# On the Role of Design Patterns in Quality-Driven Re-engineering [*]

Ladan Tahvildari and Kostas Kontogiannis
Dept. of Electrical and Computer Eng.
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
{ltahvild,kostas}@swen.uwaterloo.ca

## Abstract

*Design patterns have been widely adopted and well investigated by the software engineering community over the past decade. However, their primary use is still associated with forward engineering and the design phase of the software life-cycle. In this paper, we would like to examine design patterns from a different perspective namely, their classification and usage for software re-engineering and restructuring. Specifically, twenty three design patterns originally presented in the "Gang of Four" book are re-classified for re-engineering purposes into two major categories, primitive and complex. Moreover, their relationships and their impact to specific re-engineering objectives are presented in terms of a layered model that is denoted by six different relations namely: uses, refines, conflicts, is-similar-to, combines-with, and requires. The paper also discusses how the classification scheme can be applied for the re-engineering and restructuring of object-oriented systems.*

## 1 Introduction

A growing number of researchers consider design patterns to be a promising approach to object-oriented systems development [2, 5, 6, 7, 13, 22, 24]. The main idea behind design patterns is to support the reuse of design information, thus allowing developers to communicate design information for a subject system more effectively. New design patterns are constantly being specified, and applied by several research groups [8, 12, 16, 17, 23, 31]. Moreover, numerous development tools that support the design pattern approach are currently being developed [4, 10, 11, 15, 30].

In [13], a catalogue of design patterns is presented. The catalogue not only lists a description of the patterns but also presents how patterns are related. Furthermore, the catalogue presents a classification of all design patterns according to two criteria: *jurisdiction* (class, object, compound) and *characterization* (creational, structural, behavioral). However, these relationships in [13] are described informally and each relationship appears to be different in its formalization from the other ones.

Building on related work on refactoring [12], we propose a classification scheme of the standard design patterns [13] in a way that we believe it can assist software maintainers to better assess the impact of these design patterns when applied to object-oriented software restructuring. This scheme is based on three *primary* relationships between patterns such as: i) a pattern *uses* another pattern, ii) a pattern *refines* another pattern, iii) a pattern *conflicts* with another pattern. This paper also describes three *secondary* relationships between patterns such as: i) a pattern is *similar* to another one, ii) two patterns *combine* to solve a single problem, iii) a pattern *requires* the solution of another pattern. We also show how these secondary relationships can be expressed in terms of the primary relationships. These classifications motivate us to update the catalogue and organize the design patterns into two layers representing different abstraction levels.

This paper is organized as follows. Section 2 discusses the motivation and objectives to prepare this layered catalogue in the re-engineering context. Section 3 presents related work. Section 4 presents a tabular view of all design patterns [13] and their relationships as they appear in the aforementioned catalogue. Section 5 classifies these relationships and Section 6 modifies the structure of the catalogue. Section 7 presents how it is possible to arrange the design patterns into layers representing different abstraction levels. Finally, Section 8 discusses an application scenario of the proposed classification and Section 9 provides the conclusion and insights of future work.

[*]This work was funded by the IBM Canada Ltd. Laboratory, Center for Advanced Studies in Toronto; also by the Ontario Graduate Scholarship(OGS) of Canada.

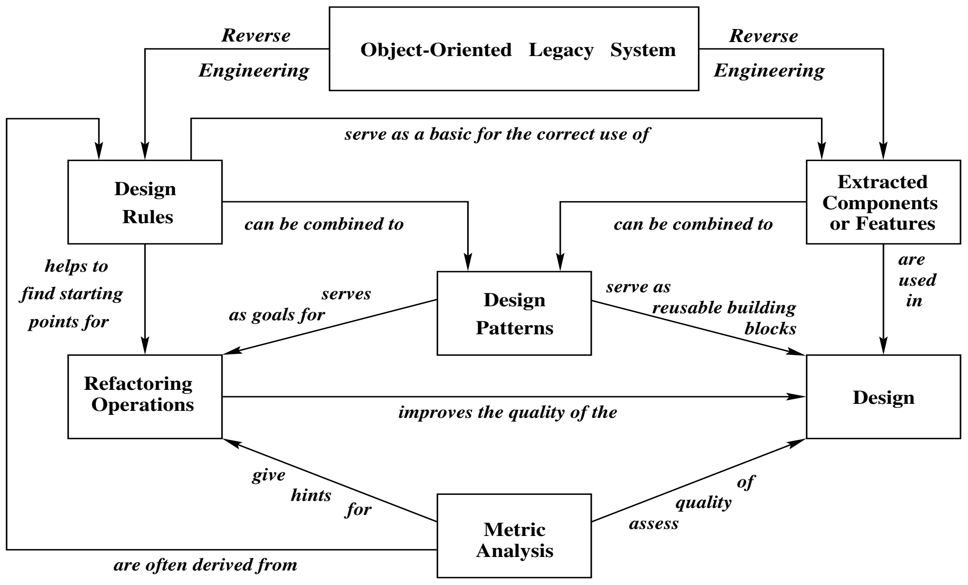

**Figure 1. Role of Design Patterns in Software Re-engineering.**

## 2 Motivation and Objectives

For our work, we are interested to investigate design patterns and their relationships as a means to restructure an object-oriented legacy system so that the new system conforms with specific design patterns and meets specific non-functional requirement (NFR) criteria. We call this approach "*Quality-Driven Object-Oriented Re-engineering*". Specifically, we aim to enhance through re-engineering and restructuring specific quality characteristics of the subject object-oriented system by migrating it into more maintainable forms [25, 26, 27, 28].

