Technical Report, University of Maryland, Computer Science Department, College Park, MD, 20742 USA.

AN INDUCTIVE METHOD FOR DISCOVERING DESIGN PATTERNS FROM OBJECT-ORIENTED SOFTWARE SYSTEMS

Forrest Shull, Walcélio L. Melo, and Victor R. Basili

Computer Science Department/
Institute for Advanced Computer Studies
University of Maryland, College Park, MD, 20742
{fshull, melo, basili}@cs.umd.edu

Abstract

Object-Oriented Design Patterns (OODPs) have been proposed as a technique to encapsulate design experience and aid in design reuse. However, so far, there is very little empirical evidence about what we can expect from this emergent technology. For instance, to date little research has focused on the development of techniques for discovering workable patterns that can be captured, formalized, packaged, and quantitatively evaluated. Our work is a step in this direction. In this paper we present an inductive method aimed at helping us discover OODPs in existing OO software systems. It encompasses a set of procedures rigorously defined in order to be repeatable and usable by practitioners who are not acquainted with reverse architecting processes. Guidelines are provided and a case study is shown that demonstrates the usefulness of the approach.

Key-words: Object-Oriented Design Patterns; Reverse Engineering; Empirical Software Engineering; C++ language.

1. INTRODUCTION

Software developers, working over time in a particular organizational setting, implicitly gain an understanding of what works and what does not work in their particular environment as they develop personal techniques for producing successful object-oriented designs. A part of these techniques lies in knowing what arrangements and interrelations of classes are useful; another part lies in knowing when and where to apply these arrangements for them to be effective. If taken together and packaged effectively, these two parts become an Object-Oriented Design Pattern (OODP) - a practical solution to a typical subproblem, along with a set of guidelines for when to apply it. From these collections of individual techniques, we can then extract techniques which are useful within the organization as a whole. Organizations that preserve a collection of successful designs have thus already made the first step toward creating an "institutional memory" of useful design techniques. Such collections can be a rich source of OODPs, as they implicitly capture designer experience. The challenge is in transforming these *ad hoc* collections of design solutions into custom knowledge bases of Object-Oriented Design Patterns which can be reused in an organization-wide context.

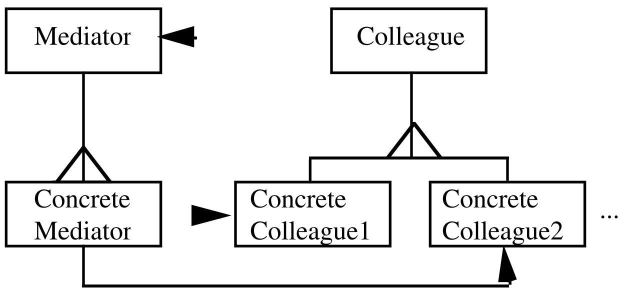
To say more specifically just what a design pattern is, we borrow from [Gamma, 1995], who define their set of design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Patterns are a means of succinctly capturing "simple and elegant solutions to specific problems in object-oriented software design." These solutions have evolved over time, and so reflect the effort spent in their redesign and recoding by greater flexibility and reuse potential. Patterns have four essential elements:

- The **pattern name** is a short handle used to refer to the pattern and the problem it solves. Naming patterns lets designers work at a higher level of abstraction.
- The **problem** description gives the problem and its context; it describes when the pattern is applicable.
- The **solution** provides an abstract description of the elements that make up the design and their relationships.
- The **consequences** are the results of applying the pattern, and reflect the fact that even very successful designs often imply a trade-off of some sort.

As an example of an OODP, consider the Mediator pattern which is illustrated in Figure 1. This pattern is part of the set of general-purpose patterns proposed by [Gamma, 1995]. It defines a problem in which its application may be useful (it should be used when an object "refers to and communicates with many other objects" or when "a behavior that's distributed between several classes should be customizable without a lot of subclassing") and specifies as a solution a set of classes that work together in specified ways (the class that corresponds to the concrete instantiation of the Mediator class maintains links to "colleague" classes; the colleagues inherit a link back to the mediator class). As a result of its use, certain consequences are observed: control is centralized, subclassing is limited, object protocols are simplified, etc. We will look at the Mediator class in somewhat more detail in Section 5.

Figure 1: The Mediator Pattern

Mediator Pattern from *Design Patterns*



Our work is a step toward the creation of custom knowledge bases of patterns. We developed an inductive method, which we call BACKDOOR (Backwards Architecting Concerned with Knowledge Discovery of OO Relationships) aimed at helping us reverse architect OODPs from existing OO software systems and package them into reusable design solutions. Such a method will allow us to:

1) capture the experience of designers, by noting patterns which consistently appear in their work;

2) package useful patterns so that designers can tailor them to their own designs;

3) collect metrics on the OODPs in order to validate the usefulness of these patterns in terms of defect-proneness, rework effort, reusability, and other attributes;

4) refine the pattern discovery process by using these patterns to hone the search mechanism.

The method encompasses a procedure defined in order to be repeatable and usable by people who are not acquainted with reverse architecting processes.

