The Phrasebook Pattern

By Yonat Sharon and Rani Pinchuk

Abstract

The Phrasebook pattern is a technique to separate expressions in one language from the main code that is written in another programming language. This is done by keeping the expressions in a separate phrasebook. Any specific expression can be generated by finding the appropriate entry in the phrasebook and substituting application parameters into it. This separation makes maintenance easier both for the main application code and for the foreign language code.

PHRASEBOOK

Intent

When the application needs to execute, create or output expressions in another language, keep these expressions in a separate phrasebook.

Motivation

Problem

Consider a bookshop application written in Perl that uses a relational database and SQL. Users can search books by title, and the application will then display a list of books with this title. The books are indexed by their ISBN, so in order to display information for any of these books, the application needs to find its ISBN. This may be done by executing an SQL statement:

```
$statement =
  q(select isbn from book where title =
    'Design Patterns');
$sth = $self->{DBH}->prepare($statement) ;
$rc=$sth->execute();
```

Listing 1 - SQL statement embedded inside Perl code

The SQL code is embedded into the application code, usually in different places. This mix of two languages decreases the readability of the code and makes it harder to maintain both the application code and the SQL code. Also, similar SQL statements, perhaps differing only slightly, may appear in several places, thus making changes even more difficult. Moreover, the programmer that maintains the SQL code, needs to know Perl in order to find the SQL statements and to change them.

Solution

To avoid this, we can replace the inline SQL statements with calls to a `PhraseGenerator` that is responsible for providing the necessary statements as shown in Figure 1. When a specific SQL statement is needed, we ask the `PhraseGenerator` to generate it by specifying its phrase lookup key and parameters (if any):

```
$statement = $sql->get("FIND_ISBN",
                   {title => "Design Patterns"});
$sth = $self->{DBH}->prepare($statement) ;
$rc=$sth->execute();
```

Listing 2 - Phrase lookup for the SQL statement

We keep the actual SQL statements and their corresponding lookup keys in a separate phrasebook that is read and parsed by the `PhraseGenerator`. We can use XML to wrap the SQL and the lookup keys:

```
...
<statement name="FIND_ISBN">
   select isbn from book where title = '$title'
</statement>
...
```

Listing 3 - Phrasebook entry of the SQL statement

The phrasebook entries may contain placeholders (like the `$title` above) and simple control structures (if-else, while loops) that will be substituted with actual values when the statement is requested.

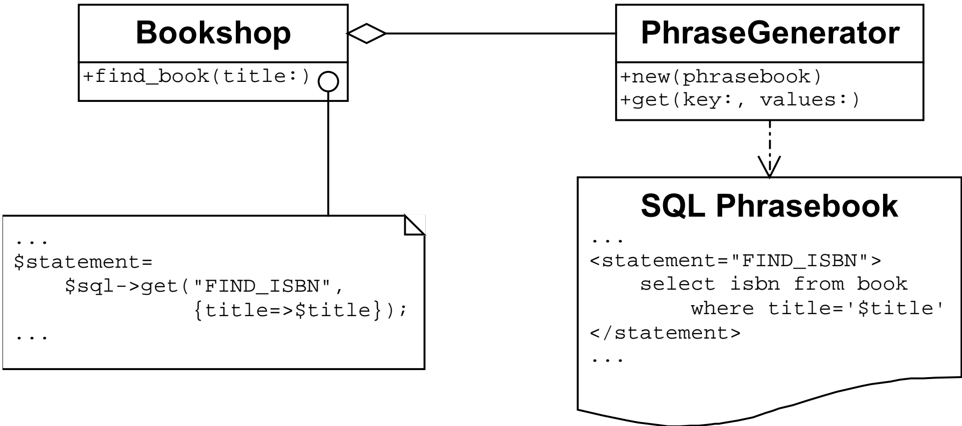

Figure 1 - Bookshop Perl application using a separate SQL phrasebook

So instead of having SQL statements embedded on the main application code, we generate these statements from generic phrases.

