Reverse Engineering Design Patterns

Detecting Design Patterns in Compiled Programs.

Thesis for MET CS Master of Science

Tom VanCourt

Reverse Engineering Design Pattern Occurrences

Contents

1 Introduction

This paper describes a system for detecting occurrences of Design Patterns (DPs) in existing software. Section 2 discusses the importance of DPs to software maintenance, and notes some of the DP-related problems that arise during software maintenance. Section 3 discusses design issues and behavior of the ExPat (*Ex*traction of *Pat*terns) tool for detecting occurrences of DPs. In particular, this section describes how ExPat's search uses three sources of information:

· static information extracted from the program's files,

· a definition of the DP being sought, and

· optional user input to guide or limit the search.

Section 4 describes the experience of using ExPat, discusses related work found in the literature, and suggests directions for future development of ExPat.

This report's next section describes an implementation of DP detection, with quantitative and subjective result of the program's operation. Finally, a summary describes the major results and limitations of the current approach, with possible directions for future work.

1.1 Design Patterns: A working definition

The term "Design Pattern" comes from Christopher Alexander's work [Ale] in architecture and urban design. Much has been written about what is and is not a DP, but this is not the place for a full discussion. This project's goal is to find DP occurrences in programs, so this project's definition must be phrased in terms of the program's content:

A design pattern is a collection of relationships between some set of classes.

The exact definition of "relationship" involves classes and their fields, methods and constructors.

1.2 Design Patterns in Program Maintenance

Just as DPs create large-scale order in a program, they can preserve the order in a program under maintenance. That means that maintainers must work within the framework of DP occurrences already in the program. Those occurrences may not be at all apparent. This section shows how it would help maintainers to have tools that automatically identify DP occurrences in the code being maintained.

1.3 Detecting Occurrences in Existing Programs

A DP is a set of relationships between some number of classes. The relationships have semantic content, but no syntactic content. In other words, the programmer creates meanings in the program that are not inherently related to the program's text. Realizing the design pattern, however, generally requires some number of classes and methods to interact in defined ways. A typical DP defines a number of related classes, subclasses, method return and parameter types, and aggregations. If the DP describes a large enough number of relationships, the DP's representation in actual code may be recognizable. Java's reflection API [Fla, JDK] allows access to all those features of compiled Java code, making the search for DP occurrences credible.

1.4 Project Results

The final section of this paper discusses the implementation of ExPat. This shows how ExPat is used, and notes its the strengths and weaknesses. This section ends with a comparison to other tools described in the literature, and suggests ways in which ExPat's pattern matching could be improved.

2 Design Patterns in Program Maintenance

In the software DP literature [Bus, Coo, GoF, Gra] , one fact stands out clearly: they are *design* patterns. Most authors address only software development. This seems odd, since the largest part of software lifetime and cost lie not in creating the code, but in maintaining it. If DPs are worthwhile, they should be worthwhile throughout the software's maintenance phase. There does not, however, appear to be any significant body of literature or practice regarding DPs in maintenance. This effort seems to address problems that have not been described previously

2.1 The maintenance problem

DPs, applied at design time, create order and high-level structure in a program. Knowing that a DP is in use, the designer can typically add large numbers of classes and features with no change to the program's structure. DPs are invisible, though. Nothing about a class, except perhaps for comments, identifies it as a member of some DP occurrence. No programming language constructs identify code items as DP elements, so knowledge of the DP is often lost at the end of design. End of design is also when program maintenance begins; very often a continuous re-design.

Maintenance programmers may have no knowledge of the DPs embodied in the program, and typically no way to find them out. Lacking knowledge of the DP structure and the code items involved in the pattern, a maintenance programmer may make changes that violate the spirit of the DP.

Thus, the maintenance programmer is the proximate source of revisions that break DP discipline and regularity. That makes it easy to blame the maintainer for "bit rot" or "entropy" in the software, creeping in as the DP occurrences weaken. The maintainer's task is nearly impossible, though. It is very rare for developers to document their code well enough for easy maintenance, let alone document occurrences of DPs involving many classes. It is also rare for maintainers to keep such documentation up to date, especially when one occurrence of a DP may use dozens of classes. When a maintainer has access to only the compiled form of a class, a DP becomes even less visible.

Prudent maintainers would most likely want to make their changes fit naturally into the program's structure, as defined by the DPs used. The problem is that there is no practical way to determine which DPs are in use, or what classes take the various roles in a DP. Lacking that knowledge, it seems inevitable that maintenance will unwittingly tend to break up the structure defined by the DPs. This is not a sign of incompetent maintenance, it is a sign of an intractable maintenance problem.

There might even be a temptation to blame the DP for its failure to guide its own maintenance. That reasoning can not be supported, though. Design principles have bees around for years, operating at a DP's level of multiple interacting modules (classes). Maintenance has gone on for years with access to some code only in executable form. Even without DPs, the literature [Bro] records the decay of a system's design due to maintenance ignorant of that design. Some methodologies, such as Yourdon's "Good Enough" [You] or "Extreme Programming" [Bec] intentionally disregard existing design. In that context, DPs have no visibly worse effect on software maintenance than any older style of design.

This project's goal is to develop a tool for locating the code elements that take part in a DP occurrence. This is a reverse engineering tool – its goal is to deduce the developer's intent based on the code.

2.2 Creating a DP occurrence

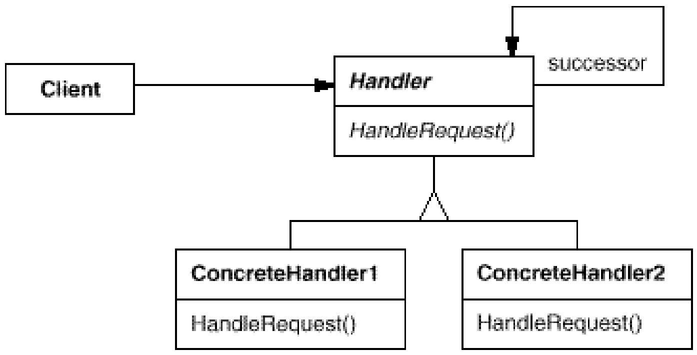
The developer has great flexibility in creating an occurrence of a DP, for example the *Chain of Responsibility* (Figure 1). The roles within that DP may be assigned (Figure 2) so that:

- class `ResidentialRate` corresponds to the *ConcreteHandler1* role and is a subclass of `TaxRate`,
- class `AgriculturalRate` corresponds to *ConcreteHandler2* and is also a subclass of `TaxRate`,
- class `CommercialRate` has no exact equivalent in Gamma's diagram but subclasses `TaxRate`,

- `TaxRate` corresponds to *Handler*. Unlike *Handler*, though, TaxRate is a concrete class that covers untaxed properties, and
- other symbol names (not shown) take the places of *successor* and *HandleRequest*.

That is all very clear to the developer. The developer may even add comments, for example, in `ResidentialRate` to note its part in a *Chain of Responsibility* DP. Nothing about the code, however, necessarily denotes participation in the DP. When the developer passes the code to the maintainers, some or all knowledge of that DP could be lost.

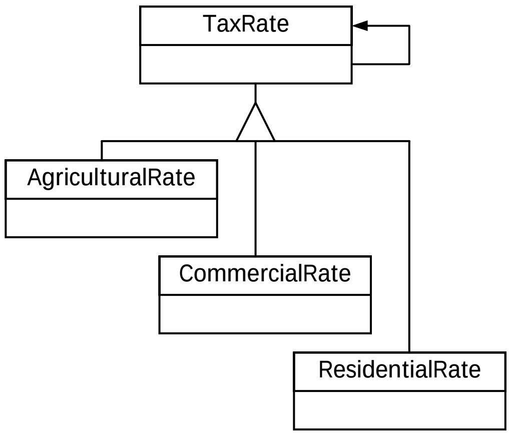
Figure 1: Chain of Responsibility
(verbatim from [GoF])



2.3 Detecting the occurrence

Looking only at `ResidentialRate`, the maintainer may find it difficult to pick out the other roles in the DP. It may even take the maintainer some thought to see that concrete superclass `TaxRate` takes a place that Gamma described as abstract. Given `ResidentialRate` and `TaxRate`, there's no simple way to determine that `AgriculturalRate` and `CommercialRate` also take roles in this occurrence of the DP.

Figure 2: Occurrence of *Chain of Responsibility*



In fact, the various `TaxRate` subclasses may live in different Java packages[1]. What's more, a subclass of `AgriculturalRate` (for example) may also act as *ConcreteHandler*, as an indirect subclass of `TaxRate`. It may take an exhaustive search to find all candidates for the *ConcreteHandler* role in this occurrence of the DP. The maintainer may need to consider inner and anonymous classes too, making the search even more cryptic and error-prone.

In short, there is very little chance that even careful maintainers will find all the classes that take part in a given DP occurrence. That means that code revisions may not be able to follow the DP. Maintenance code will tend to break up the DP, and so break up the logical structure of the software.

2.4 Sample search task

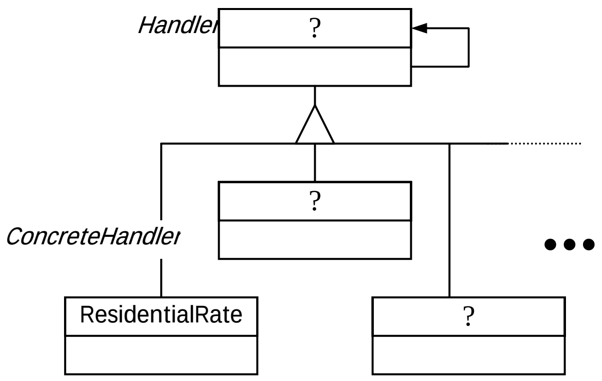
Consider the example from Figure 2. Assume that the maintainer learns, from code comments, that `ResidentialRate` takes the *ConcreteHandler* role in an occurrence of the *Chain of Responsibility* DP.

[1] This discussion centers on Java implementations of design patterns. Most of the discussion, however, can be translated into C++ or other object oriented programming language.

The intelligent maintainer may draw a blank *Chain of Responsibility* UML diagram, as in Figure 3, and fill it in as information arrives. Initially, only `ResidentialRate` as a *ConcreteHandler* is known.

The *Handler* role is relatively easy to fill in. It must be the superclass declared for the `ResidentialRate` class, or perhaps a super-superclass, … Of course, the *Handler* role could also be filled by one of the Java interfaces implemented by `ResidentialRate`, or super-interface, or interface of a superclass[2], … A persistent maintainer may be able to deduce the class acting as *Handler*, or at least find the finite set of candidates. The superclass (if any) and implemented interface[s] (if any) are written into the source code, and recursively likewise for their rest of the super*class set. Persistence, deduction, and good guessing should eventually fill the *Handler* role.

Figure 3: Guessing the occurrence structure



Filling in the *ConcreteHandler* classes, other than `ResidentialRate`, is harder. Given only one concretion of *ConcreteHandler* and the *Handler*, there is no easy way to find other classes acting as *ConcreteHandler*. The search has the form: "Locate all classes that have <the *Handler* class> as their super*class". That means "examine all classes in the program," an operation for which human patience is badly equipped. It's not even that simple, though. Another *ConcreteHandler* may be arbitrarily far down the sub*class hierarchy from *Handler*.

Such a search is tedious. Given potentially elaborate superclassing and large numbers of classes and interfaces, a person would probably find the search time-consuming and error-prone. The search is, however, deterministic and easy to specify. In other words, it is amenable to automation.

