Type-Check Elimination: Two Object-Oriented Reengineering Patterns

Stéphane Ducasse, Tamar Richner, Robb Nebbe
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland
{ducasse,richner,nebbe}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/

to appear in Proceedings WCRE '99

Abstract

In reengineering an object-oriented system we want to benefit from the expertise developed in earlier efforts. It is therefore essential to have a way to communicate expertise at different levels: from knowledge about how to approach a system to be reengineered, to knowledge about improving code by eliminating 'bad' style. In this paper we propose to use a pattern form to communicate knowledge about reengineering. A reengineering pattern *connects an observable problem in the code to a reengineering goal: it describes the process of going from the existing* legacy *solution causing or aggravating the problem to a new* refactored *solution which meets the reengineering goal. It thus gives a method appropriate for a specific problem, rather than proposing a general methodology, and makes reference to the appropriate tools or techniques for obtaining the refactored solution. In this paper we discuss the role of reengineering patterns and contrast them with related kinds of patterns. We then highlight the form of reengineering patterns and present two simple patterns for type-check elimination.*

Keywords: Object-Oriented, Reengineering, Patterns, Refactorings

1. Reengineering Patterns

Reengineering projects, despite their diversity, often encounter some typical problems again and again. These can be problems at different levels and due to different practices [7]. But it is unlikely that one methodology or process will be appropriate for all projects and organizations [14], just as no one tool or technique can be expected to solve all the technical problems encountered in a reengineering project. To allow reengineering projects to benefit from the experience gained in previous efforts, an appropriate form is required for transferring expertise. This form should be small enough to be easily consulted and navigated, and stable enough as to be useful for many reengineering projects.

In the object-oriented software engineering community Design Patterns [9] have been adopted as an effective way of communicating expertise about software design. A design pattern describes a solution for a recurring design problem in a form which facilitates the reuse of a proven design solution. In addition to the technical description of the solution, an important element of a design pattern is its discussion of the advantages and disadvantages of applying the pattern.

We propose to use a pattern form to transfer expertise in the area of reengineering. Reengineering patterns codify and record knowledge about modifying legacy software: they help in diagnosing problems and identifying weaknesses which hinder further development of the system and aid in finding solutions which are more appropriate to the new requirements. We see reengineering patterns as stable units of expertise which can be consulted in any reengineering effort: they describe a process without proposing a complete methodology, and they suggest appropriate tools without 'selling' a specific one. A more thorough discussion of the advantages of the pattern form as a vehicle for reengineering expertise can be found in [14], which discusses patterns closely related to ours.

In the context of a project developing a methodology for reengineering object-oriented legacy systems to frameworks, we are working on a system of patterns for reverse engineering and reengineering. Besides the two reengineering patterns Type Check Elimination in Clients and Type Check Elimination within a Provider Hierarchy presented here, this system of patterns also includes the following patterns: Code Navigation Elimination, Curing God Class, Changing Architectural Dependencies and Transforming Inhieritance into Composition. All these patterns address typical legacy solutions found in object-oriented code, and describe how to move from the legacy solution to a new refactored solution. The patterns presented here are of a technical nature; we expect, however, that some reengineering patterns will describe overall strategies for dealing with legacy systems, and thus be of a less technical nature. Systems Reengineering Patterns[14] are examples of such higher-level patterns which address broader methodological issues. The 'Deprecation' pattern [14], for example, describes how to iteratively change interfaces of a system in a friendly way for the client of the system under change.

The paper is structured as follows: in the next section we compare and contrast our reengineering patterns to related pattern work. In section 3 we give a brief overview of the format used for writing reengineering patterns. In section 4 we introduce the topic of type check elimination addressed by our sample patterns. The two patterns are presented in section 5 and section 6 respectively. Finally we conclude in section 7 with some discussion of the pattern form and its use in reengineering.

2. Reengineering Patterns and Related Work

Our reengineering patterns and Systems Reengineering Patterns [14] are close. The only difference is that our patterns are low level and focus in particular on object-oriented legacy systems. Note that the our patterns cannot be used to evaluate whether or not an application should be reengineered in the first place; this difficult task has been tackled by [1] and [12]. In [5] a methodology is proposed to help in the migration of legacy systems (principally legacy database systems) to new platforms.

