ProRefactor

January 2004
John Green, Joanju Limited

Revision 02: Minor fixes.

Audience

The target audience for this paper is anybody interested in empowering Progress 4GL developers with the productivity tools and techniques necessary for cost-effective code modernization. There are two parts to this paper. The first addresses refactoring. The second addresses our product, ProRefactor.

Part 1: Refactoring

What is Refactoring

Refactoring is the act of improving source code without changing its behavior. It is often discussed within the same context as *patterns*. Some reference materials are Design Patterns (Gamma et al.), Patterns of Enterprise Application Architecture (Fowler), and Refactoring (Fowler). Refactoring is historically a manual activity, but recently many automated refactoring tools have begun to emerge. These have turned refactoring into something that happens extremely quickly, and with a vastly reduced risk of error.

Why Refactoring

Application source code must constantly be refactored, otherwise it decays to chaos. For a growing and evolving application, well designed, well written code does not stay that way on its own. After a few years and a dozen programmers have touched what was originally a work of elegance, you are left with something that is difficult to maintain and very expensive to expand upon.

Advantages

Refactoring, especially coupled with automated tests, allows an entire application to be completely re-architected - one small, verifiable step at a time. Each small step takes you closer to your goal: application source that is well architected, maintainable, and capable of growth and evolution. There is little risk in refactoring, because each step is small, and each step is productive. With a good suite of automated tests, each step is also verifiable.

Cumulative Effect

It is worth re-emphasizing that most refactorings are small, and that on their own, they sometimes don't appear to provide a huge amount of value. It is when a clever architect documents a series of small steps into a method for significantly re-architecting an application that the power of refactoring becomes apparent.

Refactoring Progress 4GL

Joanju's History

Since early 2000, Joanju has been committed to the development of tools to assist Progress 4GL developers with the modernization of their applications. Four years of research and development has lead us to many conclusions, given us many insights, and left us with a large library of working tools. Over the past several months we have been packaging our refactoring tools and knowledge into a product named ProRefactor.

Working Together

Joanju has the libraries, the workbench, and the know-how for building automated refactoring tools. We have the *how*. As for the *what*, we have plenty of ideas, but we want to work closely with the architects who know what the code looks like today, and what it needs to look like tomorrow. We believe that we will find people with this knowledge in the developers using the 4GL, in the consultants, and in organizations like Progress Software's Empowerment group.

Requests for New Refactorings

Ideally, we are looking for requests for refactorings which can be described simply and mechanically. They should be accompanied by examples of 4GL source code, as simple and trivial as is reasonable, showing both the *before* and the *after*. We will use these examples as unit tests to ensure that the refactorings we build will always work with the examples provided to us.

The OO Factor

Many of the published patterns and refactorings are related to object oriented software development. Most of these OO patterns are applicable to Progress 4GL, simply by using persistent procedures and super procedures. We are aware (via John Sadd's recent post to the PEG) that Progress Software will examine the potential for extending the 4GL with more OO features. This effort is orthogonal to the development of ProRefactor and new refactoring features, and in fact, the two efforts are complimentary. OO extensions to the 4GL will provide more opportunities for the use of more sophisticated patterns and the refactorings that accompany those patterns – and the automations to make them happen quickly and reliably. ProRefactor can contribute to the OO effort by helping Progress 4GL developers migrate away from legacy procedural architectures to modern OO

architectures, in a cost effective manner.

A Selective Catalog of Refactorings

Here I will describe a few refactorings. Some of these are already automated by ProRefactor. Some of these descriptions are quite oversimplified.

Consolidate Program

This series of refactorings was described by Peter Dalbadie, et al. It is the clever application of a series of small refactorings which, combined, consolidates several closely related (tightly *coupled*) compile units into a single *cohesive* compile unit which can be treated like an OO *class*, and subjected to further, more interesting refactorings.

Bubble Declarations: This first step ensures that shared declarations are grouped separately from methods, which are grouped separately from local declarations, which are grouped separately from procedural code.

