Transform Conditionals: a Reengineering Pattern Language

Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz
Serge.Demeyer@uia.ua.ac.be
{ducasse,oscar}@iam.unibe.ch

Abstract. The reengineering pattern presented in this paper shows how you can transform conditionals in object-oriented code to improve the flexibility of application.

1. Introduction

Legacy systems are not limited to the procedural paradigm and languages like Cobol. Even if object-oriented paradigm promised the building of more flexiblesystems and the ease in their evolution, nowadays object-oriented legacy systems exist in C++, Smalltalk or Java. These legacy systems need to be reengineered to meet new requirements. The goal of the Famoos Esprit project was to support the evolution of such a object-oriented legacy systems towards frameworks.

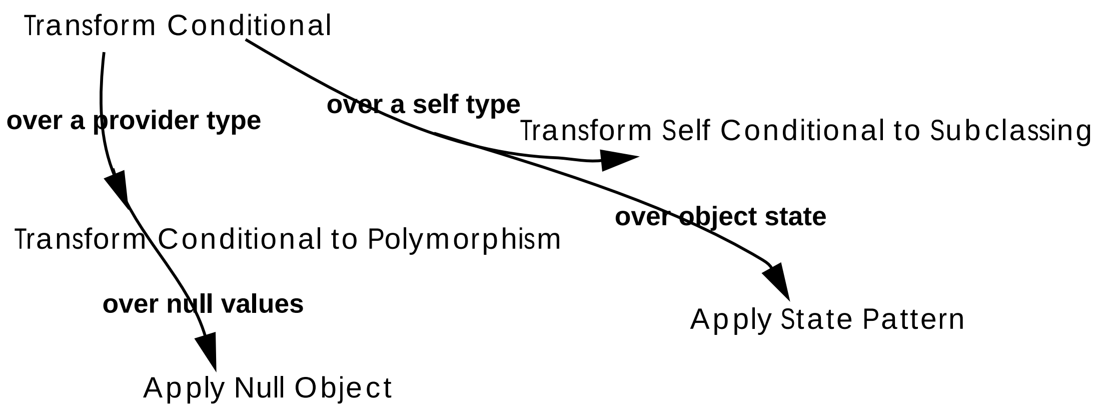
In this context, we used patterns as a way to record reengineering expertise. We wrote reverse engineering patterns that record how to extract information of the legacy systems from the code, the organization or the people [Deme99a] and reengineering patterns that present how code can be transformed to support new requirements, to be more flexible or to simply follow object-oriented design [Duca99a]. Transform Conditional is a reengineering pattern that presents how conditionals can be transformed to improve the flexibility of object-oriented application.

Acknowledgements. This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT program Project no. 21975. We would like to thanks the attendees of the SCG internal workshop writer at the Universitu of Bern: Serge Demeyer, Pietro Malorgio, Tamar Richner,Matthias Rieger and Sander Tichelaar.

Transform Conditional

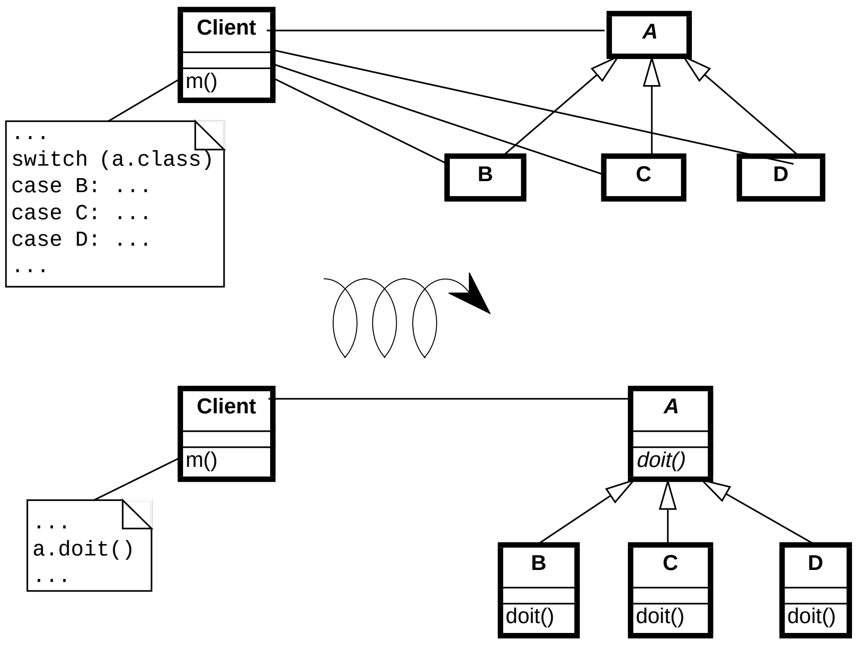
Transform Conditional is a pattern language describing how switch statements are transformed into code that is more flexible and exhibits less coupling between classes. This pattern language consists of four patterns,Transform Self Conditional to Subclassing, Transform Client Conditional to Polymorphism, Apply State and Apply Null Object. For Apply State and Apply Null Object our intention is not to copy two established design patterns : *State* and *NullObject* but rather to provide a more specific reading with a focus on reengineering. We invite the reader to read [Gamm95a, Alpe98a, Dyso98a] and [Wool98a] for the original description of the *State* and *NullObject*.