The development of a full pattern-discovery system is too huge a task to be implemented all in one step. In this paper, we take an initial look at validating the first two uses of our BACKDOOR method: pattern discovery and packaging. Features that characterize "good" patterns are still open to debate. Therefore, we choose a reference set of patterns of proven usefulness against which we compare the patterns found by BACKDOOR. In this way, we hope to gain some insight into good techniques for finding and packaging patterns. Once we have some confidence that our method is well-defined in these two areas, we will proceed to future work on adding a metrics suite. Recent work has already shown the usefulness of using metrics to evaluate OO designs [Basili, 1995]; we propose to extend this work to the characterization of patterns as well. In particular, coupling metrics such as those discussed in [Briand, 1994] show promise for application to design patterns. Another extension to the BACKDOOR process will involve the addition of feedback loops in order to refine the discovery mechanism based on previous instances of patterns found.

2. CONTEXT AND SCOPE OF THE STUDY

In this section, we outline what the completed knowledge base will look like. We emphasize that our work is only an initial step in this direction.

We feel that pattern discovery, formalization, and packaging must be done as part of the organization's Experience Factory [Basili, 1989], that is, as a separate event apart from the rest of the software development life cycle, not as an *ad hoc* process spread throughout the normal life cycle activities. Therefore we clearly differentiate the Development Organization (DO) from the Pattern Knowledge Base (PKB), though their roles are complementary.

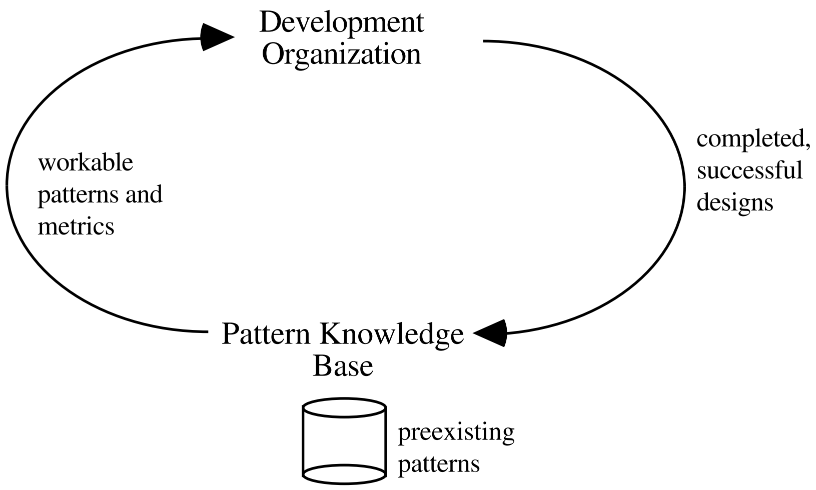


Figure 2: A pattern-gathering cycle.

The goal of the PKB is to create for the development organization a custom library of patterns that have been empirically validated to work in the given environment. This is achieved in the long run by many iterations through a feedback cycle between the DO and PKB (see figure 2). The DO provides a set of its completed designs. The PKB makes use of these designs in two ways: First, it looks for new instances of already recorded patterns; if it finds any, then it updates the suite of metrics associated with the pattern. These metrics refer to qualities of the patterns that the organization finds of interest, for instance: error-proneness, reusability, repairability. Secondly, the PKB uses its own collection of patterns that have been found on previous iterations as a basis of comparison from which it can deduce characteristic design structures in the set of designs provided that should be further studied as potential patterns. It returns to the DO a set of patterns with associated quality attributes. At the start, the PKB will only be able to identify patterns based on purely structural features (though we hope to make the method more sophisticated over time); therefore, the developers within the DO must review the patterns detected in terms of semantic content. If the designers decide that the patterns do encapsulate a useful design technique, then the pattern enters the pattern library and is available for use in future designs.

Our difficulty in developing a workable PKB has been in finding a technique for the PKB to use on the first iteration of the cycle, i.e. when the knowledge base of preexisting patterns is empty. In this paper, we examine that first iteration more closely and attempt to solve this problem by providing an initial generic reference set. For this iteration of the cycle only, we seed the PKB with patterns from a reference set that has proven its usefulness in many different environments, though perhaps not the current one. We decided to use the set of OODPs found in the book *Design Patterns* [Gamma, 1995] as our reference set. This book was an attractive choice for an initial reference set of patterns since it includes only design patterns that have been applied in multiple systems, captures the patterns in a consistent and accessible way, and concentrates on general, rather than domain-specific, patterns.

Using a suite of student projects developed at the University of Maryland, we attempt to validate the BACKDOOR method for reverse architecting OODPs by examining how it would perform on this first iteration of the cycle. Thus the scope of this current study is limited to the following two questions:

- Can our method discover instances of the reference patterns with which to seed the next iteration of the cycle? (Are we packaging the right things?)
- Are the pattern instances we find at a comparable level of detail to the patterns in the reference set? (Are we packaging things right?)

One consequence of limiting our focus in such a way is that any pattern we discover which we cannot relate back to the reference set is unusable to us. We do not investigate here whether such patterns are spurious matches produced by our method, or actually usable patterns which do not appear in the reference set. We will defer this question to future work when we are more concerned with successive iterations of the pattern discovery cycle.

