Working With Patterns and Code

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu
401-863-7641, FAX: 401-863-7657

ABSTRACT

This paper describes the basis for a suite of tools that let the programmer work in terms of design patterns and source code simultaneously. It first introduces a language for defining design patterns. This language breaks a pattern down into elements and constraints over a program database of structural and semantic information. The language supports both creating new program elements when a pattern is created or modified and generating source code for these new elements. The paper next describes tools for working with patterns. These tools let the user identify and create instances of patterns in the source code. Once patterns are so identified they can be saved in a library of patterns that accompanies the system and the patterns can be verified, maintained as the source evolves, and edited to modify the source.

1. Introduction

Patterns are central to the design and development of software systems. They encode the experience of designers; they provide tested solutions to what are sometimes difficult or tricky problems; they make it easier to describe and understand complex systems; and they raise the level of abstraction during system design and development.

Patterns in software development occur at many levels. At the lowest level, code patterns form the cliches[8] or idioms [6] that developers use to make simple algorithms and constructs easier to code [8,25,28]. Most of these common cliches have been incorporated into language features (such as *for* loops) or into simple language extensions (such as iterators in STL). At the highest level, architectural patterns form the basis for the top-level design of complex systems [24]. Pipe-and-filter and implicit-invocation models describe high-level architectures that are applicable in a variety of contexts. Architectural description languages and high-level prototyping systems have evolved to provide support for these patterns.

In between these two levels are design patterns. These are used today in object-oriented programming to prescribe the use of classes, methods, and objects in addressing many of the problems that arise in complex systems. Such design patterns range from simple, language-supported concepts such as templates to high-level ones such as interpreters. One of the goals of our work is to provide appropriate tool support for such patterns.

Design patterns in general do not fit nicely at either the language level where code patterns are defined or the metalanguage level of architectural patterns. They typically involve a small set of lower-level constructs, such as methods or data fields or even code fragments, that are spread out across a set of classes and hence in several different files in different parts of a system. While there have been some attempts at providing language-level support [3,10,15], these only handle the simpler patterns and do not address the complex and important issues raised by the more involved patterns.

In order for design patterns to achieve their full potential as a tool for programming it is important that they be fully incorporated into the development process. Currently, because there are no support mechanisms, patterns are used during the design stages but are then dropped, with the implementation possibly (but not necessarily) reflecting the original patterns. Patterns are often used for creating code or code fragments but then they are forgotten. Code added during program evolution and maintenance may or may not conform to the pattern. Finally, patterns are used to describe code. Even here, because the code might have evolved from when the pattern was originally envisioned, the description might be inaccurate or even misleading.

The PEKOE system, part of the TEA project, is our first step in providing the tools needed to make design patterns an integral part of programming. Rather than attempting to support patterns at the language level, we show that it is possible to provide software tools that can effectively promote patterns to first class objects throughout the development process. These tools let the programmer maintain the set of patterns in the system and ensure that these patterns are valid as the system evolves. They let the programmer design and create code by adding instances of patterns to the system. Most importantly, however, they let the programmer edit both the patterns and the code simultaneously as the system is maintained. Here the code can be edited by modifying the patterns and the patterns are maintained and checked as the code is modified. This is the first effort that we know of that takes a comprehensive approach to supporting patterns.

PEKOE is also a first step toward the more general goal of letting the user work at multiple levels of abstraction simultaneously. Our eventual goal is to support a wide range of editable abstractions or views of the program and to let the user work in multiple views simultaneously. While there has been much previous work aimed at addressing this problem [17], as well as on-going work involving round-trip strategies with various CASE and user interface tools, none of these efforts has been particularly successful. PEKOE illustrates a new approach that is based on user interaction and dual mappings. This approach uses the source code as the primary representation and derives semantic information from the source to build the views. The tools described in this paper show the feasibility of the approach.

The remainder of this paper describes the foundations of the PEKOE system. We start by defining what a design pattern is. Most of the books and articles that present such patterns do so in an informal manner, taking the approach that you'll know it when you see it. To provide tool support for patterns we require a much more formal definition. Our approach involves a modified query language with extensions

for creating and editing patterns. This approach is unique both in its inclusion of semantic as well as structural information and in supporting location, creation and editing simultaneously. We then cover the mechanics of what it means to find, create, maintain, verify, and edit patterns in a system. These are illustrated by the prototype PEKOE tool which serves as an implementation of the covered concepts. We conclude by noting several related projects, our limited experiences to date, and the issues we plan to address in future research.

2. Defining Patterns

A design pattern in the traditional sense is a description of a problem along with a solution. It consists of four elements: the name, a description of the problem situations where the pattern may be applied, a description of the program elements that make up the pattern, and the consequences, i.e. the results and trade-offs of using the pattern [9,20,27]. A pattern in this sense is typically described by giving one or more examples of a problem and the application of the pattern to that problem, along with a semiformal description of those elements of the solution that are considered part of the pattern.

Such a description is too informal if patterns are to be supported directly by tools. For tool support we need a formal definition of a pattern that captures its implementation but not its intent. (A system where the user can formulate the problem in a non-programmatic way and then find appropriate patterns would be interesting, but is probably beyond the limits of current technology. Simpler systems that guide the user to one of a set of design patterns based on a series of choices have been built [5].) To this end, we note that the implementation of a pattern consists of a collection of program elements that satisfy a set of conditions. Here a program element could be a class, a method, or a data field. The conditions reflect relationships among these elements, other properties of the elements such as their data types, and what the code inside the elements does.

As an example, consider the Prototype design pattern defined in [9] and shown in Figure 1. Prototype is a pattern used for creating new instances of an object where the new object's properties tend to change over time. For example, in a graphics editor, a newly created square should acquire the current color, line styles, line width, fill style, etc. To make it easier to create such objects without having to remember all the different properties, the pattern involves creating a prototypical object (representing the square in this case) that has all the current properties, and then cloning this object when a new instance is needed. This ensures that the set of properties need not be known outside of the prototype object and lets the creator build the new object without having to separately set each of the properties. It also makes it much easier to add new properties as the system evolves since changes only have to be made in the object and not in the code that creates or uses the object.

