Detecting Patterns and Antipatterns in Software using Prolog Rules

Alecsandar Stoianov
Coordonator: conf.dr.ing. Ioana Şora

***Abstract*—[1]Program comprehension is a key prerequisite for the maintainance and analysis of legacy software systems. Knowing about the presence of design patterns or antipatterns in a software system can significantly improve the program comprehension.Unfortunately, in many cases the usage of certain patterns is seldom explicitly described in the software documentation, while antipatterns are never described as such in the documentation. Since manual inspection of the code of large software systems is difficult, automatic or semi-automatic procedures for discovering patterns and antipatterns from source code can be very helpful.**

In this article we propose detection methods for a set of patterns and antipatterns, using a logic-based approach. We define with help of Prolog predicates both structural and behavioural aspects of patterns and antipatters. The detection results obtained for a number of test systems are also presented.

***Keywords*-design pattern; antipattern; detection**

I. Introduction

An important issue when having to deal with legacy software is the easiness to comprehend it, an essential prerequisite for maintaining it and assessing its quality. This is not a simple task due to the size and complexity of such software systems often combined with a lack of sufficient detailed documentation. Program comprehension is best facilitated by abstract views of the system that hide certain details and concentrate on the interactions of main elements. High-level abstactions of software are best described using concepts standardized as patterns and styles.

Design pattern, as defined in [1] provides a scheme for refining the subsystems of a software system, or the relationships between them. A design pattern

> "describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context."

Design patterns were defined in order to make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. However, design patterns also present a second advantage, of introducing a more abstract but standardized way of concisely describing designs.

As defined in [2], an AntiPattern is

> "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences."

AntiPatterns are able to synthetise most common problems or frequent mistakes made in software design. They are valuable because they can provide genaral reengineering solutions for a whole class of problems.

AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common mistakes occuring in software projects [2]. They are also a higher-level vocabulary that simplifies communication between software practitioners and enables concise description of abstract concepts.

Knowing about the presence of patterns or antipatterns in a software system can significantly improve the program comprehension through the higher-level abstraction introduced. Unfortunately, in many cases the usage of patterns is seldom documented, while antipatterns are never described as such in the product documentation. In this context, automatic or semi-automatic procedures for discovering patterns and antipatterns from source code can be very helpful. This is a form of software architecture reconstruction [3].

Regardind design pattern detection, a lot of research has been done and different research tools have been built [4], [5].

Detection of antipattern has not been investigated with the same extent [6]. Most work in an area realted to this subject is that in the field of detecting design flaws and code smells based on metrics [7], [8].

Both design patterns and antipatterns are usually defined by both static and dynamic aspects: a pattern or an antipattern specifies structural connections among the classes(association, inheritance, etc) but it also requires some specific sequences of actions and interactions of the objects of these classes. Hence, the detection algorithms become complicated by the dynamic aspects which are not directly capturable in metrics.

As analysed in [9], an alternative to algorithmic detection methods are logic-based approaches, which have demonstrated capabilities to quickly define and execute complex queries over code bases.

The main goal of our work is to enhance antipattern detection by using a logic-based approach. Section II presents shortly the infrastructure used for obtaining the AST representation in form of logic predicates. Section III briefly describes our work on design pattern detection, done with the purpose of comparing the potential of the logic-based approach with other approaches. In section IV we present our approach for antipattern detection. Experimental results are described in Section V.

[1]This article has been published in Proceedings of ICCC-CONTI 2010

II. Infrastructure used by our approach

This work uses as an infrastructure JTransformer [10], an Eclipse plugin that generates the representation of a Java program as logic facts. JTransformer uses the platform-independent, free SWI-Prolog implementation for storing, analyzing and transforming factbases. JTransformer represents the complete program, including the complete abstract syntax tree (AST) of method bodies.

When JTransformer generates the Prolog fact base, it includes all the classes that are being used by the project, even those from Java API. Since pattern identification should be limited on the parts of the project that are implemented by the development team, all queries are limited to the classess identified as part of the projects source code by the predicate myProject.

III. Design pattern detection

In this work, we define Prolog queries to detect following Go4 [1] patterns: Observer, Singleton, Strategy, Adapter, Decorator.