3. A TECHNIQUE FOR ASSESSING POTENTIAL PATTERNS

To answer the questions of our study, we find it necessary to have some point of reference against which we can compare the potential patterns we discover. We therefore divide patterns into three main components, and assess the conformance of each of our potential patterns to a pattern in the reference set in terms of each of the components:

Structure - By the structure of a pattern we refer to the classes, objects, and relationships out of which the pattern is built. As part of the pattern language developed in [Gamma, 1995] the structure of each pattern is specified to some level of detail. Therefore we used the structural component as a first step in identifying potential patterns by examining the architectures of the student projects for corresponding object relationships. Because we did not want to discount patterns that had undergone some level of tailoring, we did not look for complete structural matches, but used the structure instead as a general guideline. To use the terminology introduced in Section 1, by comparing the structures we are really taking a look at whether the patterns are equivalent **solutions**.

Purpose - Every pattern must specify what it was meant to be used for: what situations it might be useful in, what effects (both positive and negative) it should have on the system. This is what we refer to as the purpose of a pattern. To assess conformance between our potential patterns and the reference patterns in this regard, we created for each reference pattern a checklist consisting of the major points found in the pattern description in [Gamma, 1995]. (Example checklists can be found in Section 5.) This component allows us to answer the question of whether designers were using implicit patterns by demonstrating whether or not developers were trying to solve the same problems the reference patterns were meant to address. Here, we are assessing whether the potential pattern is addressing the same **problem** as one in the reference set, as well as whether the **consequences** of the patterns are the same.

Implementation - A secondary focus of our study was to see if our discovered patterns were specified at a useful level of sophistication - that is, could the patterns we had discovered be used to create a knowledge base of future use to developers? Or would they need to be extensively generalized before they would make good patterns? This information is specified as part of the implementation component of a pattern. The structure of a pattern gives the broad outline of what the pattern looks like; the implementation specifies in a more detailed way how the pattern accomplishes its associated purpose. The structure specifies only which objects communicate with one another, while the implementation describes the messages they send to one another. As we did for the purpose component, we created a checklist for the implementation by summarizing the main points in [Gamma, 1995] about the interrelationships between the component classes. This amounts to a more detailed investigation of whether the potential pattern matches the **solution** aspect of a pattern in the reference set.

The checklist was applied in the following way: the BACKDOOR method was used to examine the architectures of the student designs. When a group of classes was found that appeared to interrelate in a way that was similar to a pattern structure in the reference set, it was flagged as a potential pattern. Then we rated the correspondence of each of these to its closest reference pattern on a 4-point scale, where a rating of 4 represents a very close match. In figure 3, it will be noted that since we emphasize purpose over implementation we weight that axis more heavily:

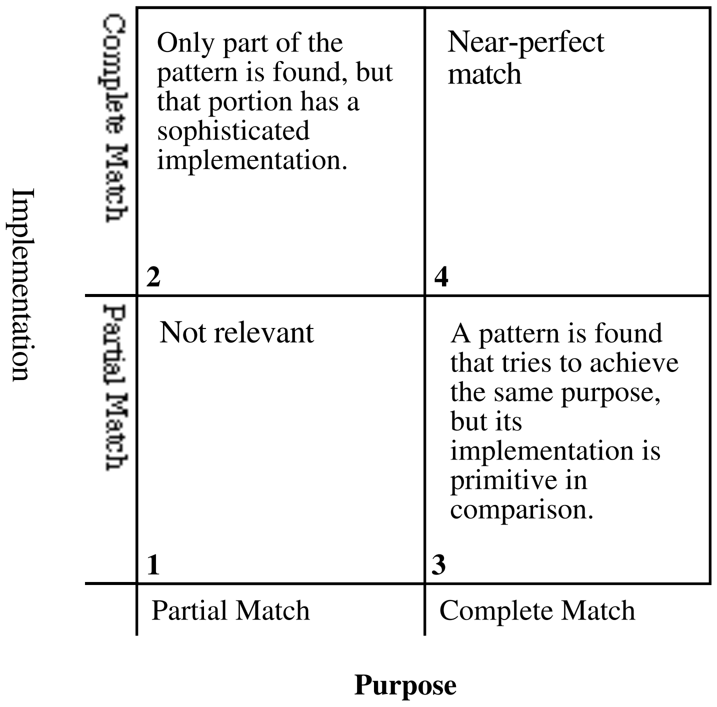


Figure 3: Ranking the correspondence of discovered patterns.

4. BACKDOOR: AN INDUCTIVE METHOD FOR REVERSE ARCHITECTING OODPS

In this section we present BACKDOOR, our inductive method for reverse architecting design patterns from Object-Oriented software systems. The main output of this process will be a knowledge base that describes patterns used to date by the organization. Once potential patterns have been identified, they can be reviewed by developers to see if there is any meaning to the set of classes identified.

To date, little research has focused on the development of techniques for creating such knowledge bases. There is a lack of guidelines to help software developers in the process of searching for, formalizing and packaging workable patterns. In a general discussion, Peter Coad [Coad, 1992] emphasized the salient characteristics such a process should have:

- The interrelations and interactions among classes and objects form the basis of OO patterns and so should be the focus of study.
- The most useful patterns are those that recur in a large number of situations; therefore pattern discovery requires careful observation of many OO models and a trial-and-error approach.

We feel that formalizing techniques for pattern discovery and packaging may prove helpful to practitioners attempting to cull such knowledge from their own past history.