For achieving this goal, we need to develop a catalogue of specific design patterns and refactoring operations [12] that can be used to enhance specific software qualities during re-engineering namely, maintainability and performance. Performance is a vital quality factor in real-time, transaction-based, or interactive systems. Such systems have to handle incoming data and transactions, provide an interface with external users, and maintain persistent storage. Without good performance, such systems would be practically unusable, leading to lack of service, lack of information, clients who are dissatisfied, and corporate loss. Hence, it is important to consider performance requirements during re-engineering such systems. Also, a major motivation for re-engineering is to make applications more maintainable. This can be partially achieved with the use of design patterns. For example, *State* design patterns make it easier to add new states in a system without altering the functionality of the existing states. Ironically, the software engineer practice reports that improved methods during system maintenance and evolution result in higher maintainability indicators. Hence, it is important to consider how to improve maintainability during re-engineering activities.

Figure 1 provides an overview on how the design pattern approach is related to several other concepts of object-oriented software development as we plan to use in the proposed requirements-driven re-engineering process. As Figure 1 depicts the components or source code features that can be directly extracted from the object-oriented legacy system provide important information for guiding the developers to detect and apply candidate patterns in a way that can improve the quality of target system. In a nutshell, our re-engineering approach consists of : *i*) *requirements analysis* to identify specific re-engineering goals, *ii*) *model analysis* to understand the system's design and architecture which *Design Rules* provided by reverse engineering gives proper information, *iii*) *source code analysis* to understand a system's implementation through *Extracted Components and Features*, *iv*) *remediation specification* to examine the particular problem through *Refactoring Operations* and to select the optimal transformation for the system, *v*) *transformation* to apply transformation rules in order to re-engineer a system in a way that complies with specific quality criteria, and *vi*) *evaluation process* to assess whether the transformation has addressed the specific requirements set that *Metric Analysis* can do this job.

Once software artifacts have been understood, classified, and stored during the reverse engineering phase, their behavior can be readily available to the system during the forward engineering phase. The forward engineering phase aims to produce a new version of object-oriented legacy system that operates on the target architecture and meets specific non-functional requirements (*i.e.*, maintainability or performance enhancements). The proposed process is iterative and incremental in nature. It means that at each iteration cycle, an evaluation procedure is applied to ensure that each transformation step conforms with the requirements set for the new system.

As a part of this project, we have carefully analyzed the existing patterns presented by "Gang of Four" [13] to determine their relationships and their usage in the re-engineering area. The classification scheme for relationships between patterns presented in this paper was developed to support the wider goal of improving the quality and the design of migrant code while maintaining its original functionality (behavior preserving transformations).

## 3 Related Work

Design patterns were discussed by Christopher Alexander, an architect, in order to describe techniques for town planning, architectural designs, and building construction techniques [1]. Each design pattern description contains a section where relationships to other patterns of a higher or of a lower granularity level are presented. These relationships influence the construction process. A classification for the patterns was given, however their mutual relationships have not been provided.

In [13], a large collection of well described design patterns was presented. The relationships between design patterns are also described, but not classified. However a clustering of related design patterns was included. Such clustering according to *jurisdiction* (class, object, compound) and *characterization* (creational, structural, behavioral) is orthogonal to the one derived in this paper. In this context, patterns in a specific cluster can be considered as similar to another one which supports the selection of an appropriate design patterns for a certain problem.

Frameworks [14, 34] are also considered as high-level design patterns, usually consisting of many interrelated design patterns of lower levels.

In [2], it is indicated that "*Patterns can be used at many levels, and what is derived at one level can be considered a basic pattern at another level*". Furthermore, it is stated that "*This is probably typical of most architects; some patterns will be generic and some will be specific to the problem domain*" which also confirms the organization depicted in our proposed layers in Figure 3.

Booch [3] also discussed that design patterns are ranging from *idioms* to *frameworks*.

In [6], several design patterns are combined in an exemplary application, but the relationships are not investigated further.

The relationships between object-oriented design patterns were first analyzed in [5] where three kinds of relationships between patterns are described. These include: i) *use* - one pattern can *use* another pattern, ii) *variant* - one pattern can be *a variant of* another pattern, iii) *combine* - two patterns can be used *in combination* to solve a problem.

Similarly, Mesazaros and Doble [18] identified five relationships between patterns, a pattern can *use*, *be used by*, *generalize*, *specialize*, or provide an *alternative* to another pattern.

## 4 Overall Structure of the Catalogue

Table 1 is a tabular presentation in alphabetic order of the design patterns and their relationships that can be found in [13]. No further information nor new patterns are proposed in this paper. However, Table 1 provides an overview of the structure of the proposed catalogue. It serves as a reference point to the rest of the paper as it contains the most detailed information about the pattern relationships. This table also serves as the starting point for the further classification and the revision of the relationships. On the basis of Table 1, we build our proposed classification scheme that is presented in the following sections.

## 5 Classifying Relationships

Our classification scheme is based on the following relationships between any pairs $(X, Y)$ of design patterns found from the description in Table 1 :

- $X$ **Uses** $Y$

  When building a solution for the problem addressed by $X$, one subproblem is similar to the problem addressed by $Y$. Therefore, the design pattern $X$ *uses* the design pattern $Y$ in its solution [1, 5, 13]. Thus, the solution of $Y$ represents one part of the solution for $X$. It means that a pattern which has a *larger* or more global impact on a design will use patterns which have *smaller* or more local impacts.

  For example, *Chain of responsibility uses Decorator*. Tools supporting the design pattern approach can benefit from this information as such a relationship can be checked in existing designs. Also design patterns like $Y$ can be visualized as blocks without internal implementation details in order to raise the abstraction level. The *uses* relationship can also be used to simplify the descriptions of more complex patterns by composition.