Reengineering patterns differ from Design Patterns [9] in their emphasis on the *process* of moving from an existing *legacy* solution to a new *refactored* solution. Whereas a design pattern presents a solution for a recurring design problem, a reengineering pattern presents a refactored solution for a recurring legacy solution which is no longer appropriate, and describes how to move from the legacy solution to the refactored solution. The mark of a good reengineering pattern is (a) the clarity with which it exposes the advantages, the cost and the consequences of the target solution with respect to the existing solution, and not how elegant the target solution is, (b) the description of the change process: how to get from one state of the system to another.

We also contrast reengineering patterns with AntiPatterns [6]. Antipatterns, as exposed by Brown et al., are presented as “bad” solutions to design and management issues in software projects. Many of the problems discussed are managerial concerns that are outside the direct control of developers. Moreover, the emphasis in antipatterns is on prevention: how to avoid making the mistakes which lead to the antipatterns. Consequently, antipatterns may be of interest when starting a project or during development but are no longer helpful when we are confronted with a legacy system. In approaching legacy systems we prefer to withhold judgment and use the term “legacy solution” or “legacy pattern” for a solution which at the time, and under the constraints given, seemed appropriate. In reengineering it is too late for prevention, and reengineering patterns therefore concentrate on the cure: how to detect problems and move to more appropriate solutions.

Finally, our reengineering patterns are different from code refactorings [11, 10, 15, 8]. A reengineering pattern describes a process which starts with the detection of the symptoms and ends with the refactoring of the code to arrive at the new solution. A refactoring is only the last stage of this process, and addresses the technical issue of automatically or semi-automatically modifying the code to implement the new solution. Reengineering patterns also include other elements which are not part of refactorings: they emphasize the context of the symptoms, by taking into account the constraints that reengineers are facing, and include a discussion of the impact of the changes that the refactored solution may introduce.

3. Form of a reengineering pattern

Pattern Name. We use a short sentence with a verb that emphasizes the kind of reengineering transformation.

Intent. A description of the process, together with the result and why it is desirable.

Applicability. When is the pattern applicable? When is it not applicable? This section includes a list of symptoms, a list of reengineering goals and a list of related patterns. Symptoms are those experienced when reusing, maintaining or changing the system. Reengineering goals present the qualities improved through the application of this pattern.

Motivation. This section presents an example: it must acquaint the reader with a concrete example so the reader can better understand the more abstract presentation of the problem which follows in the structure and process sections. The example clearly describes the structure of the existing legacy system, the structure of the reengineered system, and the relation between the two.

Structure. It describes the structure of the system before and after reengineering. As in Design Patterns [9], the participants and their collaborations are identified. Consequences discuss the advantages and disadvantages of the target structure in comparison to the initial structure.

Process. The process section is subdivided into three sections: the detection, the recipe and the difficulties. The detection section describes methods and tools that help to detect when the code is indeed suffering from the serious problems. The recipe states how to perform the reengineering operation and its possible variants. The difficulties section discusses situations where the reengineering operation is infeasible or its application is compromised by other problems.

Discussion. In this section cost and benefit tradeoffs of applying the pattern are discussed. The legacy solution is commented to show why such a solution was appropriate at the time but is now insufficient or inadapted to the current problem. What is the cost of detecting this problem and what is the benefit gained by applying the pattern? This discussion should aid an engineer in deciding whether or not it is worth applying the pattern.

Language Specific Issues. This section lists what must be specifically resolved for each programming language. What makes it more difficult? More easy?

4. Type Check Elimination

The introduction of polymorphism is an important and frequent operation in reengineering object-oriented legacy systems. Replacing hand coded polymorphism with the support built into the language both simplifies the software and makes it more flexible. Even in the presence of polymorphism it is our experience that developers continue to implement through other means functionality that would be best handled through polymorphism.

Here we present two patterns: Type Check Elimination within a Provider Hierarchy and Type Check Elimination in Clients, that deal with the absence of polymorphism in legacy systems. The essential distinction between these two patterns is the *location of the type check:* in Type Check Elimination within a Provider Hierarchy the decision structure is in a provider class and is over an instance variable of that class while in Type Check Elimination in Clients the decision structure is in the client class and is over an instance variable of another class. Note that when one class depends on another we call this class a *client* class and the class it depends on is a *provider* class. This is a general terminology and is not specific to these patterns.

Each reengineering pattern is self contained - some identical parts will therefore appear in both patterns. Patterns are written as self contained units as we have found that depending on our goals when reengineering we may be interested in one pattern and not the other. For example, if we wish to extract a subsystem then Type Check Elimination in Clients is very important. On the other hand if we wish to add functionality then Type Check Elimination within a Provider Hierarchy is more relevant.

