Eclipse, Refactoring, and More

John Green

Revision History

Revision 0	2004 Oct
Revision 1	2004 Dec 16

Table of Contents

Abstract

The first detailed written work about refactoring was published in a PhD thesis in 1992. Refactoring was first put into practice in the “Refactoring Browser” for Smalltalk, and since then, refactoring tools have become popular in other environments, most notably Java. Only within the past two years have products come available which provide automated refactoring for more challenging languages such as C++.

This session will briefly introduce refactoring and the concepts behind it. It will examine Joanju's research and development in the area of automated refactoring for the 4GL. Joanju's product, ProRefactor,

will be discussed and demonstrated. ProRefactor is an Eclipse plug-in, and Eclipse will be also be discussed and demonstrated. This session will also present future directions and the challenges that exist in refactoring systems written in the 4GL.

1. Introduction

1.1. Presentation

This is the text from a presentation that I gave at the PUG NL in November, 2004.

1.2. Thanks

I would like to thank Rene and The Netherlands Progress Users Group committee for inviting me to give this presentation.

1.3. About the Speaker

I have been programming in Progress since 1989. For four years I was with Starbase Corporation as lead developer on Roundtable. In early 2000 my wife Judy and I co-launched a new venture which became Joanju, where we created two new products: Proparse and ProRefactor.

1.4. Goals

The topics in this session are: refactoring, Eclipse, Proparse, and ProRefactor. By the end of the presentation, you should be familiar with refactoring and those three products, and have an idea of how they might be useful in your future 4gl projects.

1.5. Joanju History

How did I end up here, giving this presentation? In 1999 Judy and I started travelling and looking for work in Europe. Before we accepted a contract in England, we had talked to a few Progress 4GL shops. One theme that from more than one of those companies was that many of these companies had large, old systems that they wanted to modernize. They could not see any practical way to modernize their applications without doing a complete rewrite. What's worse than the complete rewrite was that some of they were thinking: “Well, if we have to rewrite the whole thing anyway, isn't now the right time to consider a platform change?”.

I'd like to get a show of hands: How many people here are at a company where you are either in the middle of a large modernization project, or you are *considering* a large modernization project?

After our experiences, we started to give a lot of thought to a few big questions. How, exactly, does one go about performing a major architecture change to a huge application? Can some of this process be automated? We knew that it would be extremely difficult to build the tools necessary to automate these things, but we were silly enough to take a shot at it anyway.

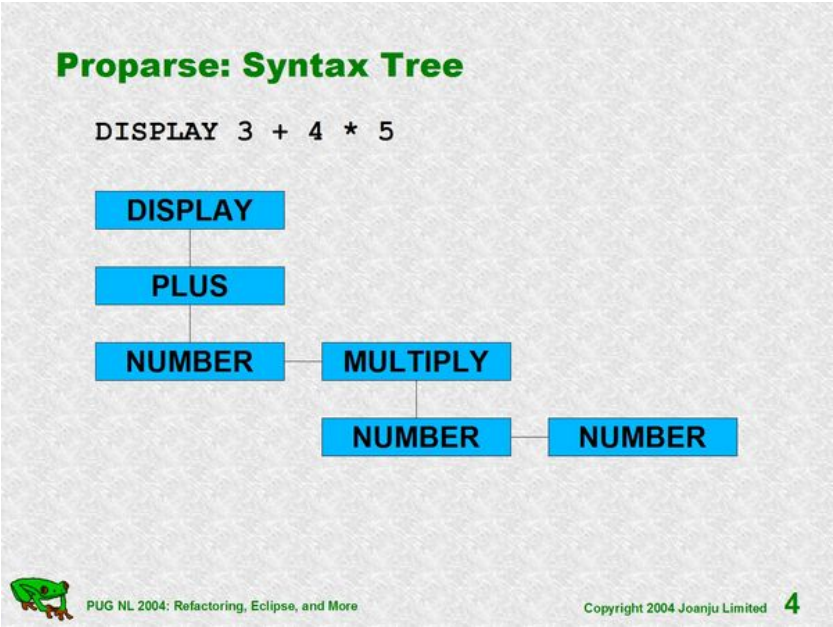
2. Proparse

In order to build tools for changing Progress code, you have to have a tool for *reading* that Progress code. Our initial prototype was built using a proprietary Lisp system, but after about six months, we decided that we wanted to be able to sell the parser as a small DLL. We rewrote the parser in C++. In mid-2001, we made a public beta of Proparse available, and Jurjen Dijkstra picked that up and built Prolint within about six weeks worth of evenings and weekends. We were delighted, of course.

Since Proparse is a DLL, we provide it with a simple 4GL API, and it has been used for various projects by various companies. It has been used in lint-like projects, code internationalization, Oracle gateway projects, code documentation, and various other projects where the parser could make code analysis much more fast and reliable than it would be to do it by hand.