Wrap Procedure Block: This second step wraps the local declarations and procedural code into a PROCEDURE block. (Note that shared declarations and existing methods cannot be wrapped, which is why Bubble Declarations was necessary.) This is a common practice when turning a legacy procedural program into a modern object, run persistently.

Once the above steps have been performed on all of the tightly coupled programs, then we move on to the following.

Append Program: We append each tightly-coupled external .p or .w, one at a time, on to the topmost compile unit. With each of these, we eliminate duplicate shared declarations, and we replace the calls from the topmost compile unit to the external .p or .w with a call to the internal procedure that was created in the Wrap Procedure Block step.

You now have a single, cohesive unit, which can easily be analyzed for further, more interesting re-architecting. For example, UI elements that were once shared across several compile units are now contained in a single, large compile unit. Those UI elements can now be *factored* out, into their own OO style compile unit – separating the UI layer from the rest of the application. (See Move Method, page 4.) The same can be done for the data access parts, in order to create a data access layer. (See Move Field, page 4).

Published Refactorings

These refactorings are described in Fowler's book on refactoring, as well as in other publications.

Rename: This is the simplest and most common refactoring. Names are critical for the well-being of an application, and as the application evolves, so must the names. In some cases, a search and replace tool is sufficient for the job. In other cases, without a tool which is context aware and able to discern one symbol from another, renaming of symbols will be avoided by the programmer because it is time consuming, tedious, and error prone. Poorly named symbols, or names which did not evolve as the application evolved, are frequently the cause of increased training time, poor productivity,

misunderstandings, and bugs.

Extract Method: The programmer selects a code block, and turns it into a method. There are several justifications for this refactoring. It is often used to increase code clarity. It is used when it is discovered that a code block needs to be re-used. It is also used as the first step when the code block should be in another compile unit – see Move Method, page 4. Many refactorings also have an inverse – in this case, the inverse is named *Inline Method*.

Move Method: Move a method from one compile unit to another. Frequently used when separating layers of an application.

Encapsulate Field: Turn a public field (or any sort of data structure, such as a temp-table) into a private one, and provide *accessors* (*getter* and *setter*) methods. In Progress 4GL terms, this might mean turning an include file full of shared variables into a *Singleton* (see Singleton, page 4), removing SHARED from the field declarations, and providing the *getter* and *setter* methods.

Move Field: Move a field (or any sort of data structure, such as a temp-table) from one compile unit to another. Frequently used when separating layers of an application. When the field is moved, *getter* and *setter* methods are created in the target compile unit. References to the field in the original compile unit are replaced with calls to the *getter* and *setter* methods.

Big Refactorings

Consolidate Program, described above, could be considered a *big refactoring*, made up of a series of small refactorings. There are many published big refactorings. In Fowler's book, for example, there are Convert Procedural Design to Objects, and Separate Domain from Presentation.

Patterns

A pattern describes a simple and elegant solution to a recurring software design problem. There are dozens of patterns described in various publications, here I will describe just one as an example – the *Singleton*.

Singleton

A *Singleton* is an object (or persistent procedure) for which only one instance should exist. The Singleton contains a method which returns the instance of itself. Singletons are sometimes used in order to implement a method library. In OO languages, that method *getInstance* is a *class* method. In Progress, one would run *singleton.p* non-persistently, which would find or create the persistent instance of itself, and return the handle. In order to avoid shared variables, using publish-subscribe might be an appropriate method for implementing Singletons in the 4GL.

Part 2: ProRefactor

An Eclipse Plugin

ProRefactor is made up of a few distinct packages. One package provides all of the core refactoring functionality, and is completely independent of the workbench it is placed upon.

Another of ProRefactor's packages provides complete integration of those refactoring features into the Eclipse workbench.

