Transform Conditionals to Polymorphism[1]

Stéphane Ducasse(+), Oscar Nierstrasz(+), Serge Demeyer(*)
(+) University of Berne - SCG - http://www.iam.unibe.ch/~scg/
(*) University of Antwerp - LORE - http://win-www.uia.ac.be/u/sdemey/

Abstract. Conditionals —i.e., switch statements, nested ifs— that are used to simulate polymorphism hamper evolution and flexibility of applications. The reengineering patterns presented in this paper show you how to transform conditionals in object-oriented code to improve the flexibility of application.

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 (FAMOOS).

A revised version of these patterns is published in: Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002. www.iam.unibe.ch/~scg/OORP

Introduction

Legacy systems are not limited to the procedural paradigm and languages like Cobol. Even if object-oriented paradigm promised the building of more flexible systems 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 [Deme99n] 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 [Duca99c].

[1] In Proceedings of EuroPLoP'2000, UVK GmbH, pp. 219-252.

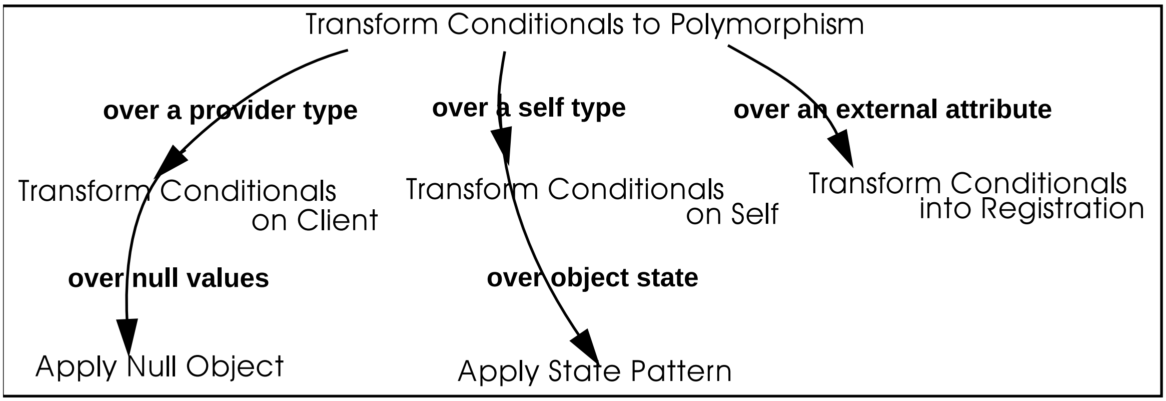

Transform Conditionals to Polymorphism
over a provider type
over a self type
over an external attribute
Transform Conditionals on Client
Transform Conditionals on Self
Transform Conditionals into Registration
over null values
over object state
Apply Null Object
Apply State Pattern

Figure 1 Relationships between the patterns constituing Transform Conditionals to Polymorphism.

Transform Conditionals to Polymorphism is a pattern language describing how conditionals —i.e nested tests, switch statements— are transformed into code that is more flexible and exhibits less coupling between classes. This pattern language consists of five patterns,Transform Conditionals on Self, Transform Conditionals on Client, Apply State, Apply Null Object and Transform Conditionals into Registration. 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 descriptions.

Figure 1 summarizes the relations and the differences between the patterns.

- Transform Conditionals on Self eliminates conditionals 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.
- Transform Conditionals on Client transforms conditionals over type information in a client class by introducing polymorphic methods in the provider and calling them from the client class.
- Apply State is a special case of Transform Conditionals on Self 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.
- Apply Null Object is a special case of Transform Conditionals on Client 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 checks for a null value is transformed by creating a *NullObject* class that performs the default behavior, liberating the client from having to type check before performing an operation.
- Transform Conditionals into Registration eliminates conditionals over an external value in users of certain tools. The solution is based on the introduction of a registration mechanism where each tool must register itself and the definition of a clear interaction protocol between the registrees and their users. The solution is then fully dynamic because new tools can be added or removed without any changes in the tool users.

Cost, Clarity and Maintainability

Transform Conditionals on Self and Transform Conditionals on Client transform conditionals based on explicit type checks into polymorphic calls. Contrary to most of the Design Patterns [Gamm95a] that introduce an extra indirection, Transform Conditionals on Self and Transform Conditionals on Client only use the semantic support, i.e., dynamic dispatch and late binding, offered by the language instead of simulating them in ad-hoc ways. However, we may legitimately ask if using conditionals instead of polymorphic calls does not come at a cost.

The general answer is that if you must pay attention not to use polymorphic calls you may do not gain the full power of object-orientation and a good use of a procedural language like C might suit you better. In such a case, you will have to live with code that might be more difficult to understand and maintain.

The detailed answer depends on the language. In Smalltalk and Java, all methods are late bound or polymorphic (except private methods in Java that are statically bound), so using polymorphic calls may be even faster than using conditionals because the virtual machine does the receiver type check and the method lookup. For these languages, the difference in speed can strongly depend on the VM and the optimization technology used, e.g., just-in-time compiler, native code generation. In C++, the fact that you can have virtual or statically bound methods may be crucial for you.

Some people may argue that certain compilers like SmallEiffel convert polymorphic calls into conditionals and that conditionals are faster and better than polymorphic calls. From a technical side it should be noticed that with polymorphic methods, the number of classes will not impact the performance whereas with nested conditionals the more classes that have to be type-checked, the more penalty you get. Then, although it is really justified for compilers to transform the code into faster forms — this is mainly why they exist, still this is not a justification to code like a compiler. The code we write is intended for developers, that’s why it should be readable, support abstraction and be more maintainable. The patterns presented in Transform Conditionals to Polymorphism improve such properties.

Why the legacy solution may have been applied?

Using conditionals instead of polymorphic calls 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 behavior 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 Conditionals on Self

Intent: 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 behaviors depending on the value of some immutable attribute.

Improving the design of such a class is difficult because:

> On the one hand, this is handy to have a global view of all the possible behaviors of the class without having to deal with multiple abstractions. All the logic is grouped in a single location. Conditional statements have the value that they provide a locality of reference for the human programmer to understand the conditional behavior flow.
>
> On the other hand, all the behaviors are mixed together leading to a more complex system to understand and modify.

