INGI2252 : Measuring & Maintenance Project
Study of PostgreSQLDriver package

Nicolas Baerts

13 mai 2004

1 Selected case : PostgreSQLDriver package

From the user point of view, the PostgreSQL library, available under license LGPL, provides access to PostgreSQL databases using a TCP/IP socket connection. Use of a socket level interface makes this library very portable, eliminating the depencency on platform specific.

The library is made up of a number of components with distinct role :
- `PostgreSQLDriver` : The Smalltalk PostgreSQL driver which is implemented in terms of PostgeSQL Frontend/Backend Protocol (version 2.0) abstractions
- `PostgreSQLLogging` : Provides frontend/backend message tracing
- `PostgreSQLEXDI` : Maps the driver layer to the VisualWorks EXDI interface
- `StoreForPostgreSQL` : Supports the use of PostgreSQL as a repository for Store
- `PostgreSQLVWCompatibility` : For the portability of the driver on environment VisualWorks

In our case, the analysis of the package `postgreSQLDriver`, present in the parcel, was justified by its complexity with interesting appearance (cfr. FIG. 6).Indeed, it includes, amongst other things, a “beautiful” hierarchy, as well as an interesting number of classes(cfr. FIG. 1).

2 Quality assessment

Generally, after having flown over some global views as well as the code, this last appears, in the form, very good, well written, and easy to read.
This first analysis will be divided into three parts. First of all, a short general discussion will be carried out, followed by analyses concerning the classes, the methods, and finally the attributes.

Conventions of writing in SmallTalk such as classification of the methods in an adequate protocol, use of names of classes, methods, and attributes adapted,... are respected. To continue with these best practice paterns, the pattern double dispatch is used in certain cases. Indeed, we will see this most in the parsing where the elements are all figuring out

what the incoming message is. It is a good thing, because like the author of this package (*Bruce Badger*) said, he doesn't think that the use of double dispatch is an automatic Good Thing ; it's a good technique if used in the right way at the right time (like all patterns, really).

Lastly, like the FIG. 1 is showing, the classes and the methods are in general not too bulky.

Number of classes	68
Number of abstract classes	9
Total weighted number of lines of code	3647
Maximum class hierarchy nesting level	5
Average number of lines of code per method	10.44
Average number of accesses per method	5.79

FIG. 1 – System Metrics

Concerning the maturity of the code, it is in one of the phase *expansion*.
Indeed, there is a "beautiful" hierarchy, utilisations of best practice patterns, and unit testing.

From the point of view of the re-usability of the code, we observe generally that a method is used several times, as well inside the class, as outside.

Let us notice that no message Chains (a client asks an object for another object and then asks that object for another obect etc), no middle Man (objects hide internal details and encapsulation leads to delegation but sometimes this goes too far) and no big switch statements ("Case of") were found.

On the level of the portability of the driver on different environments or platforms, we can observe that the code studied is independent from it.To remember, the independence of the application towards environment used is due to the regrouping of these dependences in the package `PostgreSQLVWCompatibility`. Concerning independence towards the platform, it is found in the use of the sockets.

2.1 Classes

Let us notice that no lazy class was found. In the same spirit, no inappropriate Intimacy (pairs of classes that know too much about each other"s private details), no data Clumps (a certain number of data items in lots of places), no primitive Obsession (characterised by a reluctance to use classes instead of primitive data type), no alternative classes with different interfaces (methods that do the same thing but have different signatures), and only one refused bequest (when a subclass ignores and doesn't need most of the functionality provided by its superclass) were found.

2.2 Methods

Concerning the name of the methods, they express generally well the goal of the method. Moreover, the methods are in the right protocols, which increases even more comprehension of the code. In the same spirit, the keywords (in methods selectors) communicate the parameter roles and the parameter names suggest the expected parameter standard.

Concerning the accessor and mutator methods, this point is developed at the end of the section 3.3. Let us note simply that a single convention is not always used.

Lastly, the messages “subclassResponsibility” and “shouldNotImplement” are used respectively when the method is not implemented by an abstract class and when a method defined in the mother class is not implemented.
Let us notice that no feature Envy (when a method seems more interested in a class other than the one it actually is in) was found.

2.3 Variables

For this part, there is no negative remark to say. Indeed, their names are generally revealing their roles, their properties.

Lastly, no temporary Field (instance variables that are only set sometimes are hard to understand) was found.

