

Getting Started with G2

Tutorials
Version 2015



Getting Started with G2 Tutorials, Version 2015

December 2015

The information in this publication is subject to change without notice and does not represent a commitment by Gensym Corporation.

Although this software has been extensively tested, Gensym cannot guarantee error-free performance in all applications. Accordingly, use of the software is at the customer's sole risk.

Copyright (c) 1985-2015 Gensym Corporation

All rights reserved. No part of this document may be reproduced, stored in a retrieval system, translated, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Gensym Corporation.

Gensym®, G2®, Optegrity®, and ReThink® are registered trademarks of Gensym Corporation.

NeurOn-Line™, Dynamic Scheduling™, G2 Real-Time Expert System™, G2 ActiveXLink™, G2 BeanBuilder™, G2 CORBALink™, G2 Diagnostic Assistant™, G2 Gateway™, G2 GUIDE™, G2GL™, G2 JavaLink™, G2 ProTools™, GDA™, GFI™, GSI™, ICP™, Integrity™, and SymCure™ are trademarks of Gensym Corporation.

Telewindows is a trademark or registered trademark of Microsoft Corporation in the United States and/or other countries. Telewindows is used by Gensym Corporation under license from owner.

This software is based in part on the work of the Independent JPEG Group.

Copyright (c) 1998-2002 Daniel Veillard. All Rights Reserved.

SCOR® is a registered trademark of PRTM.

License for Scintilla and SciTE, Copyright 1998-2003 by Neil Hodgson, All Rights Reserved.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.openssl.org/>).

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations, and Gensym Corporation disclaims any responsibility for specifying which marks are owned by which companies or organizations.

Gensym Corporation
52 Second Avenue
Burlington, MA 01803 USA
Telephone: (781) 265-7100
Fax: (781) 265-7101

Part Number: DOC017-1200

Contents

Preface xi

About these Tutorials xi

Version Information xii

Audience xii

Organization xii

Conventions xiii

Related Documentation xv

Customer Support Services xvii

Chapter 1 Introduction 1

Introduction 1

G2 Strengths 2

G2 Applications 3

 Understanding the Significance of Three Key Factors 3

 Understanding the Impact of Your Application 4

 Successful G2 Applications 5

G2 Industries 6

 G2 Industries and Deployed Applications 7

 Examples 8

G2 Environment 9

 G2 Application Server 10

 G2 Telewindows Client 12

 G2 Developer's Utilities 13

 G2 Application Products 15

Integration with External Systems 16

Comparison with Other Tools 18

Chapter 2 Basic Skills 21

Goals for Learning the Basic Skills 22

Starting the G2 Server and Connecting the Telewindows Client 22

Working with Knowledge Bases	24
What is a Knowledge Base?	25
What are Modules?	25
Loading a Sample Application	26
Supporting Knowledge Bases	27
Displaying the KB Workspace Menu	28
Hiding the Operator Logbook	29
Viewing the Modules	30
Summary	31
Interacting with Objects	31
What is an Object?	32
Displaying the Object Menu	33
Naming an Object	34
Cloning an Object	35
Deleting an Object	36
Summary	36
Interacting with Workspaces	36
What is a Workspace?	37
Using the KB Workspace Menu	37
Naming a Workspace	38
Hiding and Showing a Workspace	39
Shrink Wrapping a Workspace	40
Moving Objects on a Workspace	40
Operating on a Group of Objects on a Workspace	41
Cloning a Workspace	42
Deleting a Workspace	42
Creating a New Workspace	43
Moving a Workspace	43
Lifting and Dropping Workspaces	43
Using Keyboard Commands on Workspaces	44
Displaying Text on a Workspace	47
Summary	49
Connecting Objects	50
What is a Connection?	50
Connecting Objects	51
Deleting Connections	53
Summary	54
Editing Attributes in Tables	54
What are Attributes?	54
What Types of Attributes Are There?	55
Assigning Values to Attributes	56
Displaying the Attribute Table	57
Editing Attributes	58
Displaying an Attribute Next to an Object	59
Summary	61

- Creating a Simple Rule 61
 - What is a Rule? 62
 - Performing Actions in a Rule 62
 - Using Two Different Inferencing Techniques 62
 - Choosing Between the Four Basic Types of Rules 63
 - Referring to Attributes in Rules 63
 - Creating a Rule 64
 - Recovering from Syntactic Errors 67
 - Recovering from Other Types of Errors 68
 - Summary 69

Running and Pausing Applications 69

Saving Applications and Shutting Down G2 71

Summary 72

Solutions 73

Chapter 3 Creating a Schematic Diagram 75

Goals of a Schematic Diagram 75

Loading the Knowledge Base 76

Creating and Deleting Objects Dynamically 76

- What is an Action? 76

- Using an Action Button to Create an Object 77

- Exploring Permanent and Transient Knowledge 78

- Making an Object Permanent 80

- Using Local Names in Statements 80

- Performing Multiple Actions In Order 81

- Using Proper Indentation in Statements 81

- Using an Action Button to Create a Permanent Object 82

- Using an Action Button to Delete Objects on a Workspace 83

- Summary 84

Editing a Class Definition 84

- What is a Class Definition? 85

- Creating a Class Hierarchy 86

- Organizing Classes and Instances 88

- Displaying a Class Definition 88

- Creating an Instance 90

- Editing the Icon 90

- Editing the Attributes of a Class 92

- Changing Manually Overridden Attributes of Instances 93

- Editing the Stubs of a Class 94

- Summary 96

Creating Connection Stubs Dynamically 97

- Executing Actions on Classes of Objects 97

- Using Methods and Procedures for Sequential Processing 97
- What is the Format of a Method or Procedure? 98
- Declaring Arguments 101
- Declaring Arguments for Methods 102
- Calling the Method or Procedure 102
- Creating a Method 103
- Declaring the Method 104
- Creating a User Menu Choice that Starts a Method 105
- Summary 106

Summary 106

Solutions 107

Chapter 4 Building a Knowledge Base 109

Goals of the Knowledge Base 110

Loading the Knowledge Base 110

Counting the Number of Connections 110

- Creating an Attribute for the Number of Connections 110

- Using a Rule to Count the Number of Connections 111

- Invoking the Rule by Scanning 113

- Highlighting Invoked Rules 113

- Testing the Rule 114

- Making the Rule More Robust and Efficient 115

- Summary 115

Counting Connections for any Office 116

- Using the For Prefix to Create a Generic Rule 116

- Creating a Generic Rule for the Office Class 117

- Summary 117

Using Event-Driven Processing 118

- Considering How to Invoke a Rule 118

- Detecting the Event of Creating a Connection 119

- Detecting the Event of Deleting a Connection 120

- Creating a Different Form of Generic Rule 121

- Computing Total Cost Per Minute Whenever Number of Connections Changes 122

- Computing Total Cost Per Minute Whenever the Connection Cost Changes 123

- Simulating Total Cost by Scanning 124

- Creating an Attribute Display for Every Office 126

- Summary 126

Using Data-Driven Processing 127

- Using Forward Chaining to Monitor Total Cost and Delete Connections 127

	Creating an Attribute for a Fixed Budget	130
	Summary	131
	Keeping a History of Total Cost	131
	Using Variables and Parameters to Keep a History	132
	Using a Parameter to Keep a History of Total Cost	133
	Explicitly Allowing Forward Chaining	135
	Creating a Subclass of Parameter	136
	Showing the G2 Class Hierarchy	138
	Summary	140
	Creating Subclasses of Offices	141
	Creating a Subclass of a User-Defined Class	141
	Creating Instances of Each Subclass	143
	Verifying that the Rules Apply to the Subclasses	144
	Overriding the Default Method of a Class	145
	Summary	146
	Disabling Rule Highlighting	146
	Summary	147
	Solutions	148
Chapter 5	Building a User Interface	151
	Goals of the User Interface	152
	Loading the Knowledge Base	152
	Creating a Subworkspace for an Object	152
	What is a Subworkspace?	152
	Creating a Master Object with a Subworkspace	154
	Creating an Object with a Subworkspace Dynamically	155
	Summary	156
	Displaying Details on the Subworkspace of an Object	157
	Creating End User Displays	157
	Creating a Readout for the Small Office Master	158
	Creating a Readout for the Large Office Master	159
	Using Readout Tables to Invoke Rules	160
	Creating a Trend Chart that Plots Total Cost	160
	Making the Application More Realistic	164
	Summary	164
	Sending a Message to the Operator	165
	Informing the Operator When an Office is Over Budget	165
	Informing the Operator About a Specific Office	166
	Creating an Action Button for Testing Purposes	167
	Creating a Method that Informs the Operator	168
	Adding a Wait Statement to a Method	170

- Creating a Procedure that Starts a Method 170
- Updating the Rule to Start the Procedure 172
- Starting a Method 173
- Calling a Method 174
- Summary 175

- Animating Objects 176
 - Creating a Method that Animates an Office 176
 - Creating a Loop in a Method 178
 - Animating the Office When it is Almost Over Budget 179
 - Using Subsecond Timing for Animation 179
 - Summary 180

- Making Workspaces Attractive and Informative 180
 - Changing the Color of a Workspace 181
 - Creating a Workspace Frame 181
 - Creating Workspace Subclasses 182
 - Creating Background Graphics for a Workspace 183
 - Summary 185

- Showing Workspaces Programmatically 185
 - Using an Initially Rule to Show a Workspace on Startup 185
 - Creating a Button that Iconifies a Workspace 186
 - Summary 187

- Configuring the User Interface 187
 - What are User Modes? 188
 - What are User Interface Configurations? 188
 - Configuring the Office for Developer Mode 189
 - Configuring the Office for Operator Mode 192
 - Starting the Application in Operator Mode 193
 - Summary 194

- Running the Prototype 194
 - Configuring the Schematic in Developer Mode 194
 - Saving the Prototype 196
 - Running the Prototype in Operator Mode 196
 - Simulating an Over Budget Situation 197

- Loading the Finished Application 198

- Summary 199

- Solutions 200

Appendix A Error Handling 205

- Common Errors 205
 - Action Buttons Don't Work 205
 - Cannot Enter a Name 205
 - Connection Attributes Not Updating 205

Rule is Not Being Invoked **206**
Procedures and Methods Not Executing **206**
Runtime Errors **207**

Glossary 209

Index 225

Preface

Describes the contents of these tutorials, the audience for these tutorials, and the style conventions these tutorials use.

About these Tutorials	xi
Version Information	xii
Audience	xii
Organization	xii
Conventions	xiii
Related Documentation	xv
Customer Support Services	xvi



About these Tutorials

These tutorials introduce a new G2 user to the basic concepts and techniques for creating G2 applications.

These tutorials:

- [Introduce G2](#), its features, its application, and its benefits.
- Teach the [basic skills](#) for interacting with the G2 environment.
- Teach how to [create a schematic diagram](#).
- Describe how to [build a simple knowledge base](#).
- Describe how to [build a simple user interface](#).
- Provides [error handling](#) guidelines.

Throughout the tutorials, you will be building a simple application for a video tele-conferencing application. Each tutorial builds on the previous tutorial to

create a fully operational prototype for a simple video tele-conferencing application.

Version Information

Getting Started with G2 is designed to operate with G2 Version 6.0. The tutorials use features of G2 6.0 that do not exist in earlier versions. Also, it does not incorporate features of G2 Version 7.0 or later.

Audience

Getting Started with G2 is written for the new G2 user, who has no prior experience with G2. These tutorials not only teach new users the basics of using G2, they also introduce basic object-oriented programming concepts.

The tutorials cover similar topics to that of the G2 Part I course, taught by Gensym's Educational Services Department. While the course presents a somewhat broader range of topics than these tutorials, the tutorials allow you to create a single sample application that uses all the features learned.

Organization

These tutorials contain five chapters and one appendix:

	Title	Description
<u>1</u>	<u>Introduction</u>	Gives an overview of G2's strengths, general application areas, industry applications, and the overall G2 environment, including the G2 core, Telewindows client, G2 modules, and G2 layered products.
<u>2</u>	<u>Basic Skills</u>	Teaches the basic skills required for working with G2: interacting with objects and workspaces, editing attributes in a table, creating a simple rule, and running the application.
<u>3</u>	<u>Creating a Schematic Diagram</u>	Teaches the basic skills for interactively creating a schematic diagram of a video conferencing application.

	Title	Description
4	Building a Knowledge Base	Describes how to create a simple application that computes the total cost of a video conferencing office based on the number of connections and that dynamically deletes the connections if the office is over budget.
5	Building a User Interface	Shows how to create a simple end user interface for the video conferencing prototype that includes subworkspaces of objects, charts, readouts, messages, animation, and user modes.
A	Error Handling	Provides simple guidelines for recovering from some of the most common errors that novice G2 users make.

Conventions

This guide uses the following typographic conventions and conventions for defining system procedures.

Typographic

Convention Examples	Description
g2-window, g2-window-1, ws-top-level, sys-mod	User-defined and system-defined G2 class names, instance names, workspace names, and module names
history-keeping-spec, temperature	User-defined and system-defined G2 attribute names
true, 1.234, ok, "Burlington, MA"	G2 attribute values and values specified or viewed through dialogs

Convention Examples	Description
Main Menu > Start	G2 menu choices and button labels
KB Workspace > New Object create subworkspace Start Procedure	
conclude that the x of y ...	Text of G2 procedures, methods, functions, formulas, and expressions
<i>new-argument</i>	User-specified values in syntax descriptions
<i>text-string</i>	Return values of G2 procedures and methods in syntax descriptions
File Name, OK, Apply, Cancel, General, Edit Scroll Area	GUIDE and native dialog fields, button labels, tabs, and titles
File > Save Properties	GMS and native menu choices
workspace	Glossary terms
<i>c:\Program Files\Gensym\</i>	Windows pathnames
<i>/usr/gensym/g2/kbs</i>	UNIX pathnames
<i>spreadsh.kb</i>	File names
<i>g2 -kb top.kb</i>	Operating system commands
<i>public void main() gsi_start</i>	Java, C and all other external code

Note Syntax conventions are fully described in the *G2 Reference Manual*.

Procedure Signatures

A procedure signature is a complete syntactic summary of a procedure or method. A procedure signature shows values supplied by the user in *italics*, and the value (if any) returned by the procedure underlined. Each value is followed by its type:

```
g2-clone-and-transfer-objects
  (list: class item-list, to-workspace: class kb-workspace,
   delta-x: integer, delta-y: integer)
  -> transferred-items: g2-list
```

Related Documentation

G2 Core Technology

- *G2 Bundle Release Notes*
- *Getting Started with G2 Tutorials*
- *G2 Reference Manual*
- *G2 Language Reference Card*
- *G2 Developer's Guide*
- *G2 System Procedures Reference Manual*
- *G2 System Procedures Reference Card*
- *G2 Class Reference Manual*
- *Telewindows User's Guide*
- *G2 Gateway Bridge Developer's Guide*

G2 Utilities

- *G2 ProTools User's Guide*
- *G2 Foundation Resources User's Guide*
- *G2 Menu System User's Guide*
- *G2 XL Spreadsheet User's Guide*
- *G2 Dynamic Displays User's Guide*
- *G2 Developer's Interface User's Guide*
- *G2 OnLine Documentation Developer's Guide*
- *G2 OnLine Documentation User's Guide*

- *G2 GUIDE User's Guide*
- *G2 GUIDE/UII Procedures Reference Manual*

G2 Developers' Utilities

- *Business Process Management System Users' Guide*
- *Business Rules Management System User's Guide*
- *G2 Reporting Engine User's Guide*
- *G2 Web User's Guide*
- *G2 Event and Data Processing User's Guide*
- *G2 Run-Time Library User's Guide*
- *G2 Event Manager User's Guide*
- *G2 Dialog Utility User's Guide*
- *G2 Data Source Manager User's Guide*
- *G2 Data Point Manager User's Guide*
- *G2 Engineering Unit Conversion User's Guide*
- *G2 Error Handling Foundation User's Guide*
- *G2 Relation Browser User's Guide*

Bridges and External Systems

- *G2 ActiveXLink User's Guide*
- *G2 CORBALink User's Guide*
- *G2 Database Bridge User's Guide*
- *G2-ODBC Bridge Release Notes*
- *G2-Oracle Bridge Release Notes*
- *G2-Sybase Bridge Release Notes*
- *G2 JMail Bridge User's Guide*
- *G2 Java Socket Manager User's Guide*
- *G2 JMSLink User's Guide*
- *G2 OPCLink User's Guide*
- *G2 PI Bridge User's Guide*
- *G2-SNMP Bridge User's Guide*

- *G2 CORBALink User's Guide*
- *G2 WebLink User's Guide*

G2 JavaLink

- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

G2 Diagnostic Assistant

- *GDA User's Guide*
- *GDA Reference Manual*
- *GDA API Reference*

Customer Support Services

You can obtain help with this or any Gensym product from Gensym Customer Support. Help is available online, by telephone, by fax, and by email.

To obtain customer support online:

➔ Access G2 HelpLink at www.gensym-support.com.

You will be asked to log in to an existing account or create a new account if necessary. G2 HelpLink allows you to:

- Register your question with Customer Support by creating an Issue.
- Query, link to, and review existing issues.
- Share issues with other users in your group.
- Query for Bugs, Suggestions, and Resolutions.

To obtain customer support by telephone, fax, or email:

➔ Use the following numbers and addresses:

	Americas	Europe, Middle-East, Africa (EMEA)
Phone	(781) 265-7301	+31-71-5682622
Fax	(781) 265-7255	+31-71-5682621
Email	service@gensym.com	service-ema@gensym.com

Introduction

Gives an overview of G2's strengths, general application areas, industry applications, and the overall G2 environment, including the G2 core, Telewindows client, G2 modules, and G2 layered products.

Introduction	1
G2 Strengths	2
G2 Applications	3
G2 Industries	6
G2 Environment	9
Integration with External Systems	16
Comparison with Other Tools	18



Introduction

G2 is a powerful object-oriented development and deployment environment for managing and optimizing dynamic, complex decision support and control applications that leverage intelligent system technologies.

G2 allows application developers to express objects, rules, methods, and procedures, using structured natural language so application developers can readily understand, test, and modify models. G2 allows organizations to apply the knowledge in these models dynamically to their operations to reach conclusions, provide advice, and take actions in real time or simulated real time.

G2 applications can follow multiple lines of reasoning and analyze large amounts of data and numerous trends concurrently. G2 applications can maintain an understanding of the behavior of processes over time and the currency of information, both of which are critical for real-time decision support and control.

You use G2 to bring real-time intelligence to your mission-critical operations. G2 lets you rapidly deliver intelligent solutions that dramatically improve the consistency, efficiency, flexibility, and quality of your operations and the time to market of your products and services.

G2 Strengths

G2's major strengths lie in its ability to perform these capabilities:

Capability	Description
Abnormal event detection	Continuously monitor multiple asynchronous events and data streams simultaneously in real time
Rule-based reasoning	Turn complex data into useful information by reasoning about it, using object models, rules, and empirical knowledge (heuristics)
Intelligent decision support	Diagnose problems quickly, provide intelligent advice to operators, and guide operators through standard operating procedures
Advanced control and optimization	Maintain optimal operating conditions based on constraints, anticipate problems, and take the right action in a timely manner
Dynamic scheduling	Dynamically coordinate and schedule activities, equipment, resources, and information through a complex process

G2 Applications

G2 supports a wide range of applications and industries. Before you begin planning and developing your G2 application, it is useful to understand how your particular application relates to existing G2 applications.

One place to begin is to consider the relative importance of three key factors in your application. Secondly, it is helpful to understand the impact your application will have at various levels in your organization. Finally, you might want to identify your application with similar, successful G2 applications in a variety of industries.

Understanding the Significance of Three Key Factors

To better understand how to fit G2's capabilities with your application, it is useful to ask yourself the following three questions:

Q Does my application have to integrate with external systems?

For example, does your application have read or write data from or to a database, receive values from or control PLCs or DCSs, or create native end user interfaces?

Q Does my domain require user interaction, possibly somewhere on the network?

For example, do operators need to view and possibly control your process through an end user interface, or do managers need to view process information while the process is running?

Q Does my environment have a well-defined physical representation?

For example, does your process consist of physical objects or processes that represent well-defined tasks, or does your process manipulate physical objects or data structures during processing.

If you answer yes to all of these questions, then you will probably be using a large portion of G2's capabilities. However, your particular application might emphasize some capabilities over others. Most G2 applications consist of at least some data integration, some user interface and networking capabilities, and some physical representation of the process. In all cases, G2 handles all three aspects of the application seamlessly: data integration, user interface, and graphical representation.

Understanding the Impact of Your Application

Obtaining buy-in for your application is critical to its success within your organization. To assist you, as a G2 application developer, in obtaining buy-in, it is useful to think about the goals of your application along three dimensions:

- **Strategic** – High-level corporate goals, such as profitability, yield, throughput, quality control, or energy efficiency.
- **Tactical** – Mid-range solutions that support strategic goals, such as process monitoring, operator decision support, alarm validation, standard operating procedure enforcement, statistical process control (SPC), or statistical quality control (SQC).
- **Operational** – Specific implementation of the tactical goals, for example, rule-based reasoning, model-based reasoning, heuristics, or time-based reasoning.

To ensure success, it is important to focus at the correct level, depending on the audience:

- During the development phase, when you are justifying the G2 application to high-level managers and decision-makers, you need to focus on strategic goals.
- When you are working with mid-range managers to determine which aspect of the overall process the application should focus on initially and downstream, you need to focus on the tactical goals.
- When you are determining how to implement the tactical goals in G2 to support end user requirements as well as future development, you need to focus on the operational goals.

Successful G2 Applications

G2 supports a wide range of applications that span across a variety of industries. One useful way to understand your application is to map it to existing successful G2 applications. This table lists existing, successful G2 application by its strategic goal and supporting tactical solutions:

Strategic G2 Applications and Tactical Solutions

Strategic Goals	Tactical Solutions
Production Management	<ul style="list-style-type: none"> • Process monitoring • Abnormal situation analysis and diagnostics • Operator decision support • Alarm management • Production coordination • Manufacturing execution system (MES) execution • Energy management • Environmental management • Maintenance management
Intelligent Process Management and Optimization	<ul style="list-style-type: none"> • Optimization of throughput, yield, and efficiency • Statistical quality control (SQC) • Closed-loop control • Statistical analysis and control (SPC) • Grade change optimization • Process verification • Plant-wide optimization
Product and Process Transition Management	<ul style="list-style-type: none"> • Abnormal situation management • Startup/shutdown management • Standard operating procedures (SOPs) enforcement

Strategic G2 Applications and Tactical Solutions

Strategic Goals	Tactical Solutions
Network Fault Management	<ul style="list-style-type: none">• Managing availability of business applications and underlying network• Managing availability of overall network• Managing availability of a subset of network devices
Business Process Management	<ul style="list-style-type: none">• Business process modeling and reengineering (BPR)• Executive operational decision support• Billing and auditing management

G2 Industries

G2 has been deployed in a wide range of industries throughout the world. In planning your application, it can be helpful to understand where the application fits into the overall spectrum of deployed G2 applications, by industry. These industries fall into four major categories:

- **Continuous manufacturing** – Material flows continuously from process to process.
- **Discrete manufacturing** – Individual units flow one at a time from process to process.
- **Batch manufacturing** – A batch of material or individual units flow from process to process.
- **Telecommunications** – Data flows over data, wire, or wireless networks.

G2 Industries and Deployed Applications

This table shows the industries in which G2 has successfully been deployed, examples of each, and application areas for that industry:

G2 Industries and Deployed G2 Applications

Industry	Examples	Deployed G2 Applications
Continuous manufacturing	Food, pharmaceuticals, metal and mining, cement, chemical and petrochemical, oil and gas pipeline, pulp and paper, utilities	<ul style="list-style-type: none"> • Advanced control and optimization • Scheduling • Abnormal situation management • Energy management • Emissions management • Business process modeling
Batch manufacturing	Food, pharmaceuticals, metals, cement	<ul style="list-style-type: none"> • Batch management • Scheduling
Discrete manufacturing	Automotive, electronics	<ul style="list-style-type: none"> • Scheduling • Shop floor control
Communications	Data networks, ATM, phone, wireline and broadband, wireless, satellite	<ul style="list-style-type: none"> • Application availability and fault management • Network fault management • Device fault management

If you are in an industry other than one of the industries listed above, you might need to do a little more work to understand how you will deploy G2 to solve your particular problem. However, in most cases, you will find that your industry and/or application has significant overlap with existing successful G2 installations.

Examples

The next diagram shows examples of G2 applications in various industries:

Continuous Manufacturing

Oil Refinery Process Management

- Quality control
- Process control
- Closed-loop control

Food Production Management

- Product consistency
- Reduced waste
- Quality control
- SCADA
- Statistical process control
- Recipe management
- Intelligent decision support

Metal Manufacturing Materials Handling Management

- Increased throughput
- Improved materials handling

Communications

Satellite Communications Management

- Complex network management

Telecom Management Support

- Network management status
- Automatic fault correction
- Intelligent decision support

Discrete Manufacturing

Auto Manufacturing Prototyping

- Flexible manufacturing of prototypes
- Open cell controllers
- Part tracking
- Tool management
- Operator decision support

Batch Manufacturing

Pharmaceutical Materials Management

- Intelligent tracking and scheduling
- Automated storage and retrieval
- Inventory management
- Regulatory compliance for tracking batch record information
- Reduced cycle time
- Optimization of materials handling

Other

Shipping Port Design

- Simulation modeling to maximize efficiency of port design

G2 Environment

G2 is a **intelligent real-time system (IRTS)** for developing and deploying mission-critical, client/server applications. To describe the environment in layman's terms, G2 allows you to create "objects" that:

- Represent your process, both graphically and conceptually.
- Reason about the relationships between these objects and the environment in general, including simulated and historical data.
- Maintain a continuous awareness of the current process.
- "Think" in real time and "make the right decision" based on the current evidence, including directly controlling the process.
- Interact with end users as required, for example, to send an alarm.

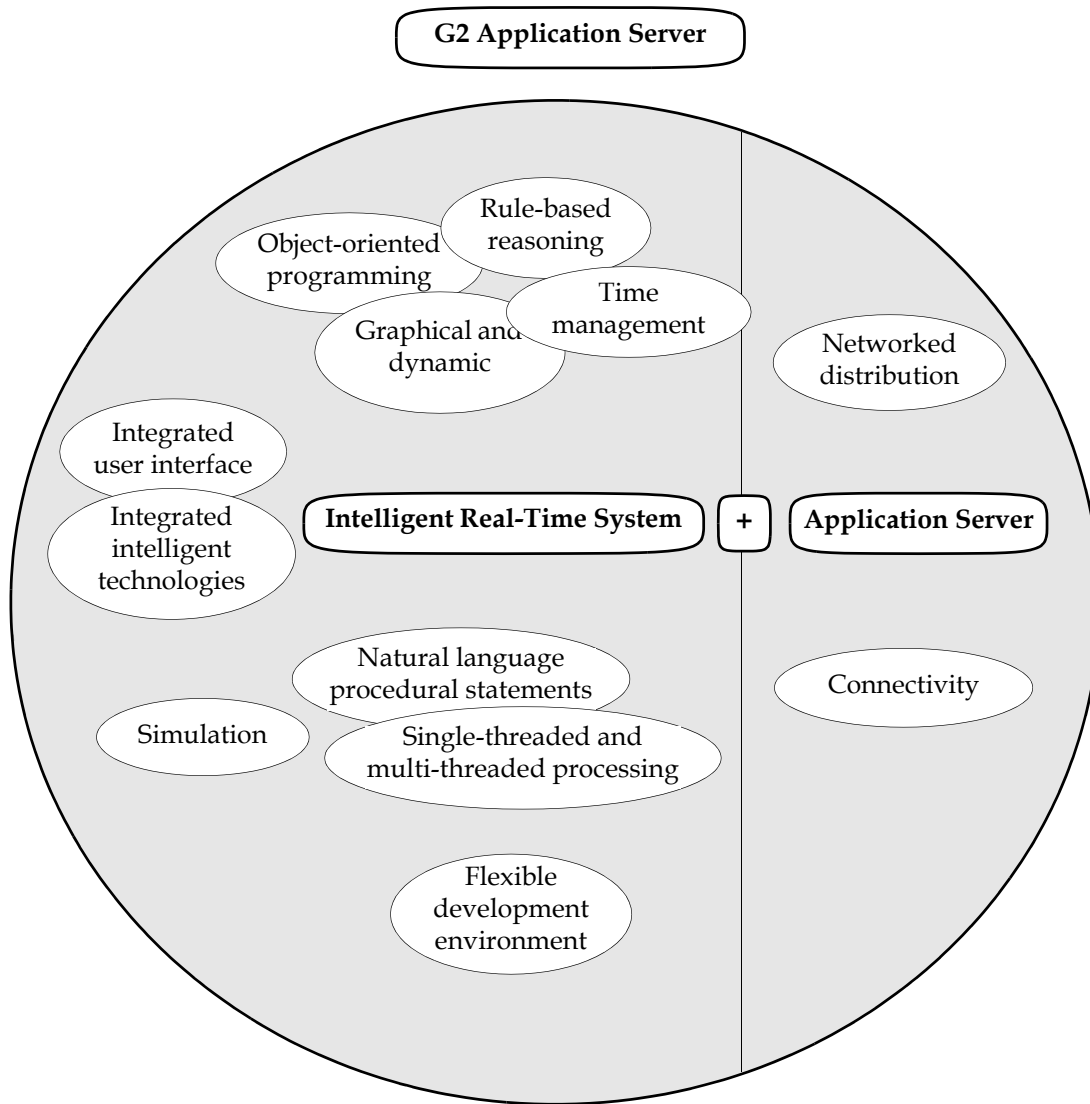
A G2 application can be distributed over a wide network of clients, using its own graphical user interface (GUI) or using some other GUI standard such as Java, Windows, or Visual Basic.

The G2 environment consists of four major areas:

- **G2 application server** – A complete development environment that provides the full range of features needed to develop and deploy intelligent real-time applications.
- **G2 Telewindows client** – A remote G2 process that allows remote users to access the server application over a network.
- **G2 developer's utilities** – Optional components of the G2 application server that allow you to perform specific functions.
- **G2 application products** – Layered applications built on top of G2 that support application development in specific domains.

G2 Application Server

The **G2 application server**, also called the **G2 core**, provides the full range of features needed to develop and deploy intelligent real-time applications. The G2 core bridges the gap between a traditional application server and an intelligent application as this diagram shows:



The G2 application server provides these features, using these implementation techniques:

Features and Implementation of G2 Application Server

Feature	Implementation
Complete object-oriented programming environment	Objects, classes, methods, inheritance, multiple inheritance, encapsulation, polymorphism
Rule-based reasoning	Event detection, decision trees, data-driven processing, forward and backward chaining, scanning, generic reasoning over classes
Time management	Continuous processing, time-stamping, historical data, update intervals, wait states, temporal reasoning
Fully integrated networked distribution	Client/server architecture, integrated network tools
Connectivity	Bridges to numerous standard databases, process control systems, files, and end user development environments
Graphical and dynamic	Icons, schematics, animation, class libraries
Natural language procedural statements	Syntax-guided text editor; local declarations; begin-end statements; case statements; if-then-else statements; looping; repeat statements; return values; nested procedure statements, error handling
Single-threaded and multi-threaded processing	Standard procedural statements for single-threaded processing; wait, allow other processing, and do in parallel statements for multi-threaded processing
Fully integrated user interface	Menus, dialogs, spreadsheets, localization
User interface	Windows, Java, X Windows displays
Integrated intelligent technologies	Fuzzy logic, neural networks
Simulation	Periodic signals, formulas, functions
Flexible development environment	Modular, configurable, rapid prototyping and continuous improvement capabilities

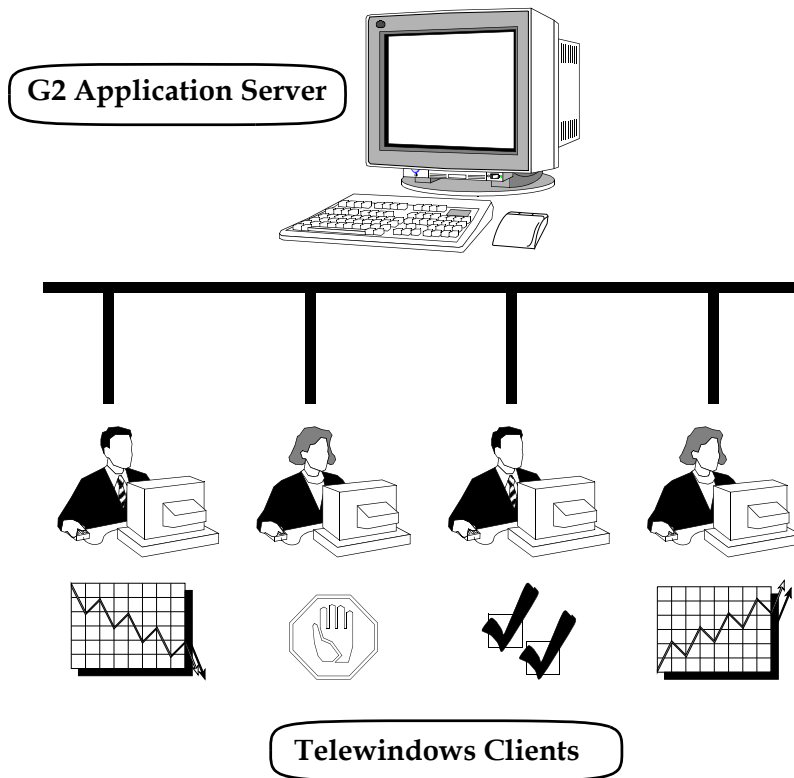
G2 Telewindows Client

G2 clients can access the G2 application server in a completely seamless manner through a **Telewindows client** application, which is a remote G2 process that allows remote users on the network to access the server application. Telewindows client simply logs in to the G2 server from another machine. The client can view or manipulate the application as if he or she were sitting in front of the server's terminal. The G2 application developer determines how much access the Telewindows client has to the application.

On Windows platforms, **Telewindows Next Generation** (*twng.exe*) provides a Windows user interface for G2 developers. G2 also provides a rich set of tools for developing Windows user interfaces for end user applications, including menus and toolbars, standard and custom dialogs, a variety of Windows views, including tree views, shortcut bars, chart views, property grids, and workspace views, and tabbed MDI mode.

This diagram shows the seamless integration of Telewindows clients to the G2 application server, where each client might see a different view of the application:

Seamless Client/Server Integration



G2 Developer's Utilities

G2 developer's utilities are components of the G2 application server that allow you to perform specific functions, such as user interface development or event management. This table describes each of the G2 Developer's Utilities:

G2 Developer's Utility	Descriptions
G2 Run-Time Library (GRTL)	A common set of utility functions, an object model, and user interface development tools to provide a consistent object model, including application configuration, localization, repositories. GRTL also provides a fully functional and extensible end user interface for decision management applications.
G2 Event Manager (GEVM)	Event management tools, including tools for creating operator message browsers and a blackboard of internal event states.
G2 Web (GWEB)	Out-of-the-box Web pages, SOAP services, WSDL support, as well as classes and APIs enabling G2 to implement an HTTP server and serve HTML pages, XML structures, SOAP services, G2GL/BPMS processes, and files.
G2 Reporting Engine (GRPE)	Out-of-the-box reports, classes, and APIs to define reports and charts, and dialogs to configure and visualize them. GRPE supports collecting values from CSV files, databases, or G2 items, displaying values in Telewindows Next Generation, and exporting values to CSV files, Excel, databases, and G2 items.
G2 Dialog Utility (GDU)	A set of APIs for building static and dynamic custom Windows dialogs.
Business Process Management System (BPMS)	Extensions to the G2 Graphical Language (G2GL) including palettes and Windows dialogs for all G2GL blocks, subclasses of G2GL processes, tools for invoking detection, test, and response processes for domain objects, and pre-defined Web services that can be called from a G2GL process, including OS processes, sending email, interacting with databases and files, generating SymCure events, and invoking BRMS rules.

G2 Developer's Utility	Descriptions
Business Rules Management System (BRMS)	An environment for editing, organizing, analyzing, and executing business rules. You define business rules for a class of G2 objects in a given category. A business rule consists of one or more conditions and actions, which you define interactively based on the class. You invoke rules programmatically by invoking all rules in one or more categories for a set of G2 objects.
G2 Event and Data Processing (GEDP)	A multi-purpose graphical language to express a flow of data, perform calculations, execute functions, and generate messages and events. GEDP flow diagrams are typically used to analyze numeric values, detect patterns, generate event states, and trigger diagnostic logic, for example, using SymCure.
G2 Data Source Manager (GDSM)	Classes and APIs for managing network connections.
G2 Data Point Manager (GDPM)	Functionality for configuring, logging, replaying, and simulating datapoints, typically related to external sensors such as temperature, pressure, and flow.
G2 Engineering Unit Conversion (GEUC)	Tools for specifying the engineering units for entering and displaying values, as well as a large number of synonyms for those conversions in both the English and metric systems.
G2 Error Handling Foundation (GERR)	Tools that provide a common approach for dealing with errors, including logging them and reporting them.
G2 Relation Browser (GRLB)	Tools for displaying G2 relations and user-defined relations in a graphical layout.

G2 Application Products

G2 offers a number of **application products**, which are layered products built on top of G2 that support development in specific domains. This table describes each G2 layered product:

Layered Product	Description
Optegrity	An extensible software platform for abnormal condition management applications for the process manufacturing industries. Optegrity applications help ensure sustained operational performance and continuous availability of production assets. Its applications detect and resolve abnormal process conditions before they disrupt productivity and threaten quality and profitability.
Integrity	A G2-based software platform that provides advanced capabilities for real-time fault and procedure management applications. Once on line, Integrity applications can help manage distributed, multi-vendor enterprise systems by automatically detecting and diagnosing network or other operational failures.
ReThink	A graphical modeling and simulation toolkit for business process and workflow simulation, analysis, and automation. e-SCOR is built on top of ReThink, so developers can use all the capabilities of ReThink to customize e-SCOR.
e-SCOR	An extensible software platform that enables businesses to design and manage complex, dynamic supply chains. Based on the Supply Council's SCOR industry-independent standard, e-SCOR allows organizations to analyze their current supply chains and perform what-if simulations to predict the performance of new supply-chain configurations.

Layered Product	Description
NeurOn-line (NOL)	Graphical block language for producing online neural-network models that predict, control, and optimize complex, non-linear processes. Real-time applications include soft-sensing for predicting product quality, mode-based sensor validation, set-point optimization and diagnosis. With NeurOn-Line, manufacturers improve efficiency, product quality and yields.
G2 Diagnostic Assistant (GDA)	Graphical block language that enables data monitoring, filtering, and diagnostics, statistical process control, alarm management, combinatorial and fuzzy logic, closed-loop control.

Integration with External Systems

G2 integrates many different software technologies together in one package, not only intelligent system technologies for which G2 is well-known but more conventional technologies – object technology, knowledge-base technology, data interfaces, graphical user interfaces, and application server support. Through a wide range of off-the-shelf bridge products, G2 and the applications built on it get data from a wide variety of data sources – databases, control systems, and various real-time data sources. You can also use G2 Gateway to build custom bridges, using the C programming language.

This table summarizes the G2 bridges that provide integration with external systems:

G2 Bridge	Description
Databases	
G2-Oracle Bridge	Provides communication with Oracle.
G2-Sybase Bridge	Provides communication with Sybase.
G2-ODBC Bridge	Provides communication with any relational database on any platform for which there is an ODBC driver.

G2 Bridge	Description
Devices and Data Historians	
G2-PI Bridge	Provides communication with the PI data historian.
G2-OPC Client Bridge (OLE for Process Control)	Provides communication with data supplied by any OPC-compliant server.
Distributed Object and Networking Standards and Protocols	
G2 ActiveXLink	Provides communication with Microsoft ActiveX/COM.
G2 JavaLink	Provides communication with Java/RMI.
G2 Java Mail Bridge	Provides communication with JavaMail (JMail).
G2 JMSLink	Provides communication with Java Message Service (JMS).
G2 Java Socket Manager	Provides communication with Java Sockets.
G2 Java SNMP	Provides communication with Java Simple Network Management Protocol (SNMP).
G2 CORBALink	Provides communication with Common Object Request Broker Architecture (CORBA).
G2-HLA Bridge	Provides communication with High Level Architecture (HLA), an integrated architecture that provides a common architecture for Modeling and Simulation (M & S).
G2 WebLink	Provides communication with HTTP, the protocol of the World Wide Web.
Core G2	Provides integration with Web services, SOAP, HTTP, and TCP/IP sockets.

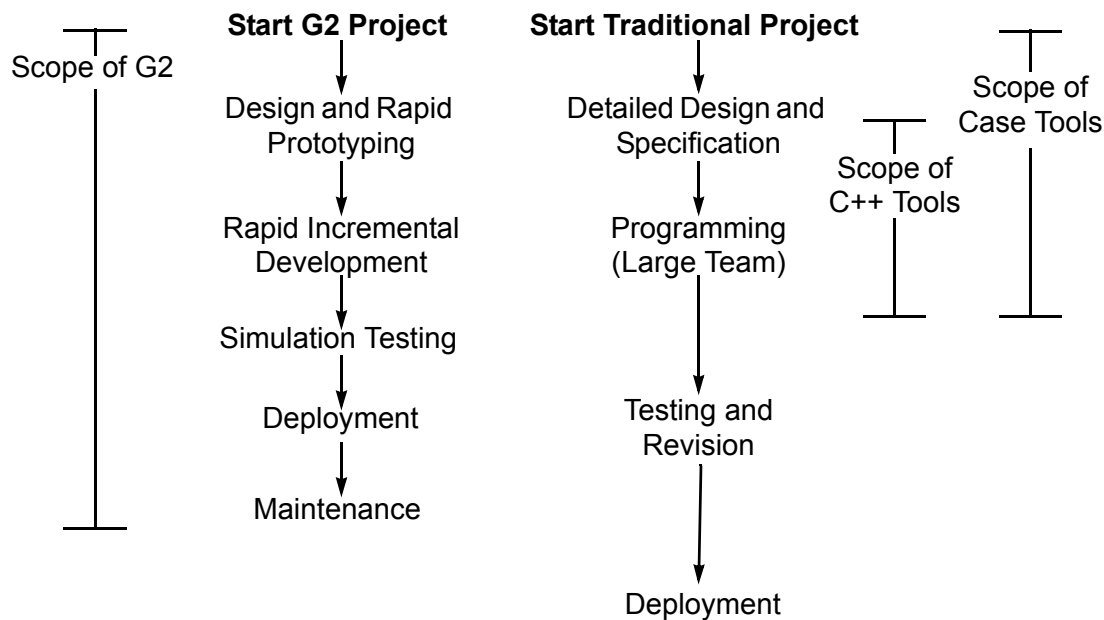
Applications built on G2 are portable and interoperate across many computer platforms, including workstations from Sun Microsystems, Hewlett-Packard, and IBM, as well as PCs by many different manufacturers running under Windows. G2 supports Windows and UNIX platforms.

Comparison with Other Tools

G2 is a complete object-oriented environment for developing and deploying intelligent applications. Unlike other approaches, you will find that G2 shortens the time to completion of applications by supporting your entire project life-style – from design, to development, simulation, deployment, through ongoing maintenance. One tool, G2, fully and effectively supports each step in the project.

Traditional tools, such as C, C++, or Computer Aided Software Engineering (CASE) are most useful during the earliest stages of a development and deployment project. As the project moves to later stages, such as testing, deployment, and maintenance, the effectiveness and development productivity of traditional tools rapidly decrease.

G2 vs. Traditional Development and Scope of Project Life Cycles



G2 is a comprehensive development and deployment environment that supports and shortens your entire project life cycle. How does G2 support the entire application life cycle?

The key lies in G2's seamless integration of the technologies you need for rapid development and deployment. G2's technologies are integrated to provide you a rich and complete environment for developing and deploying your intelligent applications. G2's technologies include:

- Concurrent real-time execution
- Object-oriented design

- Interactive graphics
- Rules, procedures, and methods
- Structured natural language
- Dynamic modeling and simulation
- Distributed processing and client/server networking
- Connectivity with online data

Basic Skills

Teaches the basic skills required for working with G2: interacting with objects and workspaces, editing attributes in a table, creating a simple rule, and running the application.

Goals for Learning the Basic Skills	22
Starting the G2 Server and Connecting the Telewindows Client	22
Working with Knowledge Bases	24
Interacting with Objects	31
Interacting with Workspaces	36
Connecting Objects	50
Editing Attributes in Tables	54
Creating a Simple Rule	62
Running and Pausing Applications	70
Saving Applications and Shutting Down G2	72
Summary	73
Solutions	74



Goals for Learning the Basic Skills

In this tutorial, you will learn the basic skills needed for working with G2, including:

- [Loading G2.](#)
- [Working with knowledge bases.](#)
- [Interacting with objects.](#)
- [Interacting with workspaces.](#)
- [Connecting objects.](#)
- [Editing attributes in tables.](#)
- [Creating a simple rule.](#)
- [Running and pausing applications.](#)
- [Saving applications and shutting down G2.](#)

Note The examples shown in this tutorial assume you are working on a Windows platform.

Starting the G2 Server and Connecting the Telewindows Client

In this lesson, you will learn how to start the G2 server process on your platform and connect the Telewindows client.

You start the G2 server from a batch file or shell script, which sets up the various environment variables required to create an application.

By default, the G2 server window appears. You can also run the server in the background with no window by using the `-no-window` command-line option.

Typically, you create applications by connecting to the G2 server through the Telewindows client. On Windows, you use the Telewindows Next Generation (`twng.exe`) client, which provides a Windows user interface for building G2 applications. The G2 server can be running on either a Windows machine or a UNIX machine.

Telewindows automatically connects to the G2 server running on the local machine on port 1111. You can also use command-line options to connect to a G2 server running on another machine and/or port.

