The jGRASP Handbook

The complete guide to the jGRASP™ integrated software development environment

Copyright © 2002 James H. Cross II and Larry A. Barowski
All Rights Reserved

June 2, 2002
DRAFT
Table of Contents

Preface...................................................................................vii
Overview of jGRASP .................................................... 1

Part 1 - Tutorials .................................................................10

1 Installing jGRASP.............................................................11
2 Getting Started .............................................................. 12
   2.1 Creating a New File .................................................. 13
   2.2 Saving a File ........................................................... 17
   2.3 Closing a File .......................................................... 18
   2.4 Loading a File .......................................................... 19
   2.5 Generating a Control Structure Diagram ................. 22
   2.6 Folding a CSD ....................................................... 24
   2.7 Line Numbers .......................................................... 25
   2.8 Compiling a Program ............................................... 26
   2.9 Running a Program .................................................. 29
   2.10 Exiting jGRASP ..................................................... 30

3 The Control Structure Diagram (CSD) ............................31
   3.1 An Example to Illustrate the CSD ......................... 31
   3.2 CSD Program Components/Units ........................... 32
   3.3 CSD Control Constructs ........................................ 34
   3.4 CSD Templates ....................................................... 37
   3.5 Hints on Working with the CSD ............................... 38
   3.6 Reading Source Code with the CSD .................... 39

4 Projects ............................................................................. 42
   4.1 Creating a Project .................................................. 42
   4.2 Adding files to the Project ....................................... 44
   4.3 Removing files from the Project ............................... 45
   4.4 Generating Documentation for the Project (Java only) 46
22.2 Print Menu ...................................................................................................................... 103
22.3 Layout Menu ...................................................................................................................... 103
22.4 Edit Menu ............................................................................................................................ 103
22.5 View Menu .......................................................................................................................... 104
22.6 Diagram Display .................................................................................................................. 104
22.7 Info Window ......................................................................................................................... 104
22.8 Goto Window ....................................................................................................................... 104
22.9 UML Settings Dialog .......................................................................................................... 105

23 Settings .................................................................................................................................. 105
23.1 General Info for Settings ...................................................................................................... 105
23.2 CSD Window Settings Dialog .............................................................................................. 105
  23.2.1 Languages Page ............................................................................................................... 106
23.3 Compiler Environment Dialog ............................................................................................ 110
  23.3.1 Commands Page ............................................................................................................. 110
  23.3.2 Parse Page (C and C++ only) ......................................................................................... 112
  23.3.3 Macros Page (C and C++ only) ...................................................................................... 112
23.4 Print Settings Dialog .......................................................................................................... 113
  23.4.1 Common Page ................................................................................................................ 113
  23.4.2 CSD Print Settings Page .................................................................................................. 113
  23.4.3 UML Print Settings Page ................................................................................................ 114
  23.4.4 Command Page (Unix / Linux only) .............................................................................. 114

24 Fixed Bugs .............................................................................................................................. 114
24.1 Bugs in Version 1.5.0 Alpha 3 Fixed in Version 1.5.0 Beta .................................................. 114
24.2 New Features in Version 1.5.0 Beta ...................................................................................... 114
24.3 Bugs in Version 1.4.4 Beta Fixed in Version 1.5.0 Alpha 3 ................................................ 115
24.4 New Features in Version 1.5.0 Alpha 3 .............................................................................. 115
24.5 Bugs in Version 1.4.3 Beta Fixed in Version 1.4.4 Beta ...................................................... 115
24.6 Bugs in Version 1.4.2 Beta Fixed in Version 1.4.3 Beta ...................................................... 115
24.7 Bugs in Version 1.4.1 Beta Fixed in Version 1.4.2 Beta ...................................................... 115
24.8 New Features in Version 1.4.2 Beta ...................................................................................... 116
24.9 Bugs in Version 1.4 Beta Fixed in Version 1.4.1 Beta ......................................................... 116
24.10 New Features in Version 1.4 Beta ....................................................................................... 117

25 How To .................................................................................................................................. 117
25.1 Change the language for a file: .......................................................................................... 117
25.2 Drag files into jGRASP: ..................................................................................................... 117
25.3 Open a file in binary mode: ............................................................................................... 117
25.4 Search for the selected text: ......................................................................................................... 117
25.5 Make ".h" a default file extension for C or C++ files only: ............................................................ 117
25.6 Change the CSD include path for C or C++ files: ........................................................................ 117
25.7 Get C and C++ files with lots of ugly macros or compiler extensions to parse: ...................... 118

Index ............................................................................................................................................. 119
Preface

GRASP (Graphical Representations for Algorithms, Structures, and Processes) is a full-featured development environment, created specifically to provide automatic generation of software visualizations for the purpose of improving the comprehensibility of the software. Over the past three years, a new version of GRASP has been developed based on pcGRASP and UNIX GRASP, which were written in C/C++. This new version, called jGRASP, is implemented in Java, and thus, runs on all platforms with a Java Virtual Machine. As with the previous versions, jGRASP supports Java, C, C++, Ada, and VHDL, and it can be configured to work with almost any compiler.

jGRASP currently provides for the automatic generation of three software visualizations: the Control Structure Diagram (Java, C, C++, Ada, and VHDL), Complexity Profile Graph (Java and Ada), and the UML Class Dependency Diagram (Java). Each is briefly described below.

The Control Structure Diagram (CSD) is an algorithmic level diagram automatically generated for Ada, C, C++, Java and VHDL. 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, which is 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 major goal was to create an intuitive and compact graphical notation that was easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and object diagrams. Section 3 provides a detailed description of the CSD.

jGRASP provides complete support for the CSD with some unique and powerful editing features. The CSD may be edited directly and regenerated when desired. Regenerating a CSD is fast and efficient and non-disruptive (approximately 5000 lines/sec). The source code can be folded based on CSD structure. A structure (method, loop, if statement, etc.) can be folded recursively with a double click of the mouse, then unfolded level-by-level. Standard features for program editors such as syntax based coloring, selection, and find-and-replace are also provided.

The Complexity Profile Graph (CPG) is a statement-level complexity or comprehensibility metric represented as a bar graph. jGRASP automatically generates the CPG by dividing the source code into segments roughly corresponding to individual program statements. Each segment is assigned a complexity that is a weighted sum of individual metrics for reachability, content, breadth, and inherent complexity. The CPG is currently generated for Ada and Java source code.

The UML Class Dependency 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. Selecting a class displays its members in the Class Info pane, and selecting an arrow between two classes displays a list of the actual dependencies. This diagram is a powerful tool for understanding a major element of object-oriented software - the dependencies among classes. See Sections 4 and 5 for details on projects and UML generation.

The jGRASP Handbook begins with a brief Overview, which is then followed by two parts, each written with a specific purpose in mind then follow the Overview.

Overview of jGRASP introduces the jGRASP Desktop and the software visualizations it can produce.

Part 1 Tutorials consists of a set of tutorials ranging from “Installing jGRASP” and “Getting Started” to more advanced topics including the “Integrated Debugger” and the “UML Class Dependencies Diagram.”

Part 2 Reference contains the much more detailed information than Part 1. The reader will find many of the topics from Part 1 repeated in Part 2 as the topic is treated in depth.

jGRASP may be freely downloaded from the following URL. See Section 1 for details on installation.

http://www.eng.auburn.edu/grasp/
Overview of jGRASP

Over the past ten years, the GRASP (Graphical Representations for Algorithms, Structures, and Process) Research Project at Auburn University has produced a series of full-featured integrated development environments (IDEs) for dozens of hardware platforms. The primary purpose of these IDEs has been to generate visualizations for the purpose of improving the comprehensibility of the software. During the past three years, a new version of GRASP has been developed, based on our earlier versions, pcGRASP and UNIX GRASP, which were written in C/C++. This new version, called jGRASP, is implemented in Java, and thus runs on all platforms with a Java Virtual Machine (e.g., Windows 95/98/NT/XP, UNIX, Linux, Mac OS X). As with the previous versions, jGRASP supports development in Java, C, C++, and Ada, and it can be configured to work with almost any compiler.

Ease of Use. jGRASP cuts through much of the learning curve associated with large complex IDEs by providing a single intuitive graphical interface to both commercial (e.g., Borland and Microsoft) and free compilers (e.g., Java 2 SDK, GNAT, Cygnus). jGRASP provides students with a single integrated tool for their Java, C, C++, and Ada programs. The jGRASP desktop, shown in Figure 1, is composed of a

Figure 1. The jGRASP Desktop
Control Panel with a menu across the top plus three panes: (1) left pane with tabs for **Browse**, **Project**, **Find**, and **Debug**, (2) right pane for CSD Windows (source code), and (3) lower pane with tabs for jGRASP messages, Compile messages, and input/output for Run.

The major thrust of the research behind the development of jGRASP was to provide for the automatic generation of graphical representations to improve the comprehensibility of software. Figure 1 shows the jGRASP Desktop prior to the generation of any visualizations. The visualizations currently supported by jGRASP include: the **control structure diagram** (CSD) for Java, C, C++, Ada; the **UML class diagram** for Java; the **graphical debugger** for Java; and the **complexity profile graph** (CPG) for Java and Ada. These are described in more detail below, followed by a short description of supported compilers.

**Control Structure Diagram.** jGRASP provides complete support for CSDs, which are designed to depict program units and control structures/paths in a compact, intuitive, non-disruptive way that provides the “look-and-feel” advantages of code plus visual advantages of graphical representation. The positive results of our empirical experiments in software comprehension (CSD vs. plain text) have been reported in the literature [Hendrix et al. 2002, Cross et al. 1998]. The CSD is automatically generated for Ada, C, C++, and Java. Figure 2 shows the jGRASP desktop with a CSD Window containing a Java class with a simple main program for which the CSD has been generated. The CSD, which is designed to fit into the

![Figure 2. CSD with source code](image-url)
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 data flow diagrams, call graphs, module diagrams, and class/object diagrams. Of course, using the CSD is optional, and after generating the CSD, it can be removed with a single click, leaving the source code formatted in the traditional way.

**Editing Features.** The CSD Window, which serves as the source code editor, has some unique and powerful editing features. As the source code is edited, the CSD may be regenerated as desired. Regenerating a CSD is fast and non-disruptive (approximately 5,000 lines/sec). The source code can be folded based on CSD structure. A structure (method, loop, if statement, etc.) can be folded recursively with a double click of the mouse, then unfolded level-by-level as shown in Figure 3. Folding can be a very useful mechanism when explaining a program, whether to one’s self, to a group of students, or to colleagues during a technical code review. Standard features for program editors such as syntax based coloring and find-and-replace are provided, as well as menus and icons for compile, run, and debug.

![Figure 3. Source code folding](image)
UML Class Diagram. The UML class dependency diagram is currently generated for Java source code from all Java class files (and jar files) affiliated with the current project. Figure 4 illustrates the various types of classes and dependencies that can be selectively shown in the diagram. This diagram has tremendous potential for teaching Java. Ideally, the UML class diagram would be included for each non-trivial example in the text. While this is impractical from the perspective of the number of pages it would require, jGRASP provides a very practical solution. In the classroom, the instructor can generate these diagrams on the fly to illustrate the Java class hierarchy and dependencies of an example. Students can generate them in the lab or at home to help them understand their programs. The user has complete control over the layout of the icons and which types of classes and dependencies to include. Our experience has been that students actually enjoy selecting and moving the icons around in the diagram, making it aesthetically pleasing to them. The layout is always saved in the current project file, which can then be sent electronically with the source files. As a bonus, the UML diagram can be used to navigate among the Java source files; clicking on a class icon opens the file in the CSD Window for reading or editing.

Figure 4. UML class diagram
**Program Documentation.** For Java programs, users can automatically generate documentation in the style of an application programmer interface (API). jGRASP calls the javadoc utility, included with the J2SDK, to create a complete hyperlinked document within a few seconds. This can be done on a file-by-file basis or for a complete project as shown in Figure 5 below. Note, in this example, even though no JavaDoc comments were included in the source file, the generated documentation is still quite useful since it provides hyperlinks to all classes in the project, plus links to the fields, constructors, and methods of each class. However, for even more useful results, JavaDoc formal comments should be included in the source code. When the documentation is generated for an individual file, it is stored in a temporary directory for the duration of the jGRASP session. For projects, the documentation files are stored in a directory that becomes part of the project, and therefore, persists from one jGRASP session to the next. If any changes are made to a project source file (and the file is resaved), jGRASP will indicate that the documentation needs to be regenerated; however, the user may choose to view the documentation files without updating them.

![jGRASP Documentation Viewer](image)

**Figure 5.** Project documentation for Java
**Graphical Debugger.** No development environment would be complete without an integrated debugger. jGRASP provides a highly visual debugger for Java, which includes all of the traditional features expected in a debugger. As shown in Figure 6, the left side of the jGRASP desktop has debugger panels for Threads, Call Stack, and Variables/Settings. The user simply toggles a breakpoint on a line of source code by right-clicking the mouse, and then runs the program in debug mode. After the program stops at the breakpoint, the user can single step, step into a method call, etc., while watching the call stack and contents of variables change dynamically. The integrated debugger is especially useful for watching the creation of objects as the user steps through various levels of constructors. Instructors using jGRASP in the classroom have reported using the debugger extensively to explain programs. A major part of understanding a program as it is explained is keeping track (mentally or otherwise) of the state of the program as one reads from line to line. This is exactly what the jGRASP debugger provides.

![Figure 6. Integrated graphical debugger](image)

```java
public class ArrayTest3
{
    public static void main(String[] args)
    {
        double[][] table = new double[2][3];
        double sum = 0;
        for (int i = 0; i<2; i++)
        {
            for (int j=0; j<3; j++)
            {
                table[i][j] = Math.pow(i+j, i+j);
            }
        }
        if (table[1][1] % 2 == 1)
        {
            sum += table[1][1];
        }
        else
        {
            sum -= table[1][1];
        }
        System.out.println("Sum of Table = "+sum);
    }
}
```
Complexity Profile Graph (CPG). The CPG is a statement-level complexity metric intended to provide a measure of source code comprehensibility [Cross et al. 1997, McQuaid et al. 1997]. Figure 7 shows the CPG for the Java class ArrayTest3 from Figure 1 above. The source code is divided into chunks or segments roughly corresponding to program statements. Each segment is assigned a complexity that is a weighted sum of individual metrics for reachability, content, breadth, and inherent complexity, and then represented as a bar in the graph. The CPG is currently generated for Ada and Java source code. The premise is that the peaks in the CPG indicate the area that may be most difficult to comprehend and, as a result, may be the most likely place to find errors in the code. These peak sections should be reviewed/tested more carefully, or the code should be rewritten to reduce the complexity.

Figure 7. Complexity Profile Graph
Compilers. jGRASP includes compiler settings for Ada, C/C++, Java, Fortran, and assembler. Users can add settings for other compilers as needed. The following languages/compiler settings are included (the default compiler settings are underlined). Links for some of those that can be freely downloaded are included for your reference.

1. Ada (GNAT)  
   ftp://cs.nyu.edu/pub/gnat/3.13p/winnt/ (e.g., gnat-3.13p-nt.exe)
2. C, C++ (GNU/Cygnum, Borland, Microsoft)  
   http://sources.redhat.com/cygwin/  
   http://www.borland.com/bcppbuilder/freecompl/cppc55steps.html
3. FORTRAN (GNU/Cygnum)  
   Included with Cygwin, see (2) above. Note that FORTRAN programs are currently treated as Plain Text so there is no CSD generation.
4. Java (J2SDK, Jikes)  
   http://java.sun.com/j2se/1.4/download.html (download the SDK rather than the JRE).
5. Assembler (MASM – Microsoft Assembler)  
   Note that assembly programs are treated as Plain Text so there is no CSD generation.

Organization of The jGRASP Handbook

The jGRASP Handbook is intended to introduce readers to the jGRASP IDE. Specifically, instructors, students, lab administrators, and professionals will find the handbook useful for learning to use jGRASP, and then later as a reference book.

In addition to the Overview, The jGRASP Handbook is divided into two parts, each written with a specific purpose in mind. Part 1 - Tutorials consists of a set of chapters ranging from “Installing jGRASP” and “Getting Started” to more advanced topics including the “Integrated Debugger” and the “UML Class Dependencies Diagram.” These short chapters are appropriate for students to read during the first few weeks of a course. Part 2 - Reference contains much more detailed information on the topics in Part 1, plus many additional topics appropriate for advances users. Hence, the reader will find many of the topics from Part 1 repeated in Part 2 as the topic is treated in depth.

Part 1 - Tutorials
1. Installing jGRASP
2. Getting Started
3. The Control Structure Diagram (CSD)
4. Projects
5. UML Class Dependency Diagrams
6. The Integrated Debugger
7. Compiler Environment Settings

Part 2 - Reference
8. Installing jGRASP
9. Running jGRASP
10. Using Help
Download and Additional Information

We invite you to take a serious look at jGRASP and the remainder of this handbook. Since both the software and the handbook are updated regularly, you are encouraged to join the GRASP Mailing List, which is used primarily to notify users of major releases. When you download jGRASP, you will have an opportunity to join.

For the most recent releases and other information, please visit our web site.

http://www.eng.auburn.edu/grasp/

References


Part 1 - Tutorials
1 Installing jGRASP

Currently, jGRASP is available from http://www.eng.auburn.edu/grasp/ in four versions: two are self-extracting for Microsoft Windows, one is for Mac OS X, and the fourth is a generic ZIP file. Although the generic ZIP file can be used to install jGRASP on any system, it is primarily intended for Linux and UNIX systems. If you are on a Windows machine, either (1) or (2) below is strongly recommended.

(1) jGRASP JRE exe (14.7 MB) – Windows self-extracting exe file for with JRE. Since this includes a copy of the JRE, no Java installation is required to run jGRASP itself; however, if you will be compiling and running Java programs, you must also install the full J2SDK (also called JDK). This version is convenient if you will be compiling programs in languages other than Java.

(2) jGRASP exe (2.3 MB) – Windows self-extracting exe file. The full J2SDK must be installed in order to run jGRASP and to compile and run Java programs.

(3) jGRASP pkg.tar.gz (1.6 MB) – Mac OS X tarred and gzipped package file (requires admin access to install).

(4) jGRASP (2.1 MB) – Zip file. After unzipping the file, refer to README file for installation instructions. The full J2SDK must be installed in order to run jGRASP and to compile and run Java programs.

For Windows 95/98/2000/XP - After downloading (1) or (2) above, simply double click on the .exe file, and the script will take you through the steps for installing jGRASP. If you are uncertain about a step, you should accept the default by pressing ENTER. When you have completed the installation, you should find the jGRASP icon on your desktop. jGRASP should also be listed on the Window's Start – Programs menu.

Compilers - Although jGRASP includes settings for a number of popular compilers, it does not include any compilers. Therefore, if the compiler you need is not already installed on your machine, it must be installed separately. Since these are generally rather large files, the download time may be quite long. If a compiler is available to you on a CD (e.g., with a textbook), you may save yourself some time and energy by installing it from the CD rather than attempting to download it.

jGRASP includes settings for the following languages/compilers. The default compiler settings are underlined. Note that links for those that can be freely downloaded are included for your convenience.

(1) Ada (GNAT)
ftp://cs.nyu.edu/pub/gnat/3.14p/winnt/ (e.g., gnat-3.14p-nt.exe)

(2) C, C++ (GNU/Cygwin, Borland, Microsoft)
http://sources.redhat.com/cygwin/
http://www.borland.com/bcppbuilder/freecompiler/cppc55steps.html

(3) FORTRAN (GNU/Cygnum)
Included with Cygwin, see (2) above. Note that FORTRAN is currently treated as Plain Text so there is no CSD generation.

(4) Java (J2SDK, Jikes)
http://java.sun.com/j2se/1.4/download.html

(5) Assembler (MASM)
Note that assembler is treated as Plain Text so there is no CSD generation.

After you have installed the compiler(s) of your choice, you will be ready to begin working with jGRASP. If you are not using the default compiler for a particular language (e.g., J2SDK for Java), you need to change the Compiler Settings as described below in Compiler Environment Settings.

You can start jGRASP by double clicking on the icon.
2 Getting Started

Java will be used in the examples in this section; however, the information applies to all supported languages for which you have installed a compiler (e.g., Ada, C, C++, Java) unless noted otherwise. In any of the language specific steps below, simply select the appropriate language and code. For example, in the “Creating a New File” below you may select C++ as the language and then enter a C++ example or select Java to enter a Java example.

If you have installed jGRASP on your own PC, you should see the jGRASP icon in the Windows desktop.

