Identification of Problematic Constructions in Object Oriented Applications: an Approach Based on Heuristics, Design Patterns and Anti-Patterns

Alexandre L. Correa Cláudia M. L. Werner Gerson Zaverucha

Federal University of Rio de Janeiro (COPPE/UFRJ)
Programa de Engenharia de Sistemas e Computação
Caixa Postal 68511
21945-970 Rio de Janeiro, RJ, Brazil
{ alexcorr, werner, gerson}@cos.ufrj.br

Abstract

Object Oriented languages do not guarantee that a system is flexible enough to absorb future requirement evolution, or that its components can be reused in other contexts. This paper presents an approach to the identification of existing problematic constructions in OO applications. Our main goal is to provide support both for reengineering legacy OO systems, and evaluating OO systems that are still under development, by detecting certain constructions that compromise their future expansion or modification, and suggesting their replacement by more adequate ones. This approach is supported by a tool (OOPDTool) comprising a knowledge base of good design constructions, that correspond to heuristics and design patterns, as well as problematic constructions (i.e., anti-patterns).

1. Introduction

One of the main motivations to use the Object Oriented development paradigm is the promise of being able to construct large and complex systems that are flexible in the sense of absorbing constant business modifications, as well as technological evolutions. This promise would be accomplished by using techniques such as encapsulation, inheritance, polymorphism, dynamic binding and interfaces, among others, resulting in a modular and reusable software structure. In this way, it should be possible to dramatically reduce maintenance efforts and achieve a significant increase in productivity while developing new systems, if compared to more traditional development paradigms.

However, it is possible to find many OO applications that are hard to maintain nowadays. These systems present rigid and inflexible structures that make the addition of new features, due to inevitable requirement changes, a difficult task. As an example, it is possible to mention some of the systems developed by Nokia and Daimler-Benz that are being studied in the context of the FAMOOS Project [27], aiming at the transformation of legacy OO systems into OO frameworks.

Although OO concepts are known by the academic community for at least two decades, the design of a flexible and reusable system is still a very difficult task. It involves the identification of relevant objects, factoring them into classes of correct granularity level, hierarchically organizing them, defining class interfaces and establishing the dynamics of collaboration among objects. The resulting design should solve the specific problem at hand but at the same time should be generic enough to be able to address future requirements and needs [10]. While performing these tasks, new OO designers have to decide among several possible alternatives, often presenting a tendency to apply non OO techniques both because of their greater familiarity with these techniques and time schedule pressures which do not allow them to “waste time” searching for the best solution.

In fact many of these problematic OO systems were developed by teams where the majority of professionals did not have enough experience to define the most adequate construction for a given problem, in order to make the design more flexible and resilient to change.

Many OO development methods appeared in this decade, among them the ones described in [1], [25] and [11]. Along with these methods, several tools became available (e.g., Rational Rose [23] and Paradigm Plus [20]). The emphasis of these methods and tools has been on how to develop semantically correct OO models regarding the constructions available in modeling languages such as UML, for example. However, a correct model does not necessarily mean that it is flexible and reusable. Due to this fact, many organizations nowadays

depend on reviews performed by experts to increase design quality and avoid too much effort on future maintenance. These reviews are very costly and such experts are scarce at the market.

This work presents an approach for both reengineering problematic legacy OO systems, and evaluating OO systems that are still under development. This approach is supported by a tool that detects good and bad OO design constructions, i.e., constructions corresponding to standard solutions to recurring design problems (*design patterns*), or constructions that can result in future maintenance and reuse problems. With this aid, it is possible to identify points in a system that need to be modified in order to make it more flexible and reusable, and to ease the system understanding as a whole, including the badly documented systems, by identifying existing design patterns.

The identification of problematic OO software constructions is very difficult to be done manually. We can highlight the following reasons for this difficulty:

- legacy systems that need to be reengineered are usually medium/large in size, making manual search for problems unfeasible;
- in most cases, the only reliable source for design information is the source code. Models, when available, either are out of date or are too superficial to support a design analysis. However, manual analysis of source code limits the scope of the problems that can be found in a timely and economically way;
- developers often do not know what kind of problems they should be looking for. A database containing potential design problems can provide a valuable support in this case.

This paper is organized as follows: section 2 introduces some basic concepts regarding design patterns and heuristics for the construction of good OO designs. Section 3 discusses the anti-pattern concept and OO design problems. Section 4 presents a tool for the detection of good and bad OO design constructions, called OOPDTool. In section 5, some related works are discussed, and in section 6 the conclusions and future works are presented.

