Overview of jGRASP

jGRASP is a lightweight integrated development environment (IDE), created specifically to provide visualizations for improving the comprehensibility of software. jGRASP is implemented in Java, and thus, runs on all platforms with a Java Virtual Machine. jGRASP supports Java, C, C++, Objective-C, Python (new in 2.0.0), Ada, and VHDL, and it comes configured to work with several popular compilers to provide "point and click" compile and run functions. jGRASP is the latest IDE from the **GRASP** (**G**raphical **R**epresentations of **A**lgorithms, **S**tructures, and **P**rocesses) research group at Auburn University.

jGRASP currently provides for the automatic generation of three important software visualizations: (1) *Control Structure Diagrams* (Java, C, C++, Objective-C, Python, Ada, and VHDL) for source code visualization, (2) *UML Class Diagrams* (Java) for architectural visualization, and (3) *Dynamic Viewers and Viewer Canvas* (new in 2.0.0) (Java) which provide runtime views for primitives and objects including traditional data structures such as linked lists and binary trees. jGRASP also provides an innovative *Workbench*, *Debugger*, and *Interactions* which are tightly integrated with these visualizations. Each is briefly described below.

The **Control Structure Diagram (CSD)** is an algorithmic level diagram which is generated for Ada, C, C++, Objective-C, Java, VHDL, and Python. The CSD is intended to improve the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit. The CSD, designed to fit into the space that is normally taken by indentation in source code, is an alternative to flow charts and other graphical representations of algorithms. The CSD is a natural extension to architectural diagrams such as UML class diagrams.

The CSD window in jGRASP provides complete support for CSD generation as well as editing, compiling, running, and debugging programs. After editing the source code, regenerating a CSD is fast, efficient, and non-disruptive. The source code can be folded based on CSD structure (e.g., methods, loops, if statements, etc.), then unfolded level-by-level. Standard features for program editors such as syntax based coloring, cut, copy, paste, and find-and-replace are also provided.

The **UML Class Diagram** is currently generated for Java source code from all Java class files and jar files in the current project. Dependencies among the classes are depicted with arrows (edges) in the diagram. By selecting a class, its members can be displayed, and by selecting an arrow between two classes, the actual dependencies can be displayed. The class diagram is a powerful tool for

understanding a major element of object-oriented software - the dependencies among classes.

The **Dynamic Viewers** for objects and primitives provide visualizations as the user steps through a program in debug mode or invokes methods for an object on the workbench. Textbook-like *Presentation* views are available for instances of classes that represent traditional data structures. When a viewer is opened, a *structure identifier* attempts to automatically recognize linked lists, binary trees, hash tables, and array wrappers (lists, stacks, queues, etc.) during debugging or workbench use. When a positive identification is made, an appropriate *presentation* view of the object is displayed. The *structure identifier* is intended to work for user classes, including textbook examples, as well as the most commonly used classes in the Java Collections Framework (e.g., ArrayList, LinkedList, HashMap, and TreeMap).

The **Viewer Canvas** allows users to create visualizations of their programs using multiple dynamic viewers. After a canvas for a program is saved, the program can be "run in the canvas" which launches the program in the debugger and opens the canvas file. The user can then "play" the program as well as use the debug controls to see the program visualization. The canvas can provide a conceptual visualization similar to what one might find in a textbook but with the added benefit of being dynamically updated as the user steps through the program. This allows for the exploration of the inner workings of a program regardless of its apparent complexity. The visualizations provided by the canvas are useful for general program understanding as well as traditional debugging.

The **Workbench**, in conjunction with the UML class diagram, CSD window, and Interactions, allows the user to create instances of classes and invoke their methods. Primitive and reference variables declared and assigned in Interactions are automatically placed on the Workbench. After an object is placed on the Workbench, the user can open a viewer to observe changes resulting from the methods that are invoked. The Workbench paradigm has proven to be extremely useful for teaching and learning object-oriented concepts, especially for beginning students.

The **Integrated Debugger** works in conjunction with the CSD window, UML window, Object Workbench, and Interactions. The Debugger provides a seamless way for users to examine their programs step by step. The execution threads, call stack, and local variables are easily viewable during each step. The Debugger provided the foundation for the dynamic viewers and viewer canvas.

The **Interactions** (new in jGRASP 1.8.7) feature allows users to enter most Java statements and expressions and then execute or evaluate them immediately. Interactions can be especially helpful when learning and experimenting with

new elements in the Java language. Results are shown in the workbench and in any open viewers or canvas windows associated with the variables and/or expressions.

Plug-ins for Checkstyle, FindBugs, Dead Code Detector (DCD), JUnit, and Web-CAT are included with jGRASP. If these tools are installed in a conventional location, jGRASP will find them automatically; otherwise, the user simply configures the tool with the appropriate path for its executable.

The ***jGRASP Tutorials*** provide best results when read while using jGRASP; however, they are sufficiently detailed to be read in a stand-alone fashion by a user who has experience with one or more other IDEs. The tutorials are quite suitable as supplemental assignments during a course. When working with jGRASP and the tutorials, students can use their own source code, or they can use the examples shown in the tutorials (..\jGRASP\examples\Tutorials\). Users should copy the examples folder to their personal directories prior to modifying them.

For additional information and to download jGRASP, please visit our web site (http://www.jgrasp.org).