An implementation or instance of the Prototype pattern consists of a class *Prototype* which contains a *Clone* method. This class can actually represent the root of a hierarchy where each of the concrete classes in the hierarchy has its own *Clone* method. There must also be a client class that has a data field to hold the prototype. The con-

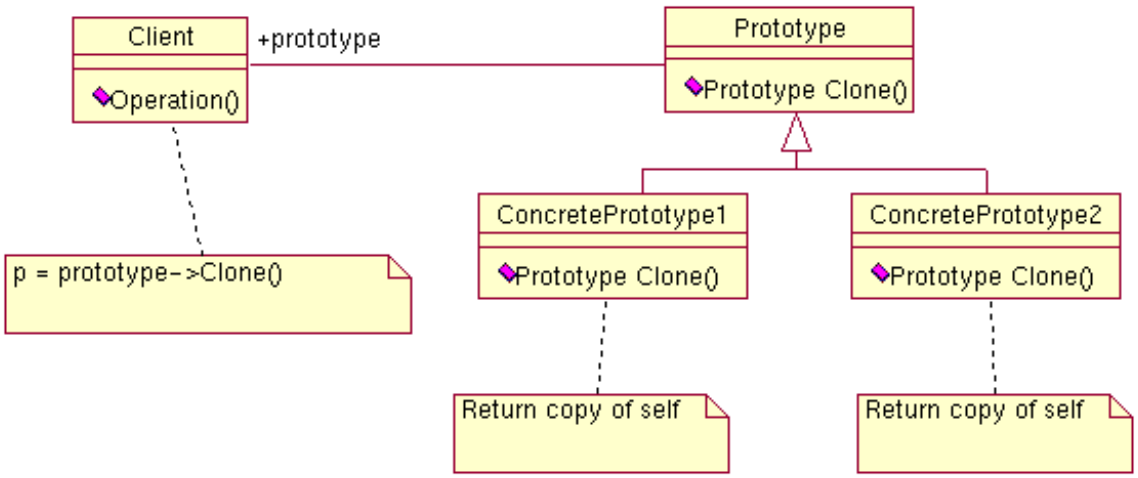

FIGURE 1. Class structure diagram for the Prototype design pattern.

ditions that are implicit here are that the *Clone* method return a new instance of the prototype, that each concrete class have its own *Clone* method, and that the client creates new instances of the *Prototype* class by calling the *Clone* method.

This informal description of the Prototype pattern can be formalized by viewing the pattern as a query against a database of program information. The query in this case returns the various elements of the pattern which can either be individual program elements or sets of such elements, for example, the set of all concrete subclasses of the *Prototype* class. The query then expresses the relationships among the program elements and the various conditions that need to hold for the pattern to be valid. To a first approximation, this is the approach that we have taken. However, this approach by itself needs to be better qualified and extended to cover the whole range of patterns.

There are three problems that have to be dealt with in converting the informal description of a pattern into a formal basis for program tools. The first involves determining what information must be contained in the database that will be queried to identify the patterns. The second involves identifying external information such as naming conventions that is needed to define patterns. The third involves attempting to accommodate the flexibility inherent in an informal description within a formal framework.

We first analyzed what information must be contained in a database used to define patterns. The obvious starting point is a cross-reference database such as that provided by Sun's source browser [26] or the .bsc files produced by Microsoft's Visual Studio. These essentially contain a symbol table for the program, providing information for each defined symbol, scoping information, file information, a description of the class hierarchy, and the locations where each symbol is referenced. Such a database can be used to define most of the patterns that have been proposed and various systems have used some sort of query or analysis of such databases for identifying patterns [2,12,14,16,18].

However, if one looks at patterns in more detail, one finds that the constraints that actually identify a pattern are sometimes more semantic in nature. Simple semantic constraints such as the return types of functions or the types of parameters may or may not be contained in a cross-reference database. These are needed fairly frequently. Other information requires analysis of what the code does. In the Prototype pattern, for example, one needs to understand that the *Clone* method actually returns a copy of the current object. Many of the patterns are defined to include code fragments that reflect a significant part of the meaning of the pattern. Therefore, the need for such analysis is common.

TEA, the framework that encompasses PEKOE, provides a general database of program information that is designed to include most of the information needed for identifying patterns. It starts with a rich cross-reference database of sets describing various program elements. In addition to information about files and file dependencies, this database contains the information a compiler would gather in its symbol table in order to do symbol lookup as well as type resolution for expressions. In particular, it includes:

- *Files*: a description of each file in the system and its dependencies.
- *Definitions*: a description of each definable program element including its name, symbol type, data type, scope, and flags such as `TASTE_DEF_SYSTEM` to denote a system definition or `TASTE_DEF_ABSTRACT` to denote an abstract definition.
- *References*: an entry for each reference to a program element that includes a pointer to the definition it references as well as flags indicating the type of reference.
- *Scopes*: information about the complete scope hierarchy of the system including all the symbols defined in each scope and the relation of this scope to others.
- *Parameters*: descriptions of the additional properties of function parameters.
- *Types*: the complete type algebra for the types defined and used in the system, including system types, user-defined types, and implicit types such as the types of each method or function.
- *Parents*: data describing the class and interface hierarchy and inheritance relationships.

The database is organized as sets of cross-linked objects. For example, reference elements point to the corresponding definition elements while definition elements point to the type element of the definition and the scope element containing the definition. TEA also supports a more dynamic database that is built for files that are currently being edited. The overall database interface lets the application query either the permanent database or, where more up-to-date information is required, a combination of the permanent database and the dynamic databases for each file currently open in an editor.

In order to provide semantic information for defining patterns, the database in TEA contains virtual links for each item to the relevant node of a full abstract syntax tree representation of the source. Queries or small programs can be written to gather specific information from the abstract syntax trees. For example, a simple query can be

written to test if the *Clone* method actually calls the constructor for the current type or the Java built-in *clone* method while testing the Prototype pattern. Furthermore, a data flow analysis routine can be applied to the syntax tree for the method to ensure that the value returned from the function is the result of this call. The ability to write arbitrary analysis routines and then make use of them for identifying patterns offers PEKOE the power to undertake most of the desired semantic tests without having to rely on the programmer to tell it what the code is doing.

The links in this case are virtual in that they are not really only references to the node id for the given source file. The actual abstract syntax trees are not stored in the database. Instead, if a query or method attempts to access one of the links, the tree is created and cached by reparsing the source. This caching strategy, similar to that of Visual Age C++ [13,19], ensures that the database remains of reasonable size (roughly 4-8 times the size of the source) while still providing adequate performance.

Identifying and creating some of the proposed patterns requires additional information in the form of mappings and associations. The Prototype pattern contains a simple association: there is a separate *Clone* method associated with each concrete subclass of the *Prototype* class. Other patterns contain more sophisticated associations. For example, a Visitor pattern is used to add functionality to a hierarchical structure without creating additional methods. Here an *AbstractVisitor* class is created to represent arbitrary functionality. This class has an abstract method for each concrete instance of the original structure. A concrete instance of this class is created whenever new functionality is to be added. Each class in the original structure implements an *accept* method that takes an instance of the *AbstractVisitor* class and calls the method specific to that class. Here there is an association from the method in the *AbstractVisitor* class to the concrete subclass in the original structure. This association is based on a mapping between the names of the two elements, for example associating the method *visitIntConstant* with the class *PekoeIntConstant*. A full definition of a Visitor pattern thus requires a specification of the name mapping in addition to the association.

The informality of the current descriptions of design patterns allows a single representation to denote many different variations. For example, in the prototype pattern, we noted that the client class has to have a data field that contains the prototype. In reality, the client class needs only to have a handle to such an instance. This handle could be a data field or it could be stored in a hash table or even accessible via some global structure.

There are two ways of handling such flexibility in a formal, tool-oriented system. The first is to attempt to define each pattern in the most general way possible. This can be done to some extent, for example by noting that the *Clone* method can either call the Java *Object.clone* method or call a constructor. This approach, however, cannot handle all the possible variations that one might want, since in doing so one would lose the essence of the pattern and be left with nothing.

