Design Pattern Detection using FINDER

Haneen Dabain
York University
Toronto, Ontario, Canada
haneend@cse.yorku.ca

Ayesha Manzer
York University
Toronto, Ontario, Canada
manzer@cse.yorku.ca

Vassilios Tzerpos
York University
Toronto, Ontario, Canada
bil@cse.yorku.ca

ABSTRACT

This paper introduces the FINDER tool that detects 22 out of 23 GoF design patterns in Java systems using fine-grained static analysis. FINDER extracts static facts from the system at hand. These facts are then evaluated against the design pattern detection scripts to produce a list of design pattern candidates. The tool has been evaluated by applying it to several open source systems. In this paper, we present one such case study that compares FINDER to other existing design pattern detection tools.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Software Architecture, Framework, and Software Design Patterns

General Terms

Design

Keywords

design pattern detection; fine-grained static analysis

1. INTRODUCTION

The usage of design patterns in software architecture helps system developers and maintainers understand the system implementation and the rationale behind the decisions made when designing the system. Therefore, various reverse engineering design pattern detection approaches have been developed [8, 11, 5, 16, 9, 3, 12, 14], and many automatic tools that are able to recover design patterns have been implemented and made available to developers to help them comprehend large software systems. Although different tools may recover design patterns from source code, there are discrepancies in these results. Tools may recover the same design pattern, but different numbers or participants of the same design patterns are recovered. One of the reasons for these discrepancies is that different design pattern detection tools use different definitions. Moreover, design pattern definitions in existing approaches use different notations to describe the properties of a design pattern implementation. All these factors make it hard to precisely compare between different tools.

In this paper, we present FINDER, a tool that attempts to address some of these issues. Section 2 presents some background in the research area of design pattern detection. Our approach is presented in Section 3. An evaluation case study is presented in Section 4. Finally, Section 5 concludes the paper.

2. RELATED WORK

Fabry and Mens [6] proposed a language independent approach for object oriented design patterns through the use of logic meta programming. Albin-Amiot et al. presented a set of tools and techniques to solve the problem of automating the instantiation and the detection of design patterns. Different approaches have been proposed to statically analyze software systems. Systems could be parsed to another form such as graphs. The new form usually consists of entities, attributes, and relations that need to be examined by tools to detect design patterns. For instance, Czibula and Serban [4] proposed an original clustering-based approach for identifying instances of design patterns. Moreover, the main idea in the study conducted by Kaczor et al. [12] was to convert the program to a string representation. Antoniol et al. [1] presented a conservative approach based on a multi-stage reduction strategy using OO software metrics and structural properties to extract structural design patterns from OO design or code. Another study that focuses on enhancing systems scalability is a study conducted by Gueheneuc et al. [10].They pursued their work by creating a benchmark for comparing and evaluating design pattern miner tools in [7].An interesting study was conducted by Kniesel and Binun [13]. The study combined five design pattern detection tools and proposed an approach called *Data Fusion* for combining their results in order to identify patterns not detected by individual tools.

3. THE FINDER APPROACH

In this section, we describe the implementation of our approach (called FINDER) to design pattern detection. An implementation of the FINDER approach can be found at `https://wiki.eecs.yorku.ca/project/dpd`.

We start by discussing supporting software used to implement our approach. Next, we present the implementation of our approach. We show how our tool works and the stages for detecting design patterns. Then we explain the design of

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695900 ...$15.00.

the tool. Finally, we provide an example of how to use this tool.

3.1 Supporting Software

The Software Architecture Group at the University of Waterloo (SWAG) has been at the forefront of software architecture and software engineering research in Canada since its inception. Primary research interests of the SWAG members are software architecture and its evolution through time, methods and applications of reverse engineering as applied to software, and development of fact extraction, analysis and visualization tools. Our approach extensively used three tools developed by SWAG: Javex, Grok, and QL. Javex extracts low-level facts from Java programs, while Grok and QL manipulate these facts to produce higher-level facts, and ultimately discover design pattern instances.

