by SSA, Fujaba and Ptidej. These tools assume that the field *myCRIF* plays the ‘Current Strategy’ role, pointing to an object of type *ContextualRenderedImageFactory* that plays the ‘Strategy’ role. However, the field *myCRIF* is updated only during initialization of an instance of *RenderableImageOp*, so the field maintenance constraint is violated.

C. *State Propagation*

In the Observer pattern, subject state changes are propagated during execution of the notification method from observer field(s) to the field(s) containing the subject state information. We say that *state is propagated* from a subject S to a related observer O if there are fields $s_{1..m}$ in S and fields $o_{1..k}$ in O and there is a dataflow from $s_{1..m}$ to $o_{1..k}$ caused by the execution of the method that plays the Notifier role.

Figure 5 describes an Observer ocurrence from JHotDraw 5.1 where state propagation occurs. The Subject role is played by instances of the type $StandardDrawing$. When a drawing d is changed, its region described by the field *invalidatedRectangle* is updated. The update is propagated to each field *fDamage* owned by a related drawing view. This propagation takes place via the data flow *d.invalidatedRectangle* $\rightarrowtail$ *e.rectangle* $\rightarrowtail$ *dl.fDamage*, where e is an instance of *DrawingChangeEvent* and *dl* is an object pointed to by the *fListeners* field of d.



```
class WrappedIterator implements Iterator {
    private Iterator iter;

    void next () {
        if (iter!=null) iter.next();
    }
}
```

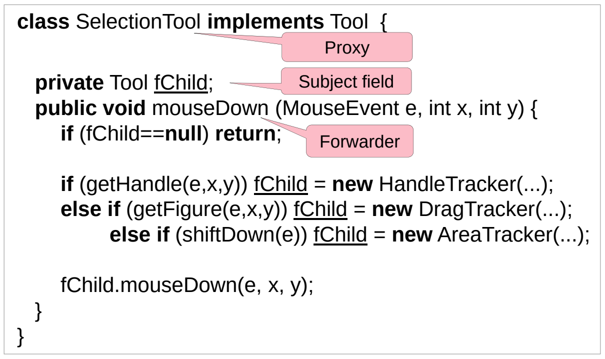
Figure 7: A Decorator – structurally similar to the Dynamicaly Typed Proxy, but without Sibling Creation (Quick UML 2001)

The scenario illustrated in Figure 6 describes a false Observer candidate from JRefactory 2.6 where no state propagation occurs. The Subject role was assumed to be played by instances of the type *ComplexTransform*, which represent complex refactorings. A complex refactoring stores basic ones in its field *transforms*, which was assumed to play the 'Observer Variable' role. Basic refactorings are executed sequentially via the method *apply*, which seemed to play the 'Notification Method' role since it calls the method *update* of each basic refactoring. However, there is no dataflow between variables representing the assumed subject and observer state. Instead, the assumed 'Notification Methods' iterate over a collection just to perform a 'service task'. We call this scenario a *Service Loop*. None of the reviewed existing DPD tools verifies the state propagation condition when searching for Observer candidates. Therefore they wrongly classify service loops as Observers.

D. Sibling Creation

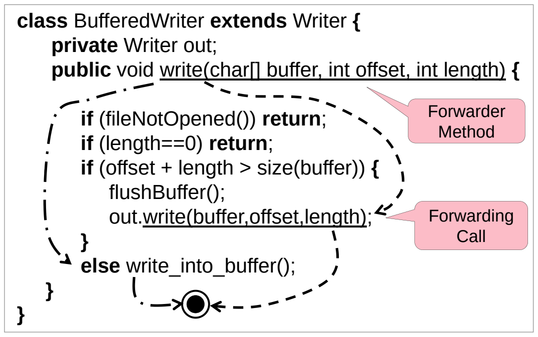
Sibling creation is an example of a behavioural constraint that helps discriminate a particular implementation variant of a pattern from its other variants and from structurally similar patterns. In the Proxy pattern, each Real Subject (i.e. each object pointed to by the field $Proxy.subject$) must have a type that is a sibling of the Proxy type. In the implementation variant presented by Gamma et al. [15] this is easy to verify since the *static* type of the field $Proxy.subject$ already is a sibling of the Proxy type. We refer to it as the Statically Typed Proxy variant. In the variant that we call a Dynamically Typed Proxy (Figure 8), the static type of the $Proxy.subject$ field can be either a sibling or a supertype of the *Proxy* type. This is structurally similar to a Decorator or CoR (Fig. 7). Therefore, proper identification of Dynamically Typed Proxy occurrences requires analysing the run-time types of the objects pointed to by the $Proxy.subject$ field. If they are all strict siblings of the *Proxy* type, we say that *Sibling Creation* occurs. The behavioral analysis needed here is the points-to analysis (Sec. II-B).

Fig. 8 presents a Dynamically Typed Proxy occurrence from JHotDraw 5.1 whose forwarding method initializes the Real Subject *fChild* so that *pointsTo*(*fChild*) = { *new HandleTracker*(), *new AreaTracker*(), *new DragTracker*() }. The types of the objects contained in $pointsTo(child)$ are siblings of $SelectionTool$, so the 'Sibling Creation' scenario applies. In contrast, it does not apply in the structurally similar Decorator instance from Fig. 7 where forwarding can



```
class SelectionTool implements Tool {

  private Tool fChild;
  public void mouseDown (MouseEvent e, int x, int y) {
     if (fChild==null) return;

     if (getHandle(e,x,y)) fChild = new HandleTracker(...);
     else if (getFigure(e,x,y)) fChild = new DragTracker(...);
         else if (shiftDown(e)) fChild = new AreaTracker(...);

     fChild.mouseDown(e, x, y);
  }
}
```

Figure 8: Sibling Creation in a Dynamically Typed Proxy (the different `Tracker` types assigned to `fChild` are no subtypes of `Tool`)



```
class BufferedWriter extends Writer {
    private Writer out;
    public void write(char[] buffer, int offset, int length) {
        if (fileNotOpened()) return;
        if (length==0) return;
        if (offset + length > size(buffer)) {
            flushBuffer();
            out.write(buffer,offset,length);
        }
        else write_into_buffer();
    }
}
```

Figure 9: Partial Coverage in a CoR occurrence (Java IO 1.4)

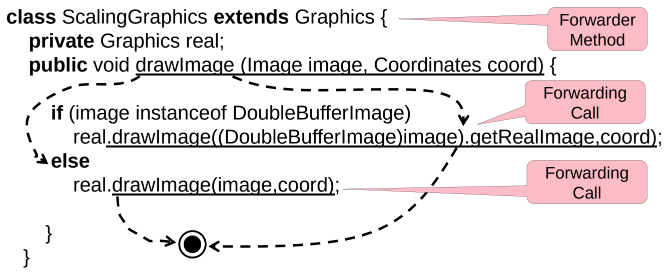
potentially be done to any object of type Decorator because $pointsTo(fComponent) = \{anonymous\}$. No DPD tool reviewed in our empirical evaluation supports concepts similar to Sibling Creation.