I'll do a really quick review of what parsers and compilers do. The first step is the preprocessor, if any. The preprocessor scans the source code and looks for special mark-up, where include files are to be added, and where text substitutions are performed. The preprocessor generates a *character stream*, which is read by the *lexer*. The lexer reads the character stream, one character at a time, and groups the characters into *tokens*. A lexer is also normally responsible for discarding unnecessary text, like white space and comments. The *parser* reads that token stream, and arranges those tokens into a tree structure, based on the syntax of the language. Proparse does everything right up to this point. A full compiler would then continue to do more work with that tree, and ultimately generate code in the *target language*, which may be machine code, byte code, or any other language.

Figure 1. Proparse Syntax Tree



The purpose of Proparse is to take Progress 4GL source code, and generate a tree in memory, where each node of the tree represents some token in the language's syntax.

Proparse provides a simple but extensive API for examining the nodes in the syntax tree. That API provides functions for doing things like finding the first child of a node, the next sibling of a node, getting the node type, getting the text of a node, or getting the file, line, or column number for the original source code that the node came from.

One difference between Proparse and the parser that you would find within a typical compiler is that Proparse does not discard the comments or whitespace. From any given node in the syntax tree, it is possible to find the comments or whitespace that precede it in the original source code. This is often useful in code analysis or code documentation.

3. Refactoring

3.1. Introduction

Refactoring is the act of improving source code without changing the behaviour of the application. There are many reasons for refactoring - we want to: make the code more maintainable; make the code easier to understand; encourage code re-use; make the code more easily *localized*, for internationalization; make the code more portable; and improve our code's architecture, so that new layers may be added such as new user interfaces, new middleware services, and so on.

3.2. Refactoring History

The first detailed written work about refactoring was published in a PhD thesis in 1992. That thesis was written by William Opdyke, who was a student of Ralph Johnson. You may recognize the name Ralph Johnson as one of the “Gang of Four” who wrote the seminal book named *Design Patterns*.

Refactoring is something that programmers have always done, but the name “refactoring” is recent, as are the names that have been created for the different kinds of refactoring that we do. In this presentation, we are mostly interested in tools which provide automated refactoring.

Refactoring was first put into practice in the “Refactoring Browser” for Smalltalk, and since then, refactoring tools have become popular in other environments, most notably Java. There are various IDEs which provide automated refactoring for Java code. Two Java IDEs of note are IDEA from IntelliJ, and Eclipse. Interestingly, most of the refactoring toolkits for Java are actually products which *plug-in* to one or more existing IDEs.

C# programmers are looking forward to Visual Studio 2005, which will add refactoring as a built-in feature. For C# programmers today, there are third party products which provide refactoring.

Only within the last couple of years have products come available which provide automated refactoring for more challenging languages such as C++. Refactoring for that language is a topic under research by one of Ralph Johnson's current PhD students. The challenge for researchers today is to find methods for allowing automated refactoring to be performed where the automation is complicated by the presence of preprocessing, and by other language features which provide tricky sorts of indirection, such as memory pointers.

Visual SlickEdit version 9, which was released this year, claims to be the first commercial editor which supports C++ refactoring.

3.3. Refactoring Examples

A lot of examples of refactorings that have been documented and given names are related to object oriented programming. Examples of those are refactorings which push members higher up into class hierarchies or pull them deeper down into the class hierarchy, refactorings which split one class into two classes, and refactorings which extract interfaces from a given class.