2. Heuristics and Design Patterns

In works such as [24], [16], and [12], it is possible to find a set of heuristics for achieving a quality OO design. A design heuristic is a sort of guidance for making design decisions. It describes a family of potential problems and provides guidelines to aid the designer to avoid them. Heuristics, such as “All data must be hidden within its class” [24], direct designer’s decisions toward a more flexible and reusable OO design. It is important to note that a heuristic should not be considered as a law that must be followed in all circumstances. It should be seen as an element that, if violated, would indicate a potential design problem.

Apart from these heuristics, the “design patterns philosophy” [10] emerged as one possible way to capture the knowledge of OO design experts allowing to easily reuse well succeeded solutions to recurring problems regarding design and software architectures.

Synthesizing the definitions of various authors ([5], [8], [1], [15], [26] and [29]), it is possible to state that a design pattern solves a *recurring problem*, in a given *context*, by providing a proven working *solution* (not speculations or theories), also indicating its *consequences*, i.e., the results and tradeoffs of its application, and providing information regarding its adaptation to a problem variant. Every pattern is identified by a *name*, forming a common vocabulary among designers. While heuristics represent generic directives to OO design, a design pattern is a solution to a specific design problem.

The key to maximize reuse and minimize software maintenance effort resides in trying to anticipate new requirements, as well as possible changes on existing ones, while designing the system. By using heuristics and design patterns, it is possible to avoid that these modifications result in great changes of the software structure, since they allow a certain design aspect to vary in an independent way, making it more resilient to a particular kind of change. In [10], we can find common causes for design inflexibility and the corresponding design patterns that can be applied. For example, the instantiation of an object by its class explicit specification makes a design dependent on a particular implementation and not on its interface. In this case, the *Abstract Factory*, *Factory Method* and *Prototype* design patterns can be applied to make the instantiation process more flexible.

3. OO Design Problems and Anti-patterns

Software design is an iterative task and designers often need to reorganize its elements in order to make them more flexible. However, design methods and their supporting tools that are available today focus on the development of new systems. The same happens with the use of heuristics and design patterns. Many organizations have inflexible OO systems nowadays; many of which are critical, incorporating undocumented business knowledge. Moreover, it is almost always unfeasible, from an economical point of view, to throw these systems

away and rebuild them from scratch using a more flexible and reusable design. A more interesting solution would be to reengineer those systems looking for constructions that are responsible for the system inflexibility, and replacing them by others that are more flexible and resilient to change.

The study of anti-patterns has recently emerged as a research area for detecting problematic constructions in OO designs [4], [13]. An anti-pattern describes a solution to a recurrent problem that generates negative consequences to the project. An anti-pattern can be the result of either not knowing a better solution, or using a design pattern (theoretically, a good solution) in the wrong context. When properly documented, an anti-pattern describes its general format, the main causes of its occurrence, the symptoms describing ways to recognize its presence, the consequences that can result from this bad solution, and what should be done to transform it into a better solution.

However, the number of catalogued anti-patterns is still small, if compared to the number of design patterns available in the technical literature [5], [7], [10], [17] and [28]. In [4], there is a list of some anti-patterns as, for instance, the *Blob* anti-pattern. This anti-pattern corresponds to an OO design solution that strongly degenerates into a structured OO design style. It can be recognized when responsibilities are concentrated on a single object, while the majority of other objects are used as mere data repositories, only providing access methods to their attributes (get/set methods). This kind of solution compromises the ease of maintenance and should be restructured by better distributing system responsibilities among objects, isolating the effects of possible changes.

Since the number of anti-patterns available in the literature is still small, one possible way to guide the process of searching for problematic solutions in OO software design is by looking at design patterns catalogues. These catalogues not only describe good solutions applicable in a particular context, but also informally discuss bad solutions that could have been used instead. Those bad solutions can be formalized and catalogued, composing a database of OO design problems.

OO design heuristics can be another source for the identification of design problems. The possible ways of violating a particular heuristic correspond to potential problems that can be found in a design. For instance, the definition of attributes in the public area of a class violates the principle “All data should be hidden within its class” [24], corresponding, therefore, to a potential design problem. It is not always true, however, that the violation of an OO design principle corresponds to a design problem, since a particular bad construction can be driven by some specific requirements such as efficiency, hardware/software constraints, among other non functional ones.