A more reasonable approach is to view these variants for what they are, additional patterns. In this approach, each instance would have its own pattern definition. This is relatively easy to do and has the additional advantage that it lets us better associ-

ate code with a pattern so that patterns can also be generated. The approach has the disadvantage in that it will eventually create a much larger library of patterns and will thus make it more difficult for the programmer to select appropriate ones and to maintain an understanding of their system. Our future research involves finding ways of addressing this issue and simplifying the definition of pattern variants.

3. A Pattern Definition Language

To meet these varied requirements for defining a pattern, we have developed a simple prototype language. This language effectively defines the set of queries needed to identify instances of the pattern from the database of program information. Since the database is essentially a collection of sets of objects, we have modeled the language on the object-oriented query language OQL [7]. The language is prototypical in that we are still refining the various syntactic details, attempting to simplify it and make it easier to use, and we are working on extensions for pattern variants, specifying the dynamic behavior of patterns, and pattern editing.

To illustrate the language, we consider the Prototype pattern discussed in the previous section. A simplified form of the pattern definition is shown in Figure 2. The first line provides the name of the pattern. This is followed by a series of GIVEN and DEFINE sections that specify the various program elements that compose the pattern and optional REQUIRE and CHECK sections which specify the constraints on the pattern elements. Elements in a GIVEN section are assumed to already be part of the code, whereas elements in a DEFINE section are specific to the pattern and will be created if the pattern is being created.

Each element is represented as a variable that can either represent a single program element or a set of elements. The first variable, *prototype*, represents a single value, the class to prototype. The second, *alts*, represents the set of all classes that are subclasses of *prototype* and that are not abstract. The ALL term indicates that all such elements should be considered part of the set. The language also permits SETOF here to indicate a more flexible selection of matching elements.

There are two things to note in these definitions. First, the order in which the variables are specified is important. This order is used to actually find candidate elements for each variable and thus controls the search. Moreover, the definition of one element can refer to the value of any preceding element but not to later ones.

The second thing to note is that the functions and types used in the definition are not actually primitives. The underlying database, as we noted, provides a limited number of sets. Class elements are actually definition elements that happen to represent a class or structure. Determining whether a class is a superclass of another requires looking at the type fields of the two types. Checking whether a type is abstract or not requires looking at the flags in the corresponding definition entry. The pattern definition language provides a schema feature similar to that of Z [30] or of database views to handle such definitions. The corresponding definitions are shown in Figure 3.

```
PATTERN Prototype

GIVEN
   prototype “Class to prototype” : Class ;
   alts : ALL Class : c | isConcreteSubclassOf(c,prototype);

DEFINE
   clone “Method used to clone the prototype” : Method |
            clone.baseClass == prototype;
   clone_methods : ALL Method : m | m.name == clone.name AND
            m.baseClass IN alts;

GIVEN
   client “Class to contain the prototype” : Class;

DEFINE
   proto_field “Field to contain the prototype” : Member |
            proto_field.type.definition == prototype AND
            proto_field.baseClass == client;

REQUIRE
   NOT prototype.testFlag(TASTE_DEF_SYSTEM);
   NOT client.testFlag(TASTE_DEF_SYSTEM);
   EXISTS c IN Class : ( NOT c.testFlag(TASTE_DEF_SYSTEM) AND
            isConcreteSubclassOf(c,prototype) );
   COUNT(alts) > 0;
   clone.returnType == prototype;
   FORALL c IN alts :
      EXISTS s IN clone_methods :
    ( s.baseClass == c AND s.returnType == prototype);

CHECK
   callconst “Clone method must call constructor” :
      FORALL c IN alts :
    EXISTS s IN clone_methods :
       ( s.baseClass == c AND s.returnType == prototype AND
            callsConstructor(s,c) );

END
```

FIGURE 2. Prototype pattern definition.

Each of the variable definitions in Figure 2 specifies a prompt string used to inform the user of the corresponding role of the variable in the pattern, a type that restricts the domain of the variable and an optional set of constraints that the variable must meet. Additional constraints are then given in the REQUIRE and CHECK sections. Constraints in these sections can be given in any order. They are generally tested as part of the selection process for a given variable when all other variables in the expression have been assigned. For example, the first REQUIRE clause is tested as part of the specification of the *prototype* variable while the last REQUIRE clause is checked as part of *clone_methods* since the other variables it uses, *alts* and *prototype*, are already assigned at that point.

The difference between the REQUIRE and CHECK sections are that the requirements must be present for the pattern be considered to exist, while the checks are tested for but not required. If a CHECK clause is violated, the pattern will still be considered present, but the user will be warned that it is either incomplete or incon-

```
PROVIDE Class
   RETURN
   SELECT d FROM d : Definitions WHERE
         d.symbolType == TASTE_SYM_CLASS_TAG
         OR d.SymbolType == TASTE_SYM_STRUCT_TAG ;
END Class


PROVIDE isSubclassOf( sub : Class, sup : Class)
   RETURN
      sub == sup OR sub.type.isSuperType(sup.type) ;
END isSubClassOf


PROVIDE isAbstract( c : Class )
   RETURN c.testFlag(TASTE_DEF_ABSTRACT) ;
END isAbstract


PROVIDE isConcreteSubclassOf( sub : Class, sup : Class )
   RETURN
      isSubclassOf(sub,sup) AND NOT isAbstract(sub)
END isConcreteSubsumedType
```

FIGURE 3. Auxiliary definitions for the Prototype pattern.

sistent. In the example of Figure 2, the single CHECK clause tests whether the clone method actually calls a constructor. (This test is actually done in a language-dependent manner since in Java it is sufficient to call the built-in *clone* method which calls the right constructor implicitly.) The primary reason for making this a check rather than requiring it is that when the pattern is first created the newly defined method will not have associated code and this condition would not hold. A deeper reason is that it is impossible in general to detect if a constructor is actually being called unless the user is willing do so quite directly.

The language provides some additional constructs that are not illustrated by this example. In particular, it allows the definition of name mappings using regular expressions and stored associations among program elements to address the naming issues described in the previous section. The name mappings are defined by specifying a regular expression to match and the substitution pattern that should be used if the match succeeds. Another language construct support associations. These are query-based functions from one program element to another. They are computed on demand and the result cached for efficiency. An example of these can be seen in the extract from a Visitor pattern definition shown in Figure 4.