A sub*class of *Handler* is not necessarily part of the design pattern occurrence, but any *ConcreteClass* is certainly a sub*class of *Handler*. It will probably be easier for the maintenance programmer to pick through a list of *Handler* sub*classes for members of the DP occurrence than to find the set by hand. Even if an automated search found more *Handler* sub*classes than necessary, it could still help maintainers find the real members of the DP occurrence.

3 Detecting Occurrences in Existing Programs

This project defines and produces a tool to help the DP maintenance programmer. ExPat examines a body of compiled code for occurrences of the DP. It is a reverse engineering tool meant to extract DPs from the code being maintained. The description of ExPat's behavior falls into several sections:

- A description of Java as a target language. This section describes Java's reflection API as a tool for examining compiled Java code. This also describes a simple scheme for loading an arbitrary set of Java classes to be examined for DP occurrences.
- A description how ExPat represents DPs. This section starts by examining the features used to describe GoF patterns. The next part of this paper gives an informal tour of a novel Pattern Description Language (PDL), used by ExPat to represent DPs. PDL descriptions of GoF DPs demonstrate how PDL is used.

[2] For convenience, the transitive closure of a class' superclasses and super-interfaces is written as *super*class*. The *sub*class* is the transitive closure of subclasses and sub-interfaces.

This section describes DPs that can not be distinguished at the Java code level, and DPs that can not reliably be detected.

- The next section describes search constraints – requirements imposed by the user to limit the search to code elements of interest. Detailed analysis of an example demonstrates the different effects that constraints have when applied to different parts of the DP description.
- The final section describes how ExPat's matching engine tests an ensemble of Java classes against a PDL description.

3.1 Basic operation of ExPat

ExPat's search for DPs has three basic sources of information:

- A list of Java classes to examine,
- A list of pattern definitions, and
- An optional list of search constraints specified by the operator.

The program will present some user interface for specifying which Java classes to use, which patterns to look for, and what constraints to apply in the search. For experimental purposes, there may be other controls that select options in the search engine's run-time behavior.

Given those, the search program will examine the list of Java files, looking for matches to pattern definitions, subject to the user's constraints. The program's UI will present the user with data representing discovered matches. More than one match may occur on any run, if there is more than one occurrence of a DP in some body of code. The report will show how each of the matched classes, methods, fields, or constructors fits its role in the DP.

3.2 Java as target language

This project has chosen a Java-based implementation. The general logic used should be applicable to a variety of object-oriented languages, however. The only important pre-requisite is that the language's APIs provide capabilities similar those of Java's reflection API. That's not much of a constraint; most languages and object code formats give debuggers all the information ExPat requires.

3.2.1 Examining one Java file

Parsing Java source code requires significant programming effort. In particular, the parser must handle grammatical errors in the source text. Once the source file is correct enough, the class, method, field, and other declarations must be interpreted. That includes information about extended superclasses, implemented interfaces, return data types, parameter lists, scoping, etc., all with regard to the class definitions provided by other files. That programming effort is large, error-prone, and unnecessary.

Java's reflection API makes all that information, and more, available with a simple programming interface. The reflection API inherently runs on Java files known to be grammatically correct, since reflection runs on the executable class files output by successful compilation. The reflection API also allows access to library classes and others not available in source form. All these reasons argue for using Java reflection for examining code.

The reflection API has only one major drawback. Java security limits reflection to the interfaces allowed by scope declarations. In practice, this generally means that only `public` symbols are visible through the reflection API. Good programming practice [Lie] dictates that most interfaces, especially fields, be scoped very tightly – often `private`. Many design patterns, however, rely on field values for aggregation or composition. This conflict creates problems in recognizing occurrences of DPs. Later sections describe indirect ways around this problem.

Java's security restrictions are acceptable limits for this project. Test code can be designed that avoids visibility problems. Broader interpretations of a class' meaning can also reduce the effects of the Java class loader security restrictions. Production-quality tools would probably need more access to the class' interface, however.

3.2.2 Locating the compiled Java files

This project finds Java class files by examining the "." directory (ExPat execution directory) and all subdirectories, recursively, for files whose names end with the six-character string ".class". For each file found, the search treats the left substring of the file name, up to the period character, as a class name and tries to load that class[3]. If the file appeared in a subdirectory, the list of directories relative to ".", down to the file's directory, is converted into the class' package name. Failure to load a class is not reported to the user, but the class is not available for analysis. This algorithm has an obvious, implicit relationship to the CLASSPATH environment variable and loader search rules. A more elaborate intelligent implementation could make more use of the search rules.

ExPat accepts one command line parameter. If present, the application treats that parameter as a subdirectory name for the "." directory. ExPat then searches for class files only within that tree.

Once the program has started, the user may select some subset of the known classes for analysis. This selection could reduce search time or restrict searching to classes known to be of interest. The ExPat prototype has no way to load additional class files after initialization.

3.3 Pattern definitions

The DP must be specified in order to be found in the code, and some language[4] must specify the pattern. That language must be able to state all the salient features of a DP. Many WWW sources describe themselves as "Pattern Languages." All such sources known to this author are *composed of* patterns, though. They are not machine-readable languages *for describing* patterns. In the words of one author:

> "The patterns in this paper form a pattern language. A pattern language is a set of patterns that are used together to solve a problem. [Un1]"

3.3.1 Existing pattern representations

A series of references from a web search turned up mention of a "Pattern Markup Language (PML)." It was interesting that all references pointed to the same web site, and that site had only a message saying that PML was not available [Suz2].

UML is an obvious candidate as a notation for describing DPs. Several GoF DPs appear in primitives of UML[5]: a UML "Boundary" class resembles a *Facade*, a "Control" class resembles a *Mediator*, and a "Subscribe" association resembles an *Observer*. The UML specification even addresses DPs[6], albeit tersely, as a possible application of UML's extension mechanism. The UML specification, however, is long and dense. The XML DTD covers more than 120 pages, and the IDL rendering is over 160 pages[7]. One author [Dso] points out that "The UML 1.3 metamodel is already quite large and is known to contain some inconsistencies." UML complex enough that its developers have difficulty working with it; that complexity UML argues against its use in small prototypes with limited distribution.

UML raises another concern, as well, that its extension mechanism lacks any apparent way of ensuring global uniqueness. That means that different developers could unknowingly create different extensions

[3] Class file names containing a dollar sign "$" generally represent inner or anonymous classes and are ignored.

[4] Even a graphical notation must be serialized for storage, and that serialization could be considered a language. In practice, a more human-readable notation is often preferable.

[5] [UML] section 4.4.2, "UML Profile for Software Development Processes / Stereotypes and Notation"

[6] [UML] section 2.10.5, "Behavioral Elements / Collaborations"

[7] [UML] sections 6.3 and 5.4 respectively.

using the same symbols. If both developers deliver code to a common customer, naming collisions during integration could cause software failures[8].

At least one author [Suz1] has proposed a simplified alternative, UXF. The biggest reason for that proposal is simplicity – its DTD is only ~2 pages. UXF's feature set is limited and possibly evolving. Also, its general popularity is not known.

Yet another author has pursued DP analysis from a different direction, and chosen Prolog as a representation [Sef]. The representation is surely effective. It does, however, put much of the DP detection logic into the description itself, rather than letting the pattern description be purely declarative. It also presupposes a complex execution engine for case matching and unification.

In any case, it seems backwards to select a representation language before determining the semantic demands that will be made on that language. The application that uses DP descriptions should be well partitioned. That means the parser can be replaced without disrupting other parts of the matching application. Despite the importance of interoperability, etc., this application's real concern is flexibility as a research tool. Simple representation handled by familiar, lightweight tools would suit this project best.

3.3.2 Pattern description language requirements

Whatever the strengths of existing tools, their logical bases were not dedicated to DPs. Rather than commit early to a representation that may not suffice, it seems preferable to restate the problem in wholly new terms. Thus, this project uses a novel language for describing DPs. Figure 1 (Gamma's *Chain of Responsibility* pattern) suggests several requirements for the pattern description language:

- No symbol name is taken literally. Every symbol is a place-holder for some name taken from an actual code element.
- The DP includes several distinct classes. Some classes (e.g. *Handler*) appear once in each occurrence of the DP. Other classes (e.g. *ConcreteHandler1*) may appear multiple times in one occurrence of the DP. Still other classes (e.g. *client*) may be omitted without disrupting any critical aspect of the DP.
- Java permits a concrete class anywhere that the DP specifies an abstract class[9], and none of the notation distinguishes Java interfaces from abstract classes. Java also permits abstract classes to be subclasses of concrete ones. This distinction between concrete and abstract classes is too blurry for simple discussion, so no distinction is made between concrete classes, abstract ones, and interfaces.
- The *client* and *Handler* classes each have a reference to an object of type *Handler*. A description of the pattern should include references, with a data type for each reference.
- The *Handler* class (and subclasses) export a *HandleRequest()* method. Clearly, methods must be part of the description. Method return values may be specified. Parameter lists may also be specified, but should not be interpreted as an exact number, order, and possibly type of parameter.
- Some symbols in the DP (e.g. the actual names of the *HandleRequest()* method, the *successor* field, and the *client* class) have no relationship to each other. Other features of the DP (e.g. the superclass of *ConcreteHandler1*, the data type of the *successor* field, and the name of the *Handler* class) are required to match. There may be several sets of symbols, not related to each other but matching within each set.
- Many patterns ask a client class to use a method in some class referenced by the client – *Bridge* is one such pattern. Unfortunately, the reflection API can see only declarations of methods, not invocations of them. This project cannot detect use of a method, so will not define syntax representing such use.

[8] Java naming conventions and many network protocols (e.g. ISO 8802 family) have solved this problem with only small, simple mechanisms.

[9] Except in cases of multiple inheritance.

3.3.3 DP specification

ExPat represents DPs using a novel Pattern Description Language. The language has very simple syntax, described using the following conventions:

- The ∅ symbol represents a null string.
- PDL keywords and literals are underlined.
- Terminals named `symbol` represent tokens that follow Java's syntax for unqualified symbol names. They are not literal Java symbols, though. There is no significance to the shared naming convention.
- Adjacent symbols and keywords must be separated by white space, using Java's white space definition. Punctuation does not require white space separation. White space has no effect on the meaning of the description.
- Java-style comments ("//" or "/* */") are used the same way as in Java. Comments have no effect on the meaning of the description.
- The `quotedString` terminal stands for a Java-style quoted string, delimited by double-quotes (" ").
- The `patternDescription` non-terminal specifies a complete PDL representation of a DP.

The grammar appears below, where `patternDescription` is the non-terminal that represents a complete PDL pattern definition:

```
patternDescription = patternStmt classDefinition*
patternStmt = pattern quotedString ;
classDefinition = classHeader { classBody }
classHeader = modifiers class symbol superclasses
modifiers = ( many | optional )*
superClasses =   ( extends symbol (, symbol)* ) | ∅
classBody = ( reference | method )*
reference = modifiers reference symbol symbol ;
method = modifiers method symbol symbol ( paramList ) ;
paramList =   ( symbol (, symbol)* ) |  ∅
```

The prototype language has Java-like syntax for readability. Other syntax, perhaps based on XML [XML] would work equally well and would not affect ExPat's architecture. With proper encapsulation, the PDL parser can easily be replaced.

PDL semantics are complex, and subject to runtime options and constraints. A later section details the pattern-matching algorithm. This section sketches the matching engine's behavior in terms of a sample PDL definition.

Consider the *Composite* DP in figure 4, and the PDL (Figure 5) that describes it. The PDL sample displays keywords in bold type. Line by line, the PDL description has the following meaning:

1. The `pattern` statement is required in every description file. It always has exactly one quoted string. This string acts as the pattern name throughout ExPat's UI, but has no other significance.

