The Essence of Design Patterns In Automatic Identification Tool

A. L. Freitas, *FURG – Fundação Universidade Federal do Rio Grande*

Abstract--**The emergence of the design patterns movement has gone a long way toward codifying a concise terminology for conveying sophisticated computer-science thinking. A design pattern is a reusable implementation model or architecture that can be applied to solve a particular recurring class of problem. In generally, it is hard to recognize pattern use in real-world ware systems, unless you know what you are looking for then carefully and methodically search for the pattern. The purpose of this research has been to demonstrate the feasibility of building programs to detect the use of software design patterns in Java programs. To this end this paper examines the structure of design patterns, determines the nature of what makes a design pattern detectable by automated means. The development of these examples allows patterns comparison, showing advantages and tendencies in using one or another kind of communication between classes and objects.**

Index Terms -- **Software Engineering, Object Oriented Programming, Pattern Recognition.**

I. Introduction

While object-oriented design methodologies and languages are in ever-increasing use, it is becoming recognized that it is harder to become an expert object-oriented programmer than it is to become an expert in traditional structured techniques. This is partly due to the fact that object-oriented builds upon structured techniques, and adds additional programming language and design features that must be comprehended, and large libraries that must be learned. However, the problem goes deeper than this.

The inherent activities to the software design are activities which use the personal experience and human intelligence. Therefore, the use of models must help to solve problems from any object-oriented design, no matter what the area of the application. A pattern matches up to a sketch of an architecture where the involved classes, their responsibilities and help are defined. To use a pattern in a design consists of including the classes of the sketch in the structure of classes of application or making the former classes become responsible for the classes of the sketch in order to put into the system the functions wanted [4].

For maintenance, reuse, and re-implementation, software developers frequently need to examine source code to understand object-oriented software systems. The ability to learn and understand software systems from source code is greatly enhanced by visualizing the software systems at higher levels of abstraction, rather than seeing them nebulous collections of classes and methods implementations [1].

Visualizing object-oriented programs as a system of patterns interacting requires detecting, identifying and classifying groups of relate classes in program code. These visualizations represent either known patterns that perform an abstract task and are not necessarily a known pattern solution. Aiming to formalize the development process of object-oriented software, this work proposes to identify the design patterns essence. Heuristics are created for identify and apply design pattern in object-oriented programs. This heuristics are applied in a tool implemented in Java that automates identification of design patterns in object-oriented applications.

This paper is organized like this: section 2 presents a study on patterns identification tools. The section 3 presents the characteristics about relationships and collaborations in design patterns and a study about Composite is carried out. The Section 4 shows the design patterns automatic identification tool. These tool aggregates the characteristics described in Section 3. In the finish we have the conclusion emphasizing the contribution.

II. Related Works

A lot of work is currently being done in both scientific contexts towards identifying design patterns, building support tools, etc.

The Krämer approach [8] presents a tool whose objective is the investigation of structural patterns starting from the code source. The denominated tool Pat System executes the extraction of pertinent information of a file source in C++ and it stores them in a repository of data. The patterns are expressed as Prolog rules and the extracted information as facts. Therefore, through consultations the information, the author proposes the research of the patterns.

The Bansiya work [1] proposes a tool that automates the discovery, identification and classification of design patterns starting from applications source in C++. The approach uses heuristic, derived empiric information of the design and metric of source code for identification of patterns. That approach for the discovery of patterns focalizes the fundamental structural relationships of interest for the identification of: inheritance, aggregation and use.

Seeman [9] presents as recovering design information starting from applications source in the language Java. That work characterizes an approach process based on several

This work was supported in part by the FAPERGS – Fundação de Amparo a Pesquisa do Estado do Rio Grande do Sul.

A. L. Freitas is with the Department of Mathematical, Fundação Universidade Federal do Rio Grande, Rio Grande, RS (e-mail: dmtalcf@super.furg.br).

growing layers of abstraction. The compiler collects information on inheritance mechanisms, collaborations and calls to methods. The result of this phase is a graph on which a grammar is applied, the one which together with some criteria, it seeks to propose the identification process.