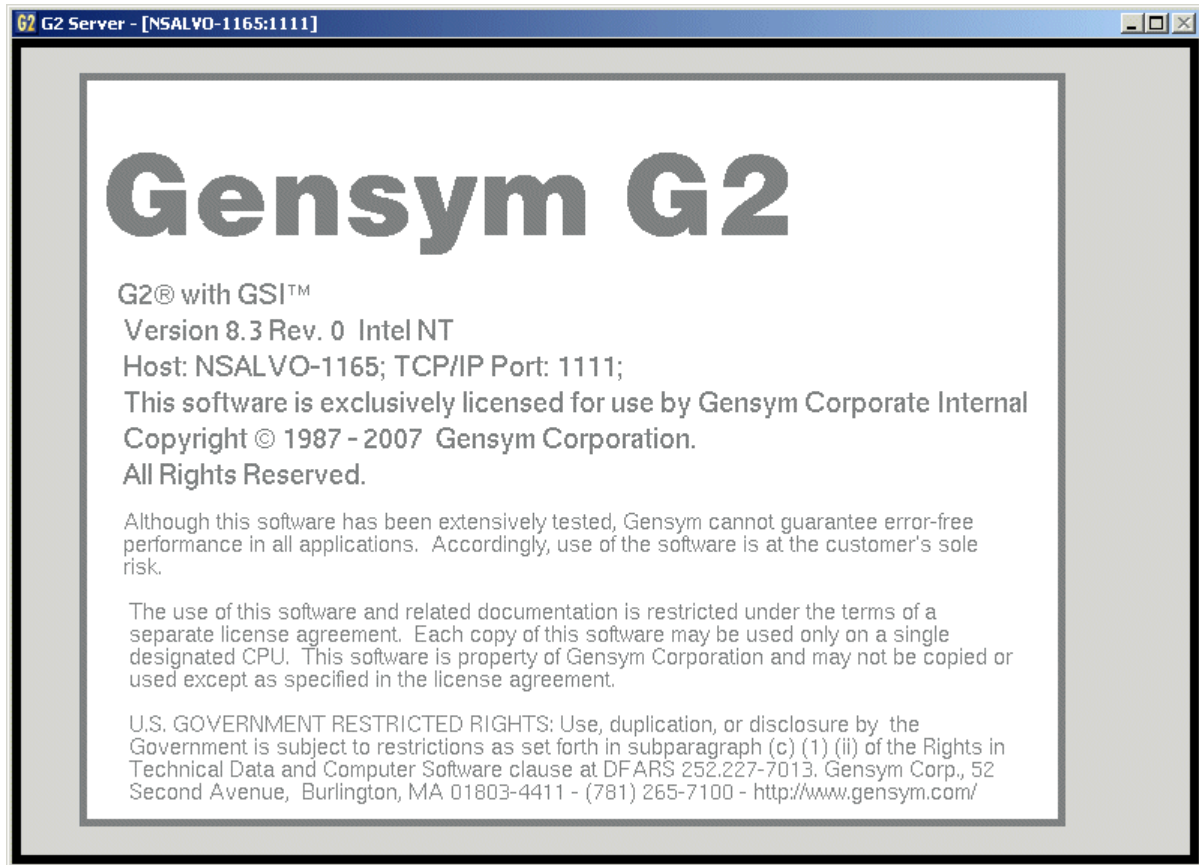
To start the G2 server on a Windows platform:


- ➔ Choose Start > Programs > Gensym G2 2011 > Start G2 Server.

To start the G2 server on a UNIX platform:

- ➔ In a UNIX shell, change to the *g2* subdirectory of your G2 installation directory and enter the command *startServer*.

The main window of the G2 server process appears:



On Windows, the G2 server icon appears in the system tray: . You can connect Telewindows to the server and shut down the server using the popup menu on this icon.

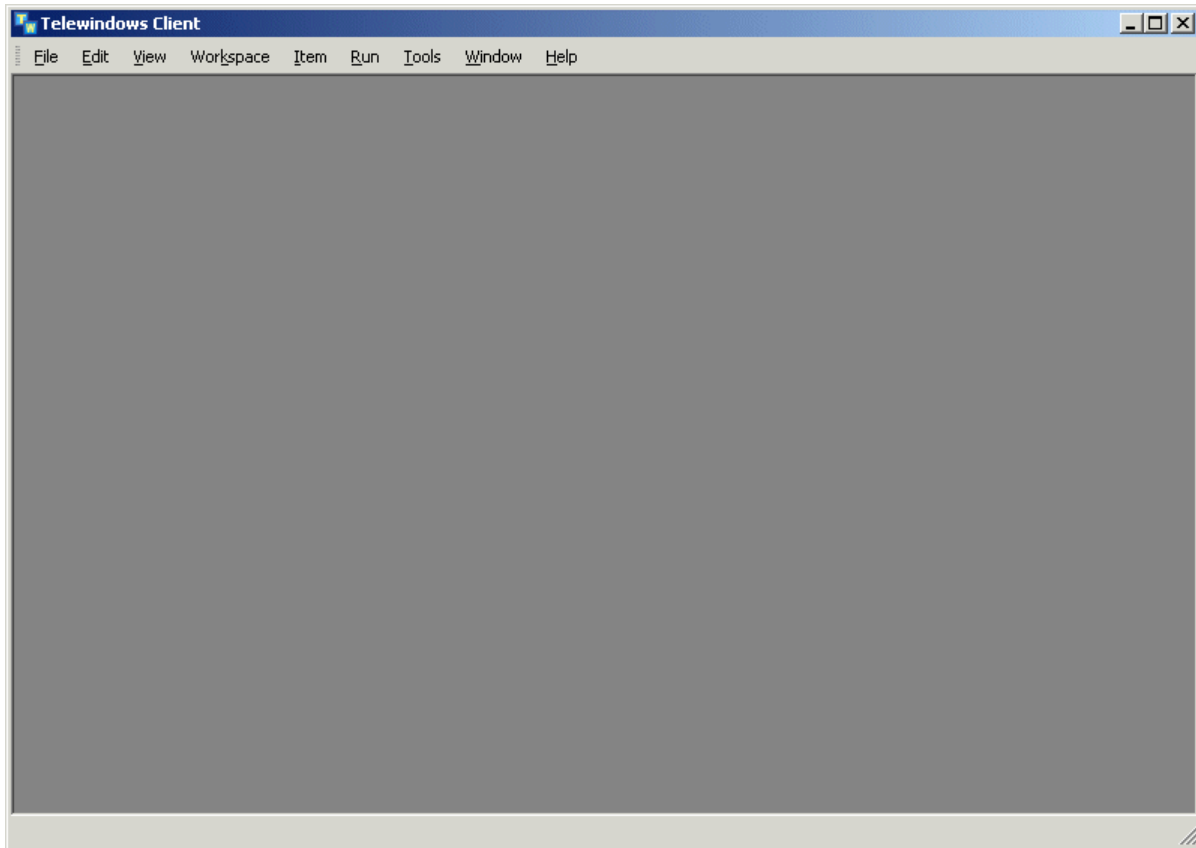
To connect Telewindows Next Generation to G2 on a Windows platform:

- ➔ Choose Start > Programs > Gensym G2 2011 > Telewindows Next Generation.

To connect Telewindows to G2 on a UNIX platform:

- ➔ In a UNIX shell, change to the *g2* subdirectory of your G2 installation directory and enter the command *tw*.

The Telewindows client window appears:



Working with Knowledge Bases

In this lesson, you will learn how to:

- Load a G2 application.
- Use the basic G2 menus.
- Delete the Operator Logbook.
- Show the module hierarchy.

What is a Knowledge Base?

G2 applications are stored in knowledge bases. A **knowledge base**, or **KB**, is an ASCII file with a `.kb` extension that contains all the information your application needs to run. G2 **applications** can have a single file or many files.

Knowledge comes in many forms in G2:

- **Objects** represent the physical systems in your application and the connections between them.
- **Definitions** describe the common features of the objects.
- **Rules, methods, and procedures** describe the behavior of the objects in the real-time environment.
- **Graphical user interface (GUI)** components enable end users to interact with the application.

What are Modules?

G2 applications typically consist of numerous KB files, each of which contains one or more modules. A **module** is a set of related information contained in the KB. For example, an application might have two modules, one for the object definitions, expert system rules, and procedures that describe behaviors, and another for the graphical user interface components.



Use modules to organize your application and reuse existing knowledge across G2 applications.

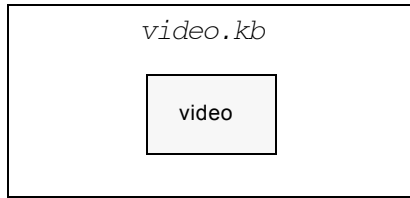
Depending on the needs of the application, some modules can be **independent modules**, which means they can run without any other KB files in the application, while other modules are **dependent modules**, which means they require additional information contained in one or more other modules to run.

G2 represents the modules of an application in a **module hierarchy** to show the module dependencies. The module at the top of the hierarchy is called the **top-level module**. If the application has lower-level modules, the top-level module is necessarily a dependent module. Modules at the bottom of the module hierarchy are necessarily independent modules, because they do not depend on any other module in the hierarchy.

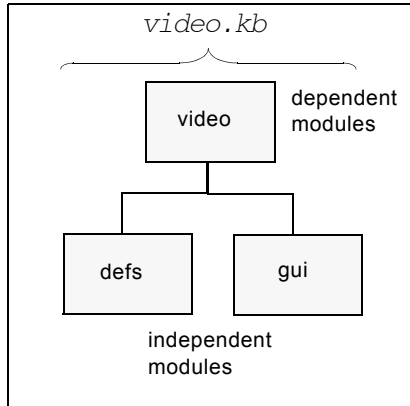
You typically name the KB file so that it corresponds to the name of the top-level module in the KB.

You use the G2 Foundation Resource (GFR) module to help manage your modules in an application, for example, which modules G2 initializes first and which modules determine the overall behavior of the application.

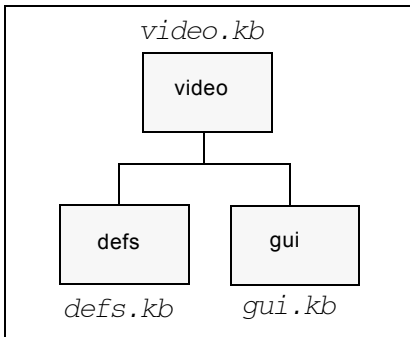
This figure shows several different module configurations of a video conferencing application:



Video application consists of a single module and a single KB file.



Video application consists of three modules, all contained in a single KB file, *video.kb*. The KB consists of one dependent module, **video**, and two independent modules, **defs** and **gui**.



Video application consists of three modules, each of which is contained in its own KB file. The KB file names correspond to the module names.

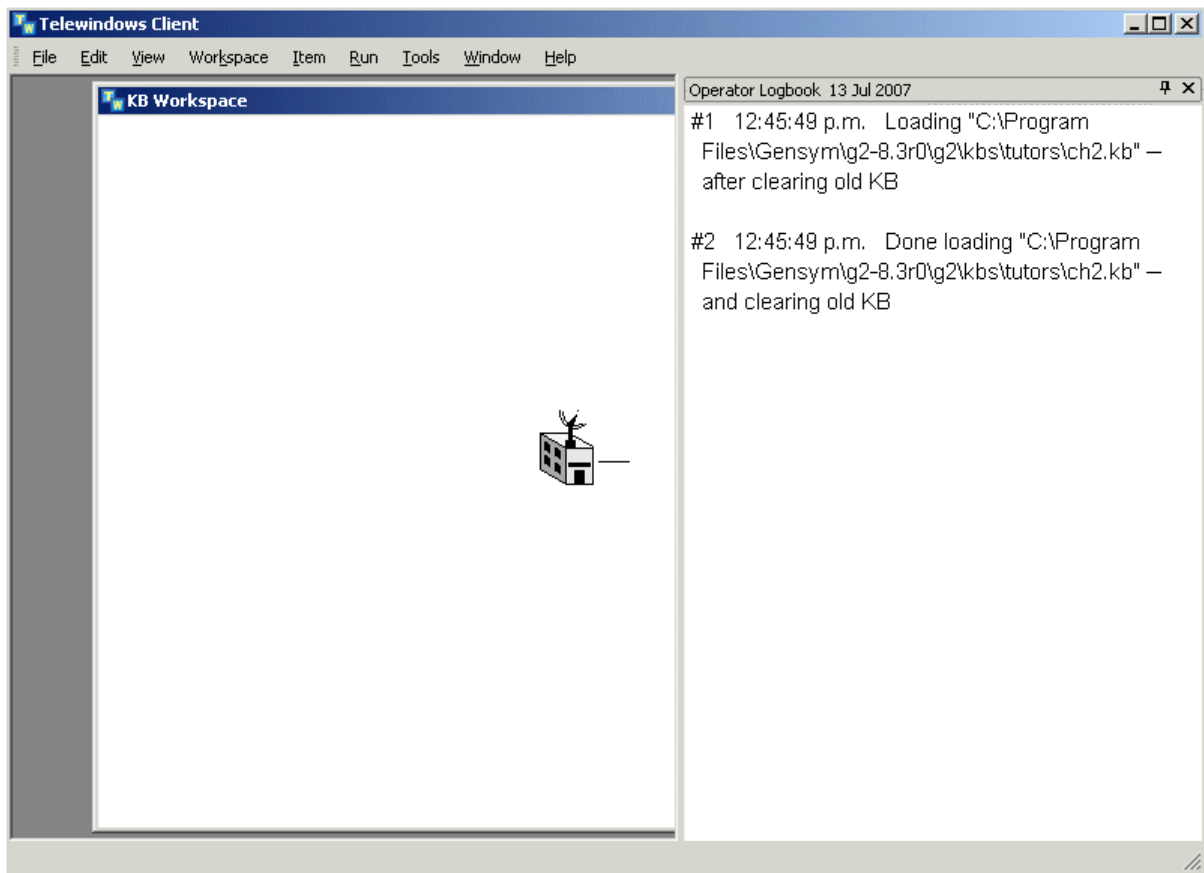
Loading a Sample Application

In this lesson, you will load a sample knowledge base that defines a single module called *basic-skills*. This module contains a single object representation of an office site. The definition of the object is not immediately visible.

To load the sample application:

- 1 Choose File > Load KB.
- 2 Navigate to the `\g2\kbs\tutors` directory and open *ch2.kb*.

When G2 finishes loading the KB, Telewindows has these contents:



The KB Workspace window is a **workspace** where you create objects. The icon on the workspace is an **object**, which is a graphical representation of a piece of knowledge. The area behind the workspace is the **background** on which you create workspaces. The area in the background at the top right is another workspace called the **Operator Logbook** where G2 displays system messages.

Supporting Knowledge Bases

These tutorials come with a set of knowledge bases, which provide the starting point for developing the video conferencing application, as well as the solutions to the tutorials contained in each chapter.

The supporting KB files are:

This file...	Contains...
<i>ch2.kb</i>	The starting point for the video conferencing application, which you load at the beginning of Basic Skills .
<i>ch2soln.kb</i>	The solution to the Basic Skills tutorial.
<i>ch3.kb</i>	The starting point for the tutorial located in Creating a Schematic Diagram .
<i>ch3soln.kb</i>	The solution to the Creating a Schematic Diagram tutorial.
<i>ch4.kb</i>	The starting point for the tutorial located in Building a Knowledge Base .
<i>ch4soln.kb</i>	The solution to the tutorial located in Building a Knowledge Base .
<i>ch5.kb</i>	The starting point for the tutorial located in Building a User Interface .
<i>solution.kb</i>	The solution to the tutorial located in Building a User Interface . This KB provides a complete solution for the entire set of Getting Started tutorials.

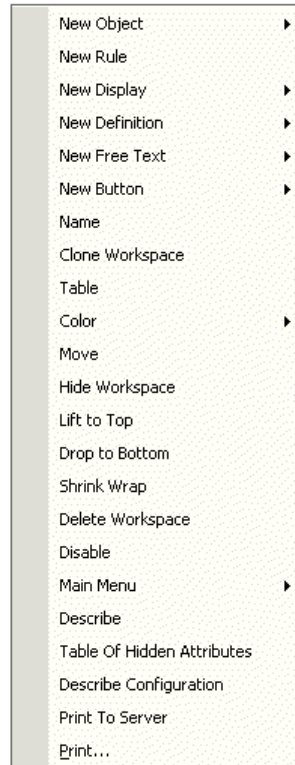
Throughout the tutorials, you are asked to save your work in a *.kb* file that corresponds to the file names above, with your initials appended to them. You can use your KB as the starting point for each subsequent tutorial, or you can use the solution KBs as the starting points.

Displaying the KB Workspace Menu

G2 has a number of different menus that allow you to perform a variety of operations. You have already seen the Main Menu, which is the control center for the overall knowledge base. Each workspace also has a menu, called the **KB Workspace menu**, which you will learn about in more detail later in this tutorial. Each object also has a menu, which you will also learn about later. For now, you will simply display the KB Workspace menu.

To display the KB Workspace menu:

➔ Right-click the workspace background.



You use the KB Workspace menu to:

- Create objects of different types, including rules, procedures, and user interface objects.
- Create definitions of objects.
- Manipulate the workspace itself, for example, hide it, move it, print it, and delete it.
- Specify information about the workspace, such as its name and its visual representation.

Hiding the Operator Logbook

The Operator Logbook is where G2 displays system messages, such as information about saving and loading files. It consists of one or more Logbook pages. In general, you do not need to have the Logbook pages showing during KB development. If G2 needs to display a message to the operator, it will automatically show the Operator Logbook.

For information about the Operator Logbook, see [Runtime Errors](#).

To hide the Operator Logbook

→ Click the close button on the Operator Logbook page.

Viewing the Modules

Each workspace in the application must be assigned to a specific module. You manually assign certain workspaces to a particular module, while G2 automatically assigns other workspaces to a default module.

You use the **Inspect facility** to view the modules in an application. The Inspect facility allows you to locate objects in the knowledge base and display information about its contents.

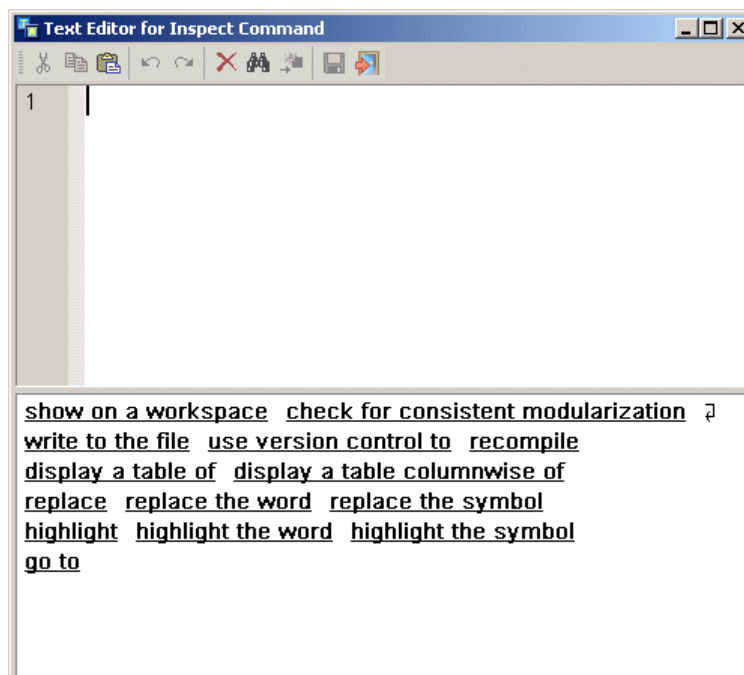
In the Basic Skills application, the workspace you see is assigned to the basic-skills module.

To show the module hierarchy:

- 1 Choose Tools > Inspect.

Tip Many menu choices are also available from the G2 Main Menu, which you access by right-clicking the background of the G2 window.

G2 displays the text editor for the Inspect facility:



The prompts at the bottom of the editor show the commands you can enter in the **syntax-guided text editor**. You enter commands by clicking on the prompts at the bottom of the Inspect workspace. Each time you click on a prompt, G2 updates the prompts to show the next set of options.

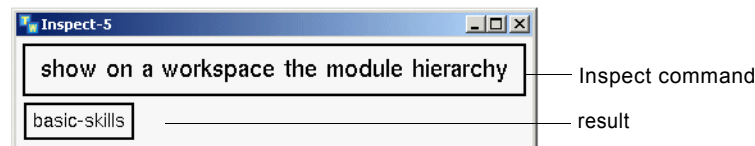
- Click on the phrases at the bottom of the editor to enter this command:

show on a workspace the module hierarchy

First click **show on a workspace**; then click the **module hierarchy**.

- When you have finished entering the command, press Ctrl+Enter or click the Save and Exit button in the toolbar.

G2 displays the module hierarchy for the Basic Skills application on a temporary workspace:



In this case, the module hierarchy consists of a single module named **basic-skills**.

- Delete the temporary workspace.

Summary

In this lesson, you learned how to:

- Display the G2 Main Menu.
- Use the Load KB command to load a knowledge base.
- Display the KB Workspace menu and object menu.
- Delete the Operator Logbook workspace.
- Use the Inspect command to show all the modules in the knowledge base.

Interacting with Objects

In this lesson, you will learn how to:

- Display an object's menu.
- Name an object, using the G2 text editor.
- Make a copy of an object.
- Delete an object.

What is an Object?

G2 is an **object-oriented** development environment. This means that G2 represents knowledge as objects in the application. However, the **item** class is the root of the G2 class hierarchy that you can customize to capture your domain knowledge, and **object** is a subclass in the **item** class hierarchy.

In keeping with object-oriented terminology, this document uses the term **object** to refer to all entities that have a graphical representation, and that includes kb-workspaces and the entities that are on kb-workspaces. To make the class of an object, and its specific instances clear, this documentation uses a sans serif type face to refer to them. For instance: **office-1** is an instance of the **office** class.

An **object** is a piece of information that contains all related knowledge in one location. An object contains all the data that defines the object and all the operations that the object can perform. In object-oriented terms, an object's data are called its **attributes**, and an object's operations are called its **methods**.

An object has particular attributes based on its type. In object-oriented terms, an object's type is called its **class**. For example, the attributes of a video conferencing office might be its location, network type, number of connected sites, and connection status.

Using object-oriented techniques, you can create classes of objects that share characteristics with other classes. This technique is called **inheritance**, where the **subclass** of an object inherits the characteristics of its **superior classes**, including all of its attributes and methods. In the subclass, you describe only the unique features of the class. You will learn more about inheritance later.

A G2 application describes the behaviors of its objects, reasons about those objects, and provides expertise about those objects in a real-time environment. For example, in a video conferencing application, you might describe the behaviors of an office when it is actively connected to another office, and you might reason about whether the office is over budget while it is connected in real time.

Each object in the knowledge base has an **icon** representation. This means you can use objects to communicate information graphically to the end user, for example, by animating the icon to reflect its status. You can create your own icon representation of an object, or you can use one of the many available icons in the G2 icon library.

In G2, almost all knowledge is represented as an object, including:

- The physical systems.
- The connections between systems.
- The rules and procedures that describe the behavior of the systems.
- The workspaces on which objects exist.
- The graphical user interface elements of the application.

Displaying the Object Menu

You interact with the office object by using its menu, called the **object menu**. The object menu lets you perform various operations on the object, including naming, cloning (copying), and deleting the object.

Almost any interactive operation that you can perform on an object by using its menu, you can also perform **programmatically** by using G2's real-time procedural language. For example, you might create a rule or procedure that automatically deletes an object from the knowledge base under certain conditions.

To display the object menu:

→ Right-click the office object.

The popup menu for the object appears:



You can perform various operations on the office, including:

- Displaying a table of attributes for the object.
- Naming the object.
- Cloning the object, which means making an exact copy of the object.
- Deleting the object.
- Transferring the object to another workspace.
- Specifying the characteristics of the object in its table.

Subworkspaces represent a powerful way of organizing knowledge in your application. They also allow you to present different levels of detail to end users

of the application. You will learn more about subworkspaces in [Building a User Interface](#).

Naming an Object

You can refer to objects directly by their **name**, or you can refer to them generically by their class. You name an object to identify it uniquely among other objects in the same class.

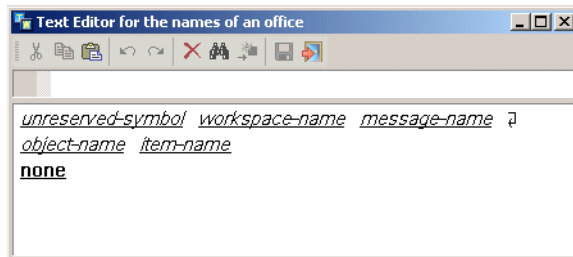
The name of any object in G2 must be a **symbol**, which is a string of alphanumeric characters with no spaces. The G2 convention is to use a dash in place of a space in all names, for example, **office-1**. Symbols are not case-sensitive.

Note All names in G2 must be unique. Thus, you cannot use the same symbol for any two objects in G2.

To name an object:

- 1 Right-click the office object to display its menu.
- 2 Choose name.

G2 displays the single-line text editor for entering the name of the object:



Similar to the text editor for the Inspect facility, this text editor indicates the type of information you can enter to name the object.

Notice the word **none** at the bottom of the editor. This is the default name.

Note Whenever you do not want to enter any value in the text editor, use the word **none** or cancel the text editor.

- 3 Enter **office-1** as the name.

You can use the Backspace key to delete characters if you make a mistake. You can also use the arrow keys to move the cursor.

- 4 Press Enter or click the Save and Exit button to accept the name.

G2 displays the object name below the icon:



Object names always appear in all upper-case letters in G2. However, we typically refer to them informally in lower case, for example, **office-1**.

Cloning an Object

You can interactively create copies of any G2 object by cloning the object. To **clone** an object means to create a duplicate of the original object except for the name and its uuid (universal identifier) attribute. You can also clone objects programmatically.



Clone objects as a way of reusing large pieces of knowledge throughout your application.

To clone an object:

- 1 Display the object menu and choose **clone**.
The cloned object is now attached to your mouse pointer.
- 2 Move the mouse to the desired location and click to place the object on the workspace.

Typically, you name cloned objects to distinguish them in the KB. You identify cloned objects by their class, as well as by their name.

To identify each object:

- 1 Name the cloned object **office-2**.
- 2 Display the object menu for **office-1** and then for **office-2**.
Notice that the object menu shows the name of the object class, which is **office**, as opposed to its specific name.
- 3 Choose **table** from each object's menu to display each object's attribute table.
Each object has an associated table of attribute names and values, which you will learn more about later. For now, notice that the title of each object's table indicates the name of the object, as well as its class.
- 4 Hide each object's table by clicking on the title of the table.

Deleting an Object

You might need to delete an object in a knowledge base. Deleting an object removes it permanently from the knowledge base. Again, you can delete objects interactively, using a menu, or programmatically.

Note There is no undo operation for deleting objects.

To delete an object:

- 1 Choose **delete** from the object's menu.
- 2 Click OK to confirm the deletion.

Summary

In this lesson, you learned how to use:

- The **name** menu choice to name an object.
- The **clone** menu choice to make a copy of an object.
- The **delete** menu choice to delete an object.

Interacting with Workspaces

In this lesson, you will learn how to:

- Display the menu for a workspace.
- Name a workspace.
- Show and hide a workspace.
- Shrink wrap a workspace.
- Move objects on a workspace.
- Operate on a group of objects on a workspace.
- Clone a workspace.
- Delete a workspace.
- Create a new workspace.
- Move a workspace.
- Lift and drop a workspace.
- Use keyboard commands on workspace.
- Display text on a workspace.

What is a Workspace?

Most G2 objects are located on a workspace. A **workspace** is an area of the knowledge base that contains other objects. A workspace is itself an object, which means you can reason about it in the same way you can reason about other objects. You use workspaces for two major purposes:

- To organize knowledge in your application.
- To display an end user interface for your application.

Workspaces represent a convenient way of storing and displaying information in your knowledge base. Applications often display and animate workspaces to communicate information to end users about the application. You typically use names and color to identify workspaces in a knowledge base.

You use workspaces for other purposes in G2, as well. For example, when you load a KB and when you enter the name of an object, you are interacting with special system-defined workspaces.

The icon representation of any object must reside on a workspace. However, objects do not always have to reside on a workspace to exist in the knowledge base or to be reasoned about. You will learn more about this in a later tutorial when you learn about transient knowledge.

Using the KB Workspace Menu

You manipulate workspaces by using the **KB Workspace menu**. You can also manipulate workspaces by using keyboard commands, for example, to scale or move a workspace.

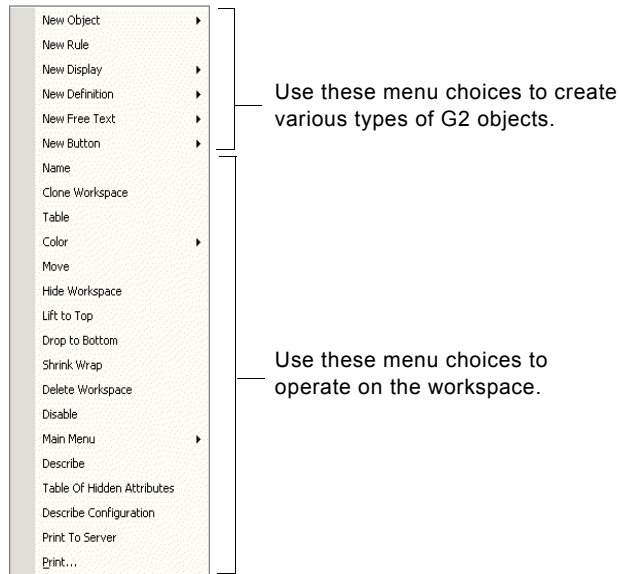
Because most objects in G2 are located on a workspace, you also use the KB Workspace menu to create different types of G2 objects. For example, you use the KB Workspace menu to create definitions of subclasses of objects.

To display the KB Workspace menu:

- ➔ Right-click the workspace background.

G2 displays the KB Workspace at the location of the mouse.

Here is the KB Workspace menu, with labels that describe the two basic categories of menu choices:



Naming a Workspace

You name a workspace to identify it in the knowledge base and to reason about it programmatically. The workspace name also provides a convenient label for the workspace. Finally, you name a workspace so that you can hide it and then show it again, using a menu choice.

Note You cannot show an unnamed workspace directly from a menu choice.



Name only a small number of workspaces in your application and use named workspaces and subworkspace hierarchies to navigate to other workspaces in the application.

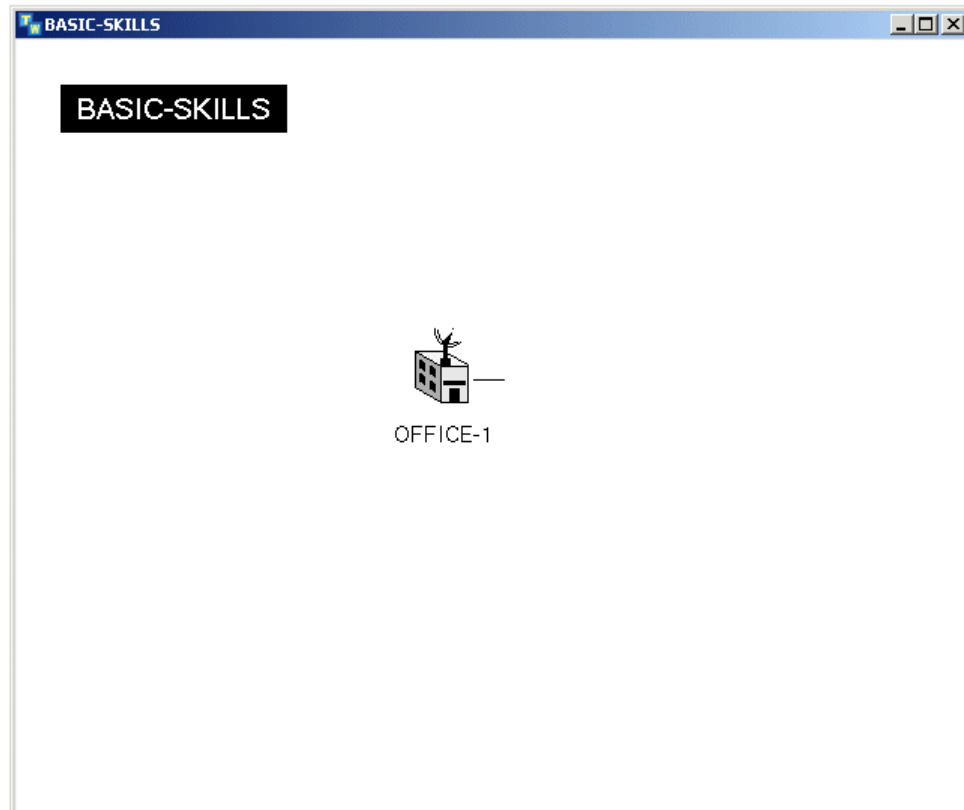
Remember, workspace names must be unique symbols, which means you must use hyphens in place of spaces.

To name a workspace:

- 1 Display the KB Workspace menu for the workspace that contains the office and choose Name.
- 2 Enter **basic-skills** as the name of the workspace in the text editor that appears.

The name appears on the workspace in all capital letters: BASIC-SKILLS. We often refer to workspace names as proper names, for example, the Basic Skills workspace.

Here is the Basic Skills workspace:



Hiding and Showing a Workspace

G2 applications typically contain numerous workspaces, some of which might be visible and others of which might be hidden at any one time.

To hide a workspace:

- ➔ Choose Hide Workspace from the KB Workspace menu or click the close button in the upper-right corner of the window.

The workspace disappears from view.

You can also iconify the workspace by clicking the iconify button, in which case the workspace window is iconified at the bottom of the window.

To show a named workspace:

- 1 From the top-level menu, choose Workspace > Get Workspace.

You can also choose Get Workspace from the G2 Main Menu.

A list of all named workspaces in the KB appears, including user-defined and system-defined workspaces. For example, the Message Board is a system-defined workspace that displays operator messages.

- 2 Choose `basic-skills` from the list of named workspaces to show it.

Shrink Wrapping a Workspace

You can adjust the borders of a workspace to minimize its size by **shrink wrapping** the workspace. Shrink wrapping a workspace is a convenient way of saving space in your G2 window.

To shrink wrap a workspace:

- ➔ Choose Workspace > Shrink Wrap.

Tip Most menu choices in the Workspace menu are also available in the KB Workspace menu, which you access by right-clicking the workspace background. When using the top-level Workspace menu, you must first select the workspace on which you want to perform the action.

The edges of the workspace adjust to contain just the office and the workspace name.

Moving Objects on a Workspace

You move objects on a workspace by dragging. You can also move objects on a workspace programmatically.

To **drag** an object means to place the mouse over an object, hold down the mouse button while moving the object to a new location on the workspace, then lift the mouse button to place the object on the workspace.

When you drag an object on a workspace, the workspace borders automatically adjust to accommodate the object.

To move an object on a workspace:

- ➔ Drag the office down and to the right on the workspace, then move it back to the middle of the workspace.

The workspace borders adjust to accommodate the object so that the workspace looks like it did before you shrink wrapped it. If you drag the object beyond the borders of the workspace, the workspace window displays scroll bars.

You can drag any object on a workspace in this way, including the workspace title, to expand the workspace borders.

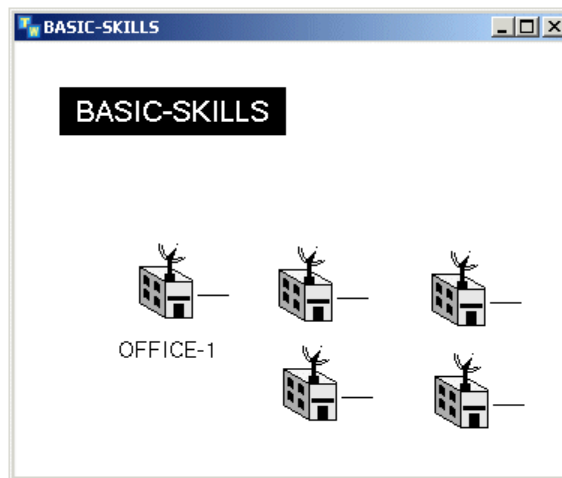
Operating on a Group of Objects on a Workspace

You might want to operate on a group of objects on a workspace, for example, to move them, clone them, transfer them, delete them, align them, or distribute them.

To operate on a group of objects:

- 1 Create several instances of the office class on the workspace by cloning the existing office.

For example:



- 2 Drag the mouse cursor to select the group of object in the rectangle, or click an object to select it, then use SHIFT + click to add or remove objects to or from the selection.
- 3 Right-click one of the selected objects to display a popup menu for the selection, choose Clone to attach the group of objects to the mouse, then move the cloned group of objects to a new location on the workspace and click to place the objects on the workspace:
The cloned objects are still selected as a group.
- 4 Drag the cloned objects to a new location and click to place them.
- 5 With two or more objects selected, choose Align on one of the selected objects, then choose Top to align the tops of the objects.
- 6 With three or more objects selected, choose Distribute on one of the selected objects, then choose Horizontally to distribute the objects horizontally.

- 7 Choose Delete on one of the objects to delete the group of objects and click OK to confirm.

Note Be sure to leave the original office on the workspace.

- 8 Shrink wrap the workspace and adjust the borders so that the office is in the middle of the workspace again.

Cloning a Workspace

Similar to the way you clone an object, you can clone an entire workspace interactively or programmatically. The cloned workspace contains duplicates of everything located on the original workspace. The cloned workspace is unnamed, as are the cloned objects, because G2 requires that the names of all objects and workspaces be unique.



Clone workspaces to save time when you develop applications. Cloned workspaces provide a starting point for describing new knowledge based on existing knowledge.

To clone a workspace:

- 1 With the workspace selected, choose Workspace > Clone Workspace.
G2 creates an exact duplicate of the Basic Skills workspace and places it next to the existing workspace. Notice that neither the workspace nor the office has a name.
- 2 Choose Workspace > Get Workspace to verify that the cloned workspace is not in the list of named workspaces.
- 3 Name the cloned workspace **clone**.
- 4 Hide the cloned workspace.
The Basic Skills workspace is still visible.
- 5 Show the **clone** workspace, which now appears in the list of named workspaces.

Deleting a Workspace

Just as you might need to delete an object, you might need to delete a workspace, either interactively or programmatically. Deleting a workspace permanently deletes the workspace and all its contents from the knowledge base.

To delete a cloned workspace:

- 1 With the workspace selected, choose Workspace > Delete Workspace on the cloned workspace to delete it.

G2 always asks for confirmation before it deletes a workspaces that contains objects.

- 2 Click OK to confirm the deletion.

Creating a New Workspace

You can create new workspaces interactively or programmatically.

To create a new workspace:

- 1 Choose Workspace > New Workspace.

The New Workspace menu choice is also available in the G2 Main Menu.

G2 creates a new workspace and places it in the center of the G2 window.

- 2 Name the new workspace my-workspace.

Moving a Workspace

If a workspace is visible, you can move it to a new location in the G2 window by dragging it with the mouse.

To move a workspace by dragging:

- ➔ Drag My Workspace up and to the right so that it partially covers the Basic Skills workspace.

Lifting and Dropping Workspaces

You often have many workspaces stacked on top of each other in the G2 window. You can bring a workspace to the foreground or drop a workspace to the bottom of the stack to make other workspaces visible.

To lift a workspace to the top:

- 1 Click anywhere in the Basic Skills workspace, including the title bar.
- 2 Repeat this operation to lift My Workspace to the top.

Sometimes you don't know what workspaces exist at the bottom of the stack. In this case, you can drop the top workspace to the bottom of the stack to reveal the lower workspaces.

To drop a workspace to the bottom:

- 1 Drag My Workspace so that it completely covers the Basic Skills workspace.
- 2 Position your mouse pointer over My Workspace.
- 3 Choose Workspace > Drop to Bottom to expose the Basic Skills workspace.

Using Keyboard Commands on Workspaces

You can perform many operations on workspaces by using keyboard commands, including some operations that are not available on the KB Workspace menu.

For example, you can lift workspaces to the top and drop them to the bottom, using keyboard commands. You can also move workspaces incrementally, as well as shrink and grow workspaces incrementally, using keyboard commands.

To lift and drop workspaces by using keyboard commands:

- 1 Drag the workspaces so that they are partially overlapping and both are visible.
- 2 Position the mouse pointer over each workspace and enter **Ctrl + t** to **lift** each workspace to the top of the stack.
- 3 Position the mouse pointer over each workspace and enter **Ctrl + v** to **drop** each workspace to the bottom of the stack.
- 4 Position the mouse pointer over the top workspace and enter **Ctrl + i** to circulate the top workspace to the **bottom** of the stack.
- 5 Position the mouse pointer over the top workspace and enter **Ctrl + p** to circulate the bottom workspace to the **top** of the stack.

Note Circulating two workspaces has the same effect as lifting and dropping two workspaces. However, circulating more than two workspaces cycles through the stack, lifting or dropping each workspace, one level at a time.

To move a workspace by using keyboard commands:

- 1 Position the mouse pointer over a workspace and enter **Ctrl + r** several times to move the workspace incrementally to the **right**.
- 2 Position the mouse pointer over the same workspace and enter **Ctrl + l** several times to move the workspace incrementally to the **left**.
- 3 Position the mouse pointer over a workspace and enter **Ctrl + u** several times to move the workspace incrementally **up**.
- 4 Position the mouse pointer over the same workspace and enter **Ctrl + d** several times to move the workspace incrementally **down**.

To adjust the size of a workspace by using keyboard commands:

- 1 Position the mouse pointer over a workspace and enter **Ctrl + s** several times to make the workspace incrementally **smaller**.
- 2 Position the mouse pointer over the same workspace and enter **Ctrl + b** several times to make the workspace incrementally **bigger**.
- 3 Position the mouse pointer over a workspace and enter **Ctrl + q** to shrink the workspace to **one-quarter** its size.
- 4 Position the mouse pointer over the same workspace and enter **Ctrl + f** to display the workspace at **full size**.
- 5 Position the mouse pointer over a workspace and enter **Ctrl + n** several times to **narrow** the workspace incrementally.
- 6 Position the mouse pointer over the same workspace and enter **Ctrl + w** several times to **widen** the workspace incrementally.

To display help for keyboard commands that manipulate workspaces:

- 1 Position your mouse pointer over a workspace and enter **Ctrl + ?**.

G2 displays a help screen for all keyboard commands that operate on workspaces:

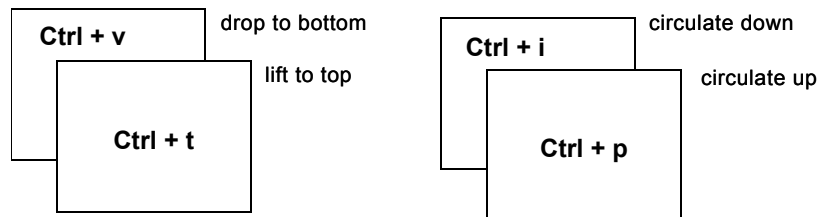
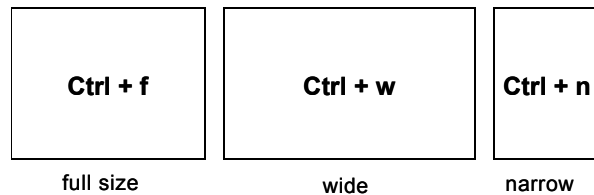
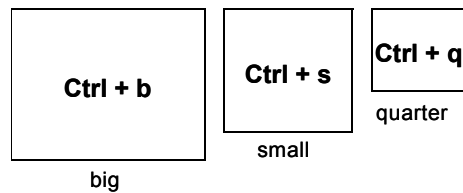
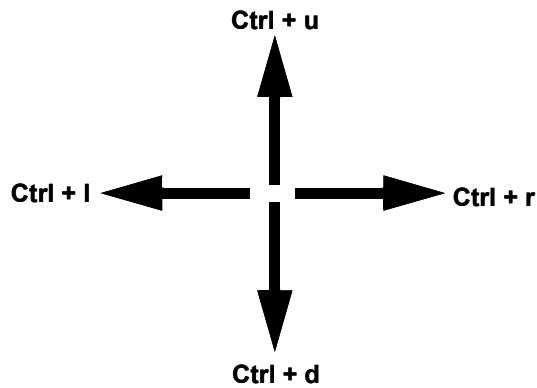
Standard Keyboard Shortcuts			
F1	Help Topics	F5	Refresh
Tab	Select next item	Shift+Tab	Select previous item
Control+Tab	Select next workspace	Control+Shift+Tab	Select previous workspace
Control+F4	Hide selected workspace	Pause	Pause
Control+G	Toggle visible grid	Alt+Return	Table
Menu or Shift+F10	Menu	Return	Default action
Space	Default action	Delete	Delete
Control+A	Select all	Escape	Deselect all or dismiss
Control+O	Load KB	Control++ or Control+=	Zoom in
Control+- or Control+_	Zoom out	Control+0	Full scale
Left-arrow	Move selection left	Right-arrow	Move selection right
Up-arrow	Move selection up	Down-arrow	Move selection down
Control+Left-arrow	Nudge selection left	Control+Right-arrow	Nudge selection right
Control+Up-arrow	Nudge selection up	Control+Down-arrow	Nudge selection down
Classic Keyboard Shortcuts			
Control+F	Full scale	Alt+F	Normalized full scale
Control+T	Lift to Top	Control+V	Drop to Bottom
Control+C	Refresh	Control+P	Circulate up
Control+I	Circulate down	Control+L	Shift left ten percent
Control+R	Shift right ten percent	Control+U or Mouse-wheel-backward	Shift up ten percent
Control+D or Mouse-wheel-forward	Shift down ten percent	Alt+L	Shift left one percent
Alt+U	Shift up one percent	Alt+R	Shift right one percent
Alt+D	Shift down one percent	Control+.	Scale to fit
Alt+.	Maximum scale to fit	Control+S or Control+Mouse-wheel-backward	Twenty percent smaller
Control+B or Control+Mouse-wheel-forward	Twenty percent bigger	Control+N	Twenty percent narrower
Control+W	Twenty percent wider	Control+Q	One quarter the scale
Control+4	Four times the scale	Control+Z	Pause
Control+?, Help or Control+/	Help		

2 Hide the help screen by clicking on the background and choosing Delete Workspace.

3 Now, delete My Workspace.

Here is a summary of the keyboard commands for workspaces:

Keyboard Commands for Workspaces



Displaying Text on a Workspace

You often need to label parts of a workspace to provide documentation for developers, as well as end users. One way is to use **free text** to label information on a workspace. You can change the color of the free text to make it more attractive.

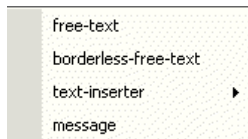
In a real application, you do not typically hard-wire text into an application, especially if the application is going to be translated into another language for local use. Instead, you use the G2 Foundation Resources (GFR) to create text keys, which are symbols that G2 substitutes with actual text when the KB runs. You can use this feature to translate text and messages into other languages, as well as to perform dynamic text substitutions at run time.