5. Type Check Elimination within a Provider Hierarchy

Intent

Transform a single *provider* class being used to implement what are conceptually a set of related types into a hierarchy of classes. Decision structures over type information, such as case statements or if-then-elses, are replaced by polymorphism. This results in increased modularity and facilitates the extension of functionality through the addition of new subclasses.

Applicability

Symptoms.

- Methods contain large decision structures over an instance variable of the *provider* class to which they belong.
- Extending the functionality of the *provider* class requires modifying many methods.
- Many *clients* depend on a single *provider* class.

Reengineering Goals.

- Improve modularity.
- Simplify extension of *provider* functionality.

Related Reengineering Patterns. A closely related pattern is Type Check Elimination in Clients where the case statements over types are in the client code as opposed to the provider code. The pattern is also related to the object-oriented heuristic : “Explicit case analysis on the type of an attribute is often an error” [13].

Motivation

Case statements are sometimes used to simulate polymorphic dispatch. This is often the result of the absence of polymorphism in an earlier version of the language (e.g. Ada’83 $\rightarrow$ Ada’95 or C $\rightarrow$ C^{++}). Another possibility is that programmer don’t fully master the use of polymorphism and as a result do not always recognize when it is applicable. Programmers often fall back to the language they are most familiar with (the the Variable State pattern [3] describes such a situation) and so they may continue to implement solutions which do not exploit polymorphism even when polymorphism is available. This could occur especially when programmers extend an existing design by programming around its flaws, rather than reengineering it.

In a language that supports polymorphism it is preferable to exploit the language support for dispatching rather than to simulate it. Case statements or other large decision structures that simulate dispatch must be coded and maintained by hand, making changes or extensions to the functionality more difficult because many places in the source code are affected. Simulating dispatch also results in long methods with fragmented logic that is hard to understand.

Initial Situation. Our example, taken in a simplified form from a case study, consists of a message class that wraps two different kinds of messages (TEXT and ACTION) that must be serialized to be sent across a network connection as shown in the code and the figure 1.

A single provider class implements what is conceptually a set of related types. One attribute of the class functions as surrogate *type* information and is used in a decision structure to handle different variations of functionality required.

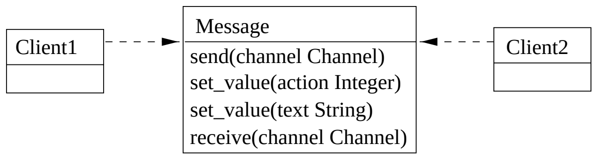

Figure 1. Initial relation and structure of clients and providers.

```
class Message {
public:
  Message();
  set_value(char* text);
  set_value(int action);
  void send(Channel c);
  void receive(Channel c);
...
private:
  void* data;
  int   type_;
}
// from Message::send
  const int TEXT   = 1;
  const int ACTION = 2;
  switch (type_) {
  case TEXT: ...
  case ACTION: ...  };
```

Final Situation. The case statements have been replaced by polymorphism and the original class has been transformed into a hierarchy comprised of an abstract superclass and concrete subclasses. Clients must then be adapted to create the appropriate concrete subclass.

Initially there may be a large number of dependencies on this class, making modification expensive in terms of compilation time, and increasing the effort required to test the class. The target structure improves all of these problems with the only cost being the effort required to refactor the provider class and to adapt the clients to the new hierarchy.

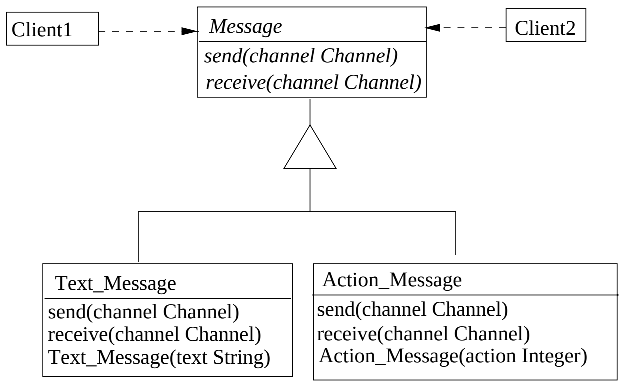

Figure 2. Final relation and structure of clients and providers.

```
class Message {
public:
  virtual void send(Channel c) = 0;
  virtual void receive(Channel c) = 0;
...
};

class Text_Message: public Message {
public:
  Text_Message(char* text);
  void send(Channel c);
  void receive(Channel c);
private:
  char* text;
...
};

class Action_Message: public Message {
public:
  Action_Message(int action);
  void send(Channel c);
  void receive(Channel c);
private:
  int action;
...
};
```

Structure

Participants.