  We can also consider the *used by* ($Y$ used by $X$) relationship which is the inverse of the *uses* relationship ($X$ uses $Y$) and can be analyzed in the same way

| Pattern Name (X) | Description of Relationship | Pattern Name (Y) |
|---|---|---|
| Abstract Factory | Are often implemented with<br>Can also be implemented using | Factory Methods.<br>Prototype. |
| Adaptor | Has a structure similar to | Bridge. |
| Bridge | Is similar to | Adaptor. |
| Builder | Is similar to | Abstract Factory. |
| Chain of Responsibility | Uses | Decorator. |
| Commands | Uses | Composite. |
| Composite | Is used for a<br>Can be used to implement<br>Creates composite<br>Defines simple instance of | Chain of Responsibility.<br>Commands.<br>Builder.<br>Interpreter. |
| Decorator | Supports recursive composition, not possible with<br>Is often used with<br>Have similar implementation as | Adaptor.<br>Composite.<br>Proxies. |
| Facade | Is similar to | Mediator. |
| Factory Method | Is usually called within | Template Methods. |
| Flyweight | Lets you share components such as | Composite. |
| Interpreter | traverses the structure by using | Iterator. |
| Iterator | Can used to traverse<br>Can be combined with | Composites.<br>Visitor. |
| Mediator | Is similar to | Facade. |
| Memento | Can keep state to undo its effect using<br>Is often used in conjunction with | Command.<br>Iterator. |
| Observer | Can use | Singleton. |
| Prototype | Is often a<br>Is competing pattern in some way with | Singleton.<br>Factory Method. |
| Proxy | Defines a representative for another object without changing the interface in the comparison with | Adaptor. |
| Singleton | Is used by | Observer. |
| State | Is similar to | Strategy. |
| Strategy | Is often best to implement as | Flyweight. |
| Template Method | Often calls | Factory Method. |
| Visitor | Localizes operations that would be distributed across<br>Can be used to maintain the behavior of | Composite.<br>Interpreter. |

**Table 1. Overall Structure of the Design Pattern Catalogue.**

as that relationship. For example, because *Interpreter uses Iterator*, *Iterator is used by Interpreter*.

- $X$ **Refines** $Y$

  A specific pattern $X$ *refines* a more abstract pattern $Y$ if the specific pattern's full description is a direct extension of the more general pattern $Y$ [18]. That is, the specific pattern $X$ must deal with a specialization of the problem the general pattern $Y$ addresses, and must have a similar (but more specialized) solution structure.

  To make an analogy with object-oriented programming, the *uses* relationship is similar to composition, while the *refines* relationship is similar to inheritance. For example, *Factory Method refines Template Method*, because Factory Methods are effectively *Hook Methods* [13] which are used by subclasses to specify the class of an object the Template Method in the superclass will create. In the description of Factory Method pattern, one of the main forces addressed by the pattern is the use of naming conventions to illustrate the particular method is in fact a Factory Method.

  We can also consider that the *refined by* or *generalizes* relationship ($Y$ is refined by $X$) is the inverse of the *refines* relationship ($X$ refines $Y$) and can be analyzed in the same way as that relationship. For example, as *Factory Method refines Template Method* then *Template Method is refined by Factory Method*.

- $X$ **Conflicts with** $Y$

  The third fundamental relationship between patterns in our classification scheme is *conflicts* which denotes that the two or more patterns provide mutually exclusive solutions to similar problems. Most pattern forms do not provide an explicit section to record this relationship but it is often expressed in the related pattern section along with the *uses* relationship.

For example, *Prototype and Factory Method patterns conflict* because they provide two alternative solutions to the problem of subclasses redefining the class of objects created in superclasses. When reading or applying a pattern, this relationship can be exploited in two ways. When looking for patterns, if a pattern seems as if it may be applicable, then the conflicting patterns should be examined because they present alternative choices but once a pattern has been chosen the other conflicting patterns can be ignored.

- $X$ **is Similar to** $Y$

  This relationship is often used to describe patterns which are similar because they address the same problem [5]. The similarity relationship seems to be much broader than just *conflicts*, and as it is also used to describe patterns which have a similar solution technique such as *Strategy* and *State*. These can be treated as *refining* a more abstract pattern, or occasionally related by *uses* relationship.

- $X$ **Combines with** $Y$

  *Patterns of Software Architecture* [5] introduces a *combines* relationship in the case where two patterns are combined to solve a single problem which is not addressed directly by any other pattern.

  In simple cases, we can model this relationship directly by the *uses* relationship where one pattern is a larger scale pattern that addresses the whole problem and the other pattern is a smaller scale pattern which provides a solution to a subproblem. For example, *Composite* and *Decorator* are often used together in applications [32]. There are also other kinds of relationships between them. For example, when looking at the solution aspect, *Decorator* can be seen as a degenerated *Composite*. When considering the pattern scope, they both support recursively structured objects whereby *Decorator* focuses on attaching additional properties to objects. Thus, the design patterns in this category are somehow similar but it is difficult to state this relationship more precisely. Therefore, we only insert a relationship of type "Combines with" and neglect the other ones.

  In more complex cases, we consider that this relationship really points to a lack in the patterns themselves. Although these patterns can be combined to provide a solution to a problem, the actual problem and the way these patterns are combined to solve it, is being represented by the *combines* relationship and it is not captured explicitly in a pattern. In these cases, we ensure that the problem is identified explicitly by locating an existing pattern or introducing a new pattern which addresses the problem directly, outlines the whole solution and *uses* the patterns that combine to solve it. For example, *Iterator* traverses *Composite* structures and *Visitor* centralizes operations on object structures. Depending upon the necessary degree of flexibility, one typically combines two or all three design patterns, for instance *Interpreter*.

- $X$ **Requires** $Y$

  We say that one pattern *requires* a second pattern if the second pattern is a prerequisite for solving the problem addressed by the first pattern. For example, *Composite can be used to implement Command*. It means that Command pattern *requires* Composite pattern before it can be implemented successfully.