For more information on GFR, see the *G2 Foundation Resources User's Guide*.

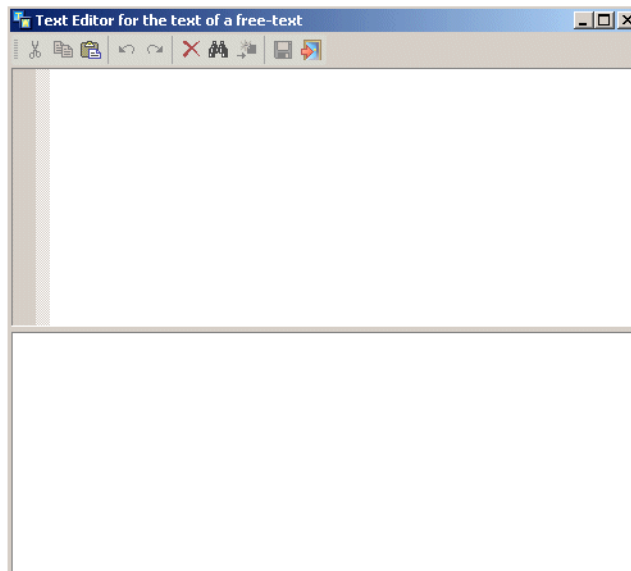
To display free text on a workspace:

- 1 Choose Workspace > New Free Text on the Basic Skills workspace to display a menu of free text options.

You can choose to create text with a border or without by choosing either **free-text** or **borderless-free-text**:



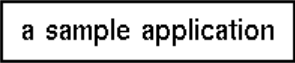
- 2 Choose **free-text** to display the text editor:



- 3 Enter the following text in the text editor: **a sample application**.
- 4 Press Ctrl + Enter or click the Save and Exit command to attach the text to the mouse pointer.

Tip You press Enter to accept text in the single-line text editor, and you press Ctrl + Enter to accept text in the normal text editor.

- 5 Position the text at the top-right of the workspace and click to place it.
By default, free text has a black border, black text, and a white background:



a sample application

You can change the color of any part of the free text to make it more attractive.

To change the colors of free text:

- 1 Right-click the border of the free text and choose **color**.

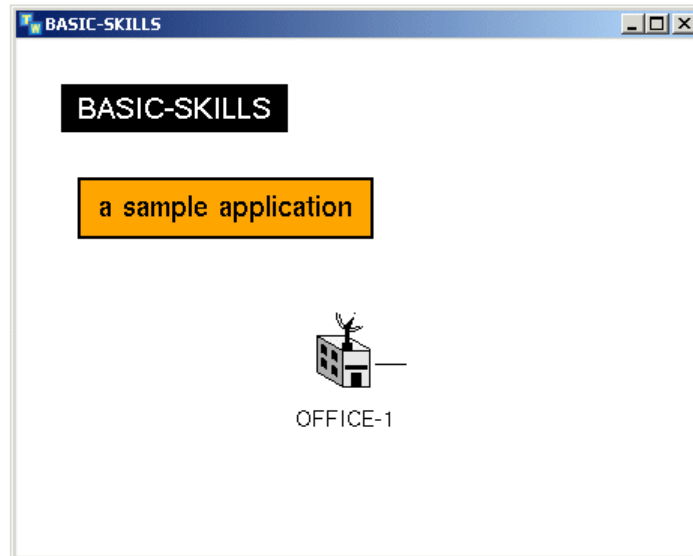
Hint If you click too close to the text, you will display the text editor for editing the text. Cancel the editor and try again.

You can change the background, border, or text color of the text.

- 2 Change each of these colors for the free text by clicking the desired menu choice, then choosing a color.

To display all the available colors in the G2 color palette, click **More** on the palette.

The workspace now looks something like this:



Summary

In this lesson, you learned how to:

- Use the Name command to name a workspace.
- Use the Hide Workspace command to hide a workspace.
- Use the Get Workspace command to display a named workspace.
- Use the Shrink Wrap command to adjust the borders of a workspace to fit the objects.
- Move objects on a workspace by dragging to adjust the borders of the workspace automatically.
- Perform various operations on a group of objects on a workspace.
- Use the Clone Workspace command to make a copy of a workspace and all its contents.
- Use the Delete Workspace command to delete a workspace.
- Use the New Workspace command to create a new workspace.
- Move a workspace by dragging.
- Lift a workspace to the top and drop it to the bottom.

- Use various keyboard commands on a workspace to lift and drop it, shrink and grow it, and so on, and display the help menu for workspace commands.
- Use the New Free Text command to display text on a workspace.

Connecting Objects

In this lesson, you will learn how to:

- Connect objects together, using connections and connection stubs.
- Delete connections between objects.

What is a Connection?

A **connection** is a graphical link between two objects. You use connections in a G2 application to represent visually how objects are related. For example, in a video conferencing application, you might use connections to show which offices have video conferencing capability with other offices. In an application that monitors the flow of a liquid through a tank, you might use connections to represent the physical pipes between the tanks.



Use connections to reason about the objects they are connecting.

For example, in a video conferencing application, you could determine the number of connected offices by counting the graphical connections for any one office. In an application that monitors the flow of liquid in a tank, you could reason about the volume of the liquid in the tank based on the intake and output flow through the connections.

A connection is a G2 object, just like the objects it is connecting, which means it has certain characteristics and behaviors. Because connections are objects, they can also contain information, for example, the current value of the data that is flowing from one object to another. You can also use connections to communicate information to the end user. For example, you might cause the connection between offices to flash when two sites are online with each other, or you might change the color of a connection between two tanks when liquid is flowing from one tank to another.

G2 also supports a related construct called a **relation**, which is a non-graphical “connection” between two or more objects. A relation represents a conceptual relationship between objects, which you can create and delete as part of processing and which you can use to reason about objects. For example, you might create a relation between all the video conferencing offices to enable reasoning about all the offices, although they might not all be physically connected.

Connecting Objects

You will notice that the office has a connection coming out of the right side of the icon. A connection that is attached to one object but is not yet connected to another object is called a **stub**. Certain objects contain connection stubs as part of their definition, in which case the stubs always appears on the icon. Other objects allow you to create stubs interactively.

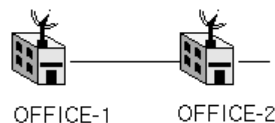
Connections can have a **direction of flow**. When a connection has direction, you can only connect objects in the proper order. Also, you can reason about objects based on the connections that are flowing into or out of an object.

You can create and delete connections interactively and programmatically.

One way of connecting two objects interactively is to drag a connection stub directly into another object.

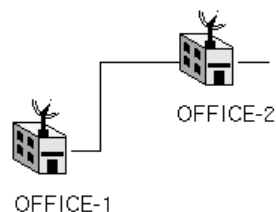
To connect two objects directly:

- 1 Clone **office-1** and place the cloned object directly to the right of the current office.
- 2 Name the new office **office-2**.
- 3 Click the stub from **office-1**, move your mouse pointer until it is directly over the center of the **office-2** icon, and click to connect the two objects:



- 4 To verify that the objects are connected, move one of the connected objects on the workspace.

If the objects are connected, the connection moves with the object, automatically creating bends in the connection:

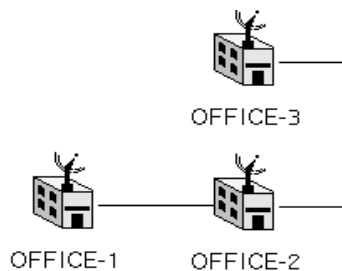


Another way of connecting two objects is to connect a stub directly to the stub of another object.

To connect two objects by connecting stubs:

- 1 Clone **office-1**, name the new office **office-3**, and place the new office directly above **office-2**.
- 2 Click the stub leading out of **office-3** and move the mouse pointer down and to the right.
G2 automatically creates a bend in the connection.
- 3 Click again to create a second bend in the connection and position the stub directly over the stub of **office-2**.
- 4 Click once more to connect the two stubs together.
- 5 Verify that the two offices are connected by dragging one of the offices.

The new diagram looks similar to this:

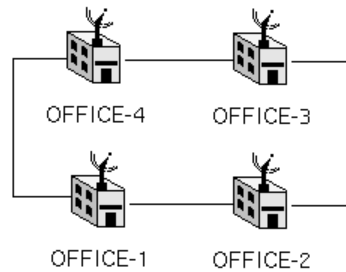


Once an object has connections, you can clone the object to create a new object with default stubs.

To create a new object with default stubs by cloning:

- 1 Clone **office-2** to create **office-4** and place it directly above **office-1** and directly to the left of **office-3**.
Notice that the new office has two default stubs.
- 2 Connect **office-4** to **office-3** by connecting the stub to the object.
- 3 Connect **office-4** to **office-1** by creating two bends in the connection and connecting the two stubs.

The offices should be connected in a closed rectangular pattern similar to this:



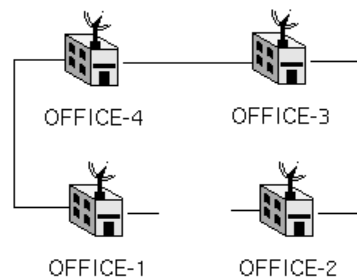
Deleting Connections

You might have to delete the connection between two objects. When you delete two connected objects, the stubs remain on the previously connected objects.

To delete the connection between two objects:

- 1 Click the connection between office-1 and office-2 to display the menu for the connection.
- 2 Choose Delete to delete the connection.

Notice that the connection stubs remain on the icons:

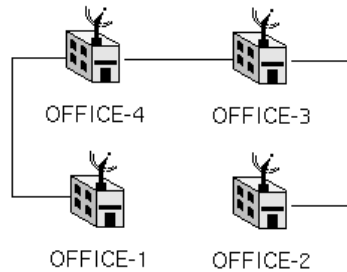


You typically delete unconnected stubs on objects to avoid confusion.

To delete a connection stub:

- 1 Click on the stub leading out of office-1, move the mouse pointer to the middle of the icon, and click to delete the stub.
- 2 Repeat this operation for the stub leading out of office-2.

The schematic should now look like this:



Summary

In this lesson, you learned how to:

- Connect two objects by dragging a connection into another object.
- Connect two objects by dragging a stub into the stub of another object.
- Create bends in a connection by clicking.
- Clone a connected object to create an object with default stubs.
- Delete a connection between two objects.
- Delete stubs on an object.

Editing Attributes in Tables

In this lesson, you will learn how to:

- Edit simple attributes by editing them in the attributes table.
- Show and hide the attribute table.
- Use editing commands in the text editor.
- Edit attributes by cutting, pasting, and deleting text.
- Display an attribute of an object next to the icon.

What are Attributes?

G2 objects have certain characteristics that make them unique. These characteristics are called **attributes**. For example, if the object is an office, the attributes might include the office location and phone number. Every object that is a type of office has these attributes. What make each office unique are the **values** you assign to the attributes of each office.



Use attributes to store a wide variety of knowledge in an application, including data values, as well as other objects.

For example, certain attributes might contain single values, such as address or number of connections. Other attributes might contain a history of values of a particular type, for example, a history of the total cost of the office based on its connections. Still other attributes might contain lists of objects that are related in some way to the object, for example, a list of objects that represent each video conferencing connection. The ability to define a wide variety of types of attributes provides great flexibility in describing object characteristics in G2.

Why do you need attributes in G2? To reason about their values in real time and infer new values under various conditions. For example, you might have an attribute that computes the total cost of a video conferencing office based on various other attributes of the office. You might reason about the total cost to infer whether the site is over budget, thereby providing expertise to an operator responsible for monitoring the video conferencing operations of a business.

What Types of Attributes Are There?

Object in G2 have **system-defined attributes**, which are attributes that G2 defines. For example, a workspace has an attribute called **background-color**, which defines its color. Object that are derived from a user-defined class, for example, the **office** class, can have **user-defined attributes**, which are attributes that you define for the entire class and its subclasses.

User-defined attribute values can be **untyped**, in which case the value is any alpha-numeric sequence of characters, without spaces. Another name for an untyped attribute is a **simple attribute**. The default value of a simple attribute is the symbol **none**.

Attribute values can also be **typed**, in which case G2 validates the type of data when the value is specified, either interactively or programmatically. You specify the type of attribute when you declare the attribute in the class definition.



Always use the most restrictive type possible when you declare attributes in a class. For example, if the value of an attribute is always an integer, declare its type to be an integer, not a quantity, which includes both integers and floating point values. This practice is called **strong typing** and is recommended for optimal performance.

G2 supports these **data types**:

- Integers
- Floating point numbers
- Quantities (integers or floating point numbers)
- Symbols
- Logical values (true or false)
- Fuzzy truth values (truth values with uncertainty)
- Text strings
- Sequences
- Structures

Assigning Values to Attributes

An application assigns values to attributes in a variety of ways, including:

- Supplying values interactively through a table.
- Supplying values programmatically, using procedural statements.
- Inferring values under certain conditions, using rules.
- Specifying default values in the definition of the object class.
- Computing values based on other values.
- Simulating values.
- Obtaining values from external data sources.

You will learn how to assign values, using most of these techniques.

Displaying the Attribute Table

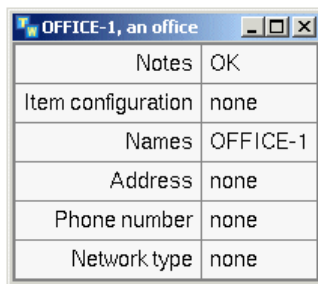
Every object in G2 has an **attribute table** that lists all of its attributes and corresponding values.

To display the attribute table for an object:

- 1 Right-click **office-1** to display its menu and choose **table**.

Tip You can also double-click the object to display its table.

You will see an attribute table that looks like this, where the title of the table indicates the name of the object and its type:



OFFICE-1, an office	
Notes	OK
Item configuration	none
Names	OFFICE-1
Address	none
Phone number	none
Network type	none

Every object contains three standard attributes, as well as any class-specific attributes for the object. The three class-specific attributes have been defined to be symbols, but no default value has been specified. G2, therefore, supplies the symbol **g2** as the default values for these attributes.

The standard attributes include:

- **Notes**, which is where G2 displays system messages and errors about the object.
- **Item configuration**, which you use for configuring how the object behaves in different user modes.
- **Names**, which displays the name of the object.

Attribute names are also symbols, which means they appear in expressions with hyphens in place of spaces, for example, **network-type**.

Note When you reason about attributes in rules and procedural statements, you use the symbolic attribute name, rather than the name of the attribute as it appears in the table.

- 2 Display the table for another office.

Notice that unlike object menus, you can view more than one attribute table at a time. This makes it convenient to compare the attributes of different objects.

You can shrink a table just as you can shrink a workspace to preserve space on the screen.

- 3 Use the Ctrl + s command to shrink one of the attribute tables, then use the Ctrl + f command to make it full size again.

Editing Attributes

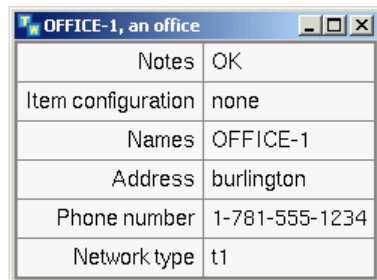
To edit the attributes of an object, you use the G2 **text editor**. Depending on the data type of the attribute, G2 validates the value as you enter it. The attributes of the office are symbolic types, which means you can enter any symbol. If you enter a non-symbolic value, G2 gives you this message in the editor:

This cannot be parsed.

To edit the values of simple attributes:

- 1 Display the table for office-1.
- 2 Click the value of the **address** attribute and enter burlington in the text editor that appears.
- 3 Click the value of the **phone** attribute and enter any phone number.
- 4 Click the value of the **network-type** attribute and enter T1.

The table looks similar to this:



Notes	OK
Item configuration	none
Names	OFFICE-1
Address	burlington
Phone number	1-781-555-1234
Network type	t1

Use the toolbar buttons in the text editor to cut, copy, and paste text, and use the Backspace or Delete key to delete text.

- 5 Hide the attribute table by clicking the close button.
- 6 Enter values for the **address** attribute of the other offices.

Displaying an Attribute Next to an Object

It is often useful to display attribute values next to the icon of an object to provide information about the object. These are called **attribute displays**. You might want to do this for static information, such as the name of the object or its status, or you might want to show information that changes over time, such as the total cost of a video conferencing office or the volume of liquid in a tank.

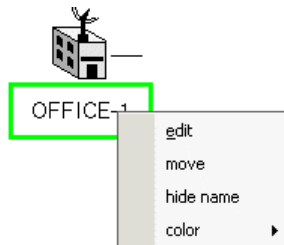
If the attribute is editable, you can edit it directly from the icon rather than through the table. You can also show the name of the attribute with its value as an attribute display.

Note In a real application, you typically only use attribute displays during the development phase; for performance reasons, you hide attribute displays when you deploy an application.

When you are using attribute displays, you often hide the name of the object.

To hide the name of an object:

- 1 Right-click the name of each office to display its menu:

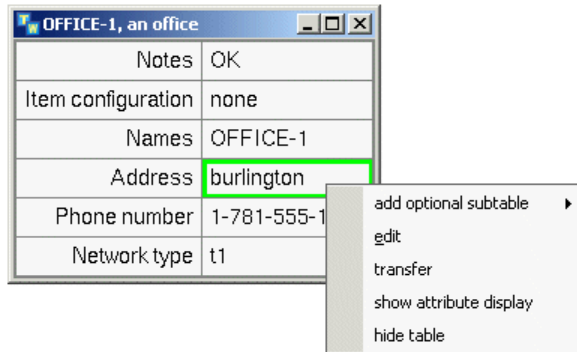


- 2 Choose hide name.

Now you can add an attribute display and move it to below the icon.

To show an attribute next to an object:

- 1 Display the attribute table for office-1.
- 2 Right-click the value of the address attribute to display a menu:



- 3 Choose show attribute display.

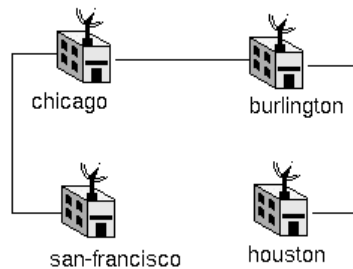
G2 displays the value of the attribute above and to the right of the icon.

- 4 Show the names of all the other offices as attribute displays.

You can move the attribute displays to a new location by dragging them. The attribute displays stay with the icon whenever you move the icon.

- 5 Move the attribute displays to just below each icon by dragging.

The diagram should look something like this:



When your icon has an attribute display, you can edit the attribute directly from the attribute display, rather than through the object's table.

To edit an attribute display directly:

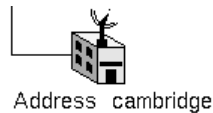
- ➔ Double-click an attribute display to display the text editor and edit the value.

You can add the name of the attribute next to its value.

To add the name of an attribute next to its value:

- 1 Right-click the attribute display to display its menu.
- 2 Choose **add name of attribute** to add the name and center the attribute display below the icon.

The icon looks something like this:



You can also remove the name of the attribute from the attribute display.

To remove the attribute name from a display:

- ➔ Right-click the attribute display and choose **delete name of attribute**, and center the attribute display.

You hide an attribute display by choosing **hide attribute display**.

Summary

In this lesson, you learned how to:

- Use the Table menu choice to display the attribute table for an object.
- Use the text editor to edit the attributes of an object.
- Use the Show Attribute Display menu choice to display and edit attribute values with objects.

Creating a Simple Rule

In this lesson, you will learn how to:

- Use the syntax-guided text editor to create a rule.
- Refer to attributes within a rule.
- Create a simple rule that animates an object when the user moves the object on a workspace.
- Perform simple error handling techniques.

What is a Rule?

The heart of any expert system is its ability to reason about the knowledge it contains. Along with procedures and methods, G2 uses rules to reason about the knowledge contained in the application. A **rule** is a special kind of statement that tests conditions and draws conclusions.

Rules have two parts:

- The first part, called the **antecedent**, tests a condition.
- The second part, called the **consequent**, draws a conclusion.

For example, the antecedent of a rule might be:

if the total-cost of office-1 > 100

which tests to see if the total cost is above a certain threshold. The consequent of this rule might be:

then conclude that the status of office-1 is over-budget

which concludes the value of the status attribute of the office.

When a rule is **invoked**, G2 evaluates the rule by testing the condition in the antecedent to see whether it is true. If the condition is true, G2 executes the actions in the consequent.

Performing Actions in a Rule

A rule can perform any type of action in its consequent, for example:

- Concluding values for attributes.
- Dynamically creating or deleting objects and connections.
- Showing or hiding a workspace.
- Animating icons.
- Invoking specific categories of rules.

You will learn more about actions in [Creating a Schematic Diagram](#).

Using Two Different Inferencing Techniques

When a rule concludes a value for an attribute as a result of testing a condition, we refer to this mechanism as **inferencing**. As a result, the internal G2 mechanism that invokes rules is called the **inference engine**.

G2 provides two basic inferencing mechanisms, using rules:

- **Event-driven processing**, whereby G2 makes inferences by responding to real-time events, for example, moving an object on a workspace, receiving a value from a data source, or failing to receive a value.
- **Data-driven processing**, whereby G2 makes inferences by detecting changes in attribute values of objects, for example, due to the periodic updating of a real-time signal or the user entering a value in an end-user display.

Depending on the needs of the application, you create different types of rules to perform one of these basic types of inferencing.

Choosing Between the Four Basic Types of Rules

Depending on the type of inferencing you want the rule to perform, you use one of four basic types of rules:

- **If rules** perform data-driven processing by testing the condition in the antecedent and taking the actions in the consequent if the condition is true.
- **Whenever rules** perform event-driven processing by detecting the event in the antecedent and taking the actions in the consequent whenever G2 detects the event.
- **Unconditionally rules** perform data-driven or event-driven processing by taking the action in the consequent automatically whenever G2 invokes the rule by whatever means.
- **Initially rules** perform event-driven processing by invoking the rule whenever you start your knowledge base.

In addition to the four basic types of rules, a **when rule** is another variation on a whenever rule, and a **for rule** is the generic form of any of the basic types of rules.

Referring to Attributes in Rules

You typically refer to the attributes of objects in rules. In the example used earlier, you might create a rule that tests to see whether the **total-cost** attribute of the object named **office-1** is above a certain threshold. You will create this rule in a later tutorial.

To refer to this attribute, you use the following English-like construct:

the total-cost of office-1

As you can see, the construct for referring to the attributes of objects in a rule requires two reserved words: **the** and **of**. A **reserved word** is a symbol in the G2 language that you can only use in the context of the G2 statement; you cannot use

a reserved word as the name of an object class or as the name of an attribute. Other reserved words include the various types of rules, such as *if* and *whenever*.

Creating a Rule

To create a rule, you use the **natural language text editor**, a feature of the G2 text editor that prompts you at each stage with the proper syntax as you construct the rule. You use the natural language text editor when you edit any type of object that uses G2's procedural language.

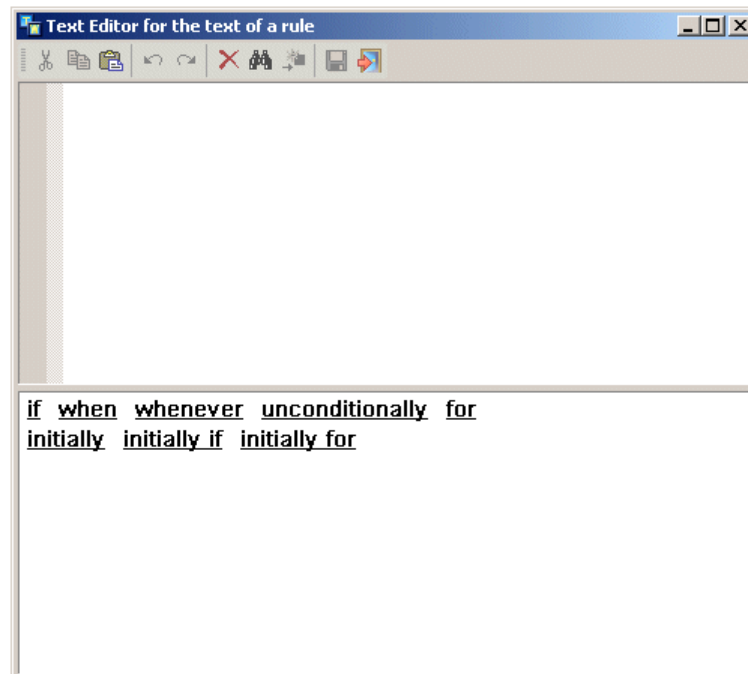
In this lesson, you will create a rule that animates a region of the office icon whenever the user moves the icon on the workspace. Thus, this rule is an example of **event detection**: the rule will be invoked whenever G2 detects movement of the office icon. Other examples of event detection include creating and deleting connections, and receiving and failing to receive a value.

You create rules on workspaces, therefore you use the KB Workspace menu to create the rule. First you will create the antecedent, then you will create the consequent.

To create the antecedent of a rule:

- 1 Choose Workspace > New Rule on the Basic Skills workspace.

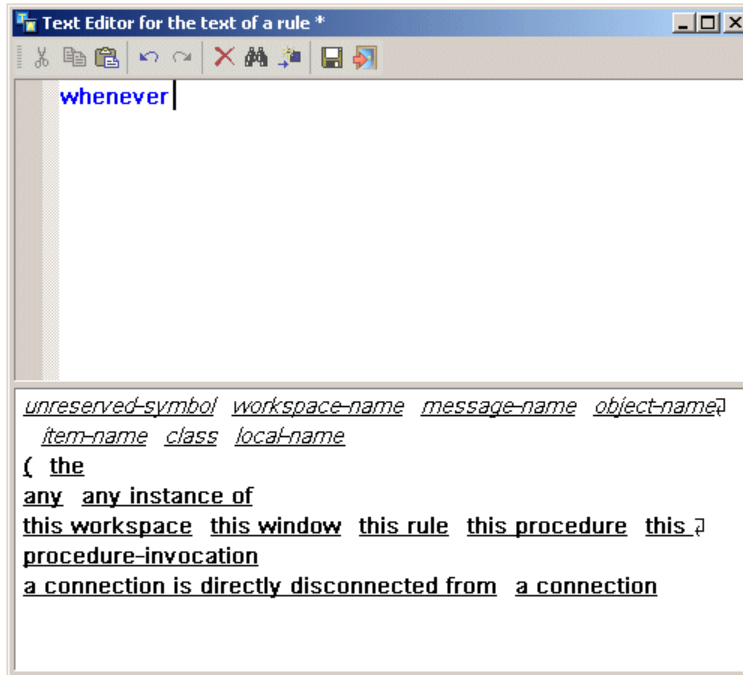
The editor shows the available rules from which to choose:



Since the rule detects an event, you will create a *whenever* rule.

- 2 Click the word **whenever** at the bottom of the text editor to insert it into the type-in area above.

G2 displays a different set of prompts:



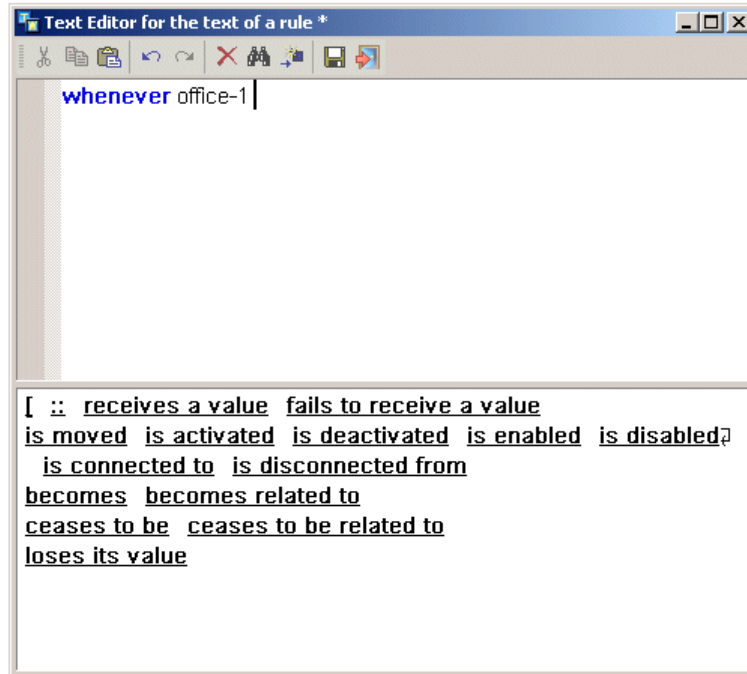
Notice that the rule now accepts any object name as the next part of the rule.

- 3 Enter the characters **office-** to display a new set of prompts.

Not only does G2 display the syntax of the rule, but it also displays any user-defined objects that exist in the knowledge base.

- 4 Click `office-1` to insert it in the rule.

G2 now displays the possible events that the whenever rule can detect:



For example, a whenever rule can detect when an attribute receives a value or fails to receive a value. Our rule will detect the movement of the object on the workspace.

- 5 Use the prompts to enter the following event in the rule:
is moved by the user

Tip You can also enter Ctrl + Space to display a list of possible completions for the word at the current cursor position. Use the arrow keys to choose the word and press Enter to insert it into the text.

At this point, G2 allows you to add other events to the antecedent of the rule, using “and” or “or” logic, or to begin the consequent. You will proceed with the consequent of the rule, which will transiently change the color of the icon to red. The icon will revert to its original black color when G2 is reset.

When you change the color of an icon, you must refer to a particular named region of a system-defined attribute called `icon-color`. You will change the **status** region, thus the syntax for referring to this attribute is: **the status icon-color**.

To create the consequent of a rule:

- 1 Click the then prompt to begin the consequent of the rule.
G2 now lists all of the possible actions that the rule can take when the event occurs. Our rule will change the color of a certain region of the icon, using the change action.
- 2 Use the prompts to help you enter the following consequent in the rule:
change the status icon-color of office-1 to red
The rule is now complete.
- 3 Press Enter to complete the rule.
G2 attaches the rule to the mouse.
- 4 Click to place the rule on the workspace.

The rule should look like this:

```
whenever office-1 is moved by the user then
  change the status icon-color of office-1 to red
```

Recovering from Syntactic Errors

G2 helps you prevent many types of syntactic errors in all types of statements, such as rules, procedures, and methods. If you do not enter the correct syntax in the natural language text editor, G2 indicates that there is an error and does not allow you to accept the edits.

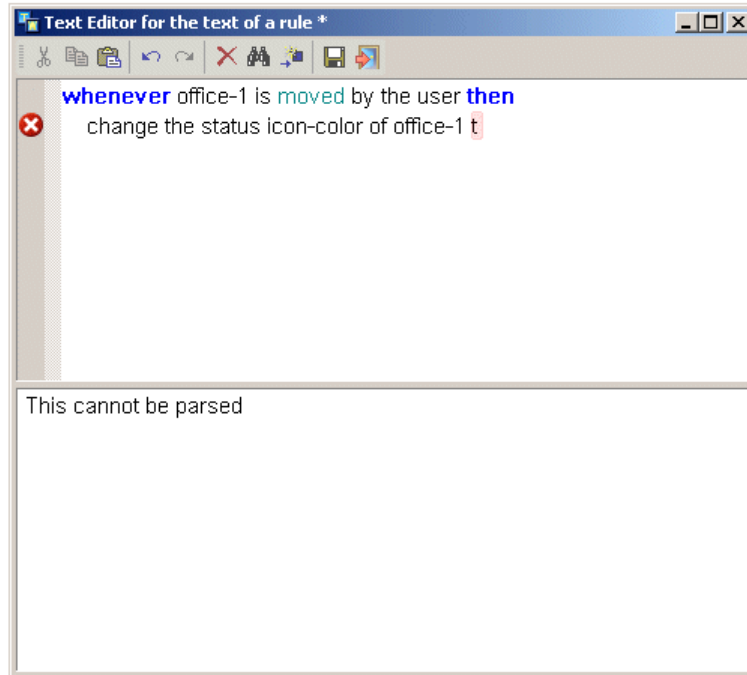
To simulate and recover from a syntactic error in the text editor:

- 1 Click just to the right of the word to on the text of the rule you just created to display the text editor.

Tip Notice that G2 places your mouse pointer at the exact location at which you clicked the text of the rule.

- 2 Use the Backspace key to delete up to the letter “o” in the word to.

- 3 Enter a space character as if you were trying to begin typing another word. G2 displays an error message indicating that the syntax you have typed is illegal:



When G2 cannot interpret a statement you have entered, it displays an X at the location of the syntax error and indicates the error with red highlighting and a message.

- 4 Backspace to delete the space and enter the letter "o" and the word "red".

Recovering from Other Types of Errors

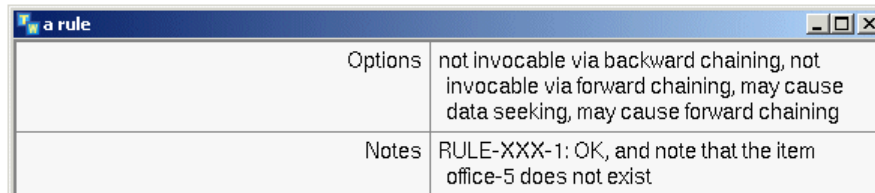
In addition to helping prevent syntactic errors, G2 helps you prevent other types of errors by displaying the status of an object in its **notes** attribute. If the object contains no errors, G2 displays a status value of OK in the **notes** attribute of the object. If the object contains an error, the **notes** attribute indicates the source of the error.

Tip If a rule is not being invoked as you expect, always check its **notes** to verify whether it has any errors.

To simulate and recover from an error:

- 1 Edit the text of the rule to specify the office named **office-5**, which does not exist.
- 2 Right-click the rule and choose **table** from the menu to display the rule's table.

The **notes** attribute of the rule indicates that the object referenced in the rule does not exist:



	Options	Notes
	not invocable via backward chaining, not invocable via forward chaining, may cause data seeking, may cause forward chaining	RULE-XXX-1: OK, and note that the item office-5 does not exist

- 3 Edit the rule to refer to **office-1** again.
- 4 Display the table again and verify that the status of the rule is **OK**.

Summary

In this lesson, you learned how to:

- Use the New Rule menu choice to create a **whenever** rule that detects the event of moving an object on a workspace.
- Use the **change** action to transiently change the color of an icon.
- Use the reserved word **the** to refer to attributes in statements.
- Recover from syntax errors in the text editor.
- Determine if other types of errors exist by viewing the **notes** attribute.

Running and Pausing Applications

In this lesson, you will learn how to:

- Run, pause, and resume the application.
- Invoke a **whenever** rule that detects the movement of an object on a workspace.

Now, you will test the rule to see if G2 detects the event.

To test the rule:

- 1 Move **office-1** on the workspace.

What happens?

Nothing happens! Why? Because the knowledge base is not currently running.

- 2 Display the Run menu in the top-level menu bar.

Notice the first menu choice is Start. This means that the application is not currently running. In fact, it has never been running since you loaded the application!

As you have seen, you can perform numerous operations in G2, even when the knowledge base is not running. However, as you also just saw, G2 cannot invoke rules unless the application is running.

Note For the inference engine to invoke rules and draw conclusions, G2 must be running.

- 3 Choose Start to start G2 running.

Tip You can also control the status of the application from the G2 Main Menu.

- 4 Now move **office-1** on the workspace.

Remember that **office-1** is the office in the lower-left corner of the connected offices.

Does the icon color change? This time it does. G2 detects the event, invokes the rule, and performs the action.

- 5 Choose Run > Restart to restart the application.

Restarting the application sets all objects back to their default state. In this case, resetting changes the icon color back to black.

- 6 Pause the application by choosing Run > Pause.

- 7 Try moving the object again.

Nothing happens. Again, G2 cannot invoke the rule while the knowledge base is paused.

- 8 Start the application running again by choosing Run > Resume.

Notice that G2 automatically changes the icon color because it already detected the event.

Note G2 can detect events while the knowledge base is paused; however, it cannot perform any actions.

9 Finally, try moving one of the other offices.

Nothing happens because the rule is written specifically for `office-1`. In a later tutorial, you will learn how to write generic rules that apply to all offices of a particular class.

Saving Applications and Shutting Down G2

To make your knowledge base permanent, you must save it to a KB file. When you do this, it is safe to shut down G2. Later, you can start G2 again and load your application.

To save your application:

1 Choose File > Save KB.

G2 displays the name of the current module and the default filename.

Notice that G2 can still save the KB while it is running; it saves the knowledge as of the current time.

2 Edit the filename to append your initials to the KB file named `ch2.kb` and press Enter or click End.

3 Click OK to confirm the save.

G2 reports its progress as it saves the file.

Before you shut down G2, you should verify that G2 saved your KB by checking your KB directory.

By default, you cannot shut down the G2 server from the menu bar. To do so, you must switch to administrator mode. You can, however, shut down G2 from the G2 Main Menu. To shut down G2 from the menu, G2 must be paused.

To shut down G2:

➔ Pause G2, then choose File > Shut Down G2 and click OK to confirm.

You can also shut down G2 by right-clicking the G2 server icon in the system tray and choosing Shut Down G2.

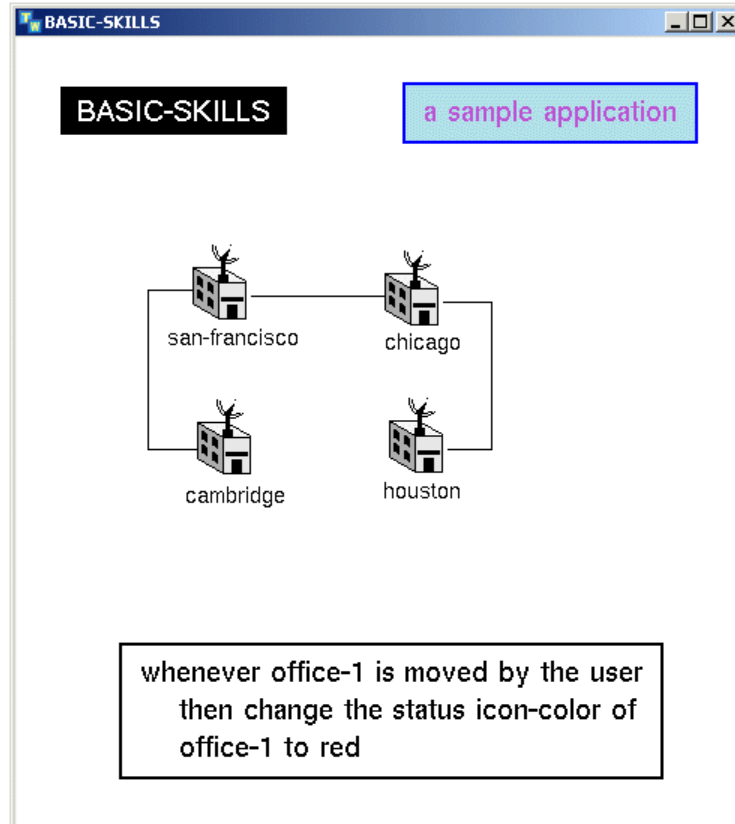
Summary

In this tutorial, you learned:

- How to run, pause, and shut down G2.
- How to load and save applications.
- That G2 applications are stored in knowledge bases that consist of modules.
- How to show the module hierarchy, using the Inspect facility.
- About objects and their attributes.
- How to refer to attributes in rules.
- How to interact with objects, using the object menu.
- How to interact with workspaces, using the KB Workspace menu and keyboard commands.
- How to use the G2 text editor to edit values of attributes.
- How to add attribute displays to objects and edit them.
- How to use connections to create graphical relationships between objects.
- About the different kinds of rules and inferencing techniques.
- How to use the natural language text editor to create a rule that detects the event of moving an object on a workspace and animates an icon.
- About some basic error handling techniques.
- How to save the KB and shut down G2.

Solutions

The Basic Skills workspace looks like this:



Creating a Schematic Diagram

Teaches the basic skills for interactively creating a schematic diagram of a video conferencing application.

Goals of a Schematic Diagram	75
Loading the Knowledge Base	76
Creating and Deleting Objects Dynamically	76
Editing a Class Definition	84
Creating Connection Stubs Dynamically	97
Summary	106
Solutions	107



Goals of a Schematic Diagram

In this tutorial, you will begin to use G2's procedural programming language to create a schematic diagram of a video conferencing application. Specifically, the application will:

- Create and delete office sites interactively
- Add connections to office sites interactively

The application will enable end users to dynamically create a schematic diagram according to the needs of the application. Different users of the application can access the same knowledge base to create different layouts of the schematic diagram, without having to rewrite any code.

Loading the Knowledge Base

You will start from the Basic Skills tutorial you loaded at the beginning of the first tutorial and build on this application to create a schematic diagram.

To load the sample application:

- 1 Load G2.
- 2 Load *ch3.kb*.

This KB is similar to the Basic Skills KB you created in the previous tutorial except that there is only one office and there is no rule.

Creating and Deleting Objects Dynamically

In this lesson, you will learn how to:

- Create an action button that dynamically creates an object
- Use the Inspect facility to locate transient knowledge
- Programmatically transfer objects to a workspace and make them permanent
- Create an action button that deletes all objects on a workspace

What is an Action?

In the first tutorial, you saw how to delete an object interactively by using the object menu. You also saw how to create and delete connections interactively by connecting stubs.



You use actions to create and delete objects and connections programmatically.

An **action** is a type of G2 statement that executes an activity on an object or on the G2 environment in general. You learned about actions in the Basic Skills tutorial when you created a rule that changed the color of the icon when it was moved on a workspace. In the rule, you used the **change** action to change the color of an icon dynamically.

You use G2 actions in numerous places in an application:

- In the consequents of rules
- In buttons that perform a sequence of actions
- In procedures and methods that execute procedural statements
- In user menu choices that perform custom operations on classes of objects

In general, you use G2 actions anywhere in the application where you want to control the behavior of objects or control the execution of the knowledge base itself. For example, you use actions to:

- Assign values to objects, using the **conclude** action
- Create and delete objects and connections, using the **create** and **delete** actions
- Change the value of an attribute, using the **change** action
- Send a message to the operator, using the **post** action
- Show or hide a workspace, using the **show** and **hide** actions
- Move or animate an object on a workspace, using the **move**, **transfer**, or **rotate** actions
- Cause specific categories of rules to execute, using the **invoke** action
- Control the execution of various parts of the knowledge base, using the **start**, **invoke**, **activate**, **deactivate**, **reset knowledge-base**, and **pause knowledge-base** actions

Using an Action Button to Create an Object

Button

Suppose you want to create an object dynamically with some kind of gesture. You might create an **action button** to perform this operation. An action button is a G2 object that executes a sequence of actions when the user clicks on the button.

In this lesson, you will create an action button that invokes the **create** action to create an object dynamically based on its class. The button will create an office object.

Action buttons execute only when G2 is running. Therefore, to specify the attributes of the action button, G2 must be paused.

To use an action button to create an object:

- 1 Choose Workspace > New Button > action-button to create an action button.
- 2 Right-click the button to display its menu and choose **table** to show its table.
The attributes you need to specify are:
 - **Label**, which determines the button label to display
 - **Action**, which determines the actions to perform
- 3 Edit the **label** attribute to be "Create Office".

The label requires a text string, which is another G2 data type. The **text** data type is a sequence of alpha-numeric characters that can include spaces and that requires quotation marks enclosing the characters. Strings are case sensitive, whereas symbols are not.

Note As noted in the previous lesson, you typically do not hard-wire text into an application, such as button labels. Instead, you use the G2 Foundation Resources (GFR) module to provide text keys, which are symbols that G2 substitutes with actual text when the application is running.

- 4 Use the syntax-guided text editor to edit the **action** attribute and specify the following action:

create an office

The **create** action creates an object of a particular class.

The button now looks like this:



- 5 Hide the table and start running G2.
- 6 Test the button by clicking on it once.

What happens?

Nothing appears to have happened! However, let's explore further to see what actually happened behind the scenes.

Exploring Permanent and Transient Knowledge

An object is part of the **permanent knowledge** in the KB when:

- It has been created interactively and still retains its automatic permanent status.
- It has been created programmatically, transferred to a workspace, and made permanent with the **make permanent** action.

A permanent object must be on a workspace. The office icon on the Schematic Diagram workspace is an example of a permanent object.

An object is part of the **transient knowledge** in the KB when:

- It has been created programmatically but not made permanent.
- It has been made transient through the **make transient** action.

A transient object may be on or off a workspace. The office that you created by using the action button is an example of a transient object; it exists in the knowledge base but is not permanent.

The difference between permanent and transient objects is what G2 does with them when you reset the knowledge base. G2 deletes transient objects when you reset the KB, whereas G2 retains permanent objects when you reset the KB.

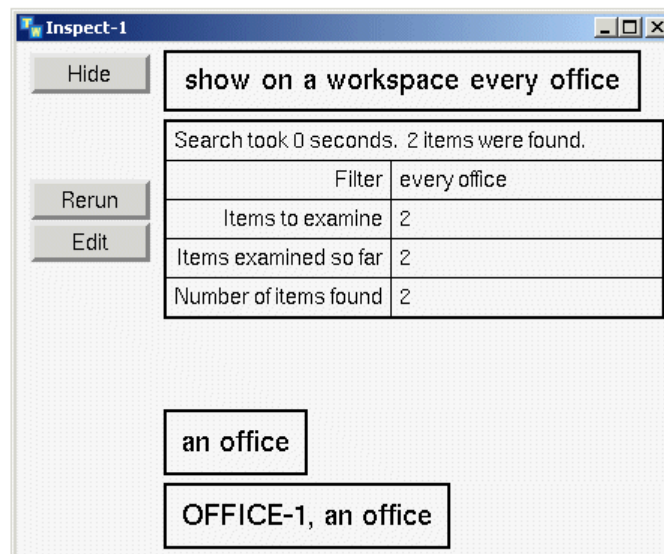
You can locate transient objects by using the Inspect facility. Now, you will verify that G2 actually did create an office through the **create** action in the button.

To use the Inspect facility to locate a transient object:

- 1 Choose Tools > Inspect and enter the following command:

show on a workspace every office

This command creates a temporary representation of every object that is a type of office:



- 2 Right-click the second office representation and choose **table**.

G2 displays the table for the existing office on the Schematic Diagram, whose attributes have already been specified.

- 3 Right-click the second office representation again and choose **go to original**.