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 behavior.
- The expression being tested in the conditional represents type information over the class containing the expression itself.
- The behavior 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.
- Adding a new behavior requires to modify the same set of methods and to add a new condition test into them.

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

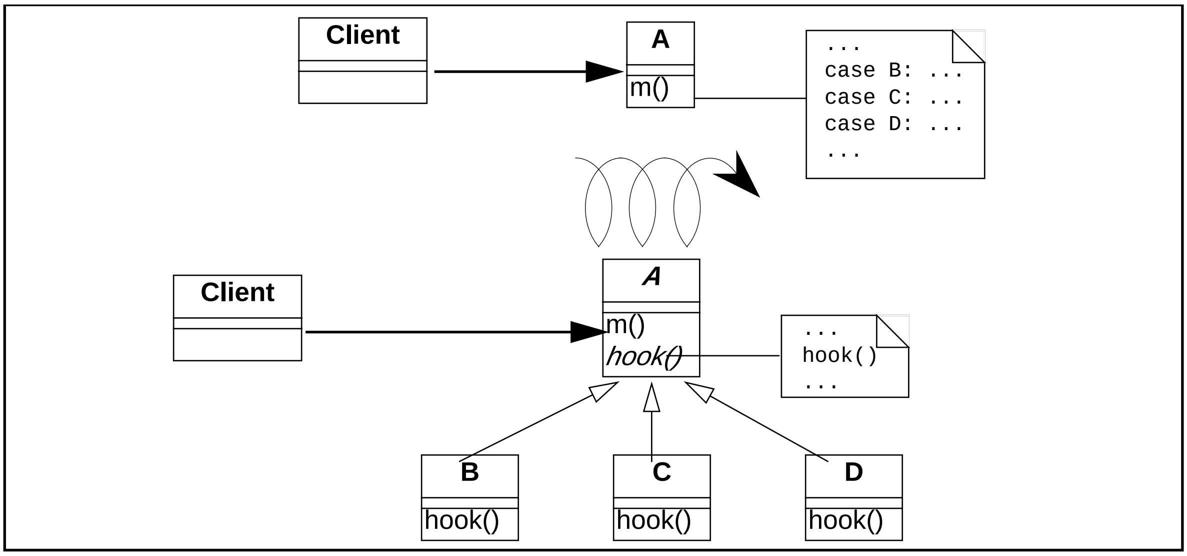


Figure 2 Transformation of explicit type check into self polymorphic method calls.

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 behavior 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 behavior 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 behavior 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 behaviors, 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

- New behaviors can now be added in a incremental manner, without having to change a set of methods of a single class containing all the behavior. A specific behavior can now be understood independently from the other variations.
- A new behavior represents its data independently from the other ones thus minimizing the possible interference and increasing the understandability of the separated behaviors.
- All behaviors now shares a common interface so helping in their understanding.

Cons

- All the behaviors are now dispersed into multiple but related abstractions, so getting an overview of the behavior may be more difficult. However, the concepts are related and share the interface represented by the abstract class reducing then the problem.
- 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.
- Explicit type checks are not always a problem and we can tolerate them. In particular they may be an alternative to the creation of new classes when:
  - ◊ the set over which the method selection is fixed and will not evolve in the future, and
  - ◊ the typecheck is only made in one place.

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 class containing the original conditionals cannot be subclassed, Transform Conditionals on Self can be composed with delegation. The idea to use the polymorphism on another hierarchy, by moving part of the state and behavior of the original class into a separate class to which the method will delegate as shown in Figure 3.

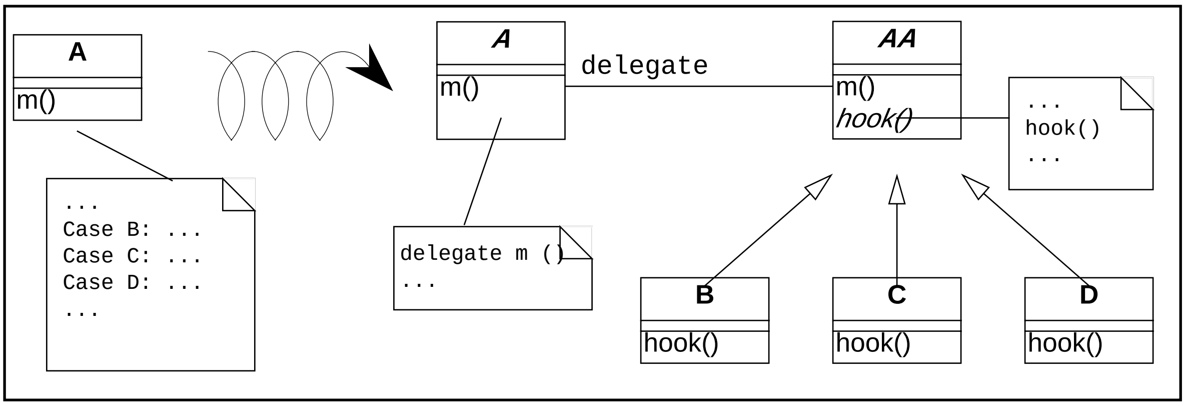


Figure 3 Combining simple delegation and Transform Conditionals on Self when the class cannot be subclassed.