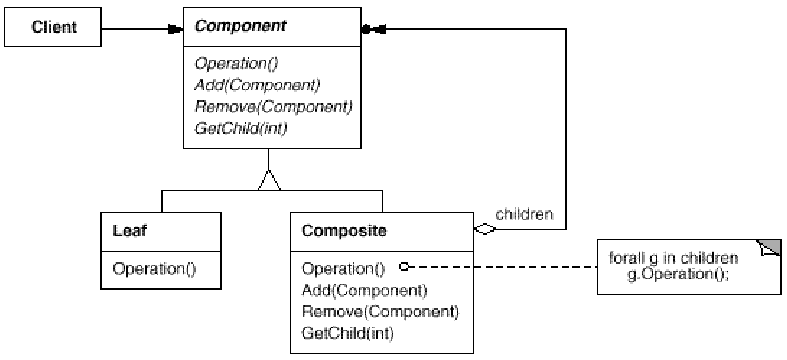


Figure 4: UML for *Composite* Design Pattern
Taken verbatim from GoF

```
1  pattern "Composite";    // Composite from GoF
2  class Component {}
3  many class Composite extends Component {
4      reference Component children;
5      method addReturn add(Component);
6      optional method delReturn delete(Component);
7  }
8  many class Leaf extends Component {}
```

Figure 5: Pattern Description Language (PDL) Example

Note that Java-style comments are allowed. As in Java, they have no semantic significance.

2. This is a class description of the simplest form. The `class` keyword starts the definition. In order to map Java's inheritance rules to into DP terms, the `class` definition actually covers both classes and interfaces. That simplifies descriptions of class patterns considerably.

   Next, the user-defined symbol `Component` takes the place of the class name. Symbols initially have no binding. During the matching process, however, the `Component` symbol will be bound repeatedly to different actual class names. The matching process is defined in the next section.

   This class has an empty body. That means that the actual class may have any content, without constraints set by this pattern definition. This definition has no superclass specification; that means this pattern will match whether or not the actual is a subclass, or subimplementation any another.

   This pattern, by the way, matches any class at all. There are no constraints in the body or in the subclassing expression.

3. This starts a more complex description. The name symbol `Composite` appears for the first time, so it is unbound when the matcher reaches this point. The class name symbol is unrelated to the quoted string in line 1, and to the name of the pattern. It is just the symbolic name assigned by GoF in their illustration of the design pattern.

   Next the `extends` keyword indicates that the pattern describing this class requires at least one super*class. More could have been required, and the pattern is still satisfied if the class extends others not specified. Java indicates inheritance by saying that

   - a class extends its superclass,
   - a class implements its super-interfaces, or
   - an interface extends its superinterfaces.

   Since PDL does not distinguish Java interfaces from classes, the `extends` keyword covers all of Java's kinds of inheritance. PDL does not require a class and its superclass need to be adjacent in the inheritance hierierchy. If class `A` extends class `B`, then `A` may be a sub-subclass of `B`, a sub^3-class of `B`, and so on, without regard to the sequence of concrete classes, abstract classes, and interfaces used.

   Note the recurrence of the `Component` symbol in this `extends` clause. Line 2 allowed the class description to match any class nameThe matcher then bound that matched class to the `Component` symbol. Later occurrences, such as this, take the `Component` symbol as bound earlier, and require that the name of this actual class match the name of the object bound to that symbol. In other words, the `Composite` class must be a sub*class of `Component`.

   Figure 4 shows only one class in the *Composite* role, but it also shows only one *Leaf*. Elsewhere in GoF the *Composite* DP is shown with several different *Leaf* classes in one occurrence of the DP. It makes equal sense to assume that GoF intends more than one class in the *Composite* role. Therefor, this class is declared with a `many` modifier. Instead of binding one code object to a matching symbol, the many keyword lets multiple objects, all meeting the PDL description match. If the pattern as a whole matches, then PDL symbols will be shown holding multiple bindings concurrently. Later sections on pattern matching describe this process in more detail.

4. This continues the class definition started in line 3. The `reference` keyword states that this class refers to some other using an association named `compositePart`, an unbound symbol. The type of that class must be `Component`, defined earlier.

   The name of the reference (`compositePart`) serves little purpose at present. The simplest interpretation of this statement is that the class must have a field of the stated type. This symbol appears in the user interface reports of successful matching, and helps the user understand how the pattern matched the actual code. Descriptions of the matching engine, below, discuss interpretations for `reference` other than field values.
5. This line defines one of the methods in the current class. The name of the method is an unbound symbol, `add`, which will match any actual method name. The return type is another symbol, `addReturn`, which is also unbound and will match any return type. The parameter list specifies another occurrence of the class that matched the `Component` pattern. This list means that `add` must have at least one parameter, and at least one of the parameters must match `Component`. Testing will determine whether the parameter should be an exact type match or should be taken as a sub*class of actual type.

   A pattern may two parameters (or more). That would match any actual method with two or more parameters, as long as some actual parameter matches the first pattern parameter and some other actual parameter matches the second pattern parameter – order of parameters is not significant.
6. This line defines another method with arbitrary name and return type, and with at least one `Component` in its parameter list. The only novelty here is the `optional` keyword. That means this pattern will match whether or not the actual class contains such a method. The `optional` keyword may also be used with `reference` and `class` definitions

   Optional elements don't affect the success of the search. Their importance lies in binding symbols to actual code elements that show more of the DP's realization.
7. This line ends the class definition started at line 3. The reader now knows that this class pattern specifies inheritance, one or two methods, and a reference to another object of specified class.
8. This line is quite simple. After as many classes as possible have matched *Composite*, look for one or more classes with the same inheritance requirement but no requirement on the class body. Like line 3, this also uses the `many` qualifier. That qualifier has one effect not mentioned earlier: it creates multiple simultaneous bindings for a symbol, representing all of the actual code elements that matched the DP.

3.3.4 Loading DP descriptions

This project uses a data file to specify the structure of each DP. Specifications are stored in a known subdirectory of the matching program, and have a `.dspat` suffix.. The filename's prefix has no meaning, but may have mnemonic significance. The pattern-matching program parses all such files during initialization, one DP per file, using the PDL described above.

This technique is just one way that pattern data could be loaded, and has no architectural significance. Patterns can be added or deleted without changing the program, but the program does need to be restarted to find the new pattern files.

3.3.5 Design patterns from GoF

This section shows how the DPs from GoF appear when cast into this PDL. Some patterns can not or should not be written in PDL. When that is true, the reasons are given. Aafter the detailed analysis of the *Composite* example above, very little additional description of PDL appears here. The reader is invited to see GoF for the DP structures represented here.

1. Abstract Factory

This pattern is unique in the way it generalizes to many kinds of *ConcreteProduct* classes. In other patterns, generalization is one-dimensional: only one list at a time has variable length. In this pattern, the main feature generalized is a Cartesian product, Factory × Product. The matching language and engine have no ways to express that kind of inter-dependency.

```
pattern "Abstract Factory";
class AbstractFactory
  {many AbstractProduct CreateProduct();}

many class ConcreteFactory extends AbstractFactory {}

many class ConcreteProduct
    extends AbstractProduct {}

optional many class Client {
  {reference AbstractFactory absFact;}
```

2. Adapter

This DP has forms based on inheritance and on aggregation. First, the aggregation model:

```
pattern "Adapter (Composition)";
class Adapter extends Target
  {reference Adaptee adp;}

class Client
  {reference Target tgt;}
```

Note a usage in this DP that may not be intuitive. The class `Target` appears in a reference and a subclassing expression, but the class seems not to be defined. Remember that a pattern symbol is implicitly defined by its first occurrence in the pattern. That means that `Target` is defined when it appears in the `Adapter` class declaration.

Here is the *Adapter* built according to the inheritance model. Note how the *Adapter* class inherits from both interfaces.

```
pattern "Adapter (Inheritance)";
optional class Client
  {reference Target;}

optional class Adaptee{}

many class Adapter extends Target, Adaptee {}
```

This pattern is quite general. It might, in practice, be used with constraints on `Adaptee` to limit the range of the search.

The declaration of the `Adaptee` symbol may seem redundant, since it accepts any class, and since the next class would have defined the symbol any way. This declaration is outside of a `many` scope so accepts only single bindings. That better suits the spirit of the design pattern, which considers only one *Adaptee* at a time.

3. Bridge

```
pattern "Bridge";
class Abstraction
  {reference Implementor impl;}

many class RefinedAbstraction extends Abstraction {}

many class ConcreteImplementor extends Implementor {}

optional many class Client
  {reference Abstraction abst;}
```

This pattern includes an `optional many` declaration. The `optional` keyword alone means zero or one, `many` means one or more, and the pair means zero or more.

4. Builder

```
pattern "Builder";
class Builder
  {method Product BuildPart();}

many class ConcreteBuilder extends Builder {}

class Director
  {reference Builder bld;}
```

This is a very general pattern. It says little more than “one class references another, which has a subclass.” A pattern like this works best when the user already knows at least one of the classes in some DP occurrence. That knowledge helps anchor the search, and keeps it drifting aimlessly around the pool of classes.

5. Chain of Responsibility

This DP has a structure very much like that of *Composite*. It will usually match about the same set of classes as *Composite*, and match them in about the same way.

```
pattern "Chain of Responsibility";
class Handler {
  reference Handler hhnd;
  method hndReturn HandleRequest();
}

many class ConcreteHandler extends Handler {}

optional client
  {reference Handler chnd;}
```

6. Command

```
pattern "Command";
many class ConcreteCommand extends Command
  {reference Receiver rcv; }

many class Invoker
  {reference Command cmdinv;}

optional many client {
  reference Command cmduse;
  optional reference Receiver crcv;
}
```

This pattern may need rewriting, in order to reduce the breadth of its wildcard many searches. This PDL risks picking up too many occurrences at a time, and presenting a report that does not clearly label matched elements by occurrence. This may work well if `Receiver` or `Invoker` is constrained before matching begins.

7. Composite

This DP has very nearly the same structure as the *Chain of Responsibility*. Most times, this DP will find all and only the classes that *Chain of Responsibility* would find.

```
pattern "Composite";
many class Composite extends Component {
  reference Component compositePart;
  method addReturn add(Component);
  optional method delReturn delete(Component);
}
many class Leaf extends Component{}
```

8. Decorator

This is another redundant member of the *Composite* family of patterns.

```
pattern "Decorator";

class Decorator extends Component
  {reference Decorator decoratee;}

many class ConcreteDecorator extends Decorator {}
many class ConcreteComponent extends Component {}
```

Composite, *Decorator*, and *Chain of Responsibility* demonstrate a basic weakness of ExPat’s analysis. They all differ at the semantic level, but have about the same representation at the syntactic (Java or PDL) level. Much of the DP’s semantic content is lost when the DP drops down to the syntactic level.

9. Façade

The UML diagram representing this class is so broad that it describes every interesting Java application. There is no way separate the unintended matches from the genuine occurrences of the DP.

10. Factory Method

The GoF UML diagram for *Factory Method* shows little more than two pairs of classes, subclass and superclass in each pair. This is another pattern that captures so many irrelevant sets of classes that it is better not used at all.

11. Flyweight

This also seems to offer too inclusive a description to be useful. Very loosely, the diagram says that the pattern is some *Factory* class A, which references class B and subclasses of B.

