Transforming Conditionals

Oscar Nierstrasz and Stéphane Ducasse
Software Composition Group, University of Berne[1]

Abstract. We describe several related reengineering patterns concerned with removing conditionals from object-oriented code. *Transform Conditional to Subclassing* makes a class more extensible by transforming complex conditional code that tests immutable state into a single polymorphic call to an operation which will be implemented by a different subclass for each case of the conditional. *Transform Conditional to Polymorphism* transforms conditional code that tests the type of an object into a polymorphic call to a new method, thereby reducing client/provider coupling.

Transform Conditional to Subclassing

Make a class more extensible by transforming complex conditional code that tests immutable state into a single polymorphic call to an operation which 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. You have access to the source code of the class *and of clients that instantiate it.*

Symptoms

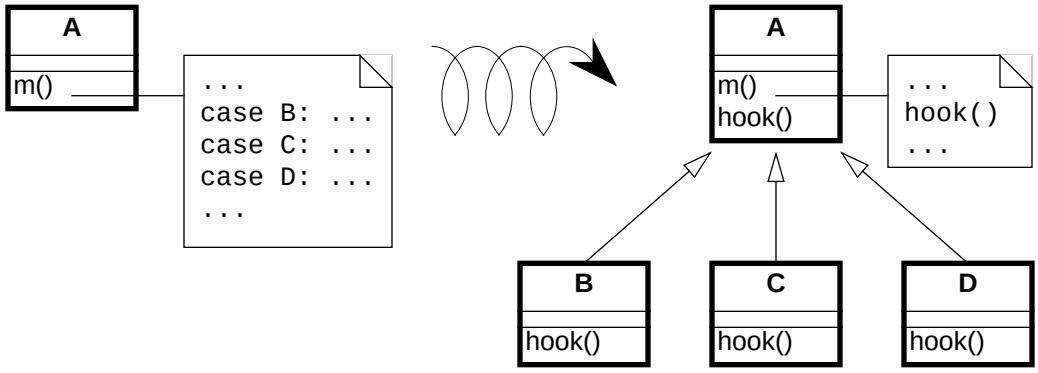
- The class you want to modify has long methods with complex conditional branches.
- Instances of the class seem to represent multiple data types each with different behaviour.
- The expression being tested in the conditional represents type information.
- Conceptually simple extensions require many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

1. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965.
E-mail: {oscar, ducasse}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~scg.

Solution

Associate each case of the conditional with a new subclass of the original class. Replace the conditional code with a call to a new hook method. In each of the new subclasses, implement the hook method with the code corresponding to that case in the original case statement.

Structure/Participants



Detection

- 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.
- 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 high incidence 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

- 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.
- 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.
- 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.
- Iteratively move the cases of the conditionals down to the corresponding subclasses, periodically running the tests.
- The methods that contain conditional code should now all be empty. Replace these by abstract methods and run the tests.
- Alternatively, if there are suitable default behaviours, implement these at the root of the new hierarchy.
- 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

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.

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.

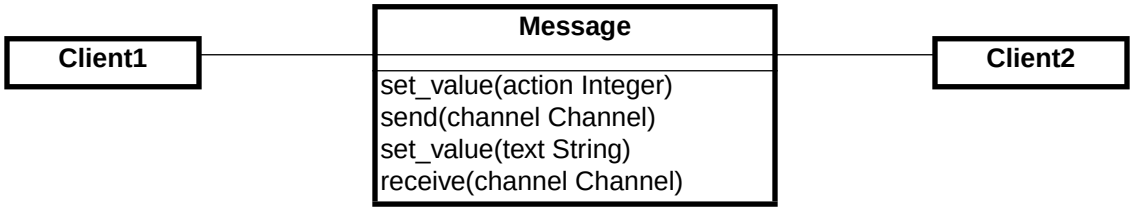
If the case statements test more than one attribute, it may be necessary to support a more complex hierarchy, possibly requiring multiple inheritance. Considering splitting the class into parts, each with its own hierarchy.

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.

If the conditional code tests *mutable* state of the object, consider instead applying Transform Conditional to Polymorphism. Otherwise, if state of other objects is tested, such as arguments to the method, then consider applying Transform Conditional to Polymorphism.

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

```
Message::send(Channel c) { ...
    switch (type_) {
    case TEXT:
        ...
    case ACTION:
        ...
    }; ...
}
Client1::doit() { ...
    Message myMessage =
        new Message();
    myMessage.setValue("...");
    ...
}
```

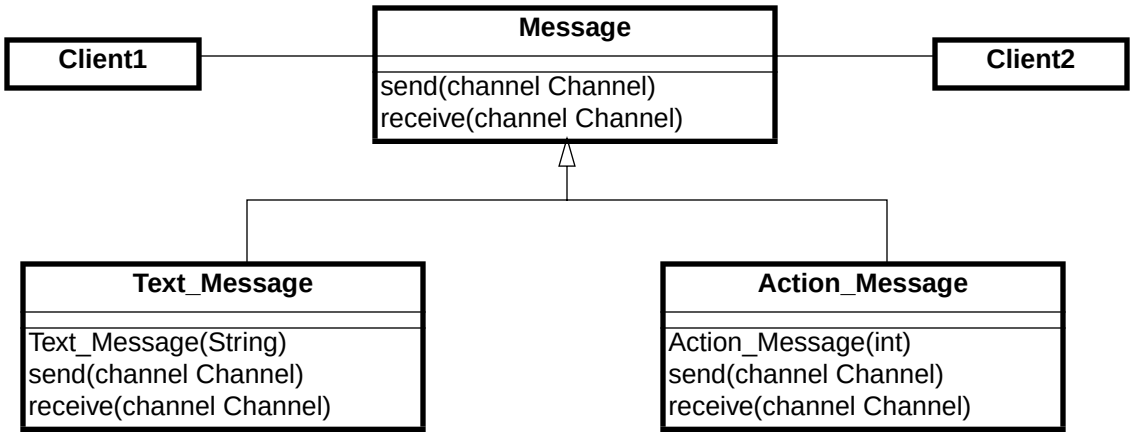
Figure 1 Initial design and source code.

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;
...
};

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 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.

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

Replace Type Code with Subclasses, Refactoring To Specialize.

Transform Conditional to State

Like Transform Conditional to Subclassing *but the attributes we switch on are not constant, so the solution applies the State design pattern.*

Related Patterns

Replace Type Code with State.

Transform Conditional to Polymorphism

Transform conditional code that tests the type of an 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.

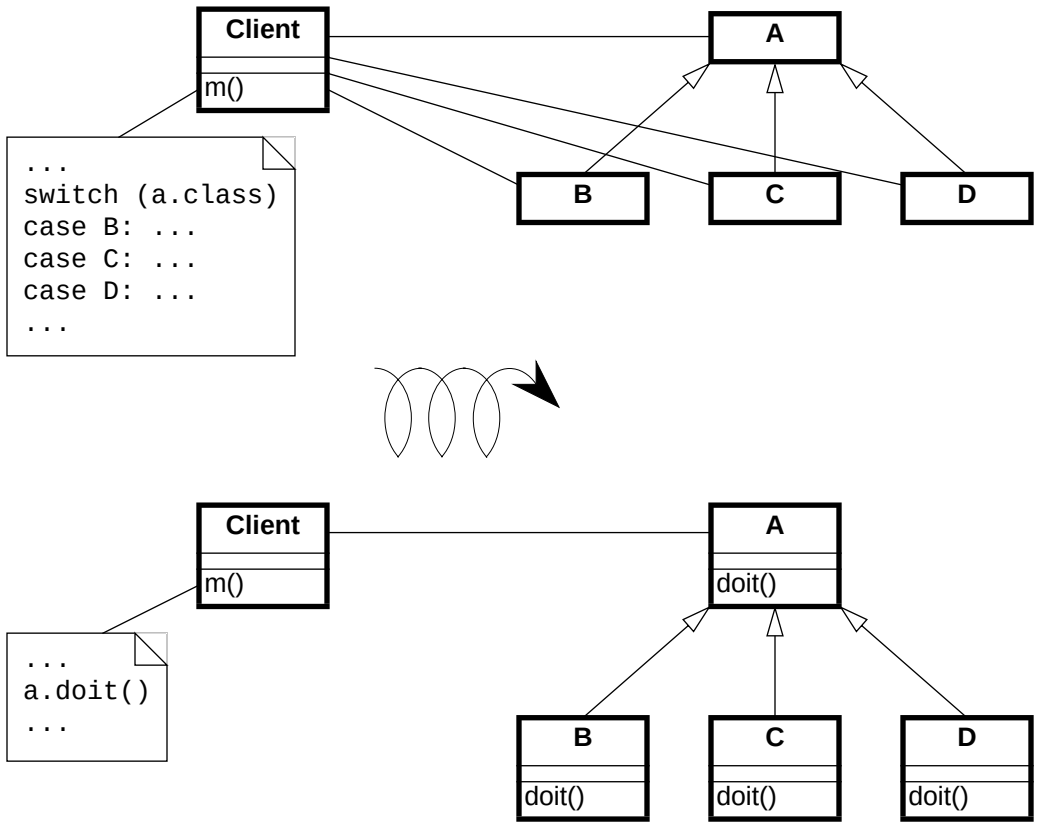
Symptoms

- 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.

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 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 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

- Identify the clients performing explicit type checks.
- Add a new, empty method to the root of the provider hierarchy representing the action performed in the conditional code.
- 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.
- 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.
- Consider making the method abstract in the provider's root. Alternatively implement suitable default behaviour here.

Tradeoffs

The pattern cannot be applied if you don't have access to the client source code.

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.

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.

If the conditionals are nested, the pattern may need to be applied recursively.

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

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

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" [5].

Transform Conditional to Subclassing, *Replace Conditional with Polymorphism*.

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).

Null Object

Like Transform Conditional to Polymorphism *except that the attribute we are switching is whether the object reference is null or not. Shift the responsibility for deciding what to do to the provider hierarchy by introducing a special Null object [6].*

Replace Type Code with Subclasses

Provides a recipe for carrying out the refactorings required for Transform Conditional to Subclassing *[2].*

Replace Conditional with Polymorphism

Provides a recipe for carrying out the refactorings required for Transform Conditional to Polymorphism *[2].*

Replace Type Code with State

Provides a recipe for carrying out the refactorings required for Transform Conditional to State *[2].*

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 [3].

Refactoring To Specialize

W. Opdyke [4] proposed using class invariants as a criterion to simplify conditionals.

Acknowledgements

The material presented here is based loosely on an earlier published paper [1]. Many thanks to members of the Software Composition Group who workshopped drafts of this paper.

[1] Stéphane Ducasse, Robb Nebbe and Tamar Richner, "Two Reengineering Patterns: Eliminating Type Checking," *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999*, Paul Dyson (Ed.), UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany, July 1998.

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

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

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

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

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