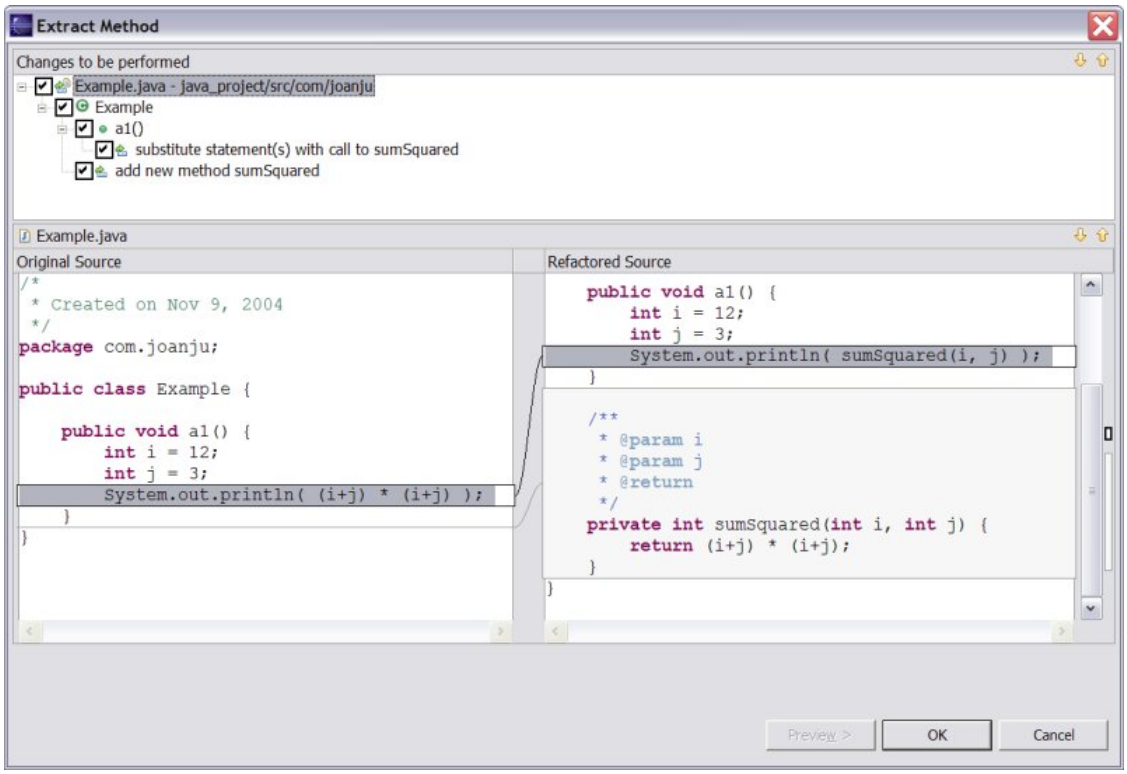
There are, however, plenty of examples of documented and named refactorings that could be applied to 4GL code.

One of the simplest examples of refactoring is the Rename refactoring. When done by hand, a programmer would perform a search for the old name and replace it with the new name where appropriate. A tool which performs the refactoring automatically does not perform a simple search and replace. In fact, it makes use of a parser, so that it can be sure that it is only changing the source code where appropriate.

On Martin Fowler's web site Refactoring.com [http://www.refactoring.com/], he describes a *Rubicon*

which he uses as a benchmark for determining if a product is serious about refactoring or not. The term Rubicon sent me straight to the dictionary. It is used to describe a point where one is truly committed to their course of action. Fowler's Rubicon for refactoring products is the Extract Method Refactoring.

Figure 2. Preview of Extract Method Refactor



When programmers perform an Extract Method refactoring, they select a block of code that they want to pull out into a standalone function. The IDE has to prompt the developer for the name of the new function to create. It also has to determine what arguments are required for the new method, and what its return value should be (if any). The IDE allows the programmer to make adjustments to the names and attributes of the method and its arguments. The selected segment of code is replaced with a call to the new function, and the function's code is generated.

Fowler uses the Extract Method refactoring as his Rubicon because of the extent to which the syntax tree must be analyzed in order to perform the refactoring. The scope of all of the symbols in the affected block of code must be determined. Although a refactoring like Rename refactoring must be sensitive to the syntax of the language in order to work, it does not require as much in-depth analysis of the syntax tree.

Other quick examples of documented refactorings which could be applied to Progress code are: Move Method refactoring, where a function or internal procedure would be moved from one persistent procedure to another; and encapsulated field, where a shared variable would be made non-shared, and access to it would be replaced with *getter* and *setter* functions.

As an example of where we might invent some specialized refactorings, you might consider software design patterns that were used in the past, but now need to be replaced with newer design patterns. Another consideration is for syntax which has been deprecated, where a clearly defined sequence of steps could be applied in order to replace the deprecated syntax with something new.

3.4. Refactoring in Real Life

There can be a difference between traditional programming and the kind of programming that happens when you approach your work with refactoring at the front of your mind. In one approach to re-engineering a part of an application, you might analyze it as it is today, design what it needs to look like once finished, and then sit down and start changing all of the affected source files, until you are ready to test. For the duration of the time that you have your source files “in pieces all over the floor”, the application is typically “broken” - you cannot execute or test the part of the application that you are currently operating on. When you take a refactoring approach to a re-engineering problem though, you find that you tend to break the problem down into a series of smaller refactorings that can be performed one at a time. After each small refactoring, your application should work just as before, but you should be one step closer to your engineering goal.

Refactoring is often associated with *Extreme Programming*. When doing Extreme Programming, there is a large emphasis on automated regression tests as well as on continuously refactoring. Refactoring and running the regression tests are done in cycles, and those cycles are kept as short as possible.

When should we use refactoring? We should use it to continually improve the architecture of our source code as we are building a new system. We should use it to keep our code easy to maintain and understand when we are doing bug fixes. Finally, we should use it when we are doing maintenance programming in order to prevent *source rot*. Source rot is what happens when well designed code deteriorates over time due to the patchwork that happens during bug fixing and application feature enhancements. So, if we should use it when building, bug fixing, and maintaining, then the bottom line is that we should use it all the time!