John Vlissides [Vlis, 1995] also touches briefly on the problem of finding patterns in OO designs. We adopt his term of "reverse architecting" to refer to the approach of analyzing many software systems in an effort to find recurring patterns and rationales. (He differentiates this from "reverse engineering," analyzing a particular system for the purpose of recovering its design.) He also emphasizes the important fact that a successful OODP is consistent and generally applicable; a particular solution is not enough, but the general principle behind the solution is required. The challenge lies in recognizing general patterns from specific instances that may be superficially quite different; the trick is in recognizing common features between solutions that are addressing the same issues even when they have been heavily tailored to the particular environment.

BACKDOOR consists of six steps. Although the steps are defined sequentially, they are really iterated within and across steps. The process as a whole could be greatly assisted by tool support. Maintenance tools exist which automatically generate cross-references (which would help in identifying associations between classes) and link the documentation to the code (which would help in identifying semantic content), among others. We developed our own guidelines for reverse engineering assuming no tool support, however, and include them here. This choice was motivated by a number of reasons: the systems we analyzed were not very large and could be done by hand (though the process is time-consuming), we wanted to gain an in-depth knowledge of the systems as part of the process, and we wanted to develop useful techniques for situations in which tool support was not available.

1) Review the problem specification and design documents.

While we would like to formalize the technique as much as possible (and in the future develop tool support) the fact remains that semantic content is a large component of pattern definition. For this reason any attempt to approach the code without getting as much semantic information as possible beforehand is bound to be unsuccessful. A study of the problem specification is necessary to identify problem constraints and issues as well as to get some idea of the functionality provided; while it remains rare in practice to encounter a design document which is an exact match to the system it purports to describe, such documents are still very helpful in identifying relationships between subsystems.

2) Develop a preliminary model of the system from the class declarations.

The class declarations represent the clearest and most concise descriptions of the objects in the system and so represent a natural place to start reverse architecting. As we discover how the classes interact, however, we will need a notation for representing this knowledge. We have found it helpful to use OMT object notation [Rumbaugh, 1991] since it provides a notation for describing specialized interactions between classes as well as the structures of the classes themselves.

As mentioned before, we are mostly concerned with the relationships between classes. Some relationships, such as inheritance, are not difficult to detect. Communication between classes can be harder to characterize because of the number of variations possible; at this step we content ourselves with identifying associations between classes without worrying about their specific characteristics. In C++, these associations are often implemented as pointers: one class keeps a reference to another and uses the link to send messages to the target class. Class attributes and the return values of class methods are thus good places to look for associations between classes.

By focusing first on the class declarations, we are able to sketch relatively quickly the broad outlines of the system. In general, large subsystems can be readily identified based on the amount of coupling between object classes; a high rate of coupling indicates a high rate of message passing, which often indicates a set of classes working closely together.

3) Refine object notation from class implementation.

It is uncommon, however, for the class declarations to capture all of the details of class associations. Therefore we take a more detailed look at the code implementing the classes. Any time one object communicates with another - by sending a message to a class attribute, to a parameter passed to a class method, to a global variable which happens to be of some class type, or to a friend class - we make sure to capture the communication in our model diagram.

At this stage, we also spend more time characterizing the associations we find, and using the appropriate OMT notational devices to represent this knowledge. Classifying the multiplicity of links (as one-to-many, one-to-one, or many-to-many) is especially important as it may point out hidden assumptions in the system. Some associations appear at first glance to be one-to-one until a closer look is taken at the implementation. For example, customer databases often keep track of phone numbers. Whether or not the system allows customers to have more than one phone number can be detected by examining the kind of data structure used to represent the field [Rumbaugh, 1991].

Discerning whether an association represents an acquaintanceship or aggregation can be a more difficult problem. Acquaintanceship signifies that one class knows of the existence of another and can send messages to the class. Aggregation is a special case of acquaintanceship in which one object entity is actually composed of other objects with which it can communicate. From a reverse engineering point of view, the difficulty arises from the fact that though acquaintance and aggregation links are often implemented the same way, they effect the model of the system very differently. For example, some component objects cannot exist apart from the aggregate object to which they belong - does it make sense to refer to a "binding" existing independently of a "book"? At the same time, other aggregations are not so constrained - a monitor can exist independently of a computer system, although it may be modeled as a system component. Such considerations are important to understanding the semantics of a system [Rumbaugh, 1991].

At the very least, we find it helpful to record whether an association is "read" or "read/write". One object communicating with another to perform a lookup of information is a very different interaction than to modify another object in some way. Such a characterization requires a careful examination of the implementation of the method in order to see if it modifies object attributes or simply returns data, but captures much information about the purpose of the association.

4) Using the refined model of the system architecture, identify potential candidates for patterns based on inheritance and communication links between classes.

At this step we focus on interesting relationships within the architectures, looking for recurring structural similarities. A few indicators that we have found to be useful:

- Classes which are at the receiving end of communication links from many other classes may play some role in mediating the interactions between these other classes.
- A class at the sending end of links with many other classes may act as an interface to those classes.
- Classes with parallel inheritance (that is, each subclass of one class is linked to a subclass of another class) are likely to be working together closely.
- An object which links two clusters of classes with high coupling may be a sophisticated communication link between two subsystems.