3 Opportunities for restructuring

3.1 Naming convention

To start, let us discuss the name given to the various classes.
In general, these names are suitable and indicate the responsibilities for the corresponding class clearly. However, by observing a sight of the complexity (with the names of the various classes) of the system, and mainly of the principal hierarchy, one can observe inconsistences in the conventions taken for under tree of the hierarchy.

By observing the subclasses of `PostgreSQLBackendMessage`, only one class does not contain the word “Message” in its name.

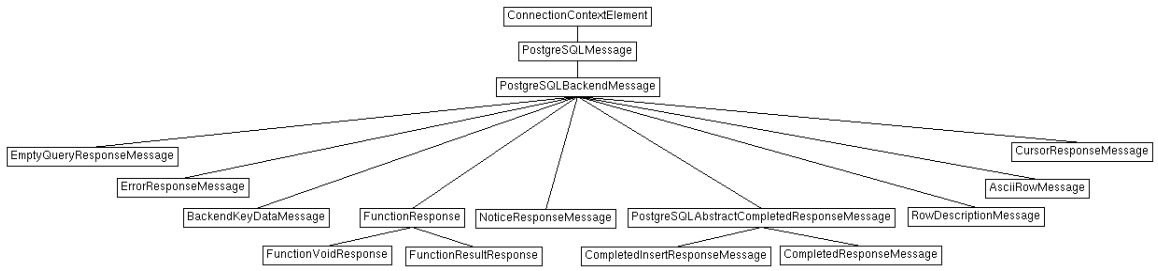

FIG. 2 – BackendMessage - Complexity View

After having observed the class `FunctionResponse`, it would be perhaps clearer and more coherent with convention used to add the keyword “Message” in the name of the class. Concerning the subclasses of `RMessage`, it does not seem useful to add keywords like “Backend” before “Message”.

Always in the part of the messages of the system, let us observe the subclasses of `PostgreSQLFrontendMessage`.

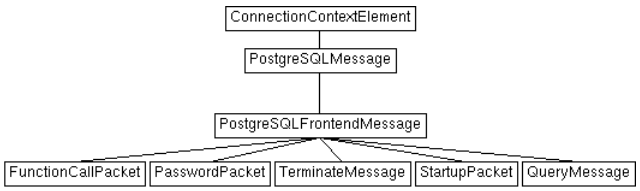

FIG. 3 – FrontendMessage - Complexity View

As previously, it would be perhaps interesting to add “FrontendMessage” at the end of the names of classes to standardize and identify this part of the hierarchy.

In the part concerning the flow of the message, a class leaves again the batch : the class `PostgreSQLRequest`.

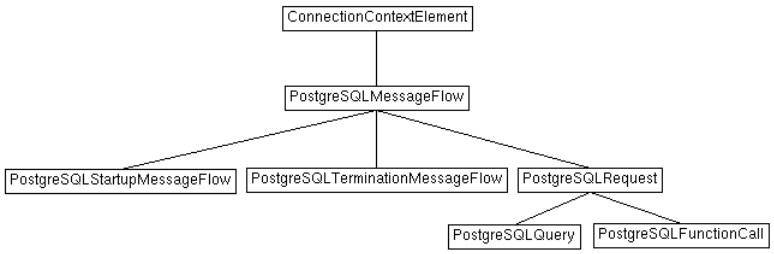

FIG. 4 – Flow - Complexity View

A request being a well-known stage of the flow message, the addition of the keyword “MessageFlow” would be to some extent redundant, and thus useless. In this case, there is thus no problem.