3.2 Approach Overview

Our implementation consists of four stages. First, we extract static relations from Java class files using Javex and QL. The second stage parses Java source files using a tool called VariableExplorer that we developed to extract additional relations that are not available in the first stage, such as method invocation on collections. In the third stage, we use QL to integrate the facts from the first two stages, and unify them in one factbase file. Finally, we filter the factbase file for candidates that satisfy the static definition of design patterns that we want to detect. We discuss each of these stages in detail next.

3.3 Stage 1: Static Fact Extraction

The static fact extraction is done in two steps. First, we generate static facts from Java class files using Javex. Second, we use QL to manipulate the factbase created by Javex and extract a set of relations that represent the structure of the program and the interaction between its elements. Following is a detailed description of these steps.

3.3.1 Static Fact Generation

For this step, we run Javex passing all Java class files as arguments to it (our implementation requires that the system at hand is already compiled). Javex produces a factbase that contains information about class hierarchies and attributes, class methods and fields, class interaction, and methods signature. Javex produces a large set of binary relations.

3.3.2 Static Relation Extraction

For this step, we use QL to manipulate the factbase produced by Javex and extract a set of relations that represent the structure of the program. This step manipulates the information produced in the first step to produce binary relations about class methods and fields, method invocation and signature, fields access types, class inheritance and interaction, and object instantiation.
Class methods and variables may not be defined in the class itself, they may rather be inherited from a superclass. For example, if class A contains a field F, this field may be declared in class A or inherited from a superclass of A. The same applies to class methods: if class A contains a method M, this method may be declared in class A or inherited from a superclass of A. The factbase produced by Javex includes contain facts only for the class declaring the method. Since design pattern detection often relies on inherited methods, this step adds contain facts for all the subclasses of a class declaring a method or a field.
In addition, a method may directly or transitively create a new object. For example, method M1 creates a new object of class C, or calls a method M2 that creates a new object of class C. Our tool supports the detection of transitive object creation. This is done by following the path of method calls that lead to an object creation, and updating all methods that are located in the path with the information about the object creation. The same technique is applied on method invocation. A method can directly or transitively invoke another method. All methods that are located on the path of invoking another method will be updated with the information about the invocation of this method.
The output of this step is static information that represents higher level facts than the ones produced by Javex. Table 1 shows the binary relations produced by QL.

3.4 Stage 2: Field Type Probing

For this stage, we developed a tool called VariableExplorer. VariableExplorer aims to find additional information about the type of objects contained in collections, as Javex fails to provide such information. It collects information about methods invoked on fields, and the parameters sent to those methods. It also collects information about field types. This information is later used to estimate the type of objects in collections.

Java source files for the system being examined are required for VariableExplorer. This tool consists of two parts: The first parses the program to an AST. The second traverses the AST to extract facts about method invocations. The traversal is performed using a visitor class called VariableExplorerVisitor in VariableExplorer. VariableExplorerVisitor is an extension to the ASTVisitor class of Eclipse which provides a visitor for abstract syntax trees. VariableExplorerVisitor visits the AST nodes in the provided AST. For this stage, VariableExplorerVisitor visits MethodInvocation nodes. A MethodInvocation node represents a method invocation expression. Below is an example of a MethodInvocation node:

```
FieldA.MethodA(FieldB)
```

FieldA represents the node expression, MethodA represents the method invoked on FieldA, and FieldB represents the node argument.

VariableExplorer produces a factbase specifying information about fields and their invoked methods. The output is a ternary relation called MethodInvocation. Below is an example of a MethodInvocation relation:

```
MethodInvocation ClassC FieldA FieldB
```

This means that ClassC invokes a method on FieldA passing FieldB as a parameter

3.5 Stage 3: Static Facts Refinement and Integration

For this stage, we use QL to integrate the static facts produced in the first stage with the facts produced in the second stage by VariableExplorer.