3.4.1. The Value of Automation

The first time you see an automated refactoring in action, you may wonder if it really makes much difference in a developer's overall productivity. After all, it looks like it would only save you a few minutes here and there.

Let's think about it from another perspective. If a developer is in the middle of a large task, and they come across a function that is poorly named, are they likely to fix it? If they are in the middle of a large task, and they come across a segment of code that really should be pulled out into a separate function, are they likely to do it? Probably not, because even if it only takes a few minutes, those things are distractions that would require the developer to shift their attention from one task to another.

When there are tools available to automate the refactorings, the refactorings tend to happen much more frequently. Because the tool takes care of the details, the refactoring does not distract the developer from their main task. As a result, the application's source code starts to become more clean and better architected just as a side effect of the developers regular work. Your code becomes easier to understand, easier to maintain, and your company is in a much better strategic position to move the application forward in the future.

4. Eclipse

4.1. Joanju and Eclipse

In the middle of 2002, I started evaluating various languages, frameworks, and development environments for a *workbench* that I could place Proparse-based tools upon. I needed a workbench where I could add menu items, dialogs, windows, preferences, project settings, and so on. At the same time, I was evaluating Java as an alternative to C++ for code analysis and refactoring.

I looked into various editors and IDEs which were extensible or used a plug-in style of architecture. After a few months I chose Eclipse. Eclipse is free, open-source, and its license explicitly allows for anybody to repackage it as a commercial product.

4.2. Eclipse History

Some time around early 2000, IBM decided internally that they wanted a common IDE framework for some of their various projects and products, such as WebSphere. They began construction of this common framework, and soon realized that to make the most of it, they needed to open it up to a community of partners. In 2001, the open source project was given a board of stewards to oversee the development of “eclipse.org”. Early this year, Eclipse was reorganized into an independent, non-profit organization.

The names of the other companies who have become involved in Eclipse are mostly pretty obscure, but I'll put a few of those names up on the screen now:

Figure 3. Eclipse Contributors

Today, there are dozens of companies who provide significant contributions to the Eclipse non-profit corporation and to Eclipse in general. There are also many dozens of commercial and free plug-ins available for Eclipse, ranging from support for various compilers and languages, to enterprise development environments, to modelling tools, and so on.

In a recent survey by BZ Media of 719 individuals, 69% stated that Microsoft Visual Studio was being used within their organization. How did Eclipse rank? 54% stated that Eclipse was being used within their organization. I find it remarkable that for such a young product, Eclipse is being used within the organizations of more than half of the respondents! Keep in mind that this was a survey of IT professionals in general - not just those who use Java. Of the reasons cited for the use of Eclipse, the top reason was that it is open-source, followed closely by the fact that it is free, and then by its extensibility.

4.3. Eclipse Overview

When giving a very brief overview of Eclipse, I like to describe two things that Eclipse is, and one aspect of Eclipse which I consider to be interesting to programmers like myself.

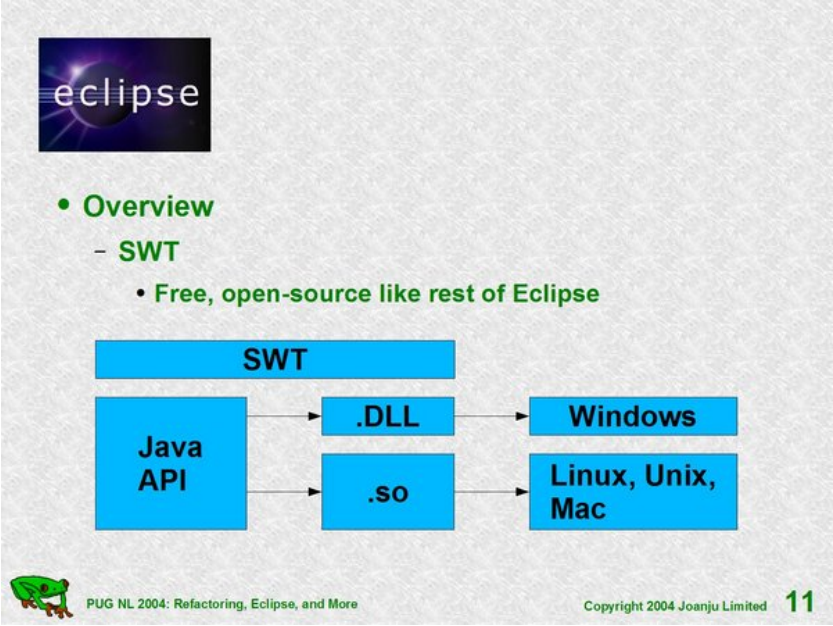
First, Eclipse is a framework for creating an IDE. This extensible framework is enabled by its plug-in architecture, which I will describe in more detail a little later on.

Next, Eclipse is an IDE for various languages and platforms, Java being the platform with the most mature support.