3.2 Duplicated Code

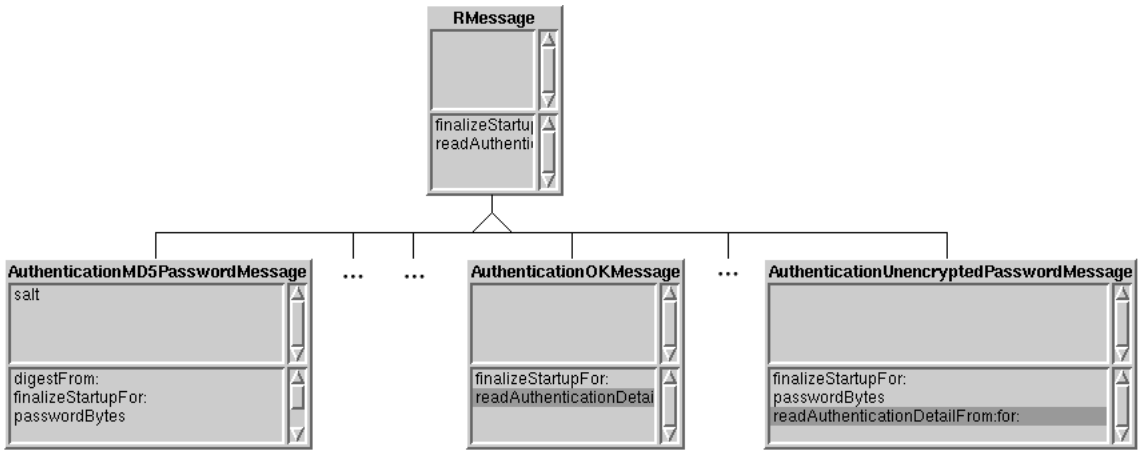

FIG. 5 – Duplicated code

After a research of duplicated code in various classes of the same level in the hierarchy, the methods `AuthenticationOKMessage.readAuthenticationDetailFrom :for :` and `AuthenticationUnencryptedMessage.readAuthenticationDetailFrom :for :` are exactly the same.

3.3 System Hotspots View

Let us carry out our analysis while concentrating on various specific views of the Code-Crawler tool.
After various choices from metric, four classes with a high number of lines of code or of methods (compared to the entirety of the package) are in the foreground.
This view is interesting to detect large class, where the more likely code is duplicated unnecessarily.

- `PostgreSQLTest` (WLOC : 852 ; NOM : 36) : this class is the only meta-class represented. Being given that it takes all the tests relating to the package, we will not discuss it any more.
- `PostgreSQLDriverLicense` (WLOC : 513 ; NOM : 6) : this class, which includes few methods compared to its number of line of code is explained by the fact that it consist of the LGPL license in the form of a string.
- `PostgreSQLConnection` (WLOC : 175 ; NOM : 25) : after I have seen the view “ Blueprint ”, all the methods are of suitable sizes (maximum 18 lines of code by method). There is thus no apparent anomaly on the size of this class.
  However, this sight enables us to detect inconsistencies on the level of the use of the access methods to the attributes. Indeed, within this classes, one reaches two attributes sometimes directly (“parameters” and “socketSession”), sometimes via an access method.
- `ConnectionParameterSet` (WLOC : 49 ; NOM : 15) : Again, there is no apparent anomaly on the size of this class, but small inconsistencies on the use of the access

methods in the class in itself.
Let us notice here that the two last remarks relating to various inconsistencies on the use of the access methods apply in the majority of the classes. We will not speak about it more.

3.4 System Complexity View

The figure below represents a global view of the system. Let us note that the metric used in this case are the number of attributes for the width, the number of methods for the height, and numbers of lines of code for the color. Also let us note that the names of classes in the figure are those previously quoted ; we will not speak about these classes more.

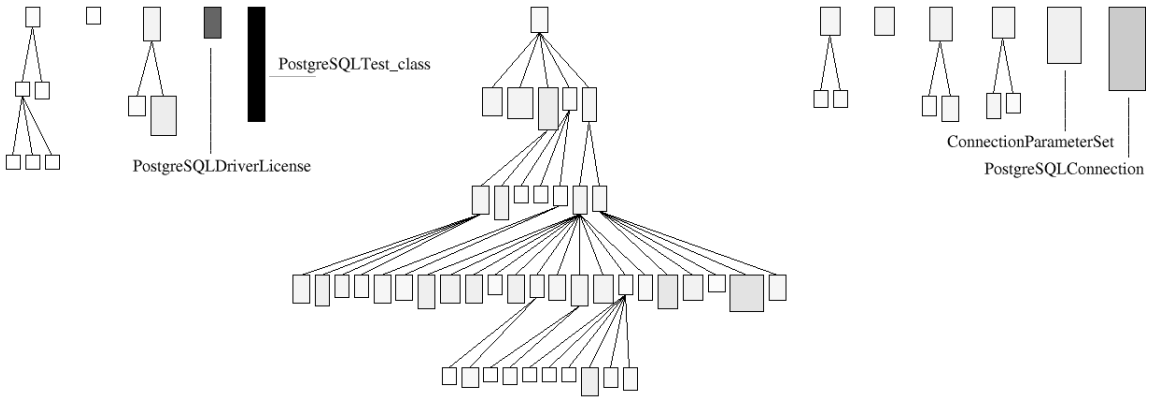

FIG. 6 – System - Complexity View

Within the small hierarchies, only the extreme left (which contains three levels) is interesting. It consists of the various exceptions and specific errors. Let us observe a Blueprint view of this one.