The following picture summarizes their differences.



- Transform Self Conditional to Subclassing eliminates switch statements over type information by introducing subclasses for each type case, and by replacing the conditional code with a single polymorphic method call to an instance of one of the new subclasses.
- Apply State is a special case of Transform Self Conditional to Subclassing in the sense they both transform a conditional within the class itself to a polymorphic call. In Apply State the conditional over the state is transformed into methods associated with different delegated classes representing the different states.
- Transform Client Conditional to Polymorphism transforms a switch statement over type information in a client class by introducing polymorphic methods in the provider and calling them from the client class.
- Apply Null Object is a special case of Transform Client Conditional to Polymorphism in the sense they both transform a conditional expression over the provider into a polymorphic call. Here the type of the provider is reduced to the most simple expression: the null value. The condition that check for a null value is transformed by creating a *NullOb-*

ject class that performs the default behaviour, liberating the client from having to type check before performing an operation.



Transform Self Conditional to Subclassing

Make a class more extensible by transforming complex conditional code that tests immutable state into a single polymorphic call to a hook method on the same class. The hook method will be implemented by a different subclass for each case of the conditional.

Problem

A class is hard to modify or subclass because it implements multiple behaviours depending on the value of some immutable attribute.

Context

You need to modify the functionality of a class or add new functionality.

Applicability

You have access to the source code of the class and of clients that instantiate it.

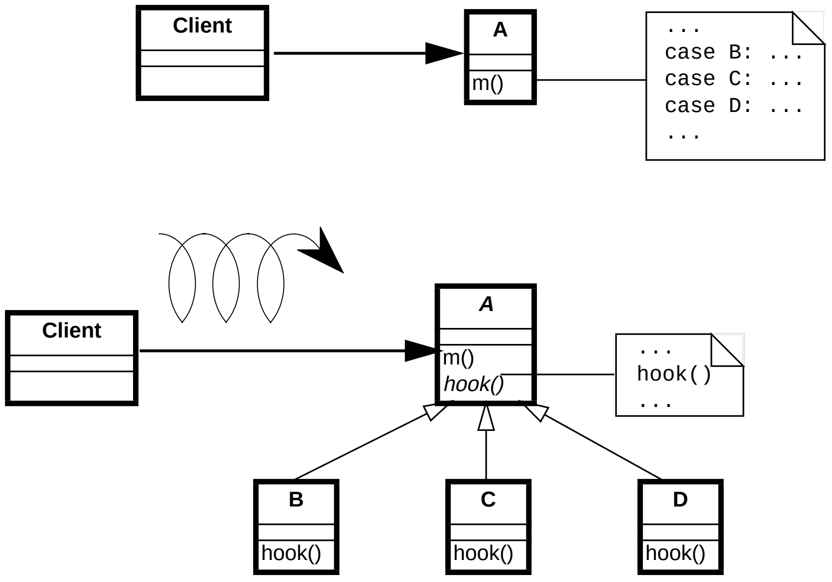
Symptoms

- The class you want to modify has long methods with complex conditional branches.
- Instances of the class seem to represent multiple data types each with different behaviour.
- The expression being tested in the conditional represents type information over the class containing the expression itself.
- The behaviour of a class depends on the value of some immutable attribute.
- Conceptually simple extensions require many changes to the conditional code.
- Subclassing is next to impossible without duplicating and adapting the methods with conditional code.

Solution

Identify the methods with complex conditional branches. In each case, replace the conditional code with a call to a new hook method. Identify or introduce subclasses corresponding to the cases of the conditional. In each of these subclasses, implement the hook method with the code corresponding to that case in the original case statement.

Structure/Participants



Detection

Most of the time, the type discrimination will jump in our face while you are working on the code, so this means that you will not really need to detect where the checks are made. However, it can be interesting to have simple techniques to quickly assess if unknown parts of a system suffer from similar practices. This can be a valuable source of information to evaluate the state of a system.

- Look for long methods with complex decision structures on some immutable attribute of the object that models type information. In particular look for attributes that are set in the constructor and never changed.
- Especially look for classes where multiple methods switch on the same attribute. This is often a sign that the attribute is being used to simulate a type.
- As methods containing switch statements tend to be long, it may help to use a tool that sorts methods by lines of code or visualizes classes and methods according to their size. Alternatively, search for classes or methods with a large number of conditional statements.
- For languages like C++ or Java where it is common to store the implementation of a class in a separate file, it is straightforward to search for and count the incidence of conditional keywords (`if`, `else`, `case`, etc.). On a UNIX system, for example,

  ```
  grep 'switch' ‘find . -name "*.cxx" -print‘
  ```

  enumerates all the files in a directory tree with extension `.cxx` that contain a `switch`. Other text processing tools like agrep offer possibilities to pose finer granularity queries. Text processing languages like Perl may be better suited for evaluating some kinds of queries, especially those that span multiple lines.