- A single **provider** (Message) class that is transformed into a hierarchy of classes (Message, Text_Message and Action_Message)
- A set of **client** classes

Collaborations. The single provider class will be transformed into a hierarchy, thereby increasing modularity and facilitating extension of functionality.

Initially, the clients are all dependent on a single provider class. This class encompasses several variants of functionality and thus encapsulates all the collaboration that would normally be handled by polymorphism. This results in long methods typically containing case statements or other large decision structures.

The situation is improved by refactoring the single provider class into a hierarchy of classes: an abstract superclass and a concrete subclass for each variant. Each of the new subclasses is simpler than the initial class and these are relatively independent of each other.

Consequences. The functionality of the hierarchy can be extended adding a new subclass without modifying the superclass. The increased modularity also impacts the clients who are now likely to be dependent on separate subclasses in the provider hierarchy.

Process

Detection. For the automatic detection of the initial situation in legacy code, a class having many long methods is a good candidate for further analysis. A line-of-code-per-method metric may help to narrow the search. If these methods contain case statements or complex decision structures all based on the same attribute then the attribute is probably serving as surrogate type information. In C++, where it is a good practice to define a class per file, the frequency of case statements in the same file can be also used as a first hint to narrow the search for this pattern.

Example: detection of case statements in C++. Knowing if the pattern should be applied requires the detection of case statements. Regular-expression based tools like emacs, grep, agrep help in the localization of case statements based on explicit construct like C++'s switch. For example, grep 'switch' ‘find . -name "*.cxx" -print‘ enumerates all the files with extension .cxx contained in a directory tree that contains switch. With agrep, the expression agrep 'switch;type' -e ‘find . -name "*.cxx" -print‘ extracts all the files containing lines having switch and type.

These tools are not well suited, however, for detecting case statements based on explicit ifthenelse structures, since their detection capabilities are restricted to one line at a time. One possible solution is to use perl scripts - a perl script which searches the methods in C++ files and lists the occurrences of case statements is given in the appendix.

Recipe.

1. Determine the number of types conceptually implemented by the class by inspecting the case statements. An enumeration type or set of constants will probably document this as well.
2. Implement the new provider hierarchy. You will need an abstract superclass and at least one derived concrete class for every variant.
3. Determine if all of the methods need to be declared in the superclass or if some belong only in a subclass.
4. Update the clients of the original class to depend on either the abstract superclass or on one of its concrete subclasses.

Difficulties.

- If the case statements are not all over the same set of functionality variants this is a sign that it might be necessary to have a more complex hierarchy including several intermediate abstract classes, or that some of the state of the provider should be factored out into a separate hierarchy.
- If a client depends on both the superclass and some of the subclasses then you may need to refactor the client class or apply the Type Check Elimination in Clients pattern because this is an indication that the provider does not support the correct interface.

Discussion

About the legacy solution. The legacy solution is a good solution when the language does not support polymorphism. The variants represent the subclasses in a object-oriented languages. The functionalities can be shared between the variants and the polymorphism can be simulated. Note also that the type checks occur only in the implementation of the provider and there is very little complexity distributed across the clients.

The major drawback is that the provider class quickly becomes very large as the number of variants increases and the particularities of each variant must be taken into account. The complexity is reflected in the methods that must distinguish the different variants and their logic becomes fragmented and difficult to follow. Adding new variants often requires making extensive modifications to the provider class.

About Detection. It is well-known that the code often tells the reengineer where the problem is and that the reengineering process must be goal driven to avoid reengineering code that is not obstructing further development. However, there are cases where it may be interesting to automatically detect where a pattern could be applied.

During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [4, 2]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [9, 2].The Strategy pattern [9, 2] is also based on the elimination of case statement over object state.

Opdyke [11] discusses "Refactoring To Specialize", in which he proposes to use class invariants as a criteria to simplify conditionals. His proposal for automatic refactoring is similar to this pattern.

