Tool Support for Object-Oriented Patterns

Gert Florijn, Marco Meijers, Pieter van Winsen

Utrecht University, Dept. of Computer Science
P.o. box 80.089, 3508 TB Utrecht, the Netherlands
E-mail: florijn@cs.ruu.nl

Abstract

A software (design) pattern describes a general solution for a recurring design problem. The solution is mostly described in terms of an abstract design structure expressed in design elements such as classes, methods and relationships (inheritance, associations).
This paper describes a prototype tool that supports working with design patterns when developing or maintaining object-oriented programs. The tool provides three integrated views on a program: the code (classes, methods, etc.), a design view (abstraction of the code plus additional information not in the code) and occurrences of design patterns in the program.
The tool assists developers using patterns in three ways:

- Generating program elements (e.g. classes, hierarchies) for a new instance of a pattern, taken from an extensible collection of "template" patterns
- Integrating pattern occurrences with the rest of the program by binding program elements to a role in a pattern (e.g. indicating that an existing class plays a particular role in a pattern instance)
- Checking whether occurrences of patterns still meet the invariants governing the patterns and repairing the program in case of problems

Through the use of an existing refactoring package, the tool supports the use of patterns both in forward engineering and in backwards engineering, i.e. documenting occurrences of patterns in existing programs and modifying the program to better reflect the pattern's structure.
The tool is implemented in Smalltalk and has been applied to identify pattern occurrences in several non-trivial (Smalltalk) applications and to reorganize these subsequently.

1. Introduction

Over the past few years, design patterns have become a hot topic in the object-oriented community. A (design) pattern describes a general solution for a recurring design problem. The solution is described in a standard format (the pattern format) that consists of a generic design structure for the solution (expressed in some terminology of design elements) together with a textual description of the pattern, indicating, for example, when to use the solution or how to apply it in certain situations.

Though most patterns are not OO specific (i.e. they could be used in non-OO programs), the design structures are commonly expressed in object-oriented terminology, i.e. in terms of classes, interfaces, methods, attributes and relationships (inheritance, associations). This means that applying patterns in the development of object-oriented systems is - in principle - fairly straightforward, since the terminology can be mapped directly on language constructs.

Patterns offer several (potential) benefits when developing (OO) software. First, there is the issue of reuse. By applying a solution that has been developed and used before, we can avoid design work that would normally take place, especially the work invested in checking out solutions that do not suffice for the problem at hand. Using patterns also means that discussions are more focused on important decisions, such as "should we allow run-time variation of this behaviour?" or "where must the creation of these objects be localized". In a similar way, patterns make communication among developers more effective. By using pattern names we can avoid detailed discussions about why certain classes are organized and programmed in a certain way. Also, understanding a program written by others becomes easier when we know whether and where certain patterns are used.

Problems when using patterns

Using patterns when developing an OO program is not a trivial process. First, one has to identify the need for a certain pattern by recognizing a problem and choosing a particular solution. Once that is done, the pattern has to be integrated with the design/program that is already available. In general, this means that the design elements from the pattern description have to be mapped to and integrated with the design elements in the program. More specifically, a developer must decide which classes in a program will play the roles defined by the classes in the pattern description, which methods in the program classes will play the role of the methods defined in the pattern's classes, etc. Of course, this can also lead to situations where the design elements in a program play multiple roles, i.e. correspond to design elements in several patterns. For example, a class playing the role of "abstract factory" in the Abstract Factory pattern[1], could also play the role of the singleton class in the Singleton pattern.

Putting pattern occurrences in a design affects the overall organization of a program, i.e. which classes there are, how they are associated and how class-hierarchies are organized. However, the inverse is also true; the existing program structure influences how we apply a pattern. For example, many patterns define inheritance hierarchies. If we want to combine two of these patterns at the same place in our program (e.g. combine a composite with an observer), we may have to decide which of these two hierarchies is taken as leading and find a different solution for the integration of the other pattern.

---

[1] All pattern examples discussed in this paper are taken from [Gamma95]. The reader is assumed to be somewhat familiar with the patterns described there.

So, when applying a pattern, a developer must be aware of the overall organization of the program, and occasionally be prepared to reorganize it based on new insights. When this occurs, the developer must make sure that the semantics of the program are left untouched and that the pattern occurrences introduced earlier are still intact.

Finally, using a pattern somewhere in a program can impose constraints on the further development of the elements involved. If we, for example, have applied a Proxy pattern somewhere, we must ensure that the proxy class implements all operations defined in its super-class and delegates them - whenever necessary - to the object it represents. If, at a later point in time, operations are added to the superclass (possibly by another developer) the proxy class may have to be adapted also. However, if no precautions are taken (eg. through documentation), it is easy to forget this, and in some cases it may be hard to find the errors that are then introduced.

Tool support

The goal of our research is to explore how tools can make the use of patterns in OO software development easier. Our focus is not on the selection of suitable patterns for particular problems but on using patterns in the creation, reorganization and evolution of a design/program. Basically, we want to introduce patterns as first-class citizens in an integrated OO development environment.

Over the past year or so we have developed a first prototype of such an environment. It provides assistance for:

- generating program elements (e.g. classes, hierarchies) for a new instance of a pattern that is taken from an extensible collection of "template" patterns
- integrating pattern occurrences with the existing program by binding program elements to a role in a pattern (e.g. indicating that and existing class plays a particular role in a pattern instance)
- checking whether (modified) occurrences of patterns still meet the invariants governing the patterns and repairing the program in case of problems.

Through the incorporation of an existing refactoring package, the environment supports program reorganization operations and can also be used for "reverse engineering", i.e. documenting occurrences of patterns in existing programs and modifying the program to better reflect the pattern's structure. The environment is implemented in Smalltalk and has been applied to identify pattern occurrences in several non-trivial (Smalltalk) applications and to reorganize these subsequently.

About this paper

In the remainder of this paper we discuss the tool we have developed. Chapter 2 discusses some the key issues and requirements for pattern tool support. Chapter 3 gives an overall overview of our system, while chapter 4 discusses program and pattern representation. Chapter 5 illustrates some of the tools that are currently available. Finally, in chapter 6, we discuss some preliminary experiences and topics for current/future research.

2. Issues and requirements for pattern tool support

As mentioned above, the primary goal of our research is to develop a tool to support (OO) program development with patterns. The underlying idea is that design patterns could be regarded as a kind of coarse-grained building blocks for OO design. A tool that provides the ability to define, instantiate, interconnect, and rearrange these building blocks could well help in rapidly producing consistent, high-quality designs. This section discusses some of the fundamental issues and requirements that such a tool must address.

The development model

A key decision to make is which role patterns play in the development environment. We have identified two models that could be used. In the first approach, the developer only works on the level of patterns, i.e. a system is developed by gluing instances of patterns together into a design. For this design, the tool could generate skeleton code (similar to the model in [Budinsky96]) in some programming language. While this approach is very interesting, it depends on the availability of a large catalogue of patterns which covers all possible design problems that can occur. At this moment, however, such a catalogue does not exist. Another drawback of this model is that it makes it more difficult to do reverse engineering of OO applications, i.e. to take an existing program, identify or put patterns in it, and reorganize.