In order to initially create our PKB, we search for structures corresponding to the OODPs in our reference set [Gamma, 1995]. As part of the pattern language developed there, the pattern structure is specified. Our first pass at finding patterns is to search the project architectures for classes and relationships that match the structures in the reference set.

5) Analyze pattern candidates detected in step 4.

This is the most difficult portion of the mining process: What characteristics indicate a useful design pattern?

We have yet to be able to formulate generally applicable guidelines in this area; this is the most human-intensive portion of the process, and pattern discovery still relies heavily on individual skill. Part of the answer comes from a knowledge of the problem domain: what are recurring subproblems that the organization must deal with? The focus here is on semantics rather than structure, making the process difficult.

For this case study, we are comparing the potential patterns identified in step 4 against a set of reference patterns. We first distilled the description of each reference pattern into a checklist of its most important features. In this way, we could get some measure of how closely the patterns we thought we had discovered actually matched in terms of structure, purpose, and implementation (we describe how we assess conformance along these dimensions in section 2.2). Our search strategy is a successive narrowing of focus (illustrated in figure 4) in which we filter sets of classes from consideration based first on structure and then on semantics.

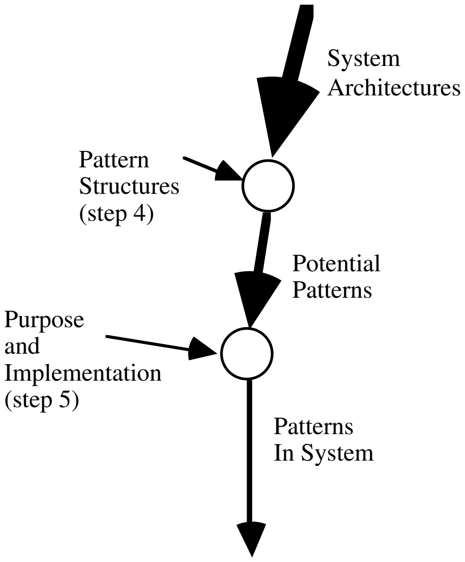


Figure 4: Focusing on patterns

We then classified the potential patterns based on their conformance to reference OODPs. If the correspondence is low, our identification of a section of the design as a pattern was presumably spurious. Or perhaps we have identified a pattern not identified in the reference set. At any rate, it is not useful in our current focus of study. A high correspondence, on the other hand, presents evidence that the classes we identified as a pattern had been found generally useful in other environments. In this way, we received feedback as to whether or not the structural features we had focused on were good indicators of patterns. Because our classification scheme is important to our identification of patterns, we discuss it further in the next section.

6) Interview designers and implementors to clarify design issues.

Of course, just because the architecture of the system and the system functionality can both be recovered does not mean that the one can be mapped perfectly to the other. Interviews with the responsible designers and coders can be a crucial step in understanding. Such interviews can sometimes be the only way to gain an understanding of what context issues influenced a particular design, or how various subsystems interrelate.

It is expected that the information gained in interviews may lead to a return to steps 4 or 5 for a more detailed iteration.

5. A CASE STUDY

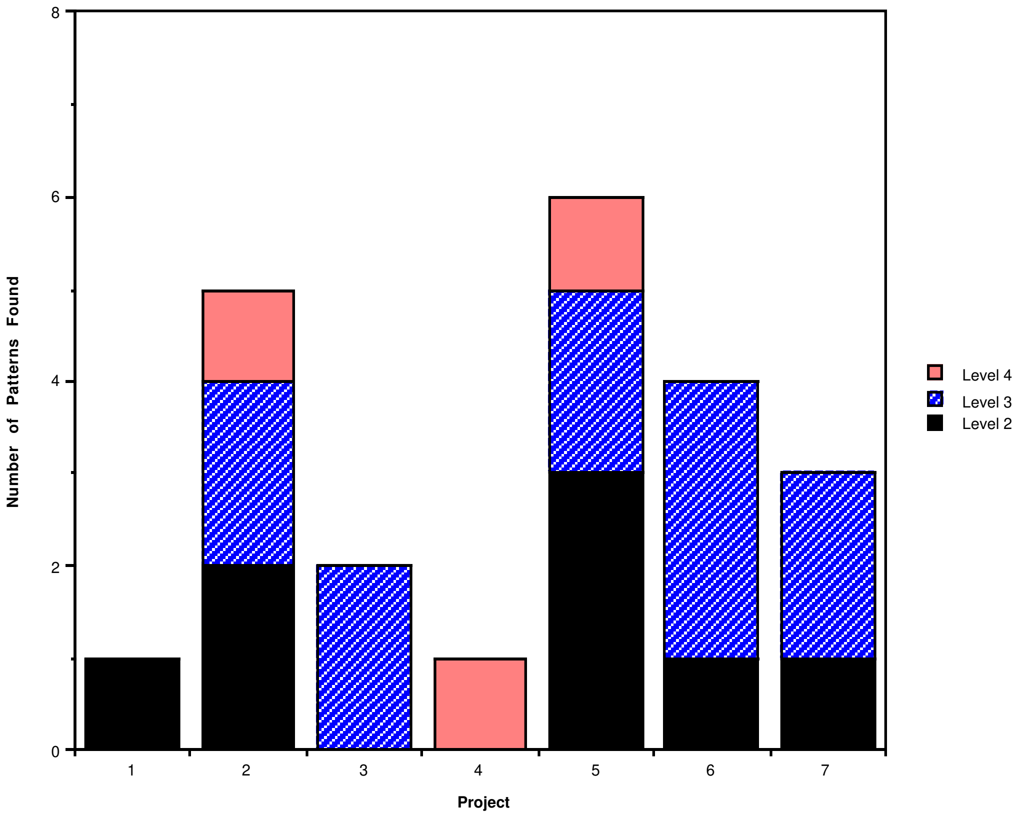
To validate our BACKDOOR method, we used a suite of seven Object-Oriented student projects developed at the University of Maryland. Because the student projects are all implementations of the same basic system, this provides us with seven distinct solutions to the same problem. We apply our method to the project architectures, in order to answer the following questions:

- Did our technique turn up any patterns similar to those found in the reference set? That is, can we use our method to find patterns like those people are already finding useful?
- How close is the correspondence - are the patterns found packaged at a level of sophistication that actual developers are likely to find helpful?

The set of projects we examined were seven small management information systems supporting the rental/return process of a hypothetical video rental business; as such they were required to maintain databases for customers, videos, and transactions. The systems were implemented in C++ by teams of students in a graduate level course offered by the Computer Science department at the University of Maryland. OO design was taught as part of the course material. The development environment and technology used are representative of what is currently used in industry and academia.

Figure 5 characterizes the seven projects used in our case study with respect to the patterns we identified. Some projects had only one or two patterns in their design, while others had several, at varying degrees of correspondence to the reference set. Of the potential patterns identified, it turned out that at least one for each project matched a pattern in the reference set. The project which yielded the most reference patterns was project 5, with 6 patterns detected. Exactly half of the 22 patterns we detected were classified at a correspondence level of 3, signifying a close match to a problem addressed by a pattern in the reference set, though not an equally sophisticated solution. Only 3 matches were rated as a level 4, while the remaining 8 were considered at level 2. ("Patterns" at level 1 are discarded as being irrelevant - we cannot define them as real patterns, at least in terms of our reference set. Whether they are of use in themselves awaits a future analysis.)

Figure 5:
Characterization of Projects



As a more concrete example of our results, we present here an example of a pattern discovered in our student projects for each of the levels 2 through 4 in our classification scheme.

Level 4

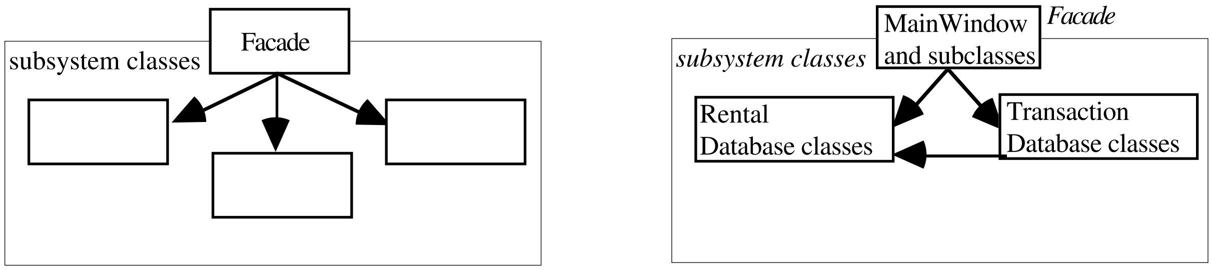
Patterns at level 4 are considered to be a near-perfect match to one of the patterns in the proposed set. As such, they provide some evidence that the pattern we discovered does in fact capture a relatively sophisticated solution as it appears in practice.

In our set of projects, however, only 3 contained a level-4 pattern. All three of these were implementations of the same pattern: the Facade. The Facade pattern provides a single, high-level interface to subsystems that makes the system easier to use. (See figure 6 for a simplified diagram of the pattern structure.) In our system, the Facade takes the form of a relatively advanced user interface that manages all requests sent to the three database subsystems. We focus on one of the instances as an illustrative example.

Figure 6: The Facade Pattern

Facade Pattern from *Design Patterns*

Facade Pattern as Implemented in System



The example found in the student project is a good match to the reference pattern in terms of purpose. It agrees on the following points (assessed via the checklist):

- It provides a unified interface to a set of subsystems - here, the subsystems implementing the various databases for the video system.
- It minimizes communication between subsystems. In the particular student project in which it was found, there was only a limited amount of interaction for cross-referencing lookups.
- The Facade provides a default view of the subsystems, so that only clients requiring more customizability need look beyond the facade.

The example found in the student project also agrees in general implementational detail to the proposal:

- Subsystem functionality is implemented only in the subsystems, which handle work assigned by the Facade object. Subsystems have no knowledge of the Facade (that is, they keep no reference to it).
- The Facade class knows which subsystems are responsible for handling requests, and delegates requests to appropriate subsystem objects.
- Clients communicate with the subsystem by sending requests to the Facade, which forwards them to the appropriate subsystem objects.
- Clients that don't use the Facade don't have access to subsystems directly - all user requests are processed by the system interface (the Facade) and translated into appropriate requests to the database subsystems.