You can start jGRASP by double clicking on the icon. If you are working on a PC in a computer lab, you may not see the jGRASP icon on the desktop. Try to find one of the following:

1) Click Start -- Programs -- jGRASP
2) Click Start -- Programs -- Programming Tools -- JGrasp IDE

Depending on the speed of your computer, jGRASP may take about 30 seconds to come up. The jGRASP virtual Desktop, shown below, is composed of a Control Panel with a menu across the top plus three panes: (1) left pane with tabs for Browse, Project, Find, and Debug, (2) right pane for CSD Windows, and (3) lower pane with tabs for jGRASP messages, Compile messages, and input/output for Run.

![jGRASP Virtual Desktop](image)

Figure 8. The jGRASP Virtual Desktop
2.1 Creating a New File

To open an empty CSD Window for Java within the Desktop, click on File -- New File -- Java. Note that the list of languages displayed by File – New File will vary with your use of jGRASP. If the language you want is not listed, click Other to see all available languages. The languages for the last 25 files opened will be displayed in the list; the remaining available languages will be under Other.

Figure 9. Opening a CSD Window for Java
After you click on **File -- New File -- Java** (Figure 9 above), a CSD Window is opened in the right pane of the Desktop as shown in Figure 10 below. Notice the title for the frame, jGRASP CSD (Java), indicates the CSD Window is Java specific.

If Java is not the language you intend to use, you should close the window, then open a CSD Window for the correct language.

![Figure 10. Empty CSD Window in Desktop](image-url)
In the upper right corner of the CSD Window are three buttons that control its display:

The first button iconifies the CSD Window. The second either maximizes the CSD Window relative to the jGRASP Desktop, or if it is already maximized, the button restores the CSD Window to its previous size. The third button closes the CSD Window.

You may also make the Desktop full screen by clicking the appropriate icon in the upper corner of it. Notice the CSD Window has its own menu and toolbar with icons across the top.

Figure 11 shows the CSD Window maximized within the virtual Desktop.

**HINT:** If you want all of your CSD Windows to be maximized automatically when you open them, then under Settings on the Desktop menu, click on (indicated by a check mark) the option called **Open CSD Windows Maximized**.

![Figure 11. CSD Window expanded in Desktop](image)
Type in the following Java program in the CSD Window, exactly as it appears. Remember, Java is case sensitive.

```java
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!\n");
    }
}
```

After you have entered the program, your CSD Window should look similar to the program shown in Figure 12. Notice in the source code that coloring is used to distinguish among comments, keywords in the language, strings, etc. Later, you will learn how set these colors to those of your on choice.

Figure 12. CSD Window with program entered
2.2 Saving a File

Save the program as "Hello.java" by clicking the Save icon on the tool bar of the CSD Window, or you can click **File -- Save** on the CSD Window menu (not the Desk Top menu). Note, in Java, the file name must match the class name (i.e., class Hello must be saved as Hello.java).

![Figure 6. Saving a file from the CSD Window](image)

After you click on Save, the Save dialog box come up as illustrated in Figure 14.

After the program has been saved, it will be listed in the browse pane. If the program is not listed in browse pane, be sure the browse pane is set to the directory where the file was saved.

![Figure 14. Save dialog to name file](image)
2.3 Closing a File

Now that the file has been saved, you can close the file by clicking the Close button in the upper right corner of the CSD Window.

After the file is closed, your Desktop should look like the figure below. Notice that Hello.java is listed in the Browse pane on the left.

Figure 15. Desktop with CSD Windows closed
2.4 Loading a File

A file can be loaded into a CSD Window in five distinct ways. Each of these is described below.

1) If the file is listed in jGRASP Browse pane (as in Figure 15), you can simply double click on the file name, and the file will be opened in a new CSD Window.

2) On the Desktop menu, click File – Open as illustrated in Figure 16. This will bring up the Open File dialog.

![Figure 16. Opening a file from the Desktop](image-url)
3) If you have a CSD Window open, click **File – Open** as shown in Figure 17. This will open the Open File dialog box, which will allow you browse up and down directories until you locate the file you want to open.

Figure 17. Opening a file from the CSD Window
4) If you have a Windows file browser open (e.g., Windows Explorer or My Computer), and the file is marked as a jGRASP file, you can just double click the file name.

5) If you have a Windows file browser open (e.g., Windows Explorer or My Computer), you can drag-and-drop a file to the jGRASP Desktop canvas where the CSD Window will be displayed. However, files usually open more quickly by double-clicking (option 4 above) rather than using the drag-and-drop option.

In all cases above, if a file is already open in jGRASP, its CSD Window will be popped to the top of the Desktop.

**Multiple CSD Windows**

You can have multiple CSD Windows open, each with a separate file. Each program can be compiled and run from its respective CSD Window. In Figure 18, two CSD Windows have been opened. One contains Hello.java and the other contains Hello2.java. If the window you want to work in is visible, simply click the mouse on it to bring it to the top. Otherwise, click **Window** on the upper tool bar, and a drop down menu will list all of the open files (also shown in Figure 18). Clicking on the file name (e.g., Hello.java) will bring its CSD Window to the top.

![Figure 18. Multiple files open](image-url)
2.5 Generating a Control Structure Diagram

Anytime you have a syntactically correct program or skeleton of a program in the CSD Window, you can generate a Control Structure Diagram (CSD). Generate the Control Structure Diagram (CSD) for the program by doing one of the following:

1) Clicking the Generate CSD icon or
2) Clicking View -- Generate CSD on the menu or
3) Pressing F2

If your program is syntactically correct, the CSD will be generated as shown in Figure 19. After you are able to successfully generate a CSD, go on to the next section below.

Figure 19. After CSD is generated
Otherwise, if you have a syntax error in your program that was detected during the Generate CSD, jGRASP will highlight the vicinity of the error and describe it in the message window.

If you do not find an error in the highlighted line, be sure to look for the error in the line just above it. In the example in Figure 20, the semi-colon was omitted at the end of the println statement. As you gain experience, these errors will become easier to spot.

If you are unable find and correct the error, you should try compiling the program, since the compiler usually provides a more detailed error message (see Compiling a Program below).

You can remove the CSD by doing one of the following:

1) Clicking the Remove CSD icon or
2) Clicking View -- Remove CSD on the menu or
3) Pressing Shift-F2

![Figure 20. Syntax error detected](image)

Remember, the purpose of using the CSD is to improve the readability of your program. While this is may not be obvious on a small simple program like the example, it should become apparent as the size and complexity of your programs increase.
TIP: As you enter a program, try to enter it in syntactically correct “chucks.” For example, the following is sufficient to generate the CSD.

```java
public class Hello
{
}
```

As soon as you think you have entered a syntactically correct chuck, you should generate the CSD. Not only does this update the diagram, it catches your syntax errors early.

### 2.6 Folding a CSD

“Folding” is another feature that many users find useful, especially as programs get larger. After you have generated the CSD, you can fold your program based on its structure.

For example, if you double-click on the class icon, the entire program is folded (Figure 21). If you double-click on the “plus” icon, the first layer of the program is unfolded. You can continue to unfold the program layer by layer as needed.

Although the example program has no loops or conditional statements, these may be folded by double-clicking the corresponding CSD control constructs. For other folding options, see the View – Fold menu.

![Figure 21. Folded CSD](image-url)
2.7 Line Numbers

Line numbers can be very useful when referring to specific lines or regions of a program. Although not part of the actual program, they are displayed to the left of the source code as indicated in Figure 22.

Line numbers can be generated by clicking the line number icon on the CSD Window toolbar, and removed by clicking the icon again. Line numbers can also be generated/removed from the View menu.

With Line numbers turned on, new line numbers are inserted and/or added to the end each time you press “ENTER” on the keyboard. If you insert a line in the code, all line numbers below the new line are incremented.

You may “freeze” the line numbers to avoid the incrementing by clicking on the Freeze Line Numbers icon. To unfreeze the line number, click the icon again. This feature is also available on the View menu.

Figure 22. Line numbers in the CSD Window
2.8 Compiling a Program

After you have a program in the CSD Window, either by loading a file or typing it in and saving it, you are ready to compile the program. If you are compiling a language other than Java, you will need to “compile and link” the program.

Compile the program in jGRASP by clicking on Compiler -- Compile (Figure 23) or you click the Compile icon (green plus for Java).

Compile and Link the program if you are compiling a language other than Java, by clicking on Compiler – Compile and Link or you click the Compile and Link icon (green plus plus). Note, these options will not be visible on the tool bar and menu in a CSD Window for a Java program.

Figure 23. Compiling a program
The results of the compilation will appear in the **Compile Messages tab** in the lower window of the Desktop. If your program compiled successfully, you should see the message “operation complete” with no errors reported, as illustrated in Figure 24, and you are now ready to "Run" the program (see next section).

![Figure 24. A successful compilation](image)

**Error Messages**

If you receive an error message indicating “file not found,” this generally means jGRASP could not find the compiler. For example, if you are attempting to compile a Java program and the message may indicate that “javac” was not found. Go back to Section 1, Installing jGRASP, and be sure you have followed all the instructions. Once the compiler is properly installed and set up, any errors reported should be about your program.

If your program does not compile, the errors reported by the compiler will be displayed in the Compile Messages window (Figure 25). The description of first error detected will be highlighted, and jGRASP automatically scrolls the CSD Window to the line where the error most likely occurred and highlights it.
Even if multiple errors are indicated, as soon you correct the first error reported, you should attempt to compile the program again. Sometimes a single error causes a cascade of reported errors. Only after you have “fixed” all these reported errors will your program actually compile. Only then you are ready to “Run” the program as described in the next section.

Figure 25. Compile time error reported
2.9 Running a Program

At this point you should have successfully compiled your program. Two things indicate this. First, there should be no errors reported in the Compile Messages window. Second, you should have a Hello.class file listed in the Browse pane, assuming the pane is set to list "All Files."

To run the program, click **Run – Run** on the CSD Window tool bar (Figure 26). The other options on the Run menu will be explained later.

You can also run the program by clicking the Run icon on the tool bar.

![Figure 26. Running a program](image-url)
Output

When you run your program, the Run I/O tab in the lower pane pops to the top of the Desktop. The results of running the program are displayed in this pane as illustrated in Figure 27.

![Figure 27. Output from running the program](image)

2.10 Exiting jGRASP

When you have completed your session with jGRASP, you should "exit" jGRASP rather than leaving it open for Windows to close when you log out or shut down your computer. When you exit jGRASP normally, it saves its current state and closes all the files you were working on. If a file was edited during the session, it prompts you to save or discard the changes. The next time you start jGRASP, it will open your files, and you will be ready to begin where you left off.
3 The Control Structure Diagram (CSD)

The Control Structure Diagram (CSD) is an algorithmic level diagram 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 is an alternative to flow charts and other graphical representations of algorithms. The major goal behind its creation was that it be an intuitive and compact graphical notation that was easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and class diagrams.

3.1 An Example to Illustrate the CSD

Figure 28 shows the source code for a Java method called binarySearch. The method implements a binary search algorithm by using a while loop with an if..else..if statement nested within the loop. Even though this is a simple method, displayed with colored keywords and traditional indentation, its readability can be improved by adding the CSD. In addition to the while and if statements, we see the method includes the declaration of primitive data (int) and two points of exit. The CSD provides visual cues for each of these constructs.
Figure 29 shows the binarySearch method after the CSD has been generated. Although all necessary control information is in the source text, the CSD provides additional visual stimuli by highlighting the sequence, selection, and iteration in the code. The CSD notation begins with symbol for the method itself symbol appended to the statement stem. The CSD constructs for the while statement is represented by the double line “loop” (with break at the top), and the if statement uses the familiar diamond icon from traditional flowcharts. Finally, the two ways to exit from this method are shown explicitly with an arrow drawn from inside the method through the method stem to the outside.

```
public static int binarySearch(int key, int[] intArray)
{
    int low, middle, high;
    low = 0;
    high = intArray.length-1;
    while (low < high )
    {
        middle = (low + high)/2;
        if (key < intArray[middle])
            high = middle - 1;
        else if (key > intArray[middle])
            low = middle + 1;
        else
            return middle;
    }
    return 0;
}
```

Figure 29. binarySearch with CSD

While this is a small piece of code, it does illustrate the basic CSD constructs. However, the true utility of the CSD can be realized best when reading or writing larger, more complex programs, especially when control constructs become deeply nested. A number of studies involving the CSD have been done and others are in progress. In one of these, CSD was shown to be preferred significantly over four other notations: flowchart, Nasi-Schneiderman chart, Warnier-Orr diagram, and the action diagram [Cross 19xx]. In a several later studies, empirical experiments were done in which source code with the CSD was compared to source code without the CSD. In each of these studies, the CSD was shown provide significant advantages in numerous code reading activities [Hendrix 20xx].

In the few sections, the CSD notation is described in more detail.

### 3.2 CSD Program Components/Units

The CSD includes graphical constructs for the following components or program units: class, abstract class, method, and abstract method. The construct for each component includes a unit symbol, a box...
notation, and a combination of the symbol and box notation. The symbol notation provides a visual cue as to the specific type of program component. It has the most compact vertical spacing in that it retains the line spacing of source code without the CSD. The box notation provides a useful amount of vertical separation similar to skipping lines between components. The symbol and box notation is simply a combination of the first two. Most of the examples in this handbook use the symbol notation because of its compactness. CSD notation for program components/Units is illustrated in the table below.

<table>
<thead>
<tr>
<th>Component</th>
<th>Symbol Notation</th>
<th>Box Notation</th>
<th>Symbol and Box Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>class or Ada package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstract class</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method or function or procedure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstract method</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.3 CSD Control Constructs

The basic CSD control constructs for Java are grouped in the following categories: sequence, selection, iteration, and exception handling, as described in the table below. Note, the semi-colons in the examples are placeholders for statements the language.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Sequential flow is represented in the CSD by a vertical stem with a small horizontal stem for each individual statement on a particular level of control.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>For section statements, the True/False condition itself is marked with a small diamond, just as in a flow chart. The statements to be executed if the condition is true are marked by a solid line leading from the right of the decision diamond.</td>
</tr>
<tr>
<td>if</td>
<td>The control path for a false condition is marked with a dotted line leading from the bottom of the diamond to another decision diamond, an else clause, a default clause, or the end of the decision statement.</td>
</tr>
<tr>
<td>else</td>
<td>By placing the second if on the same line with the first else, the unnecessary indentation of nested if statements is avoided. However, if the deep nesting effect is desired, the second if can be placed on the line after the else.</td>
</tr>
<tr>
<td>if..else..if</td>
<td>The semantics of the switch statement are different from those of if statements. The expr (of integral type: int, char) is evaluated, and then control is transferred to the case label matching the result or to the default label if there is no match. If a break statement is placed at the end of the sequence within a case, control passes “out” (as indicated by the arrow) and to the end of the switch statement after the sequence is executed. Notice the similarity of the CSD notation for the switch and if statements when the break is used in this conventional way. The reason for this is that, although different semantically, we humans tend to process them the same way (e.g., if expr is not equal to case 1, then take the false path to case 2 and see if they are equal, and so on). However, the break statement can be omitted as illustrated next.</td>
</tr>
<tr>
<td>switch</td>
<td></td>
</tr>
</tbody>
</table>

```java
switch(expr)
{
    case 1:
        break;
    case 2:
        break;
    default:
        
}
When the break statement is omitted from end of
the sequence within a case, control falls through
to the next case. In the example at left, case 1 has a break statement at the end of its
sequence, which will pass control to the end of
the switch (as indicated by the arrow).

However, case 2, case 3, and case 4 do not use
the break statement. The CSD notation clearly
indicates that once the flow of control reaches
case 2, it will also execute the sequences in
case 3 and case 4. The diamonds in front of
case 3 and case 4 have arrows pointing to each
case to remind the user that these are entry
points for the switch. When the break statement
precedes the next case (as in case 1), the
arrows are unnecessary.

The CSD notation for the while statement is a
loop construct represented by the double line,
which is continuous except for the small gap on
the line with the while. The gap indicates the
control flow can exit the loop at that point or
continue, depending on the value of Boolean
condition. The sequence within the while will be
executed zero or more times.

The for statement is represented in a similar
way. The for statement is designed to iterate a
discrete number of times based on an index, test
expression, and index increment. In the
example at left, the for index is initialized to 0,
the condition is I < j, and the index increment is
i++. The sequence within the if will be executed zero or more times.

The do statement is similar to the while except
that the loop condition is at the end of the loop
instead of the beginning. As such, the body of
the loop is guaranteed to execute at least once.
Iteration (cont’d)

**break**

```
while (cond)
{
    ;
    if (cond)
    {
        break;
    }
    ;
}
```

The **break** statement can be used to transfer control flow out of any loop (**while**, **for**, **do**) body, as indicated by the arrow, and down to the statement past the end of the loop. Typically, this would be done in conjunction with an **if** statement. If the **break** is used alone (e.g., without the **if** statement), the statements in the loop body beyond the **break** will never be executed.

**continue**

```
do
{
    ;
    if (cond)
    {
        continue;
    }
    ;
}
while (cond);
```

The **continue** statement is similar to the **break** statement, but the loop condition is evaluated and if true, the body of the loop body is executed again. Hence, as indicated by the arrow, control is not transferred out of the loop, but rather out of the sequence within the loop (**while**, **for**, **do**).

Exception Handling

```
try
{
    stmt1;
}

! catch(E)
{
    stmt2;
}
finally
{
    stmt3;
}
```

In Java, the control construct for exception handling is the **try..catch** statement with optional **finally** clause. In the example at left, if **stmt1** generates an exception **E**, then control is transferred to the corresponding **catch** clause. After the catch body is executed, the **finally** clause (if present) is executed. If no exception occurs in the try block, when it completes, the **finally** clause (if present) is executed.

The **try..catch** statement can have multiple **catch** clauses, one for each exception to be handled.
3.4 CSD Templates

In Figure 30, the basic CSD control constructs, described above, are shown in the CSD Window. These are generated automatically based on the text in the window. In addition to being typed or read from a file, the text can be inserted from a list of templates by selecting **Templates** on the CSD Window tool bar.

![CSD Control Constructs generated in CSD Window](image_url)

Figure 30. CSD Control Constructs generated in CSD Window
3.5 Hints on Working with the CSD

The CSD is generated based on the source code text in the CSD Window. When you click View -- Generate CSD (or press F2), jGRASP parses the source code based on a grammar or syntax that is slightly more forgiving that the Java compiler. If your program will compile okay, the CSD should generate okay as well. However, the CSD may generate okay even if your program will not compile. Your program may be syntactically correct, but not necessarily semantically correct. CSD generation is based on the syntax of your program.

**Enter code in syntactically correct chunks** - To reap the most benefit from using the CSD when entering a program, you should take care to enter code in syntactically correct chunks, and then regenerate the CSD often. If an error is reported, it should be fixed before you move on. If the error message from the generate step is not sufficient to understand the problem, compile your program and you will get a more complete error message.

“Growing a program” is described in the table below. Although the program being “grown” does nothing useful, it is both syntactically and semantically correct. More importantly, it illustrates the incremental steps that should be used to write your programs.

<table>
<thead>
<tr>
<th>Step</th>
<th>Code to Enter</th>
<th>After CSD is generated</th>
</tr>
</thead>
</table>
| 1.   | public class MyClass  

}{

} | public class MyClass  

{  

} |
| 2.   | public class MyClass  

{  

myMethod()  

{  

}  

} | public class Hello  

{  

myMethod()  

{  

}  

} |
| 3.   | public class MyClass  

{  

myMethod()  

{  

while (true)  

{  

;  

}  

}  

} | public class MyClass  

{  

myMethod()  

{  

while (true)  

{  

;  

}  

}  

} |

Although this program is not intended to be useful, it does illustrates the process of growing a program, and generating the CSD each step of the way.
3.6 Reading Source Code with the CSD

The CSD notation for each of the control constructs has been carefully designed to aid in reading and scanning source code. While the notation is meant to be intuitive, there are several reading strategies worth pointing out, especially useful with deeply nested code.

**Reading Sequence**
The visualization of sequential control flow is as follows. After statement \( s(1) \) is executed, the next statement is found by scanning down and to the left along the solid CSD stem. While this seems trivial, its importance becomes clearer with the *if* statement and deeper nesting.

```plaintext
s(1);
s(2);
s(3);
```

**Reading Selection**
Now combining the sequence with selection (*if.. else*), after \( s(1) \), we enter the *if* statement marked by the diamond. If the condition is *true*, we follow the solid line to \( s(2) \). After \( s(2) \), we read down and to the left (passing through the dotted line) until we reach the next statement on the vertical stem which is \( s(4) \). If the condition is *false*, we read down the dotted line (the false path) to the *else* and then on to \( s(3) \). After \( s(3) \), again we read down and to the left until we reach the next statement on the stem which is \( s(4) \).

