GemStone

GemBuilder for VisualWorks

July 1996

GemStone

Version 5.0

IMPORTANT NOTICE

This manual and the information contained in it are furnished for informational use only and are subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this manual or in the information contained in it. The manual, or any part of it, may not be reproduced, displayed, photocopied, transmitted or otherwise copied in any form or by any means now known or later developed, such as electronic, optical or mechanical means, without written authorization from GemStone Systems, Inc. Any unauthorized copying may be a violation of law.

The software installed in accordance with this manual is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Limitations

The software described in this manual is a customer-supported product. Due to the customer's ability to change any part of a Smalltalk image, GemStone Systems, Inc. cannot guarantee that the Gemstone Smalltalk Interface will function on all Smalltalk images.

Trademarks

GemStone is a registered trademark and **GemBuilder** is a trademark of GemStone Systems, Inc.

VisualWorks and **ParcPlace Smalltalk** are trademarks of ParcPlace Systems, Inc. , its subsidiaries, or successors and are registered in the United States and other countries

ENVY is a registered trademark of Object Technology International, Inc.

Microsoft, **MS-DOS**, and **Windows** are registered trademarks and **Windows NT**, and **Win32** are trademarks of Microsoft Corporation in the U.S.A. and other countries.

OS/2 and **OS/2 Warp** are trademarks of International Business Machines Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Preface

About This Manual

This manual describes GemBuilder, an environment for developing Gemstone applications using VisualWorks.

GemBuilder consists of two parts: a programming interface between your Smalltalk application code and the GemStone object repository, and a GemStone programming environment.

The programming interface provides facilities to:

- allow objects to be transparently replicated and maintained in both client Smalltalk and GemStone or allow some objects to reside only in GemStone but appear as Smalltalk objects;
- import GemStone objects into Smalltalk; and
- create and browse connections between GemStone and Smalltalk objects that optimize data transfer between the environments.

The GemBuilder programming environment provides the following integrated tools for programming in GemStone's version of Smalltalk:

- A *GemStone System Browser* for examining, creating, and modifying GemStone classes and methods.

- A *GemStone System Workspace* for easy access to commonly used code operations.
- *GemStone Inspectors* for examining and modifying the state of GemStone objects.
- A *GemStone Breakpoint Browser* and a *Debugger* for examining and dynamically modifying the state of a running GemStone application.
- A *Session Browser* for managing sessions and transactions.
- A *Connector Browser* for keeping track of and managing the connectors that establish relationships between client Smalltalk objects and GemStone objects.
- A *Class Version Browser* for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.
- A *Symbol List Browser* for examining the GemStone Symbol Lists that are used for sharing objects and protecting access to objects in a multiuser environment.
- A *Settings Browser* for examining and setting configuration parameters for GemBuilder.
- A *User Account Management Tool* for creating new user accounts, changing account passwords, and assigning group membership.
- A *Segment Tool* for managing GemStone authorization at the object level.

Assumptions

To make use of the information in this manual, you need to be familiar with the GemStone object server and with GemStone's Smalltalk programming language as described in the *GemStone Programming Guide*. That book explains the basic concepts behind the language and describes the most important GemStone kernel classes.

In addition, you should be familiar with the ParcPlace Smalltalk language and the VisualWorks programming environment as described in your Smalltalk vendor's product manuals.

Finally, this manual assumes that the GemStone system has been correctly installed on your host computer as described in the *GemStone System Administrator's Guide* and that your system meets the requirements listed in your *GemBuilder Installation Guide*.

How This Manual is Organized

Chapter 1, Basic Concepts, describes the overall design of a GemBuilder application and presents the fundamental concepts required to understand the interface between client Smalltalk and the GemStone object server.

Chapter 2, Communicating with the GemStone Object Server, explains how to communicate with the GemStone object server by initiating and managing GemStone sessions.

Chapter 3, Sharing Objects, explains how to set up connections between your application and GemStone that make your application's objects persistent and sharable and that allow your application to manipulate objects in the GemStone object server's shared object repository.

Chapter 4, Managing Replicates and Forwarders, describes how GemBuilder coordinates your application's local objects with shared objects in the GemStone repository.

Chapter 5, Managing Transactions, discusses the process of committing a transaction, the kinds of conflicts that can prevent a successful commit, and how to avoid or resolve such conflicts.