4. Finding Pattern Instances

The pattern definitions described above are designed to make it relatively easy to identify instances of design patterns in a system. Such a search is done by looking at the variables specified in the pattern definition in the order they are listed. For each variable, the set of candidate objects is determined by constructing and evaluating a query against the database. This query uses the variable's type to define the initial

```
PATTERN Visitor

MAPPING
    classToMethod (c "Class to visit" ) => "visit$1"
    methodToClass (m "Method to visit" [ "^visit(.*)$" : "$1" ]) => "$1"

GIVEN
    root : Class | isAbstract(root);
    elts : ALL Class : c | isConcreteSubclassOf(c,root);

DEFINE
    visitor : Class | isAbstract(visitor);
    accept : Method | accept.baseClass == root;
    vmethods : SETOF Method m | m.baseClass == visitor;
       amethods : SETOF Method m | m.baseClass IN elts AND m.name ==
                  accept.name;

ASSOCIATE
    aclass ( m : amethods ) => SELECT c FROM c : elts WHERE m.baseClass == c;

REQUIRE
    FORALL e IN elts : EXISTS m IN amethods : aclass(m) == e;
    FORALL m IN amethods : calls(m,classToMethod(m.baseClass));
    ...

END
```

FIGURE 4. Extracts from the Visitor pattern definition.

```
SELECT d
FROM Definition : d
WHERE
   (d.symbolType() == TASTE_SYM_CLASS_TAG OR
      d.symbolType() == TASTE_SYM_STRUCT_TAG) AND
   NOT d.testFlag(TASTE_DEF_SYSTEM) AND
   EXISTS c : Definition
      WHERE ((c.symbolType() == TASTE_SYM_CLASS_TAG OR
               c.symbolType() == TASTE_SYM_STRUCT_TAG) AND
         NOT c.testFlag(TASTE_DEF_SYSTEM) AND
         (c == d OR c.type().isSuperType(d.type())) AND
         NOT c.testFlag(TASTE_DEF_ABSTRACT)
```

FIGURE 5. Expanded query for the variable *prototype*.

set of candidates and then applies any restrictions in the variable's definition as well as all REQUIRE clauses that involve only this and previously defined variables.

As an example, consider a search for a Prototype pattern given the above definition. The equivalent OQL query for the first variable, *prototype*, that is generated by the system is shown in Figure 5. Building this query involves expanding the various auxiliary definitions and including the first and third REQUIRE clauses, both of which only involve the variable *prototype*. The system evaluates this query to find the set of candidate elements for *prototype*. Then, since the variable *prototype* represents a single element, the system continues the search by looking at the subsequent variables for each value in this candidate set.

The second variable, *alts*, represents a set of program elements. The ALL phrase indicates that all elements matching the search criteria must be included in the set. Thus, for each *prototype* candidate, the system would generate the corresponding *alts* set using the appropriate query. Given this set, the system would next find all potential values for the *clone* variable. For each value, it would continue the search, building the set of *clone_methods*, then finding appropriate candidates for *client* and *proto_field* in turn.

While this search can be done without any user interaction, we felt that it was best to make it an interactive process. There are a variety of reasons for this:

- Often, the user is looking for a particular instance of a pattern in the code and is not interested in all instances.
- Since the semantic criteria of some of the pattern definitions are difficult to test, the matching process can yield some false instances which the user will want to eliminate.
- Since we want to make patterns first-class objects, we need the user to name and save each pattern instance as it is found.
- We wanted to point out patterns that match but where one or more CHECK clauses might be violated.
- Some aspects of the pattern specification such as name mapping information, are probably best defined by the particular user rather than saved as part of the pattern definition since they are dependent on coding styles which vary from one user to another.
- Finally, in a large system a complex search can be quite time-consuming and we wanted to let the user narrow the search to those portions of the system that might be new or more relevant.

The front end that we devised consists of a series of dialog boxes that let the user select or restrict the definition of each variable. A sample series is shown in Figure 6. The first dialog box serves as a splash screen to start the search process. The second one asks the user to identify which pattern they want to use. The third occurs as the search finds two candidate objects for the first variable, *prototype*. Here the user is given the option of either selecting one of the candidates or of selecting *Restrict* and then identifying which candidates should be considered further and which should not be. The next dialog box represents the fifth variable, *client*, as denoted by the prompt “Restrict Class to contain the prototype” associated with that variable. Dialog boxes are not generated for variables identified by ALL clauses or where there is only one alternative found since asking the user in these cases is unnecessary. Note that in each case the previous selections are listed at the top of the dialog box to provide context for the current selection. The final dialog box indicates that a pattern instance has been found. At this point the user is given the option of naming and storing the pattern before proceeding to find additional instances of the pattern throughout the system.

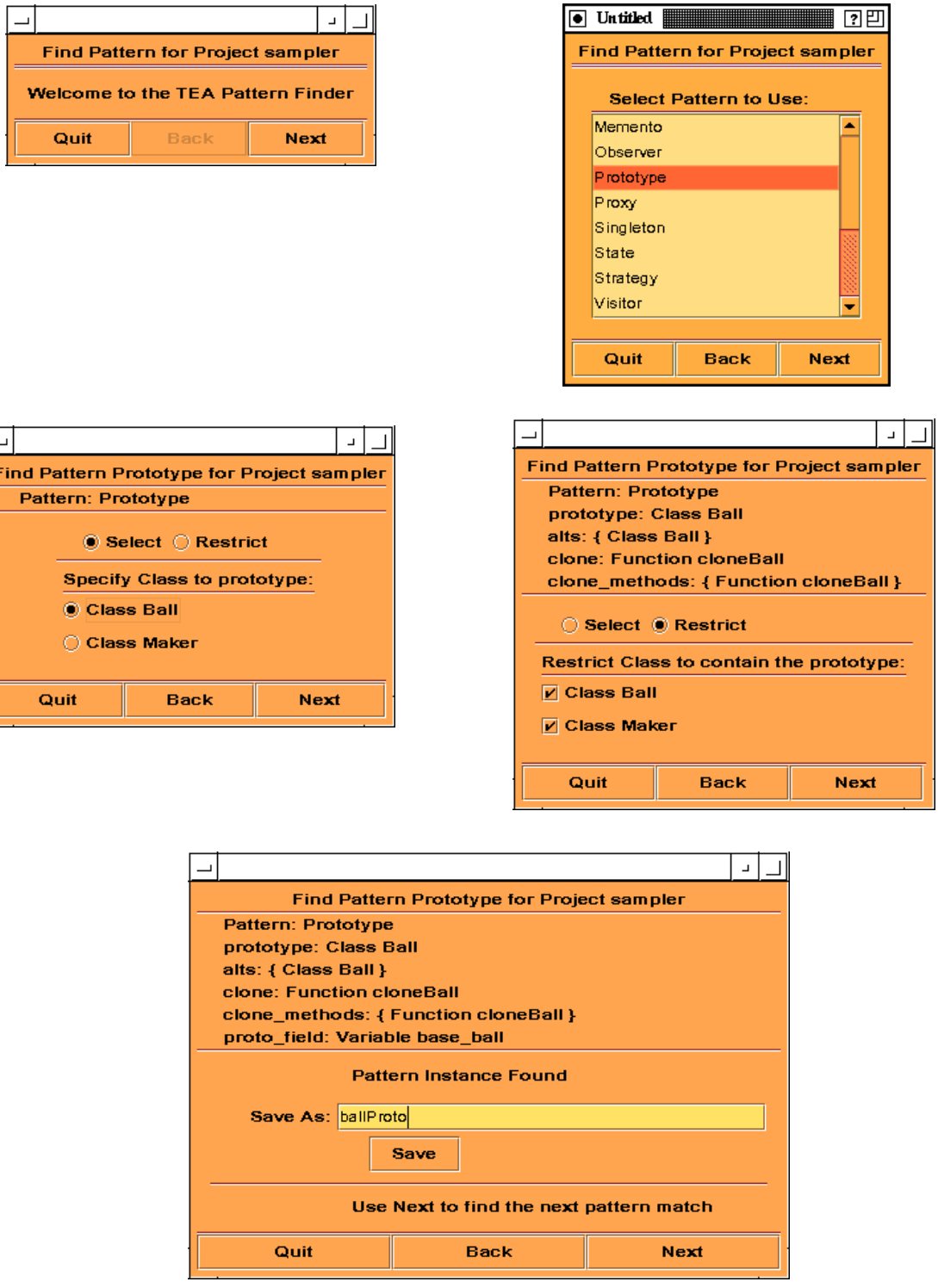

FIGURE 6. Dialog sequence for pattern finding.