Statically typed solution

The `get` method above doesn't provide type checking (that is not provided anyway by Perl). If we want to have type checking (in C++ for example) we can create a special method for each SQL statement instead of using only one method (`get`) with keys:

```
statement = sql.find_isbn("Design Patterns");
```

Listing 4 - Phrase lookup with static type checking

In this case we need two classes, as shown in Figure 2: the first is the same `PhraseGenerator` that we used above with its `get` method; the second is a wrapper around it that provides one method for each generic phrase in the phrasebook (`BookPhraseGenerator`).

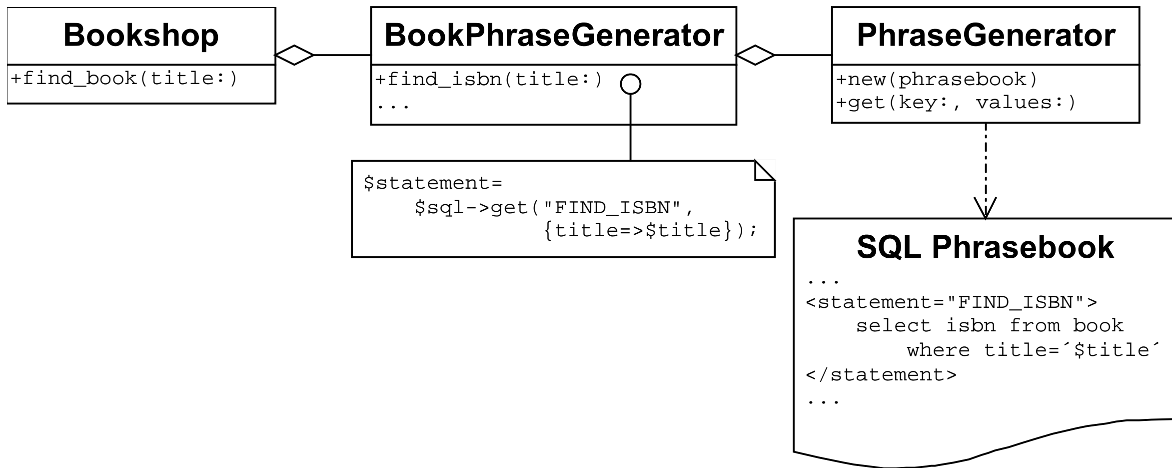

Figure 2 - The Bookshop application with a phrasebook-specific statically checked wrapper class

Both these techniques allow us to provide custom SQL statements, without polluting the rest of our code with SQL.

Applicability

Use this pattern when

1. The program needs to execute, create or output expressions in an external format or foreign language (e.g., English, SQL, PostScript); and
2. The expressions can be separated or combined to "phrases", so that these phrases are unrelated or have a simple relation (e.g., linear ordering). A phrase may contain anything from a single atom of the foreign language, to the complete output of the program (e.g., error messages, database transactions, documents); and
3. The phrases can be generated from generic phrases by applying simple transformations (e.g., copying, simple variable substitution).