When the legacy solution is the solution

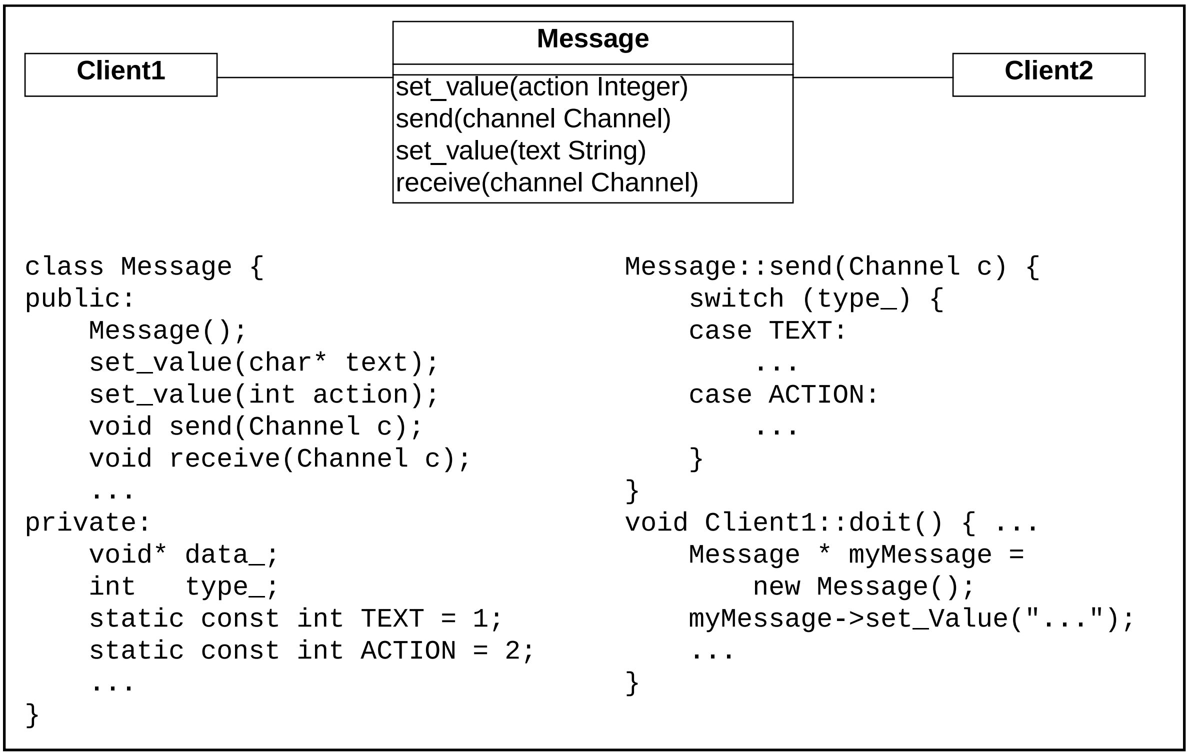
- Explicit type checks cannot always be avoided. One of the few good reasons to use type check instead of polymorphism is when polymorphism cannot be used! Indeed when the code is dealing with the limits of the paradigm like using non object-oriented libraries or when streaming in objects from files. For example 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.

Example

The example comes from one of the application we analyzed. In this application, several messsages can be sent to a complex system. These messages are represented by the class `Message` and can be of different types.

Before

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 as shown in Figure 4.



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

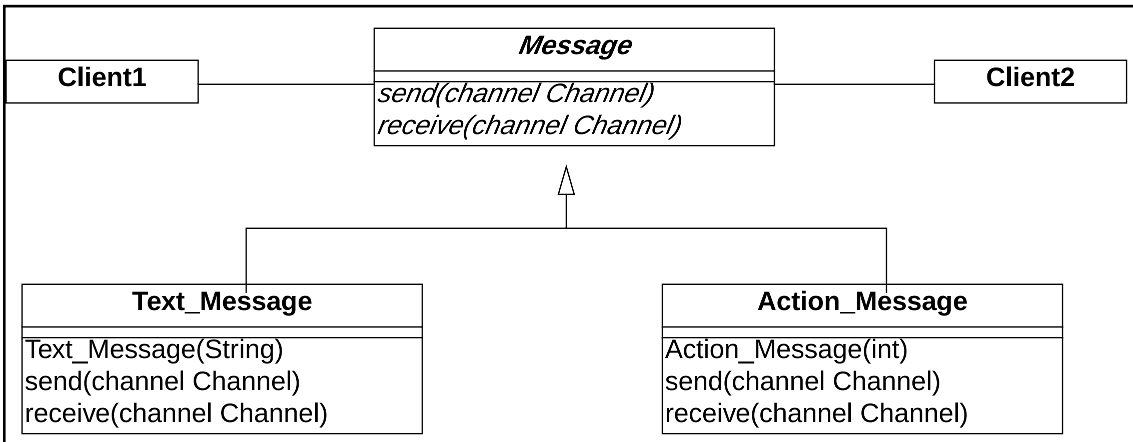
Figure 4 Initial design and source code.