The Guéhéneuc [7] approach shows that design patterns describe micro-architectures that solve recurrent architectural problems in object-oriented programming languages. It is important to identify these micro-architectures during the maintenance of object-oriented programs. But these micro-architectures often appear distorted in the source code. He presents an application of explanation based constraint programming for identifying these distorted micro-architectures.

The works of Krämer, Bansiya, Seeman and Guéhéneuc shows the focus on the static model of the application, in other words, on a source code an inspection of the entities is accomplished to determine possible identifications. The authors emphasize the need of adaptations in if treating that the model doesn't supply necessary semantic information to the identification of several patterns. The experiment here described it demonstrates the investigation process starting from applications in runtime.

III. Design Patterns

A pattern is a way to provide information in the form of a problem statement, some constraints on the problem, a presentation of a widely accepted solution to the problem, and then a discussion of the consequences of that solution. We are particularly interested in are software design patterns, which specifically deal with common problems in object-oriented design. Design patterns can be thought of as micro-architectures for solving particular design problems.

The pattern describe how methods in a single or sub-hierarchy of classes work together, more often, it shows how multiple classes and their instances collaborates. The proposal described by Gamma [3] presents a catalogue of patterns. The aim of this catalogue is to connect the problems of project more commonly found in the building of frameworks with how these problems can be solved.

A. Composite Pattern

The Composite Pattern allows you to build complex objects by recursively composing similar objects in a treelike manner. The pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring all of the objects in the tree to have a common super class or interface [4].

The key to use the pattern are two classes: one that represents simple (or Leaf) objects and one that represents an objects group. The objects group or Composite, acts like a Leaf by delegating its behavior to the objects in the group. Both classes support the same core interface, allowing clients to collaborate with the interchangeably. The Composite itself takes advantage of the common interface because its group members can include both Leaf objects and Composites. A Composite can contain other Composites and so on until the final Composites contain nothing but Leaves. The result is a tree of Composite and Leaf objects.

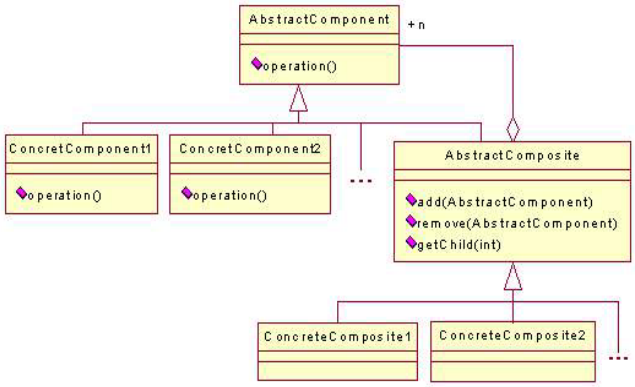

Fig. 1. Composite Abstract Structure

In generally the Composite class defines the behavior of the Composite object, such as *addChild*, *removeChild* methods etc.

B. Recursive Connection 1:N Metapattern

The Composite pattern shows a structure called metapattern with essence. In the paper described in Pree [5] some metapattens are made. In most cases the metapatterns offer a good level of flexibility considering changes of behavior because when the template and hook methods are in different classes the creation of references between the classes is needed. This reference can appear in a constant or variable way through mechanisms of association. The mechanisms of association in the metapatterns can be implemented by using attributes in the template and hook classes.

The recursive connection metapattern presents a relationship where the hook class is a super class in the hierarchy. It implies that the subclass calls the same definite methods in the super class. The recursive connection 1:N metapattern is also characterized due to the template class keeps reference to more than one object of the hook class.

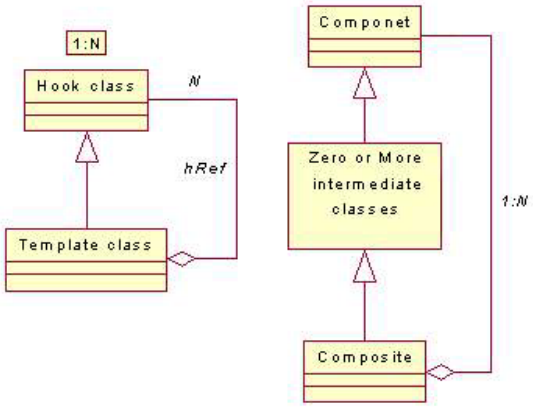

Fig. 2. Recursive Connection 1:N Metapattern and Pattern Essence

IV. The Analyzing Tool

This section presents a tool which automates the detection, identification and classification of design patterns in Java programs. This identification is made through the processes of reverse engineering and computational reflection.