12. Interpreter

This design pattern has essentially the same structure as *Composite*, *Decorator*, and others.

```
pattern "Interpreter";
class AbstractExpression {}

many class NonterminalExpression extends AbstractExpression
  {reference AbstractExpression production;}

many class Terminal extends AbstractExpression {}
```

13. Iterator

```
pattern "Iterator";

class AbstractAggregate
    { reference Element aggregate; }

class Iterator {
    reference ConcreteAggregate iterand;
    method Element next();
}
many class ConcreteAggregate extends AbstractAggregate
    { method Iterator createIterator(); }
```

14. Mediator

```
pattern "Mediator";
class Mediator {}
class Colleague
    { reference Mediator myMed; }

many class ConcreteColleague extends Colleague {}

many class ConcreteMediator extends Mediator
    {many reference ConcreteColleague colMed; }
```

This was able to discover an unintended occurrence of *Mediator* in ExPat. It is also a good example of writing a pattern from most specific towards most general. Early bindings of the individual `Mediator` and `ConcreteColleague` symbols prevent the `many` declarations from accepting repeating code elements too promiscuously.

15. Memento

```
pattern "Memento";
class Originator {
  method setReturn SetMemento(Memento);
  method Memento CreateMemento();
}

class Caretaker
  {reference Memento mto;}
```

This pattern is not likely to be helpful – it matches many code combinations that lack *Memento* semantics. For example, this is likely to match any Java Bean’s set/get interface, as long as some other class makes reference to the data type at the set/get.

16. Observer

```
pattern "Observer";
class Subject
  {method attachRet Attach(Observer);}

many class ConcreteObserver extends Observer {}

many class ConcreteSubject extends Subject {}
```

17. Prototype

The structure of *Prototype* is so small, perhaps only a single class, that it will match almost every class. If this project supported the *Prototype* pattern, its description would look something like this:

```
pattern “Prototype”;
class Prototype
  {method Prototype clone();}}
```

Clearly, this will generate too many false positives to be useful.

18. Proxy

```
pattern "Proxy";
class RealSubject extends Subject {}

class Proxy extends Subject
  {reference RealSubject realSubject;}
many class client
  {reference Subject sub;}
```

This pattern finds large numbers of false hits in code that matches *Composite*.

19. Singleton

The structure of *Singleton* is so small that it matches almost every class.

20. State

```
pattern "State";
class State
  { method hreturn Handle(); }

class Context
  { reference State state; }

many class ConcreteState extends State
  { method hreturn Handle(); }
```

This is a very broad pattern, likely to trigger many false matches. It is also one of the few patterns that makes use of a method name. This states that the subclass is required to export a method that the superclass does.

This pattern suffers several problems, outside of its broadness. First, the method patterns are meant to detect over-rides in the subclass. The patterns come close, but can not distinguish polymorphic use of the same method name. Parameter list patterns do not presently detect exact number or sequence of parameters, so there is no reliable way to detect over-ride of a polymorphic method.

Second, ExPat does has only coarse tools for saying that a class should export a method itself, not simply by inheriting that interface from it super* class. With normal matching controls, the method parts of the pattern match vacuously: a subclass of A exports the interface that A exports.

Finally, this pattern may report redundant matches. If three methods appear in both the `State` and `ConcreteState` classes, the matcher will give three reports. There is no present way to ask for the set intersection of the two class' lists.

21. Strategy

The PDL definition for *Strategy* can not be distinguished from the State definition.

22. Template Method

Very roughly translated, the GoF UML diagram says that "A is superclass of B which over-rides some of A's methods." This is another place where the gap between the pattern's meaning and representation becomes painful – ExPat can work only at the representation level.

```
pattern "TemplateMethod";

class AbstractClass
  { method preturn PrimitiveOp(); }

class ConcreteClass extends AbstractClass
  { method preturn PrimitiveOp(); }
```

This description suffers same problem encountered in *State*. ExPat's matching rules are too broad to determine that the second `PrimitiveOp`'s parameter list matches the first.

23. Visitor

```
pattern "Visitor";

class Element
  {method acReturn AcceptVisit(Visitor);}

class Visitor
  {method visReturn Visit(Element);}

many class ConcreteElement extends Element {}

many class ObjectStructure {reference Element elem;}

many class ConcreteVisitor
  extends Visitor {}
```

3.4 List of search constraints

DPs have deliberately broad definitions. That lets them be applied many ways, in many contexts, in many programming languages. This means that descriptions of DPs must be so inclusive that it risks matching many non-occurrences, as well. Any useful description will necessarily generate false positive matches; collections of classes that resemble but do not embody the desired DP. Too many false positives, however, may flood the operator with useless coincidences. Constraints let an operator focus the search by including the operator's knowledge in the search.

Constrained searches may also pick out specific DP occurrences, even when there are many other occurrences of the same design pattern. A *Chain of Command*, for example, may be constrained to use actual class `ALPHA` to fill design pattern role *Handler*. The may be very helpful to a maintainer who knows only one or two of the actual symbols filling specific roles in a pattern. In the *Chain of Command*, for example, the maintainer might be able to find all of the classes filling the multiple parts of the *ConcreteCommand* role, given only one of them to start.

3.4.1 Basic uses of constraints

Search constraints are rules that allow only some subset of possible DP occurrences as matches. The following examples display the effects of constraining different elements of the DP *Chain of Command*, Figure 1. In particular, note the following symbols defined by that abstract DP:

- *Handler*, the class linked into the command chain,
- *ConcreteCommand*, a set of possibly many subclasses of *Handler*. This has *Handler* as a superclass, noted graphically in Figure 1,
- *client*, one of possibly many users of the chain. This includes some reference to the *Handler* class.
- *successor*, the link from one *Handler* to the next, and
- *handleRequest*(), the method at which service is defined. Any method implicitly has a return type and parameter list.

Exact string matching may be the simplest kind of constraint rule. In other words, it is the requirement that the code feature filling some role have a name the same as some specified string. Consider the effect of applying such a constraint, one by one, to each of the code features named Table 1, below.

The final row of the table shows that constraints typically affect more than one part of the pattern. In the case shown, one constraint defined the name of one class in the pattern, the name of another class' super*class, an association down the chain of responsibility, and an association from the *client* class.

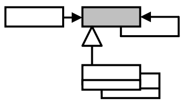
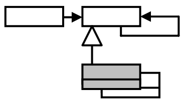
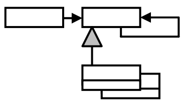
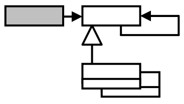
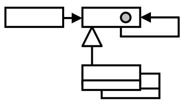
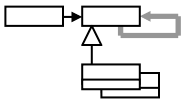
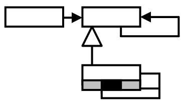
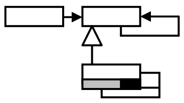
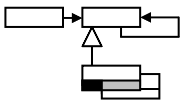
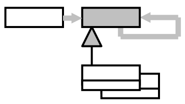
One could also imagine constraints on constructor parameters, but they have not yet proven useful. Other language constructs, such as inner classes and interfaces, exception lists, declaration modifiers, etc. could also be constrained. Design patterns tend to use simple programming constructs so that they can be applied across many programming languages. Because of that language-independence, Java-specific features may not be useful.

3.4.2 Variations on constraints and matching

There are many possible variants of constraint rules described so far, and many ways to match program features to design pattern specifications. Some have been explored in the current project's implementation. Others have yet to be tested. The following list is not exhaustive; many other possibilities exist as well.

- *Constraint string matching*: Instead of exact string equivalence, one may imagine regular expressions of arbitrary expressive power.
- *Inheritance based matching*: One may imagine many ways to incorporate a language's inheritance rules in the string matching tests. Class *X*, for example, might match another *X* or any of *X*'s super*classes. In other contexts, *X* might match any of its sub*classes. Both rules may apply: method *A(X)* with return type *Y* might match sub*classes of *X* and also super*classes of *Y*.
- *Inheritance exclusion*: Every Java class, for example, is a subclass of `java.lang.Object`. That class exports several methods into all its subclasses, i.e. into every class. The `Object` interface is rarely of interest in design pattern occurrences. Omitting `Object` from consideration will surely let pattern searching run faster, and may prevent false positive reporting. Other times, one may want only the application's content to be reported in the design patterns; all of the Java libraries (`java.*`) may be unwelcome.
- *Direct implementation*: Strict descriptions of patterns may require that a method (or field) be implemented directly in some class, not just in a super*class. Figure 1, for example, defines *HandleRequest()* in the *Handle* superclass, but implements it in all of the *ConcreteHandle* subclasses. Other times, it may be enough that the class inherit the method. Excluding super*class implementations of a method may also shorten the search for design pattern occurrences.

Table 1 – Constrained search behavior

Constraint	**Meaning**		**Consequences**
Handler = ALPHA	uniquely specifies the name of the class in the *Handler* role: ALPHA		Defines the name of the superclass required of *ConcreteCommand*, the data type of the reference from any *client*, and the type of the *successor* reference in *Handler*
ConcreteCommand = ALPHA	Demands that one of the sub*classes of *Handler* be named ALPHA		All of ALPHA's subclasses and subinterfaces are candidates for the *Handler* role.
ConcreteCommand extends ALPHA	Specifies super*class ALPHA for class *ConcreteCommand*		The pattern defines *Handler* as *ConcreteCommand*'s superclass, so this constraint also defines the name of the class acting as *Handler*.
client = ALPHA	Requires that there be a client class named ALPHA.		There may be many references from class ALPHA to various types. One or more of those references is expected to take the *Handler* role, and thus lead to all other roles.
successor = ALPHA	The name of the symbol linking the chain together is specified.		It does not seem especially useful to seek out all classes containing a field named ALPHA, but it is possible.
ALPHA *successor*	Specifies that the type of field *successor* be ALPHA		A class may contain no such fields, one, or several. Such a constraint happens not to be useful in this specific pattern.
HandleRequest = ALPHA()	Specifies that the name of some method be ALPHA		Method names, like field names, are not normally part of PDL matching. They report matches to the user.
HandleRequest (ALPHA)	Specifies that the method's parameter list includes an element of type ALPHA		
ALPHA *HandleRequest*()	Requires an object of type ALPHA as the method's return value.		
Handler = ALPHA *follow the consequences*	specifies the name of the class in the *Handler* role, and propagates name through the pattern		The *Handler* class is named in parts of the pattern other than actual *Handler*: · superclass of *ConcreteCommand* · data type of *successor* · data type of link from *client*

· *Language specifics*: Gamma's distinction of some classes as abstract has vague Java significance at best. Some may choose to ignore the distinction completely. There are also design patterns (e.g. *Singleton*) that rely on the distinction between static and instance-based fields or methods.

- *Unused language features*: Constructor matching could follow rules like those for method matching. No current design patterns refer to constructors, however, so the need has not arisen. Exception lists on methods and constructors could be added, but no present design rules use exceptions.
- *One class in multiple roles*: The strict interpretation of Figure 1 says that *Handle* is always distinct from *ConcreteHandle* classes. That does not need to be the case. Some class *C* could act as a *ConcreteHandle*, and also act as the *Handle* superclass for all other members of the *ConcreteHandle* set. Allowing such double duty may increase the number of design pattern occurrences found, but may also increase the number of false matches and the search time.
- *Method signatures*: The *HandleRequest()* method in Figure 1 is shown without parameters or return type. The strictest reading might state that *HandleRequest()* must not have any parameters and must have `void` return type. More lenient readings might allow any return type at all, and any parameter list. A method described as *F(X, Y)* might leniently be allowed any two or more parameters, as long as at least one *X* and at least one *Y* appear somewhere in the parameters, in any order. *F(X)* might even match the description, if *X* = *Y*. Constructor parameter lists might be handled the same way.

3.4.3 Implementation

The current ExPat tool uses exact string matching for locating DP occurrences. The search algorithm is recursive testing and backtracking. The search engine adds new symbol bindings as it climbs down the search tree, and releases them as it backtracks up the tree.

It is consistent with that search logic to bind some symbols to strings before the search even begins. Symbols are never over-written, and backtracking would never reach up to the level at which those symbols were bound. That is how ExPat implements constraints – as pre-bound symbols.