After

Since Message conceptually implements two different classes, Text_Message and Action_Message, we introduce these as subclasses of Message, as shown by Figure 5. 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.



```
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 5 Resulting hierarchy and source code.

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.

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 behaviors. Introducing new behavior then requires changes to be made to all such methods instead of simply specifying one new class representing the new behavior.

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 Conditionals on Self 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 Conditionals on Self 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 Conditionals on Client or Transform Conditionals into Registration the conditional expressions are used to invoke methods not of the class itself but of provider classes.

- If the conditional code tests external value identifying the client methods to be invoked, consider applying Transform Conditionals into Registration.
- If the conditional code tests *mutable* state of the object, consider instead applying Transform Conditionals on Client.

Transform Conditionals on Client

Intent: 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.

Symptoms

- Clients 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 as shown in Figure 6.

Note that the different providers do have to necessary inherit from a common ancestor. In such a case the solution is to ensure that all the providers implement a common interface that any client can use.

Structure/Participants

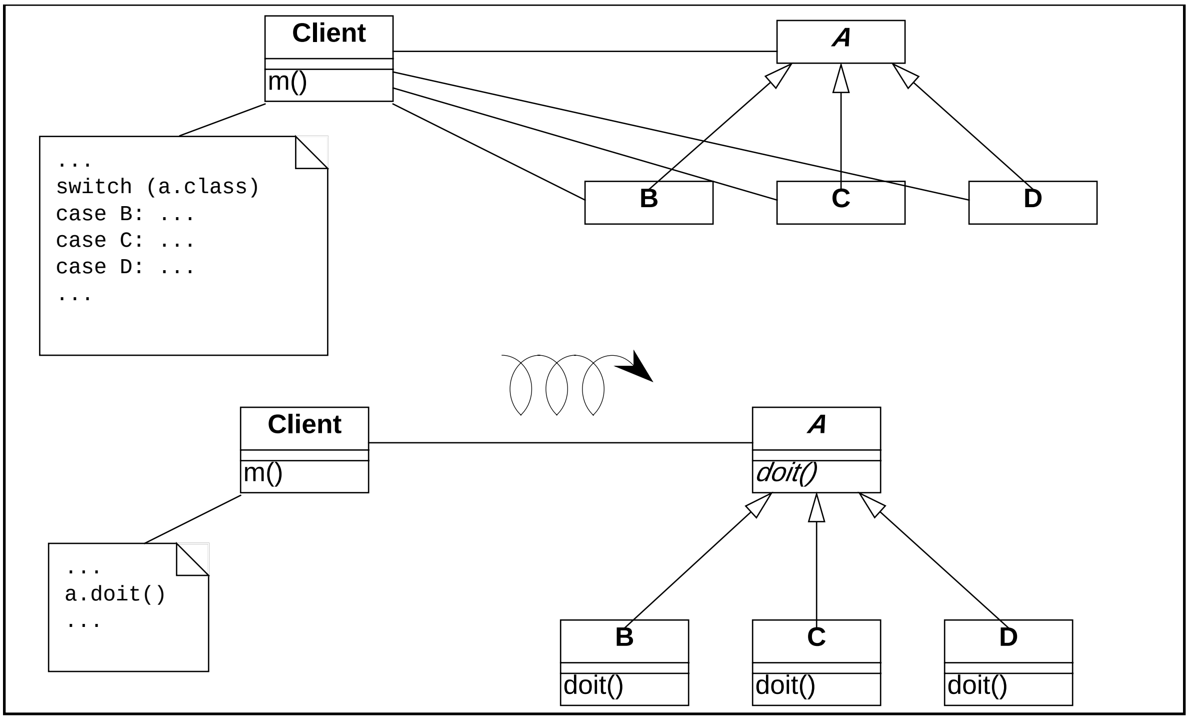


Figure 6 Transformation of explicit type check used to determine which methods of a client should be invoked into polymorphic method calls.

Detection

Apply essentially the same techniques described in Transform Conditionals on Self 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 Conditionals on Self, 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 behavior here.

Other Steps to Consider.

- If the provider is only one single class, you must transform it first into a class hierarchy:
  - ◊ identify different subclasses in the class,
  - ◊ split the class into different subclasses by moving down attributes and associated functionality.

  The way the clients use different interface parts of the original class may help you to identify subclasses.
- 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

Pros

- The code of the clients is now better organized and does not have to deal anymore with some of concerns that are now under the responsibility of the provider.

- The fact that the provider offers a polymorphic interface allows the modification of the provider without impacting or with limited impact on the client providers.

Cons

The cons of applying this pattern are the same as the ones of Transform Conditionals on Self.

- 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.
- Explicit type checks are not always a problem and we can tolerated them. In particular they may be an alternative to the creation of new 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.

Difficulties

- 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 the legacy solution is the solution

Contrary to Transform Conditionals on Self 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

Before

The following code illustrates misplaced responsibilities since the client must explicitly typecheck instances of `Telephone` to determine what action to perform. The bold code indicates problems of the solution.

```
class Telephone {
public:
  enum PhoneType { POTSPHONE, ISDNPHONE, OPERATORPHONE };
  Telephone() {}
  PhoneType phoneType() { return myType; }

private:
  PhoneType myType;
protected:
  void setPhoneType(PhoneType newType) { myType = newType; }
};

class POTSPhone : public Telephone {
```

```
public:
  POTSPhone() { setPhoneType(POTSPHONE); }
  void tourneManivelle();
  void call();
};
...

class ISDNPhone: public Telephone {
public:
  ISDNPhone() { setPhoneType(ISDNPHONE);}
  void initializeLine();
  void connect();
};
...

class OperatorPhone: public Telephone {
public:
  OperatorPhone() { setPhoneType(OPERATORPHONE); }
  void operatorMode(bool onOffToggle);
  void call();
};

void initiateCalls(Telephone ** phoneArray, int numOfCalls) {
  for(int i = 0; i<numOfCalls ;i++ ) {
    Telephone * p = phoneArray[i];

    switch(p->phoneType()) {
    case Telephone::POTSPHONE: {
      POTSPhone *potsp = (POTSPhone *) p;
      potsp->tourneManivelle();
      potsp->call();
      break;
    }
    case Telephone::ISDNPHONE: {
      ISDNPhone *isdnp = (ISDNPhone *) p;
      isdnp->initializeLine();
      isdnp->connect();
      break;
    }
    case Telephone::OPERATORPHONE: {
      OperatorPhone *opp = (OperatorPhone *) p;
      opp->operatorMode(true);
      opp->call();
      break;
    }
    default:  cerr << "Unrecognized Phonetype" << endl;
    };
  }
}
```

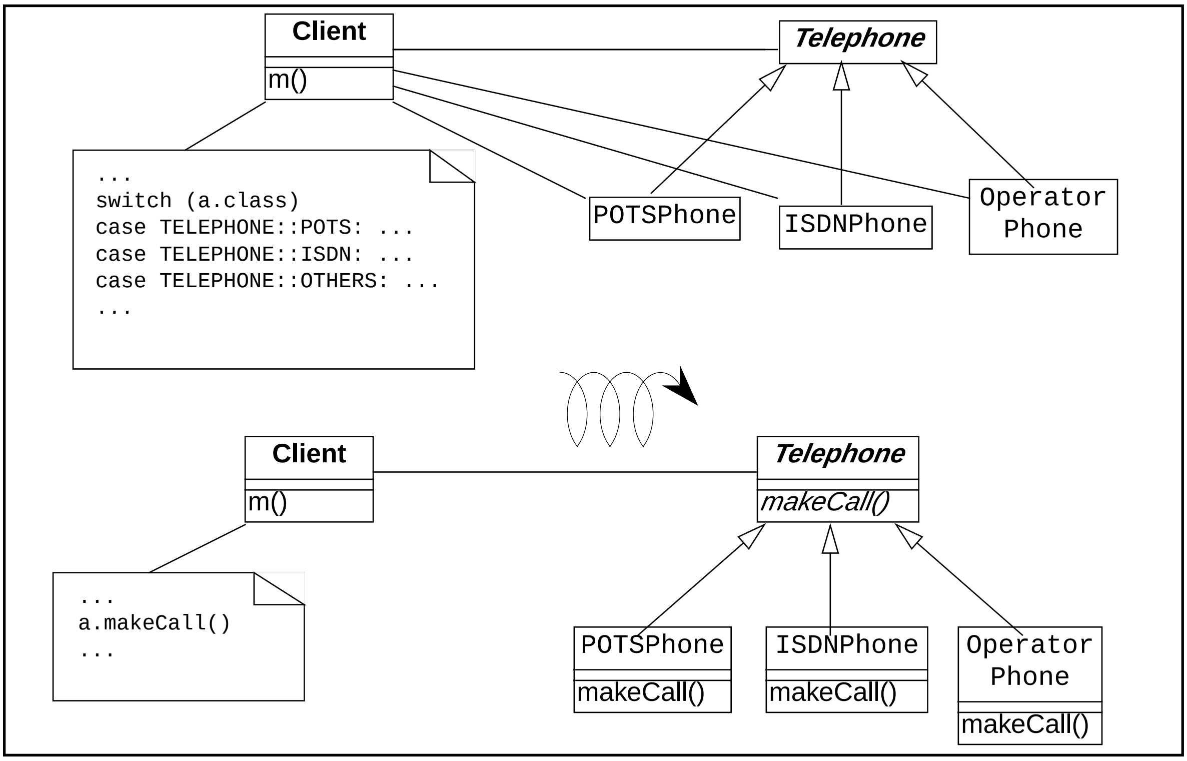


Figure 7 Transforming Telephone, its subclasses and its clients so that polymorphic methods are invoked instead of explicit type check use.

After

After applying the pattern the client code will look like the following. In bold we highlight the changes:

```
class Telephone {
public:
  Telephone() {}
  virtual void makeCall() = 0;
};

Class POTSPhone : public Telephone {
  void tourneManivelle();
  void call();
public:
  POTSPhone() {}
  void makeCall();
};
void POTSPhone::makeCall() {
  this->tourneManivelle();
  this->call();
}

class ISDNPhone: public Telephone {
  void initializeLine();
```

```
    void connect();

  public:
    ISDNPhone() { }
    void makeCall();
  };
  void ISDNPhone::makeCall() {
    this->initializeLine();
    this->connect();
  }

  class OperatorPhone: public Telephone {
    void operatorMode(bool onOffToggle);
    void call();
  public:
    OperatorPhone() { }
    void makeCall();
  };
  void OperatorPhone::makeCall() {
    this->operatorMode(true);
    this->call();
  }
  void initiateCalls(Telephone ** phoneArray, int numOfCalls) {
    for(int i = 0; i<numOfCalls ;i++ ) {
      phoneArray[i]->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 Conditionals on Client 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 Conditionals on Client.

Transform Conditionals into Registration is also based on use of conditionals to determine which methods should be called on which class. However, the conditional expression does not have to discriminate over a type but an expression like suffix files identifying the class. The solution is to let every provider register to a registration mechanism and make the client uses this registration mechanism to access registered tools using a established protocol.

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

Intent: Like Transform Conditionals on Self*, 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 [Gamm95a], [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.

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 referring to the right instance of these state objects (see Figure 8).

Structure/Participants

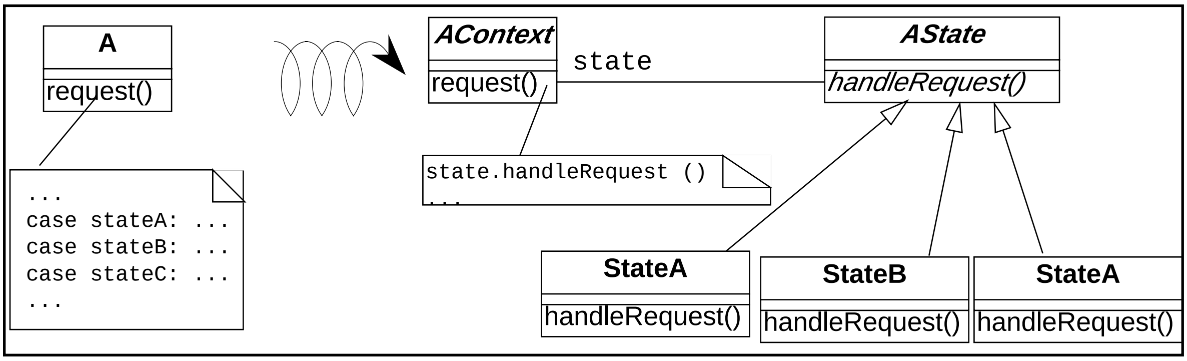


Figure 8 Transformation to go from a state pattern simulated using explicit state conditional to a situation where the state pattern has been applied.

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. Initialize 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 testable 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.
- This pattern should not be applied when:
  - ◊ the number of states are not fixed or too long, or
  - ◊ the transitions between states are not clear.

When the legacy solution is the solution.

- 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 behavior 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

Intent: 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 repeatedly 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 behavior as a separate provider class so that the client class does not have to perform a null test.

Structure/Participants

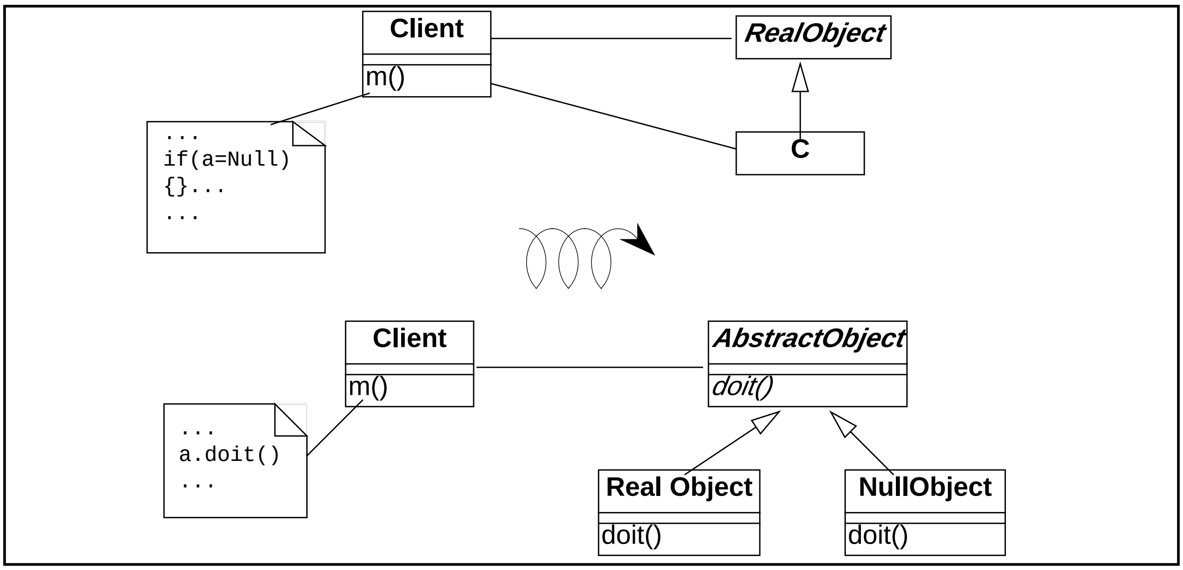


Figure 9 Transformation from a situation based on explicit test of null value to a situation where a NullObject is introduced.