The reverse engineering has as objectives to extract information of the software specification for subsequent analysis in the intention of identifying the components of the application and your relationships. Starting from the Java source code is made the identification of the application components through a cross-reference generator. The cross-reference generator was implemented using the ANTLR tool. ANTLR, ANother Tool for Language Recognition, it is a tool that provides a framework for construction of recognizer, compilers, and translators starting from grammars containing descriptions for applications C++ or Java [6].

After the construction of the cross-reference report the attributes will be her appraised and selected for sending of information for the tool in the subsequent execution process. The application designer becomes a package of the tool and as such it should be compiled. After the compilation the tool executes the application and in a united process with the execution the reflection computational is characterized.

The computational reflection defines architecture in levels, denominated reflexive architecture. In a reflexive architecture, a computational system is seen as incorporating two components: one representing the object, and other the reflexive part. The object located in the base level, it has for objective to solve problems and to come back information on the application domain, while the reflexive level, located in the goal-level, it solves the problems and it comes back information about the object computations, could add extra functionality to this object.

A. The Tool Processing

The tool presents eight different processes, represented starting from the figure 3, and described to proceed:

- 1. Generation of the cross-reference report: starting from the configuration file (*java.g*), which represents a translator's grammatical description, the referring files are generated to the processor of cross-reference, with the aid of the tool ANTLR. Of ownership of those files already compiled, the source application designer will be, then, submitted to the cross-reference process. As final product of this phase, a report is characterized which is used as entrance for the following process;
- 2. Generation of the new source code: this process uses as entrances the cross-reference report and the original source file. Modifications, as directing of control, they are proposed in the original application so that this application can supply, later, information for the inspection tool;
- 3. Compiler: it is made an external call to the standard compiler, in the intention of compiling the modified application, for the generation of the byte-code;