A. Observer

To detect an Observer Pattern, we start by indentifying an Observer class candidate. Observer class, as it is specified in the GOF Pattern Catalog, should be an abstract class or an interface. The Update method could be specified in it, but that is not a must, that is why we don't check the form of Observer candidate methods, just their existence. The next step is finding a Subject class. GOF specifies that a Subject should have an Attach method, a Detach method and a Notify method. Attach and Detach methods are very alike. They both have a single parameter, the Observer. Notify method is being identified as the Subject's method that has a loop block in which the Observer's Notify method is being called. Here is where the Notify method is being identified. Concrete Subject and Observer classes are all the classes that implement the abstract Observer and Subject. The final step is the binding of all these classes in the Observer Pattern. Below we give an excerpt of our Prolog implementation.

```
observer(Observer,ObserverN,Project):-
myProject(Observer,Project,ObserverN),
interfaceT(Observer),
classDefT(Observer,_,_,UpdateMs),
member(UpdateM,UpdateMs),
methodDefT(UpdateM,_,_,_,_,_,_).

subject(Subject,SubjectN,Observer,
      AttDetMeths,NotifyMeths,UpdateMeths):-
classDefT(Subject,_,SubjectN,Fields),
%modifierT(Subject,'abstract'),
attachDetachMeths(Fields,Subject,
               Observer,AttDetMeths),
findall(X,notifyMeth(_,X,
        Subject,Observer,_),NotifyMeths),
NotifyMeths=[_|_],
findall(Y,notifyMeth(_,_,
        Subject,Observer,Y),UpdateMeths),
UpdateMeths=[_|_].

concreteSubject(ConcrSubj,ConcrSubjN,Subject):-
impl(ConcrSubj,Subject),
classDefT(ConcrSubj,_,ConcrSubjN,_).

concreteObserver(ConcrObs,ConcrObsN,Observer):-
impl(ConcrObs,Observer),
classDefT(ConcrObs,_,ConcrObsN,_).

observerPattern(Project, SubjectN, ObserverN,
 ConcrSubjsN,ConcrObserversN, UpdateMsN,
 AttDetMeths,NotifyMeth):-
observer(Observer,ObserverN,Project),
subject(Subject,SubjectN,Observer,AttDetMeths,
           NotifyMeth,UpdateMsN),
findall(X,concreteSubject(_,X,Subject),
                              ConcrSubjsN),
findall(Y,concreteObserver(_,Y,Observer),
                          ConcrObserversN).
```

B. Singleton

The Singleton Pattern is slightly simpler than Observer. If a class is a Singleton it should have only private constructors and a static method that returns a single instance of that class. All of these are checked by the `singletonPattern` predicate. The getInstance method is being identified by the `getInstanceMeth` predicate. For shortness, the Prolog implementation details of these predicates are not presented.

C. Strategy

In [1], the Strategy class is defined to be an abstract class or an interface. The Concrete Strategy class is the one that extends the Strategy and implements the algorithm method. After collecting the Strategy class candidates with the `strategy` predicate, we resume by finding all the Concrete Strategy classes with the concrStategy predicate. This predicate also gives us the `Algorithm` method name. With this the Strategy pattern is identified.

D. Adapter

Adapter Pattern identification starts with finding all the classes that implement the `Target` interface. These are considered to be Adapter classes. The Adapter classes also have to have a reference to the Adapted class. In the `Request` method the adapted method from the Adapted class should be invoked. All of these conditions are being verified by the `adapter` predicate. The conversion from fact IDs to the class names completes the Adapter Pattern detection. The relevant excerps of Prolog code are presented below.

```
target(Target, TargetN):-
classDefT(Target,_,TargetN,_),
interfaceT(Target).

adapter(Adapter,AdapterN,Target,Project,
  RequestN,AdapteeN,SpecificRequestN):-
myProject(Adapter,Project,_),
classDefT(Adapter,_,AdapterN,_),
impl(Adapter,Target),
methodDefT(Req,Adapter,RequestN,Reqp,Reqt,_,_),
methodDefT(_,Target,RequestN,Par2,Reqt,_,_),
equalTypes(Reqp,Par2),
```

```
callT(Call,_,Req,Recv,SpecificRequestN,_,_),
getFieldT(Recv,Call,Req,_,_,Field),
fieldDefT(Field,_,type(class,Adaptee,_),_,_),
classDefT(Adaptee,_,AdapteeN,_),
myProject(Adaptee,Project,_),!.

adapterPattern(Project,TargetN,AdapterN,RequestN,
  AdapteeN,SpecificRequestN):-
  target(Target,TargetN),
adapter(_,AdapterN,Target,Project,RequestN,
         AdapteeN,SpecificRequestN).
```