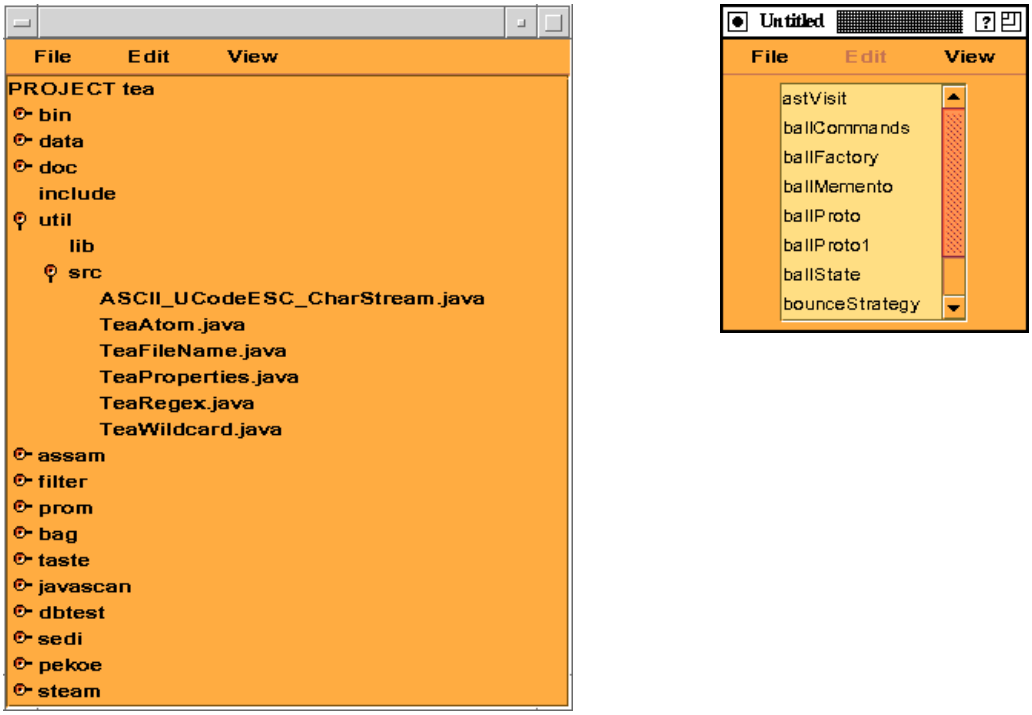

FIGURE 7. The project manager and pattern manager windows

5. Maintaining a Library of Pattern Instances

In order for patterns to be first class objects, one must keep track of the set of pattern instances in a system and let the user work with these instances. The TEA framework provides two tools for this purpose. The first lets the programmer define what is meant by a system or project while the second maintains the set of pattern instances for that system. The corresponding interfaces are shown in Figure 7.

The project manager shown on the left of Figure 7 lets the user define the system. This is done by specifying which directories and files should be included as part of the system. The manager lets the user specify both specific paths to include and patterns describing paths that should not be included.

The pattern manager shown on the right of Figure 7 keeps track of all the pattern instances that the user has saved. It lets the user delete, rename, and duplicate instances as needed. It allows patterns to be viewed either by name or by pattern type. Moreover, it provides a front end for finding and creating new patterns.

An additional facility the pattern manager provides is the ability to verify that an existing pattern instance is still reflected in the source. This can be done either for all recorded instances or for an individual instance. The underlying system handles this in a manner similar to a search for a pattern instance. However, in this case the old instance values are used to drive the search. The user is provided with the initial splash panel. If all goes well, a dialog is displayed to indicate that the pattern was verified. If the instance is no longer consistent with the source or violates a CHECK

```
DEFINE
   clone “Method used to clone the prototype” : Method | clone.baseClass ==
                 prototype;
       REQUEST  class = prototype;
                name : String;
                rettype = prototype.type.name;
       END

   proto_field “Field to contain the prototype” : Member |
                               proto_field.type.definition == prototype AND
                                proto_field.baseClass == client;
       REQUEST  class = client;
                name : String;
       END
```

FIGURE 8. Variable modifications for element definitions.

clause, appropriate dialog boxes are put up for the user to change the search criteria in order to update the pattern instance.

6. Creating Pattern Instances

To make patterns first class objects in a programming environment, we felt it was essential that the user be able to edit the source code by creating and editing pattern instances. This involved three additions to the way we find and manage patterns. The first involves augmenting the pattern definition language so that new program elements can be defined. The second involves adding a section to the pattern definition describing what code should be generated for any new pattern element. The third involves modifying the search process to let the user specify new program elements at appropriate places.

Creating a new instance of a pattern involves doing a search where GIVEN elements are found as before but where the user has the option of creating a new program element for a DEFINE clause. In order to accomplish this during the search, the system has to know the properties of the new program element. Some of the information that is required here can be derived automatically. For example, knowing that the element is of type *Class* indicates that a definition element for a class should be built. Other properties, such as the name, the superclass, the return type for a method, or the data type for a field may or may not be deducible from the underlying constraints. Moreover, how the objects are created is language-dependent. For example, creating a new class in Java involves not only building the class element, but also building corresponding type, scope and parent elements.