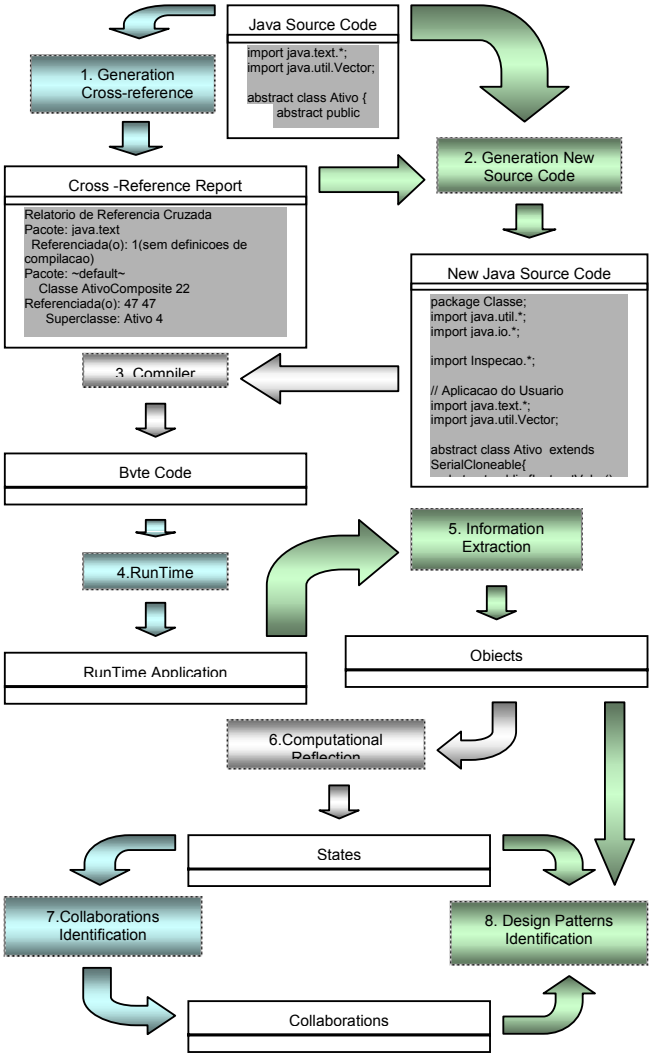

Fig. 3. Tool Processing

- 4. Runtime: the modified application behaves as a package of the inspection tool and, therefore, a call to the main method (*runprocessing*) it places it in execution;
- 5. Information extraction: due to the application in execution a line of execution of the tool worries about the extraction of the information of the objects used in the application. For each object used in the user's application a copy of your characteristics it will be generated;
- 6. Computational reflection: in this process they are appraised all the states for the which the objects have been represented during the execution of the application, as well as your attributes, methods, etc;
- 7. Collaborations identification: all the collaborations are verified between classes and objects. In this process, in runtime, all the associations and aggregations are verified that link to a class or object;
- 8. Design Patterns identification: finally, of ownership of the classes and objects, involved, your states, your collaborations, in a static and dynamic way, identify some design patterns used in the literature.

B. Runtime Tool

The following example in figure 4 represents a class hierarchy which shows a financial control called *Ativo* (Component class). *Ativo* allows the client to make his account. A client often wants to know about the value of his business which is determined by summing the value of all his property. A client can also want to know if he has a specific property (*Garantia*). This is determined by searching a *Garantia* object in the *Ativo* hierarchy.

In the first moment an *AtivoComposite* objects is created called *listaMonetario* which initializes the *ativos* attribute (Vector class) through new message. It is sent *adicionaAtivos*: message to *listaMonetario* which achieves the insertion of the leaf object (*Garantia*) in a list in order. The four objects: *aplicacoes*, *conta corrente*, *poupanca* e *acoes* are inserted in the list.

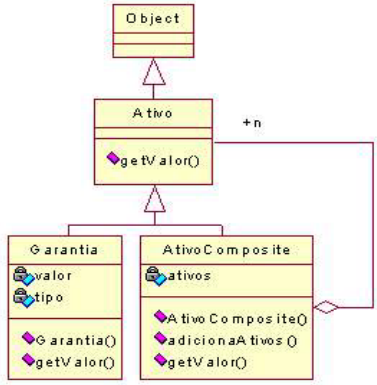

Fig. 4. Java Application

The mechanism of insertion in *AtivoComposite* allows us to insert the *listaMonetario* as an element of *listaBens* collection. The structure bellow shows an object tree.

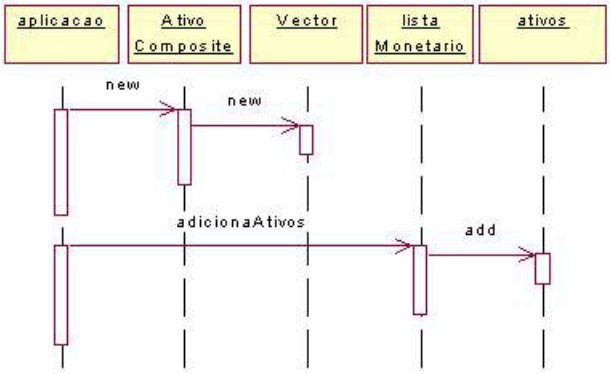

Fig. 5. Message Diagram

At the end the test message is sent to *listaBens*. This message activates the tool where it is possible to check the objects structure. The last line of the code researches in the tree using *getValor()* method which makes a comparison of the Leafs objects in the structure in a recursive way.