4. OOPDTool

As mentioned before, our main goal is to provide automated support both for reengineering legacy OO systems, and evaluating OO systems that are still under development. This support means detecting design constructions that can make future system maintenance harder, and helping replacing them by more flexible ones. Therefore, we would like to identify points in the system that should be modified in order to make it more flexible and reusable. This paper presents a tool, named OOPDTool, designed to support this task. As shown in figure 1, there are four main modules composing the OOPDTool architecture:

- **Design Extraction:** to automatically extract design information from OO source code, generating a design model in an object oriented CASE tool.
- **Facts Generation:** to generate a deductive database, corresponding to the facts extracted from an OO design. These facts are expressed in predicates according to a metamodel for object-oriented design representation.
- **Expertise Capture:** to capture knowledge about good and bad OO constructions, generating a deductive database of design patterns and anti-patterns.
- **OO Design Analysis:** to analyze a design model stored in the facts deductive database using an inference machine, machine learning techniques and the design patterns and anti-patterns knowledge base. This analysis detects design fragments that can result in future maintenance and reuse problems, showing hints to the refactoring of those problematic constructions into more flexible ones. This module also allows the identification of design patterns used by developers in an OO development.

As mentioned before, in most cases the source for the identification of those constructions in legacy OO systems is the source code, since design models, when available, are outdated or too superficial to be used as the main input to this task. Therefore, we need to extract design information from source code. Considering that we would also like to use this infrastructure when developing new OO systems, we decided to add not only this extraction module, but all other OOPDTool modules to a very popular OO CASE tool: Rational Rose [23].

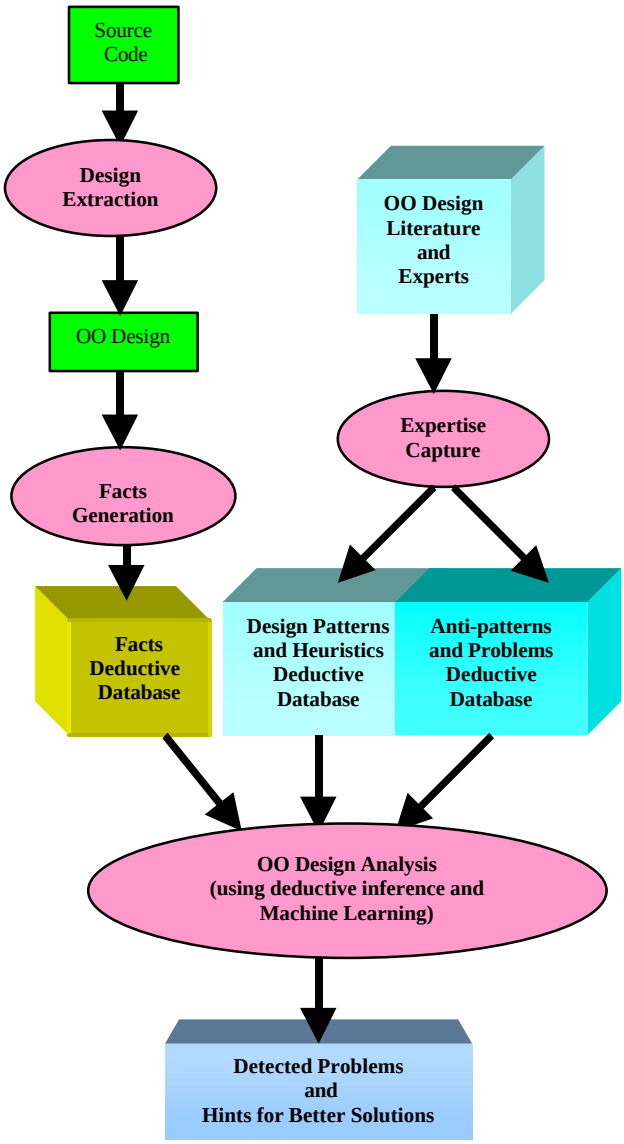

Figure 1: OOPDTool - Architecture

Rose is a CASE tool that supports OO analysis and design activities, being able to generate models according to Booch, OMT or UML notation. The information about an OO model can be accessed through OLE components, available in REI (Rose Extensibility Interface). Rose can also be customized, allowing the addition of new features as "Add-Ins". By using the features described above, we added all OOPDTool features to Rose, transforming it into an environment that supports the construction of object oriented design models and the evaluation of those models against candidate problematic constructions.

In the following subsections OOPDTool modules are described in detail.

4.1. Design Extraction

Although reverse engineering modules are available in Rose for languages such as C++ and Java, the information extracted are limited to those that can be obtained from a structural analysis of the source code. In C++, for instance, this information is captured from the analysis of "header files". Therefore, only structural information such as packages, classes, attributes, methods, and inheritance relationships can be retrieved, narrowing the scope of patterns and anti-patterns detection. To overcome this limitation, we need a design extraction module that can be able to retrieve not only the structural information captured by Rose reengineering modules, but also information related to the methods implementation. This extraction is done according to a metamodel that defines the concepts needed to the facts deductive database generation, which will be presented in the next subsection.