4.3.1. SWT

Now I'll describe one aspect of Eclipse that I find interesting. The user interface for Eclipse is built upon a widget toolkit called SWT (Standard Widget Toolkit). Like the rest of Eclipse, SWT is free and open source. It has been ported to several platforms, including Windows, Mac, Linux, and various unixes. For each supported platform, there is a DLL or shared object. There is one Java API which provides access to SWT regardless of the underlying platform.

Figure 4. SWT



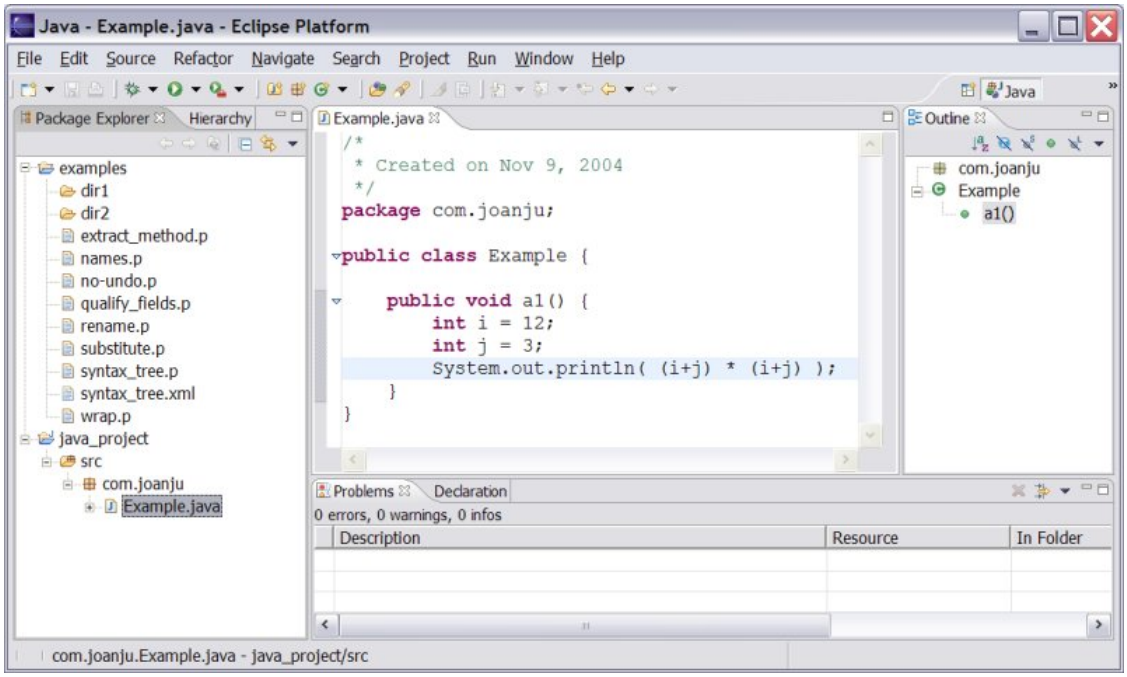
What makes SWT interesting is that it provides access to the platform's *native widgets*. This is in distinct contrast to Java's Swing UI, which is rendered by the Java Virtual Machine. A common complaint about Swing is that applications built with it feel sluggish, and have a look and feel which is not quite the same as the platform's native widgets. In contrast, an SWT application like Eclipse has the same snappy look and feel of any other native application.

There are other open-source editors and IDEs which allow for plug-ins, but I believe that SWT is one of the factors in the huge success of Eclipse.

4.4. Eclipse Components and Terms

Here, I'll introduce how some terms are used by Eclipse.

Figure 5. Eclipse Workbench



Workbench
: Eclipse's Workbench is the entire Eclipse window and all of its contents. You may on occasion have more than one workbench active, but normally you would only work in one workbench at a time.

Workspace
: A workspace consists of the list of projects and settings that you see in your workbench. The information about a workspace is stored in a workspace directory, and you may have more than one workspace.

Project
: The term *project* has the same meaning as most other IDEs. A project has a single, top-most directory, which in turn contains all of your project's sub-folders and files.

Resource
: A resource in Eclipse is a project, a directory, or a file. In simple terms, it is anything that you can see in the Navigator.

View
: Within the workbench window, you can find various frames which are separated by movable splitter bars. In Eclipse, these frames are called *views*. Access to the views is through a tabbed interface. Examples of views are the Navigator view, the Outline view, the Console view, and the Tasks view.

Editor
: Editors in Eclipse are different than views, especially as far as the implementation goes. Multiple editors can be open with a tabbed interface, and there can be an unlimited number of types of edit-

ors within your Eclipse workbench. The default, bare-bones Eclipse platform comes with a simple text editor. The Java IDE plug-in provides an editor for Java, and there are plug-ins which provide editors for C++ and other languages. There are also plug-ins which provide graphical editors, for doing GUI building, for doing UML modelling, and so on.