About the refactored solution. Applying the pattern may lead to a number of new classes, basically transforming what is often a single class into a hierarchy of classes. However, these classes already existed conceptually in the legacy solution and we are trading one very complex class for simpler but more numerous classes. The logic of each class is then much cleaner since you do not need to filter out the noise associated with the other variants and the typing can be tightened so that unwanted variants can be excluded by the type system rather than through a precondition check.

Language Specific Issues.

C++. Detection: in C polymorphism can be emulated either by using function pointers or through union types and enum's. C++ programmers are likely to use a single class with a void pointer and then cast this pointer to the appropriate type inside a switch statement. This allows them to use classes which are nominally object-oriented as opposed to unions which they have probably been told to avoid. The use of constants is typically favored over the use of enum's.

Difficulties: If void pointers have been used in conjunction with type casts then you should check to see if the classes mentioned in the type casts should be integrated into the new provider hierarchy.

ADA. Detection: because Ada83 did not support polymorphism (or subprogram access types) discriminated record types are the preferred solution. Typically an enumeration type provides the set of variants and the conversion to polymorphism is straightforward in Ada95.

SMALLTALK. Detection: In SMALLTALK the detection of the case statements over types is hard because few type manipulations are provided. Basically, methods isMemberOf: and isKindOf: are available. Detecting these method calls is not sufficient, however, since class membership can also be tested with self class = anotherClass, or with property tests throughout the hierarchy using methods like isSymbol, isString, isSequenceable, isInteger.

JAVA. Detection: look for the use of the operator instanceof. Note that this operator returns true if the object on its left-hand side is an instance of the class or implements the interface specified on its right-hand side. As classes are not real objects in JAVA a programmer cannot directly compare two references as in SMALLTALK, but could compare class names, so you may look for getClass() and getName() combined with string comparison.

6. Type Check Elimination in Clients

Intent

Transform *client* classes that depend on type tests (usually in conjunction with case statements) into *clients* that rely on polymorphism. The process involves factoring out the functionality distributed across the clients and placing it in the provider hierarchy. This results in lower coupling between the *clients* and the *providers* (class hierarchy).

Applicability

Symptoms.

- Large decision structures in the *client* over the type of (or equivalent information about) an instance of the *provider,* either passed as an argument to the client, an instance variable of the client, or a global variable.
- Adding a new subclass of the *provider* superclass requires modifications to *clients* of the *provider* hierarchy because functionality is distributed over these clients.

Reengineering Goals.

- Localize functionality distributed across *clients* in the *provider* hierarchy.
- Improve usability of *provider* hierarchy.
- Lower coupling between *clients* and the *provider* hierarchy.

Related Reengineering Patterns. A closely related reengineering pattern is Type Check Elimination within a Provider Hierarchy, where the case statements over types are in the *provider* code as opposed to the *client* code. The pattern is also related to the object-oriented heuristic: "Explicit case analysis on the type of an object is usually an error." [13].

Motivation

The fact that the clients depend on provider type tests is a well known symptom for a lack of polymorphism. This leads to unnecessary dependencies between the classes and it makes it harder to understand the program because the interfaces are not uniform. Furthermore, adding a new subclass requires all clients to be adapted.

Initial Situation. The following code illustrates poor use of object-oriented concepts as shown by Fig. 3. The function makeCalls takes a vector of Telephone's (which can be of different types) as a parameter and makes a call for each of the telephones. The case statement switches on an explicit type-flag returned by phoneType(). In each branch of the case, the programmer calls the phoneType specific methods identified by the type-tag to make a call.

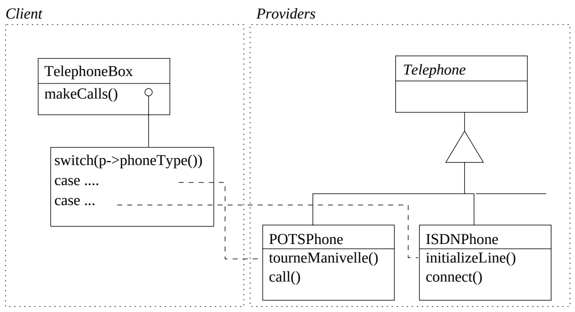

Figure 3. Initial relation and structure of clients and providers.

```
void makeCalls(Telephone * phoneArray[])
{
  for (Telephone *p = phoneArray; p; p++) {
    switch(p->phoneType()) {
    case TELEPHONE::POTS: {
      POTSPhone * potsp = (POTSPhone *) p;
      potsp->tourneManivelle();
      potsp->call(); break;}
    case TELEPHONE::ISDN: {
      ISDNPhone * isdnp = (ISDNPhone *) p;
      isdnp->initializeLine();
      isdnp->connect(); break;}
    case TELEPHONE::OPERATORS: {
      OperatorPhone * opp = (OperatorPhone *) p;
      opp->operatormode(on);
      opp->call(); break;}
    case TELEPHONE::OTHERS:
    default:
        error(....);
    } } }
```