E. Object Coverage by Calls

The motifs of certain patterns can be distinguished by verifying whether certain interactions are performed on all objects pointed to by a set of fields $f_{1..n}$ on which a method M may invoke some callee method m:

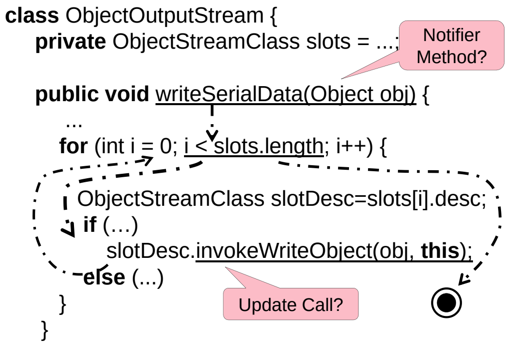
- *Full Coverage* - m is invoked *on each object* pointed to by $f_{1..n}$ *in each execution* of M.
- *Partial Coverage* - m is invoked on each object pointed to by $f_{1..n}$ *in some execution(s)* of M but there exist *diverting execution paths* in which this is not the case (that is, on a diverting execution path *m* is not invoked at all or not invoked on objects pointed to by $f_{1..n}$). Control flow paths initiated by successful null-pointer checks and exception control flow are not regarded as diverting execution paths since they represent error handling code, not parts of the application logic that is typically represented by a pattern.

Full coverage is typical for Decorator (a decorator object always forwards to its parent object), Observer (a subject



```
class ScalingGraphics extends Graphics {
  private Graphics real;
  public void drawImage (Image image, Coordinates coord) {

    if (image instanceof DoubleBufferImage)
       real.drawImage((DoubleBufferImage)image).getRealImage,coord);
    else
       real.drawImage(image,coord);

    }
  }
```

Figure 10: Total Coverage in a Decorator (JHotDrRaw 6)



```
class ObjectOutputStream {
    private ObjectStreamClass slots = ...;

    public void writeSerialData(Object obj) {
       ...
       for (int i = 0; i < slots.length; i++) {

         ObjectStreamClass slotDesc=slots[i].desc;
         if (...)
           slotDesc.invokeWriteObject(obj, this);
         else (...)
       }
    }
```

Figure 11: Partial Coverage is violated in a false Observer candidate (Java IO 1.4)

always propagates state changes to all its observers) and Composite (a container always propagates an operation to all its children). Partial coverage is typical for Chain of Responsibility (a handler object forwards to its next object only in some execution).

To the best of our knowledge, only the "conditional forwarding" check of PINOT resembles our "coverage" concept. The differences are illustrated below by figures in which the end point of a method is depicted by a black oval, non-diverting paths are depicted by dashed lines and diverting paths are depicted by dotted-dashed lines. Figure 9 illustrates a CoR occurrence from Java IO where the field *out* plays the 'Next Field' role and the method *write* plays the 'Forwarder' role. PINOT reports a CoR candidate since forwarding is done conditionally, when the write request causes moving beyond the buffer boundaries. This is correct, since there is actually a diverting path on the else branch of the conditional. In contrast, Figure 10 illustrates a Decorator occurrence from JHotDRaw 6.0. Here PINOT wrongly reports a CoR candidate because it finds a forwarding call in *some* branch of an if-statement and does not realize that forwarding is done in *each* of the executions of *drawImage*. This example emphasizes the need to analyze the full CFG of the relevant method(s).

Fig. 11 illustrates a service loop from Java IO reported as an Observer candidate by PINOT. PINOT assumes the method $writeSerialData$ to play the Notification Method role and the method $invokeWriteObject$ to play the Update Method role. This error can be avoided by verifying that full coverage ("All observer objects are addressed

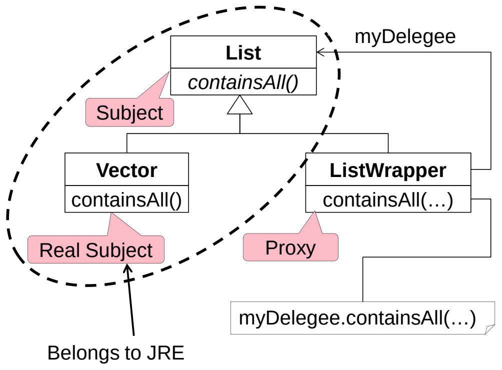


Figure 12: Role players in external library (AWT 1.14)

by the update call") is *not* fulfilled. The diverting path in $writeSerialData$ that does not reach the Update Call candidate $slotDesc.invokeWriteObject$ is depicted in Fig. 11 by the dashed-and-pointed line.

IV. Improving Recall

In general, applying strong behavioral constraints such as those mentioned in the previous section bears the risk of reducing recall. However, we have observed that most false negatives [13] produced by existing tools result from just a few general problems that will be discussed in this section:

1) Relevant role players are not found because they reside in code not included in the analysis.
2) Constraints on attribute values of role players (e.g. abstractness, visibility) are too strong.
3) Transitivity of some relations (e.g. subtyping, control or data flow) is ignored of followed insufficiently deep.

A. Relevant role players reside in missing code

In some DP candidates, the relevant role players are missed because they occur in code that is not included into the analysis, which is typical when analysing frameworks. For example, in the Proxy occurrence from JHotDraw 6.0 illustrated in Fig. 12 the program elements playing the Real Subject and Subject roles ($List$ and $Vector$) belong to the $Java.core$ libraries. SSA, Ptidej and DP-Miner do not accept candidates whose role players belong to external repositories unless the latters are compiled and their byte-code is loaded into the tool. PINOT, in contrast, accepts such a candidate by examining the textual references to external elements and assuming that the missing parts exist.

In our DPJF tool we take a mixed approach: (1) It automatically includes all *available* external dependencies (other projects, bytecode libraries) into the analysis. (2) Roles players that are still missing afterwards because of incomplete code are treated by distinguishing *optional* roles from *mandatory* roles [2]. DPJF tolerates missing players for optional roles but requires mandatory ones to be played. It accepts the occurrence from Fig. 12.

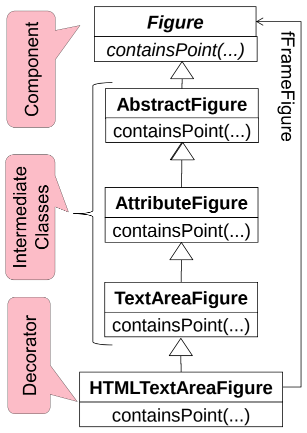


Figure 13: More than one intermediate class (JHotDraw 6.0)

```
public class Iconkit {
  private static Iconkit fgIconkit = null;
  public Iconkit(Component component) { ... }
  /** Gets the single instance */
  public static Iconkit instance() { return fgIconkit; }
...
}
```

Figure 14: Need to relax static constraints to find careless implementations: A Singleton with a public constructor (JHotDraw 6.0)

B. Too strong constraints on attribute values

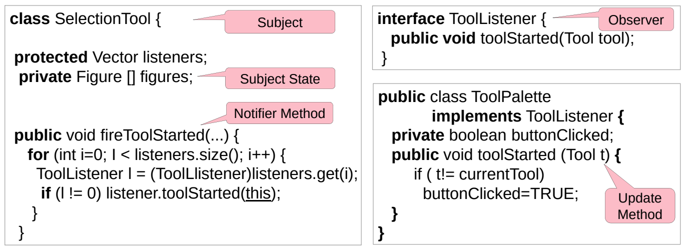
Some DPD tools impose too strong constraints on the static structure of a motif thereby missing proper occurrences. Certain constraint relaxations are already supported by SSA, Ptidej and Fujaba. They accept if roles that are supposed to be played by abstract classes or methods are played by concrete classes or methods. DPJF additionally accounts for the need to relax constraints on method visibility. For instance, the Singleton occurence shown in Fig. 14 has a *public* constructor. This is clearly against the intention of the pattern. However, a tool should not ignore such an improperly implemented occurrence (as PINOT does) but report it and give hints how to improve it. Therefore, no restrictions are placed on visibility qualifiers (private, protected or public) in DPJF.

