JGRASP: a code analyzing tool.

By Tom Thao, Chaymus Klang, and Ben Talberg

May 9, 2006

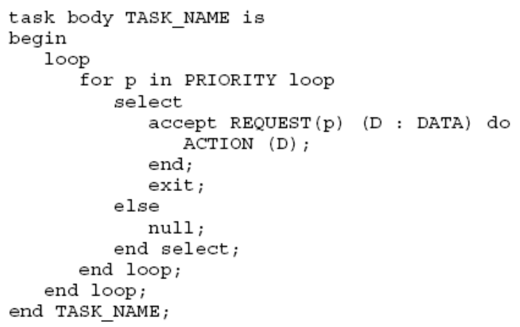
CS5802, Spring 2006

The main focus of our group project was to find and research a product that could parse source code and produce useful UML diagrams. One team member had some exposure to a product called JGRASP. After some initial readings, we thought that the tool was interesting enough to read up on. We focuses on three main areas questions to research:

1) Why would someone use this product?
2) What are some of the more general concepts and ideas on how the application could produce UML documents from source code?
3) What are the some of the exact functionalities of JGRASP?

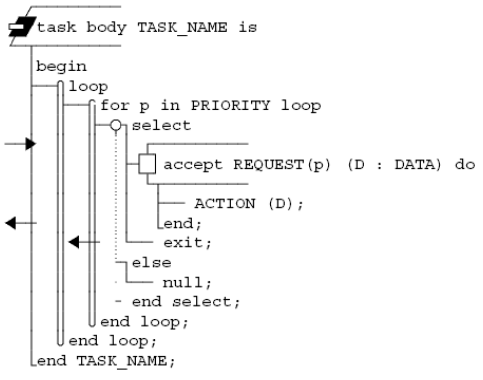
To understand JGRASP, we look at the history of it. JGRASP is the newest version of another tool called GRASP (Graphical Representation of Algorithms, Structures and Processes. The application is a small application and is a direct result of research work done by Dr. James H. Cross II from Auburn University in Alabama. Dr. Cross's goal it seemed was to improve the comprehensibility of source code, thereby reducing overall software development costs.

The GRASP tool was developed as a code analysis tool. Given an application's source code, the GRASP tool would produce two main constructs; 1) a Control Structure Diagram(CSD) and 2) a Complexity Graph Profile(CGP). The CSD would give graphical symbols to illustrate areas of sequence, decision, or iteration. Someone could basically scan a CSD and follow the code easily. Figure 1 is an example of an ADA 95 program and Figure 2 is the corresponding CSD structure that is created.



```
task body TASK_NAME is
begin
   loop
      for p in PRIORITY loop
         select
            accept REQUEST(p)  (D : DATA) do
               ACTION (D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end TASK_NAME;
```

Figure 1. Ada 95 source code

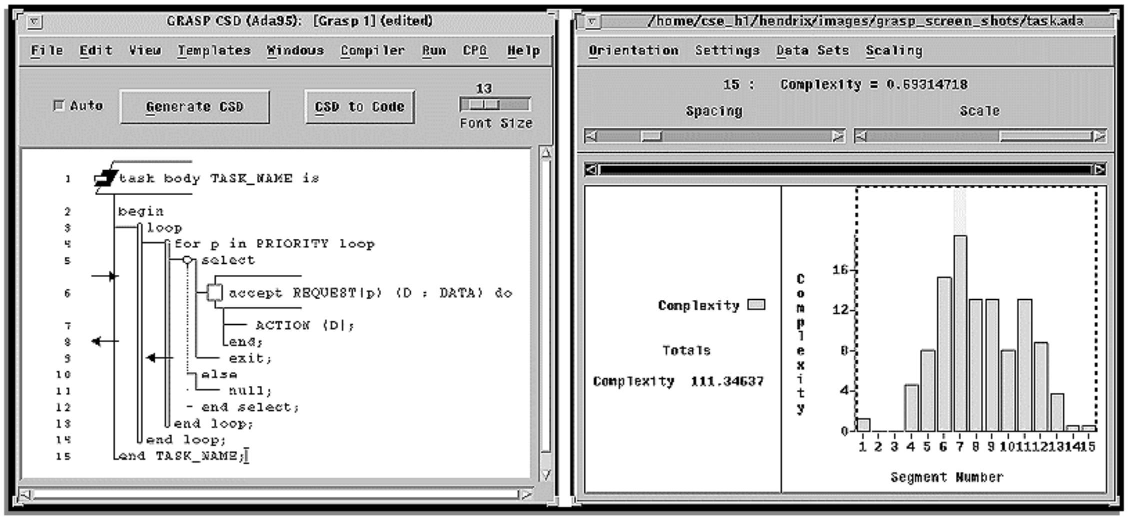


```
task body TASK_NAME is
begin
loop
for p in PRIORITY loop
select
accept REQUEST(p)  (D : DATA) do
ACTION (D);
end;
exit;
else
null;
end select;
end loop;
end loop;
end TASK_NAME;
```