```plaintext
s(1);
if (cond)
   s(2);
else
   s(3);
s(4);
```

**Reading Selection with Nesting**
As above, after \( s(1) \), we enter the *if* statement and if \( \text{cond1} \) and \( \text{cond2} \) are true, we follow the solid lines to \( s(2) \). After \( s(2) \), we read down and to the left (passing through both dotted lines) until we reach the next statement on the stem which is \( s(4) \). If the \( \text{cond1} \) is *false*, we read down the dotted line (the false path) to \( s(4) \). If \( \text{cond2} \) is false, we read down the dotted line to the *else* and then on to \( s(3) \). After \( s(3) \), again we read down and to the left until we reach the next statement on the
<p>| Stem which is s(4). |</p>
<table>
<thead>
<tr>
<th>Reading Selection with Even Deeper Nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>If cond1, cond2, and cond3 are true, we follow the solid lines to s(2). Using the strategy above, we immediately see the next statement to be executed will be s(7).</td>
</tr>
<tr>
<td>If cond1 is true but cond2 is false, we can easily follow the flow to either s(4) or s(5) depending on the cond4.</td>
</tr>
<tr>
<td>If s(4) is executed, we can see immediately that s(7) follows.</td>
</tr>
<tr>
<td>In fact, from any statement, regardless of the level of nesting, the CSD makes it easy to see which statement is executed next.</td>
</tr>
<tr>
<td>Reading without the CSD</td>
</tr>
<tr>
<td>It should be clear from the code at right that following the flow of control without the CSD is somewhat more difficult.</td>
</tr>
<tr>
<td>For example, after s(3) is executed, s(7) is next. With the CSD in the previous example, the reader can tell this at a glance. However, without the CSD, the reader may have to read and reread to ensure that he/she is seeing the indentation correctly.</td>
</tr>
<tr>
<td>While this is a simple example, as the nesting becomes deeper, the CSD becomes even more useful.</td>
</tr>
<tr>
<td>In addition to saving time in the reading process, the CSD aids in interpreting the source code correctly, as seen in the examples that follow.</td>
</tr>
</tbody>
</table>
### Reading Correctly with the CSD

| Consider the fragment at right with s(1) and s(2) in the body of the `if` statement. | s(1);
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
| `if` (cond)
| s(2);
| s(3); |

After the CSD is generated, the reader can see how the compiler will interpret the code, and add the missing braces.

| Here is another common mistake made glaring by the CSD. | `if` (cond);
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-------------------------------------------------------</td>
</tr>
</tbody>
</table>
| s(2);
| s(3); |

Most likely, the semi-colon after the condition was unintended. However, the CSD shows what there rather than what was intended.

| Similarly, the CSD provides the correct interpretation of the `while` statement. | `while` (cond)
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
| s(2);
| s(3); |

Similarly, the CSD provides the correct interpretation of the `while` statement.

| Similarly, the CSD provides the correct interpretation of the `while` statement. | `while` (cond)
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| s(2);
| s(3); |

Unintended semi-colon...
4 Projects

A project in jGRASP is based on the concept of grouping files together that may be located in one or more directories. When a “project” is created, all information about it is stored in a file with the .gpj extension. If you have not created a project, **jGRASP Project: [default]** should be displayed at the top of the Desktop, which indicates the general default project file is being used.

4.1 Creating a Project

On the Control Panel (Desktop) tool bar, click **Project – New Project…** to open the **New Project** window (Figure 31).

![Figure 31. Creating a Project](image-url)
Within the New Project window (Figure 32), go to the directory where you want the project to reside, and enter the project file name. It is recommended that the project file be stored in the same directory as the file containing `main`. A useful naming convention in Java is `ClassnameProject` where `Classname` is the name of the class that contains `main`. For example, if the PersonalLibrary class contains `main`, an appropriate name for the project file would be `PersonalLibraryProject`.

Click **Create** to save the project file. Notice the `.gpj` extension is automatically appended to the file name. As soon as the project is created, it becomes the current project, and the new project name replaces “[default]” at the top of the Desktop.

![New Project window](image)

**Figure 32. New Project window**
4.2 Adding files to the Project

The easiest way to add files to the current project is from the Browse pane (Figure 33). Simply left click on the file name to select it, then right click and select Add to Project – Relative Path. Repeat this for each file to be added, or alternatively, select multiple files (holding down the shift or control key), and add the highlighted files all at once. To see the list of files in the project, select the Project pane from among the tabs for Browse, Project, Find, and Debug at the bottom the left pane of the Desktop.

Figure 33. Adding a file to the Project
4.3 Removing files from the Project

While in the Project pane (Figure 34) you can remove files from the project by selecting one or more file(s), then right click and select **Remove from Project**. Note that removing a file from the project does not delete the file from its home directory. In fact, currently jGRASP does not provide a way to delete a file.

![Figure 33. Removing a file from the Project](image-url)
4.4 Generating Documentation for the Project (Java only)

Now that you have established a project, you have the option to generate project level documentation for your Java classes. You can automatically generate documentation in the style of an application programmer interface (API). To begin the process of generating the documentation, click on the Project menu as shown in the figure below, and select Generate Documentation. This will bring up the “Generate Documentation for Project” dialog, which asks for the directory where the generated HTML files are to be stored. The default directory name is the name of the project with “_doc” appended to it. So for the example, the default will be PersonalLibraryProject_doc. Using the default name is recommended so that your documentation directories will have a standard naming convention. However, you are free to use any directory as the target. After choosing the directory, click Generate to start the process. jGRASP calls the javadoc utility, included with the J2SDK, to create a complete hyper-linked document within a few seconds.

Figure 34. Generating Documentation for the Project
This can be done on a file-by-file basis or for a complete project as shown in Figure 35 below. Note, in this example, even though no JavaDoc comments were included in the source file, the generated documentation is still quite useful. However, for even more useful results, JavaDoc formal comments should be included in the source code. When the documentation is generated for an individual file, it is stored in a temporary directory for the duration of the jGRASP session. For projects, the documentation files are stored in a directory that becomes part of the project, and therefore, persists from one jGRASP session to the next. If any changes are made to a project source file (and the file is resaved), jGRASP will indicate that the documentation needs to be regenerated; however, the user may choose to view the documentation files without updating them.

![Figure 35. Project documentation](image-url)
4.5 Jar file Creation and Extraction

jGRASP provides a utility for the creation and extraction of Java Archive files (JAR) for your project. The “Create Jar File for Project” option, which is found on the Project menu, allows you to create a single compressed file containing your entire project.

The “Jar/Zip Extractor” option enables you to extract the contents of a JAR or ZIP archive file.

These topics are described in more detail in Part 2 - Reference.

4.6 Changing Projects

To change from one project to another, you may create a new project as described above, or you may select from other existing projects. On the Control Panel (Desktop) tool bar, click Project – Open Project... to open the Open Project window. Within the Open Project window, go to the directory where the project file is located, select to the project file. The project file can be opened by double-clicking on the file name or by clicking the Open button on the lower right of the Open Project window. After the project is opened, its name should be displayed at the top of the Desktop.

4.7 Closing a Project

On the Control Panel (Desktop) tool bar, click Project – Close Project... to close the current project. After closing a project, jGRASP Project: [default] should be displayed at the top of the Desktop. All project information is saved when you close the project, as well as when you exit jGRASP.

For more information, see Project Menu in Part 2 - Reference.
5 UML Class Dependency Diagrams

Java programs usually involve multiple classes, and there can be many dependencies among these classes. To fully understand a multiple class program, it may be necessary to keep track of the interclass dependencies. Although this can be done mentally for small programs, it is usually helpful to see these dependencies in a class diagram. jGRASP provides for the automatic generation of class dependency diagrams based on the Unified Modeling Language (UML). Since a UML class diagram is based on multiple classes, jGRASP utilizes the “Project” to determine which user classes to include.

(1) In order to generate a meaningful UML diagram, you must create a jGRASP project file (.gpj) as described in the previous section. You should include all of your source files (.java) in the project, and you may or may not include other files (e.g., .class, .dat, .txt, etc.).

(2) The information jGRASP uses to generate the UML is located in the .class files. Recall, .class files are generated when you compile your Java program files. Therefore, you must compile your .java files prior to generating the UML diagram. Note, the .class files do not have to be in the project file, but they should be in the same directory as the .java files.

The remainder of this section assumes you have created a project file, and that you have compiled your program files.

5.1 Opening the Project

In preparation for generating the UML class dependency diagram for a particular project, you need to open that project. There are two ways to open a project.

1) On the Desktop tool bar, click Project – Open Project, and then select the project from the list of project files displayed in the Open Project dialog as shown in Figure 36.

2) In the Browse pane (tab in the left pane of the Desktop), double-click the project file.

Figure 36. Open Project Dialog
In Windows, the project name will be displayed at the top of the Desktop. You can also click the Project tab in the left pane of the Desktop to see the current project. If you are not in the right project, you can double-click on the project file in the Browse pane to open a project. If you additional help with opening a project, review the previous section.

5.2 Generating the UML

In Figure 37 below, the left pane is set to the Project tab. The PersonalLibraryProject file is listed along with source files in the project. To generate the UML class dependency diagram, on the Desktop menu, click on UML, then Generate/Update UML.

Figure 37. Generating the UML
The UML Window should pop up with a diagram of all class files in the project plus possibly a number of Java library class files indicated by gray boxes as shown in Figure 38. The UML Window is divided into three panes. The top pane contains a panning rectangle that allows you to position the UML diagram in the large right pane by dragging the panning rectangle around. To the right of the panning pane are the buttons for scaling the UML: divide by 2, divide by 1.2, no scaling (1), multiply by 1.2, and multiply by 2. The Update button should be applied each time you re-compile your program. Figure 38 has been scaled to 1.2, and the panning rectangle has been moved slight to the right to re-position the UML as shown.

If your project includes class inheritance hierarchies and/or other dependencies as in the example, then you should see the appropriate red and black dependency lines. Next, you will need to learn how to adjust indicate which objects and dependencies you want in your UML diagram. However, if you are okay with the classes and dependencies shown in the diagram, go to the next section and begin laying out your diagram.

Figure 38. UML Window after initial Generate
jGRASP provides two approaches to controlling the contents/display of your UML diagram. The first (Edit – Settings) allows you to control the contents of the diagram by excluding certain categories of classes (e.g., external superclasses, external interfaces, and all other external references). After you generated/updated the diagram based on these exclusions, the second approach (View -- ) allows you to make visible (or hide) certain categories of classes and dependencies in the UML diagram. Both approaches are described below.

Consider the following example. Suppose you see a large number JDK classes (gray boxes) that you want to exclude. Then you may need to edit the UML generation settings in order to exclude these items from the diagram. Also, if you do not see the red and black dependency lines expected, then you may need to change the View settings. These are described below.

**Excluding items from the diagram** - On the UML window menu, click on Edit – Settings…, which will bring up the UML Settings dialog. For example, to exclude all JDK classes, under Exclude by Name, turn ON (check mark) JDK Classes, java.lang.Object, and Synthetic Classes as shown in Figure 39.

Note, synthetic classes are created by the Java compiler, and are usually not included in the UML settings.
After checking the items you want excluded, click the OK button, which should close the dialog. Now click the Update button in the upper right corner of the UML window. This should remove all JDK classes (the gray boxes) from the diagram and from the project as shown in Figure 40. To get them back in the diagram, you would need to turn off the exclude option and update the diagram again. If you had previously laid out the JDK classes, this formatting was lost when the classes were removed from the project by the update after the exclude. For more information see UML Settings in the Reference section.

**Making objects visible in the diagram** - On the UML window menu, click on View – Visible Objects, then check or uncheck the items on the list as appropriate. For example, for the JDK classes and/or other classes outside the project to be visible, **External References** must be checked ON. Clicking (checking) ON or OFF any of the items on the Visible Objects list simply displays them or not, and their previous layout is retained when they are redisplayed. In general, you probably want all of the items on the list checked on as shown in Figure 40. Note that if items have been excluded from the diagram, as described above, then making them visible will have no effect since they are not part of the diagram. For more information see View Menu in the Reference section.
**Making dependencies visible** - On the UML window menu, click on View – Visible Dependencies, then check or uncheck the items on the list as appropriate. The only two categories of dependencies in the example project are Inheritance and Other. Inheritance dependencies are indicated by black lines with closed arrowheads that point from child to the parent to from an is-a relationship. Red dashed lines with open arrowheads indicate other dependencies. These include the has-a relationship that indicates a class includes one or more instances of another class. If a class references an instance variable or method of another class, the red dashed arrow is drawn from the class where the reference is made to the class where the referenced item is defined. In general, you probably want to make all dependencies visible, as indicated in Figure 41.

**Displaying the Legend** - On the UML window menu, click on View – Legend, then set the desired options. Typically, you will want the following options checked on: Show Legend, Visible Items Only, and Small Font. Notice, the legend has been visible in all of UML figures. Before the JDK classes were excluded (Figure 39), they were included in the legend, but not after the Update. When you initially generate your UML diagram, you may have to pan around it to locate the legend. Scaling the UML down (e.g., dividing by 2) may help. Once you locate it, just select it, then drag to the location where you want it.
as described in the next section.

5.3 Laying Out the UML Diagram

Currently, jGRASP has limited automatic layout capabilities. However, manually arranging the class symbols in the diagram is straightforward, and once this is done, jGRASP remembers your layout from one generate/update to the next.

To begin, locate the class symbol that contains *main*. In our example, this would be the PersonalLibrary class. Remember the project name should reflect the name of this class. Generally, you want this class near the top of the diagram. Left click on the class symbol and then while holding down the left mouse button, drag the symbol to the area of the diagram you want it, and then release the mouse button. Now repeat this for the other class symbols until you have the diagram looking like you want it. Keep in mind that class–subclass relationships are indicated by the *inheritance arrow* and that these should be laid out in a tree-down fashion. You can do this automatically by selecting all classes for a particular class–subclass hierarchy (hold down SHIFT and left-click each class). Then on UML window menu, click on *Layout – Tree Down* to perform the operation. With a one or more classes selected, you can move them.

![Figure 42. First cut at laying out the UML](image-url)
as a group. Figures 42 shows the UML diagram after the PersonalLibrary class and the legend have been repositioned. For Figure 43, the UML settings (Edit – Settings) have been changed to include the JDK classes, and these have been dragged as a group to the lower part of the diagram. Note, you can experiment with making these external classes visible or not by going to View – Visible Objects – then uncheck External References.

Here are several heuristics for laying out your UML diagrams:

(1) The class symbol that contains main should go near the top of the diagram.
(2) Classes in an inheritance hierarchy should be laid out tree-down, and then moved as group.
(3) Other dependencies should be laid out with the red dashed line pointing downward.
(4) JDK classes, when included, should be toward the bottom of the diagram.
(5) Line crossings should be minimized.
(6) The legend is usually below the diagram.

Figure 43. Second cut at laying out UML (with JDK classes included)
5.4 Displaying the Members of a Class

If you are working with an example laying out your UML diagram as described in the section above, when you select a class, you may have noticed that the class members (fields, constructors, and methods) were displayed in Info tab of the left pane of the UML Window. In Figure 44, class Fiction has been selected and its fields, constructors, and methods are displayed in the left pane. This information is only available when the source code for a class in the project. In the example below, the System class from package java.lang is an external class so selecting it would result in a "no data" message.

Figure 44. Displaying class members
5.5 Displaying Dependencies Between Two Classes

An arrow or edge between two classes in the UML diagram indicates that there are one or more dependencies in the direction of the arrow. In Figure 45, the edge drawn from PersonalLibrary to Fiction has been selected, indicated by the large arrowhead. The list of dependencies in the Info tab of the left pane includes one field (pages), one constructor (Fiction), and one method (getMainCharacter). These are the resources that PersonalLibrary uses from Fiction. Reviewing all of the dependencies among the classes in your object-oriented program will usually prove insightful and provide you with a more in-depth understanding of the source code.
5.6 Finding a Class in the UML Diagram

Since a UML diagram can contain many classes, it may be difficult to locate a particular class. In fact, the class may be off the screen. The Goto tab in the left pane provides the list of classes in the project. Clicking on a class in the list brings it to the center of the UML diagram.

5.7 Opening Source Code from UML

The UML diagram provides a convenient way to open source code files. Simply double-click on a class symbol, and the source code for the class is opened in a CSD Window. For example, when the PersonalLibrary class is double-clicked, as shown in Figure 46, the corresponding CSD Window is opened on the Desktop.

Figure 46. Opening CSD Window from UML
5.8 Saving the UML Layout

When you close a project, change to another project, or simply exit jGRASP, your UML layout is automatically saved in the project file (.gpj). The next time you start jGRASP, open the project, and open the UML window, you should find your layout intact.

If the project file is created in the same directory as the .java and .class files, and if you added the source files with relative paths, then you should be able to ship the files around (e.g., email them to your instructor).

5.9 Printing the UML Diagram

On UML window menu, click on Print – Print Preview to see how your diagram will look on the printed page. If okay, click the Print button in the lower left corner of the Print Preview window. Otherwise, if the diagram is too small or too large, you may want to go back and scale it using the scale factors near the top right of the UML window, then preview it again.

For details see UML Class Dependency Diagrams in Part 2 - Reference.
6 The Integrated Debugger

Your skill set for writing programs would not be complete without knowing how to use a debugger. While the connotation of a debugger is that its purpose is to assist in finding bugs, it can also be used as a general aid for understanding your program as you develop it. JGRASP provides a highly visual debugger for Java, which is tightly integrated with the Desktop and which includes all of the traditional features expected in a debugger.

If the example program used in this section is not available to you, or if you do not understand it, simply substitute your own program in the discussion.

6.1 Preparing to Run the Debugger

In preparation to use the debugger, click Compiler on the menu of the CSD Window to be sure Debug Mode is checked as illustrated in Figure 47. If the box in front of Debug Mode is not checked, click on the

Figure 47. Setting the compiler to Debug Mode
box. When you click on Compiler again, you should see that Debug Mode is checked. When you compile your program in Debug Mode, information about the program is included in the .class file that would normally be omitted. This allows the debugger to display useful details as you execute the program. If your program has not been compiled with Debug Mode checked, you should recompile it before proceeding.

6.2 Setting a Breakpoint

In order to examine the state of your program at a particular statement, you need to set a breakpoint. The statement you select must be "executable" rather than a simple declaration. To set a breakpoint in a program, move the mouse to the line of code and left-click the mouse to move the cursor there. Then right-click to display a set of options that includes **Toggle Breakpoint**. For example, in Figure 48 the cursor is on the first executable line in *main* (which declares *Book* hemingway ...), and after Toggle

![Figure 48. Setting a breakpoint](image)
Breakpoint is selected in the options popup menu, a small red stop sign symbol appears in the left margin of the line to indicate that a breakpoint has been set. To remove a breakpoint, you repeat the process since this is a toggle action. You may set as many breakpoints as needed.

6.3 Running a Program in Debug Mode

After compiler your program in Debug Mode and setting one or more breakpoints, you are ready to run your program with the debugger. You can start the debugger in one of two ways: (1) click Run – Debug on the CSD Window menu, as shown in Figure 49, or (2) click the debug symbol on the toolbar.

After you start the debug session, several things happen. In the Run window near the bottom of the Desktop, you should see a message indicating the debugger has been launched. In the CSD Window,
the line with the breakpoint set is eventually highlighted, indicating that the program is stopped at the breakpoint, and finally, on the left side of the jGRASP desktop the debugger pane is popped to the top. Each of these can be seen in Figure 50. Notice the debugger pane is further divided into three subpanes labeled **Threads**, **Call Stack**, and **Variables/Settings**. Each of the debugger subpanes can be resized by selecting and dragging one of the horizontal or vertical borders. This has been done in some of the figures that follow. The **Threads** subpane lists all of the active threads running in the program. In the example, the red thread icon indicates the program is stopped in **main**, and green indicates a thread is running. Beginners and intermediate users can ignore the thread pane. However, advanced users should find it quite useful for starting and stopping individual threads in their programs. The **Call Stack** subpane is useful to all levels of users since it shows the current call stack and allows the user to switch from one level to another in the call stack. When this occurs, the CSD Window that contains the source code associated with a particular call is popped to the top of the desktop. The **Variables/Settings** subpane shows the details of the current state of the program. Finally, when a line of source code is highlighted, it means that the line is about to be executed.

Figure 50. Desktop after debugger is started
6.4 Stepping Through a Program

After the program stops at the breakpoint, you can use the icons at the top of the debug pane to **single step**, **step into** a method call, **step out** of a method, **run to the cursor**, **pause** the current thread, **resume**, and **suspend** new thread, while watching the call stack and contents of variables change dynamically. The integrated debugger is especially useful for watching the creation of objects as the user steps through various levels of constructors. The jGRASP debugger can be used very effectively to explain programs, since a major part of understanding a program is keeping track (mentally or otherwise) of the state of the program as one reads from line to line.

We will make two passes through the example program as we explain it. During the first pass, we will “step” through the program without “stepping into” any of the method calls, and we will concentrate on the Variable section. In Figure 50, Variables/Settings pane indicates no local variables have been declared. Figure 51 shows the results of “stepping” to the next statement. Notice that under Locals in the

![Image of debugger interface](image_url)

**Figure 51. Desktop after hemingway (Book) object is created**
Variable/Settngs pane, we now have an instance of Book called hemingway. Objects, represented by a colored square, can be opened and closed by clicking the “handle” in front of the square object. Primitives, like the integer pages, are represented by colored triangles. In Figure 51, hemingway has been opened to show the author, title, and pages fields. Each of the String instances (e.g., author) can be opened to view the individual characters. Notice that all the fields in hemingway are green, which indicates they were declared in the class Book.

When an array is opened in the debugger, only the first ten elements (indexed 0 to 9) are displayed. To see other elements, left-click the array to select it, then click one more time. Note, this is not a double-click, but rather two single clicks. The first time you do this there may be a short delay, but a slider bar will popup that allows you to display a range of any ten items.

After executing the next statement, Figure 52 shows an instance of the Fiction class called clancy that has been created. In the figure, clancy has been opened to reveal its fields. The field “mainCharacter” is green, indicating it is defined in Fiction. The other fields (author, title, and pages) are amber, which