  In general, we consider that this relationship can be modeled quite adequately by the *uses* relationship. The distinction between *requires* and *uses* seems to be based primarily on the order in which the patterns should be applied. If one pattern requires a second pattern, the second pattern must be applied before the first one can be used to produce its solution. This is also the case with the general *uses* relationship, since if one pattern *uses* a second pattern, the second pattern must be applied before the solution described by the first pattern will be completed. This is the reason that we can model the *requires* relationship with the *uses* relationship.

Based on our classification scheme, we can categorize those design pattern relationships one level further as follows :

1. **Primary Relationships**

   Our classification scheme can be based on three primary relationships such as : i) a pattern *uses* another pattern, ii) a more specific pattern *refines* a more general pattern, iii) one pattern *conflicts* with another pattern when they both propose solutions to a similar problem.

2. **Secondary Relationships**

   Our scheme also describes three secondary relationships between patterns such as : i) two patterns are being *similar*, ii) two patterns are *combining* to solve a single problem, iii) a pattern *requires* the solution of another pattern. We classify these relationships as secondary relationships because we have been able to express them in terms of the primitive relationships as discussed before.

| Pattern (X) | Relationship | Pattern (Y) |
|---|---|---|
| Abstract Factory | uses<br>refines | Template.<br>Prototype. |
| Adaptor | is used by | Bridge. |
| Bridge | uses | Adaptor. |
| Builder | uses | Abstract Factory. |
| Chain of Responsibility | uses | Decorator. |
| Commands | uses | Composite. |
| Composite | refines<br>conflicts with<br>refines | Chain of Responsibility.<br>Builder.<br>Interpreter. |
| Decorator | refines<br>uses<br>refines | Adaptor.<br>Composite.<br>Proxy. |
| Facade | is refined by | Mediator. |
| Flyweight | uses | Composite. |
| Interpreter | uses<br>uses | Iterator.<br>Visitor. |
| Iterator | uses<br>uses<br>uses | Composites.<br>Visitor.<br>Memento. |
| Mediator | refines | Facade. |
| Memento | uses | Command. |
| Observer | uses | Singleton. |
| Prototype | uses<br>conflicts with | Singleton.<br>Template. |
| Proxy | conflicts with | Adaptor. |
| Singleton | is used by | Observer. |
| State | refines | Strategy. |
| Strategy | uses | Flyweight. |
| Visitor | refines | Composite. |

**Table 2. Revised Classification.**

## 6 Modifying Relationships

This section examines the proposed classification of the design patterns relationships further. This process results in some modifications to existing relationships. The organization of the relationships in different categories is sometimes difficult because it partly depends upon subjective criteria. The difference between "$X$ *Uses* $Y$" or "$X$ *Combines With* $Y$" depends upon the subjective assessment whether the usage of $Y$ is seen as a central part of the solution $X$, or if it is more of a combination of two autonomous design patterns. Furthermore, two design patterns might be related in different ways. *Decorator / Composite* and *Abstract Factory / Prototype* are pairs of design patterns which can be combined and are also similar. In this paper, each relationship is assigned to the most adequate category.

Based on the catalogue, Factory Method does not *use* Template Method but it is often called (*refine*) by *Abstract Factory* in a Template Method. It means that Factory Method often plays the role of a primitive in a Template Method. Thus, if an Abstract Factory uses Factory Method in its solution, then it really uses the design pattern Template Method. Therefore, we do not need to consider Factory Method as a separate design pattern but as a "*X Uses Y*" relationship between Abstract Factory and Template Method. By removing the Factory Method design pattern, we need to modify the relationship between this pattern and Prototype appropriately. It means that from now on, the Prototype design pattern has a *conflict* relationship with Template Method instead of Factory Method.

The integration of the above modifications as well as the introduction of the *primary* relationships after expressing the *secondary* relationships in terms of *primary* ones into Table 1 results in Table 2.

In this context, Figure 2 is a graphical illustration of Table 2, and our proposed classification scheme.

## 7 Layers of Design Patterns

Up to now, we have classified the relationships between the design patterns and modified a few of them. As one can see in Figure 2, "$X$ *Uses* $Y$" is the most frequent relationship.

Therefore, we try to arrange the patterns according to this predominant relationship. The graph defined by the *primitive* relationships is acyclic. This property allows us to arrange the design patterns straightforwardly in different layers as shown in Figure 3.

These layered design structure techniques can also be useful in the reverse engineering context for extracting architecture as well as in forward engineering context for improving the design or enhancing non-functional requirements of existing code. These two different layers are discussed below.

### 7.1 Primitive Design Patterns

This layer contains the design patterns which are heavily used in the design patterns of higher level and in object-oriented systems in general. Table 3 gives a list of these patterns in alphabetic order and their respective purpose. The Composite design pattern seems to be the most important one as it is *used by* eight other design patterns. It is a basic pattern in the sense that the addressed problem of handling recursively structured objects is a basic problem in many contexts.

The problem addressed by these design patterns occurs again and again when developing object-oriented systems.