Perspective
: A perspective describes an arrangement and visibility of the various elements on the workbench. That includes the menus, button bars, views, and editors. There are default perspectives for Java, Java Debugging, for ProRefactor, and for many other plug-ins. Although perspectives all come with a default layout, perspectives can be modified. Eclipse remembers your perspective changes.

4.5. Plug-in Architecture

Eclipse uses a plug-in architecture. This means that the Eclipse core is extremely small, and most of the elements that you see on the Eclipse workbench are actually contributed by plug-ins - even those elements which are part of the basic workbench. In fact, even SWT itself is just another plug-in. There are also plug-ins which are not part of the basic workbench, and those include Java and C++ plug-ins. Those plug-ins come from sub-projects under the umbrella Eclipse.org project, but there are also many dozens of plug-ins that are provided by third parties.

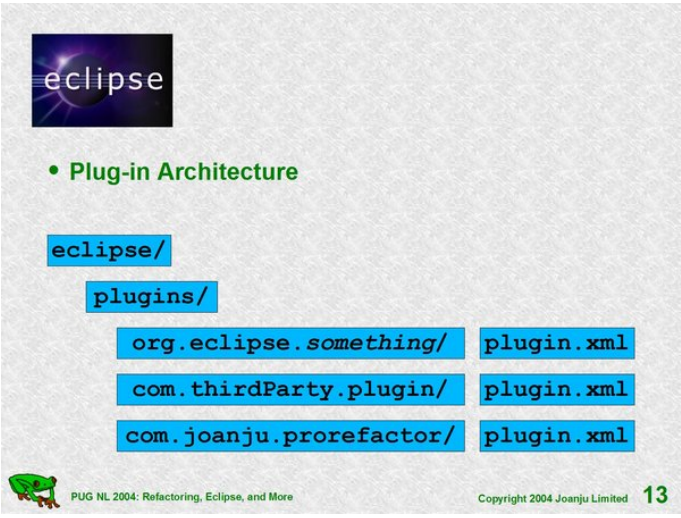
One of Eclipse's design goals is to allow for literally hundreds of plug-ins to all be working together within the same workbench.

4.5.1. Technical Overview of Plug-ins

In the Eclipse application directory, there is a subdirectory named `plugins`. To add a new plug-in to Eclipse, you essentially just add a new subdirectory to the `plugins` directory, and restart Eclipse. In the plugins directory you will find dozens of `org.eclipse.`*`something`* subdirectories, all of which make up the Eclipse workbench itself. Additional subdirectories would come from third party plug-in providers like myself.

Each plug-in subdirectory is expected to contain a file named `plugin.xml`. That file is the *plug-in manifest file* - it describes the menu items, buttons, views, and actions that are contributed by the plug-in. For each of these contributions, the plug-in manifest also provides the name of a Java class to be loaded which does the actual handling of the action. These Java class files are found in the plug-in's subdirectory, and they must follow a strict API which is defined by Eclipse.

Figure 6. Eclipse's plug-in architecture



5. ProRefactor

ProRefactor is an Eclipse plug-in. However, it is architected such that all of the refactoring features of the product could be launched independently of Eclipse. To some extent, the Eclipse workbench is just a handy place to add menu items and other UI components. We'll have a look at what ProRefactor contributes to the Eclipse workbench.

ProRefactor contributes a new project type. Eclipse is a unique sort of an IDE, in that it is an integration point for many different contributions. This becomes interesting when dealing with projects. In Eclipse, one single project may have a mix of, say, Java, C++, and Progress source code. Eclipse uses a concept that it calls *project natures* to allow for mixed projects like this. In the case here where I created a ProRefactor project, I've created a project with a ProRefactor nature. Other natures may be added to a project after it's been created.

ProRefactor contributes Project Property pages. We use one page for configuring Proparse, which needs to know about some of your Progress settings in order to correctly parse your source code. We use another page for configuring ProRefactor itself, with certain environment settings and code refactoring preferences.

Eclipse uses the term *extension point* to describe any part of the workbench that a plug-in may contribute to. ProRefactor uses an Eclipse extension point to contribute a menu to the workbench. An interesting feature of the Eclipse architecture is that any plug-in may declare its own extension points, so that your plug-in may be extended by other plug-ins. Although I have not done it yet, within ProRefactor I should contribute an extension point so that third-party plug-ins could contribute additional menu-items to my ProRefactor menu.

ProRefactor contributes a *context menu* to the navigator view, as well as to certain editors. ProRefactor contributes a utility for selecting a block of code, which can be accessed either from the editor's context menu, or else from the buttons that are visible when the appropriate editor type is being used. Although this utility has nothing to do with refactoring, it was a nice idea that was suggested by a customer. ProRefactor gets the file, line, and column position of the cursor in the editor. It then figures out what node in Proparse's syntax tree that position relates to. When you hit the **Expand** button, it climbs up the syntax tree by one node, and changes the editor such that the selected block of text is all of the code that falls within that branch of the syntax tree. We can repeatedly expand our selection until we reach our goal, which is to match the beginning and ending of a particular block.