```
import java.util.*;

public class PersonallLibrary {
    public static void main(String[] args) {
        Book hemingway = new Book("Hemingway", 
                                "Green Hills of Africa", 234);
        Fiction clancy = new Fiction("Clancy", 
                                   "The Hunt for Red October", 
                                   490, "Sean");
        System.out.println(clancy.getMainCharacter());
        System.out.println(hemingway);
        System.out.println(clancy);
    }
}
```
indicates these fields were inherited from Book.

As you continue to step through your program, you should see output of the program displayed in the Run I/O window in the lower half of the Desktop. Eventually, you should reach the end of the program and see it terminate. When this occurs the debug pane and its subpanes should become blank, indicating that the program is no longer running.

Now we are ready to make a second pass and “step in” to the methods called. Tracing through a program by following the calls to methods can be quite instructive in the obvious way. In the object-oriented paradigm, it is quite useful for illustrating the concept of constructors. As before, we need to run the example program in the debugger by clicking Run – Debug on the CSD Window menu or by clicking the debug symbol on the toolbar. After arriving at the breakpoint, we “step in” and the constructor for class Book pops up in the CSD Window (Figure 53). You can then step through this method in the usual way, eventually returning to the statement in the main program that called the constructor.

FIGURE 53. After next stepping into the Book constructor
There are many other scenarios where this approach of tracing through the process of object construction is useful and instructive. For example, consider the case where the Fiction constructor for “clancy” is called and it in turn calls the super constructor located in Book. By stepping into each call, you can see not only how the program proceeds through the constructor’s code, but also how fields are initialized.

Another even more common example is when the toString method of an object is invoked indirectly in a print statement (System.out.println). The debugger actually takes the user to the object’s respective toString method.

### 6.5 Debugging a Program

You have, no doubt, noticed that the previous discussion was only indirectly related to the activity of actually finding and removing bugs from your program. It was intended to show you how to set and unset breakpoints and how to step through your program. Typically, to find a bug in your program, you need to have an idea where in the program things are going wrong. The strategy is to set a breakpoint on a line of code prior to the line where you think the problem occurs. When the program gets to the breakpoint, you must ensure that the variables have the correct values. Assuming the values are okay, you can begin stepping through the program, watching for the error to occur. Of course, if the value of one or more of the variables was wrong at the breakpoint, you will need to set the breakpoint earlier in the program.

For additional details, see Integrated Java Debugger in Part 2 - Reference.
7 Compiler Environment Settings

[work in progress]

For details see Compiler Environment Settings in Part 2 - Reference.
Part 2 - Reference
8 Installing jGRASP

8.1 Important Notes
This is jGRASP version 1.5.0. Check [http://www.eng.auburn.edu/grasp](http://www.eng.auburn.edu/grasp) for the latest version of jGRASP.

jGRASP is written in Java. Java 1.2 or higher must be installed on your machine in order to run jGRASP. The full JDK 1.3 or higher (not the JRE) is required in order to use the integrated Java debugger.

jGRASP does not include any compilers. The JRE bundled version and all other versions of jGRASP do not include a Java compiler. In order to compile programs you must have a compiler installed.

8.2 Which JVM to Use
If you have downloaded a JRE bundled version of jGRASP, this is not a concern. Otherwise, Sun JDK 1.4 is suggested. When jGRASP starts, you may get a message telling you that jGRASP has known problems or minor problems, or has not been tested on your system.

To use the integrated Java debugger, JDK 1.3 or higher is needed (and you should develop under 1.3 or higher also).

8.3 Installing from the Self-Extracting exe on Windows
Before running the installation file, you should close all applications. If a file from a previous installation is in use, the installer may give you a "Can not copy file" or "File in use" message. This would happen, for example, if you had the readme.txt file open in MS Word. If this happens, iconify the installer, close the file or the application that is using the file, pop the installer back up, and click "Retry" on the error dialog. If the installer will not run, you will have to install manually.

On Windows NT, if you have Administrator access, the start menu items and desktop shortcut will be installed for all users.

8.4 Manual Installation on Windows
Unzip the distribution file in the directory where you wish to install jGRASP. This will create a jgrasp directory containing all the files. Add a shortcut or start menu item to jgrasp\bin\jgrasp.exe.

8.5 Installing on Mac OS X
Unzip and untar the .tar.gz file, if this did not happen automatically when you downloaded. You can use Stuffit Expander or from a terminal, "gunzip jgrasp*.tar.gz" then "tar xf jgrasp*.tar". Then double click on the .pkg file. A root password is required to install. The first time you run jGRASP, the CSD font will be installed on your system, and a soft link to the jgrasp startup script (for command line execution) will be created in /usr/bin or your $HOME/bin directory.

8.6 Manual Installation on Mac OS X
Unzip the distribution file in the directory where you wish to install jGRASP (you can use Stuffit Expander to do this). This will create a jgrasp directory containing all the files. You can then click on "jgrasp/bin/macosx/jGRASP Control Shell" to run jGRASP. The first time you run, the CSD font will be
installed on your system, and a soft link to the jgrasp startup script (for command line execution) will be
created in /usr/bin or your $HOME/bin directory.

If you want to create a "normal" OS X application, copy the "jgrasp/bin/macosx/jGRASP Control
Shell.app" directory, then move the original "jgrasp" directory into the new "jGRASP Control
Shell.app/Contents/Resources" directory.

8.7 Installing on x86 Linux, SPARC Solaris

Unzip the distribution file in the directory where you wish to install jGRASP. This will create a jgrasp
directory containing all the files. You may want to add the "bin" subdirectory of this directory to your
execution path or create a soft link to .../jgrasp/bin/jgrasp from a directory on the executable path.

8.8 Installing on Other Systems

Unzip the distribution file in the directory where you wish to install jGRASP. All the functions of jGRASP
will work except for the compile and run functions.

If you are on a Unix system other than Linux or Solaris, you can build a "wedge" program in order to use
the compile and run functions of jGRASP (C compiler and knowledge of how to use it is necessary). The
source can be found at jgrasp/src/linux_run.c. The executable must be jgrasp/jbin/sys_run. The source
for a client-server shell can be found at jgrasp/src/linux_exec.c. That executable can have any name; it is
only used externally. For the best chance of success, compile with default optimization.

Please contact us if you succeed in doing this. We can incorporate your executables in a future release of
jGRASP.

If you are on a system other than Windows or a UNIX variant and have a good knowledge of system
programming on your system, you can build a wedge based on the UNIX wedge sources found in
jgrasp/src. The executable must be jgrasp/jbin/sys_run.

8.9 Network Administration

jGRASP supports common settings for all users on a network. The common settings directory can be
specified at installation time if using the self-extracting exe on Windows. Otherwise, in the "data"
directory of the installation, edit the file called "admin". Specify the directory to be used for administrator settings in
this file. This should be the only contents of the file, and not be followed by a line feed or carriage return.
Use a full path to specify the directory. This directory must be accessible and readable by all users, and
writable only for administrators. You should select a location outside the jGRASP distribution, so that you
can continue to use the settings after upgrading.

When an administrator (anyone with write access in the common settings directory) runs jGRASP, menu
items that allow you to change administrator settings will appear on the control panel settings menu. Also,
when copying or creating compiler environments, you will have the choice of creating them for all users or
just the current user.

9 Running jGRASP

jGRASP requires a Java 1.2 or better virtual machine and system classes to run. You can get the latest
JDK from Sun Microsystems.

You will probably need about 48 Megs of memory to run jGRASP, but this depends on the virtual
machine, jit compiler, and which other applications you run concurrently.
9.1 Running on Windows

The `bin\jgrasp.exe` file will start jGRASP in single-instance mode. Running jGRASP a second time will pop up the currently running instance. If you chose to associate file extensions with jGRASP during installation, clicking on an associated file will pop up the file in the currently running instance. A Windows application titled "jGRASP Control Shell", which initially runs minimized, displays the jGRASP output (including stack dumps if there is a crash) and allows the user to kill jGRASP if it locks up. It will first try a "soft kill", and if jGRASP does not exit within a few seconds, a "hard kill" will be issued.

"bin\jgrasp.exe" will search for a java.exe to run jGRASP in the following manner:

1) get from "-a" command line parameter if present
2) get from JGRASP_JAVA environment variable if present
3) check the PATH for javac.exe - if found look for java.exe in the same place (avoids finding JRE before JDK)
4) check the PATH for java.exe - if it is found in the system directory, Windows directory, or any subdirectory of those, or if not found, check the registry for the distribution location of a Sun JDK installation.
5) check for the JRE bundled with jGRASP

The method used to find java.exe and the java command used to start jGRASP will be displayed in the "jGRASP Windows exec" window.

9.2 Running on Mac OS X

Double clicking on the "jGRASP Control Shell" file, or on "jgrasp/bin/macosx/jGRASP Control Shell" if jGRASP was manually installed, will start jGRASP in single-instance mode. Running jGRASP a second time will pop up the currently running instance. Clicking on an associated file will pop up the file in the currently running instance. An application titled "jGRASP Control Shell", which initially runs minimized, displays the jGRASP output (including stack dumps if there is a crash) and the command used to start jGRASP, and allows the user to kill jGRASP if it locks up. It will first try a "soft kill", and if jGRASP does not exit within a few seconds, a "hard kill" will be issued.

The first time you run jGRASP it will create a soft link to a startup shell in /usr/bin or $HOME/bin. Then in a terminal shell you can run "jgrasp" and open files in an already running jGRASP with "jgrasp filename".

9.3 Running on Linux, Solaris, and other UNIX

Similar single-instance mode executables exist for x86 Linux and SPARC Solaris. These can both be accessed through the bin/jgrasp shell script, which will also pass the location of the jGRASP installation to the correct executable so you don't have to. You can run the executables directly (they are in subdirectories of "bin" with obvious names) if you specify the location of the jGRASP installation through a JGRASP_HOME environment variable or "-j path" at the command line. A window titled "jGRASP exec", which initially runs minimized, allows the user to kill jGRASP if it locks up. It will first try a "soft kill", and if jGRASP does not exit within a few seconds, a "hard kill" will be issued.

On other UNIX, bin/jgrasp should run jGRASP directly, without a single-instance shell. See Installing on Other Systems for information on compiling the native parts of jGRASP.

For the "java" startup script, if the location of java is not specified through a "-a" command line parameter or JGRASP_JAVA environment variable, then "java" must be on the path.
9.4 Requirements for the Integrated Debugger

For both the Windows and UNIX/Linux startup programs, to use the integrated Java debugger, "java.exe" or "java" must be in the JDK directory structure (in /bin, where tools.jar is in /lib), unless the "-cp" command line argument or JGRASP_CLASSPATH environment variable is used. If "java.exe" is the redirecting startup program on Windows, the registry will be searched for an appropriate JDK distribution.

On Mac OS X, the integrated debugger will be available by default.

9.5 Command Line Arguments for the Single Instance Executables and Shells

A filename can be given as the last argument. The leading dash for flags can also be a forward slash on Windows.

- -h - display a help message.
- -l line_number - select line_number for the filename argument.
- -g language - force a language for the filename argument. language can be C, +, J, A, V for C, C++, Java, Ada, and VHDL.
- -d directory - alternate settings directory. See "Command line arguments for jGRASP itself" below.
- -a JGRASP_JAVA - specify the java command used to start jGRASP. This can also be set in a JGRASP_JAVA environment variable. If not specified, "java" will be called on UNIX/Linux, and on Windows the PATH, then standard JDK locations will be searched for "java.exe". If a JRE is bundled with jGRASP, that VM will be used if no other is found.
- -j JGRASP_HOME - specify the root of the jGRASP installation. This can also be set in a JGRASP_HOME environment variable. The "jgrasp" shell script attempts to "locate itself", so this should not be necessary under normal circumstances. This should never be necessary on Windows.
- -cp JGRASP_CLASSPATH - specify a classpath to be used for jGRASP. This can also be set in a JGRASP_CLASSPATH environment variable. The "jgrasp.jar" file will always be prepended to this path. Under normal circumstances using this argument should not be necessary.
- -e JGRASP_EXTENSIONS - specify a path for jGRASP elements and user extensions that should persist between versions and may be supplied in optional packages. For now, the classes/system and classes/user subdirectories will be added to the classpath when any system Java compiler environment is used. This is just a convenient way to make library packages available to your Java programs. This can also be set in a JGRASP_EXTENSIONS environment variable. In Windows, a default location is chosen during installation, and will be retrieved from the registry. Otherwise, the default is the "extensions" directory in the jGRASP distribution.
- -Jjvm_arg - pass jvm_arg to the JVM running jGRASP. For example, -J-Xmx90m will make the maximum size of the memory allocation pool for jGRASP 90 megs.

9.6 Command Line Arguments for jGRASP Itself

- -d directory - use directory as the settings directory instead of USER_HOME/.grasp_settings. directory must be a full path name. By using this, you can run two jGRASP sessions concurrently and they will not interfere with each other, or you can keep different settings for different users on the same system.
- -h - print a help message.
10 Using Help
This help is best viewed in a browser that supports frames, or from jGRASP.
If you are using jGRASP, the frame on the left side can be one of:

- Contents - links to major subjects.
- Index - alphabetized list of links to subject headings.
- Search - a search of the help text.

Clicking on any link will change the page in the frame on the right side.
The search page searches for sections containing all words entered in the "Search" field.

11 Control Structure Diagram (CSD)
The Control Structure Diagram, or CSD, is an algorithmic level diagram 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 is an alternative to flow charts and other graphical representations of algorithms. The major goal behind its creation was that it be an intuitive and compact graphical notation that was easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and class diagrams.
A primary purpose of jGRASP is to provide for automatic generation of the CSD for source code written in Java, C, C++, Ada, and VHDL (support for additional languages is planned).

12 Interface Terminology
These terms are used throughout this documentation.

- Control Panel - the top-level jGRASP window.
- CSD Window - the source code editing windows.
- Virtual Desktop or Desktop - the optional desktop window that contains CSD windows.
- Tab Page or Page - one page of a tabbed window.
- Menu Accelerator - a function key or control-key combination that performs the same function as a menu item.
- Swing - also called JFC, the Java gui class library that jGRASP uses.
- Compiler Environment - a group of settings that specify how jGRASP will interface with a particular compiler.
13 Known Bugs

Check the known bugs list at [http://www.eng.auburn.edu/grasp/](http://www.eng.auburn.edu/grasp/) for an updated list of known bugs, if you are not already there. Check the future plans page for a list of future features. Here is the bug-fix history. Back up your work - this is a Beta release, and has not been thoroughly tested. Save and back up your work frequently.

13.1 Our Known Bugs

On Windows NT, a 16 bit process that is run with the %CY flag (console mode) can not be killed until the virtual DOS machine initializes and starts the process (which can take several seconds). If this happens, the process can be killed by closing the console, which will initially be iconified.

The "Left Align" CSD setting does not work correctly. The code is not shifted before a save or compile, among other problems.

Windows native printing - CSD and text fonts will sometimes not be the same size (note: improved from "usually" to "sometimes"). The CSD font is not as symmetrical and "perfectly rendered" as it is on the screen or with JVM printing.

13.2 Bugs Caused by Java, Swing, or Window Manager Problems

These are difficult or impossible to work around. The eventual "fix" for them is (or will be) to get a newer version of the JDK. Many of them apply only to old versions of the JDK.

jGRASP may crash (most likely at startup) with "at sun.awt.font.NativeFontWrapper.initializeFont(Native Method)" at the top of the stack dump.

When using the integrated Java debugger, jGRASP and the target process will sometimes lock up. This is most likely due to bugs in the JDK, and happens much less frequently with the HotSpot vm and newer versions of the JDK. Note that on a dual processor NT or Win 2000 system and JDK 1.3 or earlier, this may happen frequently, making the integrated debugger effectively unusable.

On some JDK versions, the Help window will sometimes not come up, and a stack dump will be generated, the first time help is used. If this happens, help will not work until you shut down jGRASP and restart it.

Performance is bad and there is much unnecessary repainting with the Mac Look-And-Feel.

The help browser is not suitable for browsing the web. HTML display in Swing is limited and very buggy. The jGRASP help pages are constructed so as to work within the limitations and work around the bugs.

On some UNIX/Linux VMs, running jGRASP through the single-instance shell will cause the JVM to crash at shut-down. This seems to be harmless.

Windows installation may randomly fail. If it does, try it again.

Closing a CSD window will cause an exception on Blackdown and Sun Linux JDK 1.2.2. jGRASP will lock up when closed on these systems (you'll have to kill it manually). We will probably not work around the bug that causes these problems unless it is not fixed in the corresponding 1.3 releases. For now, do not use these systems unless you're willing to put up with the problems, and shut down now and then if you're opening and closing a lot of windows, because memory is leaked each time a window is closed.

Font problems on Windows JDK 1.2.2 (not seen on older or newer versions). If you use bold or italic text properties, text in CSD windows will appear normal, bold, or italic at random.

Click in the wrong place - when clicking in an unselected CSD window in the virtual desktop, the click position will be offset by the size of the border and menu. This can cause the cursor to move to an unintended location, the wrong text to be selected, the scrollbar to be activated (and probably stuck
down), a toolbar or menu item to be accidentally selected, and similar strange effects. This is fixed in JDK 1.3 final.

Freeze up - jGRASP may freeze up when starting. If you kill it and start again, it is likely to run (possibly after several tries). Note that after killing jGRASP you will get a warning that there is a currently running instance; choose "Run Anyway" to continue.

Memory leaks - due to Swing bugs, there are some small memory leaks. These have been minimized as much as was possible. Opening and closing a CSD window on the virtual desktop will leak up to 1K or so under some JDKs, so this should not be a problem unless you run jGRASP for weeks without restarting (like I do). There may be larger memory leaks that we are unaware of.

Caps lock - the caps lock key will not work on some systems. Having caps lock on may also cause menu accelerators not to work.

Display mode - jGRASP may not run with all display modes (color depths) on all systems. It probably won't run on Blackdown Linux with a 24-bit display (this is a problem with all Java apps) - haven't tried or investigated the latest Blackdown version yet.

Bad dialog positions - due to Swing bugs, on some systems dialogs may pop up in strange positions, possibly off-screen. Clicking the menu item that brought up the dialog again will usually move it on-screen.

 GNOME - If you are using GNOME (the default desktop environment on RedHat 6), you may not be able to run jGRASP with some JDKs because of freeze-up (this is a problem with all Java apps).

olvwm - on older versions of olvwm and possibly other window managers, the title bars may not show the title bar text (this is a problem with all Java apps). Install the latest olvwm to fix this (or bug your sysadmin into installing it).

Solaris JDK 1.2 problems (you should not use 1.2):

   Dialog layout is often messed up, with buttons off-screen or extra empty space.
   Find and replace does not see the find text most of the time. Sometimes hitting "Enter" will make it work. If you highlight the search text then pop up the dialog, that will work. Since the replace text may not be recognized, "Replace All" is very dangerous.

14 Future Features / Missing Features

Check the future plans page at [http://www.eng.auburn.edu/grasp/] for an updated list of planned future features, if you are not already there.

Short Term / Ongoing

   Navigation (for Java first).
   More features for the integrated Java debugger: show new and changed values in a different color; hex and octal display for values and array indices; a dynamically updated monitor display window, listing owning thread and waiting threads by monitor; automatic deadlock detection; byte code display; display of current watchpoints to make removing them easier; ability to set breakpoints without access to source code; repeated, animation-like stepping; counted stepping (eg: step in 42 steps); ability to change values while debugging; ability to disable/enable garbage collection for objects; assistance for memory leak (lost reference) detection; exception breakpoints; listing of all loaded classes with their fields and methods; show classpath, version, and other information for target VM; tracing variable changes; dumping of thread, stack, and variables to message window for easy cut-and-paste.
   A debugger connection and interface for languages other than Java.
Unicode support - this will require considerable work, so it will not be done soon.
Add close button to title bar of desktop CSD windows. It is not there now because of a Swing bug.
Switch to heavy menus. This is not done now because of Swing bugs on some systems.
Command line functionality for batch formatting and printing like UNIX GRASP has.
Ability to save and load named workspaces.
Improve the CSD TrueType font for better-looking native Windows printouts and RTF files.
Add more compiler setups.
Have a project realize that it has been moved, and copy previous file settings.
Add an editor for user-defined templates.

15 Reporting Bugs / Requesting Features

When jGRASP starts, you will get a warning if you are using a JVM on which jGRASP has known problems or on which we have not tested jGRASP. If you are using such a system, do not report bugs like freeze-up or layout bugs, which are probably caused by the virtual machine and system classes (you should get a message telling you this when jGRASP starts).

On Windows, check out the jGRASP exec shell. It will be minimized on the task bar. This window displays Java stack dumps from jGRASP.

On UNIX/Linux, run from the command line so you can capture any Java stack dumps.

If you are having a problem with a compile or run command, make sure the command works from the DOS command prompt before contacting us.

If you are having a problem with a compile or run command, clear the Compile Messages or Run I/O window, turn on "Verbose Messages" under the "Settings" menu, execute the compile or run command, and copy and send us ALL of the output. This will make it much easier for us to determine the nature of the problem.

Check the known bugs or future plans page at http://www.eng.auburn.edu/grasp/ before reporting a bug or requesting a feature. Email bug reports and feature requests to grasp@eng.auburn.edu. Be sure to include the version number, and for bug reports, the system you are using, and the stack dump if any.

16 About jGRASP

jGRASP™ Version 1.5.0
Copyright 1999-2002 Auburn University

Project Director:

James H. Cross II, Ph.D.
Computer Science and Software Engineering
107 Dunstan Hall
Auburn University, Alabama 36849
grasp@eng.auburn.edu
Chief Programmer:
Larry A. Barowski

Acknowledgments:

The development of jGRASP has been supported by a research grant from the National Science Foundation.

The development of previous versions of GRASP was supported, in part, by research grants from NASA Marshall Space Flight Center, the Department of Defense Advanced Research Projects Agency (ARPA), and the Defense Information Systems Agency (DISA).

Free code used in jGRASP (thanks to all who provided it):

The click-to-error regular expression matching, regular expression find-and-replace, and multi-file search use OROMatcher. See the ORO License.

The console connection mode for Windows NT is based on portions of Gordon Chaffee's very well structured and well documented port of "expect" to Windows NT. See the Expect License.

The Windows installation was created with Freeman Installer.

Most of the toolbar icons were created and copyrighted by Dean S. Jones.

16.1 jGRASP License

jGRASP™ License
Software License for jGRASP™ Version 1.5.0
Copyright 1999-2002 Auburn University

Section 1. License Grant

Auburn University grants to you a non-exclusive and non-transferable license to use jGRASP and the associated documentation provided in jgrasp/help, collectively "jGRASP." jGRASP may be installed for use on a single computer or on a local area network. The "wedge" source code provided in the jgrasp/src directory is free of license restrictions. It may be used or modified for any purpose. jGRASP is a Trademark of Auburn University.

Section 2. Restrictions

Distribution of jGRASP is not permitted without written permission (see Supplements), except that it may be distributed internally within a single organization. Distribution of components of jGRASP separately from the whole is not permitted, except that the complete associated documentation provided in jgrasp/help may be distributed separately. Reverse engineering of jGRASP is not permitted. Any use of image files, icons, or executable components of jGRASP separately from the whole is prohibited.
Section 3. Disclaimer of Warranty

jGRASP is licensed “as is.” There are no express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Auburn University makes no warranty with respect to the accuracy or completeness of information obtained through the use of this program. Auburn University does not warrant that jGRASP will meet all of your requirements or that its operation will be uninterrupted or error free or that any defect within jGRASP will be corrected. No oral or written information, representation, or advice given by Auburn University or an authorized representative of Auburn University shall create a warranty. Auburn University and its agents shall in no event be held liable to the user for any damages, including direct, indirect, incidental, or consequential damages, lost profits, lost savings, or other such damages arising out of the installation, use, improper use, or inability to use jGRASP, even if Auburn University has been advised of the possibility of such damages, or any claim by any other person or entity related thereto.

Supplements

Distribution for Educational Purposes - Publishers may distribute the jGRASP software and the jGRASP Handbook on CDs that accompany their textbooks provided that (1) the title “jGRASP™ 1.5.0 © 1999-2002 Auburn University” is included on each CD label, (2) descriptions of the CD indicate that jGRASP is included on the CD, and (3) a list of the textbooks that include jGRASP is provided to Auburn University (cross@eng.auburn.edu). Permission to distribute jGRASP for educational purposes covers all CDs created prior to May 31, 2003 for inclusion in textbooks. While it is anticipated that distribution of jGRASP for educational purposes will remain royalty free, this supplement of the jGRASP license will be re-evaluated on an annual basis.

For additional information, contact James H. Cross II, Computer Science and Software Engineering, 107 Dunstan Hall, Auburn University, AL 36849 (334-844-6315, cross@eng.auburn.edu).

16.2 ORO License

/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation. All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in
the documentation and/or other materials provided with the
distribution.

3. The end-user documentation included with the redistribution,
if any, must include the following acknowledgment:
   "This product includes software developed by the
   Apache Software Foundation (http://www.apache.org/)."
   Alternately, this acknowledgment may appear in the software itself,
   if and wherever such third-party acknowledgments normally appear.

4. The names "Apache" and "Apache Software Foundation", "Jakarta-Oro"
   must not be used to endorse or promote products derived from this
   software without prior written permission. For written
   permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache"
   or "Jakarta-Oro", nor may "Apache" or "Jakarta-Oro" appear in their
   name, without prior written permission of the Apache Software
   Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
16.3 Expect License

Copyright (c) 1997 Mitel Corporation

Permission is hereby granted without license or royalty fees to use, copy, modify, and distribute this software and its documentation for any purposes, provided that the copyright notice set forth above and in the following two paragraphs appear in all copies of this software.

IN NO EVENT SHALL MITEL CORPORATION BE LIABLE TO ANY PART FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF MITEL CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

MITEL CORPORATION SPECIFICALLY DISCLAIMS ANY CONDITIONS, REPRESENTATIONS OR WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND MITEL CORPORATION HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

17 Control Panel

This is the top-level window that contains top-level menus, the message window, and if the virtual desktop is on, the browse/project window and the virtual desktop. The empty space on the right of the menu bar and the desktop support file dropping (drag and drop).

- File Menu
- Project Menu
- Messages Menu
- Settings Menu
17.1 File Menu

New File - opens an empty CSD window for the selected language. Recently opened and currently open files will appear under the main menu, other files will be under the "Other" submenu.

Open File - brings up an open file dialog.

Recent Files - allows recently closed files to be re-opened.

Search Multiple Files - brings up the multi-file search dialog.

Save All Files - saves files in all CSD windows.

Close All Files - closes all CSD Windows. The user will be warned and given a chance to cancel if modified files exist.

Sync Files - checks file modification times for all files in jGRASP with those on disk. For each file that has been modified outside of jGRASP, the user is given the option of reloading the file from disk. If the user chooses not to reload, the next check will report a modified file only if there is an additional modification. A save operation, however, will give an external modification warning even if there is not an additional modification.

Save Current Workspace - saves all the current top-level settings: the desktop, global settings, and current project. This happens at exit anyway, so this is provided only for protection against a crash.

Exit jGRASP - quits the application. If a compile process is active and there are modified files, this will not be allowed.

17.2 Project Menu

Note that any operation that closes a project will save it first. It is not possible to discard changes to a project (unless jGRASP crashes or is killed).

New Project - allows the user to create a new empty project.

Open Project - allows a saved project to be opened.

Close Project - closes the current project and opens the default project.

Save Project - save modifications to the project immediately. This is never necessary, except to be safe against a crash.

Save Project As - allows the user to save the current project under a new name.

Recent Projects - allows the user to open recently closed projects.
Add Files To Project - opens a dialog that allows the user to add files to the project. This dialog allow files to be added with absolute paths or paths relative to the project file location (if the project is named and the file(s) are on the same file system (drive) as the project file). The relative paths are also platform independent.

Remove Selected Files From Project - remove any files selected in the project window.

Create Jar File for Project – brings up a jar creation dialog for the current project.

Jar/Zip Extractor – brings up a jar/zip extractor dialog.

Generate Documentation – brings up a dialog that allows the user to generate project documentation for Java.

Show Documentation – displays the documentation for the current project, if any. Currently, only Java documentation is supported.

CSD Window Settings For Project - brings up a CSD Window settings dialog for the current project and selects the "CSD" tab.

Compiler Settings For Project - brings up a csd window settings dialog for the current project, and selects the "Languages" tab.

Print Settings For Project - brings up a print settings dialog for the current project.

17.3 Messages Menu

Messages menu commands apply to the currently selected message window (Compile Messages, jGRASP Messages, or Run I/O).

Save As Text File - allows the user to save the contents of the current message window as a text file, with a choice of line terminators (binary, DOS format, Mac. format).

Select All And Copy - selects and copies the entire contents of the current message window to the clipboard.

Clear - clears the current message window.

Print Contents

Native Print - allows the user to print current message window contents using the native system. This is typically much faster than JVM printing. For UNIX/Linux, the native print is just a PostScript print using the command set in the global print settings. The font setting is ignored in this case, and Courier font is used.

Print to File (PostScript) - prints current message window contents to a PostScript file. The font setting is ignored and a Courier font is used.

Print to File (RTF) - prints current message window contents to a Rich Text file. For now, no formatting is done. The only print settings that are used are font, font size, and color.

JVM Print - prints message window contents using Java. This tends to be very slow, and produces very large print files.

Print Settings - opens a print settings dialog for message window, or global or project level print settings, and selects the "Messages" tab.