In the approach we have chosen for our tool, a developer works on a program on different levels of abstraction (i.e. patterns, design, code) within the same environment. The environment provides three integrated and mutually consistent views on the underlying program. Each view supports operations particular to its level of abstraction. For example, on the pattern level the developer can instantiate patterns from a repository, on the design level she can split classes into a hierarchy, while on the code level she can add the code to methods. This approach does not rely on the availability of a comprehensive catalogue. Also, we can take an existing program into the environment and reorganize it subsequently.

Of course, this approach implies that a broad collection of tools has to be available, ranging from normal programming tools like code editors via design tools to pattern level tools.

Representing patterns

The solution part of a pattern gives a (generic) design structure expressed in a vocabulary of design elements. In order to manipulate patterns, a system should be able to represent these elements and treat patterns as some organized composition of such elements.

The key question that must be addressed is: what are the elements needed to express design structures? Obviously, this includes basic OO constructs such as (abstract) classes, inheritance association/aggregation relations and (class) method and attribute declarations with particular access control. However, in existing pattern catalogues [Gamma95, Buschmann96] the pattern description codes more information.

Frequently, special graphical notations are used to highlight certain (dynamic) properties and relations like the fact that a particular class (or method) creates instances of another class. Also, a pattern can impose certain behavior on methods, e.g. the fact that a "notify" method in a subject must invoke the "update" method on all its associated observer objects in an Observer pattern.

In general, one can envisage that a pattern is expressed in a design vocabulary that is richer than that offered by programming languages or design notations. As an illustration, consider a hypothetical relation "co-located" placed between two classes, that indicates that objects of these two classes should reside on the same site in a distributed system. We can also imagine annotations for persistence or concurrency control. The conclusion therefore must be that a design environment should be open with respect to the design elements that can be expressed. Likewise, it should be possible to represent patterns as configurations of arbitrary design elements, not just of classes, relations and methods as is the case in the approach of Kim and Benner [Kim96].

Dealing with programming language variations

To implement a pattern in a particular programming language, we have to find a suitable mapping from the design elements in the pattern to programming language constructs. In some cases the mapping is straightforward, and the design structure can be mapped directly to a (skeleton) program. In other cases, however, patterns can be implemented in completely different ways depending on the available language constructs. A good example is an Adapter, which can be implemented efficiently with multiple inheritance. If that construct is not available, we must use other mechanisms like interfaces (in Java) or delegation.

One of our long-term goals is to make the environment independent of a particular programming language, e.g. by using language-specific strategies and idioms to map configurations of design elements to language constructs. For the short term we assume a direct mapping between design elements and language constructs. This means that patterns must to be defined in terms of the design elements that can be mapped directly to the programming language used. Currently, we use the Smalltalk object model as a reference point, which means that constructs like multiple inheritance are not used. We will return to this issue later on.

Instantiating and binding patterns

The pattern view of in the environment should allow the instantiation of predefined patterns from a repository and the binding of the design elements in the pattern to elements in the program. This binding is similar to the conformance declaration that specify how classes (partially) satisfy the rules of role specification in contracts [Helm90]. There are three different ways of instantiating and binding patterns:

- Top-down: given a pattern description, generate the necessary design elements in the program and bind them to the pattern. This is typically expressed by: "Give me an instance of the Observer pattern", after which the developer is given an initial

set of classes (with methods, relations, etc.) that follow the canonical structure of the Observer pattern.

- Bottom-up: given a number of elements in the program, bind them to a new instance of a pattern. This addresses situations such as: "These classes (and their methods) together reflect the Proxy pattern; let's turn it into a Proxy pattern." The difference with the top-down approach is that no new design elements are created in the program, instead existing elements are used.
- Mixed: this approach differs from bottom-up in that the program elements that only partly meet a pattern structure may be combined with newly generated elements that are generated. An example of this is: "This structure closely resembles the Composite pattern; turn it into a Composite instance." In this case, the structure will be completed with new design elements that were missing.

Program transformations and conformance checking

Since we do not assume that programs can be developed just by instantiating and composing design patterns, the environment must provide operations to edit the program on both the design level and program level. Thus, it must offer operations to add new classes, create new hierarchies, define associations, move method definitions; in short: to add, modify, move or delete arbitrary design elements.

Obviously, these operations can also affect the elements in the program that play a role in a pattern instance. For example, we could move the notify method of a subject class in an instance of the Observer pattern to a sub-class. Basically, such reorganizations are no problem. Patterns are not laws; they suggest a general solution that has to applied and specialized to the problem at hand. This also means that the design structure defined in a pattern is not the "one and only" configuration of elements that meet the pattern. Variations are possible, and sometimes even necessary, e.g. when the programming language used does not offer multiple inheritance.

On the other hand, some transformations and operations can cause situations where a program no longer meets the structural or semantic intentions of patterns that were put into it. For example, if we completely remove the notification of observers from the subject class, the program no longer meets the underlying contract of the Observer pattern. This also occurs if the developer adds a method to the common superclass of a Proxy pattern instance, and forgets to extend the proxy class with a new method that delegates to the real object. So, as pointed out by Kim and Brenner [Kim96]: "a design pattern must be used correctly" and a design tool should help the developer in doing just this, without enforcing him to a single solution.

To deal with such situations, the environment should be able to check whether the configuration of design elements in a program meet the requirements of the pattern(s) in which they play a role. In a sense this corresponds to conformance checking as defined for contracts [Helm90] and to the "pattern instantiation validation algorithm" in [Kim96]. If the program does not conform, the environment should offer the developer different strategies to deal with the situation, ranging from ignoring the problem to (semi-) automatic repair. Also, it may take quite some editing operations to

take the program from one "consistent" state to another, so a transaction model and undo facility would be useful.

Clearly, such program transformation operations, together with the ability to check pattern conformance, are crucial for reverse engineering an existing application.

Pattern specific operations

Besides editing operations on the program/design level, one could also imagine operations on the pattern level. One example could be the replacement of an entire pattern instance by an instance of another pattern that solves the same problem but in a different way. A slightly simpler situation occurs when a particular pattern can be extended in a certain dimension, but doing these extensions involves a lot of lower-level editing operations, such as adding a factory to an instance of the Abstract Factory pattern or adding a concrete builder to a Builder pattern instance. It would be convenient if each of these sub-elements of these patterns could be manipulated as a single entity and if the creation and removal of such sub-elements could be eased. In short, it should be possible to associate a set of pattern specific operations to perform tasks that are particular to that pattern.

Miscellaneous issues

The points addressed above are among the core issues that must be dealt with. However, there are many other, issues that must be considered. For example it should be easy to add new patterns to the environment. Ideally, a developer should be able to take a particular configuration of design elements and translate it - without too many difficulties - into a new pattern for reuse in other situations. Furthermore, it should be possible to save, export and reload programs while retaining the information that was added in the environment. This involves the notion of documentation, so that in exported program code or design drawings we can, for example, identify which elements play a role for particular pattern instance.