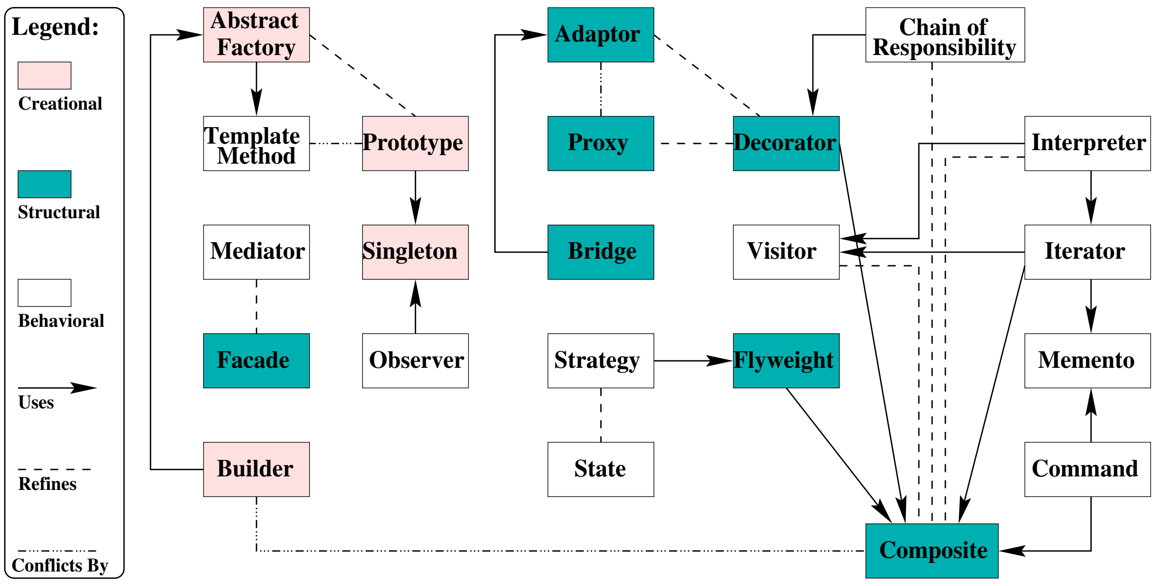

**Figure 2. Classified Structure of Design Pattern Catalogue.**

The design patterns are thus proven to be very general. When building a system, one would often look upon them more as basic design techniques than as patterns. The intentions of these design patterns are very general and applicable to a broad range of problems occurring in the design of object-oriented systems.

| Pattern | Purpose |
|---|---|
| Adaptor | Adopting a protocol of one class to the protocol of another class. |
| Composite | Single and multiple, recursively composed objects can be accessed by the same protocol. |
| Decorator | attaching additional properties to objects. |
| Facade | Encapsulating s subsystem. |
| Mediator | Managing collaboration between objects. |
| Memento | Encapsulating a snapshot of the internal state of an object. |
| Proxy | Controlling access to an object. |
| Singleton | Providing unique access to services or variables. |
| Template | Objectifying behavior. |

**Table 3. Primitive Design Patterns.**

## 7.2 Complex Design Patterns

This layer comprises of design patterns which are used for more specific problems in the design of software. These design patterns are not used in the design patterns from the *primitive* layer, but in patterns from the same layer. Design patterns in this layer are the most specific and they can often be assigned to one or more application domains.

Builder, Prototype and Abstract Factory address problems with the creation of objects, Iterator traverses object structures, Command objectifies an operation and so on.

The proposed arrangement of design patterns into two layers is one of the possible separation of concerns for design patterns. As more design patterns are described or new application area are considered, the proposed scheme should be revised to meet new requirements. For Example, when considering the restructuring and re-engineering of distributed systems the proposed classification scheme is not fully adequate.

Currently, such a classification allows for the understanding of the overall structure of the catalogue, and for relating new design patterns to existing ones. This arrangement also groups design patterns according to their typical combinations and plays an important role as typical combinations can be used as building blocks in software design. At the moment, it helps us to grasp and to understand the overall structure of GoF catalogue, and to relate new design pattern to existing one. It is also an aid for traversing or learning design patterns, as the user can choose between a bottom-up or a top-down traversal.

The *jurisdiction* and *characterization* criteria [13] are also orthogonal with our proposed criteria, and result in several clusters of design patterns with a similar intent. We believe that this arrangement can help for the retrieval and selection of an appropriate design pattern for a specific problem at hand.

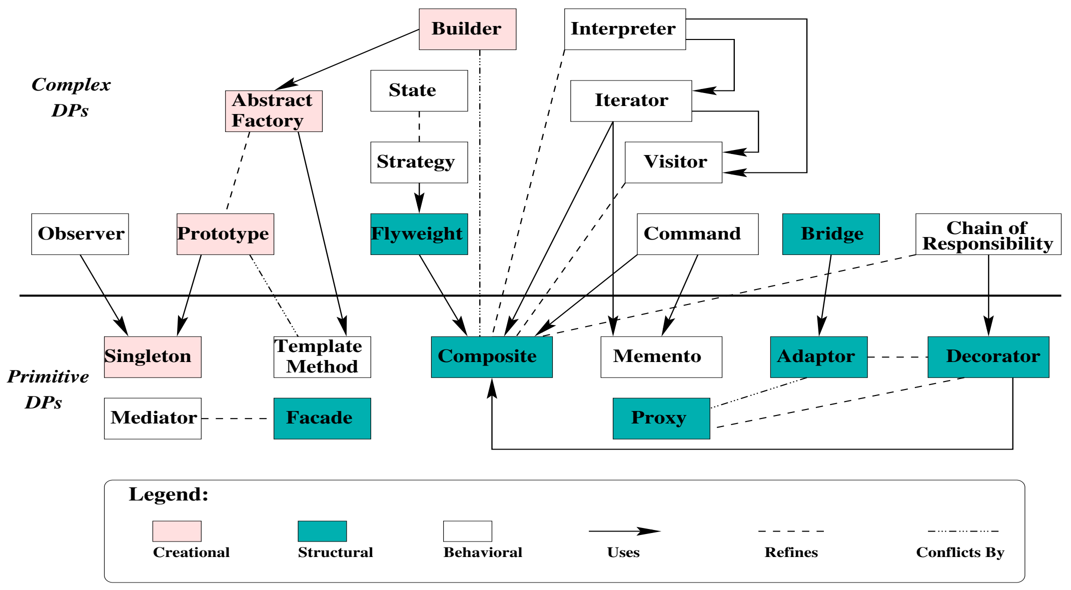

**Figure 3. Arrangement of Design Pattern Catalogue in Layers.**

## 8 Usage Application of the Layered Catalogue

In this section, we discuss the usage of the proposed layered catalogue towards the design and development of a quality and requirements-driven software re-engineering framework. The readers can also refer to [27] for more details on the framework and its associated process. Maintainability and performance are the two main non-functional requirement goals that we have considered for developing the object-oriented re-engineering framework based on our proposed layered catalogue. The reason for selecting these two non-functional requirements were described in Section 2.