3.5 Searching for DP occurrences

The search engine starts with a list of Java class files, a DP description, and an optional set of constraints chosen by the user. Initially, all program symbols are unbound, i.e. they do not yet represent anything.

The search engine uses a basic backtracking mechanism. Matching proceeds through the pattern until a mismatch occurs, then backs off to the most recent successful match. New candidate code items are tried in turn, until all candidate code items have been tested.

```
pattern "Composite";
class Component {}
many class Composite extends Component{
  reference Component compositePart;
  method addReturn add(Component);
  optional method delReturn
                     delete(Component);
}
many class Leaf extends Component {}
```

Figure 6: Composite Design Pattern

When the list of candidate code items is done, matching backs up another level. Matching could have bound a symbol at any point in the matching. As backtracking steps back through each test, it unbinds any symbol bound at that step. That means that the next code item considered for matching finds a clean context, with no stale bindings still in effect.

If the last element of the pattern matches successfully, the whole pattern has been fulfilled. The match is reported to the user, showing the names of the code elements bound to each of the symbols.

3.5.1 Constrained and unconstrained matching

Searching is an exhaustive process, running top to bottom through the pattern definition. The search engine takes each class in turn, and presents it to the first part of the pattern, the *Component* class. The *Component* symbol could have been constrained by user. In that case, the string assigned to *Component* is compared to the actual class name. If the strings are the same, matching proceeds on to the class internals. If the *Component* string differed from the actual class name, though, the match fails at that point and backs up. The match engine presents the next class for matching, and so on.

The user may not have assigned a constraint value to the *Component* symbol, though. The symbol would be unbound. In that case, the name of the class is bound to the *Component* symbol, and matching continues.

Whether or not the *Component* symbol was already assigned by the user, successful matching moves on to the *Composite* class. The matching engine tries every class as a *Composite*, except for the class bound to the *Component* symbol – once matched, a class is considered used up. Dual-roles classes are not currently supported.

Assume that none of the remaining pattern symbols were assigned constraint strings. When the matching engine starts testing an actual class against the pattern, it finds the *Composite* symbol unbound. Unbound symbols match unconditionally, and take on the value of the thing against which matching tried the symbol. Thus, the matcher binds *Composite* to the actual class name and moves on.

3.5.2 Matching classes

When the matcher presents classes to the *Composite* pattern for matching, it presents abstract and concrete classes, and interfaces as well. Java's inheritance semantics allow any of the three as superclasses (or super-interfaces). Rather than create semantic and syntactic clutter, this application makes no distinction between classes and interfaces. Through the rest of this discussion, the word "class" should be taken to mean "class or interface". Likewise, extending a superclass will also mean implementing an interface.

It is quite reasonable for a programmer to look for patterns involving only the application's own classes. Classes from the Java library may not be involved in any important design patterns, as parameters, return values, etc. An option on the matching engine allows the user to ignore any classes with names of the form java.*. Ignoring these symbols yields fewer false matches and faster searches.

3.5.3 Matching superclass patterns

Next, the matching engine sees that the *Composite* class must include at least one other as a superclass. In fact, the superclass is the *Component* symbol, recently bound to a class name. A naïve implementation would simply check whether the *Component* class was among the class's immediate superclasses. Java semantics, however, would let *Composite* work equally well whether *Component* were a superclass, super-superclass, … or anywhere up the *Composite* inheritance hierarchy. The test used by the matching engine sees whether the `java.lang.Class.isAssignable()` test allows the *Component* type to be assigned to any of *Composite's* super*classes.

If *Component* is not found as a super*class of *Composite*, the matching engine backs out of the *Composite* class and releases the binding of the *Composite* symbol. The matching engine tries the next class as a *Composite* candidate, and the next, and so on.

The Java system defines `java.lang.Object` as a super*class of all other classes. Since it is present in every inheritance hierarchy, it does not help distinguish one class from another. The user may tell the matching engine to disregard all occurrences of `java.lang.Object`, to reduce false matches and to reduce the size of the search tree.

3.5.4 Matching references

If, however, *Component* were a super*class of the *Composite* class, matching continues with the content of the *Composite* class. The first element of the class is a reference to an object of type *Component*. This is a second use of the value bound to the *Component* symbol. The reference symbol name is not used.

A `reference` represents an association or aggregation in which the current class has knowledge of some other. The simplest interpretation of a `reference` describes it as a field of the class, with the given type.The first modification of that model allows any class assignable from the pattern's type as a match to the type. This uses the same super*class logic as the test used in matching the `extends` clause. The next modification lets arrays take the place of a scalar base type. The declaration

```
public SomeClass x;
```

is a reference to SomeClass, but so are

```
public SomeClass a[], b[][], c[][][]…
```

Matching ignores any number of levels of indexing.

A field is an obvious way to create a `reference` from one class to another - the field is simply assigned a reference to the target type. It is generally good programming practice to scope fields as tightly as possibly [Lie], `private` more often than `public`. That practice, however, hides the field from Java's reflection API. Reflection normally has access only to `public` symbols. In other words, good programming practice makes fields inaccessible to the matching engine. The pattern's reference from one class to another may not be visible.

Even if the reference field is public, the field may embed the reference in another object. The matching engine looks only at each field's type, not the class-typed fields inside the class definition .

These problems lead to options in the matching engine. First, consider the idea that a field must be assigned a class reference to act as a `reference` pattern. The class must have assigned the value to the field, and must have gotten the value through its external interface. The value may have entered the class through a constructor or method parameter. When looking for a reference from class A to class B, a parameter of type B to some constructor or method in A might be considered as strong enough evidence to warrant consideration. The user has separate control over treating constructor and method parameter lists as `reference` matches.

3.5.5 Matching methods

The matching engine uses all of its familiar techniques to match a method pattern to an actual method. The matching engine tests the return type, the parameter list, and the method name. The last is almost never used for matching purposes. It is important, though, for UI match reports. Parameters in the pattern may appear in any order in the actual parameter list. Also, the actual parameter list may be longer than the list in the pattern – extra parameters are ignored. Parameters match if the pattern's parameter is bound to the actual class or a super*class. Return types match if the pattern's type is the actual return type or a sub*class. As with references, levels of array indexing are ignored.

Constructors are not handled in this implementation, but could be handled much like methods.

One weakness of the current implementation is that it does not distinguish `static` methods from instance methods. That weakness affects `reference` matching, as well. The distinction should be made in the future, but has not caused problems in experiments to date.

4 Project Results

This section describes experience with the ExPat program. This demonstrates the major features of the program, without going into full analysis of every result.

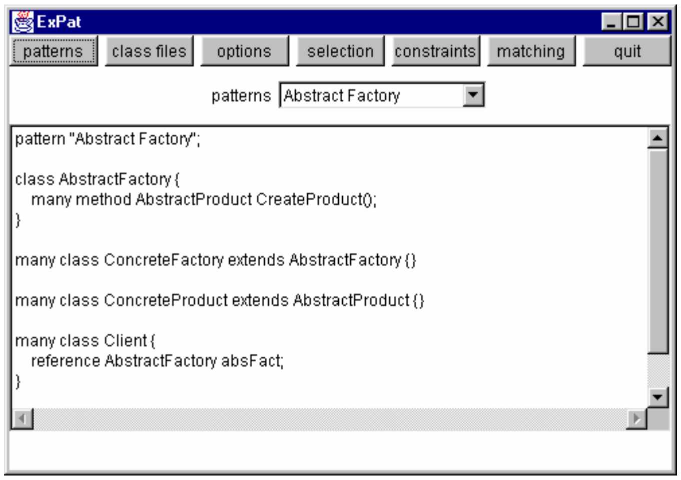
4.1 User interface

This section describes the ExPat user interface, showing how each feature supports the goal of matching DPs. The first display, Figure 7, shows the application as it appears on starting up. The top row of selection buttons is constant for all panels. A status line at the bottom also remains constant. The large display area changes according to the function selected.

4.1.1 Pattern display

The display area initially shows a pattern description, as in Figure 7. The user may select any available pattern using the pulldown near top center. The description for that then appears in the main window. This panel lets the user look at the list of available patterns and review them in detail. This shows a human-readable form of the parsed PDL, not the pattern source file. That means that comments and formatting in the original file may be lost.

Figure 7. Pattern display panel



```
pattern "Abstract Factory";

class AbstractFactory {
   many method AbstractProduct CreateProduct();
}

many class ConcreteFactory extends AbstractFactory {}

many class ConcreteProduct extends AbstractProduct {}

many class Client {
   reference AbstractFactory absFact;
}
```

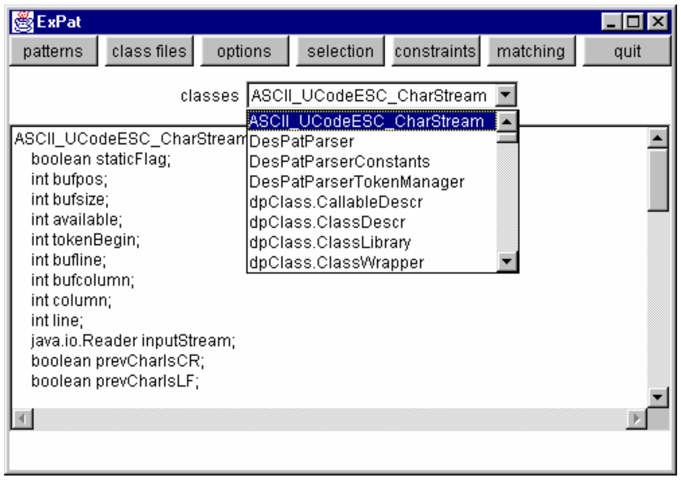
4.1.2 Class display

The next display, Figure 8, presents a pulldown for choosing among available classes. The class chosen appears in the display area.

Remember that ExPat uses only the compiled form of the class file, so it can access only the file interface, not source code. That means, for example, that method signatures can be displayed but not method bodies. Source text formatting and comments, of course, are lost.

The display includes only features used in pattern matching, methods and fields. Constructors, inner classes, and inner interfaces are not displayed. Arrays are simplified down to their base types. For example `int[][]` is simplified to `int`. ExPat uses the standard Java reflection API and security manager, so it will normally not display interface elements with `private`, `protected`, or package (default) scope. ExPat can use only `public` interface elements reliably.

Figure 8. Class display panel



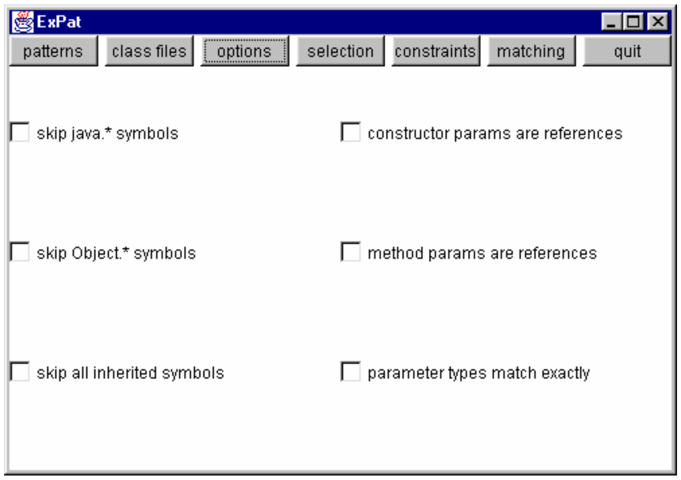
```
ASCII_UCodeESC_CharStream
   boolean staticFlag;
   int bufpos;
   int bufsize;
   int available;
   int tokenBegin;
   int bufline;
   int bufcolumn;
   int column;
   int line;
   java.io.Reader inputStream;
   boolean prevCharIsCR;
   boolean prevCharIsLF;
```

4.1.3 Option selection

The options selection panel (Figure 9) lets the user set control values that affect how pattern matching occurs. The options presently available are:

- *Skip java.** symbols. This restricts the set of classes tested as candidate members of some design pattern occurrence. When selected, any class name starting in "java." is an automatic mismatch. When deselected, matching allows Java library symbols. This feature supports users who know that all design patterns in their application are built around application class types. By eliminating the Java library symbols, the user sees fewer false positive reports and faster searches.
- *Skip Object.** symbols works the same way as "Skip java.*" symbols, but is more selective. Instead of rejecting any java.* symbol, though, this rejects only `java.lang.Object.*` symbols. Object is the common ancestor of all Java class types. That means that all classes have a common ancestor and common inherited interface elements. Skipping Object.* symbols reduces the number of matches due to unintentionally shared interface elements.