3. Tool architecture

In the previous section we have identified the key issues and requirements that a pattern-based design environment must address. In this section we discuss the overall architecture of the environment we have constructed and that addresses most of the issues raised in the previous. Subsequent sections go into details about the various parts.

Figure 1 illustrates the key components of the environment. The basis of the environment is the *fragment model* and the corresponding *fragment database*. A fragment represents a design element of a particular type (i.e. class, method, pattern, association, etc.). Fragments have roles that can contain references to other fragments. The program being manipulated is thus stored as a graph of interrelated fragments of different types. Fragments (of a particular type) can also have behavior associated with them, such a checking consistency constraints or implementing pattern-specific operations.

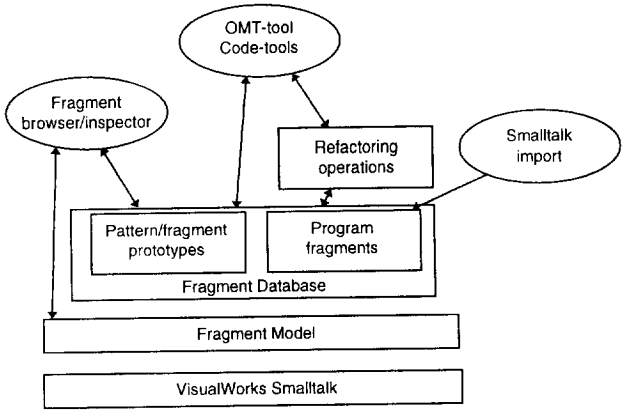

Figure 1: Key components in the environment

The pattern repository is a suite of prototypical fragment configurations that can be cloned into the program. Through editing operations on the fragment database, roles of pattern fragments can be bound to already existing fragments (of a particular type).

The fragment database can be inspected and manipulated directly with a couple of browsers. The *fragment browser* displays the available fragment graph and supports connecting fragments and invoking pattern/fragment specific operations and consistency checking. The *fragment inspector* shows details about one particular fragment and its type.

Besides these basic tools we have added higher-level tools. They support working on the three views on the program in the database. The central role is played by the *OMT-tool* which provides a (OMT) design level view of the program.

From the OMT tool the developer can perform edit operations (refactoring operations) on the program. She can also launch code-level tools, such as class browsers, which again work on fragments. In addition, the OMT tool provides more high-level support for instantiating and binding patterns into the program, and for viewing which parts of the design play a role in a particular pattern occurrence.

The development environment can be used to organize work through the creation of projects and diagrams. Projects are a means to have multiple programs loaded in the design environment. Within a project, the user can create diagrams, which are a subset of the design elements in the project. Basically a diagram is a sub-view of a program. They are mostly used for documenting and reorganizing existing programs. Finally, the tool supports import and export of Smalltalk programs into the fragment database.

4. The fragment model

Fragments play a pivotal role in our environment [Meijers96]. All design elements in the program - from pattern occurrences via classes and relations among them to methods and possibly even their code - are represented as (graphs of) fragments of particular types. Patterns (and pattern occurrences) are fragments (of a particular type)

and the repository of available patterns is just a collection of prototype fragment structures which can be cloned.

We have decided not to adopt a multi-level model for handling patterns as is done in other pattern-tools such as [Kim96]. They make a distinction between the user design level that contains the classes, their elements and relationships, and the pattern level to which user-level design elements are bound.

The main reasons for choosing a single level system were flexibility and simplicity. In our model it is easy to add new design elements (or fragment types, see below) at run-time and to use these in pattern definitions. So, we are not limited to a fixed number of design elements defined on the "pattern level" (see also chapter 2). In addition, in a single level system it is easier to "promote" existing design structures to a pattern, just by copying them to the template repository.

Fragment structure

A fragment represents a design element of a particular type (determined by a delegation parent holding shared data and behavior) that holds particular information (e.g. the name of class in case of a "class" fragment) and behavior (in code slots) and that has roles that contain references that point to other fragments (e.g. the methods of the class).

The working of the fragment model works is best illustrated by an example. Figure 2 shows an instance of the Observer pattern together with a corresponding fragment structure[2]. Note that because the system is based on prototypes and cloning, the given pattern structure could stand for a pattern instance as well as for a prototype.

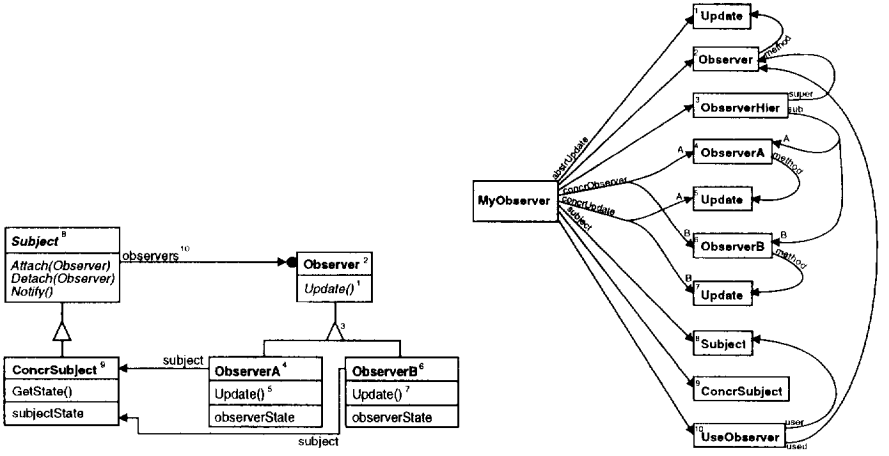

Figure 2: Observer instance and corresponding fragment structure

[2] Note that this fragment structure is not complete. Only the numbered items from the class structure have been depicted.

Fragments are connected by *roles* that are indicated by labeled arrows. A role is a typed container owned by a particular fragment. The fragments that are referenced by a role all provide a similar function to the role owner. A fragment that fulfills a role for another fragment is said to be an *actor* of that role. Note that a fragment may be actor and role owner at the same time. In our example, ObserverA fulfills the role "sub" for ObserverHier as well as the role "concreteObserver" for MyObserver, and in addition it is owner of a "method" role.

Roles are typed in order to constrain the actors of the role. For example, the Method role of a Class fragment can only be fulfilled by fragments of type Method. Additional constraints may be imposed on roles by defining the minimum and maximum cardinality to limit the number of actors linked to the role. In our example, at any time only one abstract observer class is allowed. This is expressed by setting both the minimum and maximum cardinality of MyObserver's abstractObserver role to one. The different actors of a role are distinguished by their actor IDs that uniquely identify them within the role. The A's and B's from the sample fragment structure denote the actor IDs.

Each pattern has one *root fragment*. In our example, MyObserver is the root fragment. Its main function is to hold all fragments that are relevant to the pattern instance. All the relevant fragments play a particular role for the pattern instance, otherwise it would contradict the fact that they are relevant to the pattern. To reflect this idea, all these fragments are linked to the root fragment by roles.