C. Insufficient treatment of transitivity

According to our observations, most false negatives occur because DPD tools neglect the transitivity of the relations defined in II-B.

PINOT and DP-Miner [22] do not support transitivity at all. SSA, Ptidej and Fujaba tolerate a limited case of the subtype transitivity up to one intermediate class is accepted. However, scenarios like the one in Figure 13 (three intermediate classes) are already missed. To the best of our knowledge, no reviewed tool supports control flow transitivity, so occurrences like the one from Fig. 16 are rejected. From the DPD tools known to us, only D-Cubed [23] supports data flow transitivity.

To improve recall, our tool accounts for the transitivity of subtyping, control flow and data flow (Sec. II-B3). We



```
class SelectionTool {

 protected Vector listeners;
  private Figure [] figures;


 public void fireToolStarted(...) {
   for (int i=0; I < listeners.size(); i++) {
     ToolListener l = (ToolLlistener)listeners.get(i);
      if (l != 0) listener.toolStarted(this);
    }
 }
```

```
interface ToolListener {
  public void toolStarted(Tool tool);
 }
```

```
public class ToolPalette
            implements ToolListener {
   private boolean buttonClicked;
   public void toolStarted (Tool t) {
       if ( t!= currentTool)
         buttonClicked=TRUE;
   }
}
```

Figure 15: Observer with identity flow (Quick UML 2001)

note that in the presence of data flow transitivity the 'state propagation' constraint from Sec. III-C is too strong, damaging recall. Therefore it is relaxed below to the 'contents flow' constraint and further to the 'identity flow' and 'null flow' constraints.

1) Contents flow: We say that *contents flow* occurs from a subject S related to an observer O if there are fields $sstate_{1..m}$ in an object o *transitively referenced* by S[2] and fields $ostate_{1..k}$ in O or in an object transitively referenced by O and there is a dataflow from $sstate_{1..m}$ to $ostate_{1..k}$ during execution of the Notification method. In this case we accept $sstate_{1..m}$ as the ' Observer State' role players and $ostate_{1..k}$ as the the 'Subject State' role players. We thereby relax the 'State Propagation' requirement (Sec. III-C) by letting these roles be played also by fields that are not directly contained in O or S. Contents flow acknowledges that the state of an object is not represented only by its own field values but also by the values of fields in any objects referenced transitively.

2) Identity Flow and Null Flow: According to Gamma et al. [15], the Observer pattern guarantees that "all Observer objects are notified whenever the corresponding subject undergoes any state changes". Such notifications do not necessarily need to be propagated using data or contents flow. It suffices that an observer object is made aware of which subject's state changed. We call this *identity flow*. Figure 15 illustrates such a scenario from Quick UML 2001. The Subject role is played by instances of $SelectionTool$ that notify associated instances of $ToolPalette$ that play the Observer role. When a tool is started, the method $fireToolStarted$ notifies the observers by invoking the method $toolStarted(Tool)$ that it calls on each associated *ToolPalette* instance, passing itself as a parameter. A palette thus becomes aware of the start event and of its sender and updates appropriately its 'clicked' flag. We observed some proper Observer instances in which a subject notifies its observers without passing any parameters to the update method. We call this 'null flow'. Note that even these extreme relaxations still may be transitive, that is, the flow can traverse several objects.

V. Achieving Speed

To achieve high speed, we combine (A) empirically validated simplifications of traditional structural and behavioural

[2]That is, o might be stored in some field f owned either by S or by an object pointed to by f

analysis techniques, (B) caching of cheap analysis results and (C) demand-driven application of all complex analysis techniques.

A. *Empirically Validated Simplifications*

The simplifications that we apply are inspired by empirical analysis of the source code repositories of sizes and complexities that cover most practical scenarios.

1) Limit depth of transitivity: Although DPJF acknowledges transitivity of relations as the prime technique for improving recall, it limits the depth to which transitivity is explored. For instance, it explores indirect method invocation just 1 step deep (Sec. II-B3) , based on the observation that the methods that play a role in a design pattern implementation typically interact directly or via at most one helper method in 90% of all manually reviewed cases. Similarly, following subtype relations at most three steps deep covers around 80% of all cases.

2) Avoid exploring transitivity for direct invocations : We observed that if a method M calls directly a method call mc that plays some design motif role R, it is quite unlikely that M will invoke indirectly another method call that plays R. Therefore, if a direct method invocation exists, indirect invocations are not searched for at all. Since direct invocation occurs in almost 90% of all reviewed samples, this heuristic dramatically reduces the number of cases where transitivity has to be explored.

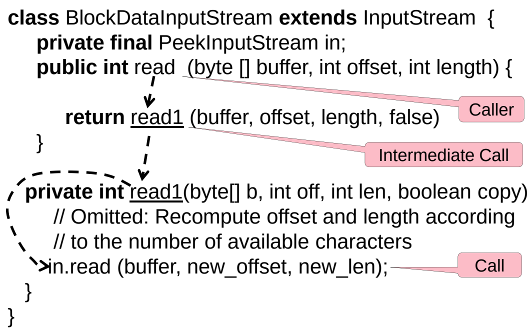
3) Simplified exploration of transitivity for intra-object control flow : A method m_1 might invoke another method m_2 indirectly, through several intermediate methods (Sec. II-B3). We talk of *intra-object control flow* if the object on which m_1 was invoked is also the receiver of all intermediate invocations. If this happens, the receiver is typically referenced by the pseudovariable *this* (in Java and C++), *current* (in Eiffel) or *self* (in Smalltalk) – or is left implicit, which is just syntactic sugar for using *this*. Figure 16 shows a CoR occurrence from Java IO that exhibits intra-object control flow. The forwarder method $public\ int\ read(...)$ invokes the forwarding call $in.read(...)$ via the intermediate method $private\ int\ read1(...)$, which is invoked on the *implicit this* receiver.

In all patterns from [15]except Visitor the control flow scope of invocations that play a role in the pattern is *intra-object*. This observation of [23], [24] lets us avoid a complex (possibly transitive) computation of the points-to set of the receiver variable. All one needs to check is whether an invocation is sent to *this* (explicitly or implicitly).