Final Situation. After applying the pattern the corresponding ringPhones() will look as follows and the structure as shown by the Fig. 4.

```
void makeCalls(Telephones *phoneArray[])
{
    for(Telephone *p = phoneArray; p; p++) p-
>makeCall();
}
```

Note that the client code, which represents distributed functionality, has been greatly simplified. Furthermore, this functionality has been localized within the Telephone class hierarchy, thus making it more complete and uniform with respect to the clients needs.

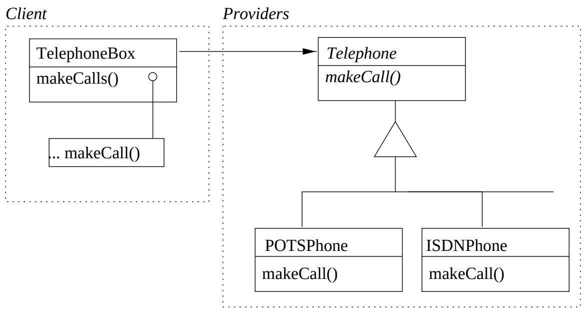

Figure 4. Final relation and structure of clients and providers.

Structure

Participants.

- **provider classes** (Telephone and its subclasses)
  - organized into a hierarchy.
- the **clients** (TelephoneBox) of the provider class hierarchy.

Collaborations.

The collaborations will change between all clients and the providers as well as the collaboration within the provider hierarchy.

Initially, the clients collaborate directly with the provider superclass and its subclasses by virtue of type tests or a case statement over the types of the subclasses. After reengineering the only direct collaboration between the clients and the providers is through the superclass. Interaction specific to a subclass is handled indirectly through polymorphism.

Within the provider hierarchy the superclass interface must be extended to accurately reflect the needs of the clients. This will involve the addition of new methods and the possible refactorization of the existing methods in the superclass. Furthermore, the collaborations between the provider superclass and its subclasses may also evolve, i.e. it must be determined whether the new or refactored methods are abstract or concrete.

Consequences.

Relying on polymorphism localizes the protocol for interacting with the provider classes within the superclass. The collaborations are easier to understand since the interface actually required by the clients is now documented explicitly in the provider superclass. It also simplifies the addition of subclasses since their responsibilities are defined in a single place and not distributed across the clients of the hierarchy.

Process

Detection. The technique described in the pattern Type Check Elimination within a Provider Hierarchy to detect case statements is applicable for this pattern. Whereas in the pattern Type Check Elimination within a Provider Hierarchy, the switches are located in the same class, hence in one file for a language like C++, in this pattern the case statements occur in several classes which can be spread over different files.

Recipe. The process consists of two major steps. The first is to encapsulate all the responsibilities that are specific to the provider classes within the provider hierarchy. The second is to make sure that these responsibilities are correctly distributed within the hierarchy.

1. Determine the set of clients to which the pattern will be applied.

2. Define a new abstract method in the provider superclass and concrete methods implementing this method in each of the subclasses based on the source code contained within each branch of the case statement.

3. Refactor the interface of the provider superclass to accurately reflect the protocol used by the clients. This involves not only adding and possibly changing the methods included but determining how they work together with the subclasses to provide the required behavior. This includes determining whether methods are abstract or concrete in the provider superclass.

4. For each client, rewrite the method containing the case statement so that it uses only the interface of the provider superclass.

Difficulties.

1. The set of clients may all employ the same protocol; in this case the pattern needs to be applied only once. However, if the clients use substantially different protocols then they can be divided into different kinds and the pattern must be applied once for each kind of client.

2. If the case statement does not cover all the subclasses of the provider superclass a new abstract class may need to be added and the client rewritten to depend on this new class. For example, if it is an error to invoke the client method with some subclasses as opposed to just doing nothing then the type system should be used to exclude such cases. This reduces the provider hierarchy to the one starting at the new abstract class.