C/C++: Legacy C code may simulate classes by means of union types. Typically the union type will have one data member that encodes the actual type. Look for conditional statements that switch on such data members to decide which type to cast a union to and which behaviour to employ.

In C++ it is fairly common to find classes with data members that are declared as void pointers. Look for conditional statements that cast such pointers to a given type based on the value of some other data member. The type information may be encoded as an `enum` or (more commonly) as a constant integer value.

Instead of defining subclasses of the class containing the conditional statement, consider also whether the types to which the void pointer is cast can be integrated into a single hierarchy.

Ada: Because Ada83 did not support polymorphism (or subprogram access types), discriminated record types are often used to simulate polymorphism. Typically an enumeration type provides the set of variants and the conversion to polymorphism is straightforward in Ada95.

Smalltalk: Smalltalk provides only a few ways to manipulate types. Look for applications of the methods `isMemberOf:` and `isKindOf:`, which signal explicit type-checking. Type checks might also be made with tests like `self class = anotherClass`, or with property tests throughout the hierarchy using methods like `isSymbol`, `isString`, `isSequenceable`, `isInteger`.

Steps

1. Identify the class to transform and the different conceptual classes that it implements. An enumeration type or set of constants will probably document this well.
2. Introduce a new subclass for each behaviour that is implemented. Modify clients to instantiate the new subclasses rather than the original class. Run the tests.
3. Identify all methods of the original class that implement varying behaviour by means of conditional statements. If the conditionals are surrounded by other statements, move them to separate, protected hook methods. When each conditional occupies a method of its own, run the tests.
4. Iteratively move the cases of the conditionals down to the corresponding subclasses, periodically running the tests.
5. The methods that contain conditional code should now all be empty. Replace these by abstract methods and run the tests.
6. Alternatively, if there are suitable default behaviours, implement these at the root of the new hierarchy.
7. If the logic required to decide which subclass to instantiate is non-trivial, consider encapsulating this logic as a factory method of the new hierarchy root. Update clients to use the new factory method and run the tests.

Tradeoffs

Pros

- Different clients now depend on different subclasses of the original class, thereby improving modularity. Furthermore, functionality can now be extended by defining additional subclasses, without affecting clients of the existing classes.

Cons

- The larger number of classes makes the design more complex, and potentially harder to understand. If the original conditional statements are simple, it may not be worthwhile to perform this transformation.

Difficulties

- Wherever instances of the transformed class were originally created, now instances of different subclasses must be created. If the instantiation occurred in client code, that code must now be adapted to instantiate the right class. Factory objects or methods may be needed to hide this complexity from clients.
- If you do not have access to the source code of the clients, it may be difficult or impossible to apply this pattern since you will not be able to change the calls to the constructors. Evaluate carefully whether it is possible to present the transformed design through the old interface or if *Double Dispatch* can be applied.
- If the case statements test more than one attribute, it may be necessary to support a more complex hierarchy, possibly requiring multiple inheritance. Considering splitting the class into parts, each with its own hierarchy.

When the legacy solution is the solution

Explicit type checks cannot always be avoided. One of the few good reasons to use type check instead of polymorphism is when polymorphism cannot be used! Indeed when the code is dealing with the limits of the paradigm like using non object-oriented libraries or when streaming in objects from files. When streaming objects in from a text file representation, the objects do not yet exist, so an explicit type check is necessary to recreate the objects. In this case, once the instances are created, methods can then be called to fill the object instance variable values.

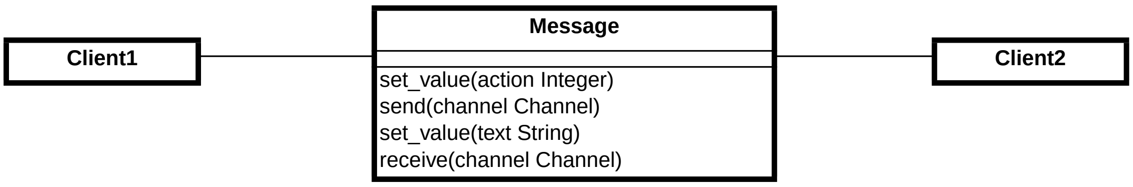
Tolerating type checks

Explicit type checks are not always a problem. In particular they may be an alternative to the creation classes when:

- the set over which the method selection is fixed and will not evolve in the future.
- the typecheck is only made in one place.