ProRefactor currently contributes the following IDE functionality to the workbench:

- a Progress *perspective*. Eclipse's workbench is made up of menus, button bars, editors, and UI frames (called *views* in Eclipse). An arrangement and configuration of those workbench elements is called a perspective.
- a Progress *project* type
- two Progress project configuration pages
- a Progress Refactoring menu, with several available refactorings
- *wizard* dialogs to gather refactoring parameters
- the ability to select one or more projects, directories, or files for a given refactoring run
- automatically open all files created or modified by a refactoring
- a *rollback* mechanism, to allow the programmer to rollback the changes made by a refactoring
- create a list of *tasks* or messages, as *file markers*, for a given refactoring

Future ProRefactor Workbench Contributions

We will add more Progress specific IDE functionality to the workbench:

- 4GL-intelligent code searches
- syntax tree (AST) browsing
- file and compile unit outlines
- macro outlines (tree of include file and preprocessor references)

Waiting for an IDE

ProRefactor provides a number of workbench contributions that it really should not. That includes: the Progress perspective, the Progress project type, and its own top level menu. Those features should be contributed to the workbench by a general purpose Progress IDE plugin. ProRefactor would contribute, in turn, additional actions and views to that general IDE plugin.

Refactoring UI Features

Different refactoring situations require different presentation models for source code

changes. Some changes should be reviewed one by one. Some refactorings should be done wholesale, across entire directories or entire projects, and the changes reviewed afterwards. Some refactorings affect several files at onces, and in that case, the changes should be made, the modified files presented to the programmer, and a mechanism for rolling back the changes must be available.

Review Changes Dialog

The **Substitute** refactoring uses this UI mechanism. It is ideal when each instance of the refactoring affects only one or a few lines of code. Each time a refactoring is done, a **Review Changes Dialog** is presented, showing the code before, the code after, and buttons for accepting or rejecting the changes. The modified code may be further modified by the programmer before accepting the changes.

Temporary Output Directory

The **No-Undo** and **Table and Field Names** refactorings use this UI mechanism. It is ideal when a tiny refactoring is applied to a vast number of source files and large numbers of lines of code within those source files. The programmer is prompted for a temporary output directory, and the modified source is written to that directory. It is expected that the programmer will quickly review the modified source before copying it over top of their workspace. (There are many tools available, including some good free ones, which make it quick and easy to review all of the differences between all of the files in two related directory structures.)

Review and Rollback

Bubble Declarations, **Append Program**, and **Wrap Procedure Block** all use this mechanism. It is ideal when several source files are modified for a single refactoring. When the refactoring is run, a list of new or modified files is generated. All new or modified files are opened in editors, for the programmer's review. Eclipse's "local history" is updated before the change is written, so that the new and old versions of each file are easily compared, using Eclipse's compare features. A rollback mechanism allows the entire refactoring to be rolled back, if the programmer decides to start over.

The Refactorings

SUBSTITUTE Refactor

This refactoring is intended for those who are making a pass through their application source in order to make its string literals more translatable. Here is the ant-pattern and the pattern:

```
/* bad – fragmented sentence */
display "You have " + string(num-widgets) + " widgets in inventory.".
```

```
/* better – whole sentence */
display substitute(
   "You have &1 widgets in inventory.", string(num-widgets) ).
```

Features:

- supports concatenating multiple plus nodes into a single *substitute* expression
- is sensitive to string attributes, will not change untranslatable (*:U*) strings
- displays warning messages when there are mixed string attributes or mixed quotation types used within string concatenations

The Substitute refactoring uses the **Review Changes Dialog**, so that each refactored string concatenation can be reviewed by the programmer (and touched up if necessary) before accepting the modification.

NO-UNDO Refactor

An obvious and simple refactoring, this adds *no-undo* to variables and parameters which are missing that keyword. There are a few wrinkles though, where both source and semantic analysis serve to make this a more reliable refactoring:

- It is not applied where a *Prolint directive* appears in front of the *define* statement. Prolint has a rule which warns about missing *no-undo* keywords, but it also has a mechanism, the Prolint directive, which suppresses those warnings if the variable is intentionally undo-able.
- It is not applied to statements which have a comment containing the string "undo".
- It is not applied where that variable being defined is assigned or otherwise updated and then explicitly undone by an *undo* statement. See the code below for an example.