Structure

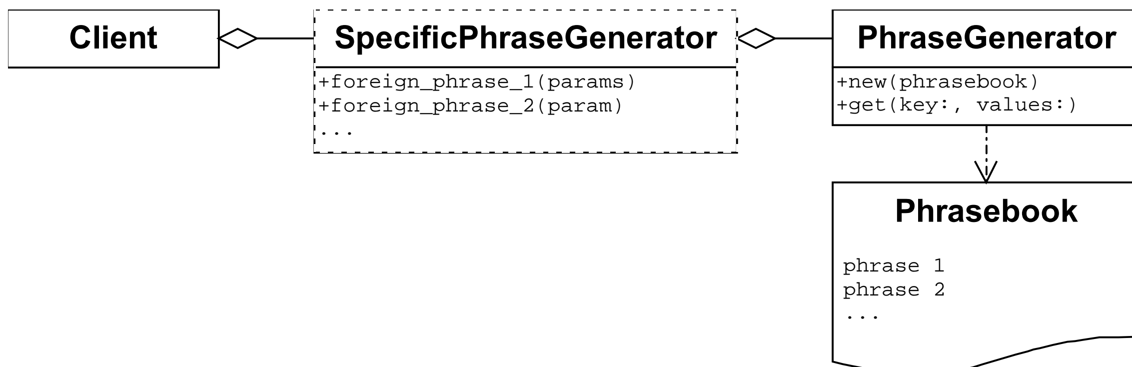

Figure 3 - Class structure of the Phrasebook pattern (dashed lines mark optional wrapper)

Participants

- **Phrasebook**
  - Lists the foreign programming language generic phrases.

- Associates each generic phrase with a lookup key.

- **PhraseGenerator**
  - Prepares a Phrasebook for usage.
  - Translates lookup keys and parameters to specific foreign language phrases.
- **SpecificPhraseGenerator (optional - for static type checking)**
  - Has a method for each generic foreign language phrase.
  - Calls PhraseGenerator with the method's parameters to create specific foreign language phrases.
- **Client**
  - Calls the PhraseGenerator (or SpecificPhraseGenerator) object to create phrases in a foreign language.
  - Uses the phrases by invoking them, displaying them, sending them etc.

Collaboration

Without static type checking:

1. The Client initializes the PhraseGenerator object with a Phrasebook for the foreign language.
2. The Client requests a specific phrase from the PhraseGenerator using its lookup key, and supplies parameters for it.
3. The PhraseGenerator object finds the generic phrase in the Phrasebook and substitutes the parameters into it. The resulting phrase is returned to the Client.
4. The Client uses the phrase.

With static type checking:

The SpecificPhraseGenerator object acts as an intermediate between the Client and the PhraseGenerator object. It performs the same operations as the Client in the typeless scenario. The collaboration between the SpecificPhraseGenerator object and the actual Client is as follows:

1. The client calls one of the methods of the SpecificPhraseGenerator in order to get a specific phrase in the foreign language, passing the necessary parameters.
2. The SpecificPhraseGenerator returns the phrase that results from substituting the parameters into the corresponding generic phrase in the Phrasebook.
3. The Client uses the phrase.

Consequences

The pattern has the following advantages:

1. *Separation of languages.* The foreign language code is separated from the rest of the application. This makes it easier to change and manage these artifacts independently. There is less danger that refactoring and bug fixing in one language will affect the code in the other language. This will also make the application code more readable and the foreign language code more accessible since it is not scattered in many

different places.

2. *Clear-cut code-ownership.* It is easier to divide the responsibility for the code according to expertise. People maintaining the code in one language do not need to be familiar with another language in order to maintain their code.
3. *Dynamic language selection.* It is possible to change the foreign language even at run time. To change the foreign language, only the phrasebook needs to change. Thus, it is possible to have several phrasebooks for several languages. This may be useful for localization and for supporting different targets for the foreign language phrases (for example, a drawing application may support exporting the drawing to PostScript, PDF and PICT by using a different phrasebook for each of these formats).

The pattern has the following liabilities:

1. *More complex design.* The extra level of indirection introduced by the PhraseGenerator adds to the complexity of the design.
2. *New format.* The foreign language programmers need to learn the phrasebook format. A way to facilitate this might be providing a tool to manipulate the phrasebook (like resource editors that are used to manipulate string tables - see Known Uses).
3. *Interface inflation.* The statically checked variation of the pattern requires adding a new method each time a new entry is added to the phrasebook. If the phrasebook is likely to contain many entries, this class may become too big.