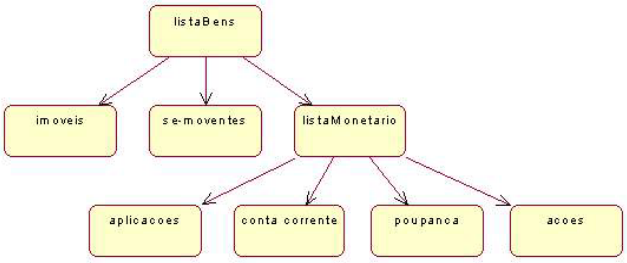

Fig. 6. Objects Diagram

The *getValor()* method sends a *getValor()* message to each Leaf object or this method recursively researches in another list in the case of Composite objects. The result is a add of values of *valor* attribute.

After the conclusion of the cross-reference it happens the transformation of the designer application where the method *main()* it is modified for a method *runProcessing()* because in run-time the designer application will behave as a package used by the inspection tool. All the identified objects in the main application will be selected for capture of information in subsequent run-time. Through the sending of these objects for a control Thread, the tool can, in run-time, to characterize the inspection of these elements.

Finished the modification process the compilation of the designer application is begun. The compilation process activates JDK compiler through an external call. Considering the compilation concluded without problems the tool it passes for the phase of package execution. During the execution of the application control Thread receives, in intervals of time, the extracted objects. The extraction is made for each object that receives different messages in the designer application and for so much the tool evaluates the states of these objects. If in the evaluation, the object to show different state from the stored previously this it will be considered, otherwise it will be discarded.

After the conclusion of the designer package the tool will present the interfaces of: cross-reference, objects found states and identified objects classes, according to figure 7. In the cross-reference the designer can precede the search in the source application of the identified entities. In this case the tool appears for the object *listaBens* of the class *AtivoComposite*. This object is presented in a window for verification of your states as well as a window for verification of your class, super class and existent attributes in these.

In the states interface they are presented the states that the object characterized in run-time. The designer can select what judges more pertinent and to proceed the inspection of these elements, through the option *VerificaEstado*.

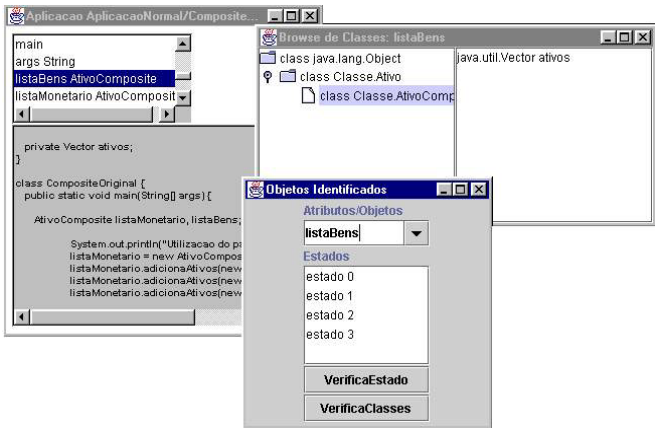

Fig. 7. First State Interface

In the case to follow the state 3 had been selected for the object *listaBens*. Starting from the identification of the collaborations of this with other objects/classes is made the visual and textual presentation of your relationships. The tool distinguishes the objects now and puts all the ones which are not literal for the compiler as a reference. The *ativos* attribute of *AtivoComposite* class presents a relationship with two *Garantia* objects and another *AtivoComposite* object. The second *AtivoComposite* presents a relationship with a four *Garantia* objects (Figure 8).

Fig. 8. Objects Structure

The window of textual inspection, figure 9, presents the content of the objects attributes identifying new objects associated by the attributes. This window identifies all the associations with other objects.

The window of the classes diagram shows the relationships between classes and objects. The figure 10 shows the *AtivoComposite* classes is related to *Garantia* and *AtivoComposite* classes.

In window of detection the class diagram which represents the pattern which was used in the evaluated application will be also available if we respect the predefined rules. Figure 11 shows the Composite pattern where an *AtivoComposite* object is related to objects of the *Garantia* type or *AtivoComposite* objects. Therefore the relationships happen to the *Ativo* super class. Therefore the evaluated object will have to present a relationship of 1:N with the classes which have the same hierarchical structure which is represented by the same abstract super class.

