Chapter 11

Cluster: Type Check Elimination

TYPE CHECK ELIMINATION IN CLIENTS

Author(s): Stéphane Ducasse, Robb Nebbe and Tamar Richner

Intent

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

Applicability

Symptoms.

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

Reengineering Goals.

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

Related Reengineering Patterns. A closely related reengineering pattern is TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY, where the case statements over types are in the *provider* code as opposed to the *client* code. The essential distinction is if the decision structure is over the type or an attribute functioning as a type of: (a) an instance of *another* class (this pattern) or (b) an instance of the class to which the method belongs (see TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY in section 11).

Motivation

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

Initial Situation. The following code illustrates poor use of object-oriented concepts as shown by Fig. 11.1. The function makeCalls takes a vector of Telephone's (which can be of different types) as a parameter and makes a call for each of the telephones. The case statement switches on an explicit type-flag returned by phoneType(). In each branch of the case, the programmer calls the phoneType specific methods identified by the type-tag to make a call.

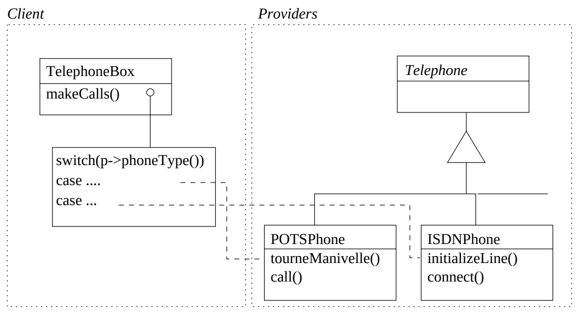

Figure 11.1: Initial relation and structure of clients and providers.

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

Final Situation. After applying the pattern the corresponding ringPhones() will look as follows and the structure as shown by the Fig. 11.2.

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

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

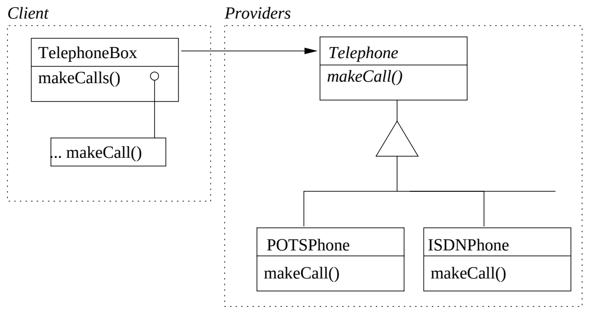

Figure 11.2: Final relation and structure of clients and providers.

Structure

Participants.

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

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

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

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

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

Process

Detection. The technique described in the pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY to detect case statements is applicable for this pattern. Whereas in the pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY, the switches are located in the same class, hence in one file for a language like C^{++}, in this pattern the case statements occur in several classes which can be spread over different files.

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

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

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

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

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

Difficulties.

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

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

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

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

Discussion

During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [BECK 94], [ALPE 98]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [GAMM 95] , [ALPE 98]. Moreover, the Strategy pattern [GAMM 95], [ALPE 98] is also based on the elimination of case statement over object state.

Language Specific Issues.

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

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

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

If a discriminated record has been used to implement the hierarchy it must first be transformed by applying the TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY pattern.

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

Tools

Glimpse and agrep can be found at ftp://ftp.cs.arizona.edu/glimpse.

Known Uses

In the FAMOOS mail sorting case study, we identified 28 matches (a match is not equivalent to a file because a same file may contain several switches) for the expression agrep 'switch;type', 185 matches for the sole expression agrep 'switch'. In the same time agrep 'if' gave us 10976 matches whereas using the perl script shown above we reduce the matches to 497.

In this case study, we identify three obvious lacks of polymorphism but they were not corresponding with the presented pattern but its companion pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY. We found also cases that implement state object [GAMM 95].

This pattern has been also 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, manual type check also occurs implemented statically via # ifdefs.

TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY

Author(s): Stéphane Ducasse, Robb Nebbe and Tamar Richner

Intent

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

Applicability

Symptoms.

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

Reengineering Goals.

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

Related Reengineering Patterns. A closely related pattern is TYPE CHECK ELIMINATION IN CLIENTS where the case statements over types are in the client code as opposed to the provider code. The essential distinction is if the decision structure is over an instance variable of the class (this pattern) or another class (see TYPE CHECK ELIMINATION IN CLIENTS in section 11).