Example

A message class 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. We would like to be able to send a new kind of message (say `VOICE`), but this will require changes to several methods of Message.



```
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_;
    static const int TEXT = 1;
    static const int ACTION = 2;
    ...
}
```

```
Message::send(Channel c) {
    switch (type_) {
    case TEXT:
        ...
    case ACTION:
        ...
    }
}
void Client1::doit() { ...
    Message * myMessage =
        new Message();
    myMessage->set_Value("...");
    ...
}
```

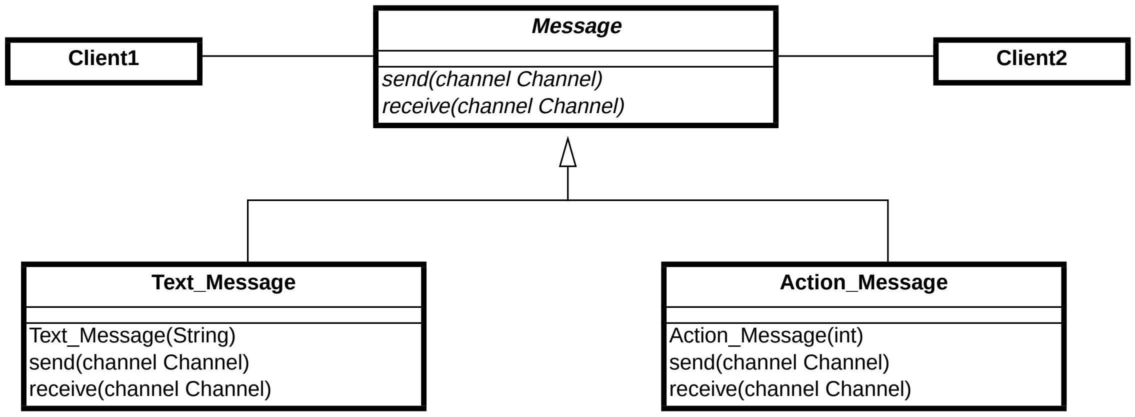
Figure 1 Initial design and source code.

Since `Message` conceptually implements two different classes, `Text_Message` and `Action_Message`, we introduce these as subclasses of `Message`. We introduce constructors for the new classes, we modify the clients to construct instances of `Text_Message` and `Action_Message` rather than `Message`, and we remove the `set_value()` methods. Our regression tests should run at this point.

Now we find methods that switch on the `type_` variable. In each case, we move the entire switch statement to a separate, protected hook method, unless the switch already occupies the entire method. In the case of `send()`, this is already the case, so we do not have to introduce a hook method. Again, all our tests should still run.

Now we iteratively move cases of the switch statements from `Message` to its subclasses. The `TEXT` case of `Message::send()` moves to `Text_Message::send()` and the `ACTION` case moves to `Action_Message::send()`. Every time we move such a case, our tests should still run.

Finally, the original send() method is now empty, so it can be redeclared to be abstract (i.e., `virtual void send(Channel) = 0`). Again, our tests should run.



```
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;
...
};

void Client1::doit() { ...
    Message * myMessage = new
        Text_Message("...");
    ...
}
```

Figure 2 Resulting hierarchy and source code.

Rationale

Classes that masquerade as multiple data types make a design harder to understand and extend. The use of explicit type checks leads to long methods that mix several different behaviours. Introducing new behaviour then requires changes to be made to all such methods instead of simply specifying one new class representing the new behaviour.

By transforming such classes to hierarchies that explicitly represent the multiple data types, you make your design more transparent, and consequently easier to maintain.

Related Patterns

In Transform Self Conditional to Subclassing the condition tests type information of the class that contains it. A similar situation is addressed in Apply State where the conditional tests over state. From this point of view, Apply State is a specialization of Transform Self Conditional to Subclassing even if the solution proposed by the *State* pattern introduces state classes that are not subclasses of the original class. On the other hand, inTransform Client Conditional to Polymorphism the conditional expressions are used to invoke methods not of the class itself but of provider classes.

Replace Type Code with Subclasses, Refactoring To Specialize are two refactorings that can be used to apply the pattern. If the conditional code tests *mutable* state of the object, consider instead applying Transform Client Conditional to Polymorphism. Otherwise, if state of other objects is tested, such as arguments to the method, then consider applying Transform Client Conditional to Polymorphism.

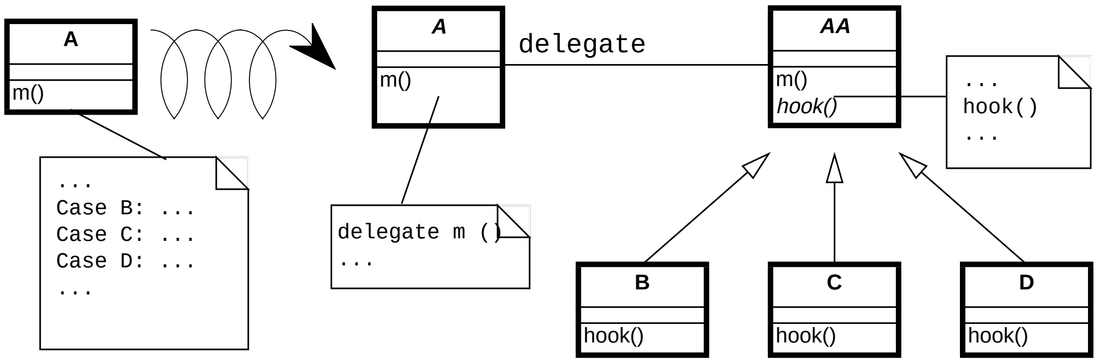
Discussion

Why the legacy solution may have been applied

The problem may arise for various reasons:

- The class may have been repeatedly extended with code to handle special cases to satisfy the needs of many different clients. Whereas the original design of the class may have been simple, it now contains several methods with complex conditional logic over its attributes.
- Programmers may have decided not to define subclasses to handle special cases to avoid cluttering the name space, or to keep changes and extensions local to a single class. It is rarely obvious when varying behaviour is better implemented by subclassing than by conditional code. (In Smalltalk, for example, True and False are subclasses of Boolean, but this is not the case in most other object-oriented languages.)
- In languages without polymorphism, case statements may be used to simulate polymorphic dispatch. Even if a later version of the language does support polymorphism (e.g., C++ vs. C, or Ada 95 vs Ada 83), coding conventions in place may encourage programmers to continue to apply the outdated idiom.

Transform Self Conditional to Subclassing can be composed with delegation when the class containing the original conditional cannot be subclassed. One solution is then to use the polymorphism on another hierarchy, by moving part of the state and behaviour of the original class into a separate class to which the method will delegate.



Transform Client Conditional to Polymorphism

Transform conditional code that tests the type of a provider object into a polymorphic call to a new method, thereby reducing client/provider coupling.

Problem

It is hard to extend a provider hierarchy because many of its clients perform type checks on its instances to decide what actions to perform.

Context

You want to add a new subclass to a provider hierarchy. You have access to both the client and provider source code.

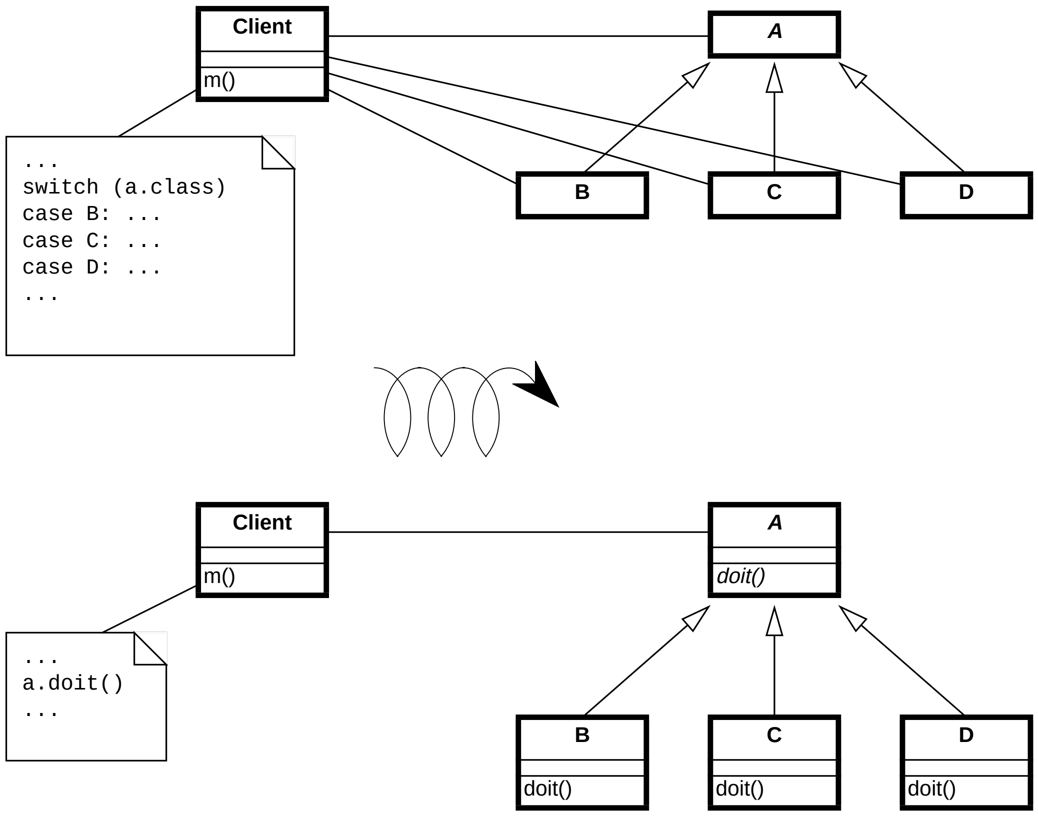
Symptoms

- Clients of the class you want to subclass have long conditional methods that test the type of provider instances.
- Adding a new subclass to the provider hierarchy requires making changes to clients, especially where there tests occur.
- The fact that the Law of Demeter is violated, e.g. that the clients access private data of the provider can be a symptom especially when combined with the fact that these private data are used to select the provider method to be invoked.