Figure 2. Ada 95 source code rendered as a CSD

With the CSD structures in place, you can clearly see the beginning and endings of loops, identify areas decision, etc.

The second construct from the software application was the CGP. The graph is useful in identifying areas where code could potentially be complex to troubleshoot. In Figure 3., the CSD is displayed alongside the CPG. Analyzing the CPG, it is apparent that the program is flagged as more complex at lines 6 and 7. It is done accurately because function calls here make it harder to troubleshoot.


GRASP CSD (Ada95): [Grasp 1] (edited)
File Edit View Templates Windows Compiler Run CPG Help
Auto
Generate CSD
CSD to Code
13
Font Size
/home/cse_h1/hendrix/images/grasp_screen_shots/task.ada
Orientation Settings Data Sets Scaling
Spacing
Scale
Complexity
Segment Number

```
task body TASK_NAME is
begin
  loop
    for p in PRIORITY loop
      select
        accept REQUEST(p) (D : DATA) do
          ACTION (D);
        end;
        exit;
      else
        null;
      end select;
    end loop;
  end loop;
end TASK_NAME;
```

```
15 :   Complexity = 0.69314718

Complexity

Totals

Complexity  111.34637
```

Figure 3

In his research, Cross II has also found other uses for this application. Computer science students liked the debugging aspect of the tool. Cross followed a group of computer science students and surveyed them on the use of the JGRASP tool. He found that 14 out of 19 students new to programming preferred to view their code using a CSD view versus a plain code view. Using the GRASP tool as a debugger, Cross found it also easier to explain concepts of Object Oriented Design. Students could easily step into their applications and see the function calls being made. Cross has also written a number of articles about the use of the tool for reverse software engineering legacy applications.

The GRASP tool set the foundations for JGRASP tool. Where the GRASP tool was using in debugging applications, identifying areas of complexity, and creating a nice CSD, the JGRASP tool would take the functionality to the next step, the creation of UML documents based on the source. Let's now examine some of the ideas behind how UML documentation can be generated from source code, namely Reverse UML.

Reverse UML is accomplished by running a string parser over source code to identify compiler keywords and the named identifiers associated with them. For programming concerns we will define the process abstractly, so when a "new class is created" for our UML representation, we assume the software does whatever is necessary to allow us to view it in acceptance with the UML 2.0 Standard. The process of reverse UML itself can be split into four major phases: Class Identification, Method Construction, Class Attributes, and Class Relations. JGRASP itself only focuses on 2 of these, the class identification and the class relations.

Class Identification

The first step in creating a class diagram from source code is to analyze which classes are being used in the program. There are several ways to do this. Namespace identification, byte code analysis, and object instantiations. Namespaces are identified by the class definition with a parse return of "class <ClassName>…" This represents the actual class itself and comes in handy for identifying what classes have been defined for the program. A good example of class identification by Namespaces is the Reveal program for C++. Byte Code analysis looks at compiled source and analyzes the available classes based on the intermediate level output between source and machine level. It can identify new classes by looking for calls to the labeled class. SuperWomble does this using Java. The third type of identification involves looking for the keyword "new <ClassName>();" in the source code. This works by looking at object instantiations of a class and setting aside that information to be built upon later. This has the advantage of only listing classes that are actively involved in the program which is useful for system level classes. If the program utilizes and new int[]; It will only list the Integer class and not all system defined classes.

JGRASP itself uses a combination of namespaces and Object Instantiations to develop its UML diagrams. It does Object instantiation to determine all the classes used within source code. If a class is not used, it is detected in the Namespace approach during class definition.

Once the classes have been uniquely identified a UML representation of each class is stored inside the program.