B. *Reuse of analysis results*

Reuse is achieved by breaking analyses into smaller analysis steps and caching intermediate results if there is empirical evidence that it will pay off in terms of a high reuse rate of the cached values. We organise the fine-grained analysis steps into dependency graphs [3] and sort them topologically.

[3]An entity I depends on another entity J if the results of computing J are used to compute I



```
class BlockDataInputStream extends InputStream {
    private final PeekInputStream in;
    public int read  (byte [] buffer, int offset, int length) {

        return read1 (buffer, offset, length, false)
    }
  private int read1(byte[] b, int off, int len, boolean copy) {
      // Omitted: Recompute offset and length according
      // to the number of available characters
     in.read (buffer, new_offset, new_len);
  }
}
```

Figure 16: Intra-object control flow in a Proxy (Java IO 1.4)

At the bottom there is the *subtyping* analysis and the analysis for *containment* of instance variables / methods. Since we only follow subtype relations three steps deep (see V-A1) precomputing subtyping means that for a subtyping path $a \to b \to c \to d \to e \to ...$ we generate the transitive pairs $a \to c$, $a \to d$, $a \to e$, (but not $a \to f$), then $b \to d$, etc.

Then follows the identification of *elementary design patterns* (EDPs [25], [26], [27]) that are characteristic for a motif group. EDPs are detected by simple and fast structural queries that are not satisfied by DP candidates outside the respective motif group. An EDP in our approach can be considered as a combination of several design microarchitectures referred to as *design clues* by Arcelli et al. [28].

The remaining analyses (re)use the cached results of the subtyping, containment and EDP analysis.

C. *Demand-Driven Approach*

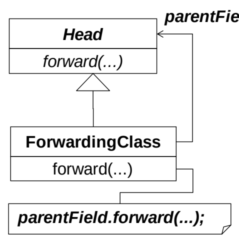
All remaining analyses are carried out on a demand-driven basis [29], [30], that is, only when necessary and only as deep as necessary to achieve a certain goal. There is no precomputation here since we found that it adversely affects performance wasting time on computing information that is never used.

VI. DPJF Implementation

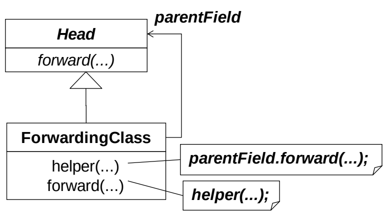
Implementation of DPJF[4] in a very short time frame was enabled by the availability of JTransformer [31], an easy to use development environment for analyses and transformations of Java source code. JTransformer deals with the parsing of source and byte-code, resolution of project dependencies and creation of an internal representation of the entire code in Prolog. Analyses can be expressed at a very high level of abstraction as Prolog rules. DPJF is implemented as a set of SWI-Prolog modules, each dedicated to the detection of candidates for a group of strucurally similar motifs. Each module performs the following steps:

1) Find all program elements that play some role in motifs of the given group.
2) Aggregate role assignments to candidates.

[4]DPJF is available at http://sewiki.iai.uni-bonn.de/research/dpd/dpjf/



(a) Direct invocation



(b) Indirect invocation

Figure 17: Transitivity check

3) Apply the behavioral constraints from Sec. III to filter out false positives and properly classify the candidates.

The second step is generic, the others have specific implementatioins in each detection module. We illustrate all steps on the example of detecting Supertype Forwarder candidates.

A. Find Role Assignments

The analysis starts by computing the subtype and containment relation. Then candidates for the Forward-to-Supertype EDP (Sec. V-B) are computed. They are stored as tuples of the form <*ForwardingClass*, *Head*, *OverridingMethod*, *Method*, *ParentField*>, where each tuple element represents the assumed player of one role in the EDP. Next, program elements that play additional roles are identified. This is done by checking for each candidate how the $OverridingMethod$ player invokes the $Method$ player. If the invocation is direct (as depicted in Figure 17a) the tuple is kept unmodified. Otherwise, the element playing the role of $IntermediateMethod$ that invokes $Method$ (see Figure 1) is identified and the candidate is extended by the role assignment for $IntermediateMethod$.

B. Aggregate Role Assignments to Candidates

Given many players of the same role one needs to distinguish players that belong to different candidates and combine players that belong to the same. Formally, a candidate is a connected component from the projection graph induced by the design pattern schema and the role assignments [14]. Informally, all players that are related belong to the same candidate. For example, assume that on some program the previous step produces the tuples

```
<Super, SubA, SubA::parent, SubA::m1(), Super::m1()>
<Super, SubA, SubA::parent, SubA::m2(), Super::m2()>
<Super, SubA, SubA::parent, SubA::m3(), Super::m3()>
<Super, SubB, SubB::super,  SubB::m1(), Super::m1()>
<Super, SubB, SubB::super,  SubB::m2(), Super::m2()>

<Sup, Sub, SubB::p, Sub::m1(), Sup::m1()>
<Sup, Sub, SubB::p, Sub::m2(), Sup::m2()>
```

Then we have two distinct instances:

- One in which the *Head* role is played by *Super* and in which there are two different players of the *Forwarding-Class* role (*SubA* and *SubB*), each having its own player of the *parentField* role (*parent* and *super*). To this instance belong also three methods in *SubA* that override methods from super and two in *SubB* that do the same.
- One that has *Sup* as its *Head* role player, *Sub* as the *ForwardingClass* role player, *p* as the *parentField* player and two pairs of overriding methods.

C. Apply Behavioral Constraints

In the final step, false candidates are eliminated and correct ones are categorized as Decorator, Dynamically Typed Proxy or Chain of Responsibility. First, TreeForwarder occurences are excluded by checking the 'Forward to a single object' constraint (Sec. III-A). Then Dynamically Typed Proxy occurrences are filtered out by verifying the Sibling Creation constraint (Sec. III-D). Finally, Decorator occurrences are distinguished from Chain of Responsibility occurrences by checking for Full or Partial Full Coverage (Sec. III-E).

VII. Evaluation

This section presents the results of evaluating the precision, recall and speed of DPJF in comparison to four other DPD tools: Fujaba [3], PINOT [4], Ptidej [5], and SSA [6]. The evaluation was carried out for the Decorator, Chain of Responsibility (CoR), Proxy, Observer and Composite motifs.

A. Benchmark projects

In [2], [32] it was observed that many DPD tools perform well on projects commonly used as reference but poorly otherwise. To avoid such overfitting, our evaluation was carried out on projects that cover a wide range of DPD-relevant scenarios (see Fig. 18 for the project sizes):

- **Known samples:** JHotDraw 5.1 and JHotDraw 6.0 are small libraries designed for teaching purposes, which contain many straightforward to detect pattern occurences that closely follow the descriptions by Gamma et al. [15]. These are included because their actual pattern occurrences are well-known.
- **Standard libraries:** Java IO 1.4 and Java AWT 1.3 are example of widely-used, industrially developed libraries that are indispensable in any non-trivial application. Their particular challenge to DPD tools are apparently incomplete pattern occurrences, where some role players are completed by the applications using the libraries.
- **Large applications:** ArgoUML 0.18.0 is a large application implementing an open source CASE tool. Being self-contained it does not impose the incompleteness problem of libraries. However, its size is a challenge for many tools.
- **Implementation variants:** TeamCore is a component of the Eclipse 3.6 rich client platform, a large, quickly evolving industrial application containing very unusual DP implementation variants.