Solution

Replace the client's conditional code by a call to a new method of the provider hierarchy. Implement the new method in each provider class by the appropriate case of the original conditional code.

Structure/Participants



Detection

Apply essentially the same techniques described in Transform Self Conditional to Subclassing to detect case statements, but look for conditions that test the type of a separate service provider which *already* implements a hierarchy. You should also look for case statements occurring in different clients of the same provider hierarchy.

C++: Legacy C++ code is not likely to make use of run-time type information (RTTI). Instead, type information will likely be encoded in a data member that takes its value from some enumerated type representing the current class. Look for client code switching on such data members.

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.

Smalltalk: As in Transform Self Conditional to Subclassing, look for applications of `isMemberOf:` and `isKindOf:`, and tests like `self class = anotherClass`.

Java: Look for applications of the operator `instanceof`, which tests membership of an object in a specific, known class. Although classes in Java are not objects as in Smalltalk, each class that is loaded into the virtual machine is represented by a single instance of java.lang.Class. It is therefore possible to determine if two objects, `x` and `y` belong to the same class by performing the test:

```
x.getClass() == y.getClass()
```

Alternatively, class membership may be tested by comparing class names:

```
x.getClass().getName().equals(y.getClass().getName())
```

(Recall that == compares object references, whereas `equals()` compares object values.)

Steps

1. Identify the clients performing explicit type checks.
2. Add a new, empty method to the root of the provider hierarchy representing the action performed in the conditional code.
3. Iteratively move a case of the conditional to some provider class, replacing it with a call to that method. After each move, the regression tests should run.
4. When all methods have been moved, each case of the conditional consists of a call to the new method, so replace the entire conditional by a single call to the new method.
5. Consider making the method abstract in the provider's root. Alternatively implement suitable default behaviour here.

Other Steps to Consider.

- If the provider hierarchy is not a real inheritance hierarchy, you must transform it first.
- It may well be that multiple clients are performing exactly the same test and taking the same actions. In this case, the duplicated code can be replaced by a single method call after one of the clients has been transformed. If clients are performing different tests or taking different actions, then the pattern must be applied once for each conditional.
- If the case statement does not cover all the concrete classes of the provider hierarchy, a new abstract class may need to be introduced as a common superclass of the concerned classes. The new method will then be introduced only for the relevant subtree. Alternatively, if it is not possible to introduce such an abstract class given the existing inheritance hierarchy, consider implementing the method at the root with either an empty default implementation, or one that raises an exception if it is called for an inappropriate class.
- If the conditionals are nested, the pattern may need to be applied recursively.

Tradeoffs

Normally the instances of the correct classes should be already created so we do not have to look for the creation of the instances, however 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. In case of multiple clients, *Double Dispatch* can be an aid for the migration.

When type checks are needed

Contrary to Transform Self Conditional to Subclassing where type checks are sometimes justified, the only time where type checks over provider type information is needed is when the code of the provider is frozen and may not be extended.

Example

The code in figure 3 illustrates misplaced responsibilities since the client must explicitly type-check instances of Telephone to determine what action to perform..

```
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(....);
        }
    }
```

Figure 3 Explicit type checks in client code.

After applying the pattern the client code will look like this:

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

Rationale

Riel states, "Explicit case analysis on the type of an object is usually an error. The designer should use polymorphism in most of these cases" [Riel96a]. Indeed, explicit type checks in clients are a sign of misplaced responsibilities since they increase coupling between clients and providers. Shifting these responsibilities to the provider will have the following consequences:

- The client and the provider will be more weakly coupled since the client will only need to explicitly know the root of the provider hierarchy instead of all of its concrete subclasses.
- The provider hierarchy may evolve more gracefully, with less chance of breaking client code.
- The size and complexity of client code is reduced. The collaborations between clients and providers become more abstract.
- Abstractions implicit in the old design (i.e., the actions of the conditional cases) will be made explicit as methods, and will be available to other clients.
- Code duplication may be reduced (if the same conditionals occur multiply).

Related Patterns

InTransform Client Conditional to Polymorphism the conditional is made on the type information of a provider class. The same situation occurs in Apply Null Object where the conditional tests over null value before invoking the methods. From this point of view, Apply Null Object is a specialization of Transform Client Conditional to Polymorphism.

Replace Conditional with Polymorphism is the core refactoring of this reengineering pattern, so the reader may refer to the steps described in [Fowl99a].

Known Uses

This pattern has been applied in one of the Famoos case studies written in Ada. This considerably decreased the size of the application and improved the flexibility of the software. In one of the Famoos C++ case studies, explicit type checks were also implemented statically by means of preprocessor commands (# ifdefs).

Apply State

Like Transform Self Conditional to Subclassing*, transform complex conditional code that tests over quantified states into delegated calls to state classes. So we apply the* State *pattern, delegating each conditional case to a separate State object.*

We invite the reader to read the *State* and *State Patterns* for a deep description of the problem and discussion [Gamm, Alpe98a, Dyso98a]. Here we only focus on the reengineering aspects of the pattern.