Figure 9. Option selection panel



☐ skip java.* symbols
☐ constructor params are references
☐ skip Object.* symbols
☐ method params are references
☐ skip all inherited symbols
☐ parameter types match exactly

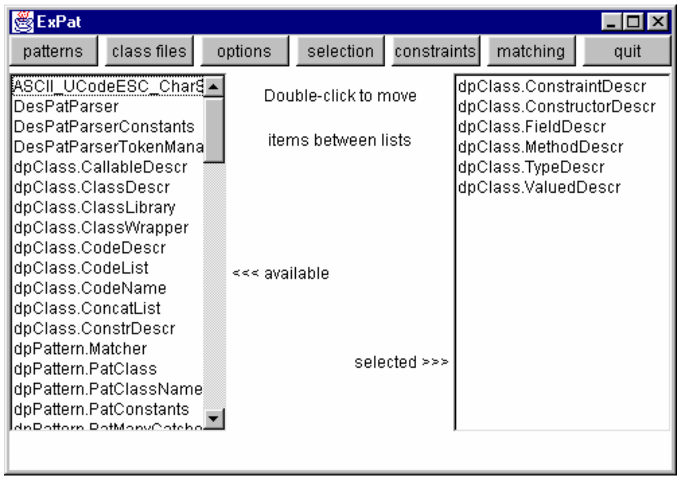
- *Skip all inherited symbols*. This means that only fields and methods defined in a class take part in pattern matching. Symbols defined in super*classes are skipped, unless over-ridden in the class being matched. Consider a class with several dozen subclasses. If the superclass takes some part in a design pattern, the subclasses trivially take part also. Some interface elements in the superclass allowed it to match that role, and subclasses all inherit those interface elements. Disqualifying inherited interfaces eliminates those subclasses from consideration, showing only the class that originated membership in that DP occurrence.
- *Constructor params are references*. Normally, a reference from one class to another would be a variable (field) of the target type. Good programming practice suggests that class data be scoped as tightly as possible to maintain good encapsulation. ExPat uses the standard Java reflection API and security manager, so ExPat can not see many properly scoped references.

  Even if the reference variable can't be seen, it is certain that the variable received a value of the proper type one way or another. The value may be created within the class, or it may be an embedded component of some other class type. In many cases, though, the value will have passed into the class as a parameter. Presence of some parameter type may be considered adequate evidence that the class holds a reference variable of that type. If that's true for constructor parameters, the user should select this option.
- *Method params are references.* There is no syntactic or semantic difference between constructor and method parameters. There may, however, be an idiomatic difference in connotation between the two. This control lets users select either or both kinds of parameter as evidence of a reference.
- *Parameter types match exactly.* An actual method or constructor with a parameter of type T normally matches sub sub*class of T. More tightly-typed applications have the option of requiring exact matching of pattern and actual parameter types.

4.1.4 Class selection

The user may want to examine only a subset of the classes loaded by ExPat, perhaps classes in one package. This panel, Figure 10, lets the user select classes to examine.

Figure 10. Class selection panel



ASCII_UCodeESC_CharS
DesPatParser
DesPatParserConstants
DesPatParserTokenMana
dpClass.CallableDescr
dpClass.ClassDescr
dpClass.ClassLibrary
dpClass.ClassWrapper
dpClass.CodeDescr
dpClass.CodeList
dpClass.CodeName
dpClass.ConcatList
dpClass.ConstrDescr
dpPattern.Matcher
dpPattern.PatClass
dpPattern.PatClassName
dpPattern.PatConstants

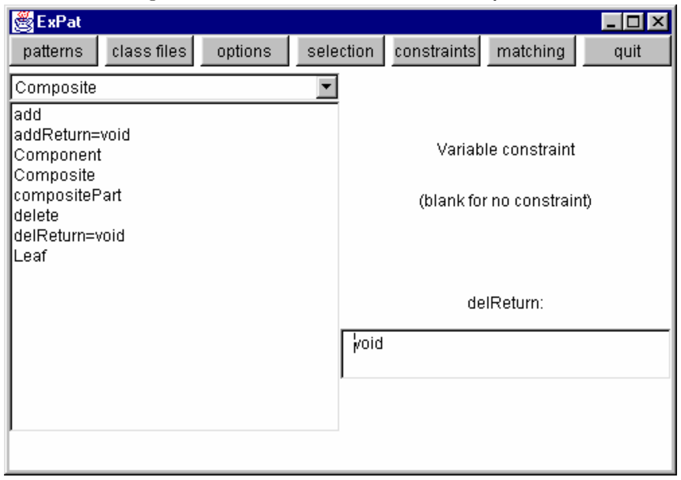
dpClass.ConstraintDescr
dpClass.ConstructorDescr
dpClass.FieldDescr
dpClass.MethodDescr
dpClass.TypeDescr
dpClass.ValuedDescr

The "selection" panel lets the user choose a subset of the loaded classes. The list on the left names the classes that are *not* to be used in matching; the list on the right is classes that should be used. If the user hasn't chosen any classes at all, i.e. if the right-hand column is empty, the whole list of classes is used for matching.

4.1.5 Constraint selection

The "Constraints" panel lets the user add knowledge to the naïve search algorithm. The first version of ExPat just uses simple strings as constraints on the pattern matching process.

Figure 11. Constraint selection panel



Composite

add
addReturn=void
Component
Composite
compositePart
delete
delReturn=void
Leaf

void

The user selects a pattern to constrain using the pulldown at upper left. The left-hand column below the pulldown lists symbols in the chosen pattern. If the symbol meanings are not clear, the user can go back to the pattern listing in the "Patterns" panel, and see the pattern symbols in context. This column is not directly editable. It does, however, show which symbols are constrained, and what the constraint string is for each. In this display, only `delReturn` and `addReturn` have constraints. Both happen to have the same value in this example, but could have been different from each other.

The caption "delReturn:" and a text entry box appear near lower right. The caption shows which symbol has been selected (by clicking in the left column) to have its constraint string modified. The user types a constraint string into the box below that legend. The new constraint string takes effect when the user hits the "Enter" key in that box. An empty constraint string represents no constraint.

4.1.6 Pattern Matching

After (optionally) setting the list of classes, choosing a pattern, and optionally setting constraint strings in some of the pattern symbols, the user is ready to run the pattern matching process. The "Matching" panel controls the pattern detection process, and appears at right.

The pulldown at upper left lets the user select which pattern to match.

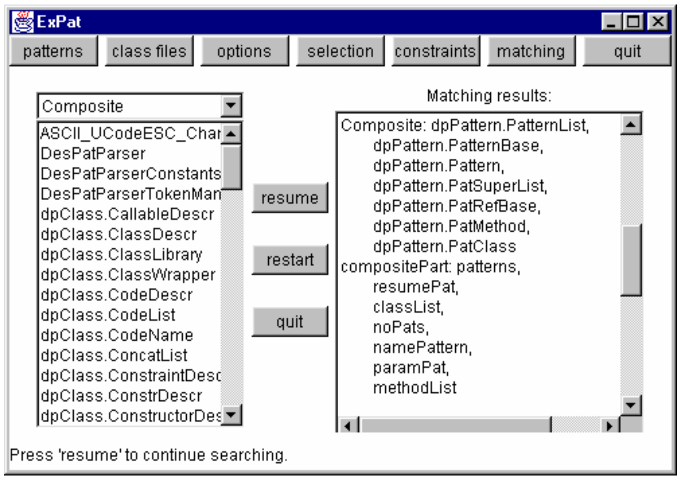
The scrolled list along the left edge lists the classes involved in the search.

The caption “Matching results:” never changes. It simply identifies the tall box on the right as output from the matching process.

Matching started the moment this panel appeared, and ran in a background thread. It may find a number of different classes that match the DP in different ways. Each time the matcher finds a new result, it shows the code elements matched each pattern symbol and waits.

The sample display shown was too long to fit the area allocated. The scroll block at right suggests how much of the result in invisible above and below the part shown. The top row reads `Composite:dpPattern.PatternList`. That means that pattern symbol `Composite` is displayed. This match bound that symbol to a code element named `dpPatten.PatternList`. Several indented lines follow that. Those additional lines are additional bindings, since `Composite` was defined in a `many` context. The next lines show bindings of actual code to the `compositePart` pattern symbol, and so on.

Figure 12. Matching panel



```
ASCII_UCodeESC_Char
DesPatParser
DesPatParserConstants
DesPatParserTokenMan
dpClass.CallableDescr
dpClass.ClassDescr
dpClass.ClassLibrary
dpClass.ClassWrapper
dpClass.CodeDescr
dpClass.CodeList
dpClass.CodeName
dpClass.ConcatList
dpClass.ConstraintDesc
dpClass.ConstrDescr
dpClass.ConstructorDes
```

```
Composite: dpPattern.PatternList,
     dpPattern.PatternBase,
     dpPattern.Pattern,
     dpPattern.PatSuperList,
     dpPattern.PatRefBase,
     dpPattern.PatMethod,
     dpPattern.PatClass
compositePart: patterns,
     resumePat,
     classList,
     noPats,
     namePattern,
     paramPat,
     methodList
```

Press 'resume' to continue searching.

The user is free to examine this match at leisure, then use one of the three buttons arranged vertically down the center of the panel. The topmost button is labeled “resume.” That lets the matching engine continue from the point at which it stopped for display. The matcher then continues until it finds another match or reaches the end of the search.

The next button is labeled “restart.” Pressing that button stops the matching process and begins again as if for the first time. The lower “quit” button (not the one in the top row) halts the matching process, but does not halt the program as a whole.

4.1.7 Quit

The main control button labeled “quit” exits the program. No data is saved; all user selections and matching results are lost.

4.2 Experience with ExPat

ExPat has been quite successful as an experimental vehicle. Its behavior, user interface, class search, and PDL file search have all proven adequate for current testing. It has been surprisingly successful in finding unintended, real instances of DPs, as well as DPs put deliberately into test classes. It has also shown where the matching algorithms require enhancement.

4.2.1 Basic example

ExPat easily recognized DPs in hand-crafted test cases. Figure 13 shows one such test. The Composite PDL (See section 3.3.3) has been presented with a set of files designed to represent the *Composite* pattern. Without looking at the actual PDL, one can read a fair bit into the assignment of actual code objects to pattern symbols.

First, it's easy to see that no method in the actual code matched the description of a method to delete elements from the aggregate. The `delete` and `delReturn` symbols are both blank, indicating that they were part of some, optional PDL declaration, and the option was not used.