To accommodate all this, we introduced a mechanism in the search whereby a program element corresponding to any variable being defined can be created. We added syntax to the definition language to specify the properties of the new element. These properties can either be defined from existing variable values or interactively. The modified syntax for the variables *clone* and *proto_field* of the Prototype pattern of Figure 2 is shown in Figure 8. If a new element needs to be created, the system will first determine the type of the element from the variable type. Then, using

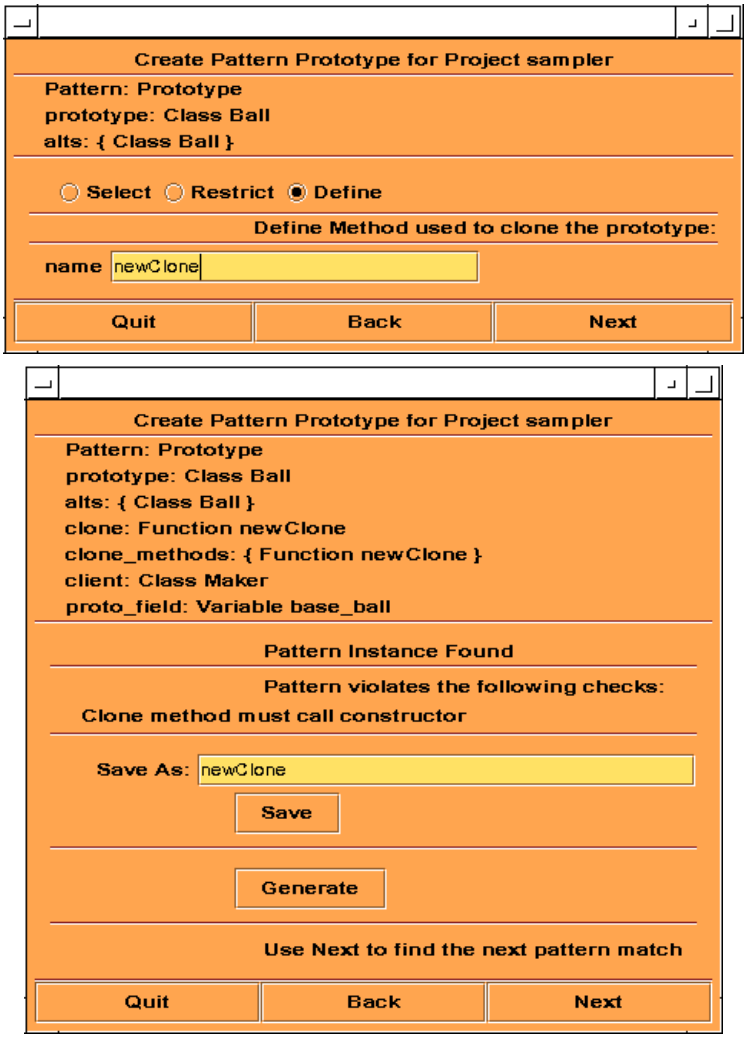

FIGURE 9. Some of the dialogs from pattern creation.

knowledge of the underlying programming language from the system and the pattern description, it will invoke an appropriate specialized routine to build that element and any other related elements. This routine uses the parameters defined in the request clause to define the element as completely as possible. A relatively complete definition is generally needed if the element is to be used later in the search procedure.

To go along with this clause, the search user interface detects when the user is creating a pattern and the current variable is one that can be defined. In this case the corresponding dialog box provides the user with the option of defining a new variable. If the user selects this option, then the dialog prompts for the values defined in the request field. An example of this for the Prototype pattern is shown at the top of Figure 9.

The dialog box at the bottom of Figure 9 shows what is presented to the user when a pattern has been newly defined. It differs from the previous conclusion dialog (the last dialog in Figure 6) in two ways. First, it includes a warning to the user that the

```
GENERATE WHEN (Language=java)

   FOREACH m IN clone_methods
      REQUEST
         protection : String = “private”;
      LOCATE BOTTOMOF m.baseClass;
      YIELD


$protection$ $prototype.type.name$ $m.name$() {
   try {
      return ($prototype.type.name$) clone();
    }
    catch (CloneNotSupportedException e) { return null; }
}
.

   FOR proto_field
      REQUEST
         protection : String = “private”;
      LOCATE TOPOF client;
      YIELD
$protection$ $prototype.name$ $proto_field.name$;
.

END
```

FIGURE 10. Generation specification for the Prototype pattern.

new clone method does not yet call the constructor. Second, it offers the user the option of generating code for the pattern instance. This generation is directed by an additional section of the pattern definition that specifies what the code should look like for each definable pattern element of the instance. A sample of such a specification for the Prototype pattern is shown in Figure 10.

The specification language allows multiple generation sections. Each such generation section starts with a WHEN clause. The conditions in this clause currently can include the user or the language. This lets it be used to generate different code for different languages or to change the code format for different users.

Each generation section next specifies how to generate the individual program elements that the pattern may define. In the case of a set variable, the language defines what to do for each new element of that set. The specification identifies any additional parameters that should be requested from the user as well as the default location where the code for the element should be inserted. This location can either be a line in a file or it can be relative (before, after, top or bottom) to another program element. The final part of the specification is the string that is to be generated for the element. This string includes expressions between dollar signs as well as straight text. The expressions can include both the pattern variables and the requested values.

The system provides a separate series of dialogs to go along with this language to ensure that the user provides the appropriate information, to give the user additional flexibility as to locations, and to ensure that the user is aware of what code is being modified as each element in turn is added. A sampling of these is shown in