Problem

It is hard to extend a class because you have to modify all its methods that perform conditional checks on its states to decide what actions to perform.

Context

You want to add a new behavior to a class. You have access the class source code.

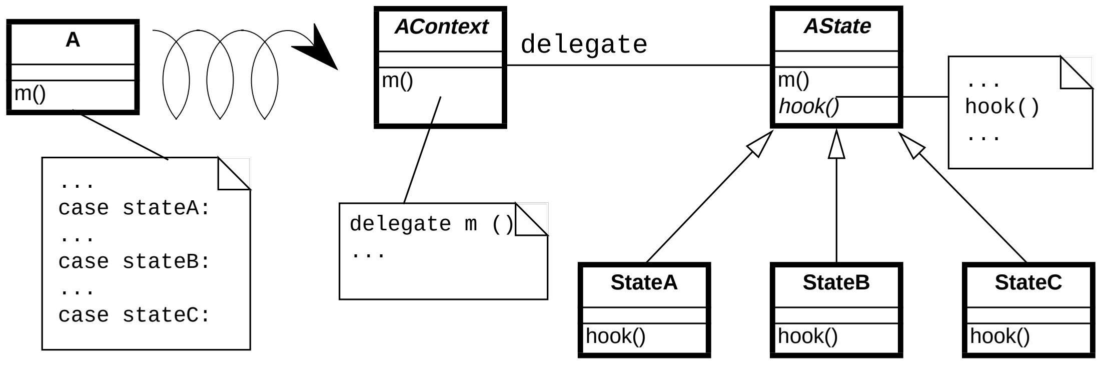
Symptoms

- Duplication of the same tests based on object state description in several methods of the object.
- New states cannot be added without having to modify all the methods containing the object state tests.

Solution

Apply the *State* pattern, i.e. encapsulate the state dependent behavior into separate objects, delegate calls to these objects and keep the state of the object consistent by refering to the right instance of these state objects.

Structure/Participants



Steps

1. Identify the interface of a state and the number of states.
2. Create a new abstract class, State, representing the interface of the state.
3. Create a new class subclass of State for each state.
4. Define methods of the interface identified in Step 1 in each of the state classes by copying the leaf of the test in the method. Pay attention to change the state of the instance variable in the Context to refer to the right instance of State class.
5. Add a new instance variable in the Context class.
6. You may have to have a reference from the State to the Context class to invoke the state transitions from the State classes.
7. Initialise the newly created instance to refer to a default state class instance.
8. Change the methods of the Context class containing the tests to delegate the call to the instance variable.

The step 4 can be done using the Extract Method of the Refactoring Browser. Note that the order of the steps are different from the ones of [Alpe98a] because we choose to apply the transformation in a way that let the system always runnable and testabel using unit tests.

TradeOffs

Pros

- Limited Impact.The public interface of the original class does not have to change. Since the state instances are accessed by delegation from the original object, the clients are unaffected. In the straightforward case the application of this pattern has a limited impact on the clients.

Cons

- Class explosion. The systematic application of this pattern may lead to a class explosion.

When not to apply

- If the number of states are not fixed or too long.
- If the transitions between states is not clear.

When not to apply? When the legacy solution is ok!

When the states are clearly identified and it is known that they will not be changed, the legacy solution is a solution that has the advantage of grouping all the state behaviour by functionality instead of spreading it over different subclasses.

Example

The Design Patterns Smalltalk Companion presents a code transformation steps by steps [Alpe98a].

Apply Null Object

Transform conditional code that tests over null values into a polymorphic call to method of a NullObject. Shift the responsibility for deciding what to do to the provider hierarchy by introducing a special Null object. [Wool98a]

We invite the reader to read the *NullObject* pattern for a deep description of the problem and discussion [Wool98a]. Here we only focus on the reengineering aspects of the pattern.

Problem

You are repeatly checking for null values before sending message.