5.1. Refactoring Features

Now I'll describe the refactoring features that we have added to ProRefactor so far.

5.1.1. NO-UNDO Refactoring

One of the first features that we added was the NO-UNDO Refactoring. To use this feature, you select one or more files, or folders, or projects, or any combination of those in the Navigator view, and then use its context menu to launch the refactoring. This refactoring will go through all of the selected code, and wherever there is a DEFINE statement for a variable, parameter, or a temp or work table, it will add the NO-UNDO option if it is missing. That by itself is fairly simple, but ProRefactor adds a couple of nice features to this.

The first is related to Prolint. Prolint would normally give you a warning about a DEFINE statement that is missing the NO-UNDO option, but you can use a special directive in the code to tell Prolint that the missing NO-UNDO option is intentional. ProRefactor also respects this directive, and it does not add NO-UNDO if it is present.

Another feature is that the refactoring logic looks for the string "undo" in comments for the statement. If that string is present, then it is assumed that the definition is intentionally missing the NO-UNDO option.

Finally, this refactoring looks at the syntax tree to determine if the variable, parameter, temp or work table in question is ever assigned a value inside of a block that has an explicit UNDO statement. In that case, ProRefactor does not add the NO-UNDO option to your DEFINE statement. This is a good example of slightly more in-depth analysis of the syntax tree in order to perform a refactoring.

For all of the source files that get changed, ProRefactor writes the newly modified files out to a temporary directory of your choice. That way you can review all of the changes before you copy them into your project directory, using any of the excellent free and commercial third-party utilities designed exactly for this sort of thing.

5.1.2. SUBSTITUTE Refactoring

Like the NO-UNDO Refactoring, you can select any combination of resources from the Navigator view. This refactoring works differently though, in that it is interactive, and it modifies your source code right within your project directory.

The purpose of this refactoring is to find instances of string fragments which are concatenated, and replace those with the SUBSTITUTE function, using one single, larger string. This is for internationalization projects where translation is required. It is easier to translate one full sentence than it is to translate a bunch of sentence fragments.

The Review Changes dialog box allows you to fine-tune the changes before you accept them. You may also reject the changes, or cancel the whole operation.

In the Review Changes dialog, we can also see some of the nice features of SWT. We have movable splitter bars, a dialog that resizes nicely, and we're using a rich editor which allows us to highlight lines of text.

5.1.3. Qualify Field Names

This refactoring is performed only on the source file that you currently have open. One of the interesting features of this refactoring is the use of Eclipse's built-in *compare* support, for visually presenting the modifications. This refactoring does enough analysis of the scopes and buffers in the syntax tree in order to determine which buffer is implicitly being used by Progress for an unqualified field name. Also, there is one type of syntax where it is completely unnecessary to qualify the field names, and that is in phrases like FIELDS, USING, and EXCEPT. ProRefactor examines the syntax tree to determine the context of field references, and it does not qualify the field names in those cases.

5.1.4. Table and Field Names Refactoring

This is another refactoring which is performed against any combination of resources selected in the Navigator view. You would typically use this refactoring utility to bring old code up to new standards and conventions. As can be seen from the wizard, there are options in this refactoring for changing name references to upper or lower case, fixing unqualified field names, and expanding any abbreviated names.

5.1.5. Rename Schema Refactoring

The refactorings that I've described so far have all been of a “one time” nature - they would be run once through the source of an older application in order to bring it up to newer standards. The Rename Schema refactoring is one that would be used on an ongoing basis. It goes through your code, and changes all of the hard-coded references for schema tables and fields which are to be renamed. It takes a list of old and new names as its input. Because it uses the parser rather than a simple search and replace, it is not fooled by buffer names, abbreviated names, or unqualified names.

5.1.6. Extract Method Refactoring

What about Fowler's Rubicon - the Extract Method refactoring? We're part way there. We have built into it the determination of the scope of the variables in play, and how those must be passed as arguments. Internally, ProRefactor is also aware of buffers and their scopes, but we have not yet used that semantic knowledge to perform appropriate code generation in this refactoring tool. We have not started looking into frames and their scopes yet.

6. Challenges

6.1. Dynamic Code

One of the caveats of our Rename Schema refactoring is that it does not attempt to refactor dynamic code. For example, you can have your table or field names within quoted strings in your source code, where those quoted strings are used for dynamic queries. In that case, this automated refactoring can not help. Those have to be found and modified by hand.