B. Computing precision and recall

The reviewed tools compute their precision and recall based on the number of found candidates. That is, if in a given candidate C some classes playing mandatory roles are properly recognized, C is accepted. This is sometimes a too

	JHotDraw 5.1	JHotDraw 6.0.1	Java IO 1.4	AWT 1.3	ArgoUML 0.18.0	Eclipse 3.6 TeamCore
packages	64	84	30	76	200	41
classes	1.002	1.732	517	1.653	3.776	702
fields	3.188	4.518	1.062	6.123	8.518	1.210
methods	8.793	14.975	4.471	14.618	31.577	5.514

Figure 18: Size of analysed projects including their source code *and* all the referenced bytecode.

coarse grained metric, since every candidate and occurrence of a design motif consists also of various program elements (classes, fields, methods, individual statements) that play roles in the motif. Some role players might

- be wrongly identified, leading to *role-level false positives*. For example, in some Decorator candidates certain methods forward conditionally, thus being Chain of Responsibility forwarder methods. However, DPD tools that do not employ sufficient behavioral analyses wrongly identify these methods as playing the Decorator Forwarding Method role.
- not be identified (although the tools search for players of that role), leading to *role-level false negatives*. For example, method role players that invoke relevant method calls *indirectly* are not recognized by some of the reviewed DPD tools.

To deal with the above mentioned problems, we use role-based precision and recall computation, motivated by the work of Pettersson et al. [33]. It is based on the notion of individual precision and recall for each candidate *C*. The *individual precision* for *C* is the number of correctly recognized role players in C divided by the number of role players reported for *C*. The *individual recall* for *C* is the number of correctly retrieved role players in C divided by the total number of actual role players in *C*. When computing scores this way, we only include players of class and method roles since these are reported by almost all compared DPD tools[5]. The *precision* of a DPD tool on a particular program is obtained by dividing the sum of individual precisions of all correctly identified candidates by the number of reported candidates. The *recall* of the tool is obtained by dividing the sum of individual recalls of all reported candidates by the number of occurrences in the program.

Formally, let C_T be a (possibly incomplete and partly wrong) candidate retrieved by tool T and O be the corresponding complete and correct occurrence that an ideal tool would report (if C_T is a false positive then O will be empty). Then we define

- $reported(C_T)$ is the number of role assignments in C reported by T
- $detected(C_T)$ is the number of correct role assignments in C reported by T
- $correct(C_T)$ is the maximum number of correct role assignments in O for the role types supported by T (e.g. if T is not able to report statement-level roles, then only the correct assignment to class, method and field level roles in O are counted).
- $prec(C) = detected(C)/reported(C)$ is the *individual precision* for C_T
- $rec(C) = detected(C)/correct(C)$ is the *individual recall* for C_T

If T reports n candidates $C_T^1 ... C_T^n$ for a program that contains m proper occurrences of some motif M then:

- $precision = \left(\sum_{i=1..n} prec(C_T^i)\right)/n$ is the average precision of T for M.
- $recall_T = \left(\sum_{i=1..n} rec(C_T^i)\right)/n$ is the average recall of T for M.

DPJF computes the tool-specific precision and recall per motif and project for its own candidates and for those of all other evaluated tools For each motif, the results are shown in one of the tables of Fig.19. We aggregated the values for different projects to an *average* per pattern and tool, computed as if all tested projects were a single big one. The average values are shown in the “Total” Column of each table of Fig. 19. For easier reference and comparison they are summarized (together with their medians) in Fig. 20a, Fig. 20b and depicted graphically in Fig. 20c and Fig. 20d.

C. Achieved Precision

Fig. 20a and 20c show that the average precision of the previously existing tools is highly dependent on the pattern and benchmark, with medians ranging from 10% for Pinot to 66% for SSA. In contrast, DPJF achieves 100% precision for each evaluated pattern and benchmark.

Obviously, the constraints supported only by DPJF effectively discard all false positives produced otherwise.

D. Achieved Recall

Fig. 20d and Fig. 19 show that DPJF has the best recall among all evaluated tools. The median of DPJF’s recall is 89%, which is is 2 to 4.5 times better than medians of 18% for Ptidej to at most 44% for SSA. The differences are even more pronounced when comparing individual recalls. We conclude that the recall-improving techniques implemented in DPJF are very effective too. The main influencing factor for the recall is the exploration depth of transitive relations.

E. Achieved Speed

All speed measurements where carried out on a Dell Precision 370 desktop computer equiped with a 3,6 GHz quadcore CPU and 2 GB of RAM running Windows XP SP 3, Eclipse 3.7, SWI-Prolog 5.11.29 and JTransformer 2.9.3. The response times were measured automatically for the tools for which we have source code (DPJF, Ptidej and SSA) and manually for PINOT and Fujaba.

[5]The only exception is Ptidej, for which we only compare class roles since it does not support others. Thus Ptidej is slightly favoured, since it is not penalized for not identifying players of method and field roles, whereas tools that support these roles are penalized when they *misrecognize* players.

Decorator	JHD 5.1 (3)				JHD 6 (10)				JavaIO (4)				AWT (9)				ArgoUML (3)				TeamCore (2)				Total (31)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	0	---	0	---	5	59	26	36	4	78	68	72	3	96	33	49	1	100	33	50	1	100	22	35	14	78	31	45
PTIDEJ	6	17	33	22	8	25	20	22	3	33	17	22	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	17	24	12	16
SSA	3	67	67	67	8	60	50	54	2	83	23	35	7	76	53	63	1	100	33	50	0	---	0	---	21	70	44	54
DPJF	3	100	100	100	9	100	90	95	4	100	93	96	8	100	89	94	3	100	100	100	1	100	50	67	28	100	89	94

(a) Results for Decorator (Fujaba is not shown since it does not detect the pattern)

Chain of Responsibility	JHD 5.1 (0)				JHD 6 (3)				JavaIO (2)				AWT (4)				ArgoUML (0)				TeamCore (3)				Total (12)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	0	---	---	---	1	0	0	---	2	25	35	29	4	55	54	54	1	0	---	---	0	---	0	---	8	34	24	28
PTIDEJ	2	0	---	---	4	25	33	29	3	67	100	80	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	9	33	18	23
DPJF	0	---	---	---	3	100	100	100	2	100	100	100	4	100	100	100	0	---	---	---	2	100	67	80	11	100	92	96

(b) Results for Chain of Responsibility (Fujaba and SSA are shown since they do not detect the pattern)

Proxy	JHD 5.1 (2)				JHD 6 (2)				JavaIO (7)				AWT (6)				ArgoUML (9)				TeamCore (0)				Total (26)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	2	0	0	---	11	9	50	15	3	67	23	34	10	10	17	13	10	10	11	11	0	---	---	---	36	14	18	16
PTIDEJ	1	0	0	---	1	0	0	---	0	---	0	---	💣	💣	💣	💣	💣	💣	💣	💣	0	---	---	---	2	0	0	---
SSA	0	---	0	---	0	---	0	---	6	67	33	44	2	100	33	50	2	100	22	36	0	---	---	---	10	80	24	37
DPJF	2	100	100	100	2	100	100	100	7	100	100	100	6	100	100	100	8	100	89	94	0	---	---	---	25	100	96	98