We have applied this layered catalogue in the context of re-engineering on a medium-size software system, namely the *WELTAB Election Tabulation System* [33] that supports the collection, reporting, and certification of election results by city and country clerks' offices in USA. It was originally written in an extended version of Fortran on IBM and Amdahl mainframes under the University of Michigan's MTS operating systems. Later, WELTAB was converted to $C$ and run on PCs under MS/DOS (non-GUI, pre-Windows). The Object-Orientation Migration Tool [21] has been applied to WELTABIII in order to migrate the $C$ source code to new object-oriented $C++$ code. The object model for this system is depicted in Figure 4. Our experiments were carried on a SUN Ultra 10 (440MHZ, 256M memory,512 swap disk) in a single user mode. We use Rigi [19, 20] for extracting facts from the source code in order to provide a high-level view of systems. We also use Together/C++ UML Editor [29] to provide an interface to the source code generated by the Object-Orientation Migration Tool [21]. For collecting software metrics, we use Datrix Tool [9].

First for the restructure transformations, we have considered a *Primitive Structural* design pattern namely, *Composite* which is the most popular ones as it is *used by* four other design patterns and is *refined by* three other design patterns as shown in Figure 3. It means that we have started from *Structural Patterns* because they are concerned with how classes and objects are composed to form larger structures.

The *Composite* pattern describes how to build a class hierarchy that is made up of different kinds of objects. For example, in the WELTABIII system, there are two classes, namely "RECORD" which produces base tables and "REPORT" which prints the tables. It shows the necessity of having an abstract class that makes up of these different kind of objects. The key point is to have a *Composite* pattern, namely "DateGen" that represents both primitives and their containers. We compose two objects into tree structures to represent part-whole hierarchy. This lets clients namely, "ReportGen" and "TableGen" treat individual objects and compositions of objects uniformly. This *Composite* pattern is found to improve maintainability because it allows for component sharing [27]. Similarly, this pattern also allows for performance increase because it allows for explicit superclass references and simplifies component interfaces [27].

Second, we have considered the *Complex* design patterns that can be built on top of the *Composite* design pattern. As the *Behavioral Patterns* are concerned with the algorithms and assignment of the responsibilities between objects, we started with two of them namely, *Iterator* and *Visitor*. The