Implementation

1. *Static prototype checking*. Each phrasebook entry has a specific prototype it supports: the number, names and types of parameters it needs and its specific lookup key. If a requested phrase does not match the prototype of any entry in the phrasebook, a runtime error may occur. To catch these errors during compile time in statically typed languages, we can wrap each specific Phrasebook with the SpecificPhraseGenerator class. However, this requires changing the class interface whenever a new entry is added to the phrasebook, so the tradeoff is between prototype-safety and flexibility.
2. *Phrasebook format*. Basically, the phrasebook is a list of {key, expression} pairs. To minimize the effort required to learn how to use the phrasebook, it is best to create it in a familiar or easy to learn format (like XML that was used in the Motivation). Another option is to create a friendly tool to manipulate phrasebooks.
3. *Phrase parameters*. If we want to be able to substitute parameters into generic foreign language phrases, we need a mechanism to identify placeholders for these parameters in phrasebook entries. Possible mechanisms are:
   a) Using escape sequences. For example, the character "$":
   ```
   select isbn from book where title = '$title'
   ```
   b) Using parameters declaration. For example
   ```
   #define FIND_ISBN(t) select isbn from book where title = 't'
   ```
   c) Using named elements. For example:
   ```
   select isbn from book where title = '<title>Refactoring</title>'
   ```
   This mechanism enables having default values when some parameters are missing.
4. *Control structures in phrasebook entries.* The parameters of the phrasebook entries

can affect the structure of the resulted phrase. For example, here is a specification to display either a list or a "not found" message:

```
    <IF condition="no_matches">
        Sorry, no matches found.
    </IF>
    <ELSE>
        <WHILE condition="next_match">
            <A HREF="$address">$name</A><BR>
        </WHILE>
    </ELSE>
```

Listing 5 - Phrasebook entry with control structures

However, it is important not to go too far and let application logic slip into the structure of phrasebook entries. (For a more detailed discussion about keeping application logic out of phrasebook entries see The Skin Pattern (1) Implementation section.)

5. *Coupling phrase execution with generation.* When all the generated phrases are always used in the same way (e.g., always displayed or always saved to file) it is possible to add this responsibility to the PhraseGenerator. However, this will make it harder to change the target of the phrases or to aggregate them into a bigger story.

Sample Code

SQLStatement class is a PhraseGenerator Perl class that lets us manage centrally parameterized SQL statements.

```
use SQLStatement;

# open the phrasebook
my $sql = new SQLStatement("library.xml");
...
# generate an SQL statement
$statement = $sql->get("FIND_ISBN",
                       { title => $title });
```

Listing 6 - Perl program that uses an SQL phrasebook

The phrasebook is written in a separate file as XML document:

```
<?xml version="1.0"?>
<!DOCTYPE sql [
<!ELEMENT sql (statement)*>
<!ELEMENT statement (#PCDATA)>
<!ATTLIST statement name CDATA #REQUIRED>
]>
<sql>
 <!-- get the isbn according to the title -->
 <statement name="FIND_ISBN">
  select isbn from book where title = '$title'
 </statement>
 ...
</sql>
```

Listing 7 - SQL phrasebook file

Each entry in the phrasebook has a name attribute that is used as a lookup key. The entries can specify parameters by using the `$` escape character.

Known Uses

The SQLStatement class is used by EM-TECH group for several Web application projects. One example is the DHL Millenium Information Center that was used by DHL world wide in the first days of the millenium in order to have real time status reports from all over the world.

String tables, commonly used to make user interface easy to localize, are a simple instance of Phrasebook. The foreign language here is a human language (like English). The string table serves as a phrasebook: it contains string lookup keys and their associated string values. The string tables can be changed without affecting the main application code. When a specific message needs to be displayed, the application searches the string table for the string with the appropriate lookup key. Apple MacOS (2), Microsoft Windows (3) and some application frameworks use string tables.

Many text editors have a user-configurable list of tools that can be invoked by the editor. Each tool is defined by a name and a shell command. The shell command may contain placeholders to parameters that are supplied at runtime by the editor application. For example, the Perl interpreter may be invoked on the active file by: `perl %f` where `%f` will be substituted with the current file name. The tool list is the phrasebook, and the shell command language is the foreign language.