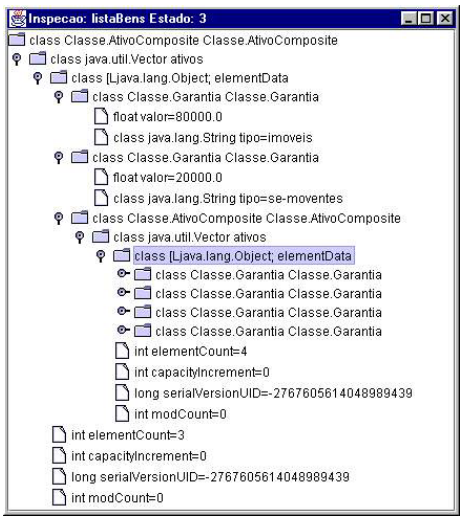

Fig. 9. Textual Objects Structure

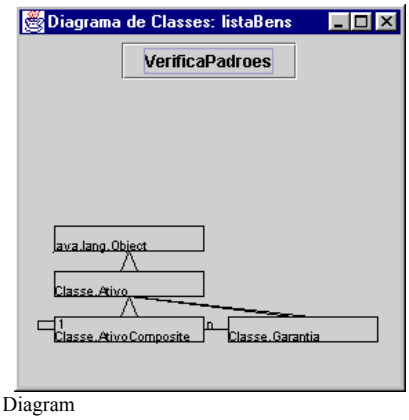

Fig. 10. Class Diagram

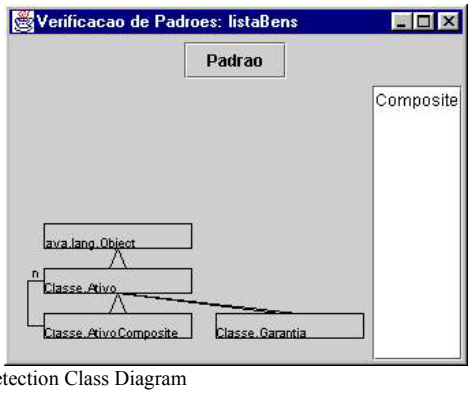

Fig. 11. Detection Class Diagram

To provide the identification of the pattern Composite the tool it uses the rules to precede mentioned. Beginning for the definition of the classes identifies the existence of two sub-classes starting from the abstract class A: C that demonstrates the objects Composite structure and L that it characterizes the objects leaf structure (ConcretComponent).

The C class demonstrates, also, a reference to objects of the abstract super class. This reference is established by an attribute in list format identified by LISTATRIBS. Each attribute of the list, denominated At (ATRIB), it represents an object of the classes C or L.

The aggregation mechanism and multiple delegation is

defined when it exists an association and a reference among the object in evaluation (OC - object Composite) and the objects of the list. Each association should translate a relationship with objects of a similar class (other objects Composite) or with objects that represent subclasses (objects Leaf), all of the same class appraised root. For the objects marked as composed it repeats the procedure, recursively, in the verification of the existence of a hierarchy in tree. These evidences, therefore, they characterize strong indications for the existence of the Composite pattern.

Extending the notation used in the Seeman [9] work it can be deduced the following formalism:

CLASS(C) = {C | C *extends* A ^ C *references* A}
CLASS(L) = {L | L *extends* A}
LISTATRIBS(LAs) = {LAs | LAs *attrib* C ^ LAs = LIST}

Label_Composite(OC)
ATRIB(At) = {At = LAs[n] | ∃At ∈ C v ∃At ∈ L}
∃OC ∈ C: ∀At: OC *aggreg*(*multiple*) At ^ OC *delegates* At
∀OC ∈ C: OC *agreg* At ⇔ OC *assoc* At(C) ^ OC *references* At(C) v
OC *assoc* At(L) ^ OC *references* At(L)
∀OC ∈ C: OC *delegates* At ⇔ ∀m_1 ∈ OC: m_1 *calls* m_2 ^
OC *owns* m_1 ^ At *owns* m_2
⇒ $\phi(m(OC))_1 = \phi(m(At))_1$
∀At ∈ C: Label_Composite(At)

The contribution presented in this formalism it is the verification of the essence of the pattern starting from the dynamic structure of the application and just not taking in consideration the static structure as in the works of: Seeman [9], Bansya [1] and Guéhéneuc [7]. Therefore, besides the pattern identification, it can also be verified if this is well used taking in consideration the entrance information and exit of the application.