(c) Results for Proxy (Fujaba is not shown since it does not detect the pattern)

Observer	JHD 5.1 (2)				JHD 6 (6)				JavaIO (0)				AWT (2)				ArgoUML (5)				TeamCore (2)				Total (17)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
Fujaba	19	11	100	19	21	14	50	22	💣	💣	💣	💣	💣	💣	💣	💣	23	13	60	21	14	0	0	---	77	10	47	17
PINOT	1	0	0	---	2	100	33	50	6	0	---	---	9	2	10	4	6	0	0	---	0	---	0	---	24	9	13	11
PTIDEJ	17	12	100	21	3	33	17	22	0	---	---	---	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	20	15	18	16
SSA	3	33	50	40	7	76	100	87	0	---	---	---	2	40	40	40	1	100	20	33	0	---	0	---	13	63	52	57
DPJF	1	100	50	67	6	100	100	100	0	---	---	---	1	100	50	67	3	100	60	75	0	---	0	---	11	100	65	79

(d) Results for Observer

Composite	JHD 5.1 (1)				JHD 6 (1)				JavaIO (0)				AWT (2)				ArgoUML (2)				TeamCore (0)				Total (6)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
Fujaba	1	100	100	100	13	0	0	---	💣	💣	💣	💣	💣	💣	💣	💣	2	50	50	50	7	0	---	---	23	9	33	14
PINOT	1	0	0	---	1	0	0	---	2	0	---	---	3	33	50	40	10	20	100	33	0	---	---	---	17	18	50	26
PTIDEJ	3	33	100	50	5	20	100	33	0	---	---	---	💣	💣	💣	💣	💣	💣	💣	💣	2	0	---	---	10	20	33	25
SSA	1	60	60	60	2	22	100	35	0	---	---	---	0	---	0	---	1	100	50	67	0	---	---	---	4	51	43	47
DPJF	1	100	100	100	1	100	100	100	0	---	---	---	1	100	50	67	1	100	50	67	0	---	---	---	4	100	67	80

(e) Results for Composite

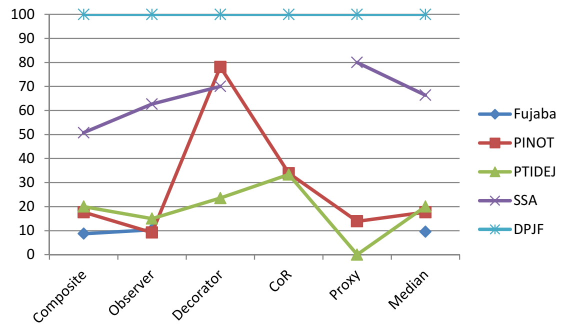
Figure 19: Evaluation results per pattern. Column headings indicate the analysed project and (in brackets) the number of known occurrences for that project. The "Total" column indicates the performance when all projects are taken as a single big one. Precision and recall are computed as described in Sec.VII-B. A bomb indicates that the tool crashed on that project. A precision (resp. recall) value that cannot be computed because there are zero occurrences (resp. candidates) is indicated by '—'.

Precision	Composite	Observer	Decorator	CoR	Proxy	Median
Fujaba	9	10	X	X	X	10
PINOT	18	9	78	34	14	18
PTIDEJ	20	15	24	33	0	20
SSA	51	63	70	X	80	66
DPJF	100	100	100	100	100	100

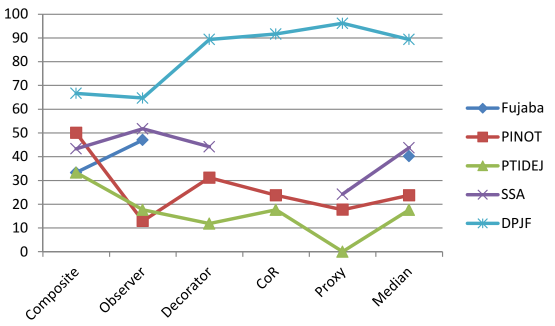
(a) Precision

Recall	Composite	Observer	Decorator	CoR	Proxy	Median
Fujaba	33	47	X	X	X	40
PINOT	50	13	31	24	18	24
PTIDEJ	33	18	12	18	0	18
SSA	43	52	44	X	24	44
DPJF	67	65	89	92	96	89

(b) Recall



(c) : Precision (graph for Fig. 20a)



(d) Recall (graph for Fig. 20b)

Figure 20: Precision and recall on all benchmark projects. The inability of a tool to detect a pattern is indicated by an 'X' in the table and a hole in the graph

Load + Query	All evaluated patterns					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Fujaba	60,0	180,0	💣	💣	600,0	50,0
Pinot (all)	**1,0**	**6,0**	**2,0**	17,0	**55,0**	4,0
Ptidej	39,3	99,2	9,0	💣	💣	14,7
SSA	1,6	9,3	3,0	**13,1**	55,1	**1,3**
DPJF	5,2	18,0	7,0	54,8	77,0	8,0

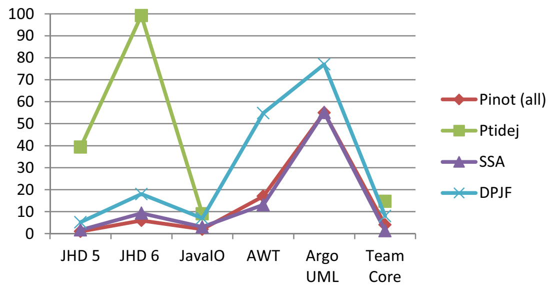
(a) Time for initialisation and analysis

Initialisation time (seconds)	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej	2,3	**4,2**	2,9	💣	💣	4,9
SSA	**0,9**	5,2	**1**	**2**	**15**	**0,7**
DPJF restart	4	14	5	40	56	6
DPJF first start	35	150	49	390	550	60

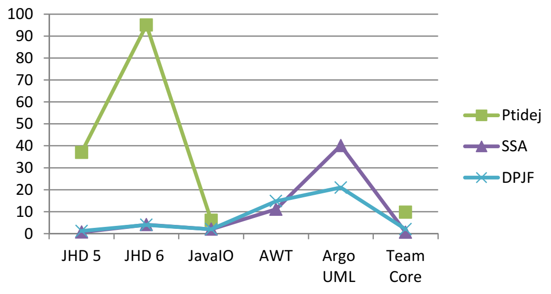
(b) Initialization time

Query	All evaluated patterns					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej	37,0	95,0	6,1	💣	💣	9,8
SSA	**0,7**	4,1	**2,0**	**11,1**	40,1	**0,6**
DPJF	1,2	**4,0**	**2,0**	14,8	**21,0**	2,0

(c) Pure analysis time



(d) Time for initialisation and analysis



(e) Pure analysis time

Figure 21: Response times of each tool for detecting all evaluated patterns. In the tables, the fastest time per project is red and a bomb indicates that the tool crashed on that project. In the graphs, Fujaba is not shown since it is an order of magnitude slower than most other tools, which would have distorted the graphs. Pure analysis time is indicated only for the tools for which it can be measured separately.