Detection

Look for idiomatic null tests.

Steps

1. Identify the interface required for the null behavior.
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. Initialize 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 Conditionals on Client 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 behavior 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 behavior. If the different clients do not agree on the common behavior but agree on a common interface, one possibility is to have a palatable Null Object in which each client may specify its desired default behavior.

When the legacy solution is the solution

- 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
```

Transform Conditionals into Registration

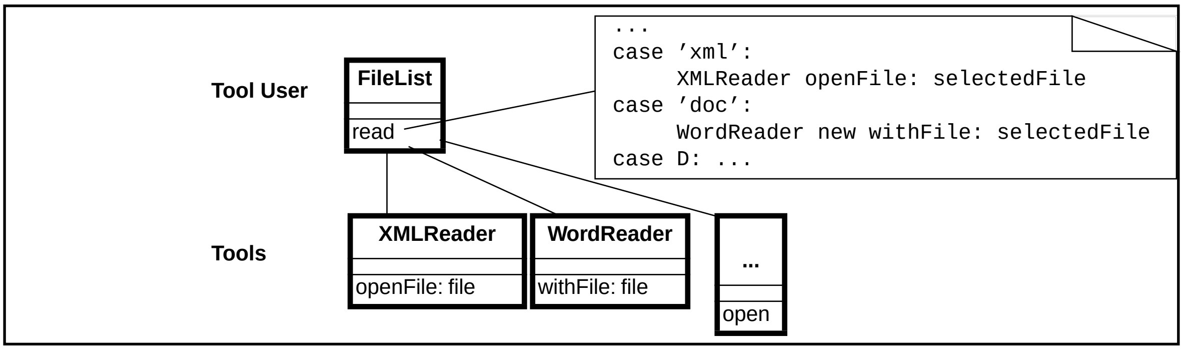
Intent: Intent: Increase flexibility between classes providing services and classes using them by transforming conditionals into a registration mechanism.

Problem

How can you reduce the coupling between tools providing services and tool users so that the addition or removal of tools does not lead to change the code of the tool users?

Symptoms

- Everytimes you remove certain functionalities from your system or a *tool*, you have to remove one case in some conditional statements, else certain parts (*tool users*) would still reflect the presence of the removed tools leading to fragile systems.
- Everytimes you add new functionality (i.e. for example importing different file formats like Flash, HTML, gif, JPEG), you have to add a new case in all the tool users that could use this new functionality.

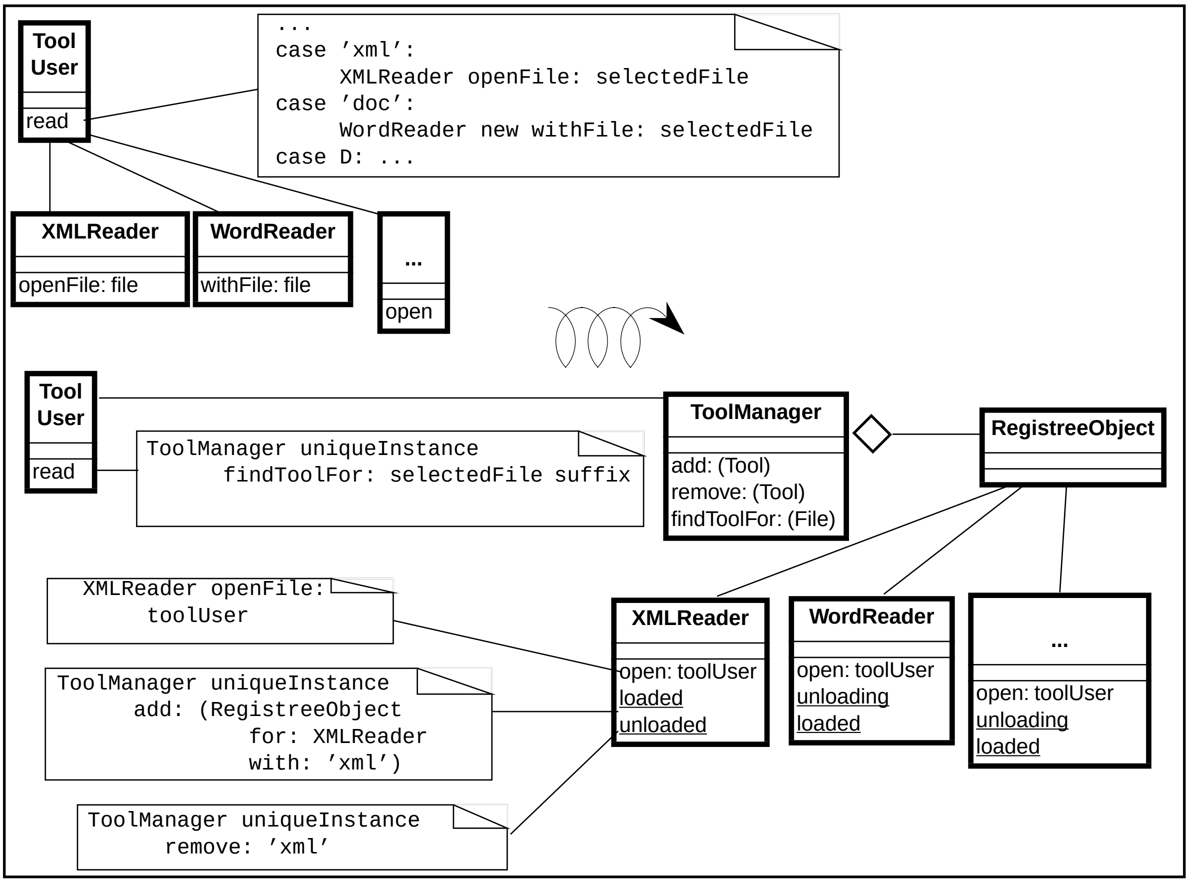


```
...
case ’xml’:
     XMLReader openFile: selectedFile
case ’doc’:
     WordReader new withFile: selectedFile
case D: ...
```

Figure 10 Tool Users use conditionals to determine which Tool should be invoked.

Solution

Replace conditional statements linking a set of classes providing services , (the *tools*), and the classes that used them , (the *tools users*), by making the tools *register* themselves to a registry mechanism and the tool users invoking the tool via the registry mechanism. (see Figure 11).