E. Decorator

The detection of the Decorator Pattern starts with identifying the Component class. This is an abstract class or interface that contains at least one method. The `decoratorPattern` predicate presented below continues by finding all Concrete Component classes with the help of the `concrComp` predicate. Next step is to find the Decorator class and Concrete Decorator classes with `decorator` and `concrDec` predicates. Decorators are found by identifying all those classes that implement the Component and have an attribute of the Component type. Concrete decorators are all the classes that extend Decorator and override a method in which they call the overridden method from the Decorator class. This is the Operation method. The `decorator` predicate verifies if in the Operation method of the Decorator the Operation method from the Component is called.

```
component(Comp,CompN,OpN,OpP,OpT):-
  interface(Comp),
classDefT(Comp,_,CompN,_),
methodDefT(_,Comp,OpN,OpP,OpT,_,_),
not(OpN='<init>').

concrComp(ConcrComp,Comp,ConcrCompN,
     OpN,OpP,OpT):-
  component(Comp,_,OpN,OpP,OpT),
impl(ConcrComp,Comp),
not(interface(ConcrComp)),
classDefT(ConcrComp,_,ConcrCompN,_),
methodDefT(_,ConcrComp,OpN,OpP2,OpT,_,_),
equalTypes(OpP,OpP2),
not(decorator(ConcrComp,Comp,_,_,OpN,OpP,
               OpT)).

decorator(Dec,Comp,Project,DecN,OpN,OpP,OpT):-
  classDefT(Dec,_,DecN,Fields),
myProject(Dec,Project,_),
component(Comp,_,_,_,_),
impl(Dec,Comp),
  member(Field,Fields),
fieldDefT(Field,_,type(class,Comp,_),_,_),
methodDefT(Op,Dec,OpN,OpP2,OpT,_,_),
equalTypes(OpP,OpP2),
callT(Call,_,Op,Target,OpN,_,_),
getFieldT(Target,Call,Op,_,_,Field),!.

concrDec(ConcDec,Dec,Comp,Project,ConcDecN,
    OpN,OpP,OpT):-
  classDefT(ConcDec,_,ConcDecN,_),
decorator(Dec,Comp,Project,_,OpN,OpP,OpT),
impl(ConcDec,Dec),
methodDefT(Op,ConcDec,OpN,OpP2,OpT,_,_),
equalTypes(OpP,OpP2),
callT(Call,_,Op,Target,OpN,_,_),
identT(Target,Call,Op,'super',_).

decoratorPattern(PName,CompN,ConcrCompsN,
    DecN,ConcDecsN,OpN):-
component(Comp,CompN,OpN,OpP,OpT),
findall(X,concrComp(_,Comp,X,OpN,OpP,OpT),
                      ConcrCompsN),
decorator(Dec,Comp,PName,DecN,OpN,OpP,OpT),
findall(Y,concrDec(_,Dec,Comp,PName,Y,
                   OpN,OpP,OpT),ConcDecsN),
not(length(ConcrCompsN,0)).
```

IV. ANTIPATTERN DETECTION

A difficulty that appears in detecting antipatterns is that, opposed to design patterns that are described by precise UML class and collaboration diagramms, antipattern definitions are usually given as textual descriptions. We defined Prolog queries for detecting a set of antipatterns described in [11] and in [2].

A. Data Class

The symptoms of a Data Class anti-pattern are described in [11] and they are objects with no behavior in them, just bags of getters and setters. The detection of this anti-pattern starts with declaring two predicates for identification of getter and setter methods:

```
  setMethod(Meth):-
methodDefT(Meth,_,Name,Params,_,_,Body),
not(Name='<init>'),
length(Params,1),
blockT(Body,_,_,[Exec]),
execT(Exec,_,_,Assign),
assignT(Assign,_,_,_,_).

getMethod(Meth):-
methodDefT(Meth,_,Name,[],_,_,Body),
not(Name='<init>'),
blockT(Body,_,_,[Exec]),
returnT(Exec,_,_,GetField),
getFieldT(GetField,_,_,_,_,_).
```

The `methodDefT` predicate is called to get the given method's name, parameters and body. For the getter method the parameter field should be an empty list because getters usually do not have parameters. The next line checks if the name is different from `<init>`, which is how JTransformer represents constructors. A setter method should have only one parameter, the one that is setting. This fact is checked by the third line of `setMethod`. With `blockT` we get the code lines. A usual getter or setter has only one line of code in which an attribute is set or returned. This fact is being verified in the remainder of two methods.