Assume that we have the following Java code:

Table 1: **Binary relations produced by QL**

Relation Name	Example	Explanation
class_abstract	class_abstract ClassA	ClassA is defined as abstract or as an interface.
class_concrete	class_concrete ClassA	ClassA is not defined as abstract or as an interface.
private	private ClassA	ClassA has a private constructor.
uses	uses ClassA ClassB	ClassA contains a reference to, or interacts with ClassB. Both direct and transitive usage between classes are included.
inherits	inherits ClassA ClassB	ClassA inherits from ClassB. Both direct and transitive inheritance are included.
implements	implements ClassA InterfaceB	ClassA implements InterfaceB.
class_contain_class	class_contain_class ClassA ClassB	ClassA contains ClassB. Both direct and transitive containment between classes are included.
class_variables	class_variables ClassA FieldA	ClassA contains a field FieldA. FieldA may be inherited from a superclass, or declared in the ClassA itself.
variable_descriptors	variable_descriptors FieldA ClassA	FieldA declaration type.
variable_static	variable_static FieldA	FieldA is static.
variable_private	variable_private FieldA	FieldA is private.
class_methods	class_methods ClassA MethodA	ClassA contains MethodA. MethodA may be inherited from a superclass, declared in the ClassA itself, or both in case of method overriding.
method_static_constructors	method_static_constructors MethodA	MethodA is a static constructor.
method_public	method_public MethodA	MethodA has a public, protected, or default access type.
method_private	method_private MethodA	MethodA has a private access type.
method_static	method_static MethodA	MethodA is static.
method_abstract	method_abstract MethodA	MethodA is an abstract method.
method_parameters	method_parameters MethodA ClassA	ClassA is passed as parameters to MethodA.
method_variables	method_variables MethodA VariableA	MethodA contains a local variable VariableA.
method_return	method_return MethodA ClassA	MethodA returns ClassA.
method_invoke	method_invoke MethodA MethodB	MethodA invokes MethodB. Both direct and transitive method invocation are included.
method_new	method_new MethodA ClassA	MethodA creates a new ClassA. Both direct and transitive object creation are included.
method_override	method_override MethodA MethodB	MethodA overrides MethodB. Both direct and transitive overriding of methods are included.

```
public class ClassA{
    ClassB variable1;
    List<ClassB> variable2;
}
```

This example represents a class ClassA that contains two variables: variable1 and variable2. variable1 is of type ClassB. variable2 is a strongly-typed list of type ClassB. The first stage will produce the following facts about ClassA:

```
class_variable ClassA variable1
class_variable ClassA variable2
variable_descriptors variable1 ClassB
variable_descriptors variable2 Collection
```

Note that the first stage is not able to capture the type of the objects in variable2. Instead, variable2 is expressed as a generic collection.
Next, we investigate the method invocations on generic collections to estimate the type of the objects in the collection. This is done by using the MethodInvocation relation provided by VariableExplorer. Assume that we have the following relation produced by VariableExplorer:

```
MethodInvocation ClassA variable2 variable1
```

This example represents a method invocation on the collection variable2 in ClassA, passing variable1 as a parameter. Method invocation on collections are the add, remove, and other methods supported by the Collection class in Java, and the passed parameters usually represent an element in the collection. Based on the type of the passed element, we can determine the type of the objects in the generic collection. Since variable1 is of type ClassB, the generic-typed collection variable2 is changed to a strongly-typed collection of type ClassB.

The final factbase consists of all the relations produced by the second stage, in addition to the relations shown in Table 2.

3.6 Stage 4: Design pattern Instance Recovery

In the last stage, we use QL to filter the factbase for instance candidates that satisfy the static definition of design patterns that we want to detect. For each design pattern, we define its participating roles. And for each role, we define the relations that it needs to satisfy. These static definitions have been created by studying each design pattern's structure, intent, and implementation. These definitions are denoted using the FINDER notation and translated to rules written in QL script. Following is an example of a QL script of a design pattern that consists of three roles:

```
DP[ClassA,ClassB,ClassC] = {
  //ClassA uses ClassB
  uses[ClassA,ClassB];
  //ClassB uses ClassC
  uses[ClassB,ClassC];
  //ClassA contains a method that returns ClassB
  class_methods[ClassA, method_id];
  method_return[method_id, ClassB];
```