17.4 Settings Menu

Virtual Desktop - turns the virtual desktop on and off. When the desktop is turned on, all CSD windows are moved onto the desktop.
Open CSD Windows Maximized - if on, CSD Windows opened in the virtual desktop will be initially maximized.

Focus Follows Mouse - when on, you can set keyboard focus to CSD windows and most other text-entry windows in the virtual desktop by moving the mouse over them. Unfortunately, using the mouse in a CSD Window on the virtual desktop will still raise it to the top, but you can type and use keyboard accelerators in a non-top-level window.

CSD Window Popup Menu Enabled - when on, there will be a popup editing menu for CSD Windows. The particular mouse event that triggers the popup is determined by the JVM and Look-And-Feel. On Windows, Solaris, and Linux JVMs, the event is generally any right mouse click. Turning this off allows you to use the CSD Window right click function that selects tokens with one click, and levels of enclosing braces with more clicks.

Auto Save - if on, modified files are saved before a compile operation. Otherwise, modified files are temporarily saved before a compile operation.

Auto Sync - if on, check file modification times every time the control panel is activated. You almost certainly do not want this on if you are in a focus-follows-mouse environment (if you are using Windows, you are not in a focus-follows-mouse environment). See Sync Files.

Verbose Messages - if on, compile, run, and CSD generation functions will give detailed messages about their operation.

CSD Window Settings - opens a CSD_Window_Settings dialog for the global or project settings, and selects the "CSD" tab.

Compiler Settings - opens a CSD settings dialog or the global or project settings, and selects the "Languages" tab.

Print Settings - opens a print settings dialog for the global or project print settings.

Administrator CSD Window Settings (Administrator Only) - opens a CSD settings dialog for the administrator settings, and selects the "CSD" tab.

Administrator Compiler Settings (Administrator Only) - opens a CSD settings dialog for the administrator settings, and selects the "Languages" tab.

Administrator Print Settings (Administrator Only) - opens a print settings dialog for the administrator print settings.

Look And Feel - allows the look and feel to be changed. This may alter some gui element positions slightly. The size and alignment of some interface elements may be slightly off due to Swing bugs.

17.5 Window Menu

Refresh Window - causes the main window to be repainted.

Cascade - lays out the windows in the virtual desktop in a cascade.

Cascade and Resize - lays out the windows in the virtual desktop in a cascade and sets them to their default sizes.

The rest of the items represent currently open CSD windows, and allow the user to pop them up. An asterisk following a file name indicates that it is modified.

17.6 Tools Menu

Regular Expression Tester - pops up a testing tool for Perl5 regular expressions, which are used in compiler error formats.

Autotest - Brings up the autotest dialog.
System Info – Displays information about the operating system and Java version under which jGRASP is running.

Run Garbage Collector - frees unused memory (in jGRASP itself) immediately.

17.7 UML Menu

Show UML Window - Brings up the UML window.
Generate/Update UML - Brings up the UML window and updates the UML diagram.

17.8 Open File Dialog

A dialog for opening files.

Filters for the languages supported by jGRASP are provided, as well as the "all files" filter. Note that language filters classify files based on the extension only if they have not been opened in jGRASP before. Once a file is opened, jGRASP remembers that language. The default language for files with a ".h" extension can be set to C++ or C only (not both) by changing the extension settings for C or C++ (see settings).

You can also type a list of extensions into the "Filter Extensions" field to filter by extensions. These must be separated by whitespace, and can start with ".", ".*", or nothing. For example: "*.c *.cpp", "c .cpp", and "c cpp" will all show only files that end in ".c" or ".cpp".

The language may be forced at load time using the language pulldown menu. This only applies to files that have not been previously opened in jGRASP. Once a file is opened, its language is remembered.

The Text / Binary radio buttons allow the file to be opened in text mode (UNIX, DOS, and Mac. line terminators accepted) or binary mode (only newlines are line terminators).

17.9 Autotest Dialog

This dialog will self-test jGRASP CSD generation against a batch of selected files. You can help us test CSD generation by running autotest on your code. If autotest indicates an error, load the file that caused the problem into jGRASP (probably by clicking on the error message) and generate a CSD. Note if a CSD will not generate, or if there is an obvious error in the CSD diagram. If you believe your source code is valid (and there is not a diagram error caused by a macro or include file in C or C++ code), you can send us the code that shows the problem (or a mocked-up piece of code, if that is not possible). The chances are good that we will fix the problem quickly.

Note that autotest may take some time if you apply it to a lot of code.

Here are the tests performed on each file, and for various combinations of CSD properties (boxes on/off, forced newlines on/off, etc.):

A CSD is generated, which tests the validity of the parser (assuming the source code is valid).

The CSD is compared to the original code to make sure the code was not altered (this test is done every time a CSD is generated).

The CSD diagram is tested against a set of rules for a well-constructed CSD. C and C++ code that uses macros or has partial structures in include files may fail this test, and this may not indicate a problem.

A second CSD is generated from the first to test for stability. That is, to make sure CSD(CSD(source)) = CSD(source).
17.10 Browse Window

This window acts as a file browser.

A combo box shows the current directory. You can type a new directory in directly and hit "Enter" to change to that directory. The "Save Current Directory" item will store the current directory in the combo list. "Clear" will clear the list.

Filters for the languages supported by jGRASP are provided, as well as the "all files" filter. Note that language filters classify files based on the extension only if they have not been opened in jGRASP before. Once a file is opened, jGRASP remembers that language. The default language for files with a ".h" extension can be set to C++ or C only (not both) by changing the extension settings for C or C++ (see settings).

You can also type a list of extensions into the filter pulldown and hit "Enter" to filter by extensions. These must be separated by whitespace, and can start with ".", "*.", or nothing. For example: "*.c *.cpp", ",.cpp", and "c cpp" will all show only files that end in ",.c" or ",.cpp".

Clicking on a file will open it (in text mode).

Clicking on a project will attempt to open it. If it does not appear to be a jGRASP project, it will be opened as a file.

A popup menu allows selected files to be opened (in text mode) or added to the current project with either relative (to the project file location) or absolute paths.

17.11 Project Window

This window displays files in the currently open project. For now, only one project can be open at a time, in the future this will probably change.

Clicking on a file will open it (in text mode).

A popup menu allows selected files to be opened or removed from the project.

17.12 Message Window

The message window contains tabbed pages for compiler messages, messages from jGRASP (CSD generation, non critical errors and warnings), and an i/o window for the run function.

Color settings for these pages use CSD color settings. Some of these are specific to the message pages. They can be set on the CSD Window settings CSD colors page.

Clicking on an error in the compile and jGRASP message pages will open the file referred to in the message (if not already open) and select the line referred to in the message. If line numbers are frozen in the CSD window, the frozen line number will be selected. The format for click-to-error in the compile message page is determined by the compiler settings. The paths that will be searched if the line contains a relative path name or Java class name are specified on the sources page of the CSD Window settings (global and project only).

The compile and jGRASP message pages have a "Copy" button. This will copy the selected text, or all text if there is no selection, to the clipboard.

The compile page has a "Stop" button that will kill the compile process. If this button is enabled it means a compile process is active.

The run page provides i/o for the run function. A "Kill" button allows the running process to be killed. If this button is enabled it means a "run" process is active.

All three pages have a "Clear" button that clears previous output.
17.13 Projects

A jGRASP project consists of a file list, and CSD and print settings (see settings), and UML diagram state. For now there are very few operations on projects; they mainly provide compiler environments through the CSD settings, and a set of files for UML generation.

There is always an active project; if no project has been loaded, a default project is active. The current project is displayed at the top of the control panel.

Any operation that will close a project, such as quitting jGRASP or opening another project, will first save the current project. There is no way to discard changes to a project, except by killing jGRASP.

The project window displays the current project and the files it contains.

The project menu allows projects to be opened, saved, and modified.

17.14 Multi-file Search Dialog

This dialog allows multiple files to be searched. The results are reported to the jGRASP Message window, in a clickable format. Row and column numbers in the results assume the file will be opened in text mode (as will happen if you click on a result and the file is not already open). If the file is opened in binary mode, the column number will be off by one for any line for which the previous line is terminated by "\n\n".

Filters for the languages supported by jGRASP are provided, as well as the "all files" filter. Note that language filters classify files based on the extension only if they have not been opened in jGRASP before. Once a file is opened, jGRASP remembers that language. The default language for files with a ".h" extension can be set to C++ or C only (not both) by changing the extension settings for C or C++ (see settings).

You can also type a list of extensions into the "Filter Extensions" field to filter by extensions. These must be separated by whitespace, and can start with ".", "*.", or nothing. For example: ".c *.cpp", ".c .cpp", and "c cpp" will all show only files that end in ".c" or ".cpp".

Files and directories can be selected. For each directory selected, any files in that directory matching the selected filter will be searched. If no files are selected and "Search Project Files" is off, the current directory will be searched. To select no files, you may need to enter a non-existent filename for "File name".

Search Pattern - the text or pattern for which to search.

Pattern Type

Plain Text - match the search text. A search will begin at the end of the previous match.

Awk/Grep Reg. Exp - match extended awk/grep style regular expression. A search will begin at the end of the previous match.

Perl5 Reg. Exp. - match a Perl 5 regular expression. A search will begin at the end of the previous match. If the pattern contains no groups, the whole match will be reported. Otherwise, there will be a match for each matching group. For example, the pattern t(e)s(t) will result in two matches for "test", one for the "e" and one for the second "t". Zero-length groups are allowed, but the selection will be one character long. For example, t(S*)est will select the "e" in "test". See the perlr man page for a description of the pattern format.

Case Sensitive - if on, the search will be case sensitive.

Search Subdirectories - if on, subdirectories of any directories in the search will be recursively scanned. Any files in the matching the selected filter will be searched.

Search Project Files - if on, all files and directories in the current project will be searched.
Max. Results Per File - specifies the maximum number of matches that will be reported for each selected file.

**17.15 Help Us Test CSD Generation**
You can help us test the jGRASP CSD generator by running the autotest on groups of your source files. The autotest dialog is on the "Tools" menu of the control panel. See [autotest dialog](#) for a description of its use.

**17.16 Generate Documentation Dialog**
Target Directory - the directory in which the "javadoc" documentation will be generated. Using a project-relative path will allow this setting to be valid if the entire project is moved (even to a different operating system).
Additional Doc. Command Flags - additional flags to the "javadoc" command. This is useful for linking to Java API docs on the local system or the web. For example:
- local file: `-link "file:///C:/j2sdk1.4.0/docs/api"
- Sun website: `-link "http://java.sun.com/j2se/1.4/docs/api"
Generate - generates documentation using the current project-level "Document" command for Java.

**17.17 Jar Creation Dialog**
This dialog allows the user to create a jar file for the files in the current project.
- **First Page**
- **Second Page**

**17.17.1 First Page**
Project Files - selects the type of files from the current project to include in the jar file. The files are shown in the window below.
Additional (non-project) Files - the "Add" button allows the user to add arbitrary files to the jar file. Using a relative path (the default) means that the correct files will be used if the entire project is moved to another directory. The "Delete" button will delete selected files from the window.

**17.17.2 Second Page**
Jar File - the name of the jar file to be created. Uses a project-relative path by default.
Jar Root Directory - the root of the jar directory. Filenames in the jar file will be relative to this directory. This is initially set to the first common ancestor of all the files and package roots in the jar, and by default it is project-relative. Any ancestor directory of this default will work. To reset the default value, delete the current value, hit "Back", then hit "Next".
Main Class - the name of the main class in the jar file. If specified, running "java -jar" on (or possibly clicking on) the generated jar file will run that class. The combo-box gives a choice of classes with entry (main) methods.
17.18 Jar/Zip Extractor Dialog

This dialog allows the user to extract files from a jar or zip archive. Once an archive is opened, all files or selected files can be extracted to a directory using "Extract Files". Files can also be dragged from the window and dropped elsewhere.

18 CSD Window

"CSD window" is the name given to source code editing windows in jGRASP. The CSD Window supports dropping of text (drag and drop). The title bar displays the language, file name, and modified state (it shows "(edited)" if modified). Note that CSD generation is not considered a modification, because even though it may change the text it will not change the code structure.

The CSD window may be split horizontally or vertically for a dual view of the text. This can be done by using the "Split View" submenu of the "View" menu. The split state of the window is remembered when a file is closed and reopened.

A tool bar is provided for commonly used functions. The default position of this bar is above the text, but it may be floated or moved to the bottom (or the sides, which is not very useful). It may also be removed using the "View" menu.

A message bar displays the insert/overstrike mode state, and the current cursor line and column, character code at the cursor, and topmost visible line. If the view is split, the topmost visible line of both views is displayed. The default position of this bar is below the text, but it may be floated or moved to the top (or the sides, which is not very useful). It may also be removed using the "View" menu.

CSD windows maintain an internal "primary selection" as is found on XWindow systems. Selecting text in one window will unselect it in another. The F9 key will search for the current selection, and Shift-F9 will search backward for the current selection, in a CSD Window. The Find key and Shift-Find will do the same thing (if your keyboard has a Find key).
18.1 File Menu

Clear - closes the currently open file and clears the text.

Open - brings up an open file dialog.

Save - saves the current file. By default it is save with the same line terminator format (binary, DOS, or Mac.) as it had when loaded. If the format could not be determined when loaded (because there was more than one type of line terminator), it will be saved in binary mode.

Save As - allows the user to save the current file under a new name, and with a choice of line terminators (binary, DOS format, Mac. format).

Add to Project - if the file is not in the current project, add it to the current project with either a relative or absolute path. Relative paths are relative to the project file location, and are platform independent. They can not be used for the default project (you must name the project using, for example, "Project" / "Save As").

Generate UML (Java only) - pops up the UML diagram and updates the information for the current file. See UML window for information on jGRASP UML.

Generate Documentation - if the current compiler environment has a "document" command, runs that command and displays the documentation produced. Currently only the Java compiler environments have "document" commands.

Print

Native Print - allows the user to print using the native system. This is typically much faster than JVM printing. For UNIX/Linux, the native print is just a PostScript print using the command set in the global print settings. The font setting is ignored in this case, and Courier font is used.

Print to File (PostScript) - prints to a PostScript file. The font setting is ignored and a Courier font is used.

Print to File (RTF) - prints to a Rich Text file. For now, no formatting is done. The only print settings that are used are font, font size, and color.

JVM Print - print using Java. This tends to be very slow, and produces very large print files.

Print Settings - opens a print settings dialog for this file, or global or project level print settings, and selects the "CSD" tab.

Escape Virtual Desktop / Enter Virtual Desktop - moves this window to/from a desktop frame to a free-floating frame.

Exit This Window - closes the CSD window.

18.2 Edit Menu

Undo - this will undo the last edit. Folding is considered an edit in this context even though it does not modify the text, because folds must be undone to properly undo a sequence of edits. There is a limit to the total size of edits stored, but the last edit will always be stored no matter how large it is, and for multiple edits like a replace all, they will all be stored.

Cut - copies selected text to the clipboard, then deletes it.

Copy - copies selected text to the clipboard.

Paste - paste text from the clipboard.

Block Cut/Paste - turn block selection mode on or off. In block mode cut, copy, paste, find, etc. work on rectangular blocks of text. If the end of a line is selected, the block will extend infinitely to the right (the
ends of all lines in the block will be included). A cut will never delete lines, and a paste will not add lines unless the text is too short, in which case lines may be added to the end of the text.

Find/Replace - brings up the find dialog.

Find Selection – search forward or backward for the current selection. The system PRIMARY selection is used on UNIX and Linux.

Search Multiple Files - brings up the (global) multi-file search dialog.

Comment - for languages that have single-line comments, comments each line of the selected text.

Uncomment - for languages that have single-line comment, removes the first such comment on each line of the selected text.

CSD Window Settings - opens a CSD settings dialog for this file, or global or project level settings, and selects the "CSD" tab.

Compiler Settings - opens a CSD settings dialog for this file, or global or project level settings, and selects the "Languages" tab.

18.3 View Menu

The settings in this menu other than split view - tool bar, tool bar button type, message bar, line numbers - can be set globally in the CSD view page of the global CSD Window settings dialog. The values set in the menu will persist only until the window is closed or a file is loaded into the window.

Generate CSD - generates a CSD.

Remove CSD - removes the CSD.

Fold - see folding.

Marks - see marks.

Bookmark - see bookmarks.

Breakpoints - see breakpoints.

Split View - allows the user to select from a single view CSD window, or a vertically or horizontally split dual view.

Tool Bar - if off, the tool bar is not shown.

Tool Bar Buttons - allows the tool bar buttons to be shown as icons, text, or both.

Message Bar - if off, the message bar is not shown.

Line Numbers - if on, line numbers are shown to the left of the text.

Freeze Numbers - when turned on, further edits will not change the line numbers. Also, click-to-error from a message window will use these frozen numbers. This is useful for finding multiple compile errors while editing the text.

18.4 Compiler Menu

Compiler Settings - opens a CSD settings dialog for this file, or global or project level settings, and selects the "Languages" tab.

This menu provides compiler functions set up in the CSD window settings dialog. Output appears in the message window, which also provides a button to stop the compile process. Due to Java bugs, on some systems the end of the compile process may not be recognized every time, and the user will have to use this button to return jGRASP to the non-compiling state.
The "Debug Mode" menu checkbox switches compiler debug mode on and off. This may change the compile commands, depending on the compiler environment that is being used. This is a global setting (applies to all CSD windows).

Before a compile, if auto save is on, all modified files are saved (this setting is on the control panel "Settings" menu).

If auto save is off, any modified files are backed up and the current text is saved. The modification times for modified files are changed only if the file has been modified since the last compile, so files will not be unnecessary recompiled. After the compile, the files are restored from backup. Because of this, while compiling with modified files, quitting jGRASP is not allowed and you should not kill jGRASP.

Actions that would conflict with compiling, such as saving the file or activating another compile process, are blocked during a compile.

### 18.5 Run Menu

This menu provides run functions set up in the CSD window settings dialog. Input and output are provided by the message window, which also contains a button to kill the running process.

The "Run Arguments" checkbox switches an arguments toolbar on and off. When on, arguments entered in the toolbar will REPLACE those in the settings.

If the "Run in MSDOS Window" checkbox (Windows only) is on, the "Run", "Run as Applet", and "Debug" will use a DOS shell for I/O. The user will be prompted in the DOS shell to hit a key to start the process and to continue after the process is finished. In Windows NT, you can right click on the title bar and change the window properties, including width and height, before starting. In Windows 95/98 you can change the properties of the conagent.exe file (standard location is C:\Windows\System\conagent.exe) to change the default number of screen lines for DOS shells. In Windows NT, you can do the same thing using cmd.exe (standard location C:\Windows\System32\cmd.exe).

During a run, actions that would cause a conflict, such as activating another run or a compile process, are blocked.

### 18.6 CPG Menu

Click here for information about the CPG and CPG window.

Generate CPG - generates a CPG for the current file.

Find CPG Window - pops up the CPG window associated with the current file.

### 18.7 Templates Menu

A menu that allows predefined templates to be inserted into the text. These are inserted at the current cursor position, and indented to match the current cursor position.

The user may define templates in the template directory of the .grasp_settings directory in the users home directory (typically C:\Windows on Windows systems). For now, there is no easy way to do this. A template editor will be added later. The template format can be seen by examining the system templates in the data directory of the jGRASP distribution. The template files are named templ.ext, where ext is an extension that indicates to which language the template applies.
18.8 Find Dialog / Page

This is the dialog or page for find and replace in CSD windows. All CSD Windows in the virtual desktop share a common find page, the actions of which apply to the currently selected CSD Window in the desktop. Each CSD Window that is outside the desktop has its own find dialog.

Find - the search text.

Replace With - the replacement text.

Case Sensitive - if off, the search will ignore case.

Match

Plain Text - match the search text. A search will begin one character past the start of the previous match.

Awk/Grep Reg. Exp - match extended awk/grep style regular expression. When on, "Find Backward" is not available. This mode is not available when Block Cut and Paste is on. Zero-length matches are ignored. A search will begin at the end of the previous match.

Perl5 Reg. Exp. - match a Perl 5 regular expression. When on, "Find Backward" is not available. This mode is not available when Block Cut and Paste is on. Zero-length matches are ignored. A search will begin at the end of the previous match. If a search pattern contains no groups, the entire match will be selected. If it contains groups, at least one must be matched. The first match will be selected. For example, t(e)s(t) will select the "e" in "test". Zero-length groups are allowed. For example, t(\S*)est will move the cursor to the position before the "e" in "test" (it will select "nothing" before the "e"). See the perlre man page for a description of the pattern format.

Search

All Text - search all of the text.

Tokens Only - search only for full tokens. For example, "test" will match only the identifier "test", not "test1" or an occurrence of "test" in a string or comment, and "12" will match the number "12", but not "127".

Strings - search only within strings.

Comments - search only within comments.

Folds

Skip - if on, folded areas will be skipped.

Stop - if on and the text is found in a folded area, the cursor will stop at the fold.

Unfold - if on and the text is found in a folded area, the minimum amount of unfolding that will reveal the text will be done.

Find - searches forward in the text.

Find Backward - searches backward in the text.

Replace - replaces the current selection with the replacement text.

Replace All - replaces all occurrences of the search text with the replacement text. If the search or replace text contains a newline (spans multiple lines), or if regular expression mode is on, all folds will be removed.

Replace All in Selection - replaces all occurrences of the search text that occur entirely within the current selection with the replacement text. If the search or replace text contains a newline (spans multiple lines), all folds within the selection will be removed.

Replace Then Find - same as a "Replace" followed by a "Find".

Find and Mark All - applies marks to all occurrences of the search text.
Clear All Marks - removes all marks.

18.9 Open File Dialog

A dialog for opening files.

Filters for the languages supported by jGRASP are provided, as well as the "all files" filter. Note that language filters classify files based on the extension only if they have not been opened in jGRASP before. Once a file is opened, jGRASP remembers that language. The default language for files with a ".h" extension can be set to C++ or C only (not both) by changing the extension settings for C or C++ (see settings).

You can also type a list of extensions into the "Filter Extensions" field to filter by extensions. These must be separated by whitespace, and can start with ".", "*.", or nothing. For example: "*.c *.cpp", "c .cpp", and "c cpp" will all show only files that end in ".c" or ".cpp".

The Text / Binary radio buttons allow the file to be opened in text mode (UNIX, DOS, and Mac. line terminators accepted) or binary mode (only newlines are line terminators).

18.10 Mouse Button Actions

Left Mouse Button

Shift Key Down - same as middle mouse button.
Ctrl Key Down - same as right mouse button with CSD Window popup disabled.
Single Click - positions cursor. If at left of window, selects a line.
Double Click - if on text, selects a whitespace delimited word or whitespace. If on CSD, folds the CSD structure recursively. If on a folded CSD line, unfolds that line.
Triple Click - selects a line.
Quadruple Click - selects the whole text.

Middle Mouse Button - Extends the current selection.

Right Mouse Button - if CSD Window popup is enabled, brings up the popup menu, otherwise:

Single Click - selects a single language-specific token or whitespace block.
More Clicks - selects layers of matching parenthesis and braces.

18.11 Folding

The jGRASP CSD window allows text folding based on CSD structure and arbitrary folding. This folding is line-based: two or more lines are folded into one. For CSD structures, the first line remains unfolded. The second line (first folded line) displays the CSD folding symbol (a box with a plus inside) and if the second line contained only a comment, that comment. The fold commands are available on the View / Fold menu, through accelerator keys, and through mouse actions on the CSD. If the menu or accelerator key is used, folding is applied to the current text cursor position. For mouse clicking, it is applied to the position clicked on.

When text is edited after a CSD is generated, the foldable structure is maintained as much as possible. An edited CSD may fold in unexpected ways - regenerating the CSD will fix this.

The CSD structure containing the cursor is considered to be the innermost structure if the cursor is in the text, or the nearest CSD stem if the cursor is in the CSD. Note that each statement and declaration is a
structure, even if it is on a single line. Activating a fold command with the cursor on the text of such a line will not fold the enclosing structure.

Fold operations are a part of the stream of edits, and can be undone using Ctrl-Z or the "Undo" item of the "Edit" menu. "Unfold All" however, can not be undone.

The fold operations are:

Fold (Shift-F3) - If multiple lines of text are selected, the first to last lines of the selection become a single fold. Otherwise, the CSD structure containing the cursor is folded. If the cursor is at the top line of a multi-part structure, such as a switch statement or if-else statement, each part of that statement is folded.

Fold Recursive (F3) - If multiple lines of text are selected, any CSD structures within the selection are folded recursively. If there are no CSD structures in the selection, nothing is folded. Structures partially within the selection are partially folded. If no text or part of one line of text is selected, the CSD structure containing the cursor is folded recursively. If the cursor is at the top of a multi-part structure, such as a switch statement or if-else statement, each part of that statement is folded recursively.

Fold All Recursive (Ctrl-F3) - Does the same thing as selecting all text then folding recursively.

Unfold (F4) - unfolds the next fold following the cursor position. For a mouse click, the click must be on the folded line.

Unfold Recursive (Shift F4) - unfolds the next fold following the cursor position, and any folds within that fold.

Unfold All (Ctrl-F4) - removes all folds.

18.12 Marks

The CSD window supports a multiple highlights called marks. Marks are separate from the selected text. They are used only for display, not for cut and copy. These highlights are transitory: they will not survive CSD generation or be saved when a file is closed.

A menu is provided on the CSD window "View" menu that allows the current selection to be marked, all marks to be removed, and the text to be searched for the next or previous mark. The find dialog also has a "find all" function that uses marks.

18.13 Bookmarks

The CSD window provides line-oriented bookmarks. These are displayed to the left of the text, as green trapezoids. They are persistent, and will be saved and restored when a file is closed and reopened.

A menu is provided on the CSD window "View" menu that allows the bookmark on the line containing the cursor to be toggled on or off, all bookmarks to be removed, and the text to be searched for the next or previous bookmark.

18.14 Breakpoints

Breakpoints are used in the integrated Java debugger. These are displayed to the left of the text, as red octagons. They are persistent, and will be saved and restored when a file is closed and reopened. If you are not using the integrated Java debugger, you can use breakpoints as additional bookmarks.

A menu is provided on the CSD window "View" menu that allows the breakpoint on the line containing the cursor to be toggled on or off, all breakpoints to be removed, and the text to be searched for the next or previous breakpoint.
18.15 GNU Extensions to C and C++

Most of the GNU C and C++ extensions are supported for CSD generation. Additional extensions are supported if the "Allow GNU Extensions" box is checked in the compiler settings.

Function attributes for C are not supported unless "Allow GNU Extensions" is checked because the syntax is too close to that of a K&R style function definition. You can get around this by defining __attribute__ to nothing in the predefined macros.

Nested functions are not supported. These can not be distinguished from declarations without identifier context. Our C parser does not distinguish variables from type names, so that header parsing can be avoided.

Breaking strings across lines without using backslashes will not be supported. CSD generation adds text at the beginning of lines, so the original string might not be recovered if there were a syntax error that caused the start of the string not be recognized. The backslash at the end of a line applies regardless of syntactic context, so there is no danger of corrupting backslash-continued strings during CSD generation.

18.16 Extended Awk Regular Expressions

From the ORO documentation:

This is the traditional Awk syntax that is supported:

Alternatives separated by |
Quantified atoms

+   Match 0 or more times.
?   Match 0 or more times.
\^  Match 0 or 1 times.
Atoms

regular expression within parentheses
a . matches everything including newline
\^ is a null token matching the beginning of a string but has no relation to newlines (and is only valid at the beginning of a regex; this differs from traditional awk for the sake of efficiency in Java).
\$ is a null token matching the end of a string but has no relation to newlines (and is only valid at the end of a regex; this differs from traditional awk for the sake of efficiency in Java).

Character classes (e.g., [abcd]) and ranges (e.g. [a-z])
Special backslashed characters work within a character class
Special backslashed characters
\b backspace
\n newline
\r carriage return
\t tab
\f formfeed
\xnn \nn or \nnn hexadecimal representation of character
\nn or \nn octal representation of character
Any other backslashed character matches itself

This is the extended syntax that is supported:

- Quantified atoms
  - \{n,m\} Match at least n but not more than m times.
  - \{n\} Match exactly n times.

- Atoms
  - Special backslashed characters
    - \d digit [0-9]
    - \D non-digit [^0-9]
    - \w word character [0-9a-z_A-Z]
    - \W a non-word character [^0-9a-z_A-Z]
    - \s a whitespace character [ \t\n\r\f]
    - \S a non-whitespace character [^ \t\n\r\f]
    - \cD matches the corresponding control character
    - \0 matches null character

19 Integrated Java Debugger

jGRASP has an integrated debugger for Java. When the debug command is run, if it responds with a transport address (a line like "Listening for transport X at address: Y."), jGRASP will start the debugger. The default "jdk (integrated debugger) - generic" compiler environment will debug an application or applet with Sun's JDK 1.3. JDK 1.2 with the JPDA for 1.2 might also work, but will probably be quite buggy.