Root fragments are just like any other fragment. Therefore, they can also fulfill roles for other fragments. Root fragments serve as a sort of placeholders for pattern instances. The Observer fragment that is called MyObserver for example stands for an instance of the Observer pattern. All Observer fragments will descend from a single fragment called ObserverBehavior from which they inherit common behavior that is specific for the Observer pattern.

It is important to note that even relationships can be represented by fragments. For example, to model the inheritance relationship between classes we can use a separate Hierarchy fragment with two roles for the superclass and subclasses respectively (as is done in figure 2). An alternative way to model this is providing a subclass role to fragments of type Class, so the subclasses are directly connected to the superclasses. It is in fact this latter approach that we have used in the current modeling of patterns.

The example shown here contains only a single pattern instance. In practice such instances could overlap in arbitrary ways, e.g. when an Observer instance is combined with a Singleton pattern applied to one of its classes. That class would simply fulfill roles for the root fragments of both an Observer and a Singleton instance. In addition, both root fragments would fulfill the design role of the global root fragment. So our current model allows overlapping pattern instances almost trivially.

Implementation model

Technically speaking, fragments are objects similar to those found in the Self programming language. Fragments have an identity, a number of slots and a pointer to a parent object to which messages that cannot be handled in the fragment itself are delegated. Fragments can share data or behavior by referring to the same delegation parent but can also override behavior locally whenever this is necessary.

Fragment slots can hold primitive values (Smalltalk objects), code or references to other fragments[3]. The code slots are Smalltalk block-closures that are evaluated with a context of the receiving fragment, the handling fragment and the parent pointer. The data slots in a fragment are used to record information about the fragment, such as the name of a class or a method, the (uninterpreted) code of a method or the type of an attribute.

Whenever we talk about a fragment "of a particular type" we actually refer to a fragment object with particular behavior defined in a particular delegation parent. Thus, a fragment is a "method fragment" if it delegates (directly or indirectly) to the fragment (called MethodBehavior) that holds the common data and behavior for methods. In a similar way, there is a "type" fragment for each pattern defined in the system, such as Observer. It holds the "intent" and the pattern specific operations for all observer patterns in the program (see figure 3).

System structure

The complete fragment database is organized similarly as a pattern, as can be seen from figure 3. Note that this picture is not complete, but is only used to illustrate the overall structure of the system.

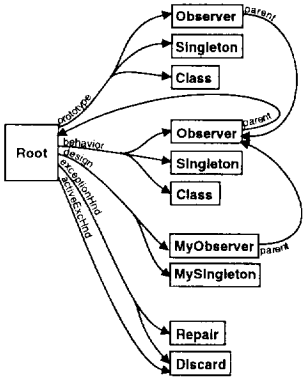

Figure 3: Fragment system structure

Like any other pattern, the system structure has a single root fragment to collect all fragments that are relevant to the system. We will subsequently refer to the root

---

[3] Roles are implemented as a particular kind of fragment object.

fragment of the system as the *global root*. All the relevant patterns are linked to the root fragment by roles.

The *prototype* role refers to the root fragments of all prototypes (of patterns and other fragments). The *behavior* role collects all fragments that define shared behavior for fragments of a particular type. For example, it holds the ObserverBehavior fragment contains all information and behavior that is shared by all Observer instances. All design elements in the program are registered in the *design* role. So for example, creating an instance of the Observer pattern and applying the Singleton pattern to the AbstractSubject class of it will result in the binding of both pattern instances to the design role. The remaining two system roles deal with exception handling. All handlers that are present in the system are registered by the *exceptionHnd* role. The exception handler that is currently active, which is at most one, is designated by the *activeExcHnd* role. More on validation and exception handling follows below.

The global root fragment performs yet another function besides collecting the relevant fragments. We have already mentioned that each fragment has a delegation parent. So a fragment's parent also has a parent, and so on. This delegation chain eventually ends with the system's root fragment as can be seen from figure 3. In other words, the global root fragment acts as the indirect ancestor of all fragments. The global root therefore provides default behavior with respect to cloning and other operations on fragments.

Operations and transactions

Operations on the fragment database are implemented by behavior slots in the fragments. Default behavior is provided to clone fragments and to bind/unbind fragments to/from roles. Pattern specific operations are defined in the shared behavior fragments for these patterns.

For Abstract Factory patterns, for example, there are four pattern-specific operations, i.e. adding and removing factories and products. An operation like addProduct involves creating and linking multiple fragments (a hierarchy with an abstract base class and a number of concrete product classes, creation methods in the various factories). These operations are programmed in code slots and use method calls to clone instances of existing fragments and bind them to the right roles.

To let these composite operations be treated as one unit the tool provides a nested transaction model. Validation checking is postponed until a (sub-) transaction is completed. Transactions are also used to let the user set check points to which the system can roll back if necessary. The user can issue commands to begin a transaction, end a transaction, and undo a transaction.

Constraints and validation

Using the basic operations defined above the structure of the "user design", i.e. the program being developed, will gradually evolve. As mentioned in section 2, this can also mean that the initial fragment structure of design patterns (as given by the prototype) is modified. This can lead to situations where the program does no longer conform to the patterns put into it.

To handle this problem we have to deal with two issues. First, how do we specify what a valid pattern occurrence is? Gamma et al. [Gamma95] use a rather informal notation to describe these constraints. They are often suggested by naming conventions or textual annotations to the class diagrams. In order to let a tool automatically check the constraints, a more explicit mechanism must be invented to describe them. Second, we must decide how to handle constraints, or more precisely, constraint violations. Do we attempt making sure that only correct configurations can be constructed with the given representation or do we check the validity of a configuration after it has been created? We have adopted the second approach, because for certain constraints it is difficult to build them implicitly into the representation.

Our initial attempt to was to hard-code the checks into a code slot called "validate" attached to the shared behavior fragment of each pattern. Each pattern instance has a number of roles, and the validation procedure checks whether these roles are correctly fulfilled by fragments of the right type and whether these fragments are well interconnected (e.g. whether instance access method for a Singleton pattern instance is bound toa class method fragment that plays a class method role in the singleton class).

Note that we use a per-pattern type validation procedure instead of one general algorithm as used in [Kim96]. This is caused by the fact that we do not have a two-level system, but also because patterns can involve arbitrary design elements, and because we wanted to handle sharing constraints as mentioned above.