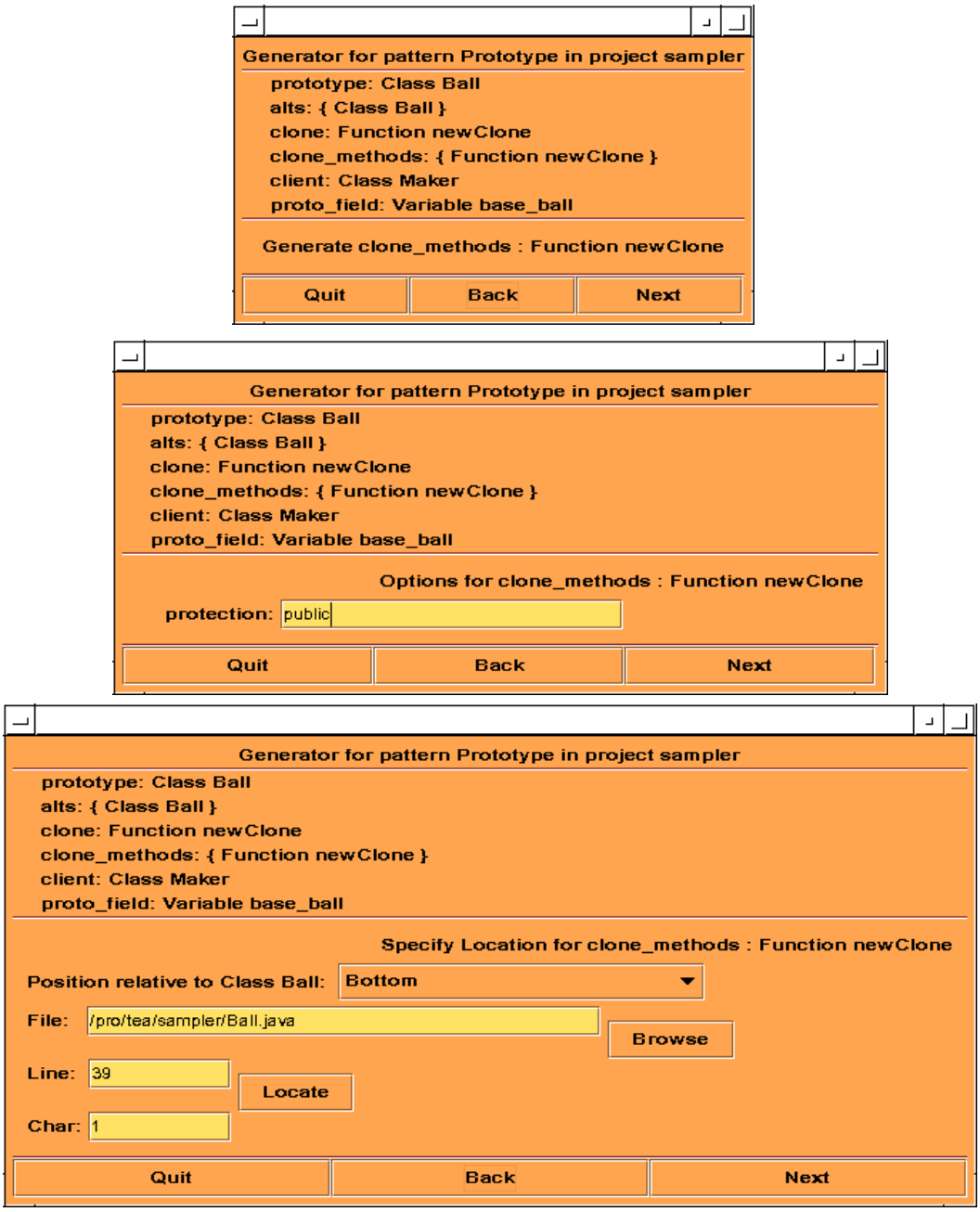

FIGURE 11. Dialog boxes used for generating pattern code.

Figure 11. They include prompts to keep the user informed, dialogs asking the user to supply the requested values, and a dialog box that lets the user specify the location for generation. The latter also allows the user to select a new file and to select a location in that file by picking the position in a read-only view of the file.

The system collects the information for all new elements of the pattern instance and then, after the user has approved all the changes, will actually make the modifications to the appropriate files throughout the system, thereby creating the instance. Making batch changes in this manner ensures that the system remains in a stable state and lets the system use the existing database to compute locations and other

information. If elements were generated one at a time, the database would need to be updated after each modification.

7. Editing Pattern Instances

In addition to being able to find, verify, and create pattern instances, we wanted programmers to be able to edit the source code through patterns. This meant that we wanted programmers to be able to modify the existing pattern instances in the system and to have the code modified accordingly when they do so. The first problem we faced here was determining what it means to edit a pattern instance.

The simplest means of editing a pattern instance is to modify the set of program elements that it contains. For example, a Prototype pattern instance might be modified by changing the field that holds the prototype or by adding or removing a concrete subclass of the class being prototyped. These changes are the structural ones that let the set of patterns better track the evolving system.

Our pattern tools handle this type of editing as a cross between defining a new pattern and verifying an existing one. They step through the verification process, looking at each variable in turn and computing the initial value of that variable based on the saved pattern instance. However, instead of just accepting that value, they put up essentially the same dialog box as in the creation case, thereby letting the user modify the selection of elements for that variable of the pattern either by restricting the set or by defining new elements. Because the system actually checks the pattern instance both against the original value and using the query associated with the variable, this approach is useful both for modifying the code to fit a modified pattern instance and for changing the pattern instance to fit modified code. The editor will again allow the user to generate code for any new pattern elements that are created here.

There are other ways, however, that the programmer might want to edit patterns. They might, for example, want to edit the properties associated with the patterns. Changing the mapping definition for a Visitor pattern could be used, for example, to change the names of the visitation methods throughout the system. Changing the user-specified properties used for generation could be used to modify the code that was generated originally for the pattern. It is also possible to have pattern variables that range over values other than program elements which are used either in selecting other program elements or in generating code. The user might want to modify these and then have the code modified accordingly. In cases where there are multiple variants of a given pattern, the user might want to change a particular instances from one variant to another, for example taking a prototype pattern and moving changing how the prototype is stored.

In many of these cases it is unclear as to what the edits to the code should be. For example, if the user changes the mapping of a Visitor pattern instance, the system must change the class name of the visitor or the names of all the visit methods. If it does the latter, it has to account for potential name conflicts. Similarly, if the user decides to move the prototype from a data field to a hash table, should the original data field still remain or should it be removed and all access to it replaced with

access to the hash table. In general, the problems that arise here are similar to the view-update problem for databases: various different modifications of the code can be used to achieve a modification in the pattern instance, and it is inherently ambiguous which should be used.

We are attempting to address these problems in our on-going work. We are currently extending our definition of patterns so that the generation specification can be used as a pattern for generated and user code so that any modifications or specializations made by the user to the code can be maintained as the code is modified. We are working on incorporating high-level editing operations such as changing a name consistently throughout the system. We are working on developing appropriate language-specific techniques for mapping changes in the internal representation of program elements to appropriate changes in the code. For example, the system needs to understand what has to be done to the code if the protection fields of a program element are modified. As these underlying mechanisms fall into place, we will incorporate them into the pattern editing facilities.

8. Related Work

Even though patterns are a relatively new concept, there have been quite a few efforts at deriving tools to support them. However, none of these provide the full range of support nor the integration between patterns and source code that our tools offer.

In order to provide tool support, all of the efforts have had to come up with some formal description of a design pattern. They take a similar approach in that a pattern is derived from a query of a design space. They differ substantially in how such a query is defined. Banisya writes code to represent each pattern [2]. Kim and Benner represent a pattern using a representation scheme similar to AI knowledge frames [14]. Both Meijers [16] and Gruijs [12] use fragments, essentially a set of objects that represent the pattern and that are matched against the code. Seemann and von Gudenberg use a graph-grammar-based query language over their own database for finding patterns [22]. The databases used in these cases are limited to structural information (i.e. what one finds in a UML class diagram such as Figure 1 without any code fragments. They do not utilize the semantic information that is necessary for a deeper understanding of patterns and to eliminate numerous false hits for some of the simpler patterns like Bridge. Moreover, these representations are limited to identifying or creating patterns and do not handle the full range of pattern operations that we have targeted. Finally, a somewhat related effort by Mikkonen uses a formal semantics to manipulate rather than identify and generate patterns [18].

The tools that have been developed for pattern manipulation typically fall into three categories. The first are tools for selecting a design pattern and then generating the skeletal code needed to support this [5]. The second are tools that attempt to find instances of design patterns in existing code [2]. The third and more sophisticated class of tools let design patterns be specified and compared to existing code, noting when a previously identified pattern is violated by changes in the code [14,23]. In

addition, efforts at Utrecht University [12,16] attempt to both let the programmer create a new instance of a pattern or identify and store existing instances. The mechanisms we provide should support all these tasks in a more generic and complete manner while letting the developer actually work in terms of the patterns.

Our work is also based on significant work in related areas. There has also been significant work on high-level program editing. This includes semantic editors such as that done at Wisconsin [31], Water's Programmers Apprentice [29], Pan from Berkeley [1], the semantics-based restructuring tools of Griswold, et al [4,11], and the program refactoring tools from Illinois [21], The latter has been used to perform behavior-preserving pattern-based edits.