3. Refactoring the interface will affect all clients of the provider classes and must not be undertaken without examining the full consequences of such an action.

4. Nested case statements indicate that multiple patterns must be applied. This pattern may need to be applied recursively in which case it is easiest to apply the pattern to the outermost case statement first. The provider classes then become the client classes for the next application of the pattern. Another possibility is when the inner case statement is also within the provider class but some of the state of the provider classes should be factored out into a separate hierarchy.

Discussion

About the legacy solution. Explicit type check are sometimes necessary. This is the case when the programmers are working at the frontier between object-oriented and non-object-oriented applications [13]. For example, writing in C++ does not save one from dealing with the way events are handled in the X window system, since X inevitably loses type information about events by placing them in the event queue. The application then receives these as events and must explicitly check their type to determine how to handle them.

Java deals with this problem by changing the approach to event handling. Widgets register for events and specify what they want done with the events. This short circuits the loss of type information and eliminates the type checks.

Type checks are made necessary when type information is lost. Loss of type information may be an artifact of the design and relatively straight forward to repair, as in this pattern and the Type Check Elimination within a Provider Hierarchy pattern. In other cases this may require an extensive redesign. In such cases the advantages of "doing it right" or at least making it look like it was done right with a wrapper must be weighed against the costs.

About detection. During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [4, 2]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [9, 2]. The Strategy pattern [9, 2] is also based on the elimination of case statement over object state.

About evolution. If the application currently reengineered has been in the past distributed as a library and is used now by other programmers, the Deprecation pattern [14] can be applied to deal with the interface conflicts between the old version and the new ones.

About the refactored solution. Applying the pattern will lead to changes in the interface and may introduce new classes in the provider hierarchy. However, the interface as well as the classes already existed conceptually in the legacy solution. The mismatch between the abstraction presented and the abstraction required created complexity that was distributed across the clients. In the refactored solution the complexity is localized within the provider hierarchy instead of being distributed across the clients of the hierarchy. This makes the client code much easier to understand and the role of the provider classes is much better documented by their interfaces.

Language Specific Issues.

C++. In C++ virtual methods can only be used for classes that are related by an inheritance relationship. The polymorphic method has to be declared in the superclass with the keyword virtual to indicate that calls to this methods are dispatched at runtime. These methods must be redefined in the subclasses.

Type information is encoded often using some enum type. A data member of a class having such an enum type and a method to retrieve these tags are usually a hint that polymorphism could be used (although there are cases in which polymorphic mechanism cannot substitute the manual type discrimination).

ADA. Detecting type tests falls into two cases. If the hierarchy is implemented as a single discriminated record then you will find case statements over the discriminant. If the hierarchy is implemented with tagged types then you cannot write a case statement over the types (they are not discrete); instead an if-then-else structure will be used.

If a discriminated record has been used to implement the hierarchy it must first be transformed by applying the Type Check Elimination within a Provider Hierarchy pattern.

SMALLTALK. In SMALLTALK the detection of the case statements over types is hard because few type manipulations are provided. Basically, methods isMemberOf: and isKindOf: are available. Detecting these method calls is not sufficient, however, since class membership can also be tested with self class = anotherClass, or with property tests throughout the hierarchy using methods like isSymbol, isString, isSequenceable, isInteger.

JAVA. Detection: look for the use of the operator instanceof. Note that this operator returns true if the object on its left-hand side is an instance of the class or implements the interface specified on its right-hand side. As classes are not real objects in JAVA a programmer cannot directly compare two references as in SMALLTALK, but could compare class names, so you may look for getClass() and getName() combined with string comparison.

7. Discussion and Conclusions

We have proposed a pattern form as the appropriate style for communicating expertise about the reengineering of software systems. Reengineering patterns are stable units of expertise which can be consulted in any reengineering effort dealing with object-oriented legacy systems.

But how do we validate patterns? Since patterns are intended to document reengineering processes that have proven to be useful, knowing that a pattern has been applied successfully several times gives us confidence in the pattern. We therefore seek to collect examples of reengineering efforts which have applied these patterns. In the FAMOOS project[1], we have studied several case studies (ranging from 50K LOC to 2,500K LOC of C++ and Ada) which are being reengineered in industry. Obtaining information about the actual reengineering processes in industry is difficult, however, since these are rarely documented, and often can not be disclosed. From discussions and project workshops with developers in industry we have confirmation that such patterns are actually applied.