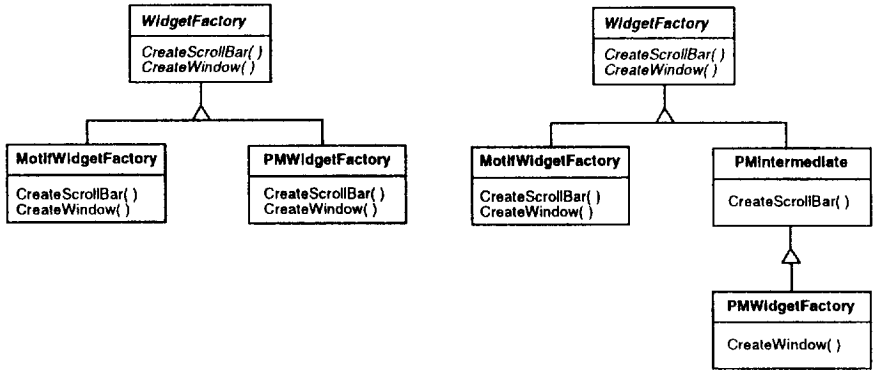

Figure 4: Example design variation

The structure checking based on role-types becomes problematic, however, when indirections are introduced, e.g. through inheritance or delegation. Consider the Abstract Factory pattern again. One of the constraints is that each concrete factory class must provide an implementation for the abstract factory class. This does not imply however that each concrete factory class must also be a *direct* subclass of the abstract factory class. The only restriction is that it must be a descendant, but it need not necessarily be a direct descendant (see figure 4). A similar situation appears when a particular class is supposed to provide a particular method. It is not necessary to require the method to be implemented by the class because it is valid to inherit the method from a superclass.

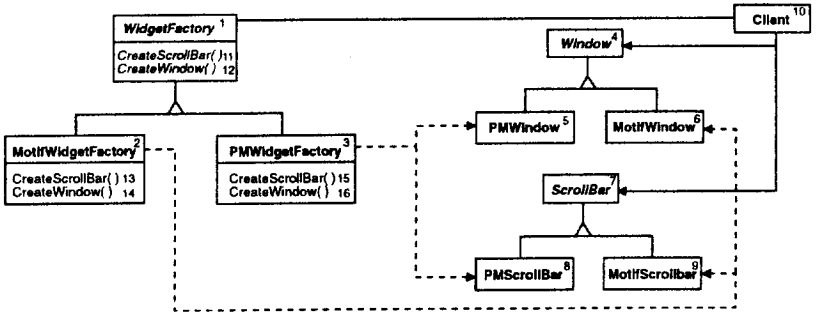

Figure 5: Abstract Factory pattern instance

A partial solution is to add such indirections to the constraint-checking, similarly to the use of the generalized path mentioned in [Kim96]. However, since our pattern structure can involve arbitrary design elements it was not easy to decide on a general approach as to whether (and which) kinds of indirections were allowed. As a consequence, the validation checking had to be modified constantly, augmenting the other drawback of procedural constraint coding: the code became difficult to write and maintain.

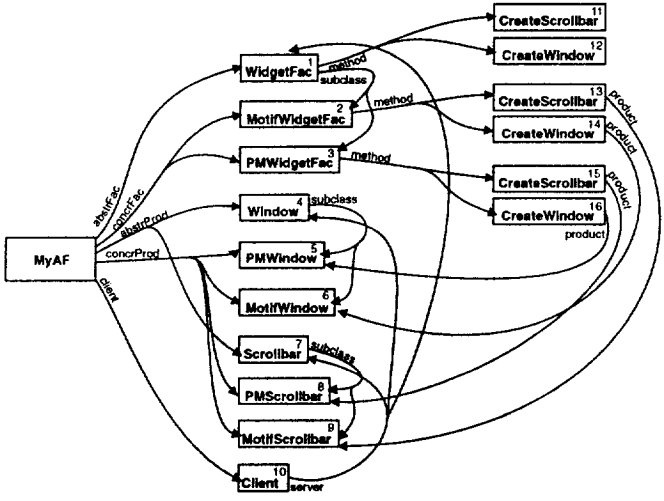

Figure 6: Fragment structure of the abstract factory instance in figure 5

In order to alleviate these problems we decided to use a declarative approach combined with a slightly different modeling of relations. As an illustration consider the abstract factory pattern instance shown in figure 5 with the fragment structure depicted in figure 6.

Constraints for patterns are now expressed as compositions of predicates defined on certain fragment types. The predicates are much like query operations; the most important ones are listed in table 1. For defining constraints, these predicates can be combined by boolean operators like and, or, not, plus the quantors forall, and exists.

Predicate	Description
aClass **provides:** aMethod	True if the class provides a concrete implementation for the method.
aClass **defines:** aMethod	True if the class at least provides a signature for the method; an implementation is not required.
aClass **implements:** anotherClass	True if the receiver provides an implementation for all methods defined in the argument class.
AMethod **creates:** aClass	True if the receiver method instantiates the argument class.
aMethod **uses:** anotherMethod	True if the receiver directly or indirectly calls the argument method
aClass **contains:** anotherClass	True if the argument class is a component of the receiver class

Table 1: Some fragment predicates

Now follows a possible specification of four Abstract Factory invariants based on the modeling of the Abstract Factory pattern given in figure 6:

$$\forall_{f \in concrFac} \; f \textbf{ implements: } abstrFac$$

$$\forall_{f \in concrFac} \; \forall_{p \in abstrProd} \; \exists_{c \in ConcrProd} \; \exists_{m \in (f \; provides)} \; c \textbf{ implements: } p \wedge m \textbf{ creates: } c \wedge abstrFac \textbf{ defines: } m$$

$$\forall_{p \in abstrProd} \; client \textbf{ uses: } p$$

$$client \textbf{ uses: } abstrFac$$

Note that for the sake of legibility, the constraints have been written in a formula-like format. In practice, they are written in Smalltalk, in a code-slot of a fragment; the first constraint then becomes:

```
_self concrFac  forall: [:f | f implements: _self abstrFac theActor]
```

Validation of a pattern fragment should take place if editing operations have (potentially) modified it. Validation must be performed on the instance to which the editing operation was applied and, in the case of role operations, it is the role owner that must be validated.

It might be necessary to propagate the validation signal further because relations might be fulfilled indirectly. The fragment that is in between in that case could well belong to an entirely different pattern instance. The only way to propagate the validation signal correctly seems to validate all the fragments to which the affected fragment is connected. This will finally end with the validation of the global Root fragment.

Exception handling

What happens when constraints are violated? When something wrong is encountered during the validation process, an exception is raised. An example of such an exception is:

```
typeErrorOnActor: aFragment ofRole: aRole
```

Exceptions are sent to the active exception handler. As can be seen from the system structure of figure 3 there are a number of exception handlers one of which is active at a particular moment. The system can easily be extended with additional exception handlers, because they are simply represented by fragments linked to the `exceptionHnd' role of the global root.

Each exception handler uses a different strategy for handling constraint violations. The handlers we have considered and tested are:

- Ignore: This handler permits everything, it will not bother the user with messages or other actions; it simply ignores any violation. Of course, the behavior of the system after ignoring a violation is at your own risk.
- Discard: This option prohibits faulty actions. On exception, an error message is displayed and the action that causes the exception is rolled back.
- Warn: A warning message is displayed and the user is asked for confirmation. It provides the opportunity to cancel the action or to ignore the warning.
- Repair: Tries to repair the defect. This appeared to be very useful in practice. More on this option follows.
- Choice: Gives the opportunity to choose interactively among different strategies. When an exception is raised, the user is informed about it and can choose to cancel (and hence roll back) the action, to ignore it, or to repair it. This option is useful if the user prefers full control over the actions that are taken on each individual exception.