Finally the designer, after visualizing the diagram of detected classes, he can select the name of the pattern identified and to press the option *Padrão* which will show a version of the original pattern considered in the literature to make possible comparisons with your package.

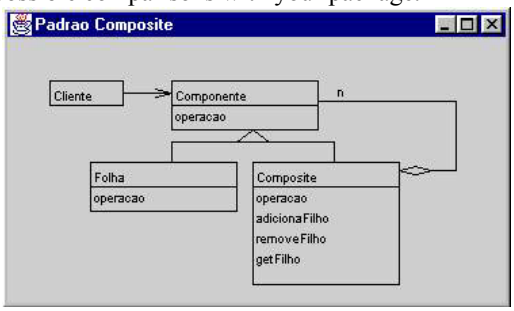

Fig. 12. Composite Diagram

V. Conclusion

The approach of identifying design patterns relies on reducing the knowledge of patterns to the minimum necessary and identifiable structures required by the pattern solutions. However, an approach based solely on pattern structures is not complete because pattern structures are not sufficiently unique. Several patterns tend to use similar basic structures.

Strategies for analyzing collaboration among classes are still immature. The main contribution of this paper is a prototype of tool for applications analyzing on searching for design patterns automatically. By developing examples we have had a visualization of a few conceptual patterns.

This work also intends to give an example of how to use the mechanisms for implementing design patterns. Heuristics that comes with the design patterns helps the construction of new projects because they are suitable to direct the development of activities which needs designer's personal thinking. However the use of design patterns does not lead designers to obtain definite answers for the problems at issue. On the other hand it establishes some ideas to optimize the construction of object oriented software.

The emphasis of the inspection tool ponders, therefore, in supplying subsidies to the designer regarding the execution of the application. The tool has mechanisms of visualization of the information regarding the states of the appraised objects. The tool has conditions of disposing the characteristics of the objects along your life cycle.

The tool is still being built but it has already implemented the identification of some patterns with Composite, Decorator, Strategy and Observer. We intend to gradually increase the tool with more case studies of patterns from real designs as bigger the samplings then better to certify the tool.

To reduce erroneous identifications it's necessary to extend the approach to use design heuristics and empirical data in resolving the presence of patterns and pattern-like solutions. The heuristics and empirical data will be derived from design and implementation metrics, which evaluate the structure and functional characteristics of classes and relationships.

VI. Acknowledgment

The author gratefully acknowledge the contributions of Terence Parr for their ANTLR software.

VII. References

Periodicals:

[1] J. Bansiya, "Automating Design-Pattern Identification", Dr.Dobb's Journal. New York, v.23, n. 6, pp.20-26, Jun. 1998.

Books:

[2] S. Alpert, "The Design Patterns - Smalltalk Companion". Reading: Addison-Wesley, 1998.

[3] E. Gamma, et al, "Design Patterns: Reusable Elements of Object Oriented Design", Reading: Addison-Wesley, 1994.

[4] M. Grand, "Patterns in Java: A Catalog of Reusable Design Patterns with UML", [S.l.]: John Wiley & Sons, 1998.

[5] W. Pree, "Design Patterns for Object-Oriented Development" Reading: Addison-Wesley, 1995.

Technical Reports:

[6] T. Parr, "What's An ANTLR ?", Available: http://www.antlr.org.

Papers from Conference Proceedings (Published):

[7] Y. Guéhéneuc and N. Jussien, "Using Explanations for Design Patterns Identification" in *Proc. 2001 Workshop on Modelling and Solving Problems with Constraints*, pp. 296-303.

[8] C. Krämer and L.Prechelt, L. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software" in *Proc. 1996 Working Conference on Reverse Engineering*, pp. 208-215.

[9] R. Seemann and J. Wolff von Gudenberg, J, "Pattern-Based Design Recovery of Java Software" in *Proc. 1998 Symposium on Foundations of Software Engineering*, pp. 10-16