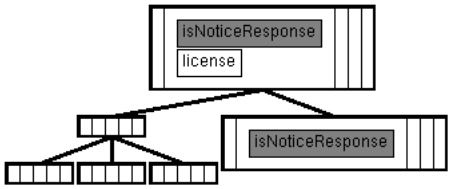

FIG. 7 – Exceptions and errors - Blueprint View

We notice that the Boolean method `isNoticeResponse` returns *false* for all the classes, except the class of right-hand side, `PostgreSQLNoticeResponse`.
This solution does not seem optimal to me within sight of a direct comprehension of the role and application of this method

Concerning the core of the package (the principal hierarchy), no particular node arises. Moreover, the use of various Blueprint views do not make it possible to detect anomalies.

3.5 Inheritance Classification View

This view helps to detect classes with a certain impact on their subclasses in terms of fonctionality.

We focus on the principal hierarchy. This part is mainly made up of flat, light nodes. These nodes represents classes with little behavior and few descendants. Consequently, the relations of heritage are mainly additions of functionalities by the subclasses.
Let us note that exceptions are however present, but no appropriate reorganization is found.

3.6 Inheritance Carrier View

The following view helps to detect classes with a certain impact on their subclasses in terms of functionality, in the principal hierarchy. The metrics used are the number of descendants for the width and the colors, and numbers of methods for the height.

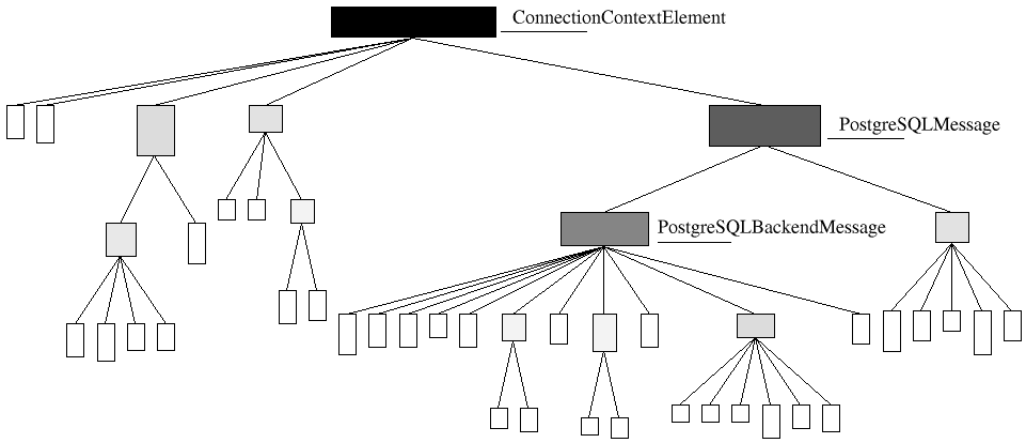

FIG. 8 – Principal hierarchy - Inheritance Carrier View

Let us notice that the principal characteristic nodes are the flat dark nodes, representing classes with little behavior and few descendants. They can be the ideal place to factor out code from to the subclasses.
Three classes having these characteristics are present :

- `ConnectionContextElement` (WNOC : 44 ; NOM : 4) : After having observed this class with its descendants, no problem was found.
- `PostgreSQLMessage` (WNOC : 28 ; NOM : 7) : This class implements in fact SQL messages. Considering that its subclasses respectively implement the messages frontend and backend, it is difficult to factor out code from to these subclasses.
- `PostgreSQLBackendMessage` (WNOC : 21 ; NOM : 5) : Again, no problem was found.

3.7 Service Class Detection View

This view relates the number of methods with the lines of code of classes and interprets this information in the context of the principal hierarchy. Ideally, this view should return a

staircase pattern from left to right.
In our case, it is not completely the case. Mainly, the staircase pattern is broken by nodes which are too short. These classes, given a certain number of methods, do not have the expected length in terms of lines of code. Such classes are often data storage and may point to sets of coupled classes being brittle to changes.

After analysis of the various classes, no problem was found.

3.8 Method Efficiency Correlation View

This very scalable view shows all methods using a scatterplot layout with the lines of code (for the X position) and the number of statements (for the Y position) as position metrics. As the two metrics are related (each line may contain statements) we end up with a display of all methods, many of which align themselves along a certain correlation axis.
This view is also interesting to detect long method. Indeed, the longer a method is, the more difficult it is to understand.