The repairer is like an intelligent assistant. On a "MissingLink" exception (i.e. when a role is not adequately bound to other fragments), for example, it will simply add the link that is missing and add a new instance of the right fragment type. If an exception occurs the user will not even experience it as something wrong, instead it looks like the system is completing his actions. Note that this technique can make the definition of pattern specific operations like adding a product to an Abstract Factory pattern a lot simpler.

There are certainly cases in which there is no automatic fix for the problem. In that case, an error message is still presented to the user. Furthermore, the user can be asked to make a decision if there is more than one way to repair a defect. Furthermore, it is often the case that a defect can be repaired in both a constructive and a destructive manner. On a MissingLink exception for example a bind operation can be applied to add the missing link. Alternatively, the fragment that requires the link could be removed in its entirety. The first approach is constructive, while the latter is clearly

destructive. Both approaches could be useful in different situations. But when an exception occurs, what repair mode should be used? Constructive or destructive? In our opinion, it would be reasonable to link up with the type of action that caused the exception. If the nature of the action was constructive (e.g. the bind operation) then it is most desirable that the action is completed in a constructive way, so constructive repair must be used. On the other hand, if the user issues a destructive operation (e.g. destroy or unlink) then it would be highly annoying if the just removed link appears again because of the repair actions. So destructive repair is obviously preferred in that case.

5. Tools

This section discusses some of the tools that are available in the environment. Detailed descriptions can be found in [Meijers96] and [Winsen96].

Fragment browser

The fragment browser is depicted in figure 7[4]. The browser displays the fragments in a straightforward way. Fragments are depicted as boxes with labels in boldface. Because roles basically are fragments, namely container fragments, they are depicted by boxes as well. However, roles are labeled in italics whereas the other fragments are labeled in boldface. Roles are connected to the role owner by arcs and the same holds for the way actors are linked to roles.

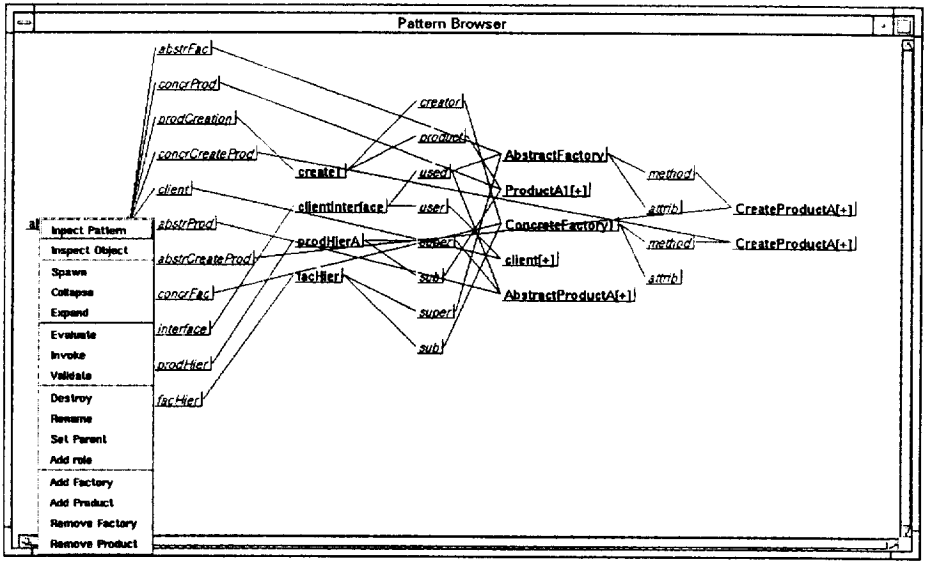

Figure 7: Fragment browser window

[4] This view was realized with the ObjectGraph package developed by Christopher Penney, Intelligent Systems Laboratory, Michigan State University.

The figure shows a browser window focused on an instance of the Abstract Factory pattern. Note that in this case the root fragment has almost entirely been covered by the popup menu.

The level of detail that is displayed in the browser can be controlled by collapsing and expanding nodes. In addition, one can spawn a separate browser on a subdesign rooted at a particular fragment in a separate window. Finally, the user can mark the nodes of a subdesign with a particular color.

What is displayed in figure 7 is the so called component view that shows how fragments are linked to other fragments by roles. There is yet another view mode however, namely the parent view, that reveals the delegation chain among fragments.

The fragment structure can be manipulated in the browser fully through menus. Standard menus provide a number of general operations, for example to change the view mode, to close the browser, and to control transactions. In addition, there is a context-sensitive menu that depends on the type of the fragment to which the cursor is pointing. Since menus are "inherited" through the delegation chain, the global root fragment provides a basic menu that is overridden or extended by descendants. Via this menu the user can perform actions such as inspect a pattern, expand, collapse or spawn a new browser on a subdesign, destroy and rename fragments, add roles, or invoke validation on a fragment. The context sensitive menu also allows the user to bind/unbind a fragment from a role.

The context-sensitive menus allow the invocation of pattern-specific operations. In the case of an abstract factory pattern this involves operations like adding a product or adding a factory. Also, one can add a new fragment to a particular role by selecting from a list of available fragment prototypes that meet the type expected by the role. In case of the *design* role of the root fragment, one can thus also add a pattern instance by selecting the appropriate name from the list. The complete fragment structure of the prototype is then cloned and added (with a supplied actor-id) to the design-role.

Adding pattern fragment copies together with rebinding allows top-down, bottom-up and mixed-mode pattern instantiation. As an example, consider applying the Singleton pattern to an existing class. First, we create a Singleton instance as described above. Then the class to which the Singleton pattern is to be applied, must be bound to the `theClass' role of the singleton's root fragment. This is done by issuing a Bind operation and selecting the same actor ID of the actor that was automatically provided, the latter will then automatically be replaced by the custom class. After that, invoking the validation operation on the singleton instance with the active exception handler set to "Repair" will takes care of moving the class method and class variable of the Singleton instance to the class designated as a singleton.

Fragment inspector

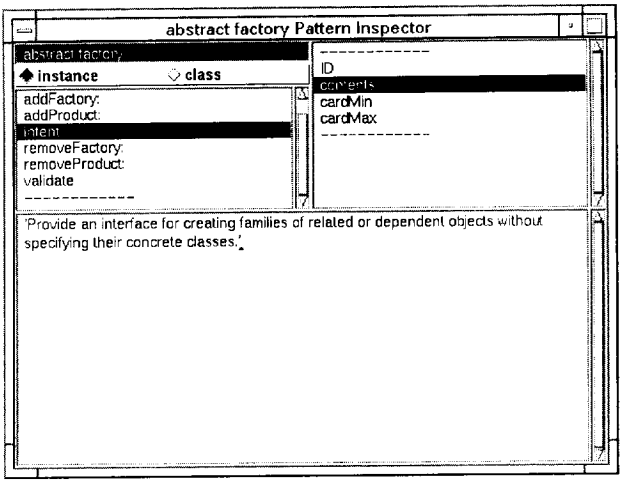

Figure 8: Fragment inspector window

Whereas the browser provides a global view on the fragment structure, the fragment inspector focuses on one particular fragment and reveals its details. Figure 8 shows a sample inspector window on the root fragment of an arbitrary Abstract Factory instance.