Total run-time: Fig. 21a and 21d show the total time that a tool needs for one run. This includes initialisation (parsing of the project, creation of the internal representation) and analysis time (searching for pattern candidates). The figures show that PINOT and SSA are the two fastest tools. However, PINOT detected 17 patterns[6] while SSA and all the other tools where selectively run on just 5 patterns. Thus, when running for the exactly same set of patterns, PINOT is presumably the fastest tool on all the benchmarks.

DPJF comes third, clearly outperforming Fujaba and Ptidej. This is remarkable, given its significantly better precision and recall. This might be an indication of the immaturity of the employed base technologies: Fujaba uses graph transformation and Ptidej explanation based constraint solving. Both technologies are much younger than Prolog and are apparently still lacking compilers and run-time systems that could compete with top Prolog compilers. If we therefore disregard Fujaba and Ptidej, the third position of DPJF after Pinot and SSA *appears* to confirm the common wisdom that improving precision and recall adversely affects speed.

Startup time: We wanted to know whether the above result also holds when differentiating the speed of initialisation and analysis. Fig. 21b shows the initialisation time of DPJF, SSA and Ptidej[7]. Comparison with Fig. 21a reveals that most of SSA's and DPJF's run-time is spent on initialisation, unlike for Ptidej. For DPJF we distinguish two startup times: Upon *first start* on a project, the internal representation is generated. This requires much time. When Eclipse is shut down, the internal representation of DPJF is cached so that every future *restart* is significantly faster (compare the rows 'DPJF restart' and 'DPJF first start'). Still, even the restart time of DPJF is slower by a factor of 3 to 20 than the startup time of SSA. This is because SSA is a specialized tool that only creates an internal model for the program elements that it needs, whereas DPJF builds on JTransformer, which always creates a full representation of the entire program (including referenced byte code and even comments) since it cannot know what an analysis application really needs.

Query time: Fig. 21c and 21eshows the pure query times for DPJF, SSA and Ptidej. The results are surprising, at least. The results clearly refute the conjecture that better quality implies lower speed: Obviously, DPJF is closely competing with SSA for the shortest query time, although in half of the cases SSA is still faster.

Query time per pattern group: Aiming to further refine our findings, Fig. 22 compares the query-time of Ptidej, SSA and DPJF when run selectively for individual pattern groups. The numbers indicate the time for detecting all candidates of all patterns in the respective group on a given project. Again, there is no clear winner. For Observer and Composite, there are two benchmarks on which DPJF is the fastest, two on which SSA is the fastest and two on which both are equal. When detecting supertype forwarders, SSA is the fastest on three

[6]PINOT cannot be run selectively just for certain patterns.

[7]We are indebted to the authors for providing us with the source code and the necessary assistance.

Query	CoR + Decorator + Proxy						Observer + Composite					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej	6,0	25,0	4,0	💣	💣	2,7	31,0	70,0	2,1	💣	💣	7,1
SSA	**0,2**	1,1	**1,0**	**1,0**	**5,1**	**0,5**	**0,5**	**3,0**	**1,0**	10,1	35,0	**0,1**
DPJF	**0,2**	**1,0**	**1,0**	8,0	10,0	1,0	1,0	**3,0**	**1,0**	**6,8**	**11,0**	1,0

Figure 22: Query times by pattern group. Shortest time is red.

benchmarks, DPJF on one, and on one both are equally fast. The detail interpretation of these observations, relating them to the employed technologies and the size / specific structure of the analysed projects, is subject of ongoing work.

Incremental update: DPJF / JTransformer supports incremental update: If a file is changed at run-time, only the related part of the factbase is updated, with no noticeable delays. In all but its very first run during an Eclipse session DPJF will therefore incur only the *pure query time* from Fig. 21c. To the best of our knowledge, none of the other tools supports incremental update. They all need to re-parse the entire project for each new analysis, incurring on *each* run the total run-time from Fig. 21a.

Comparing these times we may conclude that DPJF has achieved a substantial advance in the state of the art of design pattern detection by improving precision and recall while keeping up with the fastest existing tools.

F. Threats to Validity

Internal threats : Striving to avoid overfitting by including non-standard projects into our evaluation implies that there are no reference evaluations of these projects yet. So other researchers might disagree with our judgements. To tackle this threat DPJF is publicly available (see VI) so that everyone can use it an validate its results.

External threats : In our experiments, Fujaba and Ptidej crashed on some projects (see bombs in the tables of this section) yielding error messages or throwing exceptions, which we reported to the tool authors. For Fujaba, which does not detect the motifs from the Supertype Forwarder group, the crashes reduced the number of successful runs to just 4 for Observer and 4 for Composite, wakening the emprirical basis for this tool.

VIII. Beyond Traditional DPD

Advisory DPD: Design pattern detection is traditionally geared towards occasional exploration of unknown applications. DPJF, goes one step beyond. Taking advantage of its high query speed and incremental factbase update, DPJF enables (re)running design pattern detectors after every change of a file, acting as a permanently active development assistant that hints at wrong, incomplete or unrecommended implementations of a design pattern. We call this *advisory DPD*. For instance, DPJF detects Singletons whose constructor is public (as the one shown in Fig. 14). Detected problems are reported in DPJF's Result View and as warning markers in the editors of affected classes.

Design Pattern Improvement (DPI): DPJF exploits the versatility of its underlying infrastructure, JTransformer, to go yet one step further. Instead of just pointing out problems, it also suggests solutions and even provides *automatic corrections* via Eclipse 'quick fixes'. Thus it pioneers the step from automated design pattern detection to automated design pattern improvement (DPI). Please note that DPI is not just about enabling automated corrections but the sum of all the improvements achieved by DPJF. In particular, 100% precision is an essential prerequisite for developers accepting change suggestions given by a tool.

IX. Conclusions and Future Work

In this paper we have shown that design pattern detection can achieve ideal precision and, compared to existing tools, several factors of improvement in recall. By thorough empirical evaluation we identified behavioural characteristics of design motifs that were not captured previously. From these characteristics we distilled a set of new constraints that eliminate all false positives reported by other tools. False negatives are largely avoided by relaxations of other constraints and, in particular, deeper exploration of transitive relationships. However, transitivity is not unbounded but subject to empirically motivated depth limits. Together with a fine grained caching strategy, it makes the query time of our tool, DPJF, faster than several of the existing ones, in spite of ideal precision and significantly enhanced recall.

Beyond the techniques for improving precision, recall and speed, we advanced the state of the art by a methodological contribution: A more precise computation for precision and recall that assigns individual precisions and recall values to each reported candidate by counting its (reported and correct) role assignments.