Based on the information retrieved by this extraction, it is possible to know the attributes manipulated by a particular method; how these attributes are manipulated (read, write, parameter, operation invocation); a method stereotype (*'constructor"*, *"destructor"*, *"read accessor"*, *"write accessor"*, among others), and which collaborations are necessary to the implementation of a particular method. The analysis of method invocations is done in order to gather information about dependencies between types and not between objects, since the latter would require a run-time analysis of the system. For each method, OOPDTool generates a collaboration diagram with only the direct calls presented in the method.

```
class   TapeRental {
public:
   void addTape(Tape aTape) {..; aTape.price(); .. }
};

class Tape {
private:
   Film   theFilm;
public:
   currency price() {...;theFilm.price(); ... }
};

class Film {
public:
   void price() { ...; }
};
```

Figure 2. Collaboration example

In the example shown in figure 2, OOPDTool would

extract information about the *addTape* method corresponding to the invocation of the *price* operation of an object of the *Tape* class or some descendant, meaning that the *TapeRental* class is dependent on the *Tape* type. Besides, the information about the invocation of the *price* method of the *Film* class would be attached only to the direct caller, i.e., the *price* method of the *Tape* class. In this way, from each method, we extract only the operation calls directly invoked from it, as we are only interested in the direct dependency relationships between classifiers.

From each operation invocation identified in the implementation of a particular method, we capture the following information: the operation called, the type (class or interface) of the object referenced by the method, and how this object is being accessed by the method (i.e., as a parameter, a locally created object, a global object, an attribute, or self). Considering the example shown in figure 2, in the *aTape.price()* collaboration present in the *addTape* method of the *TapeRental* class, the called object (*aTape)* is a parameter, while in the *theFilm.price* collaboration present in the *price* method of the *Tape* class, the called object (*theFilm*) is an attribute of the invoker object.

4.2. Facts Generation

Once the design information becomes available as a Rose model, either as the result of a reverse engineering process performed by the Design Extraction module or as a result of a model construction in a forward engineering process, the Facts Generation module comes into action, generating a deductive database which represents the facts captured from this design information.

These facts are stated in predicates corresponding to the constructions defined in a metamodel for object oriented software. This metamodel was defined based on the UML semantics metamodel [2] and on other papers in this field [9], [18]. This metamodel defines the entities and relationships that are relevant to design patterns and anti-patterns identification, including not only structural elements, but also dynamic elements such as object instantiation and method calls, for instance. In Appendix A, all predicates defined by the metamodel are presented in detail.

The metamodel defines the main entities of an OO design (package, classifier, attribute, operation, parameter), relationships between those entities (dependency, realization, inheritance), and also elements corresponding to the implementation of methods such as an object instantiation, an object destruction, a method invocation, and an attribute access.

From a structural point of view, the Facts Generation module generates a set of facts that describe all the classifiers found in a model (classes, interfaces, and basic data types), how those classifiers are organized into packages, their attributes and operations, including detailed information about each one (visibility, type, scope, parameters, among others). The associations, aggregations, and compositions are captured as pseudo-attributes [2]. The inheritance relationships between classifiers and the realization of a classifier are also captured by specific predicates.

From a behavioral point of view, this module generates facts about the implementation of each method. Each object instantiation and destruction, method invocation (of the same class or not), and attribute access (read or write) are captured as predicates. The capture of these behavioral elements is essential for the identification of many design problems, such as those related to object coupling.

The information generated by the Facts Generation module is the source for pattern detection, representing the result of the analysis of structural and behavioral elements of an OO design.

4.3 Expertise Capture

Both good and bad constructions that OOPDTool is able to detect in OO designs need to be defined beforehand. The tool stores design patterns and anti-patterns as deductive rules. Good constructions are taken from patterns and heuristics available in the technical literature and also from the knowledge accumulated by OO experts in many projects, formalizing them in the form of rules that define the characteristics necessary to the recognition of a particular construction in an OO design model.

Since each design pattern and anti-pattern is captured by rules, OOPDTool allows the definition of new rules so that new patterns and anti-patterns can be detected. Therefore, the knowledge base can evolve as a result of the organization experience in developing and maintaining OO systems. These rules are expressed using the same predicates employed in the OO design facts representation, as described earlier.

Figure 3 shows an example of a design pattern formalization. This rule allows not only the *AbstractFactory* [10] pattern detection, but also the identification of all the participants in a particular instance of this pattern. In the *Abstract Factory* pattern, those participants correspond to the *AbstractFactory*, the *ConcreteFactories*, the *AbstractProducts*, and their respective *ConcreteProducts* subclasses.