Table 2: Relations produced by Stage 3

Relation Name	Example	Explanation
class_typed_collections	class_typed_collections ClassA CollectionC ClassB	ClassA contains a strongly-typed collection CollectionC of type ClassB. Moreover, if a class contains multiple separate references of another class, it is also considered a strongly-typed collection of the latter class type.
generic_typed_collections	generic_typed_collections ClassA CollectionC	ClassA contains a generic-typed collection CollectionC.

```
  //ClassB contains a static method
  //that returns ClassC
  class_methods[ClassB, method2_id];
  method_static[method2_id];
  method_return[method2_id, ClassC];
}
```

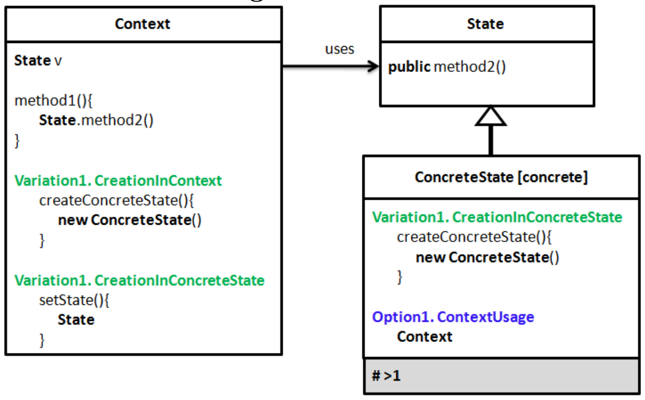
The output of this script is a list of instance candidates that satisfy these rules. A single design pattern instance candidate consists of role candidate class names corresponding to the design pattern roles. The following shows an example of two candidates for the previous script:

```
//DP RoleA, RoleB, RoleC
  DP ClassD ClassE ClassF
  DP ClassG ClassH ClassI
```

The first line defines the design pattern roles: RoleA, RoleB, and RoleC. The other two lines represent two instance candidates . The number of classes in each line has to match the number of roles. The first instance candidate consists of ClassD (a candidate for RoleA), ClassE (a candidate for RoleB), and ClassF (a candidate for RoleC).
In addition to design pattern roles, design pattern instance candidates may contain a method or a field that has an important function in the design pattern, such as the factory method in the FactoryMethod design pattern, and the Collection object in the Composite design pattern.

3.7 Example

Figure 1: State



In this section, we show how the FINDER model of the State design pattern (shown in Figure 1) is translated to a set of rules written in QL. Below are the rules represented by this model followed by the QL script that they are translated to:

1. ConcreteState must inherit from State:

```
inherits[concreteState,state];
```

2. ConcreteState must be concrete:

```
class_concrete[concreteState];
```

3. Context must contain a field of type State:

```
class_variables[context, variable_id];
variable_descriptors[variable_id,state];
```

4. Context must contain a method that calls a method in State:

```
class_methods[concreteState, method_id];
class_methods[context, method2_id];
method_invoke[method2_id, method_id];
```

5. Variation1:

   - CreationInContext:
     Context must contain a method that creates a new ConcreteState:

     ```
     class_methods[context, method_id];
     method_new[method_id, concreteState];
     ```

   - CreationInConcreteState:
     ConcreteState must contain a method that creates a new ConcreteState:

     ```
     class_methods[concreteState, method_id];
     method_new[method_id, concreteState];
     ```

     Also, the Context gets State passed to it through method parameters:

     ```
     class_methods[context, method_id];
     method_parameters[method_id, state];
     ```

     Or by calling a method that returns a Context:

     ```
     class_methods[context, method_id];
     method_invoke_direct[method_id,method2_id];
     method_return[method2_id, state];
     ```

6. Option1 (*ContextUsage*):
   ConcreteState must contain a reference to Context. Either by having a field of type Context:

```
class_variables[concreteState, variable_id];
variable_descriptors[variable_id,context];
```

Or having Context passed to it through method parameters:

```
class_methods[concreteState, method_id];
method_parameters[method_id, context];
```