9. Conclusions and Experience

The PEKOE system was implemented to demonstrate that tools could be developed to let the programmer work in terms of patterns over the full lifecycle of software development. Even in its current restricted prototype form it shows that patterns can be identified, created, verified, and edited in conjunction with existing code. Moreover, we have been able to use our pattern definition language to define almost all of the patterns in the standard text [9]. (The exceptions are patterns like interpreter that require too much semantic knowledge to be recognized.) The system is written in Java and, while much of it is language-independent, the current database framework only supports Java.

There are two principal contributions of this work. The first is the flexible query-based definition of a design pattern that is suitable for a variety of activities. This includes the database framework provided by TEA which incorporates the structural and semantic information needed to identify patterns, the pattern-element oriented OQL-based query language on top of this database, the request definitions to let new pattern elements be created, and the generation specifications for generating all or parts of a pattern.

The actual language that is used for defining patterns is our strawman implementation of such a query-based definition. While we have attempted to make it as simple and easy to use as possible, we are also looking into other approaches that would provide a different user interface to essentially the same semantic information. These include approaches that are more closely tied to the original source languages, approaches that make use of an interactive visual language, and approaches that work from examples.

The second contribution is the tool-based framework for using patterns throughout program development. This framework consists of the pattern manager that maintains the library of patterns, along with a consistent set of tools for finding, verifying, creating, maintaining, and editing patterns along with the source code.

In order to demonstrate the desirability of working in terms of patterns, we are currently working on integrating the pattern-based tools into a Java development environment. This will provide a user community and a basis for user studies of the effectiveness of the tool. The obvious question in such studies is if and by how much

tools such as PEKOE will improve programmer productivity. While the actual answer must await the result of the studies, we feel that the tool will help in several ways. First it will eliminate the drudgery involved in implementing many of the patterns, for example, the need to create multiple methods in multiple locations. Second, it will provide for higher-quality systems by ensuring that patterns developed during the design are maintained as the system evolves. Finally, it will encourage programmers to think about and work on their systems at a higher level, thus allowing larger systems to be built.

We are also exploring several avenues of related research. We are developing generic code formatting facilities so that the generated code looks good to the programmer. We are developing the higher-level editing facilities that we noted are needed for more general pattern editing. We are working on incorporating a behavior specification language into the pattern definition so that we can do dynamic checking of pattern behavior ([23] does this to a limited extent already). We are planning a front end to guide the programmer to the right pattern when a design issue arises. We are thinking about a more flexible query language that would allow some variation in the patterns. This could be tied to a graphical front end to let the user define patterns more easily. Finally, we are attempting to generalize from our experience with design patterns to more general abstractions. Here we want to look at visual design languages, domain-specific frameworks, architectural patterns, and semantic editing as technologies that can be implemented using a framework similar to that of PEKOE.

10. References

1. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, "The Pan language-based editing system for integrated development environments," *ACM Software Engineering Notes* Vol. **15**(6) pp. 77-93 (December 1990).

2. Jagdish Bansiya, "Automatic design-pattern identification," *Dr. Dobbs' Journal*, pp. 20-28 (June 1998).

3. Jan Bosch, "Design patterns as language constructs," in *Language Support for Design Patterns and Frameworks*, ed. S. Mitchell,Springer-Verlag (1997).

4. Robert W. Bowdidge and William G. Griswold, "Supporting the restructuring of data abstractinos through manipulation of a program visualization," *ACM Trans. on Software Engineering and Methodology* Vol. **7**(2) pp. 109-157 (April 1998).

5. Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal* Vol. **35**(2)(1996).

6. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *A System of Patterns*, John Wiley and Sons (1996).

7. R. G. G. Cattell and Douglas K. Barry, *The Object Database Standard*: *ODMG 2.0*, Morgan Kaufmann (May, 1997).

8. Kate Ehrlich and E. Soloway, “An empirical investigation of the tacit plan knowledge in programming,” in *Human Factors in Computing Systems*, ed. M. L. Schneider,Ablex Inc. (1982).

9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).

10. Joseph Gil and David H. Lorenz, “Design patterns vs. language design,” in *Language Support for Design Patterns and Frameworks*, ed. S. Mitchell,Springer-Verlag (1997).

11. William G. Griswold, Morison I. Chen, Robert W. Bowdidge, and J. David Morgenthaler, “Tool support for planning the restructuring of data absractions in large systems,” *Software Engineering Notes* Vol. **21**(6) pp. 33-45 (November 1996).

12. Dennis Gruijs, “A framework of concepts for representing object-oriented design and design patterns in the context of tool support,” Dept. of Computer Science INF-SCR-97-28, Utrecht University (August 1998).

13. Michael Karasick, “The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler,” *Software Engineering Notes* Vol. **23**(6) pp. 131-142 (November 1998).

14. Jung J. Kim and Kevin M. Benner, “An experience using design patterns: lessons learned and tool support,” *Theory and Practice of Object Systems* Vol. **2**(1) pp. 61-74 (1996).

15. Shriram Krishnamurthi, Yan-David Erlich, and Matthias Felleisen, “Patterns as language extensions,” Rice University Technical Report (October 1998).

16. Marco Meijers, “Tool support for object-oriented design patterns,” Dept. of Computer Science INF-SCR-96-28, Utrecht University (August 1996).

17. Scott Meyers, “Difficulties in integrating multiview development systems,” *IEEE Software* Vol. **8**(1) pp. 50-57 (January 1991).

18. Tommi Mikkonen, “Formalizing design patterns,” *Proc. 20th Intl. Conf. on Software Engineering*, pp. 115-124 (April 1998).

19. Lee R. Nackman, “An overview of Montana,” *IBM Research*, (1996).

20. Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994).

21. Don Roberts, John Brant, and Ralph Johnson, “A refactoring tool for Smalltalk,” Dept. of Computer Science, U. of Illinois at Urbana-Champaign (1997).

22. Jochen Seemann and Jurgen Wolff von Gudenberg, “Pattern-based design recovery of Java software,” *Software Engineering Notes* Vol. **23**(6) pp. 10-16 (November 1998).

23. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell, “Monitoring compliance of a software system with its high-level design models,” *Proc. 18th Intl. Conf. on Software Engineering*, pp. 387-396 (March 1996).

24. Mary Shaw and David Garlan, *Software Architecture*: *Perspectives on an Emerging Discipline*, Prentice-Hall (1996).

25. E. Soloway, K. Ehrlich, and J Bonar, “Cognitive strategies and looping constructs: an empirical study,” Research Report, Yale University Department of Computer Science (1982).

26. SunPro, *Browsing Source Code*. December 1993.

27. John Vlissides, *Pattern Hatching*: *Design Patterns Applied*, Addison-Wesley (1998).

28. R. Waters, “A Knowledge Based Program Editor,” *Proc*. *Seventh Annual IJCAI Conf*., (1981).

29. Richard C. Waters, “The programmer’s apprentice: knowledge-based program editing,” in *Interactive Programming Environments*, ed. D. R. Barstow, H. E. Shrobe and E. Sandewall,McGraw-Hill, New York (1984).

30. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).

31. Wuu Yang, Susan Horwitz, and Thomas Reps, “A program integration algorithm that accommodates semantics-preserving transformations,” *ACM Software Engineering Notes* Vol. **15**(6) pp. 133-143 (December 1990).