The condition for a class to be a Data class is that all of its fields are attributes or getter and setter methods. This is probed by getting the list of class fields and passing it to the `field` predicate.

B. Call Super

Another anti-pattern defined in [11] is Call Super. It requires users of a particular interface to override the method of a superclass and call the overridden from the overriding method. This anti-pattern appears form time to time in OO frameworks. An aproach to refactor this is to use the template method pattern, where the superclass includes a purely abstract method that must be implemented by the subclasses and have the original method call that method.

For detection of this anti-pattern we defined the predicates `isCallSuper` and `isNotCallSuper` as presented below.

```
isCallSuper(Project,CallSuperCls,
Sup,MethInSup):-
extendsT(Subcls,Sup),
myProject(Subcls,Project,
CallSuperCls),
methodDefT(Meth,Subcls,
CallSuperMethod,Param1,
Type,_,_),
methodDefT(MethInSup,Sup,
CallSuperMethod,Param2,
Type,_,_),
equalTypes(Param1,Param2),
not(CallSuperMethod='<init>'),
callToSuper(Meth,CallSuperMethod,MethInSup).

isNotCallSuper(Project,CallSuperCls,
Sup,MethInSup):-
extendsT(Subcls,Sup),
myProject(Subcls,Project,CallSuperCls),
methodDefT(Meth,Subcls,
CallSuperMethod,Param1,Type,_,_),
methodDefT(MethInSup,Sup,
CallSuperMethod,Param2,Type,_,_),
equalTypes(Param1,Param2),
not(CallSuperMethod='<init>'),
not(callToSuper(Meth,CallSuperMethod,MethInSup)).
```

The two methods are very similar, the only difference is the last line. They start with getting a class that extends the super class passed in the `Sup` parameter. The subclass should be a part from the analyzed project, this is verified with a call to `myProject`. The next four lines check if a method is being overridden and that method is not a constructor. The last line is a call to `callToSuper` predicate.

```
callToSuper(Meth,CallSuperMethod,MethInSup):-
identT(Ident,Call,Meth,super,_),
callT(Call,_,Meth,Ident,CallSuperMethod,_,
                                  MethInSup).
```

`Meth` parameter represents the ID of overriding method. `CallSuperMeth` represents the methods name, while the `MethInSup` parameter is the ID of the overridden method. This predicate checks if a call to the `MethInSup` was made from `Meth`.

Call to Super anti-pattern is identified by the `callSuper` predicate.

```
callSuper(Project,CallSuperCls,SuperCls,
CallSuperMethod):-
classDefT(Sup,_,SuperCls,_),
methodDefT(MethInSup,Sup,
CallSuperMethod,_,_,_,_),
findall(X,
isCallSuper(Project,X,Sup,MethInSup),
CallSuperCls),
not(length(CallSuperCls,0)),
findall(Y,
       isNotCallSuper(Project,Y,Sup,MethInSup),
       NotCallSuperCls),
length(NotCallSuperCls,0).
```

In the `callSuper` predicate, `Project` is the name of the project, `CallSuperCls` is the list of names of the subclass, `SuperCls` is the super class and `CallSuperMethod` is the problematic method's name. The first line gets all the classes used in the project, one by one. The next thing is to get all of its methods. In the third line a list of all the classes that fulfill the `isCallSuper` predicate is created. This list should not be empty. `isNotCallSuper` gets all the subclasses that override the problematic method but do not have a call to the super class. This list should be empty.

C. Constant interface

Constant Interface is an anti-pattern defined by Joshua Bloch. The problem is when you have an interface filled up just with constants. Interfaces from the given project are checked one by one by the `constantInterface` predicate.

```
constantInterface(Project,InterfaceN):-
myProject(Inter,Project,InterfaceN),
interfaceT(Inter),
classDefT(Inter,_,InterfaceN,Fields),
constant(Fields).
```

In the third line of this predicate we get all the fields of the given interface. The `Fields` list is passed to the `constant` predicate which checks if all of them are static attributes.

```
constant([H|[]]):-
fieldDefT(H,_,_,_,_),
modifierT(H,static).
constant([H|T]):-
fieldDefT(H,_,_,_,_),
modifierT(H,static),
constant(T).
```