G2 places the mouse pointer on the office located on the Schematic Diagram workspace.

- 4 On the Inspect workspace, display the table for the first office representation. The attributes in the table are unspecified because the object was just created.

- 5 Click on the first office representation again to display its menu.

Notice that this office does not have a **go to original** menu choice. Why? Because this office does not have an icon representation yet. The office exists in the knowledge base, but it does not yet exist on any workspace.

As mentioned earlier, when you reset the knowledge base, G2 deletes transient objects.

To verify that G2 deletes transient objects:

1 Reset G2.

The Inspect workspace should still be visible. Notice that G2 deleted the first office representation because it was transient. Only the permanent office representation now exists on the Inspect workspace, which is the office on the Schematic Diagram workspace.

2 Hide the Inspect workspace.

Making an Object Permanent

When you create an object programmatically in G2 and want the object to be a permanent part of the knowledge base, you use these two additional actions:

- The **make permanent action**, which makes the object permanent
- The **transfer action**, which transfers an object to a workspace at a particular location

Similarly, when you delete permanent objects from the knowledge base, you use the **make transient action** to make the object transient before you delete it.

When you use the **transfer action**, you can transfer most objects that exist in the knowledge base, either permanent or transient. You transfer the object to a named workspace, which you specify following the reserved word **to**. You can also transfer an object to the current workspace on which the button is located by using the **this workspace expression**. You specify the x,y coordinates of the workspace location following the reserved word **at**. For example, this action transfers **office-1** to the workspace named **schematic-diagram-2** at the coordinates 50,50:

```
transfer office-1 to schematic-diagram-2 at (50,50)
```

When you use the **make permanent action**, you place between the reserved words **make** and **permanent** the transient object to make permanent. However, how do you refer to a transient object? You do this by using a local name.

Using Local Names in Statements

When you use actions that operate on transient objects, you use a **local name** to refer to the object in succeeding statements. A local name is a symbol that represents the object locally within the scope of the compound statement, in this

case, the action of a button. You also use local names in rules, methods, and procedures.

Tip We recommend that you always use this convention for local names in statements: the local name should be a one to three letter symbol in all upper case letters, which corresponds to the name of the object it represents.

When declaring local names in statements, you place the local name after the class name that the local name represents. In this action statement, *O* is a local name that represents an object that is a type of office:

```
create an office O
```

Once you have established a local name for an object within an action statement, you can refer to the object locally within the compound statement to make it permanent. This statement makes the office object whose local name is *O* a permanent part of the knowledge base:

```
make O permanent
```

Performing Multiple Actions In Order

Often, you use action buttons to execute multiple actions. When you create an action button with more than one action, you must precede the actions with an *in order* clause. An *in order* clause indicates that G2 should execute the actions in sequence.

Whenever you use an *in order* clause, each statement must be separated by the reserved word *and*. By convention, you place the word *and* at the end of a statement and you indent each statement under the *in order* clause, as follows:

```
in order
  statement and
  statement and
  statement
```

Using Proper Indentation in Statements

Whenever you use the text editor to enter statements such as actions, rules, methods, or procedures in the text editor, you should indent them so they are easier to read; however, procedure compilation and execution are not affected by format. The general rules for indentation are:

- Always place each procedural statement alone on its own line.
- Always indent procedural statements under its syntactic element.
- Always place the separator between statements.
- If a statement is too long for one line, insert a line feed and indent the line.

In the syntax shown above for performing multiple actions in the **action** statement of an action button, the rules apply as follows: the **in order** statement begins a new procedural statement, thus it appears on its own line; the sequential statements appear indented under the **in order** statement on their own line; and the **and** separator appears at the end of each statement.

You indent statements in the text editor by using these text editor commands:

Command	Result
Ctrl + j or Return	Inserts a line feed in the statement
Tab	Inserts five spaces in the statement

Using an Action Button to Create a Permanent Object

Now you will update the action button to make an office a permanent part of the knowledge base. The action will use the **transfer** action to transfer the office object to a random location on the current workspace.

You use the G2 **random** function to generate random x,y coordinates for the location of the office on the workspace. The **random** function takes two arguments, the lower and upper bounds of the random number.

To update the action button to make an object permanent:

- 1 Edit the action attribute of the Create Office button to specify the following sequence of actions:

```
in order
  create an office O and
  transfer O to this workspace at
    (random (-300,300),
    random (-300,300)) and
  make O permanent
```

Notice that the statement follows the rules of indentation outlined in [Using Proper Indentation in Statements](#). Notice also that the text editor highlights balanced parentheses.

- 2 Test the action button again.

Hint Remember, G2 must be running to execute the action of an action button.

This time, G2 makes the object permanent and transfers it to the workspace at a random location.

- 3 Reset the KB to verify that the object survives because now it is permanent.

You have now created an action button that dynamically creates an object, which is a permanent part of your knowledge base.

Using an Action Button to Delete Objects on a Workspace

You might want to create an action button that deletes all the objects on a workspace. You use the G2 **delete** action to delete objects programmatically. Remember, before you can delete an object, you must make the object transient by using the **make transient** action.

You indicate the objects you want to make transient and delete by providing a G2 expression that refers to a set of objects. For example, you might want to delete every object on a workspace or every object connected to a particular object. You use the **every** clause to refer to all members of a class. You use the clause **upon** followed by the name of a workspace or **this workspace** to delete objects on a particular workspace. For example, this action deletes every instance of the **office** class on the workspace named **schematic-2**:

```
delete every office upon schematic-2
```

Now you will create an action button that deletes every office on the current workspace.

To use an action button to delete all the objects on a workspace:

- 1 Create a new action button by choosing Workspace > New Button > action-button.

Shortcut You can also clone the Create Office action button.

- 2 Specify the button label as "Delete All".
- 3 Specify the action as follows:
 - in order
 - make every office upon this workspace transient and
 - delete every office upon this workspace

The button looks like this:



- 4 Test the button.

G2 deletes all the offices on the workspace.

To save the application:

- ➔ Save the KB to a new file named `ch3.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you learned how to use:

- The Workspace > New Button menu choice to create an action button that executes a sequence of actions
- The `create` action to create a transient object
- The `make permanent` action to make transient objects permanent
- The `transfer` action to transfer an object to a workspace
- The `random` function to generate random numbers
- The `this workspace` statement to refer to the current workspace
- Local names within action statements to refer to transient objects
- The `in order` statement to execute multiple actions sequentially
- Proper indentation when entering statements in the text editor
- The `make transient` action to make permanent objects transient
- The `delete` action to delete transient objects
- The `every` statement to refer to every member of a class

Editing a Class Definition

In this lesson, you will learn how to:

- Show the class definition of an object
- Edit the attributes of a class
- Change manually edited attributes of a class
- Change the icon of a class
- Edit the default stubs of a class

What is a Class Definition?



So far you have been interacting with objects in the knowledge base. You learned how to create and delete them interactively and programmatically, edit their attributes through the table, and connect them interactively. Now you will explore how an object gets its definition from its associated class.

Every object is an instance of some other object, called its **class**. In object-oriented terms, an **instance** represents just one occurrence of potentially many occurrences of a class. Thus, an object has only a single class, whereas a class can have many instances.

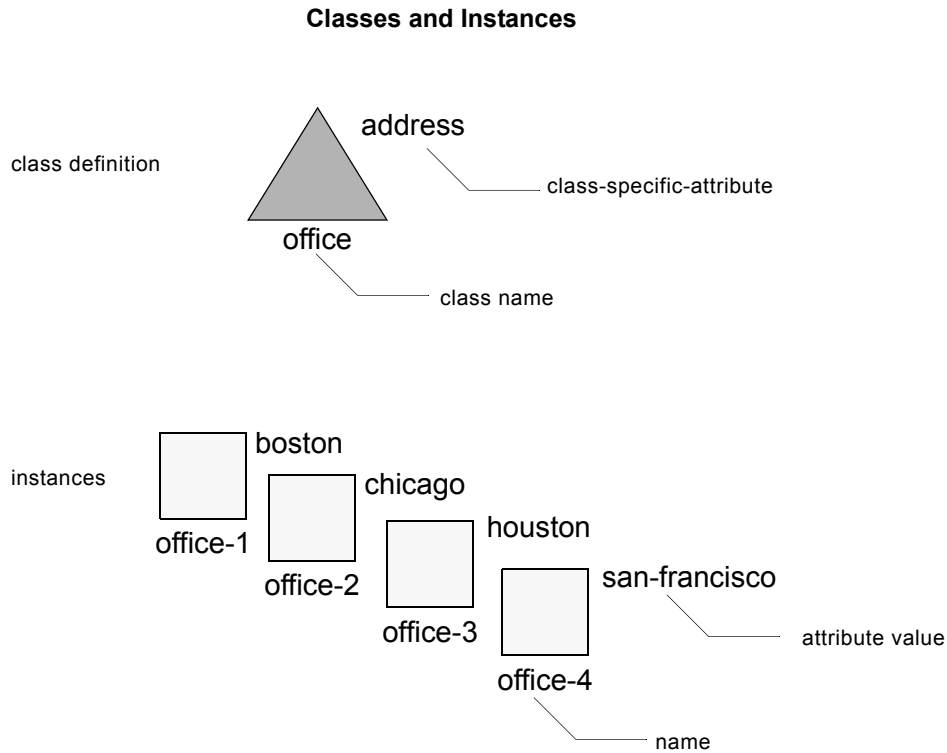
Why do you create class definitions for objects? The main reason is to eliminate redundancy in a knowledge base; you create the class definition only once, and you specify the unique attribute values in each instance. Another reason for creating class definitions is to avoid mistakes when you specify the attributes of an instance; each instance of a class has the same attribute specification by default.

You declare the common characteristics of each instance in the **class definition**. Typically, you declare:

- The **class name** of the class
- The **superior class or classes** from which the class inherits its default characteristics (attributes) and behaviors (methods)
- The **class-specific attributes** of the class, including their default values and data types
- The **icon** representation of each instance, if the class has one
- The default **stubs** on each instance of the class

Each instance has the same icon, stubs, and attributes. However, each instance has a different name and has different attribute values.

The following figure illustrates the relationship between classes and instances:



In this figure, the **office** class definition describes the common characteristics of its instances, for example, the **address** attribute. Each instance inherits its definition from its class. Each instance has the same icon but a different name and different attribute values.

Just as an instance obtains its definition from its class, a class obtains its definition from other classes by means of **inheritance**, which you were introduced to in the previous tutorial. A class can inherit its definition from built-in G2 classes or from user-defined classes. The root, or highest-level, G2 class from which user-defined classes inherit their definitions is the **item** class.

You will learn more about the available G2 classes from which user-defined classes can inherit their definitions in [Building a Knowledge Base](#).

Creating a Class Hierarchy

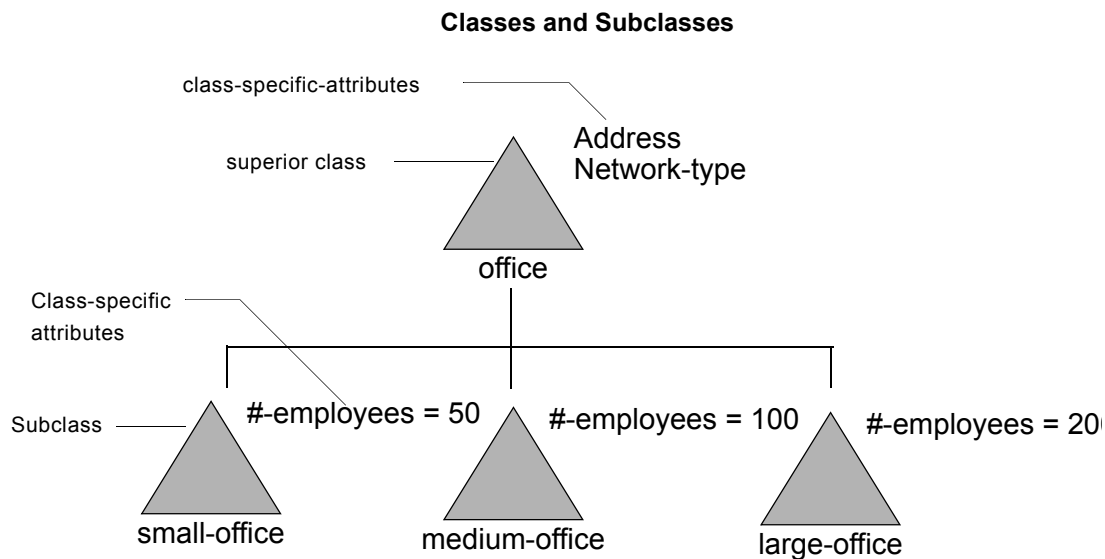
Because classes can inherit their characteristics from user-extensible classes, you can use object-oriented programming techniques to **encapsulate** knowledge at the appropriate level in your **class hierarchy**. To encapsulate knowledge means to organize related knowledge together in a single object so it can be shared with other objects. Using encapsulation, objects that share existing knowledge can hide the complexity in their definition. The class hierarchy describes how classes at each level inherit their definitions from existing classes.



Create class hierarchies to reuse information and avoid redundancy in an application.

G2's class system is based on multiple inheritance, which means that a class can inherit from one or more direct-superior classes. A class that is located above another class in the hierarchy is called a **superior class**. A class that is located below another class is called a **subclass**.

This figure shows one possible user-defined single-inheritance class hierarchy where the **office** class inherits from the system-defined class, **object**, and the direct subclasses of **office** are **small-office**, **medium-office**, and **large-office**.



In this figure, the **office** superior class defines two attributes that encapsulate information about offices in general: **address** and **network-type**. Each subclass is a kind of **office** and defines a single additional attribute and default value, which describes the unique characteristics of each subclass. Each subclass encapsulates the knowledge about offices in general by inheriting its definition from the **office** class.

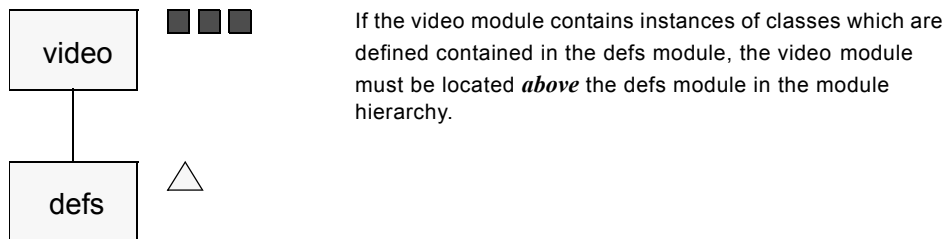
Once you have created a class definition that has a class name and an acceptable set of direct superior classes, you can create instances of the class, either interactively, using a menu choice, or dynamically, using an action.

Organizing Classes and Instances

You typically keep class definitions separate from their instances in an application. This might mean keeping classes and instances on separate workspaces. It can also mean creating separate modules for classes and instances.

If the class definitions are located in a separate module in the knowledge base, the class definitions must be loaded *before* the instances of the class. Because G2 loads the module hierarchy from the bottom up, this means that the module that contains the instances of a class must be located *above* the module that contains the class definitions. In other words, the definition module must be required, directly or indirectly, by the instance module. The fact that object instances get their definitions from classes is precisely why some modules are dependent on other modules in the hierarchy, as this figure shows:

Class Definitions and Modules



Displaying a Class Definition

Now you will look at the class definition for the **office** class. The definition is located on another workspace of the Basic Skills application.

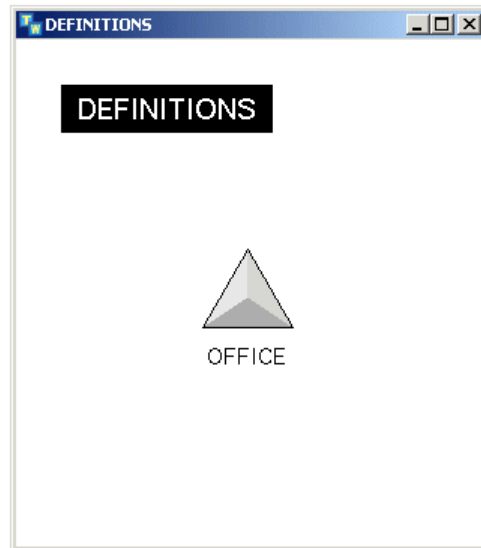
The three basic attributes of a class definition are:

- **Class-name**, which specifies the name of the class
- **Direct-superior-classes**, which defines the class or classes from which the class inherits its definition
- **Class-specific-attributes**, which defines the unique attributes of the class, including the default values and the data types

To display the class definition for the office:

- 1 Choose Workspace > Get Workspace to display the Definitions workspace.

You will see the class definition for the office:



- 2 Display the table for the office class definition.

Tip With the cursor over the table, use Ctrl + b to make the table bigger until the attributes are legible or use Ctrl + f to make it full size.

Here is a partial table for the office class definition:

OFFICE, a class-definition	
Notes	OK
Authors	administrator (1 Oct 1997 8:48 p.m.), nrs
Change log	0 entries (disabled)
Item configuration	none
Class name	office
Direct superior classes	object
Class specific attributes	address; phone-number; network-type

The office class has a single direct superior class, the **object** class, and defines three class-specific attributes.

You only need to specify the unique attributes of the class in the **class-specific-attributes**; the other attributes are inherited from the superior class.

- 3 Hide the table for the class definition.

Creating an Instance

In a previous lesson, you learned how to create an instance of a class programmatically, using an action button. Now you will create an instance interactively by using a menu choice.

To create an instance of a class:

- ➔ Choose **create instance** from the class definition menu and click to place the instance on the workspace.

The instance looks exactly like the instances you created programmatically by using the action button:



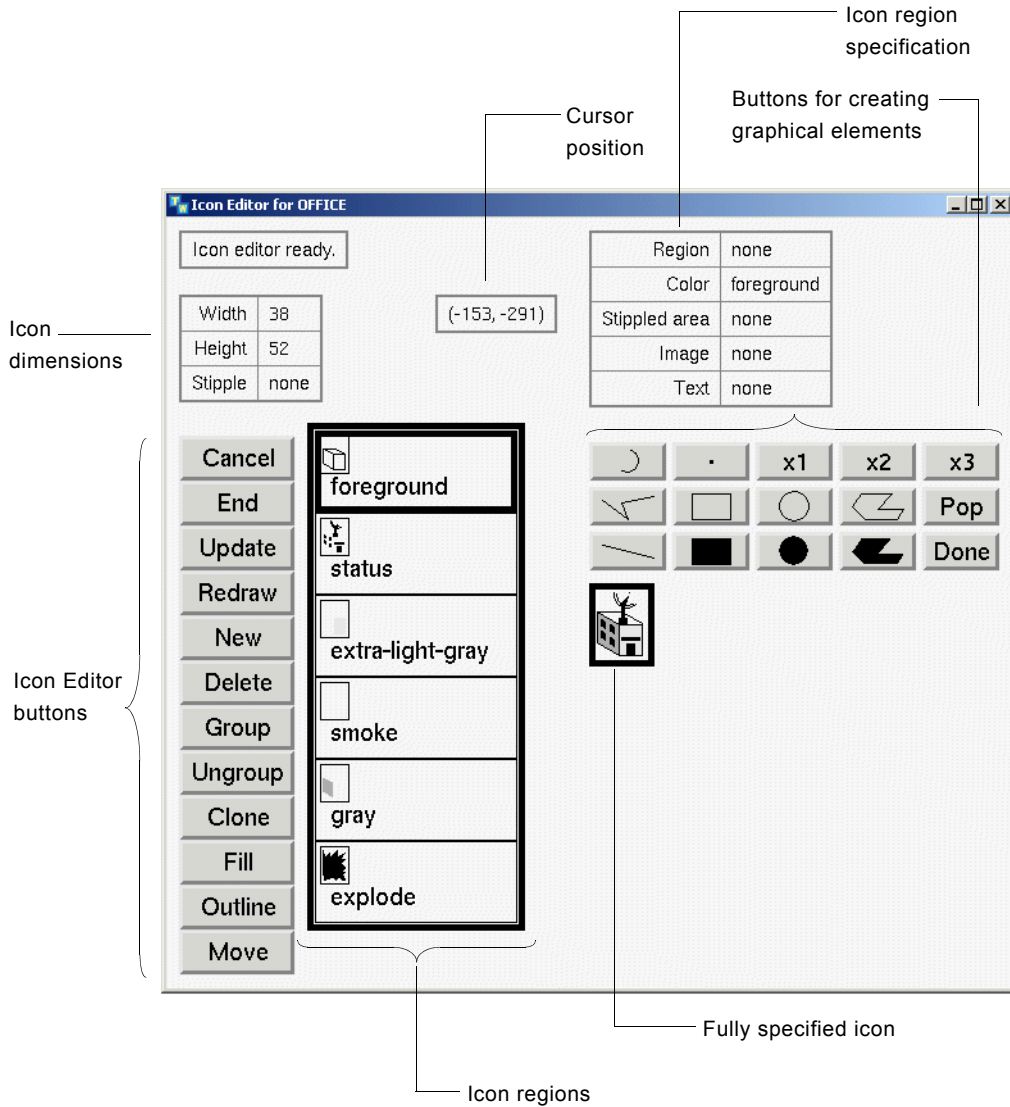
Editing the Icon

The class definition also defines the icon for each instance of the class. By default, the class inherits the icon definition of the closest superior on the class inheritance path. You can also create a new icon for a class, or you can copy the icon definition of an existing G2 class from the icon class library.

The **G2 Icon Editor** allows you to define complex icons that consist of multiple layers and regions. An **icon region** consists of one or more layers. An **icon layer** consists of any number of graphical elements of the same color. By naming icon regions, you can animate icon layers when the application runs, as you did in the Basic Skills tutorial by using a rule.

To edit the icon for a class:

- 1 Choose edit icon from the class definition menu to see the Icon Editor workspace:



Each layer also has a color and can have a bitmap image. By default, the name of each layer is the same as its color.

Notice the **status** layer. The name of the region associated with the **status** layer is the same as the layer. In the Basic Skills tutorial, you created a rule that animated the **status** icon region when you moved the icon on the workspace.

- 2 Click on each layer of the icon and notice the definition of each layer in the icon region specification area.

The **status** layer is one of the layers with a region name.

- 3 Click on the **status** layer, then click on the value of the **color** attribute of the layer.

G2 displays a color palette for choosing the color of all graphic elements in the layer.

- 4 Choose a new color for the **status** layer of the office icon.
- 5 Click the End button to accept the changes.

Notice that the existing office immediately inherits the new icon definition.

- 6 Change the color of the **status** region back to the foreground color, which is black.

Editing the Attributes of a Class

You declare the class-specific attributes of a class in the class definition. As mentioned earlier, in general, whenever you declare the attributes of a class, you should declare them with the most restrictive data type for maximum performance.

When you add new class-specific attributes to a class definition, the instances of the class update automatically to include the new attributes.

When you edit certain types of attributes, such as the **class-specific-attributes** attribute, G2 displays a text editor with scroll bars. In a **scrollable text editor**, you can simply press Enter to move to the next line, as opposed to using the Ctrl + j command. To accept the edits in a scrollable text editor, use the Ctrl + Enter command.

To edit the attributes of a class definition:

- 1 Display the table for the **office** class definition.
- 2 Edit the **class-specific-attributes** to specify a default value of T1 for the **network-type** as follows:

```
network-type is a symbol, initially is T1;
```

First you enter the data type for the attribute, then you enter the default value. Use the syntax-guided editor to help you enter the specification.

Hint Notice that each attribute is separated by a semi-colon (;). Thus, you must enter a semi-colon after the **network-type** attribute specification. You do not need a semi-colon after the last attribute.

- 3 Press Enter and declare a new attribute named `number-of-employees`, which is a type of integer.
- 4 Use Ctrl + Enter to accept the edits.

The partial table for the office class definition should look like this:

OFFICE, a class-definition	
Notes	OK
Authors	nrs (23 Jul 2007 11:48 a.m.), administrator
Change log	0 entries
Item configuration	none
Class name	office
Direct superior classes	object
Class specific attributes	address is a symbol, initially is g2; phone-number is a symbol, initially is g2; network-type is a symbol, initially is t1; number-of-employees is an integer, initially is 0

- 5 Display the table for the existing office on the Definitions workspace to verify that the attributes in the table reflect the newly declared class-specific attributes.

The table for the office should look like this:

an office	
Notes	OK
Item configuration	none
Names	none
Address	g2
Phone number	g2
Network type	t1
Number of employees	0

Changing Manually Overridden Attributes of Instances

As you just saw, when you edit the default value of an attribute in the `class-specific-attributes` of a class definition, G2 automatically updates the attributes of all its instances.

However, suppose you were to manually override the attribute value of an object by editing the table. For example, suppose you had manually specified the `network-type` of the office before you updated the class specific attributes. Should

G2 change these manually overridden values or not? By default, G2 does *not* automatically update attributes whose values have been manually overridden.

To change manually overridden attributes, you use the **change** attribute in the table for the class definition.

To verify that manually overridden attributes do not update:

- 1 Edit the value of the **number-of-employees** attribute of the office instance on the Schematic Diagram workspace by editing its table.

Choose a value other than 1 or 0.

- 2 Edit the **class-specific-attributes** of the office class definition to provide a default value of 1 for the **number-of-employees** attribute.

You use the **initially is** syntax to specify the default value.

The table for the office instance remains unchanged; G2 did not update the **number-of-employees** attribute because its value has been manually overridden.

Now you will use the **change** attribute to update manually overridden attributes.

To change manually overridden attributes of a class:

- 1 Display the table for the office class definition.

Notice the **change** attribute in the table.

- 2 Use the syntax-guided text editor to enter the following command in the **change** attribute to update all instances of the class to use the new definition:

change the attribute **number-of-employees** of each instance
to the default value

The **number-of-employees** attribute in the table for the office instance is automatically updated to reflect the default value specified in the class definition.

Editing the Stubs of a Class

The class definition for a class with an iconic representation also declares the default stubs for each instance of the class. You declare the default stubs of a class to provide a way of connecting object instances without requiring that you create the stubs first.

A stub can specify numerous characteristics, including its:

- Connection style: orthogonal or diagonal
- Line pattern: solid, dash, dot, long dash, short dash, course dot, fine dot
- Connection arrows: arrow, diamond, triangle, circle, open, filled, thick, thin, wide, narrow, small, large
- Port name, which is the name of the connection

- Direction: input or output
- Class type: any subclass of the `connection` class

An **orthogonal connection style** means the connection is either vertical or horizontal, with possible vertices. A **diagonal connection style** means the connection is at any diagonal angle. If you do not specify the connection style, G2 uses an orthogonal connection style.

The office defines a single unnamed stub that uses an orthogonal connection style by default. You will now edit the existing stub to be a diagonal stub coming out of the top of the icon.

To edit the stubs of a class definition, you initialize the system-defined `stubs` attribute in the `attribute-initializations` attribute of the class definition.

To edit the stubs of a class definition:

- 1 Display the table for the office class definition.

Notice the attribute named `attribute-initializations`, which specifies a default initialization for the `stubs` attribute. Also notice the `initializable-system-attributes` attribute, which names the `stubs` attribute as one of the two system attributes of the `object` class whose values you can initialize.

The current `attribute-initializations` specification creates an orthogonal connection, by default, on the right side of the icon, as follows:

`stubs: a connection located at right 35`

- 2 Edit the `attribute-initializations` attribute to create a diagonal connection stub coming out of the top of the icon, as follows:

`stubs: a connection located at top 15 with style diagonal`

Notice that just as G2 does not update manually overridden attributes of existing instances when you edit the class definition, G2 does not update the default stubs of the existing instances either.

- 3 Create a new instance to see the new stub.

Notice that the connection is now leading out of the top of the icon:



- 4 Drag the connection and notice that you can move it in any diagonal direction.

- 5 To drop the connection, double click the mouse or use the Ctrl + a command.
The icon looks something like this:



- 6 Delete the existing instance with the original orthogonal stub definition.

Just as you can use the **change** attribute of the class definition to update manually overridden attributes of existing instances, you can use the **change** attribute to update the default stubs of existing instances. To do this, first you use the **change** attribute to delete the existing stubs on any instance, then you use the **change** attribute to add a connection stub of the desired type at the new location on any instance.

To save the application:

- ➔ Save the KB to the file named `ch3.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you learned how to:

- View the class definition for an instance of a class
- View the `direct-superior-classes` attribute of a class to determine its type
- Use the `create instance` menu choice on a class definition to create an instance of a class
- Use the `edit icon` menu choice for a class to edit its icon
- Edit the `class-specific-attributes` attribute of a class to specify its attributes
- Edit the `attribute-initializations` attribute of a class to initialize the `Stubs` attribute
- Use the `initializable-system-attributes` attribute of a class definition to determine the names of system-defined attributes you can initialize

Creating Connection Stubs Dynamically

In this lesson, you will learn how to:

- Create a method that programmatically creates a connection stub on an object
- Create a user menu choice that starts the method

Executing Actions on Classes of Objects

In the previous lesson, you edited the class definition to define the default stubs for a class. Suppose you wanted to allow the end user of an application to add stubs to instances of a particular class interactively. Recall from a previous lesson that you use the **create** action to create objects and connections programmatically.

In an earlier lesson, you used an action button to execute a sequence of actions to create and delete objects on a workspace. To add connections to an object interactively, you execute a similar sequence of actions. However, rather than using an action button to execute the actions, you can create a **user menu choice** associated with each instance of the class, which appears on its menu.

A user menu choice specifies a sequence of actions that G2 performs on specific instances of a class when the user selects the user-defined menu choice. You define a user menu choice for a class of objects. For example, you might create a user menu choice called **create connection** for the office class, which programmatically creates a connection stub at a particular location. In this way, you allow the end user of your application to add connection stubs interactively.

Using Methods and Procedures for Sequential Processing

As you saw in a previous lesson, you often need to execute a sequence of actions to perform the desired operation, such as creating an object. A user menu choice is similar to an action button in that you specify a sequence of actions in the **action** attribute, using the **in order** statement.

However, if the sequence of actions is complex, you typically create a method or procedure that performs the sequential processing of the action button or user menu choice. A **method** is a named object associated with a particular class that executes a sequence of actions when the application **starts** the method. A **procedure** is the same as a method except that it is not associated with any class.



Use procedures and methods to manipulate objects and their relationships in real time as the core repository of knowledge in your application.

You can define the same method for different classes, each of which executes a specific set of actions for all instances of a class. Furthermore, just as objects

inherit their attributes and icons from superior classes in the class hierarchy, objects also inherit their methods from their superior classes. Thus, you can create subclasses that obtain their methods from their superior class, or you can override the definition of the method in the subclass.

For example, the office class might define a method that computes its budget. You might define several subclasses of the office class, all but one of which computes its budget based on the default method. However, the home office might define its own method for computing budget, which is unique.



Use methods to encapsulate the behavior of classes across the class hierarchy.

You use the **start** action to start a method or procedure anywhere that G2 allows actions. Remember that you can use G2 actions in the consequents of rules and in action buttons, as well as in user menu choices.

In this lesson, you will create a method that programmatically creates a connection stub for an object. You will then create a user menu choice for the office class that starts the method.

What is the Format of a Method or Procedure?

Suppose you were to create a generic sequence of actions that executes the actions of the Create Office action button. Recall that the Create Office action button performed these actions in order: create an office, transfer the office to a workspace, and make the office permanent. To create a generic sequence of actions associated with a particular class, you create a method. To create a generic sequence of actions not associated with any class, you create a procedure.

Here is a generic procedure that executes the same actions as the action button you created earlier and then posts a message to the Message Board:

```
create-office (confirmation-string: text)
O: class office;
begin
  create an office O;
  transfer O to this workspace at
    (random(-300,300), random(-300,300));
  make O permanent;
  post confirmation-string
end
```

Method or Procedure Name

The **method name** or **procedure name** is the name you specify when you programmatically execute the method or procedure, in this example, `create-office`.

Argument List

The **argument list** specifies the objects or values that are passed to the method or procedure by the caller.

The format of a method is almost identical to that of a procedure with the exception of the argument list. For a method, the first argument in the argument list is always an instance of the class to which the method applies.

For example, here is an initialization method for objects of class **office**:

```
initialize(office: class office)
begin
  conclude that the number-of-employees of office = 25
end
```

Local Name Declarations

The **local name declarations** is a list of variables that the method or procedure uses locally in the body. If the local name refers to an object, as in the procedure, it has this format:

name: class class-name

For example:

O: class office

If the local name refers to a data type, it has this format:

name: data-type

For example:

N: integer

G2 supports these value types in local name declarations for procedures and methods:

- **quantity**: a floating point number or an integer
- **float**: a floating point number
- **integer**: an integer
- **symbol**: a sequence of alpha-numeric characters with no spaces
- **text**: a sequence of alpha-numeric characters surrounded by quotes, with spaces as needed
- **truth-value**: the symbol **true** or **false**

- **sequence:** a list of values. For example:
sequence(1, the symbol one, 1.0)
- **structure:** a list of name-value pairs. For example:
structure(corporation: the symbol gensym, street-number: 125)



For maximum efficiency, use strong typing when you declare the local names of procedures and methods.

Method or Procedure Body

The **method body** or **procedure body** is where you specify the sequence of actions that the method or procedure executes. Notice that the body starts with a **begin** statement and ends with an **end** statement. Also, statements in the body are separated by semi-colons (;). Contrast this to an action button in which each action is separated by the reserved word **and**.

Procedures and methods are designed to be extremely flexible in what they can do. In fact, procedures and methods support many of the same structures as standard programming languages, such as **looping**, **if-then statements**, and **case statements**.

In addition, procedures and methods allow for multi-threaded processing in a real-time environment by supporting **wait statement**, **allow other processing statements**, and **do in parallel statements**.

You use methods anywhere in your application that requires complex sequential and iterative processing for a particular class. You use procedures to perform the same kind of processing but not associated with any class.



To take full advantage of G2's object-oriented capabilities, always create methods rather than procedures whenever you want the application to perform sequential processing on classes of objects.

Rules for Proper Indentation of Methods and Procedures

When you enter statements in the method or procedure body, you should follow these conventions for proper indentation:

- Begin-end statements should appear on their own line.
- Nested begin-end statements should be indented to the next level of indentation.
- Sequential statements within a begin-end statement should appear indented under the begin statement.

- The semi-colon separator should separate statements.
- If a sequential statement within a begin-end statement is too long to fit on one line, enter a line feed and indent to the next level of indentation.

For general information about the rules of proper indentation, see [Using Proper Indentation in Statements](#).

Declaring Arguments

One of the powerful features of methods and procedures is the ability to pass arguments to the method or procedure when it starts. This means you can write a method or procedure that performs a generic operation and pass it specific arguments each time you start it to perform specific functions.

You might want the procedure to create an office on a particular workspace, rather than on the current workspace. To do this, you would rewrite the procedure to pass the name of the workspace as an argument to the procedure. You would then refer to the argument in the body of the procedure.

Here is a procedure that takes a workspace as an argument:

```
create-office (W: class kb-workspace)
O: class office;
begin
  create an office O;
  transfer O to W at (random(-300,300),
    random(-300,300));
  make O permanent;
end
```

This procedure transfers the office to *W*, a workspace, which is the argument to the procedure.

Notice that the format of an argument list is the same as the format of the local name declaration. In this example, the argument *W* is an instance of the *kb-workspace* class, which is the built-in class for workspaces.



For maximum efficiency, use strong typing when you declare the arguments to procedures and methods.

When you start the procedure, you supply a specific workspace as the argument.

Declaring Arguments for Methods

The argument list for a method is similar to that of a procedure with one important exception:

All methods require at least one argument, which is an instance of the class to which the method applies. If there are multiple arguments, the class name argument must be the first argument.

For example, suppose you were to create a method for the office class that computed its budget based on the number of employees. The first argument to the method would be the class to which the method applies, as follows:

```
compute-budget (office: class office)
begin
  ...
end
```

Any other arguments to the method would follow the class name argument. G2 allows you to define more than one same-named method for a class as long as the number of arguments differ.

Calling the Method or Procedure

You use the `call` statement to start a method or procedure. The argument list follows the method or procedure name.

For example, to call the `create-office` procedure shown earlier, you might use this statement:

```
call create-office(schematic-diagram)
```

This statement creates an office on the workspace named `schematic-diagram`. You could create an office on any workspace simply by starting the procedure with a different workspace name as the argument.

You start a method by using the `call` action with the name of the specific instance to which the method applies as its first argument. For example, this statement starts the `compute-budget` method for an instance of the office class:

```
call compute-budget (office-1)
```

G2 determines what method to call by looking at each class in the class inheritance path of the first argument, starting with the first class. It calls the method of the first class it encounters that defines a method with that name and same number of arguments as there are in the method call.

Creating a Method



You will now create a method that programmatically creates a connection for an instance of the office class. The same method applies to all instances of the office class. In a later tutorial, you will create unique methods for different subclasses of the office class.

Because you are creating a method rather than a procedure, the first argument to the method is an instance of the class.

The method will execute the following actions in order:

- Create a connection
- Make the connection permanent

To create a method:

- 1 On the Definitions workspace, choose `Workspace > New Definition > procedure > method` to create a method.
- 2 Choose `edit` to edit the method or double-click the method.
- 3 Enter `create-connection` as the name of the method.

The type of the first argument to any method is always the class to which the method applies, in this case, the office class.

- 4 Enter the argument list as follows:

`(office: class office)`

The method will add a connection to the object and make the connection permanent; therefore, you will declare the connection as a local name of the method. The built-in class name for connections is the `connection` class.

- 5 Enter the local name declaration as follows:

`C: class connection;`

Notice that the local name declaration must end in a semi-colon.

The body of the method will use the `create` action to create a connection and the `make permanent` action to make the connection permanent. The syntax for programmatically creating a connection is similar to the syntax for the system-defined `stubs` attribute of a class definition.

6 Enter the body of the method as follows:

```
begin
    create a connection C connected to office newly locating it at top
        at the position given by random (40) with style diagonal;
    make C permanent;
end
```

G2 updates the attribute display for the method to show the **qualified name** of the method:



OFFICE::CREATE-CONNECTION

The qualified name of a method concatenates the class name and the method name, using double colons, as follows:

office::create-connection

Office is the class to which the method applies, and create-connection is the name of the method.

Declaring the Method



You must define a method-declaration for each uniquely-named method definition. If you fail to do so, each method definition without a method declaration with the same name has this warning in its **notes** attribute:

note there is no method declaration defined with this name

To create a method declaration:

- 1 Choose Workspace > New Definition > Procedure > method-declaration to create the method declaration.
- 2 Edit the **names** attribute in the table for the method declaration to be the method name you entered when you created a uniquely-named method, in this case, create-connection.

The method declaration looks like this:



CREATE-CONNECTION

Creating a User Menu Choice that Starts a Method



Now that you have defined a method that creates a connection for any object, you will use the **start** action to start the method as the action for a user menu choice. You define a user menu choice for a specific class of objects. To refer to the object associated with the user menu choice, you use the **item**.

To create a user menu choice that starts a method:

- 1 On the Definitions workspace, choose **Workspace > New Definition > user-menu-choice** to create a user menu choice.
- 2 Edit the attribute display for the user menu choice, whose default value is **none**, to be **create-connection**.

This symbol is the name of the user menu choice that will appear in the object menu.

- 3 Display the table for the user menu choice.
- 4 Edit the **action** attribute to start the **create-connection** method, as follows:

start create-connection (the item)

Notice that you do not need to refer to the class-qualified method name when you start the method; G2 determines which method is defined for which class by looking at the first argument to the method.

- 5 Edit the **applicable-class** attribute to specify **office** as the class to which the user menu choice applies.
- 6 Display the menu for one of the office instances on the Schematic Diagram workspace to verify that the user menu choice appears.

Hint G2 must be running to update the user menu choices of an object.

- 7 Test the user menu choice by choosing it.

G2 creates a connection in a random location on top of the icon.

The final method should look like this:

```
create-connection (office: class office)
C: class connection;
begin
    create a connection C connected to office newly locating it at top
        at the position given by random (40) with style diagonal;
    make C permanent;
end
```

To save the application:

➔ Save the KB to the file named *ch3.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you learned how to:

- Use the **New Definition > procedure > method** or **procedure** menu choice to create a new method or procedure
- Use the **New Definition > procedure > method-declaration** menu choice to create a new method declaration
- Declare the arguments to a method or procedure
- Declare local names in a method or procedure
- Use the **begin** and **end** procedure statements to declare the body of a method or procedure
- Use the **call** statement to start the execution of a method or procedure

Summary

In this tutorial, you learned:

- How to use action buttons to create and delete objects interactively
- How to edit the class definition of an object to update its attributes, icon, and stubs
- How to update the instances of a class, using the **change** attribute
- How to use the **create**, **transfer**, and **delete** actions
- About permanent and transient objects and how to use the **make permanent** and **make transient** actions
- How to perform sequential processing, using the **in order** statement, methods, and procedures

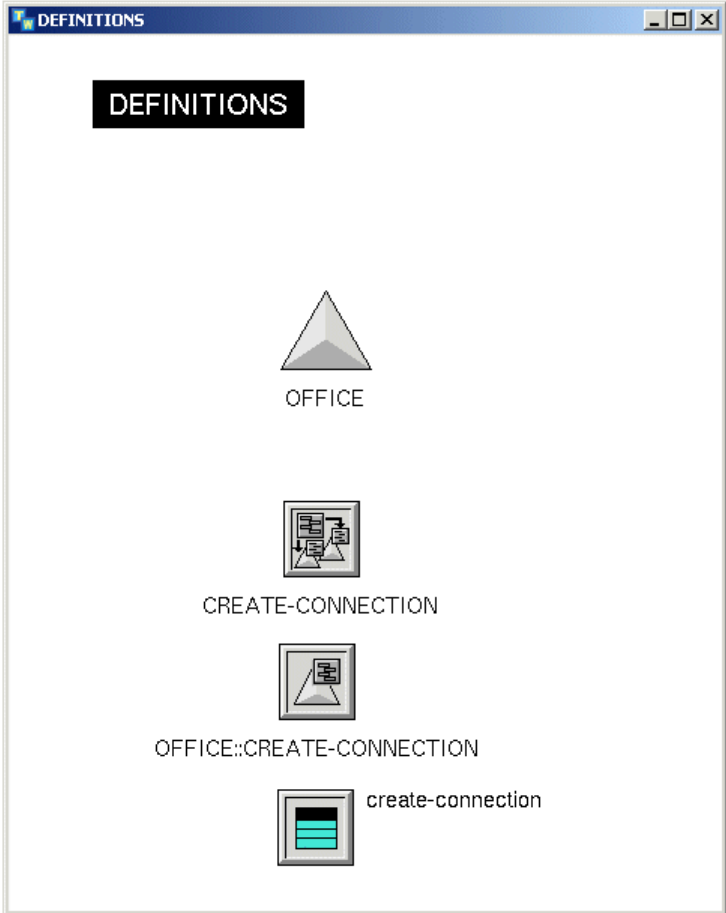
- How to start a method or procedure programmatically with arguments
- How to declare a method in the application

Solutions

The Schematic Diagram workspace looks like this:



The Definitions workspace looks like this:



Building a Knowledge Base

Describes how to create a simple application that computes the total cost of a video conferencing office based on the number of connections and that dynamically deletes the connections if the office is over budget.

Goals of the Knowledge Base	110
Loading the Knowledge Base	110
Counting the Number of Connections	110
Counting Connections for any Office	116
Using Event-Driven Processing	118
Using Data-Driven Processing	127
Keeping a History of Total Cost	131
Creating Subclasses of Offices	141
Disabling Rule Highlighting	146
Summary	147
Solutions	148



Goals of the Knowledge Base

In the previous tutorial, you created a schematic diagram for a video conferencing application. In this tutorial, you will begin to build a knowledge base that:

- Computes the number of connections of each office
- Computes the total cost of each office based on the number of connections and a fixed cost per minute
- Monitors total cost to determine if an office is over budget
- Deletes the connections when an office is over budget

Because this is not a real application, G2 will simulate total cost by incrementing its value by a fixed cost per minute. Thus, you can think of this application as a **prototype** for a real video conferencing application, which would compute total cost based on real-time video conferencing connection data.

Loading the Knowledge Base

You will start from the finished Schematic tutorial and build on this knowledge base.

To load the schematic diagram:

- ➔ Load the KB named *ch3.kb* that has your initials appended to it or load the KB named *ch4.kb* to load the solution KB that is the starting point for this tutorial.

Counting the Number of Connections

In this lesson, you will learn how to:

- Create a class-specific attribute that is an integer
- Create a rule that concludes the number of connections for an object
- Invoke the rule by scanning

Creating an Attribute for the Number of Connections

You can connect each video conferencing office to any number of other offices. The total cost of each office depends on the number of connected offices. Thus, to compute the total cost of each office, you must first compute the number of connections.

To compute the number of connections for an office, you will create an attribute of the office class to hold the value. Whereas in the previous tutorial you entered the

attribute values in the table, you will now use G2's inference engine to compute the number of connections. Because the number of connections must always be an integer, you will add **type checking** to the attribute by declaring its data type to be an integer.



Always use type checking when you declare attributes to ensure that the inference engine always computes values of the proper type.

To declare an attribute that keeps track of the number of connections:

- 1 Display the Definitions workspace.
- 2 Edit the class definition for the office class to create a class-specific attribute named `number-of-connections`, which is an integer, as follows:

`number-of-connections` is an integer

Hint Remember, class-specific attributes are separated by a semi-colon, and you can accept the edits by using the Ctrl + Enter command.

Notice that when you accept the edits, G2 automatically inserts a default value in the **class-specific-attributes** specification. G2 does this whenever you declare a data type for an attribute, without explicitly declaring a default value.

- 3 Verify that the office instance on the Schematic Diagram workspace has the new attribute in its table:

an office	
Notes	OK
Item configuration	none
Names	none
Address	none
Phone number	none
Network type	14.4k
Number of employees	0
Number of connections	0

Using a Rule to Count the Number of Connections

Now that the office defines an attribute to hold the number of connections, you can use G2's inference engine to compute the value of this attribute based on the number of physical connections to each office. You use a numeric expression, **the count of each**, to count the number of connections for an office. You use the **conclude** action to conclude the value of an attribute. You will write a rule that

concludes the value for a particular office regardless of any conditions being met. Therefore, you will use an **unconditionally** rule.

Tip In general, you keep rules and object definitions on separate workspaces in an application.

To create a rule that counts the number of connections for a particular office:

- 1 Create a new workspace named rules-workspace.