```
...
case ’xml’:
     XMLReader openFile: selectedFile
case ’doc’:
     WordReader new withFile: selectedFile
case D: ...
```

```
ToolManager uniqueInstance
      findToolFor: selectedFile suffix
```

```
XMLReader openFile:
      toolUser
```

```
ToolManager uniqueInstance
      add: (RegistreeObject
                for: XMLReader
                with: ’xml’)
```

```
ToolManager uniqueInstance
      remove: ’xml’
```

Figure 11 Transforming conditionals in tool users by introducing a registration mechanism and defining a clear protocol for communication between the tools and the tool users and for tool registration.

Detection

- Look for conditionals that dispatch on different values. The value identifies the tool(s) that have to be used.

Steps

1. Define a class representing *registree* objects, i.e. an object representing the necessary information for registering a tool. Although, the internal structure of this class depends on the purpose of the registration, a registree object should provide the necessary information so the tool manager can identify it, create instance of the represented tool and invoke methods.
2. Define a class (a *tool manager*) that manages the registree objects and that will be queried by the tool user to check the presence of the tools. This class is certainly a singleton as the registrees representing the tools available should not be lost if a new instance of the registree manager is created.

3. For each case of the conditional, define a registree object associated with a given tool. The creation of this object and its registration into the tool manager must be made automatically when the tool it refers to is loaded. In a similar manner the registree object should be unregistrered as soon as its associated tool is not available anymore.
4. Define a method or a similar mechanism that is invoked by the tool user when it needs to invoke the tool. To support the following step, a common protocol should be defined to which each registree objects (or tool should conform to depending on the mechanism used) to invoke a given tool.To pass information from the tool user to the tool, the current tool user can be passed as argument when the tool is invoked.
5. Transform the complete conditional expression into a query to the tool manager object. This query should return a tool associated to the query and invoke it to access the wished functionality.
6. If the tool user class defined methods for the activation of the tools, such methods are now been moved into the tool. These methods should be removed from the tool user class.

Example

The following example is extracted from Squeak. We slightly modified the original code to improve the example readibility. In Squeak, the FileList is a tool that allows one to load different kinds of files in the system like Smalltalk code, JPEG images, MIDI files, HTML.... Depending on the suffix of the selected file, the FileList proposes different actions to the user. We show in the example the loading of the different file depending on their format.

Before

The FileList implementation creates different menus items representing the different possibilities depending on the suffix of the files. The dynamic part of the menu is defined in the method `menusForFileEnding:` that requires a suffix as argument and returns a menu item containing the label of the menu item and the name of the corresponding method that should be invoked on the FileList object.

```
FileList>>menusForFileEnding: suffix

  (suffix = 'jpg') ifTrue:
    [^MenuItem label:'open image in a window'.
       selector: #openImageInWindow].
  (suffix = 'morph') ifTrue:
    [^MenuItem label: 'load as morph'.
      selector: #openMorphFromFile].
  (suffix = 'mid') ifTrue:
    [^MenuItem label: 'play midi file'.
      selector: #playMidiFile].
  (suffix = 'st') ifTrue:
    [^MenuItem label: 'fileIn'.
      selector: #fileInSelection].
  (suffix = 'swf') ifTrue:
    [^MenuItem label: 'open as Flash'.
```

```
        selector: #openAsFlash].
    (suffix = '3ds') ifTrue:
      [^MenuItem label: 'Open 3DS file'.
        selector: #open3DSFile].
    (suffix = 'wrl') ifTrue:
      [^MenuItem label: 'open in Wonderland'.
        selector: #openVRMLFile].
    (suffix = 'html') ifTrue:
      [^MenuItem label: 'open in html browser'.
        selector: #openInBrowser].
    (suffix = '*') ifTrue:
      [^MenuItem label: 'generate HTML'.
        selector:#renderFile].
```

The methods whose selectors are associated in the menu are implemented in the `FileList` class . We give two examples here. First the method checks if the tool it needs is available, if not it produces a beep, else the corresponding tool is created then used to treat the selected file.

```
FileList>>openInBrowser
  Smalltalk at: #Scamper ifAbsent: [^ self beep].
  Scamper openOnUrl: (directory url , fileName encodeForHTTP)

FileList>>openVRMLFile
  | scene |
  Smalltalk at: #Wonderland ifAbsent: [^ self beep].
  scene := Wonderland new.
  scene makeActorFromVRML: self fullName.
```

After

The solution is then to let every tool the responsibility to register themselves and let the FileList query the repository of available tools to find which tool can be invoked.

Step1

The solution is to first create the class `ToolRegistree` representing the registration of a given tool. Here we store the suffix files, the menu label and the action to be performed when the tools will be invoked.

```
Object subclass: #ToolRegistree
  instanceVariableNames: 'fileSuffix menuLabelName blockToOpen '
```

Step 2

Then the class `ToolsManager` is defined. It defines a structure to hold the registered tools and defines behavior to add, remove and find registered tool.

```
Object subclass: #ToolsManager
  instanceVariableNames: 'registrees '
```

```
ToolsManager>>initialize
  registree := OrderedCollection new.