Like the standard Smalltalk browser[5] the pattern inspector can be set in one of two modes. In class mode the right subpane will list all properties of the fragment such as parent and type. On selection of a particular property, the corresponding value appears in the lower subpane. When put in instance mode, the right subpane lists the properties of a single slot that is selected from the left subpane. The corresponding value is again displayed in the lower subpane. In the sample figure the contents of the primitive slot named "intent"" is being displayed. In addition to being displayed, the value of any property can also be edited in the lower subpane.

Note that this means that code-slots can also be added and modified with the fragment inspector. Together with the binding operations in the fragment browser, this also offers the means to add new fragment types and thus new design patterns. The process is roughly as follows: First, we create a new fragment of an arbitrary type, give it a new name, add it to the behavior role of the root fragment and define particular behavior/data slots for it. Typically, for design pattern fragments, we would define a "validate" slot coded using the constraint-model defined earlier. Then we create a prototype fragment structure (by instantiating and linking the right fragments or by copying an existing structure) and let its root fragment delegate to the behavior fragment defined earlier (an operation provided in the browser). After that is done, the new pattern is available for use, and new instances can be added to the design.

---

[5] The fragment inspector is actually implemented by a subclass of the system browser.

The OMT Tool

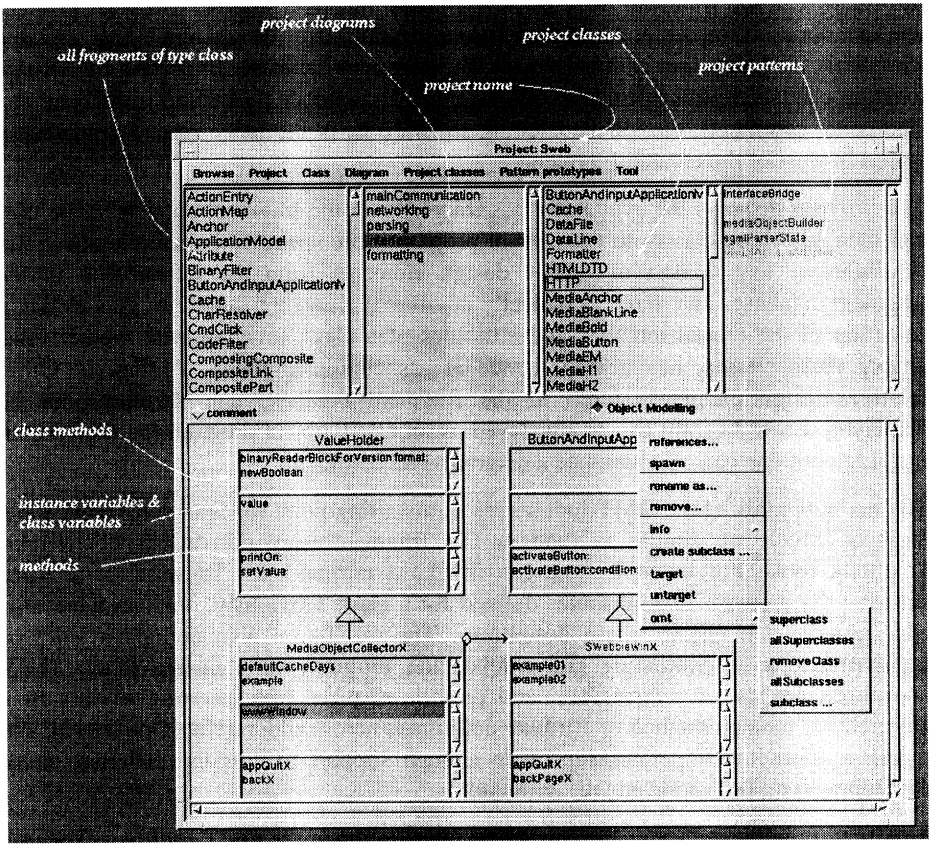

Figure 9: OMT-Tool main window

The fragment browser and inspector provide support for direct interaction with the fragment model and the fragment database. In principle, all the necessary functionality for manipulating the design and dealing with patterns is available through these tools. The OMT-tool is a first step towards a higher-level, and easier to use development environment working on the fragment database. The OMT-tool was primarily developed to support the identification of patterns in existing software and to reorganize these programs. However, new programs can also be developed with it.

The OMT tool is an extended version of the Smalltalk Refactoring Browser[6] developed at the University of Illinois, which on its turn is based on the work on design refactoring operations by William Opdyke [Opdyke92]. Obviously, the Refactory Browser has been "ported" to work on the fragment database, not the Smalltalk Image.

---

[6] See http://st-www.cs.uiuc.edu/users/droberts/Refactory.html

Figure 9 shows the main window of the OMT-tool. One of its key features is that the information in the fragment database is organized into user-defined *projects*. Projects can be focused on creating a new program but also on manipulating existing (Smalltalk) software that is imported into the fragment-database. Importing means that existing classes are transformed to class fragments, methods to method fragments, etc. The code, comments, etc. of the existing software is, in the latter case, automatically stored in slots of the created fragments.

The browser displays the available class-fragments in the fragment database. The user can then add these classes to the current project. More specifically, she creates views or *diagrams* that consist of certain design elements from the information in the fragment database. The diagram typically involves a view on a couple of classes, i.e. a selection of the (class) methods and attributes of a class, together with the relations among these classes, i.e. the inheritance or association/aggregation relations. Other relations/design elements stored in the fragment database can currently not be displayed. Each diagram can be displayed as an OMT object diagram (as depicted in figure 9), but one can also annotate a diagram with comments.

The main window of the OMT-tool serves as a starting point for launching other tools most of which are similar to (and in fact fragment-using derivations of) regular Smalltalk tools. For example, one can invoke a normal class browser on a class, showing the methods and attributes defined for a class. Obviously, one can there also add, remove and change these definitions. The OMT-tool also defines a suite of transformations to restructure and extends the program on a design level. These operations are called *refactorings*, and provide services like inserting a class in a hierarchy or moving method or attribute definitions across a hierarchy [Opdyke92]. Put together, these tools/operations provide enough support to edit the program under development on both code and design-level.

As a final point we consider the role of design patterns in the OMT-tool. As indicated in figure 9, a separate list of pattern occurrences in the current project is displayed in the main window. All these pattern occurrences are marked with a particular color. The other tools[7] annotate the program elements they display with markers in these colors if the program elements plays a role in a pattern. In this way, the developer can easily see which program elements were introduced as a consequence of applying a particular pattern. To find out which role a particular element plays, the user can pop-up an OMT-view of the pattern prototype in a separate window. After selecting the program element considered, the corresponding element in the pattern -view is highlighted.