Note Because the word **rules** is a reserved word in G2, you cannot create a workspace named rules.

- 2 Choose Workspace > New Rule to create a new rule.
- 3 Create a rule that unconditionally concludes that the number-of-connections of an office named **office-1** equals the count of each connection for the office. Use the syntax-guided text editor to help you determine the syntax of the rule.

Hint You use the **conclude** action to conclude a value for an attribute. You refer to attributes by using the reserved word **the**, followed by the symbolic name of the attribute, followed by the reserved word **of**, followed by the name of the object, for example, the number-of-connections of office-1.

You use the equal sign to conclude a value for the number of connections attribute because you declared it to be an integer.

The rule should look like this:

unconditionally conclude that the number-of-connections of office-1 =
the count of each connection connected to office-1

Before you can test the rule, you need to create an office named **office-1**. If you do not create the office first, G2 will produce an error when you invoke the rule.

To create an office:

- ➔ On the Schematic Diagram workspace, use the Create Office button to create an office named **office-1**.

Invoking the Rule by Scanning

Now you need to invoke the rule to test whether the number of connections actually updates. The simplest way of testing a rule is by **scanning** the rule at a periodic interval. G2 invokes the rule at each scan interval.

While scanning rules is not always the most efficient way of invoking a rule, scanning is an easy way of testing a rule to verify that it performs the actions in the consequent as intended. Applications also use scanning when it is important to test the current value of an attribute or in alarm situations to monitor rules that would otherwise not be invoked.

For maximum efficiency, avoid scanning rules in real applications, except when absolutely necessary.

To invoke a rule by scanning:

- 1 Right-click the rule to display its menu, then choose **table**.

Hint If you click on the text of the rule, you will open the text editor for editing the rule. If this happens, cancel the editor and try clicking on the border again.

- 2 Edit the attribute named **scan-interval** to be **2 seconds**.

The inference engine begins scanning the rule immediately every two seconds when G2 is running.

- 3 Start G2 running.

Let's explore what is happening by showing when G2 is invoking the rule.

Highlighting Invoked Rules

You can cause G2 to highlight rules as they are invoked, which gives a visual indicator each time the rule is invoked. Highlighting invoked rules can be useful when you test rules. However, in large applications, highlighting invoked rules can degrade performance significantly; therefore, we recommend that you only highlight invoked rules for testing purposes.

To highlight invoked rules:

- ➔ Choose Run > Run Options > Highlight Invoked Rules.

The rule now flashes each time it is invoked, which is every two seconds.

Testing the Rule

Now you will test the rule by interactively adding and deleting connections.

To test the rule:

- 1 Display the Schematic Diagram workspace and display the table for **office-1**.

Notice that G2 has computed the number of connections to be 1. Leave the table visible while you interactively connect additional offices.

- 2 Create two additional offices, using the action button.

- 3 Connect each of the two offices to **office-1**, using the stubs.

Each time you connect an office, the **number-of-connections** attribute changes to reflect the current number of connections. Notice that the number does not update immediately upon creating the connection; it updates after G2 scans the rule, which happens once every two seconds.

When both offices are connected, the **number-of-connections** attribute is 3, which includes the two connections attached to the other offices and the one connection stub.

- 4 Create and connect another office to **office-1**.

The **number-of-connections** attribute increases by one.

- 5 Delete a connection and notice the value of the **number-of-connections** attribute.

Nothing happens. This is because the connection stub is still attached to the object.

- 6 Drag the connection stub into the office.

The **number-of-connections** attribute decreases by one.

- 7 Connect the disconnected office to another office on the diagram and display the table for that office.

The **number-of-connections** attribute does not change. This is because the rule is written specifically for the office named **office-1**.

Making the Rule More Robust and Efficient

G2 is concluding a value for the `number-of-connections` attribute of a particular office every two seconds, regardless of whether the connection stub is attached to another office and regardless of whether the number of connections changes.

Thus, you need to:

- Make the rule more robust by:
 - Concluding the number of connections for any office
 - Computing the number of connections based on the existence of actual connections, not just connection stubs
- Make the rule more efficient by invoking it only when the number of connections changes

You will improve the rule in the next lesson.

To save the application:

- ➔ Save the KB to a new file named `ch4.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you created a rule that counts the number of connections for an office, which you invoked by scanning.

You learned how to:

- Edit the `class-specific-attributes` of a class definition to use `is an integer` to declare an attribute that performs type checking
- Create an `unconditionally` rule that concludes a value for an integer attribute
- Use the `conclude` action to assign a value to an attribute
- Use the expression `the count of each` to count the number of connections of an object
- Use the `scan-interval` attribute of a rule to invoke the rule at a periodic interval
- Use the `Run > Run Options > Highlight Invoked Rules` menu choice to highlight rules each time they are invoked

Counting Connections for any Office

In this lesson, you will learn how to:

- Create generic rules that execute actions for any member of a class
- Create a rule that counts the number of actual connections between offices

Using the For Prefix to Create a Generic Rule

You can turn any rule into a **generic rule** that applies to:

- Any instance of a class
- One or more instances of a class that meet certain criteria, for example:
 - On a given workspace
 - Connected to another object
 - Nearest to another object



Always write generic rules that reason over entire classes of objects at any level in the class hierarchy, to simplify development and maintenance, avoid redundancy, and reuse rules in other applications.

To create a generic rule, you use the **for** prefix in front of the rule, followed by an expression that refers to a set of objects of a particular class. When you create a generic rule, you typically also use a local name in the rule to reference the class within the context of the rule. Local names in rules are similar to local names in action statements, methods, and procedures.

For example, to create a generic rule that applies to any instance of the office class, the **for** prefix looks like this, where **O** is the local name:

for any office O

As another example, in an application that monitors the flow of liquid through a tank, you might create a generic rule that applies to any pump connected at the input of any tank, as follows:

for any pump P connected at the input of any tank T

To follow proper indentation conventions for generic rules, place the **for** prefix alone on its own line with the text of the rule indented below the prefix.

Creating a Generic Rule for the Office Class

Now you will update the rule that concludes the number of connections to make the rule apply to any office.

To edit a rule to make it generic:

- 1 Click on the text of the rule to edit it.
- 2 Add a for prefix and a local name before the word **unconditionally** to make the rule apply to any instance of the office class.
- 3 Edit the body of the rule to refer to the local name.
- 4 Further edit the rule so that it concludes a value based on the count of each office connected to any office, as opposed to the count of each connection.

The rule should look like this:

```
for any office O
  unconditionally conclude that the number-of-connections of O =
  the count of each office connected to O
```

Now you can test the rule.

To test the rule:

- 1 Connect various offices together on the Schematic Diagram workspace.
- 2 Display the table for any office and verify the number of connections.
- 3 Delete a connection between two offices.

The next time the rule scans, the number of connections decreases by one because the office is no longer connected.

The rule is now much more robust because it applies to any office and it counts only actual connections, not connection stubs.

To save the application:

- ➔ Save the KB to the file named *ch4.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created a generic rule that applies to any office, which counts the number of actual connections.

You learned how to use:

- A for prefix to create a generic rule that applies to all instances of a class
- An **unconditionally** rule that concludes a value for an object regardless of any conditions being met

Using Event-Driven Processing

In this lesson, you will learn how to:

- Use event-driven processing to compute the number of connections whenever a connection is created or deleted
- Use an alternate form of a generic rule
- Use event-driven processing to compute the total cost per minute based on the number of connections, whenever the number of connections receives a value
- Simulate total cost by periodically updating the value, using a scan interval
- Add an attribute display to the office class definition, which shows the total cost attribute next to each office

Considering How to Invoke a Rule

In the previous lesson, you created an unconditionally rule that G2 invokes by using a scan interval. Scanning represents one way in which G2 invokes rules. However, scanning is generally not the most efficient way to invoke rules in G2. In large applications, scanning every rule in the knowledge base could hinder performance considerably. Now you will consider a more efficient way to invoke the rule that concludes the number of connections.

In the Basic Skills tutorial, you were introduced to the two basic ways in which G2 invokes rules:

- Event-driven processing, which invokes rules when G2 detects an event
- Data-driven processing, which invokes rules when an attribute receives a value

Let's ask some important questions about the rule that concludes a value for the number-of-connections attribute to determine how it should be invoked:

- Under what conditions should G2 invoke this rule to ensure that the number of connections is accurate?
- When does the knowledge base need to know the value of the number of connections?
- What event can you think of that might trigger the invocation of this rule?

You might have come to the conclusion that a more efficient way of invoking this rule is to conclude a value for the number of connections only when the number of connections changes. When does the number of connections change? Whenever the user creates or deletes a connection.

Thus, you could rewrite the rule to conclude a value for the number of connections only when G2 detects the event of creating and deleting connections.

This is an example of event-driven processing, which you implement by using a *whenever* rule.



To minimize processing and ensure accuracy, always use the most efficient way of invoking a rule that maintains robustness.

Detecting the Event of Creating a Connection

Now you will rewrite the rule to respond to the event of creating a connection. The event expression is *connected* to *detects* when a connection is created.

You can use the *is connected* to syntax to detect when any office is connected to any other office as follows:

```
for any office O1
for any office O2
  whenever O1 is connected to O2 then
  ...
```

Notice that the generic rule has two *for* clauses to distinguish between the two offices. The *for* clauses do not require any separator. In this example, the rule detects the event for O1 only; it does not detect the event for O2.

You can also use the *is connected* to syntax to detect when any *connection* is connected to any office as follows:

```
for any connection C
for any office O
  whenever C is connected to O then
  ...
```

In this case, the rule detects the event for both offices O that are connected to the connection C. Because you want every office to compute its *number-of-connections* attribute when a connection is created or deleted, you will use the second form of the rule.

First, you will create a generic rule that detects when any connection is connected to any office; therefore, the generic rule must refer to two different local names.

To rewrite a rule to respond to the event of creating a connection:

- 1 Edit the generic portion of the rule to refer to any connection and any office, using two different *for* clauses and two different local names.
- 2 Edit the rule to make it a *whenever* rule.
- 3 Use the syntax-guided text editor to enter the event expression that detects when a connection is created.
- 4 Display the table for the rule and notice that the *scan-interval* is automatically set to *none* so that the rule no longer scans.

You cannot have a scan interval on a whenever rule; the rule is invoked when the event is detected.

- 5 Create a borderless free text next to the rule that indicates when the rule is invoked.

The rule should look like this:

```
for any connection C
for any office O
  whenever C is connected to O then
    conclude that the number-of-connections of O =
      the count of each office connected to O
```

Now you can test the rule.

To test the rule:

- ➔ On the Schematic Diagram workspace, create connections between several offices.

Notice when the rule highlights. Whenever you interactively create a connection, G2 detects the event, invokes the rule, and immediately concludes a value for the number of connections. If you look very closely when the rule highlights, you can see that the rule is actually firing twice, once for each office. Also, notice that now there is no delay in concluding the value. Recall that previously when G2 invoked the rule by using a scan interval, the value only changed when the rule was scanned.

Thus, the whenever rule is more efficient, because it only invokes the rule when a connection is created. The whenever rule is also more robust because it concludes a value immediately whenever the number of connections changes.

Detecting the Event of Deleting a Connection

You also want the number of connections to be updated to reflect the current number of connections whenever a connection is deleted. Thus, you must create another rule that detects the event of deleting a connection, using the event expression `is disconnected from`.

You cannot use the `is disconnected from` syntax to detect when any *connection* is connected to any *office*. Therefore, you must detect whenever any *office* is disconnected from any other *office* as follows:

```
for any office O1
for any office O2
  whenever O1 is disconnected from O2 then
  . . .
```

This rule is invoked once for each office that is disconnected. Deleting a connection between two offices invokes the rule twice.

To create a rule that sets connections when offices are disconnected:

- 1 Create a generic rule that refers to any two offices, using two different for clauses and two different local names.
- 2 Make the rule a *whenever* rule to respond to the event of disconnecting one office from another office.
- 3 Create a borderless free text next to the rule that indicates when the rule is invoked.

The rule should look like this:

```

for any office O1
for any office O2
  whenever O1 is disconnected from O2 then
    conclude that the number-of-connections of O1 =
      the count of each office connected to O1

```

Now test the rule.

To test the rule:

- 1 On the Schematic Diagram, create two offices and display the table for one office.

The *number-of-connections* attribute is initially 0.

- 2 Now connect the two offices.

The *number-of-connections* attribute goes from 0 to 1.

- 3 Delete the connection between the two offices.

The *number-of-connections* attribute goes back to 0. The new rule fires immediately upon deleting the connection, which sets the *number-of-connections* attribute for each office back to 0.

Creating a Different Form of Generic Rule

In the previous two rules, you used two *for* clauses to refer to two different local names to make the rules generic. You can also write a generic rule by referring to the local names within the body of the rule. For example, the following *whenever* rule antecedent is equivalent to the one given above, which detects a connection event between a connection and an office:

```

whenever any connection C is connected to any office O

```

This form of a generic rule is identical to the other form, which uses a *for* clause.

To rewrite generic rules to use the alternate generic form:

- ➔ Remove the `for` clauses from each of the two `whenever` rules that detect connection events and refer to the local names in the body of the rule.

The rules should look like this:

whenever any connection C is connected to any office O then
conclude that the number-of-connections of O =
the count of each office connected to O

whenever any office O1 is disconnected from any office O2 then
conclude that the number-of-connections of O1 =
the count of each office connected to O1

Computing Total Cost Per Minute Whenever Number of Connections Changes

In the video conferencing prototype, you will simulate the total cost of an office based on the number of connected offices and a fixed connection cost per minute. Thus, to compute total cost, you first need to compute the total connection cost per minute, which is equal to the fixed cost times the number of connections.

What kind of rule can you write that will conclude a value for the total cost per minute based on the number of connections? You can create a `whenever` rule that G2 invokes whenever the number of connections changes, using the `receives a value` event expression.

To compute total cost per minute whenever number of connections changes:

- 1 On the Definitions workspace, edit the class definition of the office class to define an attribute named `connection-cost-per-minute`, which is a `quantity` with a default value of 0.1.

A `quantity` data type allows you to enter either floating point numbers or integers.
- 2 Add another attribute named `total-cost-per-minute`, whose value is a `float`.
- 3 On the Rules Workspace, create a generic rule that concludes the total cost per minute whenever the number of connections changes.

The total cost per minute is the product of the number of connections and the connection cost per minute

Hint Use the `for` prefix to create a generic `whenever` rule that responds to the event of receiving a value, using the `receives a value` expression.

- 4 Create a borderless free text next to the rule that indicates when the rule is invoked.

The rule should look like this:

```

for any office O
  whenever the number-of-connections of O receives a value then
    conclude that the total-cost-per-minute of O =
      the number-of-connections of O *
      the connection-cost-per-minute of O

```

Now you will test the rule.

To test the rule:

- ➔ Add and delete connections for an office and verify that the total-cost-per-minute attribute reflects the current number of connections.

Notice that the total cost per minute updates whenever you add or delete connections.

Computing Total Cost Per Minute Whenever the Connection Cost Changes

Suppose you wanted to enter a different connection cost per minute for a particular office. Would the total cost per minute of the office update to reflect the change? No, it would not, because the whenever rule only detects when the value of number-of-connections attribute changes, and it never updates the value of connection-cost-per-minute attribute.

To make the rule more robust still, you must detect when the value of the connection-cost-per-minute attribute changes as well.

You can detect multiple events with a whenever rule by using the word *or* between the conditions in the antecedent. G2 detects whenever *any* of the conditions is true. To detect whenever *all* conditions in the antecedent are true, you use the words *and* and *when* between multiple conditions in the antecedent.

To compute total cost per minute when connection cost per minute changes:

- ➔ Edit the whenever rule to detect when the number-of-connections *or* the connection-cost-per-minute attribute changes.

The rule should look like this:

```

for any office O
  whenever the number-of-connections of O receives a value or
    the connection-cost-per-minute of O receives a value then
    conclude that the total-cost-per-minute of O =
      the number-of-connections of O *
      the connection-cost-per-minute of O

```

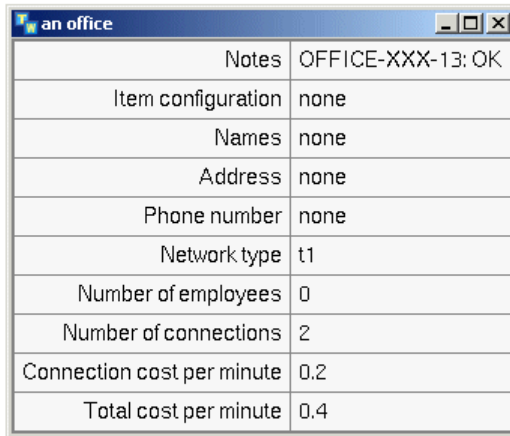
Now test the rule.

To test the rule:

- ➔ Edit the value of connection-cost-per-minute attribute in the table of any office.

The value of the total-cost-per-minute attribute now reflects the current value of the connection-cost-per-minute attribute.

Here is the table for an office whose connection cost per minute has been edited:



an office	
Notes	OFFICE-XXX-13: OK
Item configuration	none
Names	none
Address	none
Phone number	none
Network type	t1
Number of employees	0
Number of connections	2
Connection cost per minute	0.2
Total cost per minute	0.4

$$\text{total-cost-per-minute} = \text{connection-cost-per-minute} * \text{number-of-connections}$$

Simulating Total Cost by Scanning

Now you will simulate the total cost of each office by incrementing its value by the total cost per minute. You will use a scan interval to increment the value. In a real application, you would compute total cost based on the real-time video conferencing connection data.

Use scanning as a way of simulating the behavior of real-time applications during the development phase.

To simulate the total cost:

- 1 On the Definitions workspace, edit the class-specific-attributes attribute of the office class to create an attribute named total-cost, which has the quantity data type and whose default value is 0.
- 2 Create a new generic rule on the Rules Workspace that unconditionally concludes that the total cost of any office is the current value of total cost plus the total cost per minute.

Hint Remember to make the rule generic for all instances of the office class.

- 3 Edit the table for the rule to give it a scan interval of 2 seconds.
- 4 Show the attribute display and attribute name for the scan interval.

The rule should look like this:

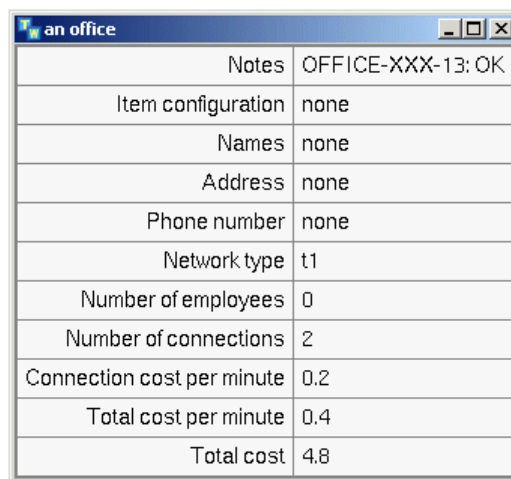
for any office O
 unconditionally conclude that the total-cost of O =
 the total-cost of O + the total-cost-per-minute of O

Now you can test the rule.

To test the rule:

- 1 Connect a number of offices to a central office.
- 2 Display the table for the central office and notice the value of the total-cost attribute.
- 3 Be sure to use the highlight invoked rules run option to verify that G2 is invoking the rule.

The total cost increments once every two seconds by a factor of the total cost per minute:



Notes	
Notes	OFFICE-XXX-13: OK
Item configuration	none
Names	none
Address	none
Phone number	none
Network type	t1
Number of employees	0
Number of connections	2
Connection cost per minute	0.2
Total cost per minute	0.4
Total cost	4.8

Hint If the total cost is not incrementing and the rule is not being invoked, make sure G2 is running.

- 4 Now delete all of the connections.

The number of connections goes to zero, and the total cost stops incrementing because the total cost per minute also goes to zero.

Creating an Attribute Display for Every Office

It is often useful to display the value of an attribute next to the icon each time you create an instance. You can create an attribute display for every instance of a class by editing the `attribute-initializations` attribute in the class definition. To do this, you initialize the system-defined attribute named `attribute-displays`.

Note While attribute displays can be useful during the development phase of an application, you do not usually use attribute displays when you deploy an application, for reasons of efficiency.

To add total cost as an attribute display for every instance of the office class:

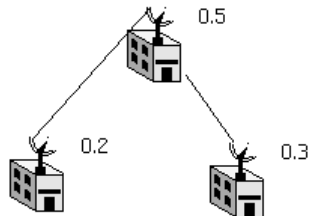
- 1 On the Definitions workspace, display the table for the office class definition.

Notice the `initializable-system-attributes` attribute, which includes the `attribute-displays` system-defined attribute.

- 2 Edit the `attribute-initializations` attribute to create an attribute display for the total cost attribute, as follows:

`attribute-displays: total-cost at standard position`

- 3 Delete all the offices on the schematic diagram and create a new diagram of connected offices.



The total cost attribute display appears next to every office and indicates the current value of the total cost attribute:

To save the application:

- ➔ Save the KB to the file named `ch4.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you edited the rule that concludes the number of connections to be a whenever rule, which G2 invokes whenever you create or delete a connection. You also created a whenever rule that concludes the total cost per minute, based

on the number of connections and the connection cost per minute. You then created a rule that simulated total cost by scanning. Finally, you created an attribute display for the total cost of every instance of the office class.

You learned how to use:

- A **whenever** rule to invoke a rule, using event detection
- The expressions **is connected to** and **is disconnected from** to detect the event of creating and deleting a connection
- Two generic **for** clauses to refer to two distinct classes and local names within a rule
- An alternate form of a generic rule that refers to the local names within the body of the rule
- The event expression **receives a value** to detect when an attribute value changes
- The **scan-interval** attribute of a rule to simulate incrementing the value of an attribute
- The **attribute-initializations** attribute of a class definition to initialize the system-defined **attribute-displays** attribute, which creates an attribute display next to every instance of a class

Using Data-Driven Processing

In this lesson, you will learn how to:

- Use data-driven processing to forward chain to a rule that monitors the total cost of each office, which checks if it is over budget
- Dynamically delete all connections when an office is over budget
- Create attributes for fixed values in the knowledge base

Using Forward Chaining to Monitor Total Cost and Delete Connections

Now that the office computes total cost, you can create a rule that compares the total cost against a fixed budget to determine whether an office is over budget. If the office is over the fixed budget amount, the rule will dynamically delete all of the office's connections.

Again you must consider how G2 will invoke the rule. The total cost of an office increments once every two seconds. When should you invoke the rule that monitors total cost? To make the rule both efficient and accurate, you should invoke the rule each time the value of total cost changes.

How do you do this in G2? When the value of any attribute in the antecedent of the rule changes, G2 automatically invokes the rule through a mechanism called **forward chaining**. To do this, you write an if rule that checks to see if the total cost is over a certain value. When the value of the total cost attribute changes, as it will every two seconds, G2 automatically invokes the rule that monitors total cost.



Use forward chaining to invoke rules that only need to be invoked when an attribute value changes.

Note that forward chaining represents only one way in which G2 uses data-driven processing to invoke rules.

Now you will create a rule that G2 invokes by forward chaining when the value of the total cost changes.

To create a rule that monitors total cost by using forward chaining:

- 1 Create a new rule on the Rules Workspace whose antecedent checks to see if the total cost of any office is over a fixed budget amount of 10.

Hint Use an if rule and remember to make the rule generic for all offices.

- 2 In the consequent of the rule, delete every connection connected to the office that is over budget.

Hint Remember, before you can delete a connection you must make the connection transient, using the **make transient** action. You use the **every** clause to refer to every connection connected to an office. You must also perform the actions in a particular order, using the **in order** statement. You use the **delete** action to delete every connection connected to an office.

- 3 Create a borderless free text that indicates that the rule is invoked via forward chaining when the value of total cost changes.

The rule should look like this:

```
for any office O
  if the total-cost of O > 10 then
    in order
      make every connection connected to O transient and
      delete every connection connected to O
```

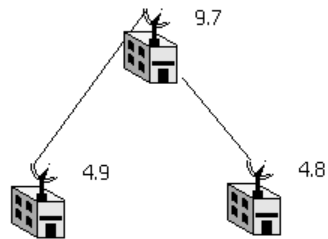
Now you can test the rule.

To test the rule:

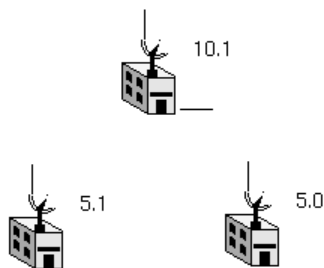
- 1 Delete all the connections to existing offices on the Schematic Diagram workspace and connect several offices to a central office.
Notice the value of the total cost of each office.
- 2 Be sure to highlight invoked rules so you can see when G2 invokes the rule that monitors total cost.
- 3 Make the Rules Workspace and Schematic Diagram workspace visible at the same time.

Tip You can shrink both workspaces to make them both visible.

- 4 Pause and resume G2 and notice what happens to the total cost.
G2 only scans rules when it is running.



The total cost of the central office increases incrementally by the amount of the total cost per minute, which depends on the number of connections. Each time the total cost changes, G2 forward chains to the rule that monitors total cost, which causes the rule to highlight.



When the value of the total cost attribute exceeds the budget you specified, in this case, 10, G2 takes the action in the consequent of the rule, which deletes the connections between the central office and all other offices.

Creating an Attribute for a Fixed Budget

In general, you create attributes for every value that a rule references, including fixed values. In the rule that monitors total cost, for example, you would create an attribute for the fixed budget amount, rather than referring to a specific value.

You create attributes for fixed values in an application for several reasons:

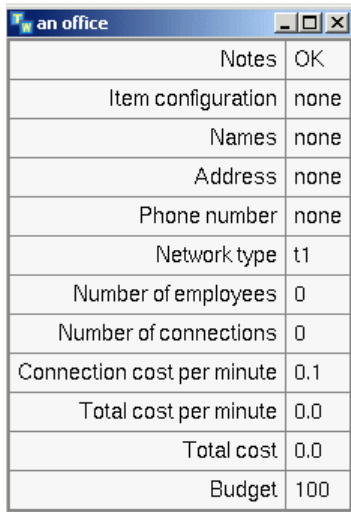
- To specify different values for each instance, as needed
- To provide a way of changing the value while the application is running
- To keep the application as generic as possible

You will now edit the rule that monitors total cost to refer to the budget.

To refer to the budget as an attribute in the rule:

- 1 Edit the `class-specific-attributes` attribute of the office class to create an attribute named `budget`, whose default value is 100.
- 2 Edit the rule to test the `total-cost` against the `budget` attribute, rather than a fixed number.

The table for an office looks like this:



an office	
Notes	OK
Item configuration	none
Names	none
Address	none
Phone number	none
Network type	t1
Number of employees	0
Number of connections	0
Connection cost per minute	0.1
Total cost per minute	0.0
Total cost	0.0
Budget	100

The rule should look like this:

```
for any office O
  if the total-cost of O > the budget of O then
    in order
      make every connection connected to O transient and
      delete every connection connected to O
```

Now test the rule.

To test the rule:

- 1 Delete all the offices on the schematic diagram and create and connect some new offices to a central office.
- 2 Edit the value of the `budget` attribute of the central office to be a number smaller than 100, such as 10.

You edit the budget for testing purposes so you do not have to wait until the total cost exceeds 100.

The behavior should be identical to the rule you just tested.

To save the application:

- ➔ Save the KB to the file named `ch4.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you created a rule that uses forward chaining to monitor total cost and delete every connection if the total cost exceeds the budget.

You learned how to use:

- An if rule that is invoked via forward chaining when the value of the attribute in the antecedent of the rule changes
- The expression `every connection connected to` to refer to every connection connected to an object
- An attribute in place of a fixed value in a rule to keep the rule generic

Keeping a History of Total Cost

In this lesson, you will learn how to:

- Create an attribute that keeps a history of the total cost
- Create a subclass of the `G2 parameter` class, which allows forward chaining and keeps a history

Using Variables and Parameters to Keep a History



You might want to keep a history of the total cost of each video conferencing office. You keep a history of an attribute's value for numerous reasons such as:

- Plotting data on a graph
- Logging and reporting
- Computing statistical information
- Performing **temporal reasoning**, which allows you to reason about objects over time

In G2, you use a special type of object to keep a history of values over time, called a **parameter**. You specify how long you want G2 to keep a history, and you can specify a default value. You can also use a similar type of object called a **variable**, which keeps a history and allows you to connect to real-time data.

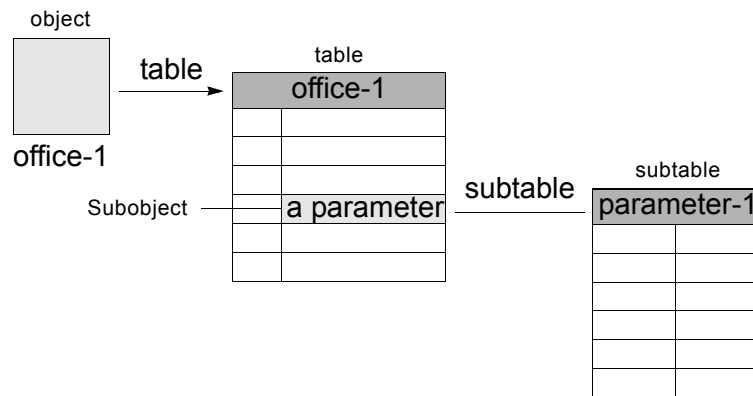
Use variables and parameters when you need to reason about objects over time.

Variables and parameters are two of the many built-in G2 classes, which you can use to describe your real-world objects. Just as G2 supports numerous data types for standard attributes, G2 also supports the same data types for variables and parameters. Thus, you can create a class-specific attribute that is a quantity (float or integer), float, integer, logical, symbolic, or text variable or parameter.

Variables and parameters are themselves objects. Thus, when you define a class-specific attribute to be a kind of variable or parameter, you are creating an object within an object, called a **subobject**. In object-oriented terms, an object that contains another object is called a **composite object**. Creating objects that contain other objects is another example of encapsulation, whereby you keep the complexity of an object hidden from other objects in the application.

A subobject has its own set of attributes, which you can view and edit by displaying the **subtable** for the object. A subtable is a table associated with a subobject, which is embedded in the object's table.

This figure illustrates the concepts of objects, subobjects, and subtables:



Use composite objects to organize knowledge in an application and take full advantage of the object-oriented nature of G2.

Using a Parameter to Keep a History of Total Cost

Now you will edit the definition of total cost in the office class to make the attribute be a kind of parameter. You use the **is given by** expression in the definition of the attribute to indicate that the attribute's value is an instance of a variable or parameter. In this case, you will specify the attribute to be given by an instance of the **quantitative-parameter** class, which is a parameter that keeps a history of floating point numbers or integers.

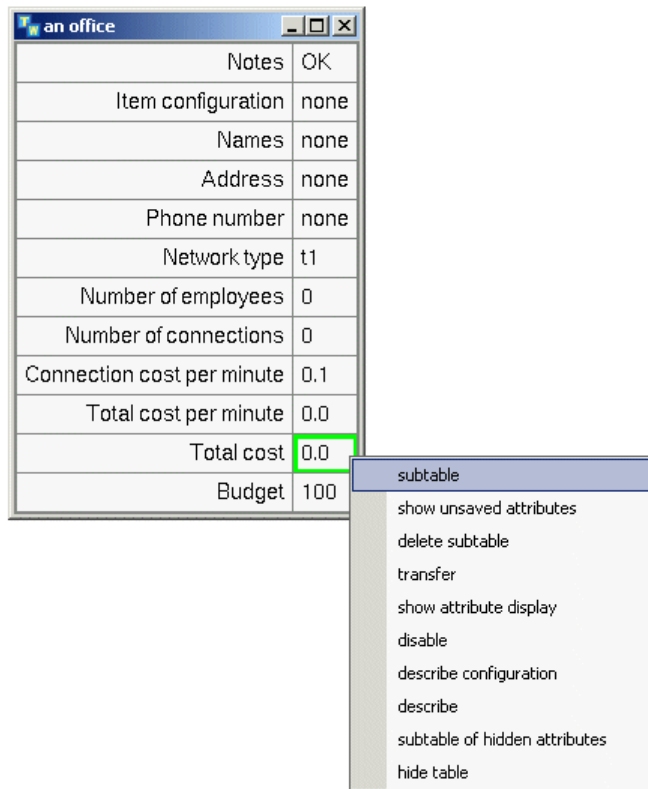
You specify the size of the history in the parameter's subtable, based on the number of points or the age of the data. You can also specify a default value for the parameter.

To create an attribute of the office that is a parameter:

- 1 Edit the **class-specific-attributes** attribute of the office class to declare the **total-cost** attribute to be given by a kind of quantitative parameter, using the following syntax:


```
total-cost is given by a quantitative-parameter, initially is given by a
quantitative-parameter
```
- 2 Create a new office on the Schematic Diagram workspace.
- 3 Display the table for the new office and click on the value of the **total-cost** attribute.

G2 displays a menu that allows you to display the subtable for the parameter:



- 4 Choose subtable to display the subtable for the quantitative parameter subobject:

The screenshot shows a window titled "a quantitative-parameter, the total co..." containing a table with the following data:

Options	do not forward chain
Notes	OK
Item configuration	none
Names	none
Tracing and breakpoints	default
Data type	quantity
Initial value	0.0
Last recorded value	0.0
History keeping spec	do not keep history

By default, the parameter does not keep a history.

- 5 Use the syntax-guided text editor to edit the `history-keeping-spec` attribute to specify that the parameter should keep a history of the last 100 points, as follows:

`keep history with maximum number of data points = 100`

- 6 Connect another office to the existing office and observe the behavior.

What happens? When the total cost of the office exceeds the budget now, nothing happens! Also notice that the rule that monitors total cost never highlights.

Note G2 does not automatically forward chain to quantitative, integer, float, or text variables or parameters. You must explicitly edit the definition of these types of parameters and variables to allow forward chaining.

Explicitly Allowing Forward Chaining

The default value of the `options` attribute for variables and parameters differs depending on the data type of the variable or parameter:

This type of variable or parameter...	Has this default value for the options attribute...
quantitative, integer, float, text	do not forward chain
logical, symbolic	do forward chain

You will now update the subtable of the quantitative parameter to allow forward chaining.

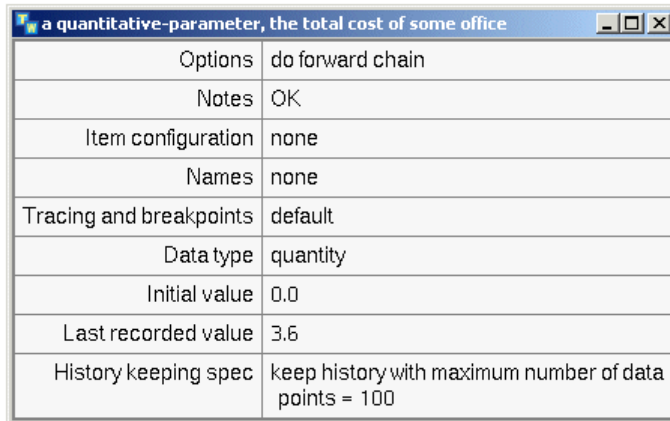
To edit the subtable of the parameter to allow forward chaining:

- 1 Display the table for an office.
- 2 Display the subtable for the total cost parameter and notice the attribute named `options`.

The default value is `do not forward chain`. This default tells G2 *not* to forward chain to rules that refer to this parameter in its antecedent. Thus, G2 does not invoke the rule that monitors total cost, even though the value of the parameter is changing.

- 3 Edit the options attribute to be do forward chain.

The subtable looks like this:



Options	do forward chain
Notes	OK
Item configuration	none
Names	none
Tracing and breakpoints	default
Data type	quantity
Initial value	0.0
Last recorded value	3.6
History keeping spec	keep history with maximum number of data points = 100

- 4 Restart the knowledge base, reconnect the offices, and observe the behavior now.

The total cost of every office now resets to 0 because it is a quantitative parameter. The rule that monitors total cost now highlights every time total cost receives a new value. When the total cost exceeds the budget, G2 deletes the connections.

Creating a Subclass of Parameter

What happens when the value of total cost exceeds the budget in one of the other offices or if you delete the current offices on the Schematic Diagram workspace and you create new offices?

Because you edited the definition of the total cost parameter in the subtable for a particular office, no other office has the **do forward chain** option specified and no other office currently keeps a history. Of course, what you want is for every office to forward chain to the rule that monitors total cost and for every office to keep a history.

Just as you can create an office class that is a subclass of the built-in **G2 object** class, you can create a subclass of any built-in G2 variable or parameter class. You can specify default values for the system-defined **options-for-parameter** and **history-keeping-spec** attributes of the subclass. You can then define total cost to be given by the subclass, rather than the built-in G2 class.

Just like any class, variables and parameters have an icon representation in G2, which you can create by creating an instance of the class. You can also create an instance of a built-in G2 class by using the **Workspace > New Object** menu choice.



Create subclasses of built-in G2 classes such as variables and parameters to extend the G2 object library.

To update the total cost attribute to be a subclass of parameter:

- 1 On the Definitions workspace, choose KB Workspace > New Definition > class-definition > class-definition to create a new class definition.

Note You use the `class-definition` menu choice to create user-defined subclasses of any G2 class, including objects, connections, messages, parameters, rules, and workspaces. The other types of definition objects are obsolete and exist for compatibility with earlier releases only.

- 2 Display the table for the class definition.
- 3 Specify the `class-name` to be `quantity-param` to give some indication of its type.
- 4 Specify the `direct-superior-classes` attribute to be `quantitative-parameter`.

Notice that when you specify the direct superior classes of a class definition, G2 automatically fills in the `class-inheritance-path` attribute. The class inheritance path indicates all the classes from which a class inherits its definition and the order in which they are inherited. Should conflicts arise, the first class in the list takes precedence. As you can see, the `quantity-param` class inherits first from itself, then from its direct superior class, then from the `parameter` class, and so on, until it reaches the highest-level G2 class, which is the `item` class. You will explore the G2 class hierarchy more in a moment.

First you need to specify the default options and history keeping specification for the parameter subclass. To specify the default value of a system-defined attribute, you use the `attribute-initializations` attribute.

Notice that the table for the quantity parameter class definition also defines an attribute named `initializable-system-attributes`. This attribute indicates the names of system attributes whose values you can initialize for the class. For this parameter, you will specify attribute initializations for the `options-for-parameter` and `history-keeping-spec` system attributes.

To specify the default options and history of the parameter class:

- 1 Edit the `attribute-initializations` attribute to specify the default options as follows:
 `options-for-parameter: do forward chain;`
- 2 Continue editing the attribute initializations to specify the default value for the history keeping spec as follows:
 `history-keeping-spec for variable-or-parameter: keep history with
 maximum number of data points = 100`

Hint Attribute initializations are separated by semi-colons.

- 3 Edit the `class-specific-attributes` attribute of the office class to declare the `total-cost` attribute to be given by `quantity-param`, rather than `quantitative-parameter`.

Now you can test the new definition.

To test the new object class definition:

- 1 Delete all the current offices on the Schematic Diagram workspace.
- 2 Create a new diagram with several connected sites and observe the behavior.
G2 deletes the connections when the total cost exceeds the budget.
- 3 To verify that all offices forward chain to the rule that monitors total cost, create a new diagram in which several offices are connected to one office, and several other offices are connected to another office.

G2 deletes the connections between the first office whose total cost exceeds the budget, then it deletes the connections between the second office whose total cost exceeds the budget. Thus, all offices forward chain to the rule that monitors total cost.

Showing the G2 Class Hierarchy

You can specify a built-in G2 class as the superior class of a user-defined class definition. For example, we specified the `object` class as the direct superior class of the office class, and you specified the `quantitative-parameter` class as the direct superior class of the `quantity-param` class.

Certain built-in G2 classes, such as `variable` and `parameter` classes, are subclasses of the `object` class. Other G2 classes are subclasses of the `item` class, which is the highest class in the G2 class hierarchy. In fact, most of the objects you create when you click on a G2 menu choice are G2 **items**, as opposed to G2 objects. For

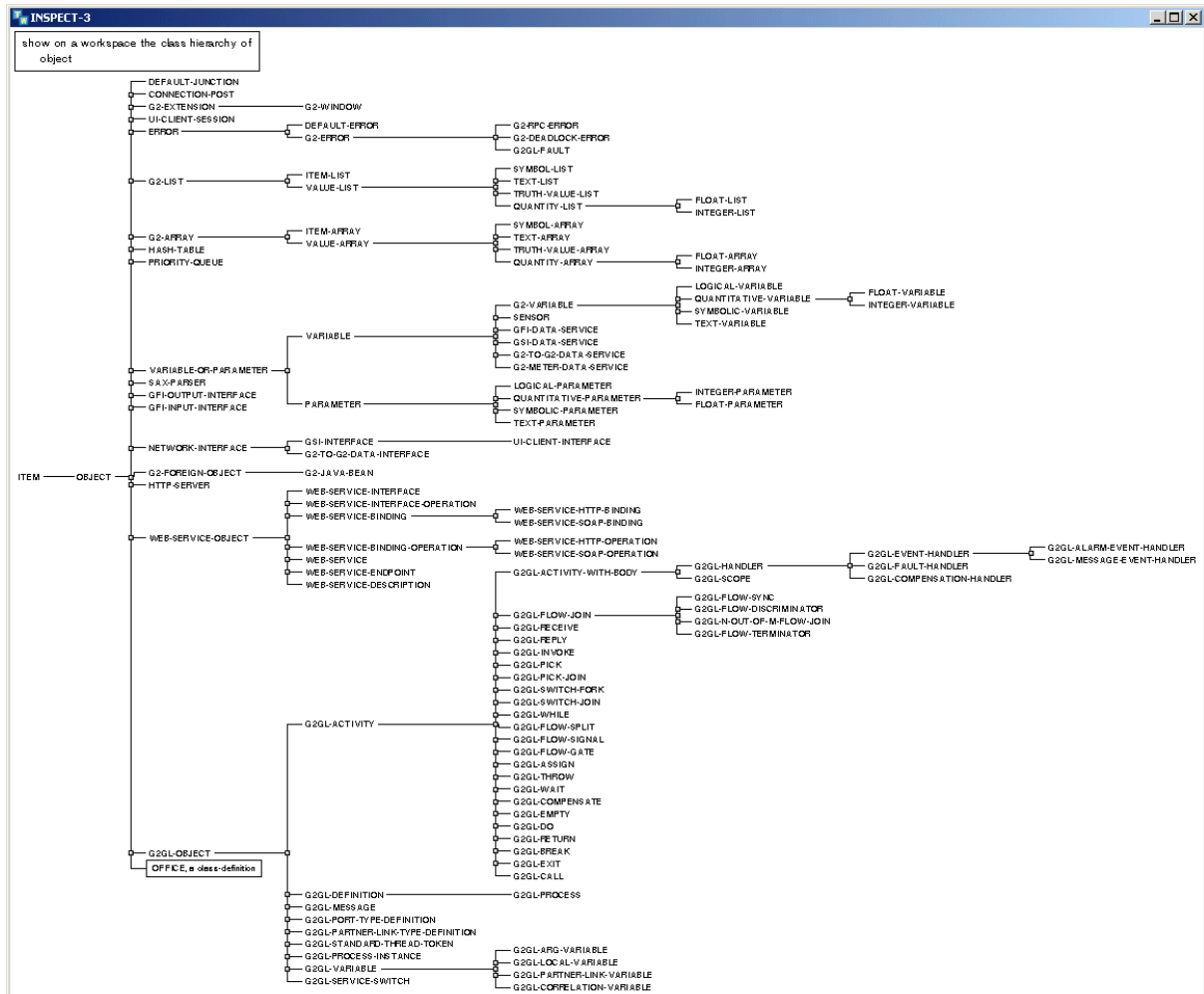
example, a workspace, a rule, and a method are all *items* in G2 terminology. However, keep in mind that in object-oriented terms, all G2 items are considered objects.

In addition to creating user-defined classes that inherit from the **object** class or any subclass of the object class, you can create user-defined classes that inherit from the **item** class or subclasses of the item class. For example, you might want to create a subclass of the rule class with a default scan interval of 2 seconds so that every rule of that class has a scan interval.

To see the classes from which a class definition can inherit its definition, use the Inspect facility to display the G2 class hierarchy.

To show the G2 class hierarchy:

- 1 Choose Tools > Inspect and enter the following command:
 show on a workspace the class hierarchy of object



G2 shows the numerous built-in classes from which a user-defined class can inherit its definition. The hierarchy also includes the two user-defined classes, **office** and **quantity-param**, which appear with a box around them to distinguish them from the built-in classes.

Notice the built-in variable and parameter classes, as well as classes that allow you to define **lists** and **arrays**, which are two other types of G2 data structures that allow you to store multiple pieces of information in a single attribute slot. A list has a variable length and an array has a fixed length.

- 2 Delete the object class hierarchy workspace.
- 3 Now use Inspect to show the class hierarchy of the **item** class.

The class hierarchy for the **item** class includes the hierarchy for the **object** class, plus all of the built-in classes you create by using menu choices. For example, free text, procedures, workspaces, and rules are all subclasses of the **item** class.

- 4 Delete the temporary workspace.

To save the application:

- ➔ Save the KB to the file named *ch4.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created a subclass of parameter that keeps a history of the total cost of an office and forward chains to a rule that monitors total cost.

You learned how to use:

- The **is given by** syntax in the **class-specific-attributes** of a class definition to specify that an attribute is given by a variable or parameter
- The **options** attribute of a parameter to specify whether the parameter forward chains to rules that reference the parameter in their antecedent
- The **history-keeping-spec** attribute of a parameter to specify how the parameter keeps a history
- The Workspace > New Definition > class-definition > class-definition menu choice to create a subclass of the quantitative parameter class
- The **attribute-initializations** attribute of a class to specify default options and history keeping spec for a subclass of parameter
- The Inspect facility to view the G2 class hierarchy

Creating Subclasses of Offices

In this lesson, you will learn how to:

- Create subclasses of objects with different default attribute values and different icons
- Create buttons that dynamically create instances of each subclass
- Apply reasoning at the appropriate level in the class hierarchy

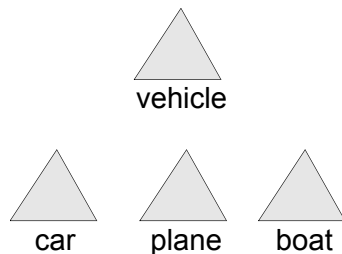
Creating a Subclass of a User-Defined Class

Just as you created a subclass of the built-in object class to create an office class and you created a subclass of the built-in parameter class to create a quantity parameter subclass, you can create a subclass of the user-defined office class.