Motivation

Case statements are sometimes used to simulate polymorphic dispatch. This often seems to be the result of the absence of polymorphism in an earlier version of the language (Ada'83 $\rightarrow$ Ada'95 or C $\rightarrow$ C^{++}). Another possibility is that programmer don't fully master the use of polymorphism and as a result do not always recognise when it is applicable. In any language that supports polymorphism it is preferable to exploit the language support rather than simulate it.

In the presence of polymorphism the process of dispatching is part of the language. In contrast, with case statements or other large decision structures the simulated dispatch must be hand coded and hand maintained. Accordingly, changing or extending the functionality are more difficult because they often affect many places in the source code. It also results in long methods with obscured logic that are hard to understand.

Programmers often fall back to the language they are most familiar with – in the Variable State pattern Kent Beck shows an example of such a situation related to Lisp programmers [BECK 97]. Thus, they may continue to implement solutions which do not exploit polymorphism even when polymorphism is available. This could occur especially when programmers extend an existing design by programming around its flaws, rather than reengineering it.

Initial Situation. Our example, taken in a simplified form from one of the case studies, consists of a message class that wraps two different kinds of messages (TEXT and ACTION) that must be serialised to be sent across a network connection as shown in the code and the figure 11.3.

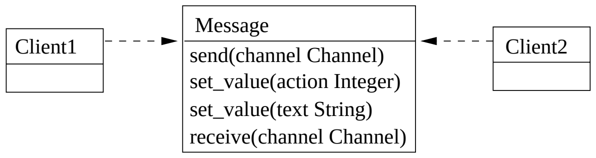

Figure 11.3: Initial relation and structure of clients and providers.

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

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

Final Situation.

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

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

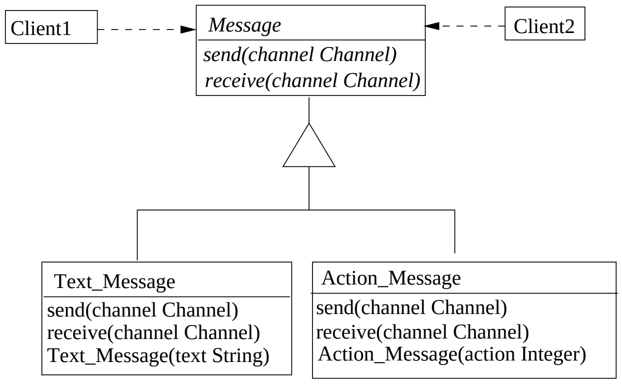

Figure 11.4: Final relation and structure of clients and providers.

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

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

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

Structure

Participants.

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

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

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

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

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

Process

Detection.

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

Example: detection of case statements in C++ Knowing if the pattern should be applied requires the detection of case statements. Regular-expression based tools like emacs, grep, agrep help in the localisation of case statements based on explicit construct like C++'s `switch` or Ada `case`. For example, `grep 'switch' ‘find . -name "*.cxx" -print‘` enumerates all the files with extension `.cxx` contained in a directory tree that contains `switch`. The grep facilities for grep are extended in `agrep` so it is possible to ask for finer queries. For example, the expression `agrep 'switch;type' -e ‘find . -name "*.cxx" -print‘` extracts all the files containing lines having `switch` and `type`.

However, even for a language like C++ that provides an explicit case statement construct, detecting case statements based on explicit `ifthenelse` structures is necessary. The tools above are not well suited for such a task, since their detection capabilities are restricted to one line at a time. One possible solution is to use perl scripts - a perl script which searches the methods in C++ files and lists the occurrences of case statements can be found in the appendix.

Recipe.

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

Difficulties.

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

Discussion

During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [BECK 94], [ALPE 98]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [GAMM 95], [ALPE 98]. Moreover, the Strategy pattern [GAMM 95], [ALPE 98] is also based on the elimination of case statement over object state.

In his thesis Opdyke [JOHN 93] discusses the automatisation of code refactoring. His “Refactoring To Specialise”, in which he proposed to use class invariant as a criteria to simplify conditionals, is similar to this pattern.

Language Specific Issues.

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

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

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

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

Tools

Glimpse and agrep can be found at ftp://ftp.cs.arizona.edu/glimpse.

Known Uses

In one FAMOOS case study several instances of this problem were found. In the example studied in depth (DialogElement) it appears in conjunction with a class that groups together user interface and core model functionality. There is a data member called _type that is used in the various switch statements. Furthermore a void pointer is frequently cast to an appropriate type based on the value of _type.