Method Construction

Method construction is the identification of functions/methods or interacting structures within a class. That being said it makes sense to begin our parsing search within a class definition to detect its methods. We begin parsing within the {….} immediately following a class declaration. Inside this area we identify methods as containing these three properties, a return type, a method name, and a parameter list. In source code it looks like <returnType> <methodName> (<args>) {….}. We extract the return type and the methodname and add this to our UML representation in the form of: methodName:returnType under the classes representation.

Class Attributes

Similar to method detection we can retrieve the data information from a class by looking within the class definition. They are detected with <accessModifier> <type> <attributeName>; The access modifier tells us whether the data is public, private, or protected and it's role within the program and who it's available to. Type defines what data structure the attribute is, and the attribute name allows us to recognize it. We then process this into our UML representation using standards. So for example a public double age; becomes + Age:double in our representation. This information is added to the attributes section of our class within our class diagram.

Class Relations

Class relations can be split into several different types: Association, Multiplicity, Directed Association, Aggregation, Composition, and Inheritance.

Association is the basic type of relation shown in our UML representation, it is a line drawn between the classes. The string parser identifies this by looking through the class and looking for object instantiations and method calls to other classes.

Multiplicity is identified when the code analysis detects a new object. When an object is instantiated by itself it gives a multiplicity of one, if we instantiate more objects from the same class we increment this. Another multiplicity occurs when our object is instantiated within a loop construct. If a loop construct contains the instantiation we see the possibility for unknown multiplicities and define a * multiplicity. If there is a conditional that includes the instantiation we see the 0…* multiplicity relation. We place the multiplicity count next to the class that is being instantiated.

Directed Association occurs when we instantiate a class and it does not interact with the creating class. If we only instantiate the class and do not continue to call methods from it we draw an arrow from that class, to the instantiated one.

Aggregation is detected within a classes constructor. If the constructor contains other class instantiation we draw a white diamond on the class, and a line to all classes instantiated within the constructor.

Composition is similar to Aggregation except we detect it within the class deconstructor and draw a solid black diamond.

Inheritance is realized through the use of the keywork extends <className> within the class definition. If a class extends another class we draw a triangle from the class that is being extended to the class we are parsing.

Putting all of these phases together we are now capable of creating a full class diagram from our source code that adheres to UML specifications and represents what the source code is defining. JGRASP is relatively lightweight in this area and primarily is concerned only with identifying the classes involved and their relation to other classes. Even though the JGRASP tool's UML document generation feature is not as thorough as it should be, the tool's entire code analysis suite is a good application to use.

JGRASP offers a range of tools which to help programmers view, work with, and understand their code. It is lightweight application, unlike other tools such as NetBeans. So the depths of functionality the tools offer are not quite as extensive as the larger NetBeans type environment. But they are easy to use, and simply put, offer good examples of techniques that reduce coding complexities, such as UML diagram generators. In the course of this section, I'll look at JGRASP's Control Structure Diagrams, UML Class Diagrams, the Java Workbench, Debugging Viewers, Complexity Profile Graphs, and the Documentation Generator.

The control structure diagram (CSD) is a dynamic tool that can be activated on the source code of the project. It breaks the code into a structured, indented view so that comprehensibility of structures such as if-then loops, for loops, variable declarations, and functions can more easily be read. The resulting structure looks similar to the visual depiction of files on a hard drive. By clicking on the 'plus' sign outside a given folder, a tree of files and folders inside of it are displayed. And those folders can then be unfolded to reveal their content. Using a CSD diagram, the contents of a function can be hidden, so only the name of it is shown. When a user wishes to view the inside of the function, they can click the 'plus' and unroll the contents so they are visible. Using this diagram can greatly reduce the complexities of source code, and help a coder focus on only what's important.

Another useful tool, which was a major focus of this paper, is JGRASP's UML Class Diagram generator. We discussed earlier how an application goes about creating a class diagram, and JGRASP follows many of these methods. However, like I mentioned above, JGRASP is not as powerful as some of the tools out there. The embodied methods are not displayed, multiplicity isn't depicted, and composite associations aren't shown. What JGRASP creates is a class diagram with all the classes and their associations between each other. So any class which instantiates another has a link that shows up on the generated diagram. JGRASP also uses different colors to depict different classes; such as project classes, interface classes, JDK included classes, and driver classes such as JDBC. What is helpful about the UML diagram is it's ability to show users whether their project is being developed so as to match the diagrams created before implementation began. It is also an easy way to help others understand how a program internally interacts; especially fellow programmers and managers.