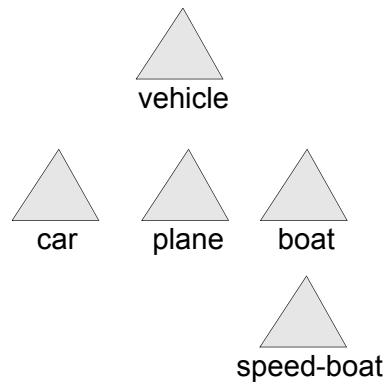
Create subclasses of user-defined classes to organize knowledge in an application, extend the user-defined object library, and reuse existing class definitions throughout an application.

You create subclasses of user-defined classes for two general purposes:

- **Generalization** or **abstraction** – to define similar information in one location, which subclasses inherit automatically; the attributes of the class are an abstraction of the common characteristics of a set of subclasses:



- **Differentiation or specialization** — to create subclasses of a common class to define exceptions and special cases:



For example, you created a single class named office, which generalizes the common attributes and methods of a video conferencing office. Because all video conferencing offices require the same set of attributes for computing the number of connections and total cost, and the same method for creating connections for the office, you defined these attributes and methods on a single class, the office class.

Now, suppose you wanted to differentiate offices from one another, for example, by size, where small offices have a smaller budget than large offices. You can easily do this by creating subclasses of office.

When defining subclasses, define only the unique characteristics and behaviors of the subclass, such as new attributes, unique default values for existing attributes, unique icons, and unique methods.

Now you will create two subclasses of the office class that define different default values for budget and different icons.

To create two subclasses of the office class with unique icons and defaults:

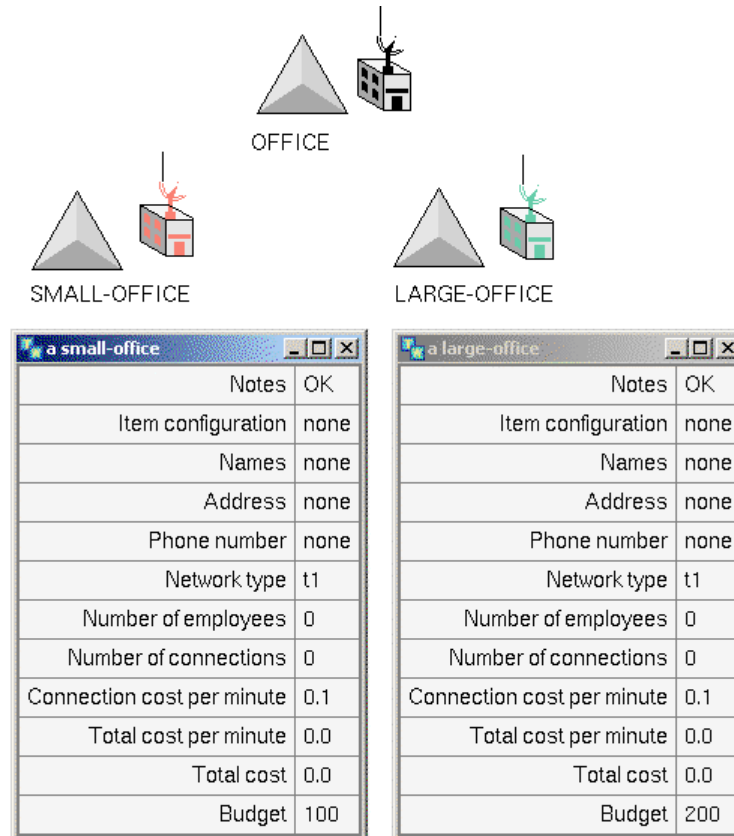
- 1 On the Definitions workspace, create a class definition named **small-office**, whose direct superior class is the **office** class.
- 2 Create another class definition named **large-office**, also a subclass of the **office** class.
- 3 Edit the icon of one of the classes to make the status icon layer a different color.

Use the **edit icon** menu choice on the class definition to edit its icon. For more information on editing icons, see [Editing the Icon](#).

- 4 Edit the class definition of each of subclass to specify a default value of 100 for the **budget** attribute of the small office and a default value of 200 for the same attribute of the large office.

Hint Remember that you override the default value of an existing attribute of a class in the `attribute-initializations` attribute of the class. Use the `initially is` syntax to specify the default value.

- To verify that the icons and `budget` attributes are unique, create an instance of each class on the Definitions workspace by selecting `create instance` from the class-definition tables:



To create subclasses of the office class, all you had to define were the unique features of the subclass, in this case, the icon and the default value of a single attribute, `budget`. The subclasses inherited the rest of their class definitions from their superior class.

Creating Instances of Each Subclass

Currently, you have a single action button, which creates an instance of the office class. Now that you have created subclasses of office, there is no longer a need for an action button that creates an instance of the office class. Instead, you now need two specific action buttons, which create instances of the small office and the large

office classes. Thus, when an action refers to a specific class, you must reference the subclass directly.

To create two action buttons that create instances of each subclass of office:

- 1 On the Schematic Diagram workspace, edit the Create Office action button to create an instance of the `small-office` and label the action button Create Small Office:

Create Small Office

- 2 Create another action button labelled Create Large Office that creates an instance of the `large-office`:

Create Large Office

The most efficient way to do this is to clone the existing action button and edit the label and action.

- 3 Test each action button and verify that the icons and budgets are unique.

Verifying that the Rules Apply to the Subclasses

When you create subclasses of user-defined objects, you must consider at what level in the hierarchy you will reason about the objects in the class.

To avoid redundancy in rules, always reason about the highest class in the hierarchy that makes sense.

In the video conferencing prototype, you have created a number of rules that apply to the office class. Because small office and large office are both subclasses of the office class, these rules will continue to work for the subclasses. Similarly, you created a method that creates a connection for the office class, which you called by using a user menu choice. The method also continues to work for both subclasses because the method is inherited by each subclass.

Thus, in this application, it is not only unnecessary but also inefficient to reference each subclass because doing so would require twice as many rules and methods.

To verify that the rules and methods continue to work for the subclasses:

- 1 Create and connect a number of offices of different types and run the prototype.

G2 deletes the connections of small offices when their total cost reaches 100, and G2 deletes the connections of large offices when their total cost reaches 200. Thus, the rules continue to work for both subclasses without modification.

- 2 Create a connection by choosing **create connection**.

The method that each subclass calls when you choose the **create connection** user menu choice continues to work for both subclasses. Thus, methods are also inherited by the subclasses of an object.

Overriding the Default Method of a Class

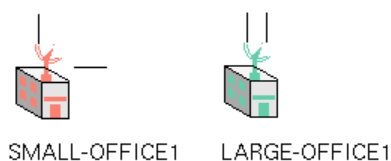
Suppose you wanted the behavior of a subclass to be slightly different from that of its superior class. You can create a new method of the same name to define the unique behavior of the subclass. For example, you might want the method that creates a connection for a small office to create the connection on the right of the icon, rather than on top.

To override the default method of a class:

- 1 Choose **Workspace > New Definition > procedure > method** to create a new method on the Definitions workspace.
- 2 Copy and paste the definition of the existing method named **create-connection** into the definition of the new method, which will also be called **create-connection**.
- 3 Edit the argument to the new method to refer to the small office class, rather than the office class.
- 4 Edit the body of the method to make the connection lead out of the right side of the icon, rather than the top.

The class-qualified name of the method is `small-office::create-connection`.

- 5 Create a new small office on the Schematic Diagram workspace and add a connection by choosing **create connection**.



The new connection for a small office is coming out of the right side of the icon, whereas the new connection for a large office is coming out of the top. This is because the large office inherits its **create-connection** method from the office class, while the small office has redefined its **create-connection** method.

Imagine if you had created numerous subclasses of office. Editing the behavior of a single subclass would simply be a matter of creating a unique method for the subclass. The unique method can also execute the method of the default class as part of its definition.

To save the application:

- ➔ Save the KB to the file named *ch4.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created two subclasses of the office class with unique default values for budget and a unique icon. You created action buttons that create instances of each subclass, and you created a specific method for one of the subclasses to override the default behavior.

You learned how to:

- Use the Workspace > New Definition > class-definition > class-definition menu choice to create a new class definition that is a subclass of an existing user-defined class
- Use the **attribute-initializations** attribute of the class definition to override the default value of an attribute of the direct superior class
- Use action buttons to create instances of specific subclasses by referring to the subclass in the **action** attribute
- Refer to the superior class of a class hierarchy in rules and methods so the rules and methods apply to all subclasses in the same way
- Specify a subclass as the first argument to a method to override the default behavior of a subclass

Disabling Rule Highlighting

Because you are finished testing the knowledge base itself, you can disable rule highlighting for the next phase of the tutorial, which is building a user interface.

To disable rule highlighting:

- ➔ If you haven't already, disable rule highlighting by choosing Run > Run Options > Do Not Highlight Invoked Rules.

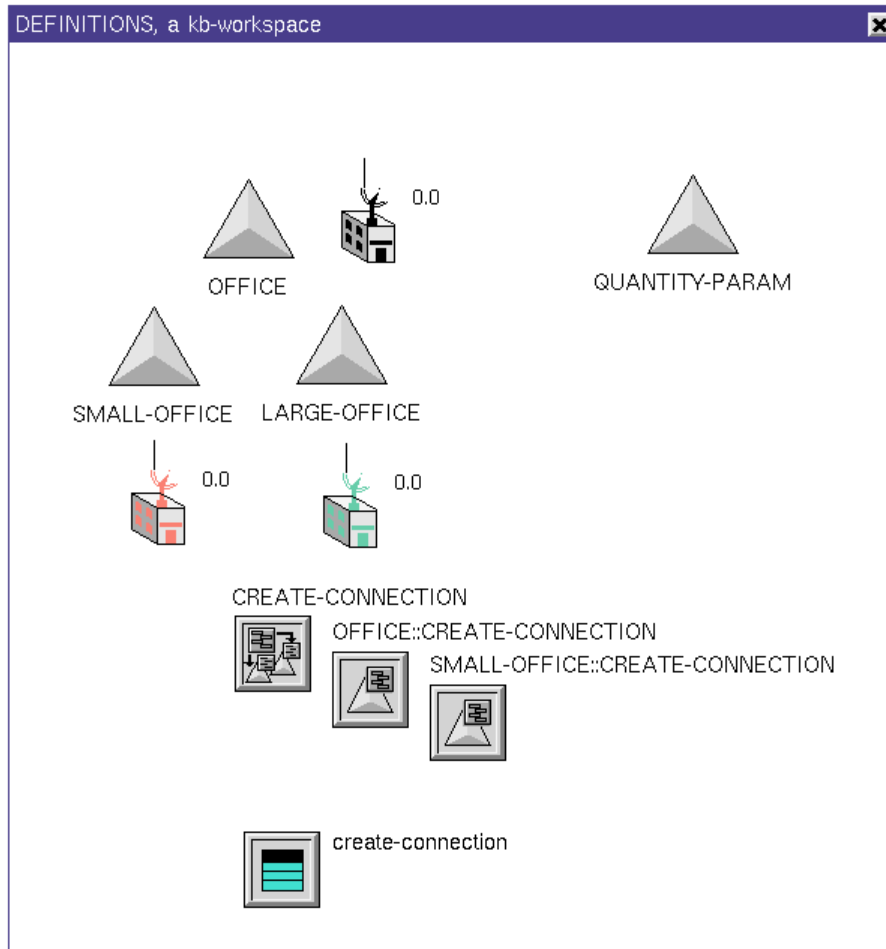
Summary

In this tutorial, you learned:

- How to use the **is a** syntax to create class-specific attributes that perform type checking, the **initially** syntax to create class-specific attributes with default values, and the **is given by** syntax to create class-specific attributes that contain instances of a class
- How to keep a history of attribute values by using variables and parameters
- How to create a subclass of variable or parameter with specific options and history keeping spec
- How to conclude a value for an attribute, using the **conclude** action
- How to count the number of connections for an object, using the numeric expression **the count of each**
- How to invoke rules by scanning for testing and simulation purposes
- How to highlight invoked rules for testing purposes
- When to use generic rules and how you create them, using the **for** prefix
- How to use the event expressions **is connected to**, **is disconnected from**, and **receives a value** to use event-driven processing to invoke rules
- About data-driven processing and forward chaining to invoke if rules that monitor the value of an attribute against a fixed value
- How to dynamically delete all connections, using the **delete** action
- How to create an attribute display for all instances of a class by initializing the **attribute-displays** system attribute
- About the built-in G2 classes and about the class hierarchy of the built-in **object** class and **item** class
- How to create subclasses of user-defined classes and override their attribute value defaults, icons, and methods

Solutions

The Definitions workspace looks like this:

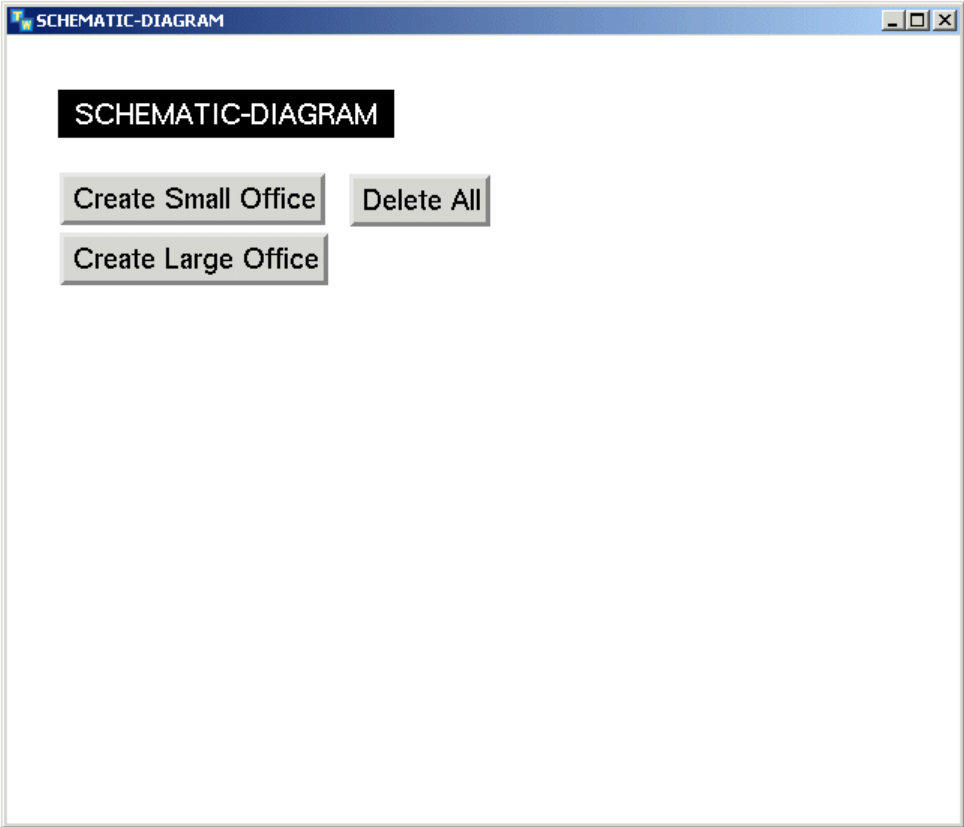


The Rules Workspace looks like this:

The screenshot shows a window titled "RULES-WORKSPACE" with a sub-header "RULES-WORKSPACE". It contains five rules, each in a box, with their respective triggers listed to the right.

Rule Text	Trigger
whenever any connection C is connected to any office O then conclude that the number-of-connections of O = the count of each office connected to O	invoked whenever a connection is created
whenever any office O1 is disconnected from any office O2 then conclude that the number-of-connections of O1 = the count of each office connected to O1 and conclude that the number-of-connections of O2 = the count of each office connected to O2	invoked whenever a connection is deleted
for any office O whenever the number-of-connections of O receives a value or the connection-cost-per-minute of O receives a value then conclude that the total-cost-per-minute of O = the number-of-connections of O * the connection-cost-per-minute of O	invoked whenever number-of-connections receives a value
for any office O unconditionally conclude that the total-cost of O = the total-cost of O + the total-cost-per-minute of O	Scan interval 2 seconds
for any office O if the total-cost of O > the budget of O then in order make every connection connected to O transient and delete every connection connected to O	invoked via forward chaining whenever total-cost changes

The Schematic Diagram workspace looks like this:



Building a User Interface

Shows how to create a simple end user interface for the video conferencing prototype that includes subworkspaces of objects, charts, readouts, messages, animation, and user modes.

Goals of the User Interface	152
Loading the Knowledge Base	152
Creating a Subworkspace for an Object	152
Displaying Details on the Subworkspace of an Object	157
Sending a Message to the Operator	165
Animating Objects	176
Making Workspaces Attractive and Informative	180
Showing Workspaces Programmatically	185
Configuring the User Interface	187
Running the Prototype	194
Loading the Finished Application	198
Summary	199
Solutions	200



Goals of the User Interface

In the previous tutorial, you created a knowledge base for a video conferencing prototype that monitors total cost, determines whether an office is over budget, and if so, automatically deletes the connections. In this tutorial, you will build a simple user interface that:

- Creates a hierarchical view of the application by creating a subworkspace for each office that contains details about the office.
- Provides visual indicators of the value of total cost over time.
- Informs the operator when an office is over budget.
- Animates an office when its total cost is approaching the budget.
- Configures the behavior of offices for different classes of users.

Loading the Knowledge Base

You will start from the finished tutorial you created in the previous tutorial and build on this application to create a user interface.

To load the knowledge base:

- ➔ Load the KB named *ch4.kb* that has your initials appended to it or load the KB named *ch5.kb* to load the solution KB that is the starting point for this tutorial.

Creating a Subworkspace for an Object

In this lesson, you will learn how to:

- Create a subworkspace for an object.
- Dynamically create an object with a subworkspace by cloning a master object that defines a subworkspace.

What is a Subworkspace?

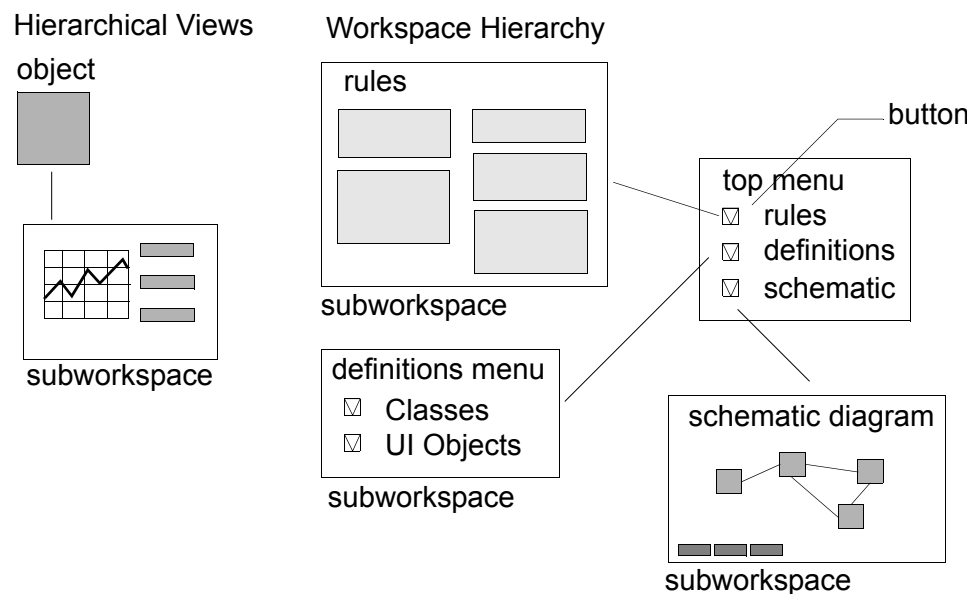
In the previous tutorial, you learned how to create a composite object in G2 by creating an attribute of an object that is given by a variable or parameter.

Now you will learn another way to create a composite object by creating a subworkspace of an object. A **subworkspace** is a workspace that is associated with a particular object and is “below” the workspace on which the object resides.

You use subworkspaces to create:

- **Hierarchical views** of an application, where the details of an object are hidden on its subworkspace.
- A **workspace hierarchy**, where you use button objects to navigate to different workspaces and submenus in an application.
- **Activatable subworkspaces** on which you place scanned rules, which you activate and deactivate when certain conditions are met.

This picture illustrates two of the three main uses of subworkspace:



Create subworkspaces to encapsulate knowledge by hiding complexity on the subworkspace of an object.

Just as you can create a class hierarchy by inheriting definitions from superior classes, you can create an **object hierarchy** by creating subworkspaces of objects. The subworkspace of an object can contain an object with a subworkspace, and so on, to create a workspace hierarchy that is many levels deep.

An object with a subworkspace is called a **superior object**, because it is located above the subworkspace in the object hierarchy.

Creating a Master Object with a Subworkspace

The goal of the end user interface is to enable the end user to create an action button that programmatically creates an office with a subworkspace. To create an object with a subworkspace programmatically, first you create a **master object**, then you interactively create a subworkspace for the master. Once you have defined the master, you create instances by programmatically cloning the master. The cloned instance automatically has a subworkspace.

To create a master object with a subworkspace:

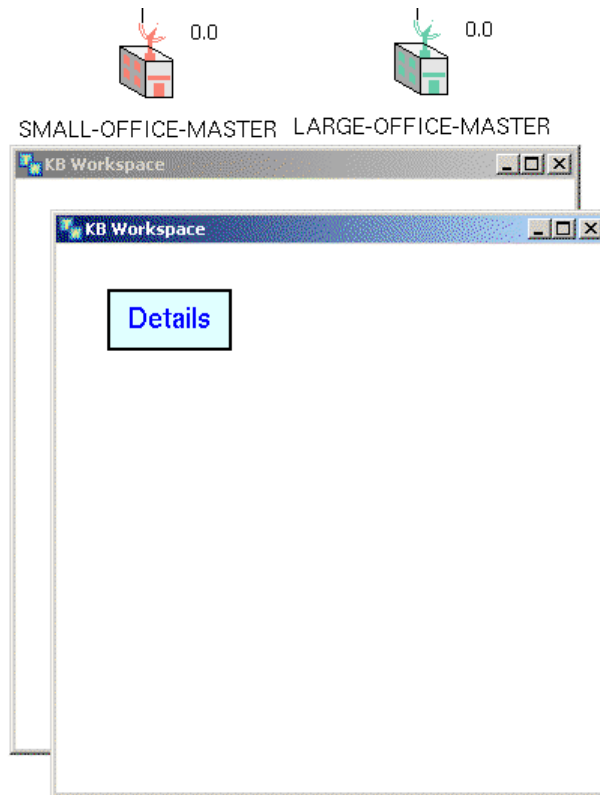
- 1 On the Definitions workspace, create an instance of the **small-office** class and an instance of the **large-office** class.
- 2 Display the menu for the small office.
- 3 Choose **create subworkspace**.
G2 creates a subworkspace associated with the object.
- 4 Use free text to label the subworkspace **details**.
- 5 Create a subworkspace for the large office instance, clone the free text from the subworkspace of the small office, and place it on the subworkspace of the large office.
- 6 Hide the subworkspaces.
- 7 Name the small office **small-office-master** and the name large office **large-office-master**.

Now you will go to the subworkspace.

To go to the subworkspace:

- 1 Choose **go to subworkspace** on each master office to display the subworkspaces again.

The small office master, large office master, and their subworkspace look like this:



- 2 Hide the subworkspaces.

Creating an Object with a Subworkspace Dynamically

To create an object with a subworkspace dynamically, you use the **create by cloning** action, which creates an instance of a class by cloning a master object. Cloning the master object also clones its subworkspace and all the objects on it. You will now edit the buttons that dynamically create each type of office to create the offices by cloning the masters.

To create instances of each type of office by cloning each master office:

- 1 On the Schematic Diagram workspace, edit the button that creates an instance of the small office class to use the **create by cloning** action, to clone the **small-office-master**:

Create Small Office

Use the syntax-guided text editor to help you. The syntax should look like this:

```
create a small-office O by cloning small-office-master
```

- 2 Edit the button that creates a large office to clone the **large-office-master**:

Create Large Office

- 3 Test the buttons and verify that each type of office now has a subworkspace.
- 4 Hide the subworkspaces.

The action attribute of the Create Small Office button should look like this:

```
in order
  create a small-office O by cloning small-office-master and
  transfer O to this workspace at
    (random (-300, 300), random (-300, 300)) and
  make O permanent
```

To save the application:

- ➔ Save the KB to a new file named *ch5.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created action buttons that create offices with subworkspaces by cloning a master.

You learned how to use:

- The **create subworkspace** menu choice on an object to create a subworkspace of the superior object.
- The **create by cloning** action to create an instance of a class by cloning an existing instance with a subworkspace.

Displaying Details on the Subworkspace of an Object

In this lesson, you will learn how to:

- Create end user displays on the subworkspace of an object that show the details of the object.
- Create a readout that shows the value of an attribute.
- Create a trend chart that displays the history of an attribute value over time.

Creating End User Displays

An end user interface typically includes end user **displays**, which provide visual representations of real-time numeric data that the knowledge base receives and computes. G2 supports numerous types of end user displays:

- **Readout tables** display numeric data in a small table.
- **Digital clocks** display the current G2 time in a small table.
- **Dials** and **meters** display numeric data in a circular dial or a vertical meter.
- **Trend charts** and **graphs** plot historical data over time.

You often create end user displays on the subworkspace of an object to encapsulate the details of the object. To display the details about an object, the end user simply displays the subworkspace of the object. However, you can also create end user displays on any workspace in the application.

Use displays to provide visual feedback to the end user about real-time data and the status of the overall application.

End user displays update their values according to their **update interval**, which you specify. The update interval tells G2 how often to update the current value of the display.

In general, the update interval of a display is independent of the interval at which the knowledge base updates the corresponding data values. Thus, you might have a parameter attribute that is receiving a value from a rule once every two seconds and a trend chart that is displaying the parameter value once every five seconds. G2 displays the current value of the parameter as of the current time interval of the display.

Note There is one exception to this rule. You can use readout tables to *control* the interval at which a variable attribute updates its value. If the readout table is visible, the update interval of the readout table can *cause* G2 to update the value of the variable attribute. This is another example of forward chaining.

Creating a Readout for the Small Office Master



You might want to create a visual indicator of the number of connections for a particular office. You could place this indicator on the subworkspace of the object to encapsulate details about the object.

You create the display on the subworkspace of each master office. As mentioned earlier, the object whose attribute values you are displaying is the superior object of the workspace, because the object is above the subworkspace in the object hierarchy. To refer to the superior object of a subworkspace, use the expression `the item superior to`.

To create a readout on the subworkspace of the small office master:

- 1 Display the subworkspace of the `small-office-master` on the Definitions workspace.
- 2 Choose `Workspace > New Display > readout-table > readout-table` to create a readout table end user display.
- 3 Display the table for the readout.
- 4 Edit the `expression-to-display` attribute to refer to the `number-of-connections` attribute of the object superior to the current workspace.

Hint Use the syntax-guided text editor to help you enter the expression. You refer to the object that is superior to the subworkspace as `the item superior to`, and you refer to the subworkspace as `this workspace`.

The `expression-to-display` attribute should look like this:

`the number-of-connections of the item superior to this workspace`

The default label for the readout is the expression to display, which you can edit.

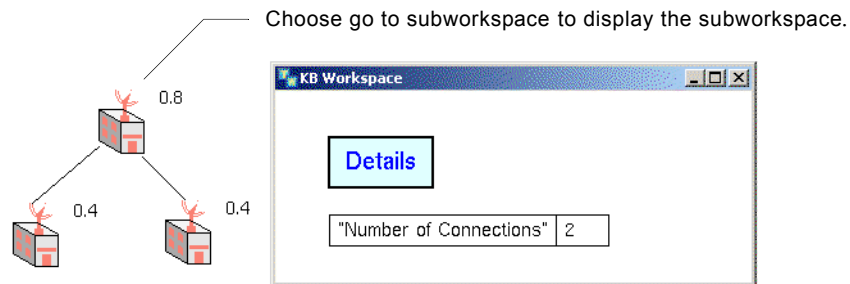
- 5 Edit the `label-to-display` attribute to be "Number of Connections".
- 6 Edit the `display-update-interval` attribute to be 2 seconds so that the readout table updates at the same interval at which the data updates.
- 7 Hide the table and the subworkspace.

Now you can test the readout table by interactively creating small offices.

To test the readout table for the small office:

- 1 On the Schematic Diagram workspace, create and connect several small offices.
- 2 Display the subworkspace of one of the offices and verify that the readout is accurate.

G2 updates the readout once every two seconds to show the number of current connections:



Creating a Readout for the Large Office Master

Now that you have tested the readout for the small office master, you can clone the readout and place it on the subworkspace of the large office master.

To create a readout on the subworkspace of the large office master:

- 1 On the Definitions workspace, display the subworkspace of the small office master and the large office master.
- 2 Clone the readout from the subworkspace of the small office master and place it on the subworkspace of the large office master.

Notice that the same generic action applies to both subclasses because the reference is to the item superior to this workspace.

- 3 Hide both subworkspaces.

Now test the readout table for the large office master.

To test the readout table for the large office:

- 1 On the Schematic Diagram workspace, create and connect several large offices.
- 2 Display the subworkspace of one of the offices and verify that the readout is accurate.
- 3 Hide the subworkspace.

Using Readout Tables to Invoke Rules

Currently, the `number-of-connections` attribute of the office is a simple typed attribute. If the `number-of-connections` attribute of the office class were given by a parameter or variable, you could use the readout table to invoke the rule that computes the number of connections of an office by forward chaining. Just as G2 forward chains to the rule that breaks existing connections when the total cost receives a value, G2 would forward chain to the rule that computes the number of connections each time the readout table updates.

However, you can only use a readout table to invoke rules by forward chaining *when the readout table is visible*. Thus, in this prototype, it does not make sense to invoke the rule this way, because most of the time the subworkspace is not visible.

Creating a Trend Chart that Plots Total Cost

One of the primary reasons for keeping a history of data values in an application is to provide visual feedback about current and historical data to the end user. G2 supports two basic ways of plotting historical data:

- **Trend charts**, which plot any number of data values and give complete control over the layout of the chart.
- **Graphs**, which only plot a single value and give you less control.

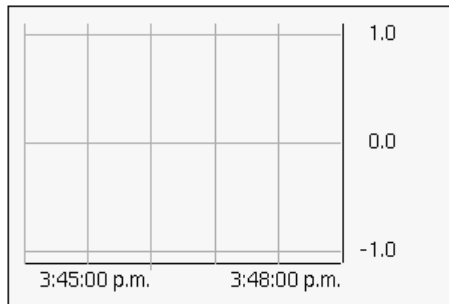
A trend chart consists of numerous subobjects, which define various information about the trend chart. Each trend chart has:

- One or more **plots**, which are the data values to plot.
- One or more **value axes**, which are the vertical axis of each plot.
- A **time axis**, which is the horizontal axis.
- One or more **point formats**, which are the graphical elements at each data point for each plot.
- One or more **connector formats**, which are the lines between the points for each plot.
- A **trend chart format**, which controls the background color of the chart.

In the video conferencing prototype, you might want to plot the total cost of an office over time and place the plot on the subworkspace of the office.

To create a trend chart on the subworkspace of the small office master:

- 1 On the Definitions workspace, display the subworkspace of the small office master.
- 2 With the subworkspace selected, choose **Workspace > New Display > trend-chart** to create a trend chart on the subworkspace:



- 3 Display the table for the trend chart and edit the title attribute to be **Total Cost**. You will use a single plot to display the total cost of an office over time.

To specify the plot for a trend chart:

- 1 Choose **plots > subtable** from the trend chart menu to display the subtable for the plot.

Note You can also display the subtable for a plot by displaying the table for the trend chart, clicking on the value of the **plots** attribute, and choosing the **subtables** menu choice. This menu shows two menu choices, **Plot Defaults** and **Plot #1**.

- 2 To plot the value of the **total-cost** attribute, edit the **expression** attribute to refer to the object that is superior to the current workspace.

Hint The syntax for this expression is similar to that of the readout table you created earlier.

- 3 To avoid displaying the expression in the trend chart, edit the **include-in-legend?** attribute to be **no** by choosing **change to "no"** for the attribute.
- 4 Edit the **update-interval** attribute to be **2 seconds** so that the chart updates with the same frequency that the values change.

By default, the value axis is floating, which means the vertical axis adjusts as the data values plot. Next, you will create a fixed value axis that corresponds to the maximum budget for the office.

To edit the value axis for a trend chart:

- 1 Display the menu for the trend chart and choose **value axes > subtable**.
- 2 Edit the **range-mode** attribute to be **fixed**.
- 3 Edit the **range-bounds** attribute to be from 0 to 100, which is the range of allowable values for total cost before a small office is over budget.

Now, you will update the background color of the trend chart.

To update the background color of a trend chart:

- 1 To edit the default background color of the trend chart, choose **trend chart format subtable**.
- 2 Edit the **data-window-background-color** attribute to be any G2 color.

Hint Click the **color** prompt in the editor to see the names of available colors.

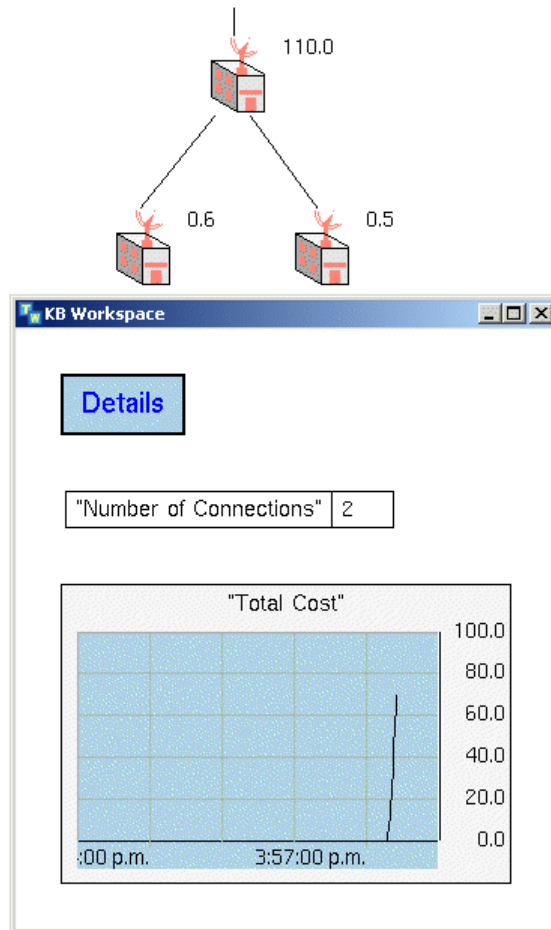
- 3 Hide the detail subworkspace of the small office master.

Next, you will test the trend chart.

To test the trend chart:

- 1 On the Schematic Diagram workspace, create and connect several small offices.
- 2 Edit the connection cost per minute of one of the offices to be 10 to increment total cost more rapidly and to see the plot.
- 3 Display the subworkspace of this office to verify that the total cost is plotting on the trend chart.

Your diagram might look something like this:



- 4 Hide the subworkspaces and reset G2.

Finally, you can clone the trend chart for the large office master.

To create a trend chart on the subworkspace of the large office master:

- 1 On the Definitions workspace, clone the trend chart from the subworkspace of the small office master and place it on the subworkspace of the large office master.
- 2 Edit the value axis of the trend chart on the subworkspace of the large office master to specify a range bounds of 0 to 200.
- 3 Test the trend chart for the large offices by creating and connecting several large offices on the Schematic Diagram and showing the subworkspace.
- 4 Hide the subworkspaces and reset G2.

Making the Application More Realistic

Until now you have tested the prototype by deleting the connections between offices when the total cost exceeded a relatively small number. Now you will update the default budgets for each type of office to be a larger number to see the effect over time.

To update the default budget for each office type and the related trend charts:

- 1 In the class definition for the **small-office** class, edit the **attribute-initializations** attribute to specify a default budget of 1000.
- 2 Edit the default budget for the large office to be 2000.
- 3 Update the range bounds in the value axes for the trend charts associated with each master office to reflect the new budget limits.
- 4 On the Schematic Diagram workspace, create and connect some large and small offices and show the subworkspaces to see the plots over time.
- 5 Hide the subworkspaces and reset G2.

The trend charts now plot data over a longer period of time.

To save the application:

- ➔ Save the KB to the file named *ch5.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created subworkspaces for each type of office and placed a readout table and a trend chart on the subworkspaces.

You learned how to use:

- The **Workspace > New Display > readout-table** menu choice to create a readout table display that shows the current value of an attribute of an object.
- Use the **Workspace > New Display > trend-chart** menu choice to create a trend chart, which plots the current value of an attribute over time.
- Use the **plots, value axes, and trend chart format subtable** menu choices of the trend chart to edit the plot, the axes, and the background color.

Sending a Message to the Operator

In this lesson, you will learn how to:

- Send a message on the Message Board to the operator.
- Create a method that sends a message for both types of offices.
- Cause G2 to wait for a specified period of time before executing the next statement.

Informing the Operator When an Office is Over Budget

An important part of an end user interface is communicating with the operator by sending messages. G2 supports numerous types of messaging:

- Sending a message to the **Message Board**, which is a special workspace on which G2 displays messages that you send by using the `post` action.
- Sending a message to a workspace by using the `inform on` action.
- Creating subclasses of the built-in `message` class, which you can dynamically create and display.
- Creating communication objects, using the G2 Foundation Resources (GFR) module, which determine how different user modes handle communications.
- Creating operator messages, using the G2 Event Management (GEVM) module, which allows you to create operator messages and Message Browsers for displaying and interacting with a variety of types of operator messages.

In general, you use the `inform` action and subclasses of `message` only during the development phase of your application for testing purposes. In a real application, you use GFR or GEVM to create communications objects, which present different types of messages to the user, depending on the user mode. For example, in operator mode, you might present a message in a dialog, whereas in developer mode, you might present a different type of message, which is more meaningful to a developer.

You can also use GEVM to create an internal blackboard of event states, about which you can perform event detection, for example, using the G2 Event and Data Processing (GEVM) module, and/or diagnostic reasoning, using the G2 SymCure module.

In this prototype, you will inform the operator when the office is over budget by sending a message to the Message Board.

To inform the operator when an office is over budget:

- ➔ Edit the rule on the Rules Workspace that checks to see if a site is over budget by adding a **post** statement before the rule deletes the connections.

The **post** statement should inform the operator for the next number of seconds that the office is over budget. You specify the text of the **post** statement as a string with no embedded line feeds. Also, remember to use the reserved word **and** between actions in the rule.

The rule should look like this:

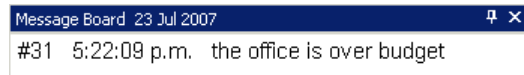
```
for any office O
  if the total-cost of O > the budget of O then in order
    post for the next 5 seconds "the office is over budget" and
    make every connection connected to O transient and
    delete every connection connected to O
```

Now test the rule.

To test the rule:

- 1 Override the budget of one of the offices so you do not have to wait for the total cost to exceed 1000.
- 2 Create and connect several small offices and observe the behavior when the total cost exceeds the budget.

G2 deletes the connections and displays a message on the Message Board that tells the end user that the office is over budget:



- 3 Reset G2.

Informing the Operator About a Specific Office

The text of the message can include any expression, which G2 evaluates when it displays the message. For example, the message can refer to the name of the object or any of its attributes, using the reserved word **the**. To use an expression in the text of a message, you enclose the expression in square brackets (`[]`).

To refer to a specific office by its address:

- ➔ Edit the post statement in the rule to refer to the **address** attribute of the office as an expression.

Hint You refer to an attribute of an object in an expression like this: `[the address of my-office]`.

The rule should look like this:

```

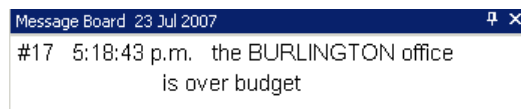
for any office O
  if the total-cost of O > the budget of O then in order
    post for the next 5 seconds "the [the address of O] office
      is over budget" and
    make every connection connected to O transient and
    delete every connection connected to O
  
```

Now test the rule.

To test the rule:

- 1 Create several offices, edit the address of one of the offices in the table, and edit the budget of one of the offices to be 10.
- 2 Connect several offices to the office whose address you specified.

G2 displays a message that refers to the particular office by its address when the office is over budget:



- 3 Reset G2.

Notice that the text string of the message contains no line feeds or tabs; the text wraps to the next line. In a text string, if you enter a line return or a tab as part of the text string, they appear as part of the message.

Creating an Action Button for Testing Purposes

You can create an action button for testing purposes that temporarily changes the budget of every office to 10. To do this, you use the **change the text of action**, which temporarily changes the text of an attribute of an object to a new string. When you reset the KB, the attribute value reverts to its default value.

Note You can conclude a value directly into any system-defined attribute by using the **attribute access** feature of G2. Because this feature involves more complex data structures, you will not learn about this feature here. However, note that using the **change the text of action** to conclude the value of a system-defined attribute is not the recommended technique.

To create an action button that overrides the budget:

- 1 With the Schematic Diagram workspace selected, choose Workspace > New Button > action-button to create a button.
- 2 Edit the label attribute to be "Override Budget".
- 3 Edit the action attribute to use the change the text of action to change the text of the budget attribute of one or more offices.

For example:

change the text of the budget of every office upon this workspace to "100"

Creating a Method that Informs the Operator

As you can see, the rule that checks to see whether an office is over budget is becoming somewhat complicated: it sends a message, makes the connections transient, and deletes the connections.

To take a more object-oriented approach, you could create a method for the office that performs the actions of the rule. You could create a single method or multiple methods, one for each distinct task. The consequent of the rule would then start the method or methods. By creating methods for the actions in the consequent of a rule you:

- Encapsulate information in one place, which helps to organize the knowledge base.
- Make the actions generic so that other objects can use them.

To create a method for the office class that informs the operator:

- 1 Create a new method on the Definitions workspace by choosing Workspace > New Definition > procedure > method.
- 2 Create a method named `send-message` for the office class.
Thus, the first argument to the method is `office`.
- 3 In the body of the method, add an `inform` statement that indicates an office is over budget, using an expression.

Shortcut You can use the same `inform` statement that you specified in the rule by pasting the text of the rule directly into the text of the method while you are editing the method.

- 4 Create a method declaration for the `send-message` method by choosing Workspace > New Definition > procedure > method-declaration.
- 5 Update the rule so that it starts the `send-message` method for any office, rather than informing the operator.

Tip By convention, you should use all upper case letters for embedded procedures or methods within a rule, or within another procedure or method. That way, it is easy to scan the code to identify embedded procedures and methods.

The rule should look like this:

```
for any office O
  if the total-cost of O > budget of O then in order
    start send-message(O) and
    make every connection connected to O transient and
    delete every connection connected to O
```

The method should look similar to this:

```
send message (O: class office)
  begin
    post for the next 5 seconds that "the [the address of O] office
    is over budget
  end
```

Here are the method and method declaration that allow an office to send a message:



office::send-message

SEND-MESSAGE

Now test the rule.

To test the rule:

- 1 Create several small offices, specify the address and budget for one of the offices, and connect several offices to that office.

The results should be identical to the previous test; G2 now invokes the method rather than informing the operator in the rule.

- 2 Test the rule for a large office to ensure that the method works for both subclasses of office.
- 3 Reset G2.

Note To further encapsulate the knowledge, you could also create a method named `delete-connections`, which performs the last two actions of the rule.

Adding a Wait Statement to a Method

You might have noticed that the message G2 displays when an office is over budget is slightly delayed. This is because G2 invokes the actions in the rule as fast as possible in the order specified. Thus, first it starts the method and then it makes the connections transient and deletes them. However, starting a method does not guarantee that G2 will execute the actions of the method before it executes the other actions in the rule; starting simply begins executing the method.

To control the timing of execution of statements within methods and procedures, you use the `wait` action, which causes G2 to wait some number of seconds before executing the next statement. Because methods and procedures guarantee that each statement is executed in order, you can control the exact timing of any action.

Caution In a real application, you should avoid using `wait` statements as much as possible, because the state of the KB can change radically after a wait state. To ensure consistency, you must validate the state of the KB after each wait state, which is very inefficient.

To edit the method to add a wait statement:

- 1 Edit the `send-message` method to cause G2 to wait for two seconds after it sends the message.
- 2 Create several offices, specify the address and budget for one of the offices, and connect several offices to that office.

Does G2 wait before deleting the connection? It does not appear to.

- 3 Change the wait statement to wait for 10 seconds as opposed to 2 seconds.

G2 still does not wait. Let's explore why not.

The method should look like this:

```
send-message (O: class office)
  begin
    post for the next 5 seconds that
      "the [the address of O] office is over budget;
    wait for 2 seconds
  end
```

Creating a Procedure that Starts a Method

When G2 executes the actions in the consequent of a rule, it executes them as fast as possible in the order specified. However, the `start` action begins executing the method and does not wait for the method to complete. To control more precisely

the order in which G2 takes the actions in the consequent of a rule, you can create a procedure that executes all the actions in the consequent of the rule. The consequent of the rule then starts the procedure.

Here are the actions that the procedure will execute as they appear in the consequent of the rule:

```
start send-message (O) and
make every connection connected to O transient and
delete every connection connected to O
```

Several key differences exist between the syntax of rules and procedures:

- In a rule, you separate statements with an **and**, whereas in a procedure, you separate statements with a semi-colon.
- In a rule, you use the word **every** to refer to every instance of a class, whereas in a procedure, you create a local name for use with a **for each** statement and a **do** loop to iterate over each instance of a class.

For example, to rewrite the two **every** statements in the rule, the procedure syntax would look like this:

```
for C = each connection connected to O
do
    make C transient;
    delete C;
end
```

In this statement, **C** is a local name, which you declare in the local name declarations of the procedure, and **O** is the argument to the procedure.

Now you will create a procedure that starts the **send-message** method and deletes all the connections. When a procedure or method starts another procedure or method as one of its actions, the procedure or method that is being started is called an **embedded procedure** or **method**.