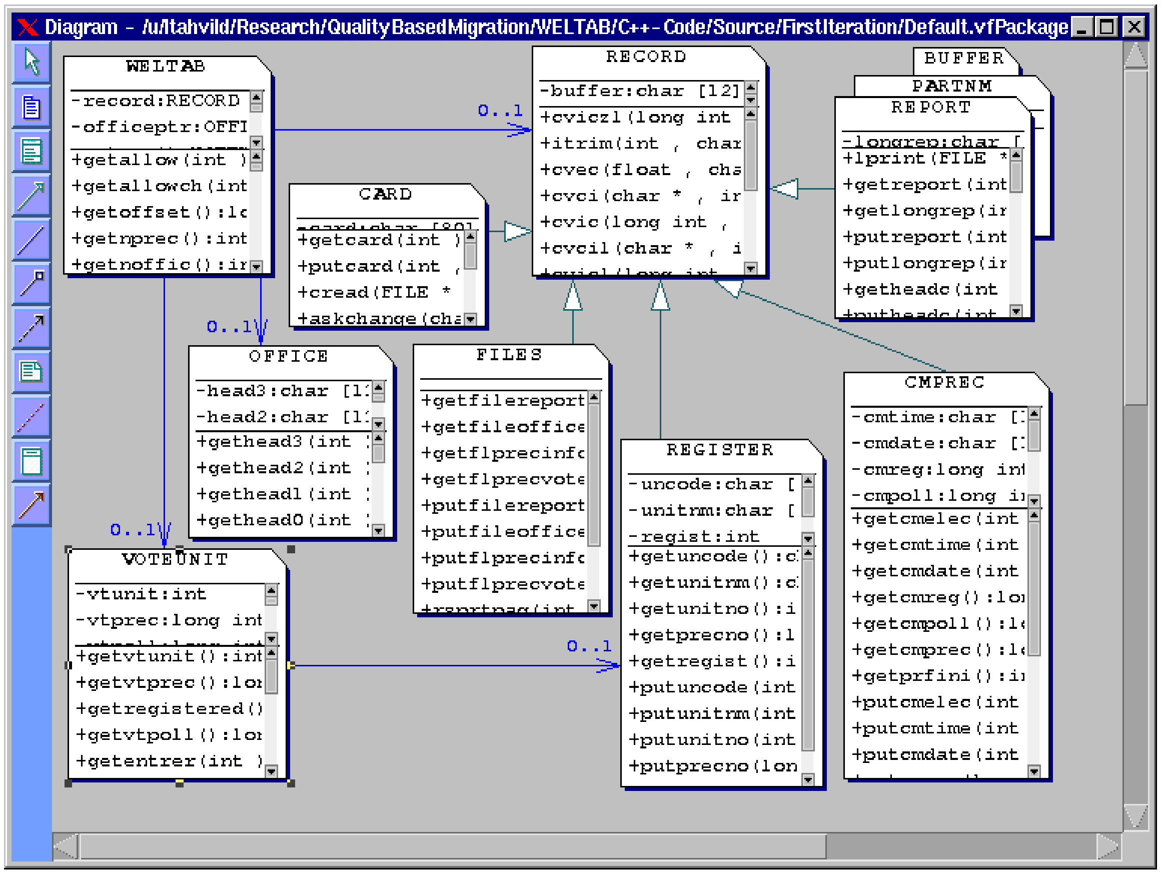

**Figure 4. Object Model of WELTABIII.**

*Iterator* pattern allows to access an aggregate object's contents without exposing its internal representation. The pattern also supports multiple traversals of aggregate objects. Providing a uniform interface for traversing different aggregate structures is another reason to use this pattern that supports polymorphic iteration. Our experimental results [27] indicate that by applying the *Iterator* pattern, we obtain an increase in maintainability. However, the *Iterator* pattern reduces the performance because of more than one traversal can be pending on an aggregate object. One proper example of this pattern can be "Info" class. As illustrated in Figure 4, there are different basic information such as units, offices, precincts, and candidates that have been spread correspondingly over the classes, namely "VOTEUNIT", "OFFICE", "CMPREC", and "RECORD". Structuring this part of the system into subsystems helps to reduce its complexity. For achieving this goal, we introduced an iterator object that provides a single simplified interface, called "Info" as shown in Figure 5, to the more general facilities of the basic information. This new class defines an interface for accessing and traversing elements. This results to a system that is more maintainable because of the simplification of the aggregate interface.

On the other hand, the *Visitor* pattern helps making the migrant system more maintainable because a new operation over an object structure can be modified simply only by adding a new visitor class. An explanation is that as a restructuring operation, *Visitors* help in applying operations to objects that don't have a common parent class. This has as a result the reduction of the tree traversal time and therefore it may be considered as a heuristic that improves performance. For example, consider the "REPORT" class in Figure 4 supports multiple reports. Different reports have different appearances and headers for printing. To be portable across reports, an application should not be hard-coded for a particular report. We can solve this problem by defining an abstract "ReportGen" class that declares an interface for creating each kind of reports as shown in Figure 5. This class acts as a *Visitor* pattern. Our results indicate that this pattern can be considered as a heuristics that improves both maintenance and performance [27].

In this context, the proposed classification scheme helped us to select those target design patterns that may have the maximal measurable impact with respect to per-

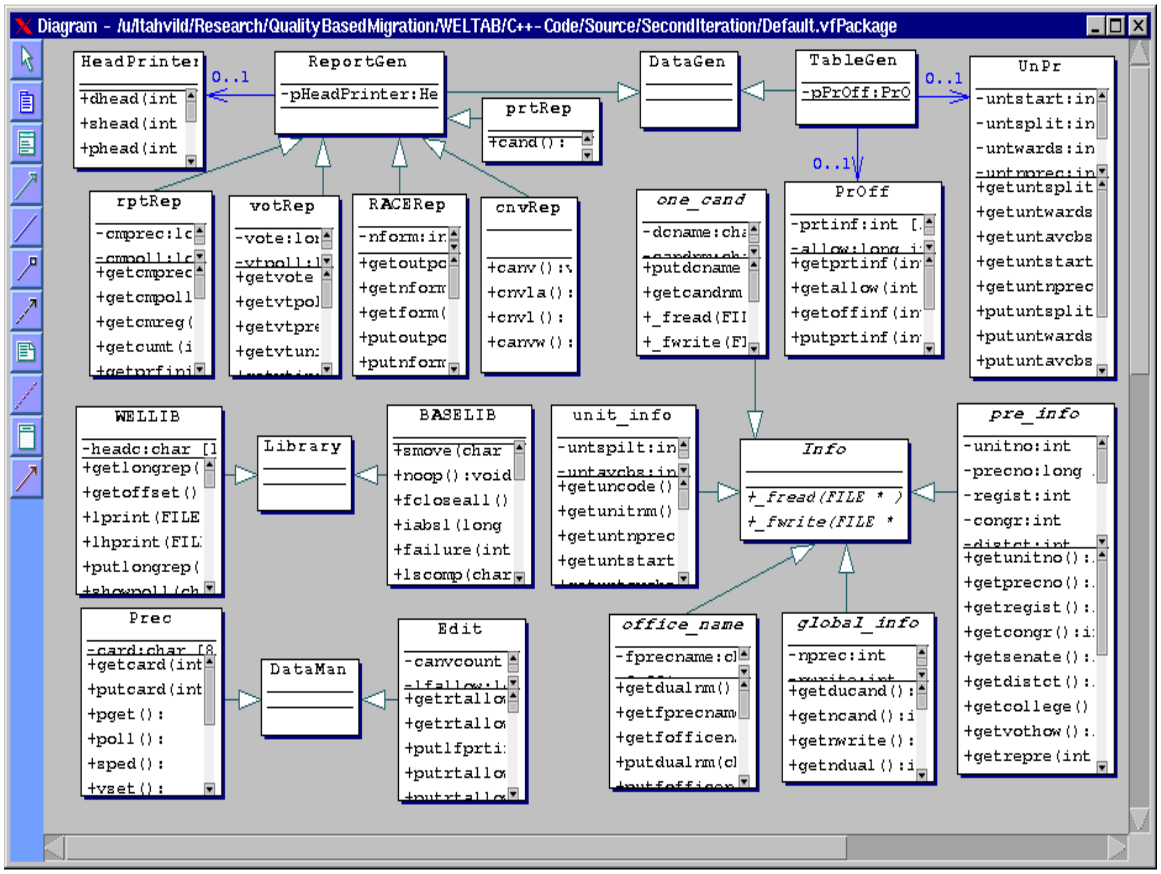

Figure 5. Object Model of WELTABIII with Design Patterns.

formance and maintainability enhancements on the migrant code. These enhancements can be measured in term of software maintainability index metrics and profiling measurements [27].