Or by calling a method that returns Context:

```
class_methods[concreteState, method_id];
method_invoke_direct[method_id,method2_id];
method_return[method2_id, context];
```

The result of the previous scripts is a relation called DP that consists of three roles: Context, State, and ConcreteState. This relation contains instance candidates for the State design pattern that satisfy the FINDER rules specified in the State FINDER definition. Each line represents a single instance candidate. Below is a sample output for the DP relation:

```
//DP Context State ConcreteState
  DP Class1 Class2 Class3
  DP Class1 Class2 Class4
  DP Class1 Class2 Class5
  DP Class6 Class7 Class8
  DP Class6 Class8 Class9
```

7. Post processing rule: There must be least two ConcreteState role candidates in the same State anchor instance:

```
contextCandidates = dom DP
for context in contextCandidates
{
 concreteState = {context} . DP
 stateCandidates = dom concreteState
 for state in stateCandidates
 {
  concreteStates = {state} . concreteState
  if(#concreteStates[&0] < 2)
  {
   DP = DP - DP[&0 =~ context && &1 =~ state]
  }
 }
}
```

Running this post processing rule eliminates instance candidates that do not satisfy this rule. The last two instance candidates in DP will be deleted and the final output will be as follows:

```
//DP Context State ConcreteState
  DP Class1 Class2 Class3
  DP Class1 Class2 Class4
  DP Class1 Class2 Class5
```

The results show that there are three instance candidates of the State design pattern. Class1 plays the role of Context, Class2 plays the role of State, and Class3, Class4, and Class5 play the role of Concrete States.

Table 3: Table of results obtained with SSA for GTD-Free

Design Patterns	Candidates
Factory Method	1
Singleton	15
(Object)Adapter_Command	27
Composite	1
Decorator	1
Proxy2	1
State_Startegy	11
Template	2

4. CASE STUDY

The purpose of the case study presented below is to compare FINDER to two successful design pattern detection tools that have publicly available implementations, i.e. PINOT [15] and SSA [17]. We applied the three tools to GTD-Free version 0.6. GTD-Free is a personal TODO/action manager inspired by the GTD (Getting Things Done) method by David Allen. GTD-Free is a medium sized application with 22451 lines of code. GTD-Free consists of six subsystems:

1. Journal: provides date and time related features, deadlines, due dates, and events.
2. HTML: provides input/output functionalities managing attributes, elements and structure.
3. Add-ons: provides export options in different formats: PDF, XML, Text, and HTML.
4. Model: contains the actual implementation of systems functionalities, such as events, tasks, and action reminders.
5. Common: provides common features for the system: Global properties, system messages, GTD engine, and application helpers.
6. GUI: contains all the GUI elements and attributes.

PINOT could not detect any design patterns in GTD-Free. Therefore, further analysis could not be conducted. This could be due to the fact that PINOT is not able to detect design pattern implementations that vary from their original definitions. On the other hand, SSA uses a similarity scoring algorithm which enables it to recognize patterns that are modified from their standard representation. SSA detected 8 design patterns of the 12 patterns it supports. Table 3 shows the number of anchor instance candidates for design patterns detected by SSA.

FINDER detected 19 design patterns of the 22 patterns it supports. Running FINDER on GTD-Free took about 45 minutes. Extracting the factbase from the system under study took most of the time, while detecting the design patterns took less than a minute for each design pattern. Table 4 shows the number of instance candidates for the design patterns detected by FINDER [1].

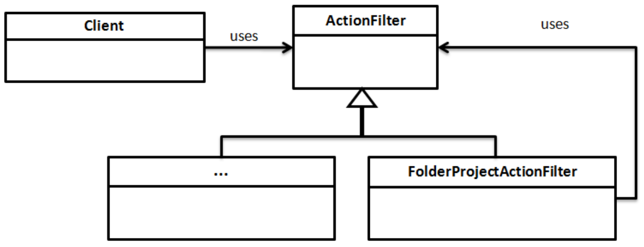
In the following, we discuss some of the design pattern instance candidates detected by SSA and FINDER:

[1]Clients of design patterns were excluded from the results.