To create a procedure that executes a sequence of actions:

- 1** With the Definitions workspace selected, choose **Workspace > New Definition > procedure > procedure** to create a new procedure.
- 2** Name the procedure **take-over-budget-actions**.
- 3** Declare a local name **C**, which is a type of the connection class.
This local name will represent the connections to delete.
- 4** In the body of the procedure, start the **send-message** method for the office that is the argument to the procedure.

Thus, the procedure has an embedded method.

- 5 After sending the message to the operator, delete each connection connected to the office by first making the connection transient and then deleting the connection.

Use the **for each** syntax shown above for the statement.

The procedure should look like this:

```
take-over-budget-actions (O: class office)
C: class connection;
begin
  start send-message(O);
  for C = each connection connected to O do
    make C transient;
    delete C
  end
end
```

Updating the Rule to Start the Procedure

Now you will update the rule that checks for over budget status to start the `take-over-budget-actions`.

To start a procedure in the consequent of a rule:

- ➔ Edit the consequent of the rule that tests whether an office is over budget to delete the actions and start the `take-over-budget-actions` procedure.

You use the `start` action to start the procedure with the office as its argument. Remember, you do not need the `in order` statement in the rule.

Shortcut You can paste the name of the procedure directly into the rule by dragging your cursor over the name of the procedure while editing the rule.

Notice that the procedure name appears in the rule in upper case.

The rule now looks like this:

```
for any office O
  if the total-cost of O > the budget of O then
    start take-over-budget-actions(O)
```

Now you can test the rule to ensure that the application still works.

To test the rule:

- 1 Create several small offices, specify the address and budget for one of the offices, and connect several offices to that office.
- 2 Reset G2.

The results should be identical to the previous test; G2 now executes the procedure rather than executing the actions in the consequent of the rule.

However, you will notice that G2 *still* does not wait after it sends the message before it deletes the connections. Recall, we added a `wait` statement to the `send-message` method. Finally, you will edit the procedure to cause G2 to wait.

Starting a Method

Let's look at how G2 is executing the actions in the rule, the procedure, and the method. This diagram shows the rule that checks for over budget status; the procedure that starts the method and deletes the connections; and the method that sends the message and waits for 2 seconds:

<pre>for any office O if the total-cost of O > the budget of O then start take-over-budget-actions(O)</pre>	<p>(1) G2 invokes this rule by forward chaining when the value of total-cost changes. When total-cost exceeds budget, the rule starts the procedure.</p>
<hr/> <pre>take-over-budget-actions (O: class office) C: class connection; begin start send-message(O); for C = each connection connected to O do make C transient; delete C end end</pre>	<p>(2) The procedure starts the <code>send-message</code> method and immediately executes the next statement, which loops over all the connections, makes them transient, and deletes them.</p>
<hr/> <pre>send message (O: class office) begin post for the next 5 seconds "the [the address of O] office is over budget"; wait for 2 seconds end</pre>	<p>(3) The method sends a message to the operator and waits for two seconds.</p>

G2 executes statements in procedures and methods in order. However, the `start` action simply starts the method. Once the method is started, G2 immediately executes the next procedure statement; it does not wait for the method to complete before it executes the next statement. Thus, in the example above, the `wait` statement in the method does not necessarily execute before G2 deletes the connections.

Calling a Method

You can control whether or not G2 finishes executing all the actions in an embedded procedure or method before it begins executing the next statement by using the **call** action, as opposed to the **start** action.

One difference between the **start** action and the **call** statement is:

- The **start** action executes the embedded procedure or method and immediately returns control to the calling procedure or method.
- The **call** statement executes the embedded procedure or method and only returns control to the calling procedure or method when the entire embedded procedure or method has finished executing.

Another difference between the **start** action and the **call** statement is:

*You can only use the **call** statement in a procedure, not a rule.*

This is why you could not originally use the **call** action to start the method directly from the rule. Instead, you needed to create a procedure to invoke the method, using the **call** statement, and start the procedure from the rule.

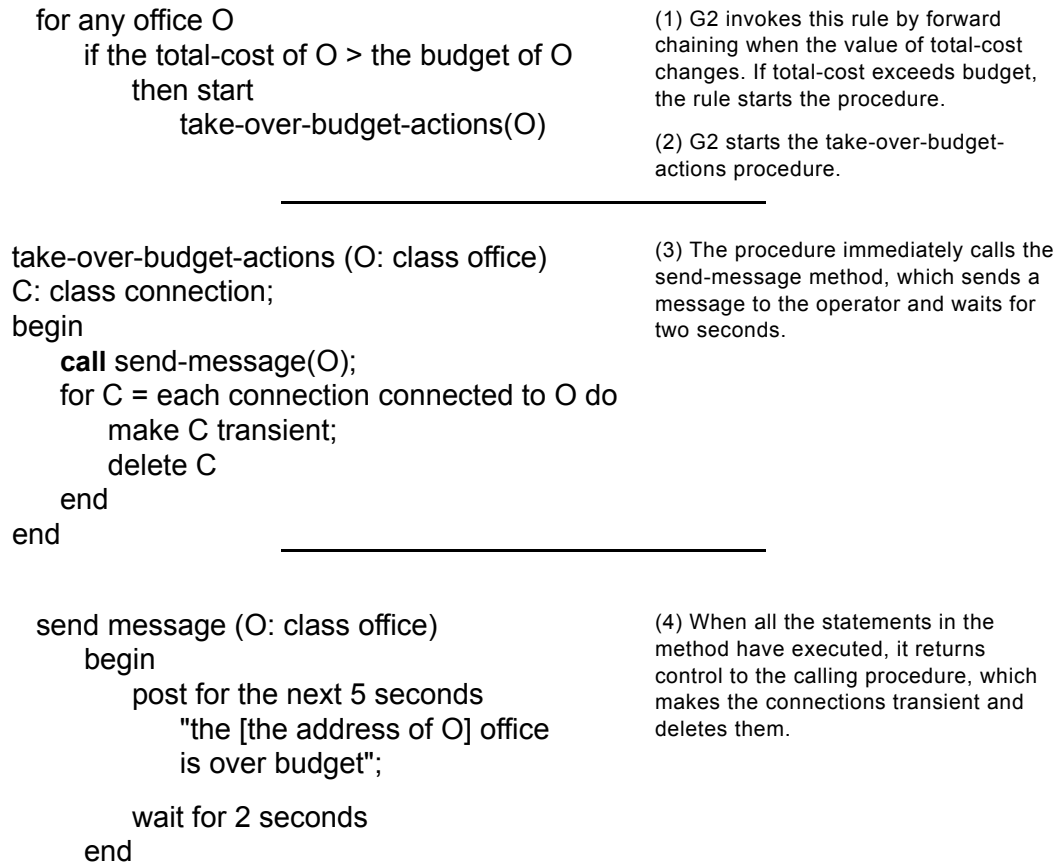
Now you will edit the procedure that calls the **send-message** method to use the **call** statement instead of the **start** action.

To start a method by using the call action:

- 1** Edit the **take-over-budget-actions** procedure to start the **send-message** method by using the **call** statement.
- 2** Create several small offices, specify the address and budget for one of the offices, and connect several offices to that office.
- 3** Reset G2.

G2 detects when the office is over budget, sends a message to the Message Board, waits for two seconds, and then deletes the connections.

This diagram shows how the rule, procedure, and method interact now:



To save the application:

- ➔ Save the KB to the file named *ch5.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created a method that informs the operator when an office is over budget and waits before deleting the connections. You also created a procedure that calls the method and deletes the connections.

You learned how to:

- Use the **post** action to send a message to the operator.
- Use an expression within the text of a **post** statement that refers to an attribute of an object.
- Encapsulate an inform statement in a method and start the method in a rule.

- Use the **wait** action to control the timing of sequential statements within a method.
- Encapsulate knowledge and control the order of execution and the timing of events by creating a procedure that executes the actions in the consequent of a rule.
- Use the **for each** statement with a **do** loop within a procedure to refer generically to each instance of a class by looping.
- Use the **call** statement to execute an embedded method from within a procedure, which causes G2 to finish executing all the statements in the embedded method before passing control back to the procedure.

Animating Objects

In this lesson, you will learn how to:

- Create a method that animates an object by looping.
- Create a rule that starts the animation method under certain conditions.
- Use subsecond timing to make the animation more realistic.

Creating a Method that Animates an Office

Another way to communicate with the end user is to animate objects. In G2, you **animate** an object by creating a method that repeatedly changes the color of its icon or moves the icon, waits a split second, then changes the color or moves the object again. The effect is an object whose icon animates.

For example, you might want to animate an office when the total cost is approaching the budget, thereby providing a visual indicator to the end user that an office is about to be over budget.

To create a method that animates each type of office:

- 1 On the Definitions workspace, create a method called **animate-icon** for the **small-office** class.
- 2 Specify the body of the method to perform these actions:
 - Change the **status icon-color** to red.
 - Wait 1 second.
 - Change the **status icon-color** to its original color.

- 3 Create a method declaration for the `animate-icon` method.

The method for the small office should look similar to this:

```
animate-icon(O: class small-office)
begin
  change the status icon-color of O to red;
  wait for 1 second;
  change the status icon-color of O to salmon
end
```

Now test the method.

To test the method:

- 1 On the Definitions workspace, create an action button that starts the method for the small office master.

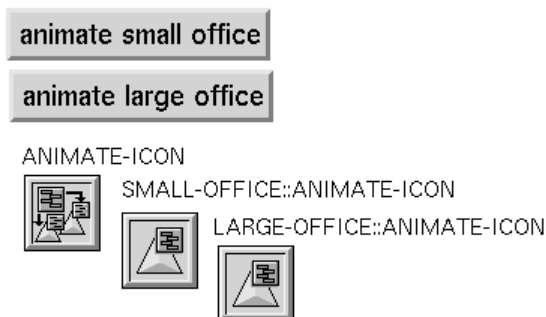
G2 turns the status icon region red for a second, then changes it back to its original color.

- 2 Clone the `small-office::animate-icon` method to create a method for the large office class that changes the color of the status icon region to red and then back to its original color.

- 3 Clone the action button that tests the animation of the small office master to create an action button that tests the animation of the large office master.

G2 turns the status icon region red for a second, then changes it back to its original color.

Here are the two methods, the method declaration, and the two action buttons for testing the methods:



The expression attribute for the animate small office action button looks like this:

```
start animate-icon(small-office-master)
```

Creating a Loop in a Method

To create real animation, you need to repeat the statements in the method so that the colors appear to flash. To do this, you use the **repeat** statement to execute the statements in a loop. A repeat statement starts with the word **repeat** and ends with the word **end**.

Typically, you add an **exit if** statement after the last statement in the repeat loop to cause the looping to stop under certain conditions. For example, when you animate the office icon, you want the animation to stop when the total cost of the office exceeds the budget and the connections are broken.

To add a repeat statement to the animation methods:

- 1 Edit the `small-office::animate-icon` method to add a **repeat** statement just after the **begin** statement.
- 2 Add another **wait** statement after the last **change** statement so that G2 waits before changing the icon color back to red.
- 3 Add an **exit if** statement after the last **change** statement that causes G2 to exit the loop if the total cost of the office is greater than the budget.
- 4 Add an **end** statement after the last **exit if** statement to end the loop.
- 5 Edit the `large-office::animate-icon` method to add the same set of statements.

The method for the small office should look like this:

```
animate-icon(O: class small-office)
begin
  repeat
    change the status icon-color of O to red;
    wait for 1 second;
    change the status icon-color of O to salmon;
    wait for 1 second;
    exit if the total-cost of O > the budget of O
  end
end
```

Now you can test the methods.

To test the methods:

- 1 Click each action button.
G2 animates the offices indefinitely.
- 2 Restart G2 to stop the methods from executing.

Animating the Office When it is Almost Over Budget

Now that you have created a method that animates each type of office, you can create a rule that animates the office when the total cost is approaching the budget. For example, you might start the animation method when the total cost is 50 dollars less than the budget.

To create a rule that animates the office when total cost is close to the budget:

- 1 On the Rules Workspace, create a new rule that starts the `animate-icon` method for any office when the total cost of the office is 50 dollars less than the budget.

Hint Make the rule generic. The antecedent of the rule tests to see if the total cost is greater than the budget minus 50. The consequent of the rule starts the animation method for a particular office. The rule is invoked by forward chaining when the total cost receives a value.

- 2 Use free text to label the rule to indicate how it is invoked.
- 3 Test the rule.

The rule is invoked by forward chaining each time the total cost of an office changes. When the total cost of any office is 50 dollars less than the budget, the office begins to animate. When the total cost exceeds the budget and the connections are broken, animation stops.

The rule should look like this:

```
for any office O
  if the total-cost of O > (the budget of O - 50) then start animate-icon(O)
```

Using Subsecond Timing for Animation

To make the animation more realistic, you can use **subsecond timing** to control the timing of `wait` statements in methods and procedures, down to a fraction of a second.

By default, G2 schedules its actions at one second intervals. Thus, to use subsecond timing, you must specify:

- A subsecond wait period in a method or procedure.
- A subsecond minimum scheduling interval in the Timing Parameters system table.

G2 defines numerous **system tables** that allow you to control the default behavior of various aspects of the G2 environment, such as timing parameters, fonts, log files, and the text editor.

To use subsecond timing to animate an icon:

- 1 Edit the `animate-icon` method for each subclass to specify .1 seconds in each `wait` statement.
- 2 Choose Edit > System Tables > Timing Parameters to display the system table that controls the default timing parameters in G2.

This table allows you to control the default behavior of various timing parameters, including the minimum scheduling interval. Notice that the minimum scheduling interval is 1 second, by default.

- 3 Edit the `minimum-scheduling-interval` attribute to be .1 seconds.
- 4 Test the animation by clicking the animation buttons on the Definitions workspace.

G2 now animates the icons at a much faster interval when the offices are about to be over budget.

To save the application:

- ➔ Save the KB to the file named `ch5.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you created a method that animates the office icon, using subsecond timing, when the total cost is approaching the budget.

You learned how to use:

- A `repeat` statement in conjunction with a `wait` statement to cause an object to animate by looping.
- An `exit if` statement to determine when a repeat loop finishes executing.
- The Timing Parameters system table to specify the `minimum-scheduling-interval` attribute to be a number less than one, to use subsecond timing.

Making Workspaces Attractive and Informative

In this lesson, you will learn how to:

- Change the background color of a workspace.
- Add frames to a workspace.
- Add background graphics to a workspace.

Changing the Color of a Workspace

You can change the background color of any workspace to:

- Make end user workspaces more attractive.
- Identify different types of workspaces by their color.

For example, you might want your end user workspaces to be one color, your workspaces that act as menus and submenus to be another color, and your workspaces that contain rules and definitions to be a different color.

Workspaces have two special colors: background and foreground. The **background color** of a workspace is the color of the workspace itself. The **foreground color** of a workspace includes all objects whose color is not otherwise specified, for example, free text, button text, and connections.

To change the background color of a workspaces:

- 1 With the Schematic Diagram workspace selected, choose Workspace > Color > background-color to display a palette of colors.
- 2 Click the More button to display the full color palette.
- 3 Change the background color to another color.

Tip You typically choose neutral pastel colors for the background of end user workspaces.

- 4 Change the background color of the Definitions and Rules Workspace workspaces to be another color.

Creating a Workspace Frame



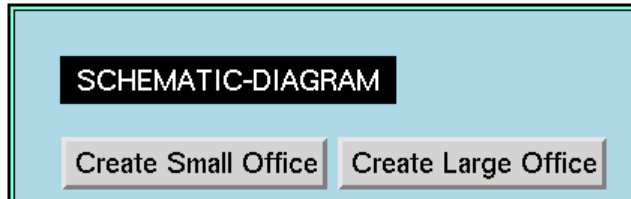
You can create borders around a workspace to make it more attractive. A workspace border is called a **frame style**. A frame style consists of any number of borders of different colors and pixel widths.

To create a frame style for the Schematic Diagram workspace:

- 1 Select the Definitions workspace and choose Workspace > New Definition > frame-style-definition to create a new frame style.
- 2 Display the table for the frame style and specify its name to be green-frame.

- 3 Edit the `description-of-frame` attribute to specify three borders as follows:
border 2 black, 2 aquamarine, 1 black
- 4 Select the Schematic Diagram workspace and choose Workspace > Table to show its table.
- 5 Edit the `frame-style` attribute to refer to the green-frame.

The workspace now uses the green frame style as its border:



Creating Workspace Subclasses

Suppose you wanted to use the same color and frame style for different types of workspaces in an application. For example, you might use one color scheme for rules and definitions, and another color scheme for the end user workspaces. This technique makes it easy to tell the type of workspace you are viewing.

You can create a class definition that is a subclass of the `kb-workspace` class, which specifies the default color and frame style for the workspace.

Each time you create a new workspace, you would create a workspace of the appropriate subclass so that all your workspaces look consistent.

To create a definition workspace subclass with a default color:

- 1 On the Definitions workspace, create a class definition whose direct superior class is the `kb-workspace` class.

For example, you might create a workspace subclass named `definition-wksp` on which you place class definitions, methods and procedures, user menu choice definitions, and so on.

- 2 Using the `attribute-initializations` attribute, specify the `background-color` system-defined initializable attribute.

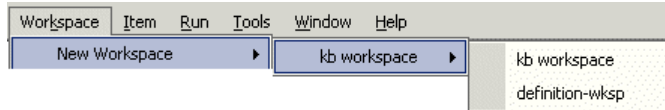
For example:

```
background-color: antique-white
```

Now you can make another workspace for definitions relating to graphics, using the user-defined workspace class.

To create a definition workspace:

- 1 Choose Workspace > New Workspace > kb workspace to display a menu for choosing the workspace class:



- 2 Choose definition-wksp to create an instance of the user-defined workspace class.
- 3 Name the workspace graphics-definition.
- 4 Rename the existing definitions workspace class-definitions.
- 5 Transfer the frame style definition named green-frame from the Definitions workspace to the Graphics Definitions workspace.

Creating Background Graphics for a Workspace

You can add a bitmap image as the background of any workspace. You define the background image by creating an **image definition**, which specifies the name of the image and a filename of JPEG, GIF, or XMB image.

You can use background graphics to make a workspace more interesting or to provide information to the end user. For example, you might use an image of a map to indicate the approximate location of video conferencing offices around the world.

To add a map to the Schematic Diagram workspace:

- 1 Select the Graphics Definitions workspace and choose Workspace > New Definition > image-definition to create a new background graphic image definition.
- 2 Display the table for the image definition and edit names to be world.
- 3 Edit the file-name-of-image attribute to specify a string that specifies the complete pathname of the *world.gif* file.

This file is located in the kbs subdirectory of your G2 directory.

For example, on Windows, the pathname would be:

"c:\Program Files\gensym\g2-2011\g2\kbs\demos\world.gif"

- 4 Display the table for the Schematic Diagram workspace.

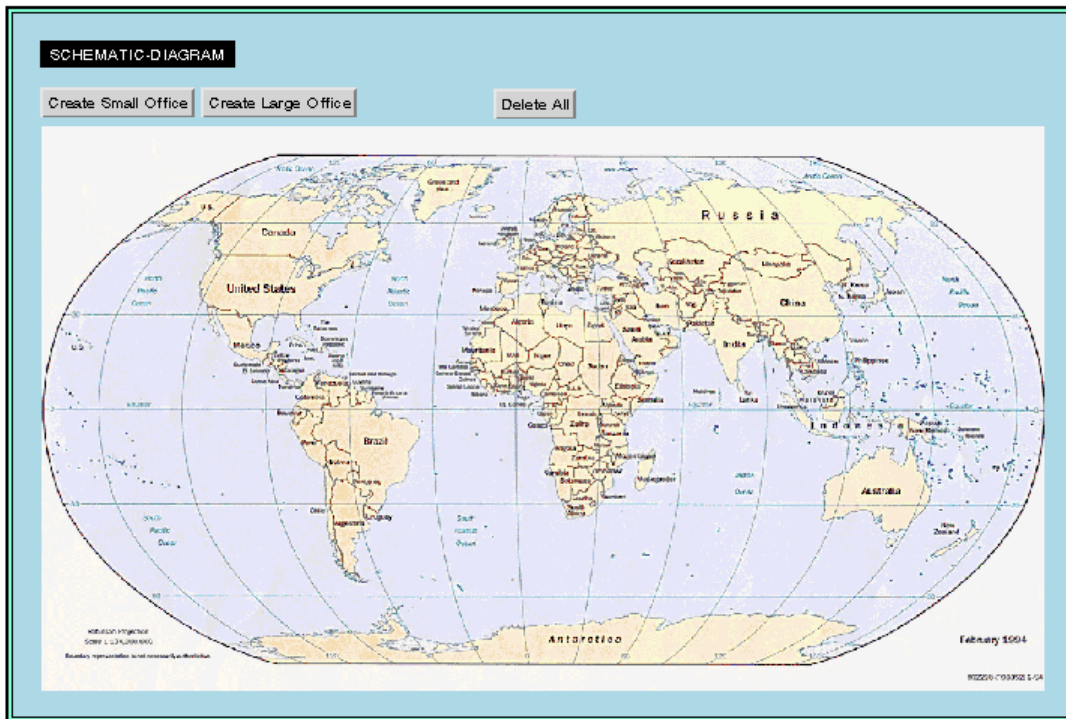
- 5 Edit the background-images attribute to refer to the name of the image definition and its x,y location on the workspace, as follows:

world at (0,0)

This specification locates the image in the center of the workspace.

- 6 Move the title and buttons on the Schematic Diagram workspace so they are aligned with the top of the image.

The Schematic Diagram workspace now allows the end user to place offices at various locations around the map:



To save the application:

- ➔ Save the KB to the file named *ch5.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you made the application more visually attractive and informative by adding color, borders, and background graphics to the workspaces.

You learned how to:

- Use the **Color** menu choice for a workspace to change its background and foreground colors.
- Use the **Workspace > New Definition > frame-style-definition** menu choice and the **frame-style** attribute of a workspace to create a border around a workspace.
- Use the **Workspace > New Definition > image-definition** menu choice and the **background-images** attribute of a workspace to add background graphics to a workspace.
- Create a subclass of **kb-workspace** that defines a default background color.

Showing Workspaces Programmatically

In this lesson, you will learn how to:

- Use a rule to display a workspace upon startup.
- Display a workspace at a particular location on a screen at a particular scale.

Using an Initially Rule to Show a Workspace on Startup

In an end user environment, you typically want to show the end user workspace automatically upon startup.

One way to perform actions automatically upon startup is to use an **initially** rule. An **initially** rule indicates all of the actions that G2 is to perform when the user starts the application running.

In a real application, you use the **G2 Foundation Resources (GFR)** module to create a startup object that determines the startup behavior of every module in the KB and the order in which each module starts.

You can show a workspace programmatically at any location on the window and at any scale by using the **show** action. To specify the location of any object on a workspace, you provide the **x** and **y** coordinates. You refer to the center of the window by using (0,0).

To create an initially rule that shows the Schematic Diagram workspace:

- 1 On the Rules Workspace, create a new rule that initially displays the workspace named `schematic-diagram` at the center of the screen at three-quarter scale.

Hint Use the syntax-guided text editor to help you with the syntax of the `show` action. You can show a named workspace at an `x,y` location in the screen, and you can show a named workspace at three-quarter scale.

Label the rule to indicate how it is invoked and place it at the top of the workspace.

- 2 Hide all the workspaces of the application.
- 3 Restart the knowledge base to test the rule.

G2 displays the end user workspace in the middle of the screen at three-quarter scale.

The initially rule should look like this:

```
initially
  show schematic-diagram at (0,0) in the screen and
  show schematic-diagram at three-quarter scale
```

Creating a Button that Iconifies a Workspace

You can use the `show` action to create an action button that scales a workspace and “iconifies” a workspace by showing it at a very small scale.

To create an action button that iconifies the Schematic Diagram workspace:

- 1 Create a new action button on the Schematic Diagram workspace that shows the current workspace at 10% of its current size:



Hint You can use the `show` action to show a workspace scaled by its current scale times a number.

- 2 Test the action button.

G2 shrinks the workspace to 10% of its size:



- 3 Restart the application to display the workspace at three-quarter size again.

The action of the button should look like this:

```
show this workspace scaled by its current scale times .1
```

To save the application:

- ➔ Save the KB to the file named *ch5.kb*, with your initials appended to the end of the filename.

Summary

In this lesson, you created a rule that shows the end user workspace when you start the application, and you created a button that “iconifies” the workspace.

You learned how to use:

- An *initially* rule to show a workspace upon startup.
- The *show* action to show a workspace at a particular location on the window and at a particular scale.
- An action button to show a workspace at a particular scale.

Configuring the User Interface

In this lesson, you will learn how to:

- Create different user modes for different classes of users.
- Configure the behavior of objects depending on the user mode.
- Create action buttons to switch the user mode.
- Use an *initially* rule to start the application in a particular mode.

What are User Modes?

The last step in building an end user interface for an application is to configure the behavior of the knowledge base for the various classes of users. For example, the video conferencing application might support three classes of users:

- Administrators, who are the developers of the application.
- Developers, who are the users that configure the layout of the video conferencing schematic and specify the attributes of the offices.
- Users, who are the end users that monitor the application and display information about each office while the application it is running.

Thus, you might think of creating three modes in which the application operates:

- **Administrator mode**, where users can perform any G2 operation.
- **Developer mode**, where users can configure the layout of the schematic diagram, display the subworkspace of an office, and edit its table.
- **Operator mode**, where users can just display the subworkspace of an office.

These **user modes** determine the actions the user can perform on various objects, the visible attributes of an object, and the behavior of the overall application.

Typically, you only allow the administrator to change the user mode. You switch between the various modes by using the Tools > Change Mode menu choice.

The only built-in mode in G2 is administrator mode. You create the other modes according to the needs of the application.

What are User Interface Configurations?

You can configure the behavior of any class of objects or the overall KB in different user modes by specifying **user interface configurations**. You specify the three types of user interface configuration statements in different locations to configure different aspects of the KB, as this table describes:

This type of configuration statement...	Configures the behavior of...	And you specify it in...
Instance configuration	Each instance of a class	The class definition
Item configuration	Specific G2 items	The item
KB configuration	The overall KB	The KB Configurations system table

You can configure various aspects of class instances, specific objects, or the overall knowledge base, including:

- The visible attributes.
- The available menu choices.
- The available “non-menu choices,” such as moving an object or displaying its menu.
- The behavior of mouse clicks and keyboard commands.
- The behavior of an object when you select it.
- Whether objects snap to a grid on a workspace.

For example, in developer mode, you might want the user to be able to display the table for an office but not to transfer the office to another workspace. Further, in developer mode, only the user configurable attributes and attributes shown in end user displays need to be visible in the table. Thus, you might want to restrict the “internal” attributes such as `connection-cost-per-minute` and `total-cost-per-minute`.

In operator mode, you might not want the user to be able to move an office on the workspace. Furthermore, you might not want the user to be able to display the menu for an office at all. Instead, you might want the end user to be able to click on an office to display its subworkspace.

In administrator mode, nothing would be restricted.

Always define user interface configurations so that they restrict functionality incrementally with each user mode and always define one more that has no restrictions.

Configuring the Office for Developer Mode

In developer mode, you want the user to be able to do the following and only the following:

- Display the table for an office.
- Show the subworkspace of an office.
- Delete an office.
- Create a connection for an office.

To do this, you specify an instance configuration for the office class.

Configuring the Menu Choices of Every Instance

First, you will configure the available menu choices.

To configure the available menu choices for all offices in developer mode:

- 1 On the Definitions workspace, display the table for the office class definition.

Notice that the table has an item-configuration and an instance-configuration attribute:

- You use instance-configuration to specify the behavior of all instances of the class.
- You use item-configuration to specify the behavior of the class definition object itself.

- 2 Use the syntax-guided text editor to edit the instance-configurations attribute as follows:

configure the user interface as follows:

when in developer mode:

menu choices for office include: table, delete, create-connection,
go-to-subworkspace

Now you will change the mode to see the effect.

To change the mode to developer mode:

- ➔ To change the mode, choose Tools > Change Mode, edit the g2-user-mode attribute to be developer, and click End.

Shortcut You can also use the Ctrl + y command to display the login panel on which you can change the user mode.

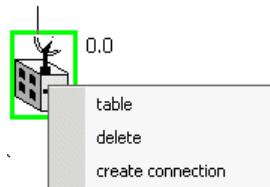
You define the available user modes by specifying a configuration statement, as you just did.

Now you can test the application in developer mode.

To test the application in developer mode:

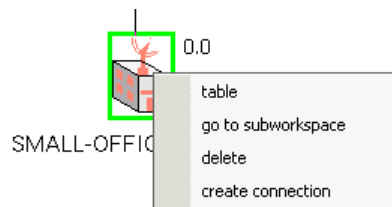
- 1 Display the menu for the office instance on the Definitions workspace.

The menu includes three of the menu choices mentioned in the configuration statement: table, delete, and create-connection:



- 2 Display the table for the small-office-master object.

The menu includes the three menu choices you specified in the instance configuration, as well as the go to subworkspace menu choice, because the subclass defines a subworkspace.

**Configuring the Attributes of Every Instance**

Now you will limit the attributes that are visible in the table.

To configure the visible attributes in the table for all offices in developer mode:

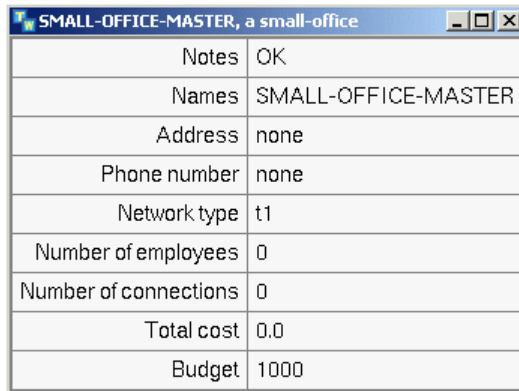
- 1 Update the instance configuration statement for the office class to exclude the two internal attributes, as follows:

attributes visible for office exclude: connection-cost-per-minute,
total-cost-per-minute

Hint Configuration statements are separated by a semi-colon.

- 2 Display the table for the small office master.

The table excludes the attributes you specified in the configuration statement:



Attribute	Value
Notes	OK
Names	SMALL-OFFICE-MASTER
Address	none
Phone number	none
Network type	t1
Number of employees	0
Number of connections	0
Total cost	0.0
Budget	1000

Configuring the Office for Operator Mode

In operator mode, you want the user to be able to do the following and only the following:

Display the subworkspace of the office by clicking the object

To do this, you update the instance configuration for the office class to specify the behavior when you are in operator mode.

Configuring the Menu Choices of Every Instance

First, you will configure the menu choices.

To configure the menu choices for all offices in operator mode:

- 1 Edit the instance-configuration attribute for the office class to configure the selection behavior in operator mode, as follows:

when in operator mode:

selecting any office implies go-to-subworkspace

The other options for the selection behavior of an object include naming the object, cloning the object, and showing the table for the object.

- 2 Switch to operator mode.

Hint Remember, you can use Tools > Change Mode or Ctrl + y.

- 3 Click on the `small-office-master` object on the Definitions workspace.

G2 automatically displays the subworkspace of the office.

Configuring the “Non-Menu” Choices of Every Instance

Now you will further restrict the user interface to disallow any “non-menu choices,” such as moving the object.

To configure the non-menu choices for all offices in operator mode:

- 1 Edit the instance-configuration attribute for the office class to restrict non-menu choices to allow nothing, as follows:

```
non-menu choices for office include: nothing
```

This configuration statement means that you cannot perform any “non-menu” operation on the office, such as displaying its menu or moving the object.

- 2 Try moving the small office master.

G2 disallows moving the office in operator mode.

Starting the Application in Operator Mode

In addition to displaying the Schematic Diagram workspace upon startup, you might want the application to start in operator mode. The G2 server and each connected Telewindows client has an associated `g2-window` object, which keeps track of various information, including the G2 user mode of the window. You can refer to the current window by using the expression `this window`, or you can refer to every window by using `every g2-window`.

To start the application in operator mode:

- 1 Edit the initially rule on the Rules Workspace to change the text of the `g2-user-mode` attribute of every `g2-window` to "operator".
- 2 Restart the KB and verify that you are in operator mode.

You should not be able to move any offices on the workspace, and, when you click on an office, G2 displays its subworkspace.

The initially rule should look like this:

```
initially
  show schematic-diagram at (0, 0) in the screen and
  show schematic-diagram at three-quarter scale and
  change the text of the g2-user-mode of every g2-window to "operator"
```

Now save the application after hiding all the application workspaces.

To save the application:

- ➔ Hide all the workspaces except the one with the user mode buttons and save the KB to the file named `ch5.kb`, with your initials appended to the end of the filename.

Summary

In this lesson, you configured the user interface for operator mode and developer mode, you created action buttons to change the mode, and you updated the initially rule to change the user mode on startup.

You learned how to use:

- The `instance-configurations` attribute of a class to specify the behavior of each instance of the class in various user modes.
- The `configure the user interface as follows`: statement in an instance configuration to configure the menu choices, attributes visible, selecting, and non-menu choices instance configurations.
- The `change the text of action` to change the user mode of every G2 window upon startup.

Running the Prototype

In this lesson, you will:

- Run the prototype in developer mode.
- Run the prototype in operator mode.
- Save and load the finished prototype.

Configuring the Schematic in Developer Mode

First you will configure the layout of the video conferencing schematic diagram in developer mode and configure the attributes of each office.

To run the prototype in developer mode:

- 1 Start the application running.
- 2 Choose Tools > User Mode > Developer to switch to developer mode.
- 3 Create a number of offices on the Schematic Diagram workspace.
- 4 Configure the `address` attribute of all offices by editing their tables.

You must configure the address of each office because the KB must evaluate this attribute when it is running when an office is over budget.

When G2 evaluates an attribute that has no value (`none`), it generates this runtime error: The attribute `attribute-name` exists in `object`, but the attribute contained nothing, causing this reference to fail.



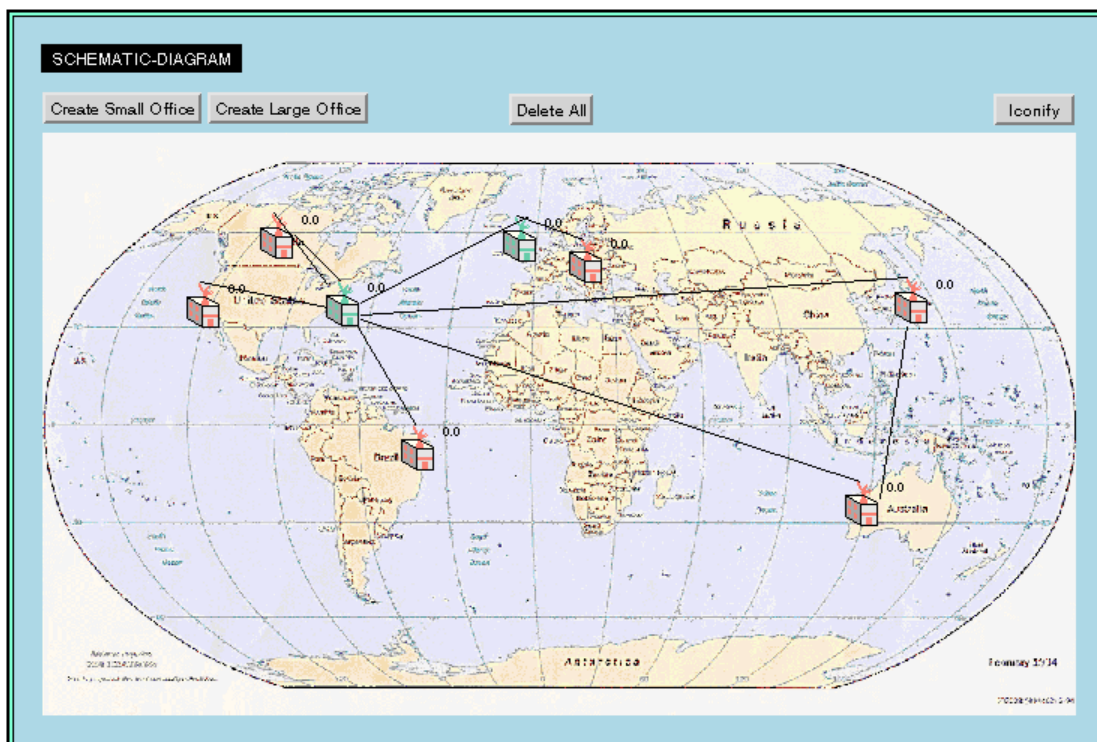
To avoid having to specify a value for each office attribute, you can add default values in the class-specific attribute declaration of the class as shown in the tutorial kbs.

- 5 Create connections as desired by choosing `create connection` and connect the offices together.

If you connect the offices while G2 is running, G2 begins computing the number of connections and the total cost right away.

- 6 Connect the offices together.
- 7 Reset G2 to reset the total cost of each office to zero.

Your schematic diagram might look something like this:



Saving the Prototype

You have now created the end user schematic diagram for a particular video conferencing application. You could create different layouts of the schematic diagram and save them in different knowledge bases to run different versions of the same application.

To save the application:

- ➔ Save the KB to the file named *ch5.kb*, with your initials appended to the end of the filename.

Running the Prototype in Operator Mode

Now you will run the fully configured prototype in operator mode.

To run the prototype in operator mode:

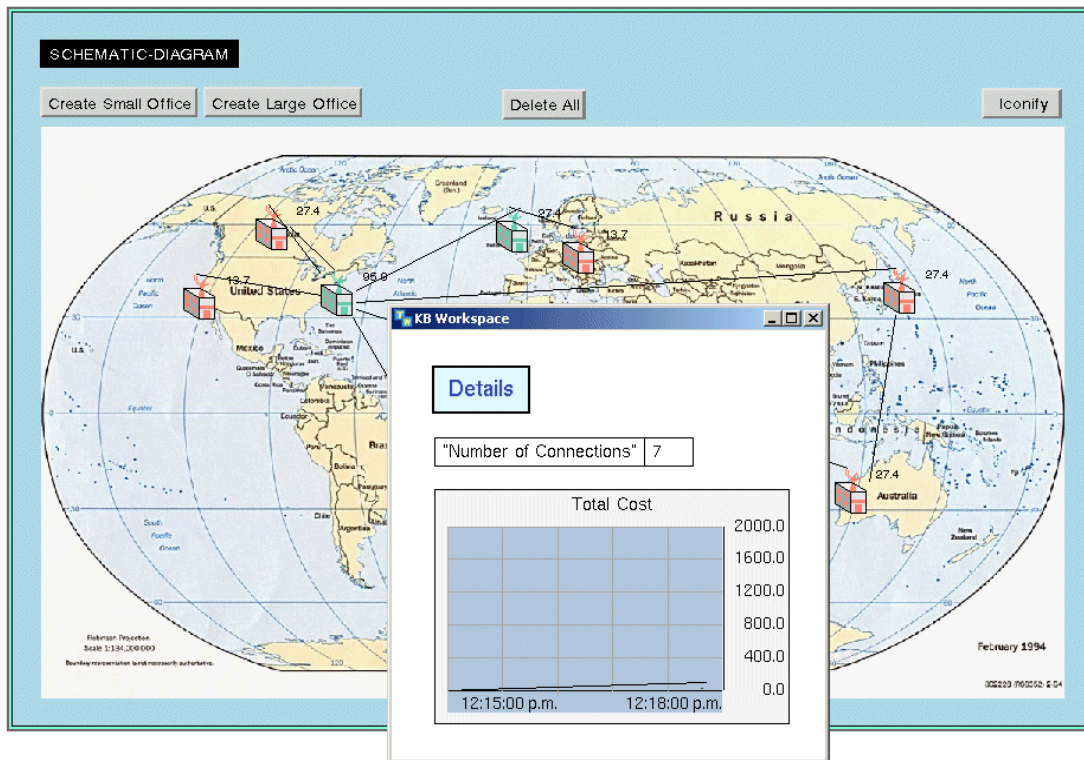
- 1 Start the application.

G2 displays the Schematic Diagram in the center of the window and switches to operator mode. The connected offices compute the number of connections and begin computing total cost.

- 2 Click an office to display its subworkspace.

G2 shows the number of connected offices and plots the total cost over time.

Your application might look something like this:



Simulating an Over Budget Situation

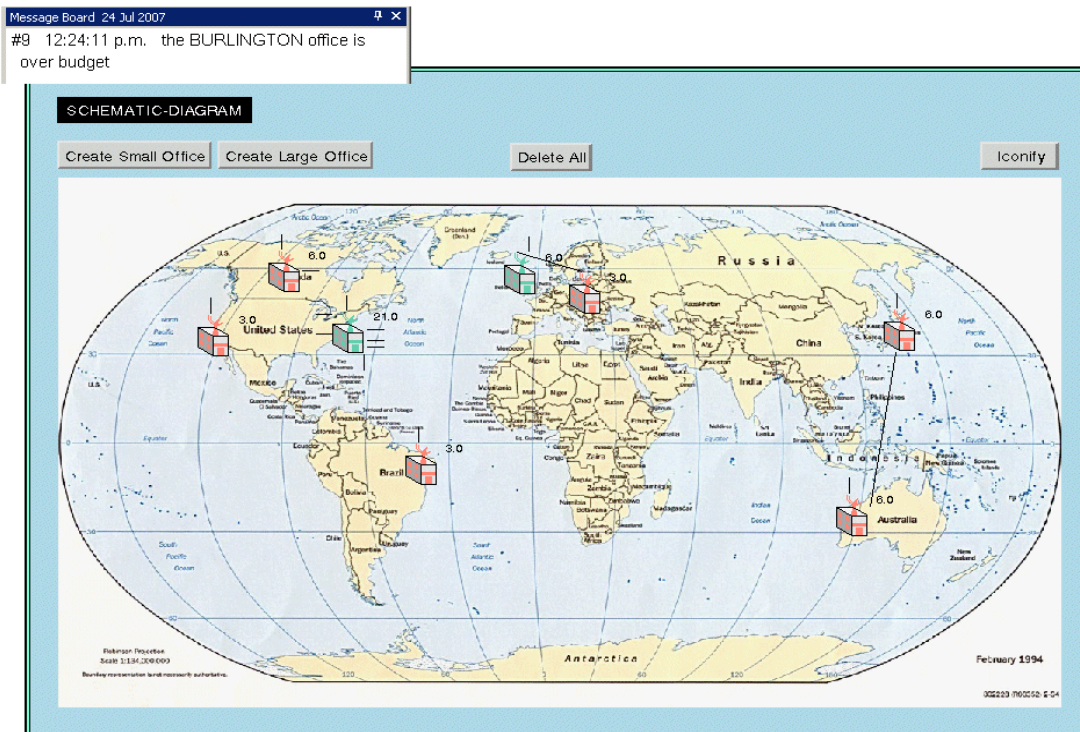
To simulate an over budget situation, you can edit the budget of one of the offices to see the results.

To simulate an over budget situation:

- ➔ Switch to Developer mode and edit the budget of the central office to be a small number such as 20.

G2 animates the office prior to being over budget to warn the operator. G2 then displays a message indicating which office is over budget and deletes the connections. The rest of the application continues to run until eventually all the sites are over budget.

Your application might look something like this:



Loading the Finished Application

You might want to rerun the application from its original state before the connections were deleted. You can do this by loading the finished application from the point at which you saved it.

To load the original application:

- ➔ Load the file named *ch5.kb* with your initials appended to it to restore your completed application or load the file named *solution.kb* to load the solution KB for the entire *Getting Started with G2 Tutorials*.

You can now test the application again using different budget constraints.