Since this refactoring is intended for running against an entire directory structure or an entire project all at once, it uses a temporary directory for writing the modified files, which should then be reviewed before merging back into the workspace.

Here is a snippet of code which is the output from the unit test used for the no-undo refactoring. All of the *no-undo* options on *define* statements were added by the no-undo refactoring, but more interesting is to note where the refactoring did not add *no-undo*.

```
/* n o - u n d o . p
 * This file contains tests for our  n o - u n d o  refactoring.
 * IMPORTANT!! Comments containing "u n d o" (without the spaces)
 * have an impact on the refactoring's behaviour!
 */

{&_proparse_ prolint-nowarn(noundo)}
define variable myInt as integer.
```

```
procedure myProc1:
  define input parameter p1 as logical no-undo.
end.

procedure myProc2 external "whatever.dll":
  define input parameter p2 as long.
end.

/* Test for  u n d o  statement. */
define variable myChar as character.
define variable myChar2 as character no-undo.
define variable myChar3 as character no-undo.
do:
  mychar3 = "".
  do:
    myChar = "".
    undo, leave.
    myChar2 = "".
  end.
end.

/* u n d o  statement tests for named block and output val */
define variable myChar10 as character.
define variable myChar11 as character.
my-block:
do:
  run changeVal(output myChar10).
  do:
    run changeVal(output myChar11).
    undo my-block, leave.
  end.
end.

procedure changeVal:
  define output parameter changed as character no-undo.
end.

/* this should remain undo */
define variable c1 as character.

/* this var should be undo         */
/* with this two line comment.    */
define variable c2 as character.

define /* undoable */ variable c3 as character.

define variable c4 as character. /* not no-undo */
```

```
/* This comment does not change undo for the next define,
   because of the blank line between the comment and the statement.
*/

define variable c5 as character no-undo.

define variable c6 as character no-undo.
/* Comment on line after does not change undo for previous statement. */
```

Table and Field Names Refactoring

This simple refactoring is mostly for aesthetics, and it has two optional features. First, for table and field names within the source, it applies the desired character style (upper or lower case). Second, for table and field names which are abbreviated, it expands those names to the full name. This second refactoring is not purely aesthetical – abbreviated table and field names have been known to be contributors to bugs.

For companies who have inherited large quantities of source which does not conform to their preferred coding styles, aesthetic refactorings such as these can help alleviate some of the time consuming frustration associated with combining two code bases.

Like the **No-undo** refactoring, this refactoring is intended for application to entire directory structures or entire projects in a single pass, and therefor uses a temporary output directory.

Here is an example:

```
/* before */
find first cust.
display cust.bal.

/* after */
find first customer.
display customer.balance.
```

Bubble Declarations

This is the first in a series of refactorings designed by Peter Dalbadie et al. As of current writing (Jan, 2004) this refactoring is going through beta testing and some design changes, as we find more simple ways of accomplishing the same refactoring goal. For that reason, code examples provided below may be out of date.

As the first step in many, this refactoring's goal is to organize a compile unit's source code into the following: shared declarations, methods, local declarations, and then finally, procedural code. We refactor the source into these logical groups because the next step in the series is a further refactoring to insert *procedure* blocks to group the local declarations and procedural code.

This refactoring is another prime example of automated analysis the semantics of the compile unit as well as the macro layers of the compile unit in order to facilitate

automated source changes.

This refactoring may affect dozens of files in order to refactor a single compile unit, therefore it uses the review and rollback mechanism described previously.