Second, the `Leaf` symbol matched several classes, including one with a name that suggests a composite. That class, `COcomposite5`, contains the following Java source code:

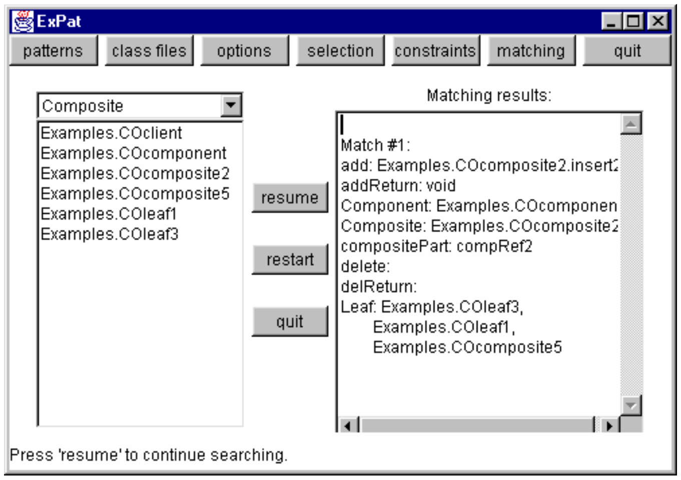
```
// COcomposite5
package Examples;
public class COcomposite5 extends COcomposite1 { }
```

By itself, that class might or might not fill the *Composite* role in the DP. It depends on the superclass, `COcomposite1`:

```
//COcomposite1
package Examples;
public class COcomposite1 extends COcomponent {
      public COcomponent compRef;
      public void insertIntoComposite(COcomponent insertMe) {}
      public void deleteFromComposite(COcomponent insertMe) {}
}
```

When `COcomposite1` was included in the set of classes for analysis (not shown), then `COcomposite5` appeared as one of the Composite matches. With `COcomposite1` included, its `public` reference to a `COcomponent` was included, so the matcher could see that reference in `COcomposite1`. Since `COcomposite5` subclasses `COcomposite1`, the matcher could see the fields and methods it inherited. With `COcomposite1` in the set under test, `COcomposite5` was recognized in the *Component* role of the DP.

Figure 13: Sample Matching Output



Examples.COclient
Examples.COcomponent
Examples.COcomposite2
Examples.COcomposite5
Examples.COleaf1
Examples.COleaf3

Matching results:

```
Match #1:
add: Examples.COcomposite2.insert2
addReturn: void
Component: Examples.COcomponen
Composite: Examples.COcomposite2
compositePart: compRef2
delete:
delReturn:
Leaf: Examples.COleaf3,
      Examples.COleaf1,
      Examples.COcomposite5
```

Press 'resume' to continue searching.

The next question is, with `COcomposite1` missing from the set of classes being analyzed Figure 13, why was `COcomposite5` accepted in the DP's *Leaf* role? Any leaf must be a subclass of the object in the *Composite* role. The answer comes from ExPat's notion of "subclass". By default, it means sub*class. When ExPat encounters "`class Alpha extends Bravo`", ExPat will treat another class `Charlie` as superclass if the `java.lang.Class.isAssignableFrom()` test states that `Bravo` is assignable to `Charlie`. That test may count on the Java class search rules and loader to find out that `COcomposite5` is a sub-subclass of `COcomponent`, even though the intermediary `COcomposite1` was not part of the test.

A few more points deserve attention. A class was allowed in the `Composite` role if it was a subclass of `Component` and referenced `Component`. The matcher cast `COcomposite2` in that role. The source file `COcomposite2.java` follows:

```
package Examples;
public class COcomposite2 extends COcomponent {
      public COcomponent compRef2[][];
      public void insert2(int i, COcomponent j) {};
}
```

The `insert2` method was put into the add (method) role of the DP, even though the actual `insert2` method has more parameters than described in the PDL, and the parameter of interest is not first in `insert2`'s parameter list.

Also note that the *Composite* PDL requires a reference from the *Composite* class to a *Component*. The most straightforward form of reference is a public field. `COcomposite2` does in fact have a public field (visible to the matcher), but the field is not of `COcomponent` type. It is an array of arrays. The base type of the array^N is of desired type, and that is good enough for the ExPat matcher.

ExPat correctly rejected those classes when tested for the *Mediator* and *Observer* DPs, and found only the one design pattern occurrence.

4.2.2 Demonstration of backtrack searching

The matcher does not assume that only one occurrence of a DP exists in any body of code. After reporting a match, ExPat waits until the user lets ExPat resume the search where it left off. Figure 14 shows such a case: a pastiche of outputs describing four occurrences in one run of the program.

This example shows an annoying problem, though. It uses two classes, `COcomposite1` and `Cocomposite1a`, which export identical interfaces (shown earlier). The Composite PDL specifies optional `add` and `delete` methods, if they exist. The PDL descriptions of the two methods are identical, and the actual pairs of `insert` and `delete` methods have identical signatures. That means the actual method named `delete` will serve equally well in the PDL's `add` or `delete` roles, and the same is true for the actual `insert` methods.

The four displays report only trivial changes, without reporting a truly new DP occurrence. Here is what happened:

Match #1: `1a.insert` and `1.insert` match the *add* role (also, `1a.delete` and `1.delete` match the delete role),

Match #2: `1a.delete` and `1.insert` match *add*,

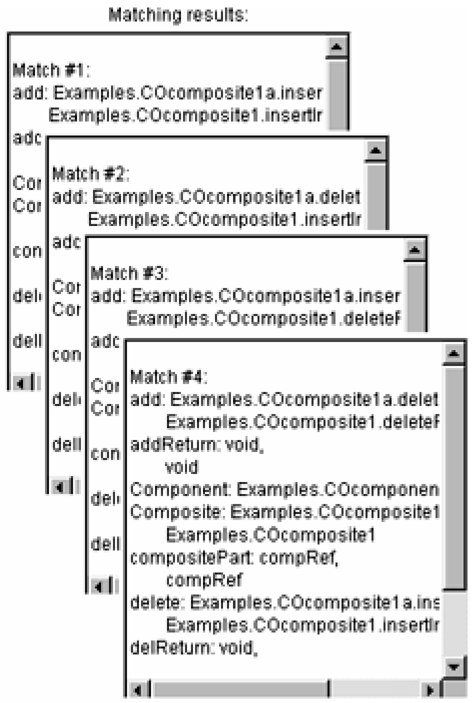
Match #3: `1a.insert` and `1.delete` match *add*,

Match #4: `1a.delete` and `1.delete` match *add*.

This demonstrates the thoroughness of the backtracking search algorithm, but also reports many trivially different matches.

If it seems odd that a method named "delete" is assigned to a pattern symbol named "add", remember that the current implementation of ExPat makes no use of any name string except as in an equality test with another. Program symbol names have no meaning to ExPat.

This problem gets exponentially worse when more actual classes take part in the match, or when more methods in each class match the PDL description. This problem can be eased by removing the problematic method declarations from the PDL, but not wholly solved.



```
Match #1:
add: Examples.COcomposite1a.inser
      Examples.COcomposite1.insertIr

Match #2:
add: Examples.COcomposite1a.delet
      Examples.COcomposite1.insertIr

Match #3:
add: Examples.COcomposite1a.inser
      Examples.COcomposite1.deleteF

Match #4:
add: Examples.COcomposite1a.delet
      Examples.COcomposite1.deleteF
addReturn: void,
      void
Component: Examples.COcomponen
Composite: Examples.COcomposite1
      Examples.COcomposite1
compositePart: compRef,
      compRef
delete: Examples.COcomposite1a.ins
      Examples.COcomposite1.insertIr
delReturn: void,
```

Figure 14: Repeated pattern discovery

A better solution might preprocess the actual class information. Given knowledge of the pattern to be matched, the preprocessor might divide the actual code elements into equivalence classes, specific to that pattern. The matcher might then be constrained to match only once with a specific configuration of equivalence classes. The solution is not obvious, and the problem can interfere with ExPat's usability. Swamping the user with trivially different results makes the significantly different harder to pick out.

4.2.3 Surprise discovery

One of the early tests with ExPat was a pleasant surprise. It detected an occurrence of the *Composite* data pattern in ExPat's own code that the author had not anticipated.

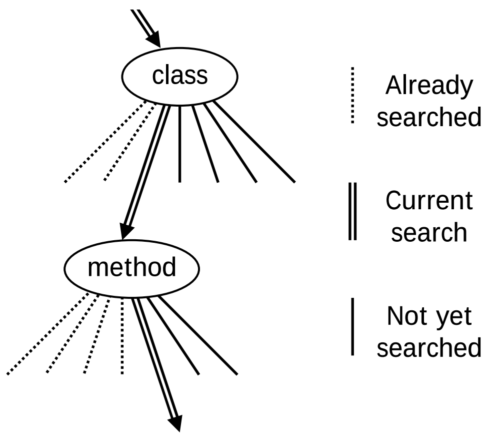
The matching algorithm is a backtracking tree search, simplified for current discussion. Very roughly, each PDL syntax element, in source order, represents an element along a linked list. Each `class` description in the PDL represents a set of branches spreading out from that node, one per actual class. Having matched some class, the matcher moves on to the next list element. That PDL `method` pattern acts as a set of branches from that `method` node in the linked list, corresponding to actual methods in the class just entered. Matching of `reference` patterns works much the same way, but chooses different parts of the class interface.

Figure 15 illustrates how the matching might work: the matcher tries each actual method from the current actual class as a possible match for the `method` PDL element. When all methods in that class have been tried, the matcher returns to the previous `class` PDL element. The matcher then tries the next actual class in the PDL `class` pattern, then tries all actual methods in that class against the `method` pattern, and so on.

Consider the case where all actual methods have been tried in the final class, i.e. the matcher has just finished with rightmost branches at several consecutive levels. ExPat maintains back-links telling the matcher where to resume, possibly many levels back along the path. Each PDL element may be followed by many others, so that all ExPat classes that examine actual methods, classes, etc. subclass a common ancestor. That ancestor maintains the back-link logic in common for all subclasses.

Figure 15: Sample search tree

```
class aClass
  { method rType aMethod(); }
```



ExPat correctly identified the back-link, the common ancestor, and the matching classes as members of a *Composite* DP. The author had been working from a different conceptual model, and was surprised to see this well-formed but unintended occurrence of a *Composite* DP.

4.3 Comparison to previous work

Only a few tools appear to have been built with design pattern maintenance in mind:

- Language extensions that check a program's logical consistency. Eiffel and R++ [Cra] both fall into this category, with assertions or similar logical extensions. Neither, however, seems able to specify relationships among several classes.
- Pattern skeleton generators, which have only limited popularity. Some generators run once, before the programmer adds any application code. Others [Bud] assume that that code skeleton conforming to the design pattern will be owned, more or less, by the skeleton tool, somewhat the way UI-oriented programming environments often work. Such tools have the obvious disadvantage of requiring a specific set of development tools. Also, they disregard the reverse engineering question completely.

- Meta-data standards (e.g. RDF [RDF], MOF [OMG], or the Dublin Core [DC]) seem to lack clear semantics relating different objects to each other, and lack any obvious mechanism for expressing indirect relationships such as super*classing. Perhaps, with further study, they could be specialized for DP description.
- Sefika [Sef] described a tool, *Pattern-Lint* for handling both static and dynamic pattern behavior. It made static and dynamic checks DP compliance, but appears to required pre-assignment of specific classes to specific roles in a DP.

Pattern-Lint seems closest in spirit to ExPat, so warrants further attention.

4.3.1 Comparison: Pattern-Lint

Pattern-Lint's purpose was slightly different from ExPat's. It seems that Pattern-Lint would be given a known set of classes with known roles in the design pattern. The tool would then verify that classes in the occurrence of the pattern followed the behavioral restrictions set by the pattern. ExPat, on the other hand, discovers classes involved in a pattern. ExPat does not, however deal with the system's runtime (dynamic) behavior.