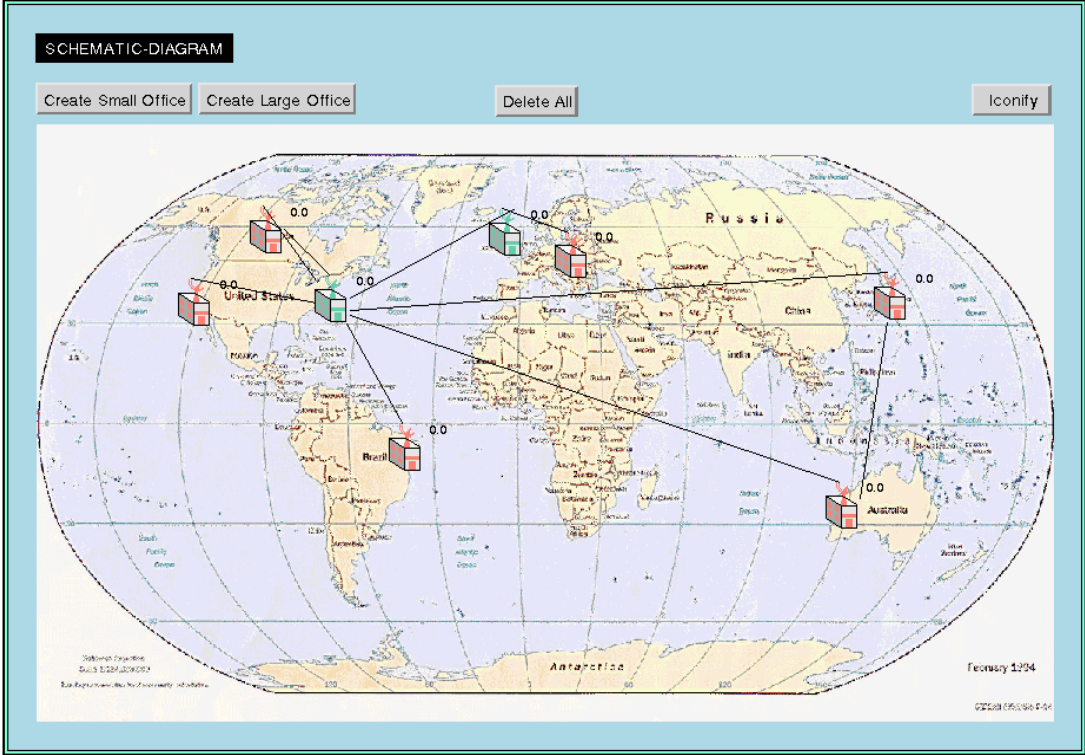
Summary

In this tutorial, you learned how to:

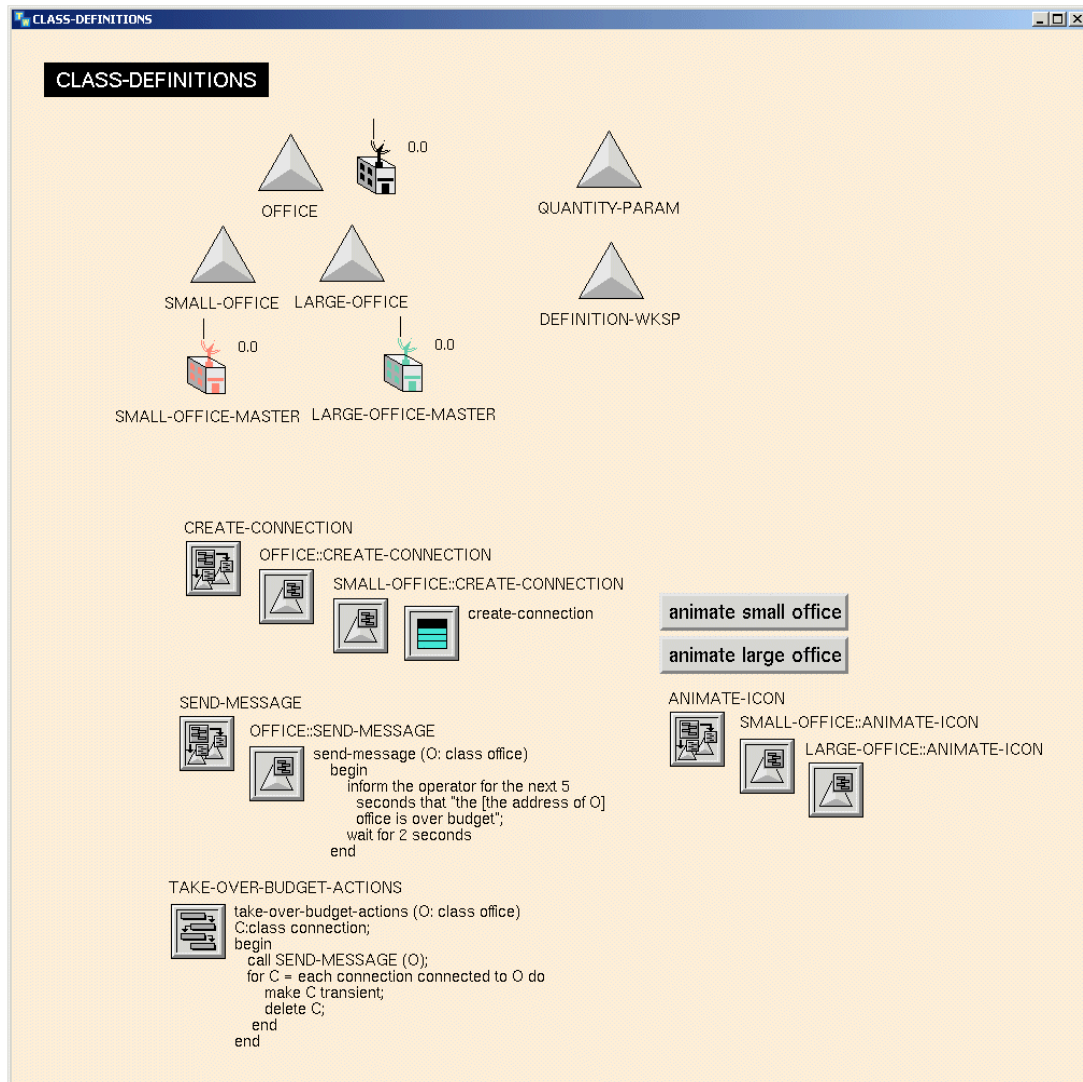
- Create hierarchical views of an application by creating subworkspaces of objects.
- Dynamically create an object with a subworkspace by using the **create by cloning** action to clone a master object.
- Create and configure a trend chart to display the history of an attribute value over time.
- Create a readout table that reports the current value of an attribute.
- Create a method that sends a message to the operator on the Message Board, using the **post** action.
- Create a procedure that calls a method, using the **call** action to more finely control the timing of events.
- Create a method that animates an object, by using the **change** action, the **wait** action, the **repeat** statement, and subsecond timing.
- Make workspaces visually attractive and informative by changing the background color, adding frame styles, and creating image definitions for background graphics.
- Create different user modes for different classes of users, using instance configurations, and configure the behavior of objects based on the user mode.
- Switch the user mode, using several different techniques: the Tools > Change Mode menu choice, the Ctrl + y command, and the **change the text of** action.
- Use an **initially** rule and the **show** action to display a workspace at a particular location in the window upon startup.
- Change the user mode on startup, using an **initially** rule and the **change the text of** action for every **g2-window**.
- Create a button that “iconifies” a workspace, using the **show** action.

Solutions

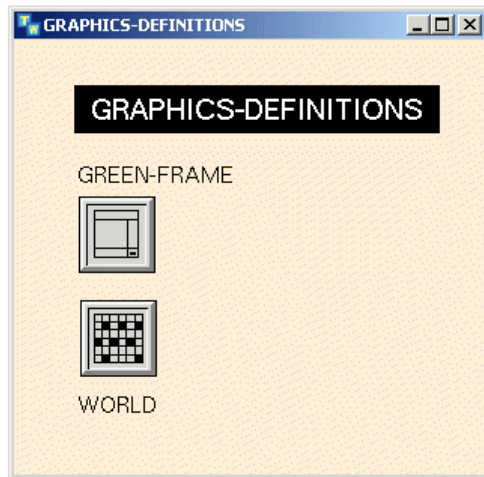
The Schematic Diagram workspace looks like this:



The Class Definitions workspace looks like this:



The Graphics Definitions workspace looks like this:



The Rules Workspace looks like this:

The screenshot shows a window titled "RULES-WORKSPACE" with a list of rules. Each rule is enclosed in a box, and its trigger is listed to the right. The rules are as follows:

Rule Text	Trigger
<p>initially show schematic-diagram at (0,0) in the screen and show schematic-diagram at three-quarter scale and change the text of the g2-user-mode of every g2-window to "operator"</p>	invoked at startup
<p>whenever any connection C is connected to any office O then conclude that the number-of-connections of O = the count of each connection connected to O</p>	invoked whenever a connection is created
<p>whenever any office O1 is disconnected from any office O2 then conclude that the number-of-connections of O1 = the count of each office connected to O1 and conclude that the number-of-connections of O2 = the count of each office connected to O2</p>	invoked whenever a connection is deleted
<p>for any office O whenever the number-of-connections of O receives a value or the connection-cost-per-minute of O receives a value then conclude that the total-cost-per-minute of O = the number-of-connections of O * the connection-cost-per-minute of O</p>	invoked whenever number-of-connections receives a value
<p>for any office O unconditionally conclude that the total-cost of O = the total-cost of O + the total-cost-per-minute of O</p>	Scan interval 2 seconds
<p>for any office O if the total-cost of O > the budget of O then start TAKE-OVER-BUDGET-ACTIONS (O)</p>	invoked via forward chaining whenever total-cost changes
<p>for any office O if the total-cost of O > the budget of O - 50 then start ANIMATE-ICON(O)</p>	invoked via forward chaining whenever total-cost changes

Error Handling

Provides simple guidelines for recovering from some of the most common errors that novice G2 users make.

Common Errors 205



Common Errors

This appendix lists some common errors and how to recover from them.

Action Buttons Don't Work

Make sure that G2 is running by choosing Main Menu > Start or Resume.

Cannot Enter a Name

Make sure that the name has hyphens in place of spaces. All object names must be unique symbols in G2, for example, `basic-skills-workspace`. Also, make sure that the name is not a reserved word in G2, such as `rule`.

Connection Attributes Not Updating

If an object has an attribute that is based on a connection existing and that attribute is not updating, be sure that the object is actually connected to another object. Sometimes, objects can appear to be connected when in fact they are not.

To test to see if two objects are connected:

➔ Move one of the objects to verify that the connection moves with the object.

Also, connection attributes might not be updating because the rule and/or procedure or method that is controlling the updating might not be correct.

For more information, see [Rule is Not Being Invoked](#) and [Procedures and Methods Not Executing](#).

Rule is Not Being Invoked

To verify if a rule is being invoked:

➔ Choose Main Menu > Run Options > Highlight Invoked Rules.

If the rule does not highlight when you think it should, do one or more of the following:

- Check the **notes** attribute in the rule's table to see if there are any errors.
- If the rule is supposed to be invoked by scanning, verify that the rule has a value for the **scan-interval** attribute in the rule's table.
- If the rule is supposed to be invoked by event detection, verify that the antecedent of the rule is a valid event.
- If the rule is supposed to be invoked by forward chaining, verify that the attribute in the antecedent of the rule is invocable by forward chaining. In general, simple attributes are automatically invocable by forward chaining. If the attribute is a parameter or variable, ensure that the parameter or variable specifies **do forward chain** as one of its options.
- If the rule is not being invoked for the desired class of objects, verify that all references symbols such as attribute names, class names, and local variables are correct. This includes the **for** prefix for creating a generic rule and references to attributes of objects within the rule.

Procedures and Methods Not Executing

If a procedure or method is not executing when you think it should, do one or more of the following:

- Check the **notes** attribute in the procedure or method's table to see if there are any errors.
- If the method is not being invoked for the desired class of objects, verify that the first argument to the method is the class to which the method applies and ensure that the method has a method declaration.
- If the procedure or method is not performing the actions you expect, verify that all references to symbols such as attribute names, class names, and local names are correct.

Runtime Errors

G2 helps to prevent most types of errors during the development process. For example, when you are editing the text of any statement, such as a rule, method, or procedure, G2 prevents you from making syntactic errors as you type.

For an example of this type of error, see [Recovering from Syntactic Errors](#).

G2 detects other types of errors by displaying messages in the **notes** attribute of the object. For example, if you create a rule that refers to an object that does not exist, G2 indicates this in the **notes** attribute of the rule.

For an example of this type of error, see [Recovering from Other Types of Errors](#).

In some cases, G2 cannot detect an error ahead of time. In this case, G2 displays an error on the Operator Logbook. These types of errors are known as **runtime errors** because G2 detects them at runtime, that is, when G2 is running.

For example, suppose you have an expression within a method that refers to the value of an attribute, which has not been specified. When G2 evaluates the expression, it generates a runtime error indicating that the value of the attribute has not been specified.

An example of this in the Getting Started tutorials is the **send-message** method, which refers to the value of the **address** attribute of an office as an expression in square brackets ([]) within an **inform** statement. If you do not specify a value for the **address** attribute of an office and the **send-message** method is invoked for that office, G2 generates a runtime error indicating that the **address** attribute has no value.

Tip You can prevent many runtime errors by providing default values for all attributes when you declare them in the class definition.

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

A

abstraction: *See* generalization.

action: A type of G2 statement that programmatically performs an operation on an object or on the G2 environment in general. G2 supports numerous types of actions, for example, creating and deleting objects, concluding values for attributes, and showing and hiding workspaces.

action-button: A G2 button that performs a sequence of actions when the user clicks the button.

activatable-subworkspaces: A subworkspace of an object on which you place scanned rules, which you activate and deactivate programmatically when certain conditions are met. You use activatable subworkspaces to enable and disable entire portions of your diagram.

administrator-mode: The only built-in user mode, where users can perform any G2 operation.

allow-other-processing-statement: A procedural statement that allows other processes to occur while the procedure is executing.

animate: To change the color of an object's icon or move the icon, then wait a split second, then change the color or move the object again, repeatedly. To perform animation you use subsecond timing and the wait action.

antecedent: The first part of any rule, which tests a condition.

application: Another term for a G2 knowledge base.

argument-list: In a method or procedure, the objects or values that the method or procedure requires to evaluate. In a method, the first argument in the argument list is always the class of objects to which the method applies.

array: A G2 data structure that allows you to store multiple pieces of data in a single attribute slot. The length of an array varies depending on the number of elements in the array.

attribute: A unique characteristic of an object. In object-oriented terms, attributes represent an object's data. You declare the attributes of an object in the class definition, and you specify attribute values in the instance.

attribute-display: An attribute value displayed next to the icon, with or without the attribute name.

attribute-table: A display that lists all of an object's attributes, which you can edit.

B

background: The gray area of the G2 window that is behind the workspaces. You click the background to display the G2 Main Menu. The only type of object you can place on the G2 background is a workspace.

background color: The color of the area of the workspace on which you place objects.

begin statement: Marks the beginning of the body of a procedure or method. A begin statement must appear with an end statement.

C

call action: Executes an embedded procedure or method and does not return control to the calling procedure or method until the entire embedded procedure or method has finished executing. *Compare with* start action.

case statement: A procedural programming statement that allows you to test an attribute against multiple possible values.

change action: A G2 action that dynamically changes various aspects of the knowledge base, for example, the color of an icon or the text of an attribute.

change attribute: An attribute of a class definition that allows you to change manually overridden attribute values of instances so they correspond to an updated class definition. G2 does not automatically change manually overridden attribute or stubs of existing instances whose class definition has been updated.

change the text of action: A G2 action that changes the text of an attribute to a new string.

class: An object-oriented term for a "template" that defines the common characteristics (attributes) and behaviors (methods) of its instances. A class can inherit its definition from a built-in G2 class or user-defined class.

class definition: A built-in G2 object that lets you define the common characteristics of each instance of the class. You specify the class name, the class-specific attributes, the icon, and the stubs in the class definition. You also specify the user interface configurations of instances in the class definition. For example, you use class definitions to create subclasses of objects, connections, variables, and workspaces. *See also* subclass.

class hierarchy: A representation of a set of related classes that describes how the classes at each level inherit their definitions from other classes. *See* inheritance.

class name: The name of a class of objects. When you create an object dynamically, you create it by using the create action and by specifying the class name.

class-specific attributes: The attributes of a class, which you declare in the class definition. The class-specific attribute declarations include the attribute name, the default value, and the data type or subobject class, for example, a variable or parameter.

clone: To create an exact duplicate of the original object except for the name. You can clone objects interactively or programmatically.

composite object: An object-oriented term for an object that contains another object. G2 supports two ways of creating composite objects: attributes of objects that contain subobjects, such as variables or parameters, and objects that define subworkspaces.

configuration: *See* user interface configuration.

connection: A graphical link between two or more objects. You use connections in a G2 application to represent visually how objects are related. You can reason about objects based on their connections. *See also* stub.

connector formats: In a trend chart, the lines between the points for each plot.

consequent: The second part of a rule, which draws a conclusion when the condition in the antecedent is satisfied. *See also* antecedent.

create action: A G2 action that dynamically creates an object based on its class.

create by cloning action: A G2 action that creates an instance of a class by cloning a master object. One reason for creating objects by cloning is to create objects with subworkspaces dynamically.

D

data type: A description of the type of value. The available data types are: integers, floating point numbers, quantity (integer or float), symbols, logical values (true or false), fuzzy truth values, and text strings. You can specify data types in the class-specific attributes of a class definition, in the arguments of methods and procedures, and in the local names of methods and procedures.

data-driven processing: One way in which G2 invoke rules. When the value of an attribute referenced in the antecedent of a rule changes, G2 invokes the rule, using data-driven processing. For example, when a real-time signal updates or when the user enters a value in an end user display, G2 invokes rules that refer to the attribute whose value has changed. *See also* event-driven processing.

definition: *See* class definition.

delete action: A G2 action that deletes objects programmatically. To delete an object, you must first make it transient by using the make transient action.

dependent module: A module that requires additional information contained in one or more other modules to run. Specifically, a dependent module contains instances of classes defined in another module. *See also* independent module.

developer mode: A user-defined mode in which users can configure the layout of a schematic diagram, display the subworkspace of an object, and edit certain attributes in tables.

diagonal connection style: A type of connection that connects objects with diagonal lines. *Contrast with* orthogonal connection style.

dial: An end user display that displays numeric data in a circular dial.

differentiation: A technique for creating subclasses in a class hierarchy, whereby the subclasses define exceptions and special cases. *Also known as* specialization.

digital clocks: An end user display that shows the current G2 time.

direction of flow: The direction in which information flows between connected objects. You can reason about connected objects based on whether the connection is an input or output connection.

display: A visual representation of real-time numeric data that the knowledge base receives or computes. You use displays to create an end user interface for an application.

do in parallel statement: A procedural statement that allows G2 to execute multiple procedural statements simultaneously. *Also known as* parallel processing.

do loop: A procedural programming statement that iterates over a set of instances of a class. You define a **do** loop within a **for** statement in a procedure or method.

drag: To hold down the mouse button while moving the mouse to a new location, then to lift the mouse button to place an object on a workspace or to place text in a text box.

dynamically: *See* programmatically.

E

embedded procedure or method: A procedure or method that another procedure or method starts as one of the calling procedure or method's actions.

end statement: Marks the end of the body of a procedure or method. An **end** statement must appear with a **begin** statement.

encapsulation: An object-oriented term that means to organize related knowledge together in an object and to keep the details of an object hidden from other objects. You encapsulate knowledge by creating subclasses, subobjects, and subworkspaces.

end user display: *See* display.

event detection: When G2 responds to a certain set of prescribed events, such as moving an object on a workspace, creating or deleting connections, and receiving or failing to receive a value. You use a **whenever** rule to perform event detection.

event-driven processing: One way in which G2 invokes a rule by responding to real-time events. For example, G2 can detect these events to trigger a rule, using event-driven processing: moving an object on a workspace, receiving a value from a data source or failing to receive a value, and creating and deleting an object or connection. *See also* [event detection](#) and [data-driven processing](#).

every clause: In a rule, a statement that allows you to refer to all members of a class. You cannot use an **every** statement in a procedure or method; you use a **for** loop instead.

exit if statement: A procedure statement that you place after the last statement in a **repeat** loop to cause the looping to stop under certain conditions.

F

for each statement: A procedural programming statement that refers to all members of a class of objects that meet a certain criteria. G2 iterates over the set, using a **do** loop.

for prefix: A prefix for any type of rule that makes the rule apply to all members of a class that meet a certain criteria. *See also* generic rule.

foreground color: The color of all objects on a workspace whose color is not otherwise specified, for example, free text, button text, and connections.

forward chaining: A mechanism for invoking if rules, whereby G2 invokes the rule each time the value of the attribute in the antecedent changes. Forward chaining is one example of data-driven processing.

frame style: An object that describes the border of a workspace.

free text: An object that you use for labeling a workspace.

G

G2 application server: The G2 software development environment that provides the full range of features needed to develop and deploy intelligent real-time applications. The G2 application server bridges the gap between a traditional application server, an intelligent application, and a client. *See also* [G2 utility](#).

G2 core: *See* [G2 application server](#).

G2 Dialog Utility (GDU): A G2 module that provides tools for creating custom Windows dialogs.

G2 Error Handling Foundation (GERR): A G2 module that provides tools for error and communication handling.

G2 Foundation Resources (GFR): A G2 module that provides module management, messaging, localization, and communications handling for large applications.

G2 Menu System (GMS): A G2 module that allows you to create a standard top menu bar.

G2 Online Documentation (GOLD): A G2 module that supports online documentation, context-sensitive help, and online search capabilities from within G2. GOLD launches an external browser that displays HTML files.

G2 Run-Time Library (GRTL): A G2 module that provides tools for managing large applications and building user interfaces.

G2 User Interface Development Environment (GUIDE): A G2 module that supports the creation of end user dialogs.

G2 User Interface Library (UIL): A supporting module of GUIDE that allows programmatic controls over end user interactions.

G2 utility: Optional components of the G2 application server that allow you to perform specific functions

G2 XL Spreadsheet (GXL): A G2 module that provides a spreadsheet-like display for entering and displaying tabular data.

generalization: A technique for creating subclasses in a class hierarchy, whereby you define similar information in a single class, which subclasses inherit automatically. *See also* differentiation.

generic rule: A rule that applies to any instance of a class or one or more instances of a class that meet certain criteria, for example, on a given workspace, connected to another object, or nearest to another object. To create a generic rule, you use the for prefix.

graphical user interface (GUI): G2 objects that enable end users to interact with and view the running application. Action buttons, trend charts, and readout tables are examples of graphical user interface components.

graph: A type of display that plots a single attribute value over time.

H

hierarchical view: A technique for organizing an application where the details of objects are hidden on their subworkspaces. *See also* composite object.

I

icon: A graphical representation of an object. Every permanent item in G2 has an icon representation.

Icon Editor: A tool that lets you define complex icons for objects that consist of multiple layers and regions, which you can animate.

icon layer: An area of an icon that consists of any number of graphical elements of the same color. Any number of icon layers can form an icon region.

icon region: One or more icon layers that specify a name. You can change the color of an icon region dynamically.

if rule: A type of rule that performs data-driven processing by testing the condition in the antecedent and taking the actions in the consequent if the condition is true.

if-then statement: A procedural programming statement that allows you to test conditions within the program and take actions based on the conditions being met.

image definition: An object that specifies the name of a bitmap image and a filename, which you can use as the background of a workspace.

in order clause: Indicates how G2 should execute a sequence of actions; G2 executes them in the order specified. You use an **in order** clause in several places in G2, including rules, action buttons, and user menu choices.

independent module: A module that contains all the information it needs to run in a single module. Specifically, an independent module contains class definitions for instances contained in the module. *See also* dependent module.

inference engine: The internal G2 mechanism that invokes rules. *See* inferencing.

inferencing: A mechanism whereby rules infer attribute values when certain conditions are met. *See* rule.

inform action: A G2 action that sends messages to the Message Board or to a workspace.

inheritance: An object-oriented term that describes the mechanism whereby classes of objects obtain their definitions from other classes. You only need to define the unique attributes and methods of a subclass; the subclass inherits all the attributes and methods of the superior class. *See* class hierarchy.

initially rule: A type of rule that performs event-driven processing by invoking the rule when the user starts the application running.

Inspect facility: A feature that allows you to locate objects in the knowledge base and view various hierarchies, for example, the module hierarchy, the class hierarchy, or the workspace hierarchy.

instance: An object-oriented term that refers to the individual members of a class. *See also* object.

instance configuration: A type of user interface configuration that specifies the behavior of each instance of a class in different user modes. *See also* item configuration.

intelligent real-time system (IRTS): A software environment, such as G2, for developing and deploying intelligent, mission-critical, client/server applications.

invoke: To evaluate a rule by testing the antecedent of the rule. When G2 invokes a rule, it tests the antecedent of the rule and takes the actions in the consequent of the rule when the antecedent is true. G2 can invoke rules, using event-driven processing and data-driven processing, depending on the type of rule.

is given by expression: An expression that allows you to declare a class-specific attribute of a class that is an instances of another class. An object with an attribute that is given by another object is one example of a composite object. *See* composite object.

item: A class of objects that encompasses all built-in G2 classes, including workspaces, definition classes, rules, methods, and procedures. User-defined classes can inherit their definitions from the **item** class or any subclass of item. While many instances of classes in G2 are actually items, for example, workspaces, rules, procedures, and methods, we typically refer to them in object-oriented terms as objects, rather than items. *See also* object.

item configuration: A type of user interface configuration that specifies the behavior of a particular object in different user modes. *See also* instance configuration.

K

KB: *See* knowledge base.

KB configuration: A type of user interface configuration that specifies the behavior of the overall KB in different user modes. *See also* item configuration and instance configuration.

KB Workspace menu: A menu that allows you to manipulate workspaces and create objects on workspaces. You use this menu to create objects such as class definitions, rules, methods, procedures, and displays.

knowledge base: An ASCII file with a *.kb* extension that contains all the information your application needs to run. A knowledge base consists of one or more modules.

L

layered products: Applications built on top of G2 that support development in specific domains.

list: A G2 data structure that allows you to store multiple pieces of data in a single attribute slot. A list has a fixed length and allows you to refer to any of its elements.

localization: The translation of an application into another language for local use. In G2, you use the G2 Foundation Resources (GFR) module to provide text keys instead of hard-wired text, which G2 substitutes with actual text when the KB runs. You can also use this feature to perform dynamic substitution of text.

local name: A symbol that a statement uses locally. You use local names in many places in G2, including rules, methods, procedures, and action buttons. *See also* local name declaration.

local name declaration: A list of symbols that a method or procedure uses locally in the method or procedure body.

looping: A procedural programming technique that allows you to repeat a statement until a condition is met. *See* **repeat** statement and **do** loop.

M

make permanent action: A G2 action that makes an object a permanent part of the KB. When you create an object dynamically, you must make it permanent for it to be saved in the knowledge base. *See also* [permanent knowledge](#), [transient knowledge](#), and [make transient action](#).

make transient action: A G2 action that makes an object transient. When you delete an object dynamically, you must first make it transient. *See also* transient knowledge, permanent knowledge, and **make permanent action**.

master object: An object that you clone dynamically to create an instance of a class with a fully configured subworkspace. *See* **create by cloning** action.

Message Board: A special workspace on which G2 displays messages, which you send by using the **inform** action.

message: An object that allows you to create text, which you can use to communicate with the operator.

meter: An end user display that shows numeric data in a vertical bar format.

method: A named object associated with a particular class that executes a sequence of actions when the application starts the method. In object-oriented terms, a method represents the object's behaviors.

method body: The part of a method where you specify the sequence of actions that the method executes. The method body starts with a **begin** statement and ends with an **end** statement.

method declaration: A type of definition object that establishes the name of each method that exists in the knowledge base.

method name: The name you specify when you invoke the start action to execute an object's method.

module: A set of related information contained in a KB. A knowledge base must define at least one module. A module that does not require any other modules is

an independent module, and a module that requires one or more other modules to run is a dependent module. *See also* module hierarchy.

module hierarchy: A representation of a set of related modules that describes the dependencies between modules. Modules that are located above other modules in the hierarchy depend on modules that are lower in the hierarchy.

N

name: A symbol that identifies an object in a knowledge base. Class names must be unique; however, object names do not.

natural language text editor: A feature of the G2 text editor that prompts you at each stage as you edit a statement that requires syntax, for example, a rule, method, or procedure.

notes: A built-in attribute of any object where G2 displays system messages and errors about the object.

O

object: A piece of information that contains all related knowledge in one location. In object-oriented terms, an object contains all the data (attributes) and behaviors (methods) that completely describe the object. In G2, you use objects to represent the physical systems in your application and the connections between those systems.

object class: A built-in class from which most user-defined classes inherit their definition. *See also* item.

object hierarchy: A representation of a set of related objects that describes how the objects at each level are related. You create an object hierarchy by creating subworkspaces of objects or by creating attributes of objects that contain other objects, such as variables and parameters. *See also* composite object.

object menu: A menu that you use to interact with an object. The object menu allows you to perform numerous operations on an object, including delete, clone, and create a subworkspace. You can also define user-defined menu choices on the object menu for instances of a class. You display the object menu by clicking on any object. *See also* user menu choice.

object-oriented: A software development environment that represents certain kinds of knowledge as objects. G2 is an object-oriented development environment for real-time applications. Object-oriented programming has numerous features including: classes, inheritance, instances, encapsulation, attributes, methods, and composite objects. The benefits of object-oriented design are also numerous: faster development time, easier to maintain, scalability, adaptability to other applications, and reduced cost.

operate on area: A workspace operation that lets you perform operations on a group of objects, such as move, clone, or delete.

Operator Logbook: An area of the screen where G2 displays system messages and errors.

orthogonal connection style: A type of connection that connects objects with either vertical or horizontal lines, with possible bends in the connection.

P

parallel processing: *See* do in parallel statement.

parameter: A built-in G2 class that keeps a history of values over time. *See also* variable.

permanent knowledge: Knowledge that exists in the knowledge base and is located on a workspace. G2 retains permanent knowledge when you reset the knowledge base. *See also* transient knowledge.

plot: In a trend chart, the data value to plot. A trend chart can contain multiple plots.

point format: In a trend chart, a graphical element at each data point for each plot.

procedure: A named object that executes a sequence of actions when the application starts the procedure. A procedure is independent of any class. *See also* [method](#) and **start** action.

procedure body: The part of a procedure where you specify the sequence of actions that the procedure executes, which starts with a **begin** statement and ends with an **end** statement.

procedure name: The name you specify when you invoke the **start** action to execute the procedure.

programmatically: A technique for performing operations on objects and the G2 environment that uses G2's real-time procedural programming language.

prototype: A model of a real application that simulates the real-time environment.

Q

qualified name: The name of a method, which concatenates the class name and the method name, using double colons. The class-qualified name indicates which specific method an object should execute when the application starts the method. For example, `small-office::create-connection`.

quantity: A G2 data type that is either a floating point number or an integer.

R

readout table: An end user display that displays numeric data in a small table.

relation: A non-graphical “connection” between two or more objects. *See also* connection.

repeat statement: A procedure statement that executes actions in a loop.

reserved word: A symbol that is part of the G2 environment, which you cannot use as the name of an object or attribute. Reserved words include any G2 syntax element or built-in class name, for example, *rule*, *method*, *class*, *object*, *the*, *create*, and so on.

rule: An object that tests conditions and draws conclusions. G2 invokes rules in one of two ways, depending on the kind of rule: using event-driven processing or using data-driven processing. Rules represent the heart of G2’s inference engine.

runtime error: An error that is displayed on the Operator Logbook while G2 is running. Many runtime errors occur because a default value has not been specified for an attribute whose value is evaluated at runtime.

S

scan: To invoke a rule at a periodic interval, which you specify.

scrapbook: A G2 workspace that holds text you copy and cut from the G2 text editor.

scrollable text editor: A text editor with scroll bars, which you get when you edit certain types of attributes. To move to the next line in a scrollable text editor, press Enter. To accept the edits in a scrollable text editor, use the Ctrl + Enter command.

show action: A G2 action that displays a workspace programmatically at any location on the screen and at any scale.

shrink wrap: To adjust the borders of a workspace to minimize its size.

simple attribute: *See* untyped attribute.

specialization: *See* differentiation.

start: To evaluate a method for a specific object with specific arguments or to execute a procedure with specific arguments. *See also* **start** action.

start action: A G2 action that starts a method or procedure. The **start** action executes the embedded procedure or method and immediately returns control to the calling procedure or method.

strong typing: A technique whereby you always declare an attribute value, using the most restrictive data type possible.

stub: A connection attached to one object but not yet attached to another object. You specify the default stubs for a class in the class definition.

subclass: An object-oriented term for a class that is located below another class in the class hierarchy.

subobject: An object that contains another object, such as an attribute of an object that contains a variable or parameter or an object that defines a subworkspace. *See also* composite object.

subsecond timing: A way of controlling the timing of any procedural statement down to any fraction of a second. You use subsecond timing to do animation.

subtable: A table associated with a subobject, which is embedded in the object's table.

subworkspace: A workspace associated with an object. *See also* composite object.

superior class: A class from which a subclass inherits its definition. A superior class is located above another class in the class hierarchy. *See also* object hierarchy.

superior object: An object that defines a subworkspace or an object that defines an embedded object in an attribute.

symbol: A G2 data type that is a string of alpha-numeric characters with no spaces. All names must be symbols in G2, including class names, object names, attribute names, method names, and procedure names.

syntax-guided text editor: *See* natural language text editor.

system-defined attribute: An attribute that G2 defines for items, for example, the Background-color of a workspace.

system tables: A set of default parameters that determine the behavior of various aspects of the G2 environment, including fonts, colors, log files, the text editor, and timing.

T

temporal reasoning: A technique for reasoning about objects over time by looking at a history of attribute values. You use variables and parameters to perform temporal reasoning in G2.

Telewindows: A client application that provides remote access to the G2 server by users on a network. You use Telewindows in a concurrent team development environment, as well as in a client/server deployment environment.

Telewindows Next Generation: A Windows user interface for G2 developers and end users. G2 provides a rich set of tools for developing Windows user interfaces for end user applications, including menus and toolbars, standard and custom dialogs, a variety of Windows views, including tree views, shortcut bars, chart views, property grids, and workspace views, and tabbed MDI mode.

text: A data type that is a sequence of alpha-numeric characters, which can include spaces, and which requires quotation marks surrounding the characters.

text editor: A special workspace for entering values into attributes and editing the text of procedural statements, such as rules, methods, and procedures. *See also* scrollable text editor.

the clause: A reserved word that allows you to refer to an attribute of an object in an expression. For example, the **Network-type** of **office-1** refers to the attribute named **Network-type** for the object named **office-1**.

the item expression: An expression that refers to the current item. For example, you use **the item** in the action of a user menu choice to refer to the object with the user menu choice.

the item superior to expression: An expression that refers to the item that is above another item in the object hierarchy. For example, you use this expression to refer to an object that defines a subworkspace.

this workspace expression: An expression that refers to the current workspace. For example, you use this expression in the action of an action button to create an object on the workspace on which the action button exists.

time axis: The horizontal axis of a trend chart.

top-level module: The module at the top of the module hierarchy.

transfer action: A G2 action that programmatically moves an object from its current workspace to another workspace at a particular location. When you create objects dynamically by using the **create** action, you must transfer them to a workspace after you make them permanent by using the **make permanent** action.

transient knowledge: Knowledge that exists but is not a permanent part of the knowledge base. G2 deletes transient knowledge when you reset G2. When you create an object programmatically, G2 creates a transient item. To make an item permanent, you use the **make permanent** action. Before you can delete an item programmatically, you must make it transient by using the **make transient** action. *See also* permanent knowledge.

trend chart: An end user display that plots a history of any number of data values and gives complete control over the layout of the chart.

trend chart format: In a trend chart, a subobject that controls the background color of the chart.

type checking: A technique for validating the data type of an attribute when the value is supplied, either by G2 or by the user. You declare an attribute to have type checking in the class definition.

typed attribute: An attribute that specifies a data type and performs type checking.

U

unconditionally rule: A type of rule that performs data-driven or event-driven processing by taking the action in the consequent automatically whenever the rule is invoked.

untyped attribute: An attribute with no data type that performs no type checking. The value of an untyped attribute can be any alpha-numeric sequence of characters, without spaces.

update interval: The interval at which G2 updates a data value or the current value of a display.

user-defined attributes: Attributes that users declare in the class definition. *See also* system-defined attribute.

user interface configuration: A statement that configures the behavior of any class of objects or the overall KB for different user modes. *See also* [item configuration](#) and [instance configuration](#).

user menu choice: A menu choice associated with each instance of a class that appears in the object menu. A user menu choice specifies a sequence of actions that G2 performs on a particular object when the user chooses the user-defined menu choice.

user mode: A designation of the class of users of an application, which determines the actions the user can perform on various objects and the behavior of the overall environment. *See also* user interface configuration.

V

value axis: In a trend chart, the vertical axis of each plot.

value: A unique setting for the attribute of an instance. Attribute values can be typed or untyped. You specify attribute values in an instance.

variable: A built-in G2 class that keeps a history of values over time and allows you to connect to real-time data.

W

wait statements: A procedure statement that causes G2 to wait a period of time before it executes the next statement.

when rule: A type of rule that performs event-driven processing by invoking the rule when the expression in the antecedent is true.

whenever rule: A type of rule that performs event-driven processing by detecting the event in the antecedent and taking the actions in the consequent when G2 detects the event.

workspace: An area of the knowledge base that contains objects and that sits on the background of the G2 window. You create objects and definitions on a workspace by using the KB Workspace menu.

workspace hierarchy: A representation of a set of related workspaces that describes how the workspaces at each level are related. You create a workspace hierarchy by creating subworkspaces of objects. *See also* composite object.

@ A B C D E F G H I J K L M
 # N O P Q R S T U V W X Y Z

Symbols

... in text editor
 [] in statements

A

abstraction
 action attribute
 in action buttons
 in user menu choices
 action buttons
 creating
 permanent objects, using
 transient objects, using
 deleting objects, using
 iconifying a workspace, using
 testing applications, using
 action-button menu choice
 actions
 introduction to
 performing
 in order
 in rules
 on classes of objects
 activatable subworkspaces
 add name of attribute menu choice
 and reserved word
 animation
 animating objects, using
 creating methods for
 using subsecond timing for
 antecedents, of rules
 applicable-class attribute
 applications
 video conferencing
 argument lists
 declaring
 first argument for methods
 for procedures and methods
 of procedures and methods
 attribute access
 attribute displays

 creating for classes of objects
 displaying with objects
 attribute tables
 displaying
 attribute-initializations attribute
 initializing
 attribute displays
 stubs
 user-defined attributes
 attributes
 assigning values to
 changing manually overridden
 class-specific
 concluding values for
 declaring
 as given by a parameter
 for computed values
 displaying values of
 editing
 in tables
 simple
 using action buttons
 introduction to
 of objects
 referencing in rules
 types of

B

background graphics
 background, G2
 background-color
 attribute
 menu choice
 begin statement
 borderless-free-text class

C

C, C++, comparison with G2
 case statements
 CASE tools, comparison with G2
 change action

- Change Mode menu choice
- class definitions
 - attribute displays of
 - class-specific attributes of
 - displaying
 - editing
 - icons of
 - introduction to
 - organizing into modules
 - stubs of
- class hierarchy
 - creating for objects
 - showing for G2
- class name
- class-definition menu choice
- classes
 - creating instances
 - interactively
 - programmatically
 - creating rules for instances of
 - executing actions on
 - introduction to
 - overriding default methods of
- Class-inheritance-path attribute
- Class-name attribute
- class-specific attributes
- Class-specific-attributes attribute
- clone menu choice
- Clone Workspace menu choice
- color menu choice
- colors
 - changing
 - for icons, programmatically
 - for workspaces, interactively
- composite objects
- conclude action
- configurations
 - See also* user interface configurations
 - configuring
 - diagrams
 - instances
 - introduction to
- connected to statement
- connection class
- connection styles
 - diagonal
 - orthogonal
- connections
 - counting
 - creating interactively
 - deleting

- interactively
- programmatically
- detecting
 - when connected
 - when deleted
- introduction to
- stubs
 - creating interactively
 - creating programmatically
 - deleting interactively
- connector formats
- consequents, of rules
- create instance menu choice
- create subworkspace menu choice
- Ctrl + j command
 - inserting line feed, using
- customer support services

D

- data types
- data-driven processing
- data-window-background-color attribute
- delete menu choice
- delete name of attribute menu choice
- Delete Workspace menu choice
- dependent modules
- description-of-frame attribute
- dials
- differentiation
- digital clocks
- direction of flow
- direct-superior-classes attribute
- displays
- display-update-interval attribute
- do in parallel statements
- do loop
- Do Not Highlight Invoked Rules menu choice
- dragging objects
- Drop to Bottom menu choice
- dynamic
 - See* programmatic

E

- edit icon menu choice
- editor
 - natural language
- embedded procedures and methods
- encapsulation

- using classes
- using methods

end statement

end user displays

errors

- common
- in **Notes** attribute
- runtime
- syntactic

event detection

event-driven processing

every statement

expression-to-display attribute

F

file-name-of-image attribute

float data type

for each statement

for prefix, in rules

forward chaining

- data-driven processing, using
- explicitly allowing in parameters and variables

frames, workspace

frame-style attribute

frame-style-definition menu choice

free-text class

G

G2

- integration with other technologies
- overview of
- pausing
- running
- showing the class hierarchy of
- shutting down
- starting server

G2 Foundation Resources (GFR)

- communicating with users, using
- localizing text, using
- managing modules, using
- starting an application, using

G2-user-mode attribute

GDI

- See* G2 Developer? Interface

generalization

generic rules

- using alternate form

- using for prefix

Get Workspace menu choice

GFR

- See* G2 Foundation Resources

GMS

- See* G2 Menu System

go to original menu choice

go to subworkspace menu choice

GOLD

- See* G2 Online Documentation

graphs

GUIDE

- See* G2 User Interface Development Environment

GXL

- See* G2 XL Spreadsheet

H

hide attribute display menu choice

hide name menu choice

Hide Workspace menu choice

hierarchical views

Highlight Invoked Rules menu choice

history

- creating attributes that keep
- plotting on trend charts
- using variables and parameters to keep

history-keeping-spec attribute

I

Icon Editor

icons

- changing colors of
- editing
- of classes
- of objects

if rules

- forward chaining, using
- introduction to

if-then statements

Include-in-legend? attribute

indentation

- of methods and procedures
- rules for

independent modules

inference engine

inferencing

inform the operator statement

- inheritance
- initializable-system-attributes attribute
 - for subclasses of parameters
 - for user-defined classes
- initially rules
 - introduction to
 - showing workspaces on startup, using
- Inspect menu choice
 - locating transient objects, using
 - showing the class hierarchy, using
 - showing the module hierarchy, using
- instance configurations
- instance-configuration attribute
- instances
 - See also* objects
 - creating
 - for subclasses
 - interactively
 - introduction to
 - organizing into modules
- integer data type
- invoking rules
- is connected to event expression
- is disconnected from event expression
- is given by statement
- item class
- item configurations
- item-configuration attribute
- items

K

- KB configurations
 - .kb* files
 - video conferencing application
- KB tutorials
 - KB files for
 - loading
- KB Workspace menu
- KBs
 - See* knowledge bases
- kb-workspace class
- knowledge bases
 - introduction to
 - loading
 - pausing
 - running
 - saving
 - starting in particular user modes
 - video conferencing application

- working with

L

- label attribute
- label-to-display attribute
- line feeds
 - inserting in statements
- lists
- Load KB menu choice
- loading
 - knowledge bases
- local names
 - declaring for procedures and methods
 - using in statements
- Logbook, Operator
- loops

M

- master objects
- menus
 - See also* user menu choices
 - KB Workspace
 - object
- Message Board
- messages
 - sending to operators
 - using expression in
- meters
- method declarations
- method menu choice
- method-declaration menu choice
- methods
 - animating objects, using
 - body of
 - calling
 - creating
 - declaring
 - arguments for
 - first argument for
 - using method declarations
- errors in
- format of
- informing the operator, using
- introduction to
- local name declarations of
- looping in
- name of
- of objects

- overriding for classes
- sending a message, using
- sequential processing, using
- starting
 - example of
 - from a procedure
 - waiting between actions in
- minimum-scheduling-interval attribute
- modes, user
- modules
 - introduction to
 - organizing knowledge in
 - samples for video conferencing
 - application
 - viewing

N

- Name menu choice
 - KB Workspace menu
- name menu choice
 - object menu
- names attribute
- natural language text editor
- New Free Text menu choice
- New Object menu choice
- New Rule menu choice
- New Workspace menu choice
- non-menu choices
 - configuring
- Notes attribute
 - displaying in table

O

- object class
 - inheriting definition from
 - showing class hierarchy of
- object hierarchy
- object menu
- object-oriented development environment
- objects
 - animating
 - cloning
 - interactively
 - programmatically
 - connecting
 - introduction to
 - using stubs
 - creating

- interactively
- permanent
- programmatically
- using action buttons
- deleting
 - interactively
 - programmatically
 - using action buttons
- interacting with
- introduction to
- making
 - permanent
 - transient
- moving on workspaces
- naming
- operating on groups of
- subworkspaces of
- of reserved word
- Operator Logbook
 - hiding
 - runtime errors in
- operators, informing
- options attribute

P

- parameter class
- parameters
 - creating subclasses of
 - keeping a history, using
- Pause menu choice
- permanent knowledge
- plots
- plots menu choice
- point formats
- procedures
 - body of
 - declaring arguments for
 - errors in
 - format of
 - introduction to
 - local name declarations of
 - name of
 - sequential processing, using
 - starting
 - from rules
 - methods, using
- programmatic
- prototypes

Q

quantitative-parameter class
quantity data type

R

random function
range-bounds attribute
range-mode attribute
readout tables
 creating
 invoking rules, using
readout-table menu choice
receives a value event expression
relations
reserved words
Restart menu choice
Resume menu choice
rules
 animating objects in
 counting connections, using
 creating
 disabling highlighting for
 errors in
 forward chaining in
 generic
 creating for classes of objects
 creating, using alternate form
 creating, using for prefix
 highlighting invoked
 if
 inferencing techniques for
 initially
 introduction to
 invoking
 by scanning
 techniques for
 using data-driven processing
 using event-driven processing
 using readout tables
 making robust and efficient
 performing actions in
 referencing
 attributes in
 superior objects in
 simple
 types of
 unconditionally
 when
 whenever

Run menu
runtime errors

S

Save KB menu choice
scan-interval attribute
scanning
 invoking rules by
 simulating real-time data by
sequential processing
show attribute display menu choice
shrink wrapping
Shut Down G2 menu choice
simple attributes
 editing
 introduction to
simulation, of real-time data
specialization
start action
Start menu choice
starting, methods and procedures
statements
 performing multiple actions in order in
 using proper indentation in
strong typing
stubs
 connecting objects, using
 creating programmatically
 editing for classes
 introduction to
stubs attribute
subclasses
 creating
 for parameters
 for user-defined classes
 creating class hierarchies, using
 of objects
subobjects
subsecond timing
subtable menu choice
subtables
subworkspaces
 creating
 for objects
 interactively
 programmatically
 creating end user interface objects on
superior classes
 creating class hierarchies, using

- of objects
- superior objects
- symbol data type
- symbols
- syntax-guided text editor
- system tables
- system-defined attributes

T

- table menu choice
- tables
 - displaying
- tabs
 - inserting in statements
- Telewindows
 - starting client
- temporal reasoning
- text
 - displaying on workspaces
- text data type
 - in Label attribute
 - in local name declarations
- text editor
 - ellipses (...) in
 - indenting statements in
 - introduction to
 - natural language
 - recovering from errors in
 - scrollable
- the count of each statement
- the item superior to expression
- the reserved word
- then reserved word
- this workspace statement
- time axes
- Timing Parameters system table
- top-level directory
 - video conferencing application
- top-level module
- transient knowledge
- trend chart format subtable menu choice
- trend chart formats
- trend charts
- trend-chart menu choice
- truth-value data type
- tutorials
 - KB files for
 - loading
- type checking

- typed attributes

U

- UIL
 - See G2 User Interface Library
- unconditionally rules
 - concluding values, using
 - introduction to
- untyped attributes
- update interval
- update-interval attribute
- user interface configurations
- user menu choices
 - creating
 - introduction to
- user modes
 - introduction to
- user-defined attributes
- user-menu-choice menu choice

V

- value axes
- value axes menu choice
- values, of attributes
- variables
- video conferencing application
 - knowledge bases of
 - loading
 - module configurations of
 - top-level directory of

W

- wait statements
- when rules
- whenever rules
 - creating
 - introduction to
 - performing event-driven processing, using
- workspace hierarchy
- workspaces
 - background graphics for
 - changing colors of
 - cloning
 - creating
 - deleting
 - displaying text on
 - dropping

- frames for
- hiding
- iconifying
- interacting with
- introduction to
- keyboard commands for
- lifting
- making attractive and informative
- moving
 - objects on
 - to new location
- naming
- operating on groups of objects on
- showing
 - interactively
 - on startup
 - programmatically
- shrink wrapping
- subclasses of