The source examples below shows what the code might look like, before and after. In order to facilitate code grouping, include files which contain mixtures of shared declarations, local declarations, and/or procedural code, are split into multiple include files. (Note that Peter is recently finding that splitting the include files might not actually be necessary, and that we might be able to deal with the include files with a more simple approach later in the refactoring series. So – the include file splitting aspect of this refactoring may go away in a future release.)

Before:

```
def shared var int1 as int.

return.

def var int2 as int.
def var c1 as char.

def shared var c2 as char.

/* c3 comment line 1 */
/* c3 comment line 2 */
def shared var c3 as char. /* c3 comment after */

/* c4 comment before
 * with multiple lines */
def {1} var c4 as char. /* c4 comment after
 with multiple lines */

on whatever anywhere do:
 def var onBlock as char.
end.

procedure myProc:
 def var procBlock as char.
end.

function myFunc returns logical:
 def var funcBlock as char.
end.

return. /* this comment
 should not move */
def var c5 as char.
```

```
{data/bubble/test/incmess1.i hello shared world}

{data/bubble/test/include2.i "new shared"}
```

After:

```
def shared var int1 as int.

def shared var c2 as char.

/* c3 comment line 1 */
/* c3 comment line 2 */
def shared var c3 as char. /* c3 comment after */

{data/bubble/test/incmess1-shared.i hello shared world}

{data/bubble/test/include2.i "new shared"}

procedure myProc:
  def var procBlock as char.
end.

function myFunc returns logical:
  def var funcBlock as char.
end.

def var int2 as int.
def var c1 as char.

/* c4 comment before
 * with multiple lines */
def {1} var c4 as char. /* c4 comment after
                           with multiple lines */

def var c5 as char.

{data/bubble/test/incmess1-local.i hello shared world}

return.

on whatever anywhere do:
  def var onBlock as char.
end.

return. /* this comment
                     should not move */
{data/bubble/test/incmess1-proc.i hello shared world}
```

Wrap Procedure Block

The second in the series of refactorings designed by Peter Dalbadie et al., this wraps local declarations and procedural code into an internal *procedure* block. There are a couple of benefits to this refactoring. First, it is necessary for converting a procedural *.p* into a persistent procedure. Second, it is necessary in Dalbadie's series of refactorings, because several procedural *.p* programs will be combined into a single, large compile unit – and the way to accomplish that is to wrap each *.p* procedural section into a *procedure* block before appending one *.p* file to another.

The effects and features of this refactoring are easy to see with an example. Before:

```
def input param p1 as char.
def parameter buffer bcust for customer.

display
"
This
string
should
not
be
indented
"
.

display "All done".

return.
```

After (note that the name of the .p is “t01.p”):

```
def input param p1 as char.
def parameter buffer bcust for customer.

RUN t01_main (input p1, buffer bcust).

PROCEDURE t01_main:

 def input param p1 as char.
 def parameter buffer bcust for customer.


 display
 "
This
```

```
string
should
not
be
indented
"
  .

  display "All done".

  return.

end procedure. /* t01_main */
```

Append Program

By combining many tightly-coupled, procedural *.p* program files into a single compile unit, we are in a new position to start *factoring* that one large compile unit into a completely new architecture. For example, our goal may be to split UI from business logic, or our goal may be to create a more object oriented architecture, or we may have both of these goals as well as combinations of other goals.

The automated refactoring takes into consideration the modifications that were done in the **Wrap Procedure Block** refactoring, and it also deals with the removal of duplicate declarations. Duplicate declarations are likely, because we are appending two or more tightly coupled *.p* program files.