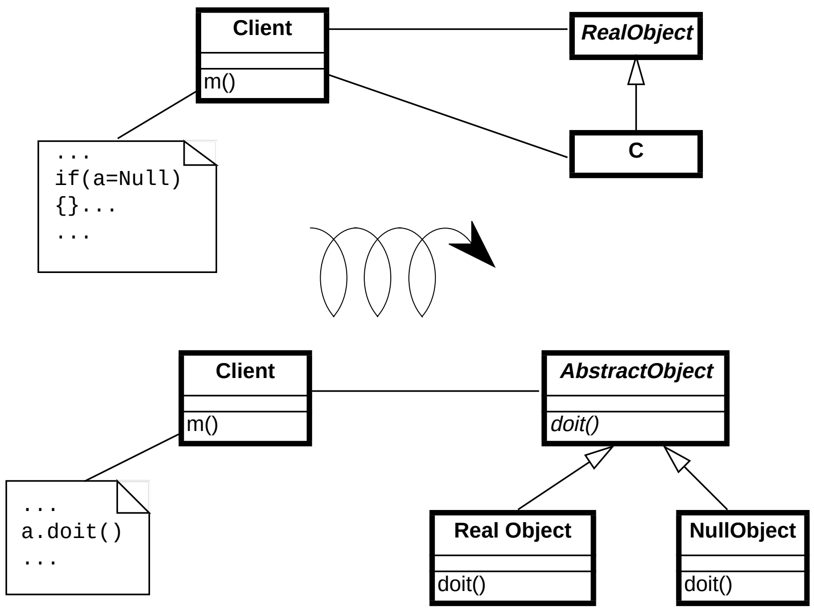
Symptoms

- Client methods are always testing that certain values are not null before actually invoking their methods.
- Adding a new subclass to the client hierarchy requires testing null values before invoking some of the provider methods.

Solution

Apply the *NullObject* pattern, i.e. encapsulate the null behaviour as a separate provider class so that the client class does not have to perform a null test.

Structure/Participants



Detection

Look for idiomatic null tests.

Steps

1. Identify the interface required for the null behaviour.
2. Create a new abstract superclass as a superclass of the RealObject class.
3. Create a new subclass of the abstract superclass with a name starting with No or Null.
4. Define default methods into the Null Object class.
5. Initialise the instance variable or structure that was checked to now hold at least an instance of the Null Object class.
6. Remove the conditional tests form the client.

If you want to be able to still be able to make some conditional over null values in a clean way, you may introduce in RealObject and Null Object classes a query method isNull as described in Introduce Null Object [Fowl99a].

Tradeoffs

Pros

- As the client normally just checks whether it can invoke some methods of the provider, the interface of the provider class does not have to be modified when applying *NullObject*. Contrary to other patterns like Transform Client Conditional to Polymorphism where the interface of the provider may change considerably to propose a coherent interface to the clients, the application of the *NullObject* pattern has a limited impact.

Cons

- The application of *NullObject* can lead to a class explosion, indeed for every realObject class, three classes are created, RealObject, NullObject and AbstractObject. However, several techniques exist to circumvent this problem, such as implementing the null object as a special instance of RealObject rather than as a subclass of AbstractObject. Read *NullObject* for deeper explanations.

Difficulties: Multiple Clients

- If several clients have the same notion of default behaviour and share the same interface they can be treated independently of each other. However, one of the difficulties that may arise when applying this pattern is the fact that several clients may have a different notion of default behaviour. If the different clients do not agree on the common behaviour but agree on a common interface, one possibility is to have a palatable Null Object in which each client may specify its desired default behaviour.

When not to apply? When the legacy solution is ok!

- If clients do not agree on the same interface.
- When very little code uses the variable directly or when the code that use the variable is well-encapsulated in a single place.

Example

The following example code is taken from [Wool98a]. The original code is the following one:

```
VisualPart>>objectWantedControl
    ...
    ^ctrl isNil
        ifFalse:
            [ctrl isControlWanted
                ifTrue:[self]
                ifFalse:[nil]]
```

It is then transformed into :

```
VisualPart>>objectWantedControl
    ...
    ^ctrl isControlWanted
                ifTrue:[self]
                ifFalse:[nil]
Controller>>isControlWanted
    ^self viewHasCursor
NoController>>isControlWanted
    ^false
```

External Pattern Thumbnails

Replace Type Code with Subclasses

Provides a recipe for carrying out the refactorings required for Transform Self Conditional to Subclassing *[Fowl99a].*

Replace Conditional with Polymorphism

Double Dispatch

Deprecation

Replace Type Code with State

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [Gamm95a]

Refactoring To Specialize

W. Opdyke [Opdy92b] proposed using class invariants as a criterion to simplify conditionals.

NullObject

A Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, the Null Object encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators [Wool98a].

Introduce Null Object

Provides a recipe for carrying out the refactorings required for Apply Null Object *[Fowl99a].*

State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class [Gamm95a].

State Patterns

The State Patterns pattern language refines and clarifies the State Pattern [Dyso98a].

References

[Alpe98a], Sherman R. Alpert, Kyle Brown and Bobby Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998.

[Gamm95a], Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,*Design Patterns*, Addison Wesley, 1995.

[Deme99a], Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, *A Pattern Language for Reverse Engineering, Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999*, Paul Dyson (Ed.), UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany, July 1999.

[Duca99a],Stéphane Ducasse, Tamar Richner and Robb Nebbe, *Type-Check Elimination: Two Object-Oriented Reengineering Patterns*, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*, Francoise Balmas, Mike Blaha and Spencer Rugaber (Eds.), IEEE, October 1999.

[Fowl99a],Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Opdy92a],William F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois, 1992.

[Riel96a],Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[Wool98a], Bobby Woolf, *Null Object*, *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle and Frank Bushmann (Eds.), pp. 5-18, Addison-Wesley, 1998.