The rule illustrated in figure 3 defines the conditions necessary for detecting the *AbstractFactory* pattern. This rule uses the auxiliary predicates listed in figures 4 to 9.

```
abstractFactoryPattern(AbstractFactory,ConcreteFactories,
AbstractProducts,ConcreteProducts)
:- abstractFactory(AbstractFactory,AbstractProducts),
findAll(ConcreteFactory,concreteFactory(ConcreteFactory,
AbstractFactory), ConcreteFactories),
findAll (X, concreteProduct (X, AbstractProducts),
ConcreteProducts).
```

Figure 3 – Abstract Factory Pattern (Prolog rule)

```
abstractFactory (Class, AbstractProducts) :-
classifier(_, Class, _, "Class", "Abstract", "NotLeaf", "Root"),
findAll(AbstractProduct,instantiatesAbstractProduct(Class,
AbstractProduct),AbstractProducts), listSize(AbstractProducts,
Size),
Size > 1).
```

Figure 4 – Identification of an A*bstractFactory* participant

```
concreteFactory (Class, AbstractFactory) :-
classifier (_, Class, _, "Class", "Concrete", _, _),
descendant (Class, AbstractFactory),
abstractFactory(AbstractFactory, AbstractProducts),
findAll(ConcreteProduct, instantiatesAbstractProduct (Class,
ConcreteProduct), ConcreteProducts),
listSize(ConcreteProducts, Size), Size > 1).
```

Figure 5 – Identification of a *ConcreteFactory* participant, descendant from an *AbstractFactory*

```
abstractProduct (Product) :-
classifier(_,Product,_,"Class","Abstract","NotLeaf","Root"),
classifier(_,ConcreteProduct, _, "Class","Concrete",_,_),
ancestor (Product, ConcreteProduct).
```

Figure 6 – Identification of an *AbstractProduct* participant

```
concreteProduct (ConcreteProduct, AbstractProducts) :-
descendant (ConcreteProduct, X),
member (X, ListaAbstractProducts).
```

Figure 7 – Identification of an *ConcreteProduct* participant, descendant from an *AbstractProduct*.

```
instantiatesAbstractProduct (Class, AbstractProduct) :-
operation(Class, Oper, _, "Instance", "Public", _, "Virtual", _, _),
parameter(Oper, _, _, "Return", AbstractProduct),
abstractProduct (AbstractProduct).
```

Figure 8 – Verification whether a classifier has an operation that returns an *AbstractProduct*.

```
descendant (X,Y) :- inheritsFrom (X, Y).
descendant (X, Y) :- inheritsFrom(Z, Y), descendant (X, Z).
ancestor  (X, Y) :- inheritsFrom (Y, X).
ancestor  (X, Y) :- inheritsFrom (Z, X), ancestor (Z, Y).
```

Figure 9 – Identification of direct or indirect inheritance

In the same way, we can catalogue anti-patterns with rules defined from predicates corresponding to the basic OO constructions presented in the metamodel. For each anti-pattern formalized using Prolog rules, we also capture information that describes its general format, the main causes of its appearance, the consequences that this bad solution can generate and what should be done to replace this solution by a better one. This better solution can often point to the application of a design pattern.

The anti-patterns can be captured directly from the anti-patterns literature [18], or indirectly by looking for violations of well known design heuristics and patterns. Figures 10 to 12 show some examples of anti-patterns that we have formalized using Prolog, originated from possible violations of design heuristics.

The *PublicVisibility* anti-pattern (figure 10) detects the definition of attributes in the public area of a class. This contradicts the design heuristic: "All data should be hidden within its class" [24].

```
publicVisibility (Class, Attribute) :-
classifier(_, Class, _, _, _, _, _),
attribute(Class, Attribute, _, "Public", _, _, _, _, _).
```

Figure 10 – *PublicVisibility* anti-pattern

The *ProtectedVisibility* anti-pattern (figure 11) detects the definition of attributes in the protected area of a class. This fact contradicts the design heuristic "All data in a base class (superclass) should be private" [24].

```
protectedVisibility(Class, Attribute) :-
classifier(_, Class, _, _, _, _, _),
attribute(Class, Attribute, _, "Protected", _, _, _, _, _),
ancestor(Class, OtherClass).
```

Figure 11 – *ProtectedVisibility* anti-pattern

The *ExpositionOfAuxiliaryMethod* anti-pattern (figure 12) detects the definition of methods in the public interface of a class that are used only as auxiliary methods for the implementation of other methods of this class. This contradicts the design heuristic "Do not put implementation details such as common-code private functions into the public interface of a class" [24].

Beyond the anti-patterns derived from object oriented design heuristics, we can also identify other potentially problematic constructions by analyzing the problems discussed in design patterns. As an example of this strategy, we present the formalization of two anti-patterns related to some *creational* design patterns [10]. The rules defined in figure 13 detect the presence of an object with global scope, whose class would be better defined by applying the *Singleton* design pattern.