Chapter 6, Security and Access to Objects, describes the security mechanisms that are available in GemBuilder and explains how to control access to objects in a multiuser environment. It explains how to use the GemBuilder tools to manage access to objects and administer user accounts.

Chapter 7, Schema Modification and Coordination, explains how GemStone supports schema modification by maintaining versions of classes in class history objects. It describes the Class Version Browser and explains how to use it. It also explains how to synchronize schema modifications between Smalltalk and GemStone.

Chapter 8, Using the GemStone Programming Tools, explains how to use the GemStone browsers and tools to create classes and methods in GemStone and to debug GemStone Smalltalk code.

Chapter 9, Performance Tuning, discusses ways that you can tune your application to optimize performance and minimize maintenance overhead. It describes the configuration parameters available for tuning a GemBuilder application, and it explains how to use the Settings Browser to optimize your application's performance.

Chapter 10, Nontransparent Access to GemStone Objects, discusses some very low-level approaches to tuning GemBuilder applications.

Chapter 11, Exception Handling, discusses errors: how to handle them and how to recover from them.

Appendix A, GemBuilder Classes and Proxies, lists the GemStone objects that are predefined as "proxy objects" within your client Smalltalk application.

Appendix B, Network Resource String Syntax, describes the syntax for network resource strings, a means for uniquely identifying a GemStone file or process by specifying its location on the network, its type, and authorization information.

Appendix C, ParcPlace Smalltalk and GemStone Smalltalk, outlines the few general and syntactical differences between the client Smalltalk and GemStone Smalltalk languages.

Other Useful Documents

You will find it useful to look at documents that describe other components of the GemStone data management system:

- The *GemStone Programming Guide* describes the GemStone System and the GemStone Smalltalk language.
- A complete description of the behavior of each GemStone kernel class is available in the *GemStone Kernel Reference.*
- In addition, if you will be acting as a system administrator, or developing software for someone else who must play this role, you will need to read the *GemStone System Administration Guide.*

Technical Support

GemStone provides several sources for product information and support. GemStone product manuals provide extensive documentation, and should always be your first source of information. GemStone Technical Support engineers will refer you to these documents when applicable. However, you may need to contact Technical Support for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.

- You want to submit a feature request.

Questions concerning product availability, pricing, keyfiles, or future features should be directed to your GemStone account manager.

When contacting GemStone Technical Support, please be prepared to provide the following information:

- Your name, company name, and GemStone license number,
- the GemStone product and version you are using,
- the hardware platform and operating system you are using,
- a description of the problem or request,
- exact error message(s) received, if any.

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

For non-emergency requests, you should contact Technical Support by email, Web form, or facsimile. You will receive confirmation of your request, and a request assignment number for tracking. Replies will be sent by email whenever possible, regardless of how they were received.

Email: support@gemstone.com

The preferred method of contact. Please do not send files larger than 100K (for example, core dumps) to this address. A special address for large files will be provided on request.

World Wide Web: http://www.gemstone.com

Technical Support is located under Services. A Help Request Form is available for request submissions. This form requires a password, which is free of charge but must be requested by completing the Registration Form, found in the same location. Allow 24 hours for your registration to be recorded and a password assigned.

Facsimile: (503) 629-8556

When you send a fax to Technical Support, you should also leave a voicemail message to make sure your fax will be picked up as soon as possible.

We recommend you use telephone contact only for more serious requests that require immediate evaluation, such as a production database that is non-operational.

Telephone: (800) 243-4772 or (503) 690-3503

Emergency requests will be handled by the first available engineer. If you are reporting an emergency and you receive a recorded message, do not use the voicemail option. Transfer your call to the operator, who will take a message and immediately contact an engineer.

Non-emergency requests received by telephone will be placed in the normal support queue for evaluation and response.

24x7 Emergency Technical Support

GemStone offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact GemStone 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone account manager for more details.

Contents

Chapter 1. Basic Concepts

1.1 The GemStone Object Server 1-2
1.2 GemBuilder for Smalltalk 1-3
The Programming Interface 1-4
Transparent Access to GemStone 1-4
GemStone's Smalltalk Language 1-5
The GemBuilder Tools 1-6
1.3 Designing a GemStone Application: an Overview 1-7
Which objects should be stored and shared? 1-7
Which objects should be secured? 1-8
Which objects should be connected? 1-8
How should transactions be handled? 1-9
How can performance be improved? 1-9

Chapter 2. Communicating with the GemStone Object Server

2.1 GemStone Sessions 2-