Obviously the OMT-tool also supports instantiating patterns, using the fragment level implementation. Bottom-up definition is supported by adding a fragment structure for the pattern, and then using the OMT views of the pattern and the design to rebind the roles. In doing so, the user can check at any time whether all roles are already bound or not.

---

[7] Currently, this is only supported in the OMT view of the main OMT-tool window

6. Conclusions

This paper has described a prototype OO design environment that supports working with design patterns. The tool has been implemented in Visualworks Smalltalk. The implementation incorporates all the functionality described in the previous section. The system currently holds around 12 of the patterns defined in [Gamma95] and includes invariant-definitions for some of these. We are adding more pattern definitions as work continues and the need for them arises (e.g. in reorganizing an existing application).

Though we are still in the early stages, the experiences gained so far are quite good. The representation that was chosen (i.e. the fragment model) makes it easy to represent arbitrary program structures and design elements, including those that go beyond the regular OO constructs. It also means that we could handle patterns that incorporate less conventional design elements, such as fault-tolerance indications or timing constraints.

Until now we haven't encountered patterns that could not be represented in the system. However, most benefits occur when the structure of the pattern is the most prominent part, as is the case with patterns like Observer or Abstract Factory. In principle, this could also hold for patterns in which the behavior of methods is a key point (e.g. delegation). However this requires that the "code" of methods is represented as fragments. Experiments with this have not yet been done.

For patterns that focus more on methodical advice, such as Bridge, Mediator or - to some extent - Composite, support is inherently less advanced. For example, in a Composite pattern instance, a key issue is whether methods defined for item types are to be pushed-up to the level of the abstract base class for all item types. Such a decision cannot be made automatically. Only if the designer adds the method on the abstract base class, the system can - by validating and repairing constraints - add the necessary method fragments to all derived classes.

The prototype approach to represent and instantiate patterns works quite well in practice. Allowing design variations based on an initially consistent structure together with validation and (semi-) automatic repair seems a natural metaphor when designing/developing object-oriented programs. Obviously, capturing the "essence" of a pattern in an invariant requires is not always straightforward, and can be subject to discussion. Here too, the choice for a flexible system has its benefits. It is easy to represent multiple versions of the same pattern that differ only in the invariants. One could even adapt the invariants for a particular pattern occurrence.

Some of the metaphors used in the OMT-tool are appealing. Especially the ability to view which design elements are part of a particular pattern proved to be very useful when redocumenting the existing system.

The tool upto now has been mainly used in backwards engineering, i.e. to find patterns and reorganize a WWW-browser written in Smalltalk. This program with supporting software (after reorganization) involves around 150 classes stored in the fragment database. During reorganization several more or less obvious pattern occurrences were detected and documented (and subsequently made more explicit through refactorings)

such as State, Builder, Memento and Bridge. See [Winsen96] for more details on this work.

This experience has made us somewhat skeptical about the possibility to automatically detect predefined patterns in existing software. Pattern occurrences were often "degenerated" in that many conceptual roles did not exist as distinct program elements, but were cluttered onto a few, more complex ones. For example, the idea of abstract coupling found in many design patterns was not encountered often, especially not if only one concrete sub-class existed. In such cases, the roles were collapsed into single concrete class. In some cases, the use of (Smalltalk) language-specific techniques obscured the existence of a pattern. A good example of this was a State pattern occurrence in which the state dependent behaviour was not defined in separate classes (one for each state) but in the original class itself, by using method names for the state sensitive operations that were prepended with the name of the state. So, instead of delegating the operation "display" to a state-object by an expression like "`self currentState open`" the code for the methods looked like this:

```
self perform: (currentStateString, "open")
```

Automatically recognizing a real State pattern from such a case appears difficult.

A different question is whether *new* patterns could be detected in a large body of software, e.g. by looking for similarities in fragment structures of a large body of software. We have not investigated this issue yet, though it seems technically viable. Whether such an experiment would provide interesting results remains to be seen. A priori, a serious problem is that original intentions of the designer are not fully represented in program code. It would seem that the interesting design patterns thus also cannot be seen. Recovering these intentions is the human part of any re-engineering effort.

Open issues

The current environment, addresses most, but not all of our design requirements. Currently we are working on some extensions/new issues:

- The support for visualization of design elements in the OMT-tool should be extended. Especially non-conventional design elements (creation relation, etc.) should be represented. As a special case of this, we will consider a visualization of a program purely in terms of pattern instances.
- It is necessary to derive more design information from programs loaded into the system (e.g. associations, creation relations, etc.). Currently, we only show the information that can directly be derived from Smalltalk code without any analysis. Applying the environment for statically typed languages like C++ or Java will make gathering of this kind of information easier.
- More aspects of the program should be represented as fragments. In particular we are considering the "fragmentation" of method code, so that we can distinguish fragments like message-sends or instance creation. This opens the possibility to do behavioral conformance checking in pattern validation, i.e. the checking of causal

obligations as defined in [Helm90]. Also, we can then annotate parts of the code with indications that it plays a role in a particular pattern

- A related, but as yet unresolved, issue is to see how we could make the use of the environment less language dependent or even language independent. The general idea is to map a design with patterns automatically to constructs in a particular programming language. More in general, we would like to see how one pattern description could be mapped to multiple object models. The approach we are currently considering is to create a description of a language in relevant programming constructs (inheritance model, access control mechanisms, static vs. dynamic typing information, etc.) and to explore what a mapping function could look like. Specifying parameters that generate implementations as is done in [Budinsky96] will be considered.

- We are also studying "high-level" design refactorings that are based on certain design patterns. Basically, these operations transform the existing design by introducing and integrating a particular design pattern. To illustrate the idea: the designer identifies a particular class and indicates that he wants to have a proxy for it. The system then should instantiate the proxy pattern (with the hierarchy involved), derive the interface methods for the base class and generate the basic delegation code in the proxy.

References

[Gamma95] E. Gamma et.al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[Buschmann96] F. Buschmann, et. al. Pattern-Oriented Software Architecture - A System of Patterns, Wiley and Sons Ltd., 1996.

[Helm90] Richard Helm, Ian M. Holland and Dipayan Gangopadhyay, ``Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,'' pp.169-180 in Proceedings of the 1990 OOPSLA/ECOOP Conference, ed. Norman Meyrowitz (October 1990).

[Kim96] 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, No. 1, 1996, pp. 61-74.

[Budinsky96] F.J. Budinsky et.al., "Automatic code generation from design patterns", IBM Systems Journal, Vol 35, No. 2, 1996.

[Meijers96] Marco Meijers, Tool Support for Object-Oriented Design Patterns, Master's Thesis, Utrecht University, CS Dept, INF-SCR-96-28, August 1996.

[Winsen96] Pieter van Winsen, (Re)engineering with Object-Oriented Design Patterns, Master's Thesis, Utrecht University, CS Dept, INF-SCR-96-43, November1996.

[Opdyke92] William F. Opdyke, Refactoring object-oriented frameworks, University of Illinois, Urbana Champaign, 1992.