The source for “t01.p”:

```
def shared var s1 as char.

def var a1 as char.

{data/appendprogram/t01/test/t01.i}
```

The source for “t01b.p”:

```
def input param bp1 as char.

def shared var s1 as char.

def var b1 as char.

{data/appendprogram/t01/test/t01.i}

run t01b_main (input bp1).

procedure t01b_main:
  def input param bp1 as char.
```

```
end procedure. /* t01b_main */
```

The result of running the automated refactor to append “t01b.p” onto the end of “t01.p”:

```
def shared var s1 as char.

def var a1 as char.
def var b1 as char.

{data/appendprogram/t01/test/t01.i}

procedure t01b_main:
  def input param bp1 as char.
end procedure. /* t01b_main */
```

This refactoring also introduces a new programmer UI element – the use of Eclipse *resource markers* and the **Tasks** view. Resource markers are persistent markers for any resource – project, directory, or source file. They may mark the resource, a line, or a selection of text within a file. The **Tasks** view displays a list of markers of selected types, and double-clicking on a marker takes you to that resource, or to the specific line of code or the specific selection.

This refactoring searches within the compile unit's source files for the name of the appended program file. In this example's case, within the “t01.i” include file, the following line of code was found:

```
run data/appendprogram/t01/test/t01b.p (input "yada").
```

A task was created, indicating that the string “t01b.p” was found at line two of the include file. By double-clicking on that task, the programmer is taken directly to the line of code, where they can examine the usage of the filename and make a decision as to whether or not the code needs to be modified. In this case, it should be changed to:

```
run t01b_main (input "yada").
```

For the time being, we do not attempt to automate these changes, due to the likelihood of the program name appearing in string literals, etc. It is a quick and simple change for the programmer to make.

Near Future Refactorings

The following is a partial list of some of the refactorings that we are considering for the current project. In other words, we expect most of these refactorings to be built and beta tested over the next couple of months.

Extract Method

This is a common, published refactoring. The programmer selects a section of source

code (blocks, statements, or even a single expression), and the automated refactoring:

- creates a new method (*procedure* or *function*), and moves the selection to it
- replaces the removed text with a call to the method
- takes care of any parameters that are necessary for passing between the method and the caller

Move Method

This refactoring moves a method from one compile unit to another. This would be frequently used in combination with **Extract Method** when factoring a large compile unit into a new architecture (OO, split UI and BL, etc.).

Create Accessors

This is not so much a refactoring as just simple code generation. Given a data element, *accessor* methods (*function* or *procedure*) are created. (ex: *getMyVar()*, *setMyVar(val)*)

Replace References with Accessors

This is another common, published refactoring. Wherever a data structure is accessed directly, that reference is replaced with the use of an accessor method.

Extract Data Structure

Like **Move Method**, this moves code from one compile unit to another, except in this case, it is moving a data structure (variable, frame, table, etc.) from one compile unit to another. It is used in combination with **Create Accessors** and **Replace References with Accessors**. This refactoring can combine those two other refactorings into this one larger refactoring. It moves the data definitions and accessors from one compile unit to another, and ensures that the data references from the first compile unit are changed such that they make use of the accessor methods which are now in the second compile unit. Like **Move Method**, this would be used frequently when factoring a large compile unit, especially for separating UI from BL.

How it all Works

We use several technologies in order to perform these refactorings. The first is Proparse, our Progress 4GL parsing product. We use its *Abstract Syntax Tree* in order to examine the semantics of a compile unit. Proparse is significantly different than the parser used within Progress's compiler. Proparse generates much more of a “concrete” syntax tree, with a much larger amount of information about the original source code stored in the tree. Proparse is designed for code analysis and transformation – not for the fast code translations that need to be done within a compiler.

The second is an in-memory representation of the source files, stored as a linked list of

atomic tokens which can be moved, combined, or deleted in order to perform code changes, before writing the token list back out to disk.

Third is a preprocessor output listing from Proparse, which allows us to quickly and easily determine what include arguments, preprocessor branches, macro expansions, etc, were used when processing a compile unit.

Fourth, we use *tree parsers* which we generate using a parser generator named “Antlr” (www.antlr.org). Tree parsers allows us to quickly and easily build libraries which perform complex semantic analysis on a compile unit's syntax tree.

Closing

I hope that this review of refactoring and ProRefactor has been beneficial. If you have any questions or comments, please don't hesitate to contact me.

John Green
john@joanju.com
+1-604-767-8587