To use the integrated debugger, jGRASP must be running under the JDK (not a JRE). If you are having some problem with this (you get a "tools.jar file not on classpath" message when attempting to debug), putting the JDK bin directory at the beginning of the PATH should fix this. For a detailed description of how the jGRASP startup programs find/choose java, see Running jGRASP.

Also, you must be running (debugging) your programs under the JDK. The default Java compiler environment uses the classic VM, which is not included in the JRE (you could try copying the compiler environment and modifying it to use HotSpot, but we don't recommend it, as the HotSpot/debugger combination is slow and buggy). If you have the JDK bin directory at the beginning of your PATH and use the default Java compiler environment in jGRASP, there should be no problem. If you can't set things up that way, you will need to copy, modify (add to PATH environment variable), and use a Java compiler environment - see Settings and Compiler Environment Dialog.

Do not use the integrated debugger on a dual-processor Windows NT system, as this will most likely lock up jGRASP and the target process, frequently if not always.

Using the integrated debugger may lock up jGRASP occasionally. If this happens consistently (every time or every time for some particular target program), or if there are stack dumps from jGRASP, let us know.
Many additional features are planned for the debugger. See Future Plans (future/missing features) for details.

### 19.1 Debug Window
- Toolbar
- Threads Window
- Call Stack Window
- Variables Window
- Breakpoints
- Watches

### 19.2 Toolbar
Step Over - steps the selected thread within the current method or code block. The thread must be suspended to step. In normal step mode, this will execute approximately one line of source code in the current method. If "byte code size steps" is selected, approximately one byte code instruction will be executed. While the thread is stepping, "(stepping)" is displayed in its description in the threads window. Multiple threads can be stepped at the same time. Hitting "Suspend" will kill the step. When the step is completed, if the stepping thread or no thread is selected, the current source line will be selected; if another thread is selected, no action will be taken.

Step In - like step over, but will enter any method calls.

Step Out - like step over, but will step out of the current method or code block.

Step to Cursor - runs to the cursor for the active CSD window within the virtual desktop. All breakpoints are ignored during this operation, but watchpoints are not.

Suspend - suspends the selected thread, or all threads within the selected group.

Resume - start the selected thread, or all threads within the selected group.

Byte Code Steps - when on, use the smallest step size possible Otherwise, the step size is approximately one source line.

Suspend New Threads - if on, any new threads that start will immediately be suspended. If on when the debugging process is started, all startup threads are suspended as soon as is possible.

### 19.3 Threads Window
Shows the state of each thread, organized within the thread group tree. Suspended threads are shown in red, running threads in green. This window in updated dynamically, so you can see the state of running threads change. The thread description is as follows:

name <x, y> [state] (debugger action/state) monitors

name - name given to the thread. You should name any threads you start, to make them easier to track.

<x, y> - shows jGRASP's current suspend count for the thread (x), and the actual count.(y). This is for our debugging purposes only, it should not be of interest to the user unless jGRASP is not working correctly. The actual suspend count may be one higher than jGRASP's suspend count temporarily, but they should be the same most of the time. jGRASP's suspend count should always be 0 or 1.
[state] - shows the current suspend state: running, waiting, waiting on monitor, sleeping, not started, zombie, or unknown.

(debugger action/state) - shows what the debugger is currently doing with respect to the thread: at breakpoint, suspended, running to cursor, or stepping. If none of these are true (the thread is running normally), nothing is shown.

monitors - shows the monitors owned by the thread, by id number. This is the same id number shown for objects in the variables window. If the thread is waiting on a monitor, this monitor is shown in [braces].

For a simple, non-gui application, there will be your "main" thread, any threads that your code creates, and several system support threads (Signal Dispatcher, Reference Handler, Finalizer). For applets or gui applications, there will be numerous event queue, gui, and system threads.

19.4 Call Stack Window
Shows the current call stack for the selected thread, if the thread is suspended. When the thread is resumed, this window is disabled. Each frame of the stack shows the method, filename and line number if available, and program counter. The program counter is the current byte code index within the method. Selecting a frame will select the source line, if the corresponding file can be found, and show the associated variables in the variables window. Source files are located using the "Sources" entry of the global settings.

19.5 Variables Window
Shows the available variables for the current frame of the call stack. Composite objects can be expanded to see the fields or array elements. The target code must be compiled in debug mode, or not much information will be available here.

this or static - shows fields for the current object if within an instance method or code block, or static fields available within the current static context.

Arguments - shows arguments to the current method if within a method.

Locals - shows the current local variables.

Objects are shown as squares, primitive types as triangles. All non-fields are colored blue. Field icons are color coded based on the declared type of the object containing them (the actual type may be a subclass or interface implementation). The coding is:

- orange - field is declared in a superclass of the declared type of the containing object or an interface implemented by a superclass.
- green - field is declared in the declared type of the containing object.
- cyan - field is declared in an interface implemented by the declared type of the containing object.
- yellow - field is declared in a subclass of the declared type of the containing object or an interface implemented by a subclass.

Static fields are shown in italic type.

When a field is selected, the type in which it was declared is shown at the bottom of the variables window.

For arrays, at most ten elements are shown at once. To change the elements that are shown, select the object then click on it (but don't double click). A slider bar will appear, on which you can select the first element shown.

Any masked fields are shown with a grey bar over the icon.
Watches: Fields can be watched for access or modification if the target JVM supports it. To create a watchpoint, select a field and then right-click. A popup menu will allow you to set a modification or access watchpoint, or to remove them if they are set. A "Watch All" will set a watchpoint on all fields of the same signature as the selected field (including those in subclasses and superclasses of the field's object type). A simple "Watch" will set a watchpoint only for the selected field's object. Fields that are being watched for modification are shown with a black triangle pointing up in the center of the icon, those with an access watch are shown with the triangle pointing down, while those with both will have a black square. Setting a lot of watches can slow down debugging.

19.6 Breakpoints

Breakpoints can be set in the CSD windows using the "View" / "Breakpoints" menu, or the context menu. Only breakpoints in CSD Windows are active (when a CSD Window is closed, the breakpoints will go away, when one is opened, the breakpoints will become active. Breakpoints can be added or removed while the debugger is running.

While the debugger is running, invalid breakpoints (breakpoints set on lines that do not contain executable code) are shown crossed out. If the class files for a particular source file can not be found, the breakpoints are shown with a slash through them. In order for this mechanism to work, the source file and associated class files must be located in the same directory.

20 Complexity Profile Graph (CPG)

The Complexity Profile Graph, or CPG, is a statement-level complexity or comprehensibility metric. Each segment, which corresponds roughly to a program statement, is assigned a complexity that is a weighted sum of:

- Reachability - the number of conditions that must be evaluated to reach the segment from outside of the enclosing method, function, or other top-level statement block. This is the traditional definition of reachability.
- Content - the log of the number of significant tokens in the segment. Some punctuation, such as block-enclosing braces or a statement-ending semicolon, is not considered significant. A pair of parenthesis is counted as one token.
- Breadth - the number of statements, methods, etc. in the innermost block containing the segment.
- Inherent - value assigned based on the inherent complexity of the innermost enclosing structure. For example, segments in switch or case statements are assigned an inherent complexity of 3.0. This reflects the view that some structure types are inherently more complex than others.

The CPG window "Data Sets" menu allows any combination of these values and the total complexity to be viewed.

Currently, the CPG is implemented only for Java and Ada.

21 CPG Window

The main section of this window shows the complexity values for the currently selected data sets. If the source is unedited, selecting segments in the CPG window will select the corresponding text in the source CSD window, and vice versa. The selection in the CPG window is shown as a yellow background. If the source is unedited, a black rectangle shows the segments that are currently visible in the CSD Window.
Each data set value is scaled in the display, by the weight used in computing the total complexity, unless there is only one data set, in which case it is shown unscaled.

The label at the top of the CPG window shows the complexity values of the segment under the mouse cursor. If the weight of a particular data set is not 1.0, the value and weight are displayed.

The scale button allows the graph to be scaled, from two-pixel wide bars to screen-width bars.

If "Scroll Lock" is on, the CPG window and the CSD window from which it was generated will scroll as one. This option is not available if the source has been edited.

A scrollbar at the bottom of the window allows the CPG to be scrolled if it does not fit in the window.

- File Menu
- Data Sets Menu
- View Menu

21.1 File Menu

Save Data - allows CPG complexity values to be saved in a straightforward text format.

Escape From Virtual Desktop / Enter Virtual Desktop - takes the CPG window in and out of the virtual desktop.

Disconnect From Source - dissociates the CPG window from the source CSD window. A second CPG window can then be generated from the same source.

Exit This Window - closes the CPG window.

21.2 Data Sets Menu

This menu allows each data set to be turned off or on.

21.3 View Menu

Vertical Orientation - if on, the segments axis is vertical, and the complexity horizontal.

Show Key - if on, a key that shows the color for each data set is shown (if there is more than one active data set).

Show Axis Labels - if on, the segment and complexity axes are labeled. If there is only one active data set, the complexity axis is labeled with the name of that data set, otherwise it is labeled with "complexity".

Show Number Labels - if on, segment and complexity axes have value labels. The labels are evenly spaced based on a minimum inter-label separation.

22 UML Class Dependency Diagrams

jGRASP can produce a UML diagram for Java code. The diagram is generated for classes in the source, class, and jar files in the current project. These diagrams are generated directly from the class (or jar) files, so source files must first be compiled.
22.1 UML Window
This window displays the UML diagram.

- Print Menu
- Layout Menu
- Edit Menu
- View Menu

- Diagram Display
- Info Window
- Goto Window

22.2 Print Menu
Print – print the diagram. The scale of the printout is based on the scale of the image. 100 pixels of image will be 1 inch on the printout.

Print Preview – preview the printout. The preview should be a close approximation of the printed result, but there may be minor differences because of scaling.

Print Settings – opens a print settings dialog for global or project level settings, and selects the "UML" tab.

22.3 Layout Menu
Tree Down - lays out selected objects, dependents of selected objects, or all objects in a vertically oriented tree. Only inheritance links are considered.

Tree Right - lays out selected objects, dependents of selected objects, or all objects in a horizontally oriented tree. Only inheritance links are considered.

Spring - lays out selected objects or all objects using a spring embedding layout algorithm. All links are considered.

22.4 Edit Menu
Delete Selected Edge Bends - the tree layout algorithms will insert bends in inheritance edges. This will delete them.

Remove From Project - removes selected or unselected files from the project. That is, for each selected or unselected object in the diagram, the source file from which that object was generated will be removed from the project. Note that external objects do not have a corresponding source file.

Add To Project - add selected or unselected external object files to the project. All package root directories for files in the project will be searched, and if exactly one source file for a particular external object is found, that file will be added to the project.

Select - extend the current selection by selecting any objects related in a particular way to the currently selected objects. For example, "Recursive Relations" will select all objects directly or indirectly related in any way to the currently selected objects.

Settings - brings up the UML settings dialog.
22.5 View Menu

Visible Objects - controls the types of objects that are shown. For example, you can hide non-public objects, or external objects (objects that are not in the project).

Visible Dependencies - controls the types of dependencies that are shown. For example, you can limit the view to inheritance dependency edges only.

Hide - hide selected or unselected objects, or show all objects. This is only temporary; on the next update, all objects will be shown.

Info - controls various properties of the info display.

Legend - controls various properties of the legend display.

22.6 Diagram Display

This window displays the UML diagram. A legend shows the meanings of the various object colors and edge styles and colors.

A scroller window at the upper left allows the view to be scrolled quickly.

A scale control allows the magnification of the view to be changed. Due to uneven scaling of fonts, the spacing between objects may change slightly as the scale is changed.

Objects can be selected using the mouse. Left click on an object to select it. Left click and drag a selection rectangle in the window to select multiple objects. Hold down the shift key while selecting to add to the current selection. To select the reverse direction of bidirectional edge, click it again.

When an edge is selected, the dependencies it represents are shown in the info window. When an object that is a part of the project is selected, its members (fields and methods) are shown.

Objects can be moved by selecting then dragging them with the left mouse button.

You can move an edge bend by selecting the edge, then left clicking and dragging the bend. Hitting the delete key while dragging will remove the bend.

22.7 Info Window

This window displays the fields, constructors (actually <init> methods), and methods for the currently selected edge (dependency) or node (class or interface).

Primitive types are shown with a triangle icon, objects with a square, and constructors and methods with a CSD "method" unit symbol.

These are the actual dependencies or members from the class file. Some may be generated implicitly, as ("string" + int_variable) will use several StringBuffer methods. Others may be entirely synthetic, such as the "access$" methods used to access private members of a class from its inner class and vice-versa. If the "real" dependency behind a synthetic method can be determined, it will be shown, with the synthetic reference name noted at the end. Synthetic methods for a class are shown with a leading asterisk, as *someMethod(). Synthetic classes and synthetic methods of a class or interface are not shown by default. Synthetic methods can be shown by changing the "View"/"Info"/"Show Synthetic Methods" setting. The display of synthetic classes is controlled by a setting in the UML settings dialog. If shown, synthetic class names also have a leading asterisk.

22.8 Goto Window

This window displays all the nodes (objects) currently in the diagram. Clicking on one will center the diagram on that object. The intention is to make navigating a large diagram easier.
22.9 UML Settings Dialog

Settings for the UML diagram. All of these settings take effect when the diagram is updated.

Exclude By Type of Use - external objects (dependencies of objects in the project that are not themselves in the project) can be left out of the diagram.

Exclude By Type of Class - JDK classes and interfaces, just the class java.lang.Object, and synthetic classes (those generated by the compiler that don't correspond to any source code class), can be left out of the diagram.

New Node Layout - specifies which layout is applied to objects as they are added to the project.

23 Settings

- General Info
- CSD Window Settings Dialog
- Print Settings Dialog

23.1 General Info for Settings

There are three levels of CSD window and print settings in jGRASP: global, project, and file. Global settings apply to all files. Project settings apply to all files when the associated project is selected (there is a default project if none has been selected). File settings apply to a specific file. Each global setting can be overridden by a project setting, and each project setting by a file setting. In order to change the setting at any level (override the setting at the next level down), the "Use Default" checkbox must be checked. If it is not checked, the current default setting is displayed, and is not editable. Note that this default setting is not a fixed value, but the value of the corresponding setting at the next level down (which can be changed), or for global settings, the system default value (which could change with a new release of jGRASP - but won't unless absolutely necessary).

In most cases, you will want personalized settings to apply to all CSD windows, in which case you should edit the global settings. If you use a variety of compiling environments for different projects, you would most likely edit compiler settings at the project level. Since the settings of the current project apply to all files, a single file can be used in different projects without conflict. File level settings would most commonly be used to override the default language for the file, and to add any file-specific compiler flags. Some settings, such as the CSD colors and font, can only be set at the global level.

23.2 CSD Window Settings Dialog

This dialog is used to set most properties not related to printing. The CSD Text and Font pages are available only for global settings.

- OK Button - apply the settings and close the dialog.
- Apply Button - apply the settings without closing the dialog.
- Reset Button - reset the dialog to the currently active settings.
- Cancel Button - close the dialog without saving the settings.
- Languages Page
23.2.1 Languages Page

Language specific settings page on the CSD window settings dialog.

- Language Combo Box
- Environment Page
- Flags/Args/Main Page
- HTML Page
- Extensions Page
- Variables

23.2.1.1 Language Combo Box

The "Language" combo box specifies the language for which settings are being viewed or edited, and for file level settings, the language of that file.

23.2.1.2 Environment page

"Compiler Environments" specify the compile and run commands (Make, Compile, Compile and Link, Semantic Check, Run, Debug, and for Java, Run Applet and Debug Applet), working directories, error formats, and environment variables.

System compiler environments, supplied with jGRASP, can be viewed or copied, but not edited or removed.

Local compiler environments, created by the administrator, can be edited or removed only by an administrator. These have names beginning with "local:"

User compiler environments, created by the user, can be edited or removed by the user. These have names beginning with "user:"

To use a compiler environment, select it in the list, then hit the "Use" button on the environment page (to the right of the list). This will immediately activate the environment, there is not need to click "Apply" or "OK" on the settings dialog.

Use - use the selected environment (immediately applied).

New - create a new environment using a compiler environment dialog.

Copy - copy the selected environment using a compiler environment dialog. It must be saved under a new name.

View - for system environments, view the environment using a compiler environment dialog.

Edit - for user created environments, edit the selected environment using a compiler environment dialog.
Remove - remove a user created environment.

23.2.1.3 Flags/Args/Main Page
Main File - specifies the file used to substitute command variables beginning with MAIN_ or main_. This is intended to be the file containing the executable module for languages such as Java or Ada. If no main file is used, "main" variables will be substituted with the current file (useful for single file projects). The %PROJECT_PATH and %SEP variables can be used in the path.

FLAGS or ARGS - for each command, this will replace any FLAGS or ARGS variables. This is intended to be used for passing flags to a compiler or arguments to an executable.

FLAGS2 or ARGS2 - for each command, this will replace any FLAGS2 or ARGS2 variables. This can be used when multiple sets of flags are needed. In the Java compiler environments supplied with jGRASP, these are the arguments sent to the jvm during a "run", while "FLAGS or ARGS" are the arguments sent to the application.

23.2.1.4 HTML Page
This specifies the contents of the temporary HTML file used for "Run as Applet" in Java. %PATH, %FILE, %BASE, %file, %base, %PACKAGE, and %PACKAGE_PATH variables may be used here.

23.2.1.5 Extensions Page
Allows the default extensions for each language to be set. For a file that has not been opened in jGRASP before, these extensions determine which icon will be shown in the file browser page or in a file dialog, and which language will first be used when the file is opened (once opened, the language associated with a file is remembered). Extensions must be listed without a preceding dot and separated by whitespace. If the same extension is set for more than one language, an "unknown" icon will be shown, and the user will be prompted to select one of the appropriate languages before opening the file.

23.2.1.6 Variables
%FLAGS - flags or arguments.
%ARGS - same as %FLAGS.
%FILE - full path to file.
%file - filename.
%BASE - full path to file without extension.
%base - filename without extension.
%PATH - full path to directory containing file.
%EQ - equals sign.
%SEP - name separator character (\ on Windows / on UNIX).
%PATH_SEP - path separator character (; on Windows : on UNIX).
%A - "a.exe" on Windows systems, otherwise "a.out".
%PROJECT_PATH - path to the directory containing the current project, or the empty string if the current project is the default project.
%TRANSPORT - JDK jdpa (debug interface) transport appropriate to current OS. his is dt_shmem if available, dt_socket if dt_shmem is not available, and the first transport found otherwise.

%CLASSIC_OPT - "-classic" on systems where the latest JDK ships with a classic VM, otherwise the empty string.

%PACKAGE - (Java) full package name.
%CLASS - (Java) full package/class name.
%class - (Java) class name.

%PACKAGE_PATH - (Java) path to package root. Java "Run as Applet" only.

%HTML_FILE - (Java) full path to temporary html file.
%html_file - (Java) temporary html file name.

%EXTENSION_PATH - path to jGRASP extensions directory. This was set during installation on Windows, by a JGRASP_EXTENSIONS environment variable, or -e flag at startup.

%D - begins and ends a part of the command that is only used when compiler debug mode is on.
%N - begins and ends a part of the command that is only used when compiler debug mode is off.

%EY - turn on input echo. This is the default mode. In this mode, input in the Run I/O window remains visible after it is entered.
%EN - turn off input echo. In this mode, input in the Run I/O window disappears after it is entered. This is useful if the target process will echo input, or if the input should not be seen in the output stream.

%CP - turn on pipe connection mode for a command (Windows only). This is the default mode. In this mode, input and output to the target process will be through pipes.
%CD - turn on DOS window mode for a command (Windows only). If on, the target process will always run in a DOS window.

%CY - use Cygwin wedge for a command (Windows/Cygwin only). If on, the target process will be called from a Cygwin-aware native wedge program. This should eliminate output buffering problems for Cygwin applications. This wedge can not be used for non-Cygwin executables, or if you have an old version of Cygwin. For old versions of Cygwin, you will likely get an error message stating that "Cygwin1.dll" is missing, in which case you need to use a different connection mode. You might have the same problem with a new version of Cygwin if your PATH is messed up (possibly because you have multiple versions of Cygwin installed), or because the Cygwin mounts are messed up.

%CC - turn on console connection mode for a command (Windows only). If on, input to the target process will be through a console, while output is through pipes. This can eliminate I/O buffering problems seen when input is through a pipe. It does fix buffering problems on executables compiled with cygnus gcc and g++. On VC++ and Borland bcc compiled executables, it makes buffering worse (fully buffered instead of line buffered output). It will cause some jvms to crash.

%CX - turn on debug connection mode for a command (Windows NT only). This mode has not been thoroughly tested, it may have problems. If on, the target will run as a debug process in Windows NT, and console functions will be intercepted to collect output. This will make the run I/O window behave more like a console. An actual console which echoes I/O will pop up, initially iconified. This window can be used to enter escape sequences, etc. This mode will not work for processes that move the cursor around in the console. You will most commonly also need %EN to avoid a double input echo, in which case input and output will not be colored differently.

%V1 - variable specified in environment.
%V2 - variable specified in environment.
A %FLAGS, %ARGS, %FLAGS2, or %ARGS2 variable must be surrounded by whitespace; the others need not be. To put a percent symbol in a command, use %%%. To put a double quote in a quoted string, use %'.

Variables that are related to files, paths, and classes may be prepended by MAIN_ or main_. In that case, the main file will be used in the substitution, or if a main file was not specified, the current file will be used.

23.2.1.7 CSD Page

CSD generation settings.

Auto Generate CSD - if on, a CSD is generated at load time, after a template is inserted, etc.

Show Data Syms. - if on, data and type symbols are shown.

Intra-Stmt. Align - if on, parenthesis and assignment symbols are aligned.

Indent Braces - if on, curly braced in C, C++, and Java are indented to match the enclosed block.

Left Align - (experimental) if on removes the initial three-space indent when the CSD is removed. Turning this setting off and on can change comment alignment.

Show Unit Symbols - if on, package/method/function symbols are shown.

Show Boxes - if on, boxes are drawn around major code structure headings (packages, methods, functions, etc.).

Force Newlines - if on, each statement is forced to a new line.

Auto Indent - if on, hitting return will insert whitespace even with the CSD, space and tab indentation of the previous line.

Soft Tab - if on, spaces are used instead of tabs, for the tab key, block indent, and auto indent.

Font Size - CSD font size in pixels.

Tab Size - width of a tab in characters. For variable-width fonts, the with of a "W" is used.

23.2.1.8 Colors Page (global settings only)

Colors for lexical types, CSD and background color, and message and I/O window colors. Click the "Use Default" checkbox off and click on the color square to change the color.

23.2.1.9 Font Page (global settings only)

Font attributes for lexical types, and font for CSD, message, and I/O windows. Also, the choice of font and whether or not it should be antialiased. Antialiasing generally improves the appearance of text at higher resolutions (or large font sizes), but at low resolutions it may make text harder to read. On some systems, bold and italic fonts will not align with the plain font and/or each other.

23.2.1.10 Font Size Page (global settings only)

CSD Font Size - base size of the CSD Window font.

Message Font Size - base size of the message and I/O window font.

Font Scale - scale factor for all fonts.
23.2.1.11 View Page

These are global defaults for settings that appear on the CSD window view menu. The view menu settings are temporary.

23.2.1.12 Sources Page (Global and Project Settings Only)

Allows the search path for jGRASP to be specified. This path will be used for click-to-error in the jGRASP message windows as well as for finding classes shown in the call stack of the integrated debugger for Java. These are used when the message or call stack reference contains a relative path name or a Java class name. They are searched in order. If this is not set, only the working directory for the command that produced the message will be searched. If this is set, and one of the paths is "working directory", the working directory will be searched at that point. In most cases you will need to include "working directory" as one of the paths, probably the first.

Paths are entered one per line, and must be absolute.

This can be used, for example, to get click-to-error to work for JDK classes in Java stack dumps, if you have the JDK sources. In that case, the root of the JDK source tree (the directory containing the java, javax, etc. directories) should be on one line, and "working directory" on another. If you have Java sources and classes in separate directories, or use a make file in one directory with sources in another, you should specify the source directory or directories and not "working directory".

23.3 Compiler Environment Dialog

This dialog allows a compiler environment to be created, edited, or viewed. These environments specify compile and run commands, working directories, and error formats.

- Name Field - name of the environment. Not editable for system environments.
- Copy Button - copy the environment to the clipboard.
- Paste Button - paste the environment from the clipboard.
- Export Button - save the environment to file.
- Import Button - load the environment from file.
- Commands Page
- Parse Page
- Macros Page
- OK Button - save the environment and close the dialog.
- Save Button - save the environment without closing the dialog.
- Clear Button - clear all the settings.
- Reset Button - reset the dialog to the current state of the environment.
- Cancel Button - close the dialog without saving the environment.

23.3.1 Commands Page

This page specifies settings for Compile, Compile and Link, Check, Run, Debug, and (for Java) Run as Applet and Debug as Applet commands. For each command a working directory and error format can be specified. If a compile command is empty, that command will not appear on the "Compiler" menu of the
CSD window, and the icon will not appear on the toolbar. If a run command is empty, that command will not appear on the "Run" menu of the CSD Window, and the icon will not appear on the toolbar.

A variety of execution-time variables may be applied to commands and directories. Also, two static variables, %V1 and %V2, can be specified at the bottom of the page.

Environment variables can also be modified.

Whitespace separates arguments in commands. Use double quotes to specify an argument containing whitespace. Use %' (percent-single quote) to include a double quote in a quoted argument. Any variable can also be used in a quoted argument, and %% (percent-percent) must be used to include a percent symbol.

"Command" specifies the command. All variables may be used.

"Directory" specifies the working directory for the command. If left blank, the directory containing the file will be used. %PATH, %base, and for Java, %PACKAGE_PATH variables may be used here.

"Error Format" specifies the error format for the command. The format string begins with a list of target flags, followed by a dash. The rest of the string is a Perl5 regular expression. See the perlre man page for a description of this format.

The target flags are:

- f = filename
- c = class
- 1 = start line
- 2 = start column
- 3 = end line
- 4 = end column

For click-to-error to work, the numbers of target flags must be equal to the number of matching groups in the regular expression. If a filename or class is matched and the corresponding file exists, a click will open the file. If the start line is matched, that line will be highlighted. If the start and end lines and columns are matched, that piece of text will be matched. If a filename and class are matched, the filename will be tried first. If the same target type matches multiple times, the last match is used. Unmatched groups are ignored, so you can use or'ed expressions and repeated target flags to match multiple possible formats. No error message is reported if the format is bad, or the number of target flags does not match the number of groups. On the control panel "Tools" menu, there is a testing tool for pattern matches.

As an example, "f1-(\S(?:\s*\S)*):(\d+):.*" will match the file and line number of a GNU-style error message. The regular expression reads as: a matching group containing (a non-whitespace character followed by any number of non-matching groups consisting of any number of whitespace characters followed by one non-whitespace character); followed by a colon; followed by a matching group consisting of one or more digits; followed by a colon; followed by anything. In other words, "filename:line_number:other_text", where filename must begin and end with non-whitespace characters and be at least one character long, and line_number consists of one or more digits.

"Environment" allows environment variables to be changed. Variables are set one-per-line, using the following format:

- Replace: VAR=VALUE
- Prepend: VAR+=VALUE
- Append: VAR=+VALUE
Remember to add a path separator when prepending or appending to a path, like "PATH+=C:\newpath;" on Windows or "PATH+=/usr/local/newpath:" on UNIX/Linux, or "PATH+=/usr/local/newpath%PATH_SEP" for either.

All variables may be used, so you'll need to escape "%" with "%%". All whitespace is significant, so don't leave space before the end-of-line or within the command unless you mean it to be there. For example, "VAR = VALUE" will set "VAR " (VAR followed by a space) to " VALUE" (VALUE preceded by a space).

You can have a plus sign at the beginning of a value by using the %PLUS variable. For example "VAR=%PLUSval" will set "VAR" to "+val". In most environments, equals signs can not appear in environment variable names, but for completeness, you can use %EQ to include an equals sign (this will most likely produce unexpected results).

By default, settings apply to all commands. You can change the commands they apply to using "-Command" and "+Command" or "Command". For example "-Run" will make the following settings not apply to the run command, and "+Run" or just "Run" will do the opposite. These commands can be grouped on a line, separated by whitespace. Use the exact labels in the dialog for command names, only for Run Applet, use "Run_Applet" (replace space with an underscore).

For example: add C:\mypath to the path but not for run or debug and set DEBUG_TEST to 1 for debug only.