Our current work focuses on interfacing DPJF to the public DPB design pattern benchmark repository [**?**], [34]. To guarantee that the recall values computed by DPJF are based on validated test sets we are working on automated comparions of DPJF's results to the results from DBP (which includes those from P-Mart [35]). In addition, we will implement converters of our data to DBP format, so that other researchers can easily explore, verify and annotate DPJF results even without using DPJF. Mid- and long-term continuations of our research include:

- Extending the set of supported design motifs.
- Exploring further trade-offs between precision, recall and speed.
- Investigating the usage of auxiliary information (program element names, bad smells etc) witnessing the existence of design patterns. This is especially useful when we detect patterns whose solutions do not have unique structural or behavioral characteristics - such as Visitor or Observer. For example, the notion of "object state" exploited in Observer is quite domain-specific.

Acknowledgements

We are indebted to Yann Gaël-Guéhéneuc, Nikolaos Tsantalis, Alexander Chatzigeorgiou, Péter Hegedüs, Lajos Jenö Fülöp and Rudolf Ferenc for long discussions on the details of their tools and approaches; Tobias Rho and the JTransformer team for all their support and in particular Fabian Noth and Andreas Becker for implementing the JT Control Center and Result View (which became the main user interface of DPJF) and for a significant share of the points-to analysis library used in DPJF; Daniel Speicher and Jan Nonnen for the Cultivate Java analysis API used in DPJF; Daniel Speicher, Holger Mügge, Jan Nonnen for many discussions on their own work and its relation to DPD; Andrea Caracciolo and Francesca Arcelli for providing us with an XML dump of the DPB database, explanation of the underlying meta-model and extension of DBP by data export facilities requested by us; Mokhtar Abdu for conversion of DBP's XML dump to Prolog facts.

The research of the first author was generously supported by a PhD grant from Deutscher Akademischer Austauschdienst (DAAD).

References

[1] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *IJSEKE*, 2008.

[2] G. Kniesel and A. Binun, "Standing on the shoulders of giants – a data fusion approach to design pattern detection," in *ICPC 2009*, A. Marcus and R. Koschke, Eds. IEEE, 2009.

[3] M. von Detten and M. Platenius, "Improving dynamic design pattern detection in reclipse with set objects," in *Proceedings of the 7th International Fujaba Days*, 2009.

[4] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *ASE'06*. Washington, USA: IEEE Computer Society, 2006, pp. 123–134.

[5] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *CASCON'04*. IBM Press, 2004, pp. 28–41.

[6] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE TSE*, vol. 32, no. 11, pp. 896–909, 2006.

[7] H. Albin-Amiot and Y.-G.Guéhéneuc, "Meta-modeling design patterns: application to pattern detection and code synthesis," in *Proceedings of First ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[8] H. Zhu and M. Zhou, "Role-based collaboration and its kernel mechanisms," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 36, pp. 578–589, 2006.

[9] S. Neal and P. F. Linington, "Tool support for development using patterns," in *EDOC '01: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*. Washington, DC, USA: IEEE Computer Society, 2001, p. 237.

[10] D.-K. Kim, R. France, S. Ghosh, and E. Song, "A role-based metamodeling approach to specifying design patterns," in *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*. Washington, DC, USA: IEEE Computer Society, 2003, p. 452.

[11] D. Riehle, "Describing and composing patterns using role diagrams," in *Proceedings of the Ubilab Conference, 1996*. Zuerich, Switzerland: Universitaetsverlag Konstanz. Pages 137-152, 1996, p. 452.

[12] G. Kniesel, A. Binun, P. Hegedus, L. J. Fülöp, N. Tsantalis, A. Chatzigeorgiou, and Y.-G. Guéhéneuc, "A common exchange format for design pattern detection tools," CS Department III, Uni.Bonn, Germany, Technical report IAI-TR-2009-03, ISSN 0944-8535, Oct. 2009. [Online]. Available: https://sewiki.iai.uni-bonn.de/research/dpd/dpdx/

[13] B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2009.

[14] G. Kniesel, A. Binun, P. Hegedus, L. J. Fülöp, A. Chatzigeorgiou, Y.-G. Guéhéneuc, and N. Tsantalis, "DPDX–towards a common result exchange format for design pattern detection tools," in *CSMR*, R. Capilla, R. Ferenc, and J. C. Dueñas, Eds. IEEE, 2010, pp. 232–235.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1994.

[16] J. K.-Y. Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioral and creational design patterns through dynamic analysis," *JSME*, vol. 22, no. 8, 2010, submitted.

[17] L. Wendehals, "Struktur- und verhaltensbasierte entwurfsmustererkennung," PhD thesis, Universität Paderborn, Institut für Informatik, September 2007.

[18] L. Wendehals and A. Orso, "Recognizing behavioral patterns at runtime using finite automata," in *WODA'06*. New York, USA: ACM, 2006, pp. 33–40.

[19] L. Wendehals, "Improving design pattern instance recognition by dynamic analysis," in *WODA'03*. Portland, USA: IEEE Computer Society, 2003.

[20] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2000, pp. 281–293.

[21] L. J. Fulop, R. Ferenc, and T. Gyimothy, "Towards a benchmark for evaluating design pattern miner tools," in *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–152.

[22] J. Dong, D. S. Lad, and Y. Zhao, "DP-Miner: Design pattern discovery using matrix," in *ECBS'07*. Washington, USA: IEEE Computer Society, 2007, pp. 371–380.

[23] K. Stencel and P. Wegrzynowicz, "Detection of diverse design pattern variants," in *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–32.

[24] P. Wegrzynowicz and K. Stencel, "Towards a comprehensive test suite for detectors of design patterns," in *Proceedings of the ASE 2009, Auckland, New Zealand*, Nov 2009.

[25] J. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code," in *ASE'03*. IEEE, 2003.

[26] J. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code," The University of North Carolina, CS Department, Technical report TR03-016, May 2003.

[27] J. Smith and D. Stotts, "Elemental design patterns and the rho-calculus: Foundations for automated design pattern detection in SPQR," The University of North Carolina, CS Department, Technical report TR03-032, Sep. 2003.

[28] F. A. Fontana, S. Maggioni, and C. Raibulet, "Design patterns: a survey on their micro-structures," *Journal of Software Maintenance and Evolution: Research and Practice*, 2011. [Online]. Available: http://dx.doi.org/10.1002/smr.547

[29] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," *SIGPLAN Not.*, vol. 41, no. 6, pp. 387–400, 2006.

[30] M. Sridharan, "Refinement-based program analysis tools," Ph.D. dissertation, EECS Department, University of California, Berkeley, Oct 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-125.html

[31] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," in *Proceedings of LATE '07*. New York, USA: ACM, 12th March 2007.

[32] G. Kniesel and A. Binun, "Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection," CS Department III, Uni.Bonn, Germany, Technical report IAI-TR-2009-02, ISSN 0944-8535, Jan. 2009. [Online]. Available: https://sewiki.iai.uni-bonn.de/research/dpd/fusion/

[33] N. Pettersson, W. Lowe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 575–590, July 2010. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.92

[34] F. Arcelli, M. Zanoni, and A. Caracciolo, "A benchmark platform for design pattern detection," in *2nd International Conference on Pervasive Patterns and Applications (PATTERNS'10)*, 2010, Lisbon, Portugal, November 21-26.

[35] P-Mart homepage. [Online]. Available: http://www.ptidej.net/downloads/pmart/