D. The Blob (God Class)

This antipattern is described in [2] and has also many variations known as God Class, God Object. It is a diagnostic of a procedural-style design, where one object holds the majority of responsibilities, while most other objects only hold data or execute simple processes.

```
godClass(Project,GO):-
myProject(Cls,Project,GO),
cycloClass(Cls,V),
V>55,
callToDataClass(Cls),
classDefT(Cls,_,GO,Fields),
length(Fields,Nr),
Nr>=25.
```

In order to detect God Classes the `godClass` predicate, listed above, first calculates the complexity of the class. All

the classes that have a big complexity are considered to be candidates for a God Class. The next condition for class to be identified as a God Class is that it works with data classes. This is verified by the `callToDataClass` predicate.

```
callToDataClass(GO):-
methodDefT(Meth,GO,_,_,_,_,_),
getFieldT(_,_,Meth,_,_,Field),
not(fieldDefT(Field,GO,_,_,_)),!.

callToDataClass(GO):-
methodDefT(Meth,GO,_,_,_,_,_),
callT(_,_,Meth,_,_,_,Field),
setMethod(Field),
not(fieldDefT(Field,GO,_,_,_)),!.
callToDataClass(GO):-
methodDefT(Meth,GO,_,_,_,_,_),
callT(_,_,Meth,_,_,_,Field),
getMethod(Field),
not(fieldDefT(Field,GO,_,_,_)),!.
```

E. Refused Interface

Interface Bloat or Refused Interface is a form of the Refused Bequest antipatterns as it is defined by Martin Fowler in [11]. It describes interfaces that try to incorporate all possible operations on some data into an interface, abstract class or root of an inheritance hierarchy, but most of the concrete classes cannot perform the given operations or don't need the inherited data. Our predicate interfaceBloat checks wether a cass doesn't implement all methods from its interface. We consider that an interface is not implemented if at least one of its methods has an empty method body - here we count also method bodies that consist only of a return null or return constant value.

```
doesntImplement(Interface):-
impl(Class,Interface),
hasEmptyMethod(Class,Meth),
methodDefT(Meth,Class,Name,Param1,Type,_,_),
methodDefT(_,Interface,Name,Param2,Type,_,_),
equalTypes(Param1,Param2),!.

interfaceBloat(Project,Interface):-
myProject(Id,Project,Interface),
interface(Id),
classDefT(Id,_,_,_),
doesntImplement(Id).
```

F. Yoyo problem

A project has a Yoyo Problem if the programmer has to keep flipping between many different class definitions in order to follow the control flow of the program and the structure is hard to understand due to excessive fragmentation. This problem appears for example when we have a very deep inheritance graph, and the implementation of methods in derived classes calls many methods defined at a higer level in the inheritance hierarchy.

```
callTree(_,0):-!.
callTree(Meth,N):-
methodDefT(Meth,ClassA,_,_,_,_,_),
callT(_,_,Meth,_,_,_,CalledMeth),
methodDefT(CalledMeth,ClassB,_,_,_,_,_),
plus(N,-1,N2),
mostenire(ClassA,ClassB),
callTree(CalledMeth,N2).

hasCallTree(ProblemTree,Factor):-
N is Factor // 2,
member(ClassN,ProblemTree),
classDefT(Class,_,ClassN,_),
methodDefT(Meth,Class,_,_,_,_,_),
callTree(Meth,N),!.

yoyoProblem(Project,Factor,ProblemTree):-
myProject(Super,Project,SuperN),
superClass(Super,Project),
extends(Sub,Super),
extendTree(Sub,[SuperN],ProblemTree),
length(ProblemTree,L),
L>=Factor,
hasCallTree(ProblemTree,Factor).
```

G. Poltergeist

This antipattern is described in [2]. Poltergeists are classes with very limited roles and effective life cycles. They often only start processes for other objects.

```
poltergeist(Project, Poltergeist):-
myProject(Pol,Project,Poltergeist),
not(interfaceT(Pol)),
not(fieldDefT(_,Pol,_,_,_)),
findall(X,notConstructor(Pol,X),L),
L=[Meth],
hasCall(Meth),
shortLivingParam(Pol),
shortLivingLocal(Pol),
shortLivingFields(Pol).

shortLivingParam(Class):-
isType(Class,Sup),
findall(X,paramDefT(X,_,type(_,Sup,_),_),Params),
areOfType(Params,Class,Params2),
onlyOneReference(Params2),!.
```

V. Experimental results

A. Experimental results of our design pattern detection approach

We applied our pattern detection mechanism (described above in section III) on different systems. The table I presents the results of our approach compared with the results provided by the Pinot tool [12] on the same analysed systems.