Table 4: Table of results obtained with FINDER for GTD-Free

Design Patterns	Candidates
Factory Method	6
FactoryMethod Degenerate	7
Prototype Cloneable	1
Singleton	7
Adapter Object	6
Adapter Class	2
Bridge Degenerate	15
Composite	8
Decorator Degenerate	16
Flyweight	4
Chain of Responsibility	3
Command	4
Interpreter	5
Iterator	3
Iterator Degenerate	7
Iterator Java Degenarate	6
Memento	48
Observer	12
Observer with Subject	5
Observer Degenerate	76
Observer Degenerate with Subject	16
State	13
Strategy	31
Strategy with Context	5
Template Method	16

Figure 2: Decorator instance in GTD-Free detected by SSA and FINDER



- Decorator:
The class FolderProjectActionFilter in the Model subsystem was detected by SSA as a part of a Decorator design pattern instance, such that it plays the ConcreteDecorator role for the class ActionFilter that plays the role of Component. Figure 2 shows the UML diagram of this Decorator instance. This instance was also detected by FINDER as a Degenerate Decorator, which shows that FINDER is able to detect modified design patterns implementations. Moreover, FINDER detected an additional instance as a Degenerate Decorator, where the class ANDActionFilter plays the role of a ConcreteDecorator. Careful investigation of the source code showed that these instances are in fact Decorator design pattern implementations.

- Singleton:
All the Singleton design patterns detected by SSA and FINDER were enumerator classes. Using enumerator classes represents one way of implementing Singletons, and the reported candidates were true positives. According to Joshua Bloch in his second book "Effective Java" [2], a single-element enum type is the best way to implement a Singleton.

Figure 3: Composite instance in GTD-Free detected by SSA and FINDER

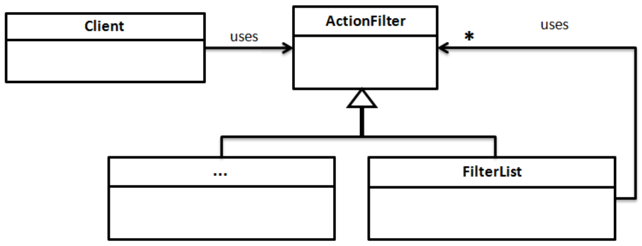
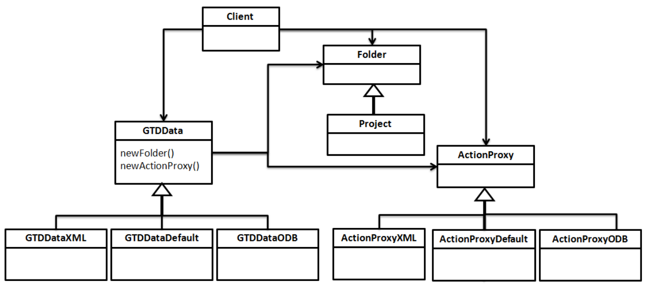


Figure 4: Abstract Factory instance in GTD-Free detected by SSA and FINDER

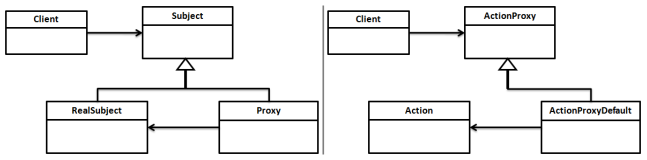


- Composite:
The class FilterList in the Model subsystem was detected by SSA as a part of a Composite design pattern, such that it plays the Composite role for the class ActionFilter that plays the role of Component. Figure 3 shows the UML diagram of the Composite instance. This instance was also detected by FINDER. FINDER generated 8 instance candidates for this anchor instance. This is because there is a large inheritance tree of ActionFilter subclasses, which causes FINDER to report all possible combinations of classes that play the role of a Leaf component.