We have chosen to present here two patterns dealing with symptoms of missing polymorphism in legacy systems because they present situations that many developers have encountered, and are closely related to known design patterns[9] and to some refactoring operations[11]. In other words, they appear convincingly familiar already to most developers. We have used our detection script to find instances of type checks, both in clients and within a provider hierarchy, in both C++ and Ada case studies. Based on our exchange with developers we have elaborated the discussion of the advantages and disadvantages of the pattern application, and gained a better understanding of the constraints under which the 'legacy solution' was chosen.

As mentioned in the introduction, these two patterns are part of a larger pattern language which addresses a range of different problems encountered in reengineering object-oriented legacy systems. Such a pattern language is intended for use with navigational help, which takes several different forms. First, patterns are related to the stage of the reengineering process at which they are to be applied - some patterns are to be used in reverse engineering at the first encounter with the system, others are intended for refactoring the code once a better understanding of the system has been gained. Second, each pattern is associated with observed symptoms and with a set of reengineering goals - this enables engineers to look for patterns which could be applicable in their project. Finally, patterns are related to each other, and so can be navigated through these relationships. We are now working on this kind of navigational guidance which will take the form of a reengineering handbook.

[1]The FAMOOS ESPRIT project investigates tools and techniques for transforming object-oriented legacy systems into frameworks. See *http://www.iam.unibe.ch/∼famoos/*

Acknowledgments. Our work on reengineering patterns is part of ongoing work on a reengineering handbook, in the context of the FAMOOS ESPRIT project. We thank our colleagues - Serge Demeyer, Matthias Rieger and Sander Tichelaar - for their collaboration in this project and for comments on the paper. Thanks also to Eduardo Casais, Oscar Nierstrasz, and to our shepherd at EuroPLOP'98, Don Roberts, for their comments on an earlier draft of this document. Patrick Steyaert helped us with the development of the appropriate pattern form. This work has been funded by the Swiss Government under grant MHV 21-41671.94 (to T.R.), project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

References

[1] Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997. (http://stsc.hill.af.mil/RENG).

[2] S. R. Alpert, K. Brown, and B. Woolf. *Design Patterns in Smalltalk*. Addison-Wesley, 1998.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[4] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, LNCS 821, pages 139–149. Springer-Verlag, July 1994.

[5] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan Kaufman, 1995.

[6] W. J. Brown, R. C. Malveau, H. W. S. McCormickIII, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley and Sons, 1998.

[7] B. Foote and J. W. Yoder. Big Ball of Mud. In *Proceedings of PLoP'97*, 1997.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[10] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, Nov. 1993.

[11] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[12] J. Ransom, I. Sommerville, and I. Warren. A Method for Assessing Legacy Systems for Evolution. In *Proceedings of Reengineering Forum'98*, 1998.

[13] A. J. Riel. *Object Oriented Design Heuristics*. Addison-Wesley, 1996.

[14] P. Stevens and R. Pooley. System reengineering patterns. In *Proceedings of FSE-6*. ACM-SIGSOFT, 1998.

[15] L. Tokuda and D. Batory. Automating three modes of evolution for object-oriented software architectures. In *Proceedings COOTS '99*. USENIX, 1999.

A. Detecting Case Statements.

This perl script searches the methods in C^{++} files and lists the occurences of statements matching the following expression: `elseXif` where `X` can be replaced by {, `//...` or some white space including carriage return.

```
#!/opt/local/bin/perl
$/ = '::';
# new record delim.,
$elseIfPattern = 'else[\s\n]*{?[\s\n]*if';
$linecount = 1;
while (<>) {
 s/(\/\/.*)//g;    # remove C++ style comments
 $lc = (split /\n/) - 1; # count lines

 if(/$elseIfPattern/) {
  # count # of lines until first
# occurence of "else if"
   $temp = join("",$`,$&);
   $l = $linecount + split(/\n/,$temp) - 1;
   # count the occurences of else-if pairs,
   # flag the positions for an eventual printout
   $swc = s/(else)([\s\n]*{?[\s\n]*if)
                       /$1\n\t\t* HERE *$2/g;
   printf "\n%s:  Statement with
           %2d else-if's, first at: %d",
           $ARGV, $swc, $l;
 }
 $linecount += $lc;
 if(eof) {
   close ARGV;
   $linecount = 0;
   print "\n";
 }
}
```