There are also a two tools that JGRASP has implemented which are most helpful in debugging code. The first is the java workbench. When a code is run, java workbench can be initiated to watch over the active variables and data structures of a class. Arrays can be opened up, hashes can be scanned, and variables values are all shown through this tool. However, it also offers the user the ability to change the values held by these variables/structures. This is very useful for quickly testing and watching the result of off by one errors, recreating a particular occurrence which cause and error then stepping through, or setting up testing conditions that may otherwise would rarely occur in the normal operation of the program.

The other tool is named the debugging viewer. This is an entirely visually driven tool which shows linked lists, hashes, arrays, heaps, etc. in their standard human visual models. The nice thing about these types of models is their ability to help the user understand quickly what the data they are working with 'looks' like internally, and it allows a programmer quick access to different positions in the data structure with a simple drag of the scrollbar. Again, like the java workbench, every node in the data structure can be modified dynamically so the values are immediately changed inside the running program. This tool again is great for quick debugging and testing. Used in unison with the java workbench, the simplicity this brings to the internal representation of data can greatly reduce the time it takes to get a program running how the programmer wishes.

One of the most interesting tools that JGRASP offers is called the complexity profile graph. On a given class, a bar graph representing the 'complexity' of each line is created. The range of values is really not as important, but the height of particular lines as compared to the average height of the lines of code in a file show where the most complicated code is written. Judged on methods calls, data structures, level of embedded code, and various other coding difficulties, the CSD graph can guide a programmer to the areas where the most problems will likely occur and where the most testing should be done.

The last feature that JGRASP has is a documentation generator which converts the current project into a Java API type document. This is a very quick and easy way to document a program's classes, show their internal methods and variables, the data types that must be passed into methods, and the interaction each class has with the others in the project. The documentation may not be formal enough to be considered a professional/publishable report on the classes, but for a quick reference and simple way to describe code to fellow programmers and managers, this works very well.

So we can see that each of the collection of these tools can have many advantages in terms of understanding code, explaining code to managers and other coders, debugging, and documentation. Other development environments such as visual studio, eclipse, websphere, studio creator, etc. all offer more tools and improved features on much of what JGRASP offers. However, JGRASP is a completely free environment and is much lighter in terms of load time and memory usage than these other environments. For students and smaller programs, this can be a very educational and helpful program to use in the development of software.

References

Using the Debugger as an Integral Part of Teaching CS1 - by James H. Cross II, T. Dean Hendrix, Larry A. Barowski. Proceedings of FIE 2002.

Extraction and Use of Class Dependency Information for Java - by L.A. Barowski, J.H. Cross II. Proceedings of WCRE 2002. (free to IEEE Computer Society members)

Software Visualization and Measurement in Software Engineering Education - by James H. Cross II, T. Dean Hendrix, Karl S. Mathias, and Larry A. Barowski. Proceedings of FIE 1999.

Visual Support for Incremental Abstraction and Refinement in Ada 95 - by T. Dean Hendrix, James H. Cross II, Larry A. Barowski, and Karl S. Mathias. Proceedings of SIGAda 1998.

Scalable Visualizations to Support Reverse Engineering: A Framework for Evaluation - by James H. Cross Ii, T. Dean Hendrix, Larry A. Barowski, Karl S. Mathias. Proceedings of WCRE 1998. (free to IEEE Computer Society members)

Visualization and Measurement of Source Code - by James H. Cross II, Kai H. Chang, T. Dean Hendrix, Richard O. Chapman, and Patricia A. McQuaid. Crosstalk, Dec 1997.

Tool Support for Reverse Engineering Multi-Lingual Software - by T. Dean Hendrix, James H. Cross Ii, Larry A. Barowski, Karl S. Mathias. Proceedings of WCRE 1997. (free to IEEE Computer Society members)

Control Structure Diagrams for Ada 95 - by Dr. James H. Cross II, Larry A. Barowski, Dr. T. Dean Hendrix, and Joseph C. Teate. Proceedings of TRI-Ada 1996.

GRASP/Ada 95: Visualization with Control Structure Diagrams - by Dr. James H. Cross II, Dr. Kai H. Chang, T. Dean Hendrix. Crosstalk, Jan 1996.

Reveal: A Tool to **Reverse Engineer** Class Diagrams- by S. Matzko, P. Clarke, T. Gibbs, B. Malloy, J. Power, R. Monahan. in Proceedings of TOOLS, Sydney, Australia, Feb 2002.