The *ManyPointsOfInstantiation* anti-pattern indicates that many points of the design are coupled to a particular class and, therefore, to a particular implementation. The design would be more flexible if a *creational* design pattern as, for example, *AbstractFactory* or *Prototype*, were used instead of the direct instantiation spread in many points of the software.

```
expositionOfAuxiliaryMethod (Class, Method) :-
classifier(_, Class, _, _, _, _, _),
operation(Class, Method, _, _, "Public", _, _, _, _),
findAll(ClientClass,externalClient(Method,ClientClass),
ClientClasses),   listSize(ClientClasses, Size), Size = 0,
findAll(ClientClass,internalClient(Method,ClientClass,
InternalClients),   listSize(InternalClients, InternalSize), InternalSize
> 0).

externalClient (Method, ClientClass) :-
operation (Class, Method, _, _, _, _, _, _, _),
operation (ClientClass, Caller, _, _, _, _, _, _, _),
invokes(Caller, Class, Method, _),
not (sameHierarchy(ClientClass, Class)).

internalClient (Method, ClientClass) :-
operation (Class, Method, _, _, _, _, _, _, _),
operation (ClientClass, Caller, _, _, _, _, _, _, _),
invokes (Caller, Class, Method, _),
sameHierarchy(ClientClass, Class).

sameHierarchy (X, Y) :- X = Y.
sameHierarchy (X, Y) :- descendant (X, Y).
sameHierarchy (X, Y) :- ancestor (X, Y).
```

Figure 12 – *ExpositionOfAuxiliaryMethod* anti-pattern

```
globalScopeObject (Object, Class) :-
classifier(_, "Logical View::Global", _, _, _, _, _),
attribute ("Logical View::Global", Object, _, "Public", _, Class, _, _,
_).

globalScopeObject (Object, Class) :-
classifier(_, X, _, _, _, _, _),
attribute(X, Object, "Class", "Public", _, Class, _, _, _).
```

Figure 13 –*Global Scope Object* anti-pattern

```
manyPointsOfInstantiation (Class, NPoints) :-
findAll (Creator, doInstantiation(Creator, Class), Creators), listSize
(Creators, Size), Size> NPoints).

doInstantiation (Creator, Class) :-
operation (Creator,Operation, _, _, _, _, _, _, _),
creates (Operation, Class, _).
```

Figure 14 – *ManyPointsOfInstantiation* anti-pattern

4.4. OO Design Analysis

The last OOPDTool module, the Design Analysis module, is responsible for analyzing the facts deductive database corresponding to the OO design being verified, and trying to find some match with the constructions captured by the Expertise Capture module, using the Visual Prolog 5 inference machine.

The user selects one or more problem categories, and one or more problems classified in the selected category. This module identifies all the design fragments that satisfy the Prolog rules defined for the detection of those problems.

A report is generated indicating each problem found, the elements responsible for its occurrence, and also possible ways to overcome it. These possible better solutions correspond to the anti-pattern information captured by the Expertise Capture module. For example, if the tool finds the *PublicVisibility* anti-pattern, it points the attribute and the class where it occurs, showing that a possible solution would be to move the attribute to the private area of the class, and to create accessor methods (get and set methods) for retrieving and modifying this attribute.

Another result that can be achieved with this module is the identification of design patterns used in the evaluated design. By selecting the desired design patterns, the user obtains as a result, a report indicating the design patterns found and also all the elements matching each participant role in the pattern. For example, If the tool detects an instance of the *AbstractFactory* design pattern, it would show not only the presence of this pattern in the design but also all classes corresponding to the *Abstract Factory, Concrete Factories, Abstract Products and Concrete Products* participants found in this design pattern instance.

5. Related Works

Several works related to the reengineering of legacy object oriented systems and design patterns detection have appeared in the last five years.

Cinnéide [6] describes a tool called Design Pattern Tool that does some refactorings in programs written in Java. This tool is limited to refactorings related to object instantiation anti-patterns.

Zimmer [30] reports experiences using design patterns, general OO design rules, and metrics in reorganization of object oriented systems. He describes a method used in the restructuring of a hypermedia OO application with the incorporation of more flexible design constructions. However, no supporting tool is mentioned in this work.

KT is a tool developed by Brown [3] that can reverse engineer OO designs from source code written in Smalltalk. KT detects three design patterns as described in [10]: *Composite*, *Decorator*, and *Template Method*. Algorithms specially designed to detect them do the

identification of instances of those patterns. However, the tool does not provide support to the detection of other patterns or even code written in other languages, because it searches for specific Smalltalk constructions using an algorithm also specific for the detection of each pattern.