Table I shows that our approach produces relatively similar results with the Pinot tool for Singleton, Decorator and Strategy.

Significant differences between our results and Pinot are in the case of Observer and Adapter.

In case of the Adapter pattern, we transposed in Prolog predicates all the conditions of the pattern definition in [1]. Our tool discovered more occurences of this pattern, but the manual inspection showed that they are correct according to the pattern definition.

In the case of the Observer pattern, our tool detects fewer occurences than Pinot. The differences in the number of Observer pattern occurences come from the fact that our

TABLE I
EXPERIMENTAL RESULTS FOR DESIGN PATTERN DETECTION

	JHotDraw6.0b1		Java AWT 1.3		Java Swing 1.4		Java.io 1.4.2		java.net 1.4.2		Apache Ant 1.6.2	
	Own	Pinot	Own	Pinot	Own	Pinot	Own	Pinot	Own	Pinot	Own	Pinot
Observer	5	9	4	9	49	68	0	13	0	0	2	5
Singleton	1	0	3	3	1	0	0	0	0	0	2	1
Strategy	51	51	43	54	110	96	5	3	0	4	64	53
Adapter	9	5	9	3	30	17	0	4	0	0	9	13
Decorator	4	5	3	3	13	15	3	4	0	1	3	4

TABLE II
EXPERIMENTAL RESULTS FOR ANTIPATTERN DETECTION

	JHotDraw6.0b1	Java AWT 1.3	Java Swing 1.4	Java.io 1.4.2	java.net 1.4.2	Apache Ant 1.6.2
Data Class	10	23	55	13	9	33
Call Super	72	34	223	8	0	86
God Class	13	23	86	8	2	63
Interface Bloat	25	27	48	1	0	27
Yoyo problem	4	0	16	0	0	9
Poltergeist	18	2	7	0	0	25

approach is more rigourous in checking all the characteristics of the pattern, as defined in [1]:

- our approach explicitely checks that the Subject's notify method has a loop block where the Observer's notify method is called, as opposed to the approach implemented by [12] where it is not checked if the call is made from inside a loop
- our approach checks the presence of attach and detach methods, also not checked in [12]

Thus we consider our detection approach for Observer patterns to be more accurate according to the pattern definition than the aproach implemented in Pinot [12].

B. Experimental results of our antipattern detection approach

We applied our antipattern detection approach (described above in section IV) on different systems and present the detection results in table II. As we checked the results by manual inspection of the code, no false positives were found among the automatic detected antipatterns. Future work of validation has to be done in order to check for false negatives (presence of antipatter not detected by the automatic approach).

VI. CONCLUSION

In this article we propose detection methods for a set of patterns and antipatterns, using a logic-based approach. The main advantage of this approach is the simplicity of defining Prolog predicates able to describe both structural and behavioural aspects of patterns and antipatters. This advantage becomes more visible in the case of patterns and antipatterns that are characterised not only by structural aspects, but also by complex behavioural aspects that are difficult or not possible to describe by code metrics.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[2] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wilet and Sons, 1998.

[3] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Campan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 137–148, 2007.

[4] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *The International Journal of Software Engineering and Knowledge Engineering (IJSEKE),*, vol. 19, pp. 823–855, 2009.

[5] G. Kniesel and A. Binun, "Standing on the shoulders of giants – a data fusion approach to design pattern detection," in *17th International Conference on Program Comprehension (ICPC' 2009)*. IEEE Computer Society Press, 2009, pp. 208–217.

[6] M. T. Llano and R. Pooley, "UML specification and correction of object-oriented anti-patterns," in *Proceedings of Fourth International Conference on Software Engineering Advances*. IEEE Computer Society Press, 2009, pp. 39–44.

[7] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, pp. 20–36, 2010.

[8] M. Lanza and R. Marinescu, *Object oriented metrics in practice*. Springer Verlag, 2006.

[9] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," in *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*. New York, NY, USA: ACM, 2007, p. 6.

[10] The JTransformer project. [Online]. Available: http://roots.iai.uni-bonn.de/research/jtransformer/

[11] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] The Pinot project. [Online]. Available: http://www.cs.ucdavis.edu/ shini/research/pinot/index.html