- Abstract Factory:
GTD-Free supports three types of databases, and it implements an Abstract Factory design pattern to handle the creation of database proxies. SSA and FINDER correctly identified this implementation. Figure 4 shows the UML diagram for the Abstract Factory instance detected by SSA and FINDER. Note that SSA does not support the detection of Abstract Factory. However, SSA reported this instance as a Factory Method. GTDData plays the role of a Creator, while the subclasses play the role of ConcreteCreators. The two detected factory methods are *newFolder* and *newActionProxy* which return a Folder and an ActionProxy correspondingly, such that Folder and ActionProxy play the role of Product.

- Flyweight:
FINDER detected a true Flyweight instance in the

Figure 5: UML diagrams of the Proxy design pattern (to the left), and the Proxy in GTD-Free (to the right)



Model subsystem. The class GTDModel plays the role of Flyweight Factory, and contains a method (*getDataRepository*) that returns GTDData, which represents an instance of GTD database.

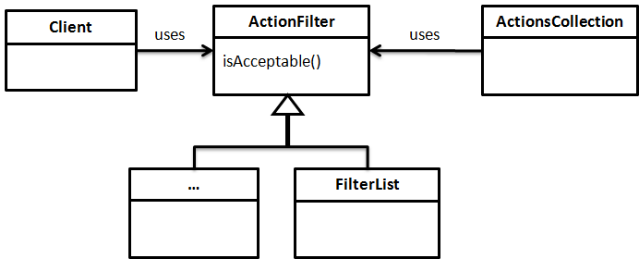
- Command:
  FINDER seems to agree with SSA on a number of instance candidates of the Command design pattern. The GUI subsystem contains the GUI elements, such as TextAreas, Dialogs, Panels and Trees. Actions on the GUI send updates to the DB in the Model subsystem. The ConcreteCommand (*ActionSpinner*) would trigger some actions on the Receiver (*Folder*).

- Iterator:
  FINDER successfully detected a number of Iterators and Java Iterators: FoldersProjectsActionsIterator, ActionTableIterator, ProjectsActionsIterator, and FoldersActionsIterator.

- Proxy:
  Class names often refer to a design pattern implementation. We noticed a number of classes in the GTD-Free source code that could represent a Proxy design pattern. For example, the Model subsystem contains an interface called ActionProxy, which has a number of subclasses that provide its implementation. FINDER and SSA unexpectedly did not report any Proxy instance candidates. Further investigation of the source code showed that the classes that play the roles of the Proxy and the RealSubject did not inherit from the class that should play the role of the Subject. Figure 5 shows two UML diagrams. To the left is the UML diagram of the Proxy design pattern. To the right is the UML diagram of the Proxy implementation provided in GTD-Free. This implementation does not represent a Proxy design pattern. GTD-Free can improve their implementation by abstracting out the common interface of Proxy and Action and adding it to the ActionProxy interface.

- Memento:
  FINDER correctly recovered Memento design pattern implementations. The GlobalProperties and GTDModel classes represent the Memento objects that contain the global properties and database options of the GTDFree application. The large number of candidates reported by FINDER is due to the fact that the Memento objects are referenced by a large number of classes. These classes play the role of CareTakers, such that they make changes to the memento objects, and are able to undo these changes.

Figure 6: Observer Degenerate instance in GTD-Free detected by FINDER



- Observer Degenerate:
  FINDER reported a large number of instance candidates for the Observer Degenerate design pattern. Most of them were false positives. Figure 6 shows the instance reported by FINDER as an Observer Degenerate candidate.
  A class called ActionFilter plays the role of an Observer, and contains a method called *isAcceptable*. This method is assumed by FINDER to be the update method in the Observer role. However, this method only returns a boolean, and does not perform an update on other system components. Since the implementation of the update method cannot be determined, no fine-grained rules in the update method can be defined. As a result, FINDER was unable to eliminate these candidates. Moreover, the large number of the recovered instance candidates is due to the large inheritance tree of the ActionFilter and ActionsCollection subclasses.