Krämer [14] presents a supporting tool, Pat, for the design recovery process which searches for structural design pattern in an object oriented design model. The design constructions are also represented in Prolog. However, the patterns detected by this tool are limited since the reverse engineering task is done by a Case tool (Paradigm Plus) that is only able to recover structural design elements. Therefore, Pat cannot detect design patterns that require semantic information about behavior and collaboration between classes as, for instance, object instantiation, method invocation or attribute access. We consider that recovering information about the object collaborations, including the use of polymorphism, is indispensable for serious recovery of pattern-based design constructions.

6. Conclusions and Future Work

This work has shown that it is possible to provide automated support to the detection of problematic constructions in object oriented applications. We have incorporated this support as an "add-in" to a largely used OO CASE tool, Rational Rose. Besides providing support to the reengineering process of legacy OO applications, this tool can also be a valuable aid to the development of new systems. As the developer builds an OO model, he can use OOPDTool to help him in detecting potentially problematic constructions. This support can bring benefits such as maintenance cost reduction since it is much cheaper to fix a bad design fragment before it is already codified and tested.

By using Prolog rules to define the constructions detected by OOPDTool, it is easier to expand the scope of detection. It is possible to add new heuristics, patterns, and problematic constructions as new reports in the technical literature appear and developers gain more experience.

OODPTool can also be used to identify existing design patterns in an application, even if it were not designed with those patterns explicitly in mind. The design patterns identification makes it easier to understand a legacy system, and also highlights deficiencies in specific parts of the system or even the lack of knowledge of well-known patterns. This support allows the project manager to detect the need to invest on training particular team members in a way that they can learn and become proficient on the use of those standard design patterns.

Since our approach is based on a metamodel comprising not only static elements of an OO model such as classes, their operations and attributes, but also behavioral ones (method invocations, object instantiation, among others), it is possible to detect several design patterns and anti-patterns that are not captured by other approaches based solely on structural components.

We are currently working on a new version of OOPDTool, incorporating machine learning techniques, in particular Inductive Logic Programming (ILP) and Case-Based Reasoning (CBR) [19]. The main objective is to detect design constructions with structures similar to a particular pattern but with some sort of variation that makes it undetectable on an perfect matching algorithm using the Prolog inference machine. To achieve this goal, we are using notions of similarity applied to first order logic based on works such as [21] and [22]. As future work, we want to extend the tool so that it will be able to learn new patterns and also refine existing ones. This will increase the accuracy of the pattern detection process.

Appendix A – Metamodel predicates

The predicates used in the representation of OO design constructions are :

- *package(Name,Stereotype,ParentName)*: represents a package definition. A package is a general mechanism used for organizing model elements in groups.
- *packageDependency(PackageClient, PackageSupplier)*: represents a dependency relationship between two packages: one is a client and the other is a supplier.
- *classifier(Package,ClassifierName,Stereotype,Type, Abstract/Concrete,isLeaf,isRoot):* indicates a class, interface or basic data type definition.
- *visibilityInPackage(Package, Classifier, Visibility)*: represents the classifier visibility (public, protected, private) in a package.
- *realizes(Classifier,RealizedClassifier):* represents the existence of a realization relationship between two classifiers as, for example, a class implementing the operations defined by an interface.
- *inheritsFrom(SpecificClassifier,GenericClassifier):* represents an inheritance relationship between two classifiers.
- *attribute(Classifier,AttributeName,Scope,Visibility, Type,Modifiability,Multiplicity,Aggregation):*

indicates the presence of an attribute or a pseudo-attribute (an association with a classifier) [21] in a class definition. It also captures the attribute scope (class or instance), its visibility (public, protected or private), if its value can be modified or not, its type, multiplicity (1 or many), and the semantic of the association with the attribute type (association, aggregation or composition).

- *operation(Classifier,OperationName,Scope, Visibility,Stereotype,Polymorphism,Abstract/ Concrete,Const/NonConst):* represents an operation defined in a class, indicating its scope (class or instance), its visibility (public, protected, private), stereotype (constructor, destructor, read accessor, write accessor, other), if the operation can be redefined by subclasses, if it modifies the object state, and whether it is only a declaration (Abstract) or a method implementation (Concrete).
- *parameter(Operation, ParameterName, Order, Direction, ParameterType):* represents a parameter expected by an operation. The direction indicates whether it is an input, output, input/output or a return value.
- *creates(Caller,Classifier,Constructor)*: represents the invocation of a class constructor resulting in an object instantiation. This predicate indicates which operation is responsible for the invocation (caller).
- *destroys(Caller, Classifier, Destructor)*: represents the invocation of an object destructor. This predicate indicates which operation is responsible for this invocation.
- *invokes(Caller, Classifier, Operation, AccessType)*: represents the invocation of an operation. Caller corresponds to the method where this invocation occurs, Classifier is the type of the called object. Operation is the name of the called operation and AccessType tells how the called object is known in the caller method. AccessType can be the object itself (self), a parameter, an object created in the caller method (local object), a global scope object or an attribute of the caller object.
- *access(Operation, Attribute, AccessType):* indicates that an operation accesses a particular attribute. This access can be a value retrieval or modification, an operation call or even passing this attribute as an argument in some operation call.