Pattern-Lint searches were based on the Prolog language, using phrases such as the sample in Figure 16. There's a clear attraction in using a mature engine for backtrack searches, and in using a full-scale programming language for expressing DP structure. Closer examination shows some problems, however:

- The sample *Mediator* is hard-coded to handle exactly two *colleagues* under mediation. Presumably, there would need to be a mediator of each arity to handle a *Mediator* ensemble of each size. There's not apparent induction mechanism for handling an arbitrary number of colleagues
- It appears that the size of a mediator description grows O(N^2) with the number of colleagues[10]. The `control` and `data` terms in the `mediator` clause appear to relate every colleague to every other, in ordered pairs. Similar O(N^2) behavior appears in the `invokes` and `friend` terms of the `violation` clauses.
- The Prolog code is generated from other representations with yet another set of tools. Significant effort would be required creating correct translations from each tool, and providing adequate expressive power to the front end.
- The full set of system primitives was probably not exposed. Questions remain about handling of function return values, parameter lists, etc.

The system uses the same code to generate runtime analysis tools. The idea is attractive, but the authors did not state how the application would handle two or more unrelated Mediators, either instances of one pattern occurrence or multiple different occurrences.

```
mediator(M1, C1 , C2 ) :-
      invokes(C1 , M1),
      invokes(M1 , C1),
      invokes(C2 , M1),
      invokes(M1 , C2),
      control flow( C2 , C1 ),
      control flow( C1 , C2 ),
      data dependent( C1 , C2 ),
      data dependent( C2 , C1 ).

violations mediator(M1, C1 , C2 ) :-
      invokes(C1,C2).
violations mediator(M1, C1 , C2 ) :-
      not(invokes(M1,C1)).
violations mediator(M1, C1 , C2 ) :-
      is friend of(C1,C2).
```

Figure 16. Sample from [Sef]

The authors did not indicate how optional elements in a DP would appear (presumably as new Horn clauses). The authors also failed to mention any problems due to language visibility rules. Perhaps the tool suite had highly privileged access to the target code's private symbols, perhaps the C++ tools for

[10] If each Mediator has size O(N^2) and there are different Mediators for each number of colleagues M_2, M_3, …, then the size of the entire ensemble may be said to grow O(N^3)

handling object code are not constrained the way Java tools are, or perhaps they simply disregarded the issue. Still, there is much to be admired in this effort. By the way, Sefika also noted how rarely design patterns are mentioned after the earliest stages of a program's life.

All other DP support tools seem to consist of skeleton generators for outlining code that matches the pattern, or textual lists of discovered design patterns.

4.4 Directions for future exploration

Although ExPat is successful as an experimental vehicle, there are many ways it could be improved. Area worth further exploration include:

- Dynamic as well as static analyses - Such analysis could help verify correct use of communication relationships between classes.
- Re-representation of PDL. The semantic content of PDL is quite distinct from its syntax. The PDL syntax shown was selected for readability and compatibility with available tools. Some day there may be reason to recast PDL's semantics into a different syntax, perhaps one based on XML.
- More program information - Java's security manager limits the amount of class data available for analysis. Future versions of ExPat could use the Java source code itself, or use the same program information available to a debugger. This project declined those possibilities, focussing the pattern characterization and matching process, instead.
- Some patterns in GoF could not be managed by ExPat's PDL or matching engine. Those cases require further analysis.
- No effort was made to make the search run quickly. Although test runs executed quickly enough for comfort, the broad tree search would likely take an infeasible amount of time to examine a large commercial body of code. ExPat requires optimization before it can be considered a viable tool.
- Inner classes, inner interfaces, and anonymous classes are not included in any ordinary part of DP recognition. No effort is made to avoid them, though, and they may create erratic output. Future implementations should formalize semantics for dealing with them, or should prevent them from entering the matching process.
- Constructors are not named in the PDL. GoF omits constructors from DP definitions to preserve language independence. A tool committed to Java, however, could use constructors at least in some of the ways that methods are used.
- This tool searches only for relationships that must exist between code elements. It does not check for relationships that must be missing. The Mediator DP, for example, requires that none of the mediated classes refer to each other. References between mediated classes constitute violations of the pattern, but can not be phrased in the current PDL.
- This tool does not detect all forms of a pattern where one class occupies more than one role. In Figure 1, the *Chain of Responsibility* could be built from only one class type, where links in the chain differ in their data content. In that case, the UML diagram might consist of a client class and just one other, a concrete *Handler* class that links to itself.
- Some semantics of the `many` qualifier, particularly when there are several `many` qualifiers, need review. Bindings made after a `many` match may depend on which of the `many` bindings it relates to, but the matching report does not show those dependencies.
- Reduce redundant reporting. Rework the logic that prevents a matching configuration from being reported repeatedly. Redundancy occurs in some reports of `many` patterns. Given two code elements A and B that both match the `many` pattern, one matching report should report both A and B. That occurs, but sometimes B and A will be reported – the same code elements in a different order. Other times A and B are reported singly, in successive matches.

4.5 Summary

There is a long step from the semantic specification of a DP to its syntactic representation in a programming language. There is another step from source code, with full visibility of fields, methods, etc., to the interface presented by the Java reflection API. Even so, the structure of a DP can often be detected by exhaustively searching the interfaces of each class and combinations with other classes.

Complex DPs tend to work best in this search. Their stringent requirements tend to cut large branches from the search trees, letting the searches run faster. Those same requirements, specifying complex relationships of many classes, also make false matches less likely. Some DPs, however, have so few distinguishing characteristics that their description may match almost any set of classes.

ExPat is only an experimental tool. It has demonstrated some of the weaknesses of reverse engineering DP occurrences from compiled Java code. ExPat as also had some striking successes. These results suggest that automated reverse engineering of DPs from compiled code could help in maintaining some DP-based software. Further development of the reverse engineering tools would certainly improve their usefulness.

5 References

This project used a parser generator, JavaCC, from Metamata. That program accepts the grammar for PDL, and builds several files for reading PDL input. The PDL grammar is based on a sample Java grammar that is part of the standard JavaCC software package.

Except for those files, all work on this project was done by the author, specifically for this project.

5.1 Glossary

API: Application Programming Interface. The list of classes, methods, fields, constructors, and other program elements made available by some service to an application program.

Code element: This term refers generically to a class, method, parameter, field, or any other identifiable, indivisible entity that makes up a body of code.

DP: Design pattern

ExPat: (Extraction of Patterns) Experimental tool for detecting occurrences of DPs in a body of code

Field: The conventional Java term for a data element in a class.

GoF: The book or CD-ROM "Design Patterns" by Gamma et. al.

OMG: Object Management Group, an industry consortium for developing communication standards.

Pattern Instance: A dynamic juxtaposition of run-time objects that matches the set of roles and relationships specified by a design pattern. This project does not examine pattern instances.

Pattern Occurrence: A static collection of class and interface declarations, examined without executing them, that matches the set of roles and relationships specified by a design pattern. This project examines only pattern occurrences.

PDL: Pattern description language used by ExPat.

sub*class: The set of classes and interfaces composed of a class' subclasses or implemented interfaces, and their subclasses and sub-interfaces, and so on to transitive closure. Depending on context, the class itself may be included in the set.

super*class: The set of classes and interfaces composed of a class' superclass and implemented interfaces, and their superclasses and super-interfaces, and so on to transitive closure. Depending on context, the class itself may be included in the set.

UML: Uniform Modeling Language. Unless otherwise stated, this document refers to version 1.3 of the OMG UML specification.

XML: Extensible Markup Language. Standard created by World Wide Web Consortium [XML]. Originally a slightly restricted form of SGML, the XML standard has grown to include style sheet processing and other non-SGML features.

5.2 Bibliography

[Ale] Alexander, Christopher, et al. (1977). *A Pattern Language*. Oxford University Press, NY.

[Bec] Beck, Kent (2000). *Extreme Programming Explained*. Addison-Wesley, Reading MA

[Bud] Budinsky, F. J., Finnie, M. A, Vlissides, J. M., Yu, P. S, (1996) "Automatic Code Generation from Design Patterns", IBM Systems Journal, 35(2)151-171

[Bro] Brooks, Frederick. *The Mythical Man-Month, 20th Aniversary Edition*. Addison-Wesley, Reading MA

[Bus] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal (1996). *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley and sons NY

[Coo] Cooper, James W. (2000). Java Design Patterns: A tutorial. Addison Wesley Longman Inc. Reading MA

[Cra] Crawford, James M. et, al. *Path-Based Rules in Object-Oriented Programming.* Proceedings of the 1996 AAAI

[DC] Dublin Core Metadata Initiative. *Dublin Core Metadata Element Set, Version 1.1: Reference Description*. Available on 7 March 2001 at http://dublincore.org/documents/dces/

-- ANSI/NISO Z39.85-200x. "The Dublin CoreMetadata Element Set"

[Dso] D'Souza , Desmond, Aamod Sane, Alan Birchenough. *First-Class Extensibility for UML — Packaging of Profiles, Stereotypes, Patterns*. Proceedings of UML 1999 Available on 8 March 2001 at http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws25-dsouza.pdf

[Fla] Flanagan, David (1997). *Java in a Nutshell (second edition)*. O'Reilly & Associates, Sebastopol CA.

[Gra] Grand, Mark (1999). *Patterns in Java (volumes 1, 2)*. John Wiley and sons NY

[GoF] Gamma, Erich, Richard Help, Ralph Johnson, John Vlissides (1998). *Design Patterns CD-ROM: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman Inc. Reading MA.

[JDK] Sun Microsystems (2000). *Javadoc 1.3*. Available as jdk1.3/docs/tooldocs/javadoc/index.html, part of the Java documentation package at http://java.sun.com/j2se/1.3/docs.html, available on 3 July 2000.

[Lie] Lieberherr, Karl and Ian Holland, *Assuring Good Style for Object-Oriented Programs*, IEEE Software, September 1989, pages 38-48.

[OMG] The Object Management Group "Meta Object Facility (MOF) Specification, version 1.3", Available on 27 Oct 2000 as ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf

[RDF] World Wide Web Consortium. *Resource Description Framework (RDF) Schema Specification 1.0 (W3C Candidate Recommendation 27 March 2000)*, http://www.w3.org/TR/2000/REC-rdf-syntax-20000327

[Sef] Sefika, Mohlalefi ,Aamod Sane, Roy H. Campbell. Monitoring Compliance of a Software System With Its High-Level Design Models. Proceedings of ICSE 18.

[Suz] Suzuki, Junichi, Yoshikazu Yamamoto *Making UML models exchangeable over the Internet with XML: UXF approach*, Available AT http://www.yy.cs.keio.ac.jp/~suzuki/project/pub/uml98.pdf.zip on on 30 Oct 2000 .

[Suz2] Suzuki, Junichi. "Pattern Markup Language (PML)". Available on 9 March 2001 at http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/pml.html

[UML] OMG(99). *UML 1.3 (ad/99-06-08).* Available on 23 June 2000 at ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf

[Un1] Unknown author #1. Available as http://st-www.cs.uiuc.edu/users/droberts/evolve.html on 4 Mar 2001

[XML] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0W3C Recommendation 10-February-1998*. Available at http://www.w3.org/TR/1998/REC-xml-19980210. On 7 March 2001.

[You] Yourdon, Edward (1997). Death March. Prentice-Hall inc. Upper Saddle River NJ

5.3 Trademarks

These and all other trademarks are property of their respective owners. The author looks forward to correcting any errors in proper attribution of trademarks, service marks, etc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, and Windows NT, are trademarks or registered trademarks of Microsoft Corporation.