Would it be possible to automatically change those as well? To some extent, it would be. It is possible to perform data-flow analysis, and determine which strings are passed into dynamic queries. However, there are reasons why that might not be a complete solution, for example it is also possible that some strings or parts of strings are coming from sources external to the code, such as from a database, from text files, or even from user input.

Another example of dynamic code is with the use of *handles* within the 4GL. Because Progress's handles are not associated with a *class*, Progress is essentially loosely typed, and we would again have to get into data flow analysis in order to resolve language features such as DYNAMIC-FUNCTION.

6.2. The Preprocessor

As you can probably imagine, it takes a pretty large effort to perform sufficient syntax analysis in order to gather all of the semantic information necessary for automated refactoring.

Now, if you add Progress's preprocessor as a layer on top of all that semantic information, the number of complexities multiplies dramatically.

The use of include files and the preprocessor make it extraordinarily difficult to build tools for automated refactoring. In fact, the use of preprocessing can make it difficult to even perform certain refactorings by hand! One example is the Extract Method refactoring. When there is preprocessing present, you can't just grab a segment of code and move it around. Semantically, it is straightforward to determine all of the symbols and scopes that are in play. For the preprocessor though, you have to determine all of the text substitutions that are in play, and in fact, all possible combinations of text substitutions! By moving a segment of code, you could potentially be breaking or changing the effect of any preprocessing that happens to be present in that segment.

One workaround that we use is to simply restrict the tools so that they are only available in cases that are not complicated by presence of preprocessing. Unfortunately, we find that for much of the 4GL code that exists today, this is a fairly significant limitation.

6.2.1. My Unpopular Opinion

I have an opinion which is largely unpopular and often meets with a good deal of resistance, but I'm going to throw it at you anyway.

We find that there are a large number of excellent products for automated refactoring in Java. Those tools are relatively easy to build for the Java language, because it does not use a preprocessor.

Another factor which benefits Java is that there are loosely typed pointers or handles in Java. Everything is done using a strongly typed *reference*.

Java is a much smaller language than Progress, which helps, but the effect of the size of the language turns out to be minimal compared to the effect of the presence of preprocessing.

In my opinion, the use of preprocessing makes code brittle. Software should be *soft* - you should be able to grab a segment of code and move it around. You should only have to be aware of the syntax and semantics of the code in play. You should not have to also worry about text substitutions that might be taking place at the macro level.

7. Opportunities and Directions

There is a pretty good list of opportunities for us and for others that come out of combining Proparse and the Eclipse IDE.

7.1. Prolint

I'll make the assumption that everyone here is familiar with Prolint, and I'll just point you to prolint.org [http://www.prolint.org/] if you are not. The potential for the integration of Prolint into the Eclipse IDE is fantastic. Eclipse provides a number of features which are ideal for lint plug-ins, such as the Tasks view, decorators down the sides of the editors for marking problem code, and rich editors which allow for things like underlined text.

7.2. Code Explorers and Highlighters

I'm aware of at least one private project already underway which combines Proparse and Eclipse in order to build tools for highlighting code and searching through code. Interestingly, that project keeps Proparse's syntax tree persistently stored on disk, so that those syntax trees can be easily searched through.

7.3. XREF Extensions

Progress's compile with XREF generates much of the program cross reference information that programmers need to do their jobs, but there are always some key pieces that are missing. One example has been the lack of sufficient information about the parameters required for functions and internal procedures. Without that piece of cross reference information, there have not been many tools available to statically check for parameter mismatches, and those result in errors at run-time. By using Proparse's syntax tree, we could build an extended cross reference database, which would be used to statically check for errors like this. Even more exciting is the opportunity for adding automatic code completions as well as pop-up information about the parameters for functions and procedures that you reference within your code.

An additional cross reference extension that could be built has to do with the data flow analysis that I mentioned earlier. Finding all call references to a program, a function, or an internal procedure is limited, due to features of the language such as RUN VALUE and DYNAMIC-FUNCTION. By using data flow analysis, we should be able to fill in some of the blanks in our cross reference databases.

7.4. More Refactorings

Aside from completing the Extract Method refactoring, there are a number of other refactorings that are high on the priorities list. Move Method would move a function or an internal procedure from one compile unit to another. Change Method Signature would allow you to change the name or parameters list for a function or internal procedure, and automatically update all code that references the method. Rename Shared Element would allow you to rename something like a shared frame, and the call tree would be examined to determine what other compile units need to be automatically updated.

8. Summary

One goal of this presentation was to introduce Eclipse and refactoring in the context of working with the 4GL. If you were already familiar with Eclipse and refactoring, then I hope that this presentation has given you some ideas about using of these technologies in your own development environment.

8.1. More information

For more information, please contact me by emailing john@joanju.com, or by visiting our web site at www.joanju.com [http://www.joanju.com/]. Additional sites mentioned within this presentation were prolint.org [http://www.prolint.org/] and refactoring.com [http://www.refactoring.com/].