5. CONCLUSION

We successfully built the FINDER tool that recovers 22 GoF design patterns. The tool performs static analysis using well defined scripts that correspond to fine-grained design pattern detection rules. We evaluated our new approach by running FINDER on a large system, and examining the correctness of the detected design pattern instance candidates. FINDER gave more accurate results than the tools it was compared against. We have also evaluated FINDER in terms of scalability and robustness when running it on large Java systems and have shown that it scales well.

Several possibilities for future work exist. These include:

- Conducting further analyses of design patterns implementation in Java systems to produce more refined and flexible fine-grained definitions, which may lead to improved results. Moreover, creating FINDER tool definitions for new design patterns and adding them to our catalogue, and translating them into scripts to enable the detection of these design patterns using our tool.

- Performing additional tests to determine the best value for the threshold rules for the Adapter, Bridge, and Proxy design patterns. These values have been arbitrarily set to 50% with no further analysis of common design pattern implementations.

- Improving the performance of the FINDER tool, by optimizing the scripts used for factbase production and design patterns recovery. FINDER tool was able to run on a large scale real life system called JHotDraw. However, running the fine-grained analysis may become memory and CPU intensive, and it is likely that the tool would experience performance problems with other very large systems.

- Standardizing FINDER tool output and producing files in XML format, to allow the integration with other reverse engineering tools. FINDER tool can be integrated to a dynamic analysis tool such as PDE, such that FINDER tool provides instance candidates for PDE's dynamic analysis which performs more false positive elimination techniques. Moreover, providing results in XML format may facilitate building a Graphical User Interface(GUI) to improve the usability of FINDER tool.

- Enhancing the reporting of multiple instance candidates that belong to the same design pattern anchor instance. FINDER tool produces a large number of candidates for a design pattern because it reports all possible combinations for roles of a design pattern anchor instance. FINDER tool reports all design pattern instance candidates separately. The results may be refined by defining anchor roles for each design pattern, and then grouping different instance candidates by their anchor roles.

- Performing further tests to better evaluate the performance of our tool. FINDER tool precision and recall rates were only calculated for the AJP implementation results, and this does not reflect the true effectiveness of our tool.

6. REFERENCES

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 23, Washington, DC, USA, 1998. IEEE Computer Society.

[2] J. Bloch. *Effective Java*. Prentice Hall, 2 edition, 2008.

[3] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Design Pattern Recovery by Visual Language Parsing. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 102–111, Washington, DC, USA, 2005. IEEE Computer Society.

[4] I. G. Czibula and G. Serban. Hierarchical Clustering Based Design Patterns Identification. *International Journal of Computers, Communications and Control*, 3:248–252, 2008.

[5] J. Dong, D. S. Lad, and Y. Zhao. DP-Miner: Design Pattern Discovery Using Matrix. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 371–380, Washington, DC, USA, 2007. IEEE Computer Society.

[6] J. Fabry and T. Mens. Language-Independent Detection of Object-Oriented Design Patterns. *Computer Languages, Systems and Structures*, 30(1-2):21 – 33, 2004. Smalltalk Language.

[7] 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*, pages 143–152, Washington, DC, USA, 2008. IEEE Computer Society.

[8] L. J. Fulop, T. Gyovai, and R. Ferenc. Evaluating C++ Design Pattern Miner Tools. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 127–138, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Y.-G. Gueheneuc and G. Antoniol. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transactions on Software Engineering*, 34:667–684, 2008.

[10] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi. Fingerprinting Design Patterns. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic Design Pattern Detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[12] O. Kaczor, Y.-G. Gueheneuc, and S. Hamel. Efficient Identification of Design Patterns with Bit-vector Algorithm. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.

[13] G. Kniesel and A. Binun. Standing on the Shoulders of Giants - A Data Fusion Approach to Design Pattern Detection. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on Program Comprehension*, pages 208 –217, 2009.

[14] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. An Approach for Reverse Engineering of Design Patterns. *Software and Systems Modeling*, 4(1):55–70, February 2005.

[15] N. Shi and R. A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[16] J. M. Smith and D. Stotts. SPQR: Flexible Automated Design Pattern Extraction From Source Code. *Automated Software Engineering, International Conference on*, 0:215, 2003.

[17] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006.