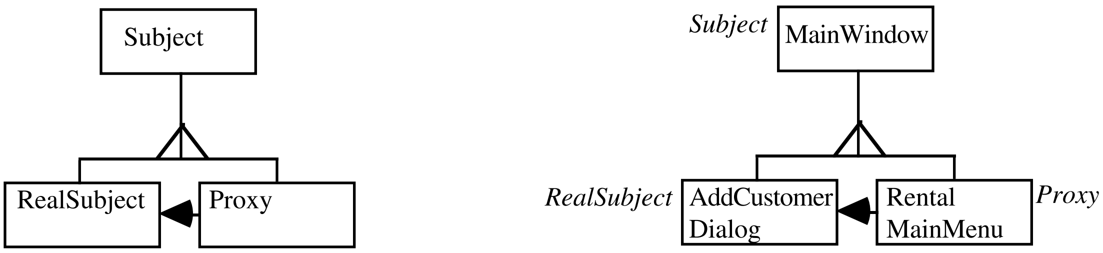
Level 3

Level-3 patterns indicate that our student subjects were attempting to solve the same problem as is addressed by one of the reference patterns, but did not manage to implement the solution in a manner as sophisticated as the reference pattern. We found 11 instances of 7 different level-3 patterns in our projects. The Proxy pattern is a good example. Its purpose is to provide a surrogate object that controls access to some other object. (See figure 7.) The instance we detected in the student project is a mechanism encapsulated within a window for the main menu. As options are selected, the window creates and initializes windows for submenus, transfers control to the new windows, and then receives control back again when the new window has finished processing. In this way, creation and access to the other windows of the program are controlled through the main menu window.

Figure 7: The Proxy Pattern

Proxy Pattern from *Design Patterns*

Proxy Pattern as Implemented in System



The pattern we discovered is a good match to the intended use of the Proxy pattern:

- An object is provided which controls access to some other object.
- The full cost of object creation and initialization is not incurred until absolutely necessary, if ever. If a submenu is never accessed in a session with the program, the object corresponding to its window is never created.
- Proxies can be used to create expensive objects on demand.

However, the Proxy as presented in *Design Patterns* exhibits a much cleaner design than the pattern which we identified.

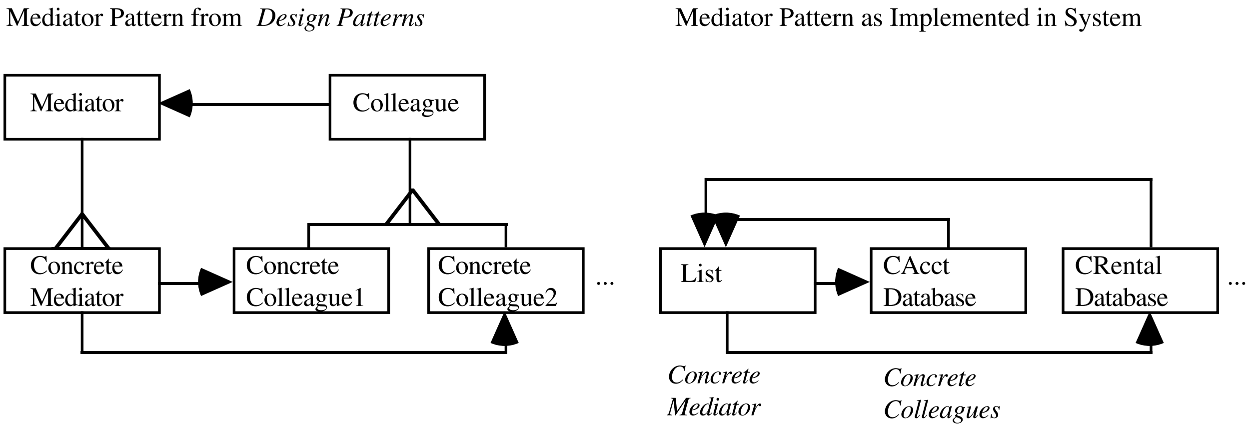
- The Proxy does provide an interface identical to Subject's. All windows have the same interface for their management as their superclass, the abstract MainWindow.
- Proxy does control access to the RealSubject, and is responsible for its creation (when an option is selected) and deletion (when the program is exited).
- However, the student implementation is not as sophisticated as the implementation suggested in *Design Patterns*. The Proxy object is expected to forward requests to its appropriate RealSubject. However, the student implementation simply creates the RealSubject and then passes control to it, letting the RealSubject handle user requests directly after that rather than forwarding requests itself. Because only one window can be active at a time in this system, the student implementation achieves the same effect as the reference pattern. However, it should be apparent that it is neither as flexible nor as reusable as the solution presented in *Design Patterns*.

Level 2

Patterns at level 2 are patterns which we detected in our student projects with some - but not all - of the same functionality as is found in a reference pattern, implemented at a comparable level of detail. It is possible that the same problem is addressed, but a key piece of functionality is missing. Or perhaps our pattern represents a very specific solution, but no effort was made to make it more generalizable.