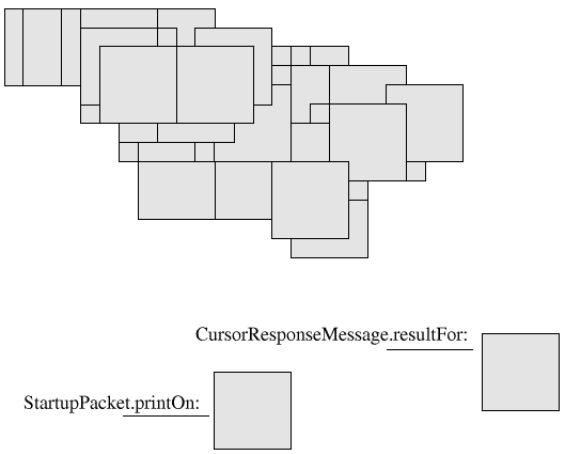

FIG. 9 – Principal hierarchy - Method Efficiency Correlation View

If we observe the general allure of the figure, there are more lines of code than statements.
This tendency is explained by the fact that many introductory comments are present in almost the whole of the methods. Moreover, one statement is generally put, for a better comprehension, on several lines.

However, two methods leave the batch :

– `StartupPacket.printOn :` (LOC : 13 ; NOS : 20) : This method states simply the various characteristics (header) of the packet on which the method is called. This method does not need a refactoring.
– `CursorResponseMessage.resultFor :` (LOC : 27 ; NOS : 18) : This method, which returns the result for a message and represents a cursor response, is more interesting,

and has need for refactoring.

If we change the view's metrics with the number of (input) parameters (NOP), we can detect the long parameter list, which are hard to understand. After analysis of this view, the maximum number of (input) parameters is 3, which is raisonable for comprehension.

3.9 Direct Attribute Access View

This view uses the number of direct accesses for the width, height, and color of each attribute node, and sorts the nodes according to this metric.
A good thing is that there is no node which is never accessed.
For large, dark nodes at the bottom point to attributes which heavily directly accessed, which may lead to problems, in case the internal implementation changes. For such nodes, one should also check whether accessor methods have been defined and, if yes, why they are not always being used.

One attribute leaves the batch :

– `PacketValue.bytes` (NAA : 12) : In fact, its number of local accesses is 4 and its number of accesses from outside of the class is 8
Because an accessor method is defined, nothing has to be modified.

With the number of local access as metric, one attribute leaves the batch :

– `QueryMessage.queryString` (NLA : 5) : Again, an accessor methods is defined.

4 Suggested improvement

The studied application being already well structured, only small refactorings will be considered.
Let us notice that only suggestions regarding "how" to restructure will be made.

To start with the name convention, and like said previously, certain names of classes can be changed.
It is the case of `FunctionResponse` (cfr. FIG. 2), becoming, with an aim of a better comprehension, `FunctionResponseMessage`.
In the same spirit and like said previously on FIG. 3, `FunctionCallPacket` can become `FunctionCallPacketFrontendMessage` ; `PasswordPacket`, `PasswordPackettFrontendMessage` ; `TerminateMessage`, `TerminateFrontendMessage` ; `StartupPacket`, `StartupPacketFrontendMessage` ; `QueryMessage`, `QueryFrontendMessage`.

Concerning the duplicated Code on FIG. 5, the refactoring "Pull Up Method" can be applied.
The idea is to create a class which contains the method `readAuthenticationDetailFrom :for :`, this class being the parent of `AuthenticationOKMessage` and `AuthenticationUnencryptedMessage`, and the child of `RMessage`.
In this case, the same code will be present only once and will not interfere with the same

abstract method present in `RMessage`.

Then, concerning exceptions and errors on FIG. 7, an idea is to make the method `PostgreSQLException.isNoticeResponse` abstract using the “self shouldNotImplement”convention. So, it implies to implement `PostgreSQLError.isNoticeResponse` in such manner that it return *false*.
This solution seems to me much optimal within sight of a direct comprehension of the role and application of this method.

Lastly, concerning the method `CursorResponseMessage.resultFor :` on FIG. 9, the refactoring seems more complicated.
Indeed, the method is the more ugly method of the package and, compared to the other methods of the package, relatively large.
An idea to make this part a little bit more readable is to apply the refactoring “Extract Method” to separate this method from the construction of the *nextMessage*.

5 Conclusion

To conclude, the code of `PostgreSQLDriver` appears good (perhaps too good), well written, and easy to read.
To a lesser extent, there are some bad smells implying the need for small refactorings. In the majority of the cases, these refactorings will not be complicated, but will bring a better quality of the code.