References

[1] G. Booch, "Object Oriented Analysis and Design with applications", 2nd ed., Addison-Wesley, 1994.

[2] G. Booch, I. Jacobson, J. Rumbaugh. "The Unified Modeling Language for Object Oriented Development – UML Semantics – version 1.1", URL: http://www.omg.org. – 1997.

[3] K. Brown. "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk" . Master Thesis. URL: http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm, 1997.

[4] W. Brown, R. Malveau, H. McCormick III, T. Mowbray. " Anti-patterns – Refactoring Software, Architectures, and Projects in Crisis", Wiley Computer Publishing, 1998.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, " Pattern-Oriented Software Architecture: A System of Patterns" , John Wiley & Sons, 1996.

[6] M. Cinnéide, P. Nixon. "Program Restructuring to Introduce Design Patterns. Proc. of the Workshop on Object Oriented Software Evolution and Re-Engineering", ECOOP, 1998.

[7] J. Coplien, D. Schmidt. "Pattern Languages of Program Design 1" . Addison Wesley, 1995.

[8] J. Coplien, " Software Patterns", SIGS Books, 1996.

[9] S. Demeyer, S. Tichelaar, P. Steyaert. "FAMOOS – Definition of a Common Exchange Model". URL: http://www.iam.unibe.ch/~famoos/InfoExchFormat/

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley, Reading, MA, 1995.

[11] I. Jacobson, M Christerson, P. Jonsson, G. Overgaard.. "Object Oriented Software Engineering". Addison-Wesley, Workingham, England. 1992.

[12] R. Johnson and B. Foote. Designing reusable classes. Journal of OO Programming, 1(2): 22-35, June 1988.

[13] A. Koenig. "Patterns and Anti-patterns". Journal of Object Oriented Programming. March-April 1995.

[14] C. Krämer, L. Prechelt. " Design Recovery by Automated Search for Structural Design Patterns in Object Oriented Software". Proceedings of the Working Conf. on Reverse Engineering, IEEE CS press, Monterrey, November 1996.

[15] D. Lea, Christopher Alexander. "An Introduction for Object-Oriented Designers". ACM SIGSOFT Software Engineering Notes 19, 1, 1994.

[16] K. J. Lieberherr. "Adaptive Object Oriented Software. The Demeter method with propagation patterns". PWS Publishing Company. 1996.

[17] R. Martin, D. Riehle, F. Buschmann. "Pattern Languages of Program Design 3". Addison Wesley, 1998.

[18] G. Maughan, J. Avotins. "A Meta-model for Object Oriented Reengineering and Metrics Collection". Eiffel Liberty Journal. Vol. 1, No. 4., 1998. URL: http://www.elj.com/elj/v1/ n4/metamodel/

[19] T. M. Mittchel. "Machine Learning" . McGraw Hill, 1997.

[20] Platinum Technology IP, Inc., Oakbrook Terrace, IL – Paradigm Plus - www.platinum.com

[21] J. Ramon, M. Bruynooghe, and W. Van Laer. "Distance measures between atoms". In Proceedings of the CompulogNet Area Meeting on 'Computational Logic and Machine Learning', pp. 35-41, 1998.

[22] J. Ramon and M. Bruynooghe. "A framework for defining distances between first-order logic objects. In Proc. of the 8th International Conference on Inductive Logic Programming".

Lecture Notes in Artificial Intelligence, pp. 271-280. Springer-Verlag, 1998.
[23] Rational Software Inc., Santa Clara, CA. – Rational Rose 98 - www.rational.com
[24] A. Riel, “Object Oriented Design Heuristics”. Addison-Wesley, 1996.
[25] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. “Object Oriented Modeling and Design”, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
[26] D. Schmidt, R. Johnson, M. Fayad, “Software Patterns”. Communications of the ACM, Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10, October 1996.
[27] Software Composition Group (SCG) - FAMOOS Project. URL: http://iamwwww.unibe.ch/~famoos.
[28] J. Vlissides, J. Coplien, N. Kerth. “Pattern Languages of Program Design 2”. Addison Wesley, 1996.
[29] J. Vlissides. “Patterns: The Top Ten Misconceptions”. Object Magazine, March 1997, SIGS Publications, Inc.
[30] W. Zimmer. “Experiences using Design Patterns to Reorganize an Object Oriented Application”. Proc. of the Workshop on OO Software Evolution and Re-Engineering, ECOOP, 1998.