Our projects contain 8 examples of 4 different level-2 patterns. One good example is the Mediator pattern. The Mediator is meant to serve as a mechanism for handling the interaction of other classes. It defines the way in which the Colleague classes can interact. One of our student projects implemented something along the same lines as this, using C++ templates to allow a single class to handle communication requests between database subsystems. In this implementation, all interactions between objects in the databases (the Concrete Colleagues) occur by means of communication going through the List class (the Concrete Mediator). Note that the pattern is only an inexact match because the students implemented no inheritance (i.e. there are no general Colleague or Mediator classes).

Figure 8: The Mediator Pattern and how it was implemented in our data set.



The pattern we discovered is a loose match to the Mediator in the reference set:

- Both define an object that controls and coordinates the interactions of a group of objects.
- Objects in the group do not refer to each other explicitly; they only know the mediator.
- Mediator limits subclassing, centralizes control, and decouples colleagues.
- However, our Mediator class is not subclassed from an abstract type as in *Design Patterns* - the objects implemented by the students are very specific instances that were not generalized.

The student implementation is very different, but still quite sophisticated, as the student project relies on C++ templates:

- Our pattern defines no interface for communicating with Colleague objects, but communicates via templates instead.

- ConcreteMediator implements cooperative behavior by coordinating Colleague objects. However, it "knows" its colleagues through templates only.
- Colleague classes do know their Mediator object.

6. LESSONS LEARNED

In this section, we take a qualitative look at the results of our validation. All in all, we found 22 instances of 9 different patterns across the three levels of our classification system. Given that the input to the PKB was only seven relatively small systems, this seems to us to be a relatively good number of patterns to store in the knowledge base for use in the next iteration. Realistically, however, we shall have to wait for further experimentation with PKBs to be able to tell if this is sufficient.

In terms of whether we are packaging patterns at the proper level, we note that only three of the instances we discovered were at a the same level of sophistication similar to the patterns in our reference set. We can formulate several hypotheses as to why this might be true, but unfortunately have no way of testing them currently:

- Our student subjects are not as experienced designers or implementors as practitioners, who would generally make use of such a level of sophistication.
- The system to be created did not require a high level of functionality, and our student subjects did not make their solution any more general than it needed to be for the problem at hand.
- The patterns as specified in *Design Patterns* are very sophisticated; practitioners in general do not use this level of functionality.

Obviously, this is an aspect that requires further study before we will be able to implement custom pattern libraries most effectively.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we described an inductive method making it possible to characterize OO systems with respect to their use of OODPs. The method was validated using a suite of OO projects developed in a controlled study performed at University of Maryland in a graduate level class. Although it was shown that we may not yet be packaging patterns at the right level of generality, we have shown that our method is useful for finding implicit design patterns in system architectures. We have also shown that our method can provide an initial pattern base which can be expanded on further iterations of the pattern discovery process.

We view this current work as part of an ongoing series of experiments aimed at clarifying our knowledge of patterns. Having found instances of OODPs in the suite of student projects, we next aim to experimentally assess the capabilities of OO design patterns with respect to quality, repairability, and reusability. These patterns will be statistically analyzed in order to determine which are the most error-prone of the patterns we uncover.

We plan to further validate our work by using our method to analyze a sample of OO projects developed at the NASA Software Engineering Laboratory. The SEL is currently in the process of moving to C++ from Ada, and we intend to apply our reverse architecting method in order to analyze their OO products and provide a library of useful domain-specific patterns which can be honed over time.

The results from this work seem to indicate that the creation of an OODP library would be a worthwhile endeavor; practitioners are using patterns in practice, but in general not using very sophisticated implementations. By associating metric values with the patterns in our knowledge base in the next step, we will be able to get a much clearer insight into the potential for a library. In particular, if system designers introduce extra errors into the system by reinventing these patterns from scratch, then system design could clearly be improved if a library of tailorable, error-free patterns was provided.

Finally, we intend to create tools to assist in the process of reverse architecting and automate as much of the feedback loop as possible. We feel that such tools could be of significant benefit to the design community (and intend to validate this assumption through pilot studies) as well as representing a significant challenge to construct, combining several areas at the state of the art: Program slicing techniques have great potential as a means of investigating control flow and class interactions. Reasoning about pattern matching will be necessary to find tailored examples of patterns from the knowledge base within system architectures. Perhaps most importantly, much work must be done with knowledge-based Software Engineering databases, in order to package, store, manage, reason about, and retrieve patterns which have been detected.

REFERENCES

[Basili, 1989] Victor Basili, "Software development: A paradigm for the future" (keynote address). In Proc. of COMPSAC'89 (Orlando, Fl, Sept. 1989), pp. 471–485.

[Basili, 1995] Victor Basili, Lionel Briand, and Walcélio Melo. "A Validation of Object-Oriented Design Metrics." Department of Computer Science, Univ. of Maryland, April 1995. CS-TR-3443.

[Briand, 1994] Lionel Briand, Sandro Morasca, and Victor Basili. "Defining and Validating High-Level Design Metrics." Department of Computer Science, Univ. of Maryland, June 1994. CS-TR-3301.

[Coad, 1992] Pl. Coad. "Object-Oriented Design Patterns". Communications of the ACM, 35(2):152–159. Sep. 1992.

[Gamma, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Rumbaugh, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Vlis, 1995] John Vlissides. "Reverse Architecture" Position paper for *Software Architectures Seminar*, Schloss Dagstuhl, Germany.