```
-Run -Debug
PATH+=C:\mypath;
-Compile -C/L -Check +Debug
DEBUG_TEST=1
```

### 23.3.2 Parse Page (C and C++ only)

CSD parsing options for C and C++. These apply only to CSD generation, not to compiling.

Include Path - search path for include files. The directory containing the file is always searched (last).

Include Files - all, none, or only local include files may be parsed. Local includes are defined as those include by "#include "filename" rather than "#include <filename>". Include file parsing is necessary for CSD generation if the include files contain unstructured macros (macros that when used, do not look like legal C or C++ code) or partial code structures. Parsing include files can make CSD generation slow, so it should be turned off if unnecessary.

Expand Macros - if true, macros are expanded during CSD generation. This may be necessary for unstructured macros, and for others, some CSD structure for the substitution result may be shown if this is on.

Use Alternate Tokens - the C++ language allows alternate tokens such as "<%" for "{".

Allow C++ Comments - allow // comments in C. This can not be done by default because of things like: x //= this is x divided by 3 in ANSI C * 3 .

### 23.3.3 Macros Page (C and C++ only)

Specifies macros that are pre-defined for CSD generation of C and C++. By default, _GRASP_IGNORE is defined: by using ifndef _GRASP_IGNORE, blocks of code that can't or shouldn't be involved in CSD generation can be ignored.
23.4 Print Settings Dialog

This dialog is used to set properties related to printing. The OK, Apply, Reset, and Cancel buttons operate in the same way as those of the CSD window settings dialog. The “Units” combo box changes the displayed units for items in this dialog.

- **Common Page**
- **CSD Page**
- **UML Page**
- **Command Page**

23.4.1 Common Page

These settings apply to both CSD and UML printing.

Standard Paper Sizes - choosing an item from this pulldown list will set the paper width and height.

Paper Width / Height - these are ignored for native windows CSD printing, and can be set in the windows print dialog (if allowed) in that case.

Horizontal / Vertical DPI - the horizontal and vertical resolution of the target device or printer. If this is not known, choose a high density (the default is 300x300dpi). If the resolution set here does not match the resolution of the target device or printer, the alignment of the printout may be off very slightly for all types of printing. The error can be 1 pixel per character, which at high resolutions is not noticeable. For Java printing, the resolution is also used to build the CSD characters, so the CSD may look chunky or asymmetrical if the resolutions are not correct.

Color - if on, printing will be in color.

23.4.2 CSD Print Settings Page

Page Numbers - if on, the page number is printed on each page.

Filename Header - if on, the filename is printed at the top of each page.

Break Pages on Form Feed - if on, form feed characters (hex code 0C) will cause page breaks in the printing. If off, these characters will be printed in the text font.

Landscape - if on, printing will be in landscape mode. This is ignored for native windows printing, and can be set in the windows print dialog in that case.

Book Format - if on, left and right margins will alternate. This is useful if the printouts will be bound.

Left Page First - if on and book format is on, the first page printed will have reversed margins.

Left / Right / Top / Bottom Margin - page margins.

Gutter - spacing between columns if more than one column is used.

Line Spacing - the CSD will stretch when line spacing is increased. At an aspect ratio of 3 or above, the CSD will lose its symmetrical appearance.

Columns - allows multiple columns of text per page.

Font Size - the size of the printed font.

Header - text for a header to be printed at the top of each page.

Font - the printed font.
23.4.3 UML Print Settings Page
Margin – The same margin is used on all four sides.
Multi-page Overlap – for multiple-page printouts, the pages must be overlapped by this amount to align them properly. This makes it easier to assemble multiple-page printouts.
Landscape - if on, printing will be in landscape mode.
Maximum printed width, height – if the printed image exceeds this size, the user will have the option to scale to this size.
Horizontal and vertical image alignment – alignment within the printed page or pages.

23.4.4 Command Page (Unix / Linux only)
Allows the printing command to be changed.

24 Fixed Bugs

24.1 Bugs in Version 1.5.0 Alpha 3 Fixed in Version 1.5.0 Beta
jGRASP will not run under some JDKs when installed in a directory path containing spaces. (JDK functionality change / bug)
Integrated Java debugger will not stop at breakpoints in classes with package access and a different base name from the source file, or jump to lines in those classes.
Integrated Java debugger does not support watches on individual fields, only on field types.
jGRASP itself is too slow when running the Java integrated debugger.
CSD printing font will not automatically install on Windows XP.
Native Windows printing of plain text may not work if CSD font is not installed.
If a compiler environment is set and settings are not applied, when a compiler environment is set at a higher level (global vs project, project vs file, global vs file), the lower level setting will be reset to the last applied value.
When settings are applied, any open settings dialogs at a lower value will be reset (any unsaved changes will be lost).

24.2 New Features in Version 1.5.0 Beta
Javadoc documentation can be generated for whole projects.
Jar (for project) and unjar (general purpose) tools added.
Java UML diagram now shows the "real" dependencies for synthetic methods.
24.3 Bugs in Version 1.4.4 Beta Fixed in Version 1.5.0 Alpha 3
If a global font scale other than 1.0 is used, changing the Look and Feel will not change the fonts (changing the global font scale afterward will change them).
If "Sources" directories are specified in the settings, clicking on an error message for an unnamed file will cause a stack dump, and the error will not be highlighted.
Plain C++ class template declarations will not parse for CSD generation.
Java integrated debugging is much too slow.

24.4 New Features in Version 1.5.0 Alpha 3
Javadoc documentation can be generated and viewed for Java files.

24.5 Bugs in Version 1.4.3 Beta Fixed in Version 1.4.4 Beta
HTML panes in help window do not scale with the jGRASP global font scale. This now works on JDK 1.3 or higher.
More than one paste of multiple lines from a CSD Window with line numbers on to a CSD Window will paste extra characters on Windows.
Network administration setup in Windows installer does not work. The admin settings directory is created but the link is not written to the data\admin file.
Administrators may get a warning that there is a current jGRASP session using the same administrator settings each time jGRASP is started.
When jGRASP is popped up by clicking on an associated file in Windows, the right and bottom edges of the Control Panel may not be painted. (JDK bug)

24.6 Bugs in Version 1.4.2 Beta Fixed in Version 1.4.3 Beta
A new project will "inherit" the UML diagram from the previous project, and it will keep it until a non-empty UML diagram is generated.
UML settings are not saved if the diagram is empty.
UML diagram may come up with zero size the first time it is used for a particular project.
Middle mouse button click function is disabled.
For Mac OS X, while moving through the CSD Window popup menu, the selection in the window changes.

24.7 Bugs in Version 1.4.1 Beta Fixed in Version 1.4.2 Beta
Default Java debugger transport (from \%TRANSPORT variable) is incorrect on IBM JDK 1.3 for Windows, and probably on other systems also.
CSD Window toolbar buttons must be double clicked to achieve single-click functions on IBM JDK 1.3 for Linux. (IBM vs Sun JDK difference)
Splash screen will display briefly, then blank out on IBM JDK 1.3 for Linux. (IBM vs Sun JDK difference)
Bottom message panel on CSD Windows may display in the wrong font, or with mixed fonts on IBM JDK 1.3 for Windows or Linux. (IBM JDK bug)
File syncing may cause jGRASP to lock up on JDK 1.4. (JDK bug)
"Run" function with Cygwin wedge will not work with newer versions of Cygwin. (Cygwin functionality change)
May not be able to shift array view (for arrays with more than ten elements) in debug variable window.
If a file is opened as binary and jGRASP is closed, the file will open as text the next time jGRASP is opened.
Files with more than one line terminator type will have all terminators converted to \"\n\" when opened in text mode (the default mode).
Various focus problems when focus-follow-mouse is on and jGRASP is run under JDK 1.4. (JDK functionality change)
Can't use newlines in "Find" find and replace fields under JDK 1.4. (JDK functionality change / bug)
File dialogs may pop up at odd sizes under JDK 1.4. (workaround for JDK bug not needed on 1.4)
CSD may look choppy under JDK 1.4. (JDK functionality change / bug)
May be far too much spacing in CSD Window on Linux under JDK 1.4. (JDK bug)
Some C and C++ macros may parse incorrectly or in very rare cases cause a stack dump.
C++ alternate keywords (not all alternate tokens) - or, xor, etc. are always turned on.
C++ template declarations and multiply templated definitions are flagged as errors during CSD generation.

24.8 New Features in Version 1.4.2 Beta
Dependencies are shown when an edge is selected in the Java UML diagram. Class members are shown when a class is selected.
Mouse wheel works for CSD Windows when run under JDK 1.4.
System selection (PRIMARY selection on XWindows) works for CSD Windows when run under JDK 1.4.
Function and menu item for pasting primary selection added.
History list added to multi-file search.
Added a "GNU extensions" switch to C and C++ compiler environments which will allow __attribute__ to be used.
Loosened up the C and C++ parsers to allow some illegal structures allowed by gcc/g++.

24.9 Bugs in Version 1.4 Beta Fixed in Version 1.4.1 Beta
CSD generation for C++ does not work with most VMs.

Bugs in Version 1.3.7 Beta Fixed in Version 1.4 Beta
During CSD generation for C, C++, and Objective-C, __FILE__, __LINE__, __DATE__, and __TIME__ macros are not set.
Native print (PostScript using lpr) not available on Mac OS X.
24.10 New Features in Version 1.4 Beta
The Objective-C language was added.
Message window contents can now be saved to file or printed.

25 How To

25.1 Change the language for a file:
To change the language of a file, select "Edit" / "Compiler Settings" / "File" on the CSD window menu. Choose the "Language Specific" tab on the settings dialog. Select the desired language from the "Language" combo box. Once a language is selected, jGRASP will remember this setting. Also, the icons in file dialogs and in the browse window will reflect this setting.

25.2 Drag files into jGRASP:
Single or multiple files can be dragged onto the virtual desktop, or onto the control panel menu bar, to the right of the menus. The files will be opened in text mode.

25.3 Open a file in binary mode:
Using the "File" / "Open File" menu on the control panel or the "File" / "Open" menu on the CSD window menu will bring up a dialog that has "Text" vs. "Binary" radio buttons. All other methods of opening a file will open it in text mode. Opening a binary file in text mode is likely to corrupt it, since all three types of line terminators will be converted to a single type when the file is saved.

25.4 Search for the selected text:
F9 and Shift-F9 or Find and Shift-Find will search forward and backward for the selected text in a CSD window. This works across CSD Windows - the selected text does not have to be in the window being searched.

25.5 Make ".h" a default file extension for C or C++ files only:
Default extensions can be set at the global or project level. Select "Edit" / "Compiler Settings" / "Global" or "Project" on a CSD window, or "Settings" / "Compiler Settings" / "Global" or "Project" from the control panel. Select "C" or "C++" from the "Language" combo box, whichever one you don't want to be the default. Select the "Extensions" tab, click the default box for extensions off, and add the desired extensions (without "h" or "H"). You can use Ctrl-C and Ctrl-V to copy the old extensions and paste them back after clicking the default box off.

25.6 Change the CSD include path for C or C++ files:
To do this, you must create and use a new compiler environment. It usually makes sense to use that environment at the project level.
Under "Settings" / "Compiler Settings" / "Project", select "C" or "C++" as the language. Select the current compiler environment and hit "Copy". Under the "Parsing" tab, you can specify include directories for CSD generation (one path per line, or with path separators on one line). Next, save the new or edited environment. On the settings dialog, select the new environment and hit "Use".

If you have multiple project with different include paths, just create a compiler environment for each. Then, you can use jGRASP projects to easily switch between environments.

25.7 Get C and C++ files with lots of ugly macros or compiler extensions to parse:

If you set up the CSD generation environment (include path, predefined macros) exactly like the environment your compile command sees, and set the parse mode to "All Files", and if no structures are partly in a header file and partly in a source file (like an included function header), and if the code is ANSI C, CSD generation should work for your code. Setting all that up can be a pain, and parsing thousands of lines of headers can be slow, so this is usually not an option, unless you just want to generate a CSD once for printing or viewing.

If you can edit the code, the easiest thing to do is surround weird code with #ifndef _GRASP_IGNORE and #endif. _GRASP_IGNORE is set in the predefined macros for all compiler environments by default. For example:

```c
#ifndef _GRASP_IGNORE
BEGIN_MESSAGE_MAP(CPrintApp, CWinApp)
ON_COMMAND(ID_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
#endif
```

This is an MFC message map. Without the ifndef, if all headers are parsed and the include path is set correctly, a CSD will be generated for this structure, but the indentation will not be what you expect, because the real structure is not what you expect.

For compiler extensions, setting predefined macros in the compiler environment can solve many problems. For example, you might define `far` to be nothing in an old 16 bit compiler.

For your own code, it is a good idea to use macros that look like real code. For example, you should leave a trailing semicolon out of a macro so it will be required in the code.
Index

(Work in Progress)

About jGRASP .............................................9, 78
Autotest Dialog.................................................86
Block Cut/Paste................................................91
Bookmarks .................................................90, 96
Breakpoints ............................................ 90, 92, 96, 99, 101
Browse Window .....................................83, 87
Call Stack Window ................................99, 100
Colors Page ...........................................106, 109
Commands Page ...........................................110
Compiler Environment Dialog ..................98, 110
Compiler Menu...........................................90, 92
Compiling ...................................................23, 26
Complexity Profile Graph (CPG).. vii, 2, 7, 9, 101
Control Panel .......... 2, 9, 12, 42, 48, 75, 82, 115
Control Structure Diagram (CSD)vii, 2, 8, 9, 22, 31, 75
Courier........................................................84, 91
CPG Menu..................................................90, 93
CPG Window................................................9, 93, 101
CSD Page........................................................106, 109, 113
CSD Print Settings Page .........................113
CSD Window2, 3, 4, 9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 25, 26, 27, 29, 37, 38, 59, 61, 63, 67, 75, 83, 84, 85, 87, 90, 92, 94, 95, 101, 105, 109, 111, 115, 116, 117
CSD Window Settings Dialog .....................105
Data Sets Menu .............................................102
Debug Window...........................................83, 99
Debugging....................................................68
Diagram Display.........................................103, 104
Edit Menu...................................................90, 91, 103
Environment page.........................................106
Error Messages..............................................27
Expect License...........................................79, 82
Extended Awk Regular Expressions...........97
Extensions Page.............................................106, 107
File
  Closing....................................................18
  Creating.................................................12, 13
  Loading...................................................19
  Saving......................................................17
File Menu ............................................ 82, 83, 90, 91, 102
Find Dialog / Page ...........................................94
Find Window...............................................83
Fixed Bugs ............................................. 9, 114
UNIX.........................................................72, 114
Using Help....................................................8, 75
Variables . 6, 64, 65, 99, 100, 106, 107, 109, 111
Variables Window ....................................99, 100
View Menu.............................. 53, 90, 92, 102, 103, 104
View Page..............................................106, 110
Watches ...................................................99, 101
Window Menu ..........................................83, 85
Windows1, 2, 11, 12, 15, 21, 30, 50, 71, 72, 73, 74, 76, 78, 79, 83, 85, 93, 98, 107, 108, 112, 114, 115