Many code generators use a phrasebook with code snippets or complete source files. For example, Microsoft Visual C++ has a feature called Custom AppWizard (4) that generates source files from "TextTemplates". These templates can contain "macros" that are substituted with actual values when the files are generated. For example: `class $$APP_CLASS$$` can be used to generate a class declaration. The class `CCustomAppWiz` is the PhraseGenerator class, its member `m_Dictionary` is the phrasebook, and its method `ProcessTemplate` is used to generate the code phrases.

The Linux administration tool linuxconf (5) uses Phrasebook for localization. All the text for its user interface is placed in a phrasebook file. Each phrasebook entry may contain printf-like parameters. For example: `Jobs listing for %s`.

Related Patterns

The Skin pattern (1) guides to create a view specification language. Then it guides to use this language to create Skins that are templates for views. It can then use Phrasebook to access skins and generate views. In this case, each skin is a generic phrase in the view specification language of the application.

After the foreign language phrases are generated, they can be encapsulated inside Command (6) objects.

Phrasebook can be used instead of Builder (6) if the product is in a foreign language (e.g. TeX or RTF) and the relations between its elements are simple (e.g., the product is simply an aggregation of its elements). Each element can be represented by a foreign language phrase. This allows more flexibilty and easier configurability, by using different phrasebooks instead of different subclasses of Builder. It also makes it easier to have complex elements, since these elements are maintained seperatly in the phrasebook. Both these patterns separate the foreign language from the main code. Builder is better when the relations between foreign language phrases are complex, while Phrasebook is better when the phrases themselves are compex, as shown in Figure 5. When both the phrases and their relations are complex, it may be possible to use a combination of

Builder and Phrasebook: The phrases are created using Phrasebook and assembled using Builder.

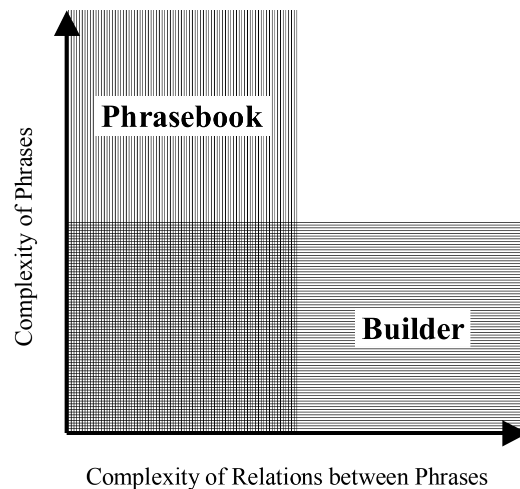

Figure 4 - Applicability domains of the Phrasebook and Builder patterns

Acknowledgements

We would like to thank our PLoP2000 shepherd Alejandra Garrido for her insightful comments, and the writers' workshop participants for their helpful feedback.

We thank EM-TECH group for providing the environment where the SQLStatement class could be created, and for providing resources that helped in writing the pattern.

References

1. R. Pinchuk and Y. Sharon. *The Skin Pattern*. In this volume.
2. Apple Computer, Inc. *Inside Macintosh: Macintosh Toolbox Essentials*. Apple Developer Connection, 1996. http://developer.apple.com/techpubs/mac/Toolbox/Toolbox-2.html.
3. MSDN Library. *Platform SDK: Windows User Interface*. Redmond, MA: Microsoft, 1998. http://msdn.microsoft.com/library/psdk/winui/resource_1inn.htm.
4. MSDN Library. *Visual C++ Programmer's Guide.* Redmond, MA: Microsoft, 1998. http://msdn.microsoft.com/library/devprods/vs6/visualc/vccore/_core_creating_custom_appwizards.htm.
5. C. Williams. *Translation system for Linuxconf*. http://www.solucorp.qc.ca/linuxconf/translat/translat.html.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.