## 9 Conclusion

We have presented a classification of the relationships between design patterns which can lead to a new categorization of the GoF design patterns into different layers. Our classification scheme is based on three *primary* relationships between patterns such as *use*, *refine*, and *conflict* and three *secondary* relationships such as *similar*, *combine*, and *require* which can be expressed in terms of the *primary* ones.

The proposed classification can assist software engineers with : i) understanding better the complex relationships between design patterns, ii) organizing existing design patterns as well as categorizing and describing new design patterns, iii) building tools which support the application of design patterns during restructuring.

The next steps of our work focus on the formalization of the impact different transformations have on the migrant system with respect to maintainability and performance enhancements, and the formal description of the source code transformations required for the migrant system to conform with the proposed design pattern classification scheme.

## About the Authors

Ladan Tahvildari is a Ph.D. candidate at the Department of Electrical and Computer Engineering, University of Waterloo. Her research interests include software evolution, program understanding, quality based re-engineering. She may be contacted at ltahvild@swen.uwaterloo.ca.

Kostas Kontogiannis is an Associate Professor at the Department of Electrical and Computer Engineering, University of Waterloo. His research interests include software reengineering, software migration, software reuse and knowledge based software engineering. He may be contacted at kostas@swen.uwaterloo.ca.

## References

[1] C. Alexander. *A Pattern Language*. Oxford University Press, 1977.

[2] K. Beck. Patterns and software development. *Dr. Dobbs Journal*, 19(2):18–23, 1993.

[3] G. Booch. Patterns. *Object Magazine*, 3(2), 1993.

[4] K. Brown. Design reverse engineering and automated design patterns detection in smalltalk. Master's thesis, Department of Computer Engineering, North Carolina State University, 1996.

[5] F. Buschmann et al. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley and Sons, 1999.

[6] P. Coad. Object-oriented patterns. *Communications of ACM (CACM)*, 35(9):153–159, September 1993.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[8] J. O. Coplien and D. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1994.

[9] Datrix metric reference manual, version 4.1. Bell Canada, 2000. Also available at http://www.iro.umontreal.ca/labs/gelo/datrix.

[10] A. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Proceedings of the IEEE Automated Software Engineering (ASE)*, pages 143–152, November 1997.

[11] G. Florijn, M. Meijers, and P. Winsen. Tools support in design patterns. In *Proceedings of the ACM European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 427–495, June 1997.

[12] M. Fowler, editor. *Analysis Patterns*. Addison-Wesley, 1997.

[13] E. Gamma, R. Helm, R. Jahnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] R. E. Johnson and V. F. Russo. Reusing object-oriented designs. Technical report uiucdcs 91-1696, University of Illinois, May 1991.

[15] R. Keller, R. Schaure, S. Robitaille, and P. Page. Pattern-based reverse engineering of design components. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 226–235, Los Angeles, USA, 1999.

[16] D. H. Lorenz. Tiling design patterns - a case study using the interpreter pattern. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 206–217, 1997.

[17] R. Martin, D. Riehle, and B. F., editors. *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.

[18] G. Meszaros and J. Doble. A pattern language for pattern writing. In R. Martin, D. Riehle, and B. F., editors, *Pattern Languages of Program Design*, volume 3, pages 529–574. Addison-Wesley, 1998.

[19] H. Muller. Rigi as a reverse engineering tool. Technical Report DCS-160-IR, University of Victoria, Victoria, BC, Canada, 1991.

[20] H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[21] P. Patil. Migration of procedural systems to object-oriented architectures. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, 1999.

[22] W. Pree. Meta patterns : A means for capturing the essentials of reusable object-oriented design. In *Proceedings of the ACM European Conference on Object-Oriented Programming (ECOOP)*, volume 0821, pages 150–162, 1994.

[23] D. Riehle. Composite design patterns. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 218–228, 1997.

[24] M. Shaw. Heterogeneous design idioms for software architecture. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 158–165, Como, Italy, October 1991.

[25] L. Tahvildari, R. Gregory, and K. Kontogiannis. An approach for measuring software evolution using source code features. In *Proceedings of the IEEE Asia-Pacific Software Engineering (APSEC)*, pages 10–17, Takamatsu, Japan, December 1999.

[26] L. Tahvildari and K. Kontogiannis. A workbench for quality based software re-engineering to object-oriented platforms. In *Proceedings of the ACM International Conference in Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) - Doctoral Symposium*, pages 157–158, Minneapolis, Minnesota, USA, October 2000.

[27] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Requirements-driven software re-engineering. In *Proceedings of the IEEE $8^{th}$ International Working Conference on Reverse Engineering (WCRE)*, pages 71–80, Stuttgart, Germany, October 2001.

[28] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Weaving between non-functional requirements and object-oriented re-engineering. In *Proceedings of the 11th Annual Canadian Conference on Intelligent Systems*, pages 64–65, Ottawa, ON, Canada, June 2001.

[29] Together/c++ uml editor. Also available at http://www.togethersoft.com/.

[30] P. Tollena and G. Antoniol. Object oriented design patterns inference. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 230–238, September 1999.

[31] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[32] A. Weinand, E. Gamma, and R. E. Johnson. $et++$: An object-oriented application framework in $c++$. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 46–57, 1988.

[33] Weltab election tabulation system. Also available at http://pathbridge.net/reproject/cfp2.htm.

[34] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of ACM (CACM)*, 33(9):105–123, September 1990.