ToolsManager>>addRegistree: aRegistree
  registrees add: aRegistree

ToolsManager>>removeRegistree: aBlock

  (registrees select: aBlock)
    do: [:each| registrees remove: each]

ToolsManager>>findToolFor: aSuffix
  "return a registree of a tool being able to treat file of format
   aSuffix"

  ^ registrees
      detect: [:each| each suffix = aSuffix]
      ifNone: [nil]
```

Note that the `findToolFor:` method could take a block to select which of the registree objects satisfying it and that it could return a list of registree representing all the tools currently able to treat a given file format.

Step 3

Then the tools should register themselves when they are loaded in memory. Here we present two registrations, showing that a registree object is created for each tool.As the tools need some information from the `FileList` object like the filename or the directory, the action that has to be performed take as parameter the instance of the `FileList` object that invokes it (`[:fileList |` in the code below).

In Squeak, when a class specifies a class (static) `initialize` method, this method is invoked once the class is loaded in memory. We then specialize the class methods `initialize` on the class `Scamper` and `Wonderland` to invoke the class methods `toolRegistration` define below:

```
Scamper class>>toolRegistration

  ToolsManager uniqueInstance
    addRegistree:
    (ToolsRegistry
        forFileSuffix: 'html'
        openingBlock:
          [:fileList |
          self openOnUrl:
            (fileList directory url ,
              fileList fileName encodeForHTTP)]
        menuLabelName: 'open in html browser')

Wonderland class>>toolRegistration

  ToolsManager uniqueInstance
```

```
        addRegistree:
        (ToolsRegistry
            forFileSuffix: 'wrl'
            openingBlock:
              [:fileList |
              | scene |
              scene := self new.
              scene makeActorFromVRML: fileList fullName]
            menuLabelName: 'open in Wonderland')
```

In Squeak, when a class is removed from the system, it receives the message `removeFromSystem`. Here we then specialize this method on every tool so that they unregister themselves.

```
Scamper class>>removeFromSystem

  super removeFromSystem.
  ToolsManager uniqueInstance
    removeRegistree: [:registree| registree forFileSuffix = ’html’]

Wonderland class>>removeFromSystem

  super removeFromSystem.
  ToolsManager uniqueInstance
    removeRegistree: [:registree| registree forFileSuffix = ’wrl’]
```

Step 4

The FileList object now has to use the `ToolsManager` to identify the right registree object depending on the suffix of the selected file. Then if a tool is available for the suffix, it creates a menu item specifying that the FileList has to be passed as argument of the action block associated with the tool. In the case where there is no tool a special menu is created whose action is to do nothing.

```
FileList>>itemsForFileEnding: suffix

  registree := ToolManager uniqueInstance
          findToolFor: suffix ifAbsent: [nil].
  ^ registree isNil
    ifFalse: [Menu label: (registree menuLabelName)
          actionBlock: (registree openingBlock)
          withParameter: self]
    ifTrue: [ErrorMenu new
          label: ’no tool available for the suffix ’, suffix]
```

Tradeoffs

Pros

- By applying Transform Conditionals into Registration you obtain a system which is dynamic, letting the responsibility to each tool to declare its presence. The proposed solution allow the transparent addition of new tools without implying any modification from the tool users.

- The actions that you moved from the tool user class to the associated tool may be much simpler because you do not have to test anymore that the right tool is available. The registration mechanism ensures you that the action can be performed.
- The interaction protocol between every tool and the tool user is now normalized.

Cons

- You have to define two new classes, one for the object representing tool registration and the object managing the registered tools.
- You may be forced to define a method in the tool for the action been invoked by the tool user. This way you are linking the tool with the tool user class in the tool class whereas the legay solution was linking them in the tool user. In Smalltalk, as shown by the example such a method can replaced by the definition of a block limiting the link between both classes. In Java, an inner class can be used instead of defining a new method.
- If not already existing new protocols for loading and removing tools have to be put in place and follow all the tools. For example, the tool programmer must have the possibility to specify actions at load time and at unload time.

Difficulties

- While transforming one case of the case statement into a registree object, you will have to define an action associated with the tools via the registree object. To ensure a clear separation and full dynamic registration, this action should be defined on the tool and not anymore on the tool user. However, as the tool may need some information from the tool user, the tool user should be passed to the tool as parameter whne the action is invoked. This changes the protocol between the tool and the tool user from a single invocation on the tool user to a method invocation to the tool with an extra parameter. This also implies that in some cases the tool user class have to define new public or friend methods to allow the tools to access the tool user right information.
- If each single conditional branch is associated only with a single tool, only one registree object is needed. However, if the same tool can be called in different ways we will have to create multiple registree objects. You need to identify the right criteria, usually the expression used in the conditional give a good discrimator. Creating multiple registree objects may lead to multiple registration aspects and registree classes depending on the level of felxibility of the implementation language.

When the legacy solution is the solution.

If all the tools are always available and you will never add or remove at run-time a new tool, a conditional is perfect.

Related Patterns

Transform Conditionals into Registration is related to Transform Conditionals on Client by the fact that they both eliminate conditional expressions that discriminate to select which method should be invoked on which object. The main difference between these two patterns relies in their use of the flexibility they provide. Indeed, both allow one to add new tools (service providers) without having to change clients. However, Transform Conditionals into Registration provides an architecture that supports the dynamic use of the available service providers while Transform Conditionals on Client only provide code flexibility without infrastructure support. In this sense, Transform Conditionals into Registration can be seen as a generalization of Transform Conditionals on Client.

Script: Identifying simulated switches in C++

This perl script searches the methods in C++ files and lists the occurrences of statements used to simulate switch statement with if then else i.e., 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
  # occurrence of "else if"
   $temp = join("",$‘,$&);
   $l = $linecount + split(/\n/,$temp) - 1;
   # count the occurrences of else-if pairs,
   # flag the positions for an eventual printout
   $swc = s/(else)([\s\n]*{?[\s\n]*if)
                       /$1\n* 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";
 }
}
```

Replace Type Code with Subclasses

Intent: Provides a recipe for carrying out the refactorings required for Transform Conditionals on Self *[Fowl99a].*

Replace Conditional with Polymorphism

Double Dispatch

Deprecation

Replace Type Code with State

Template Method

Intent: 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

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

NullObject

Intent: 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

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

State

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

State Patterns

Intent: 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.

[Deme99n]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.

[Duca99c]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.

[Dyso98a]

[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 Buschmann (Eds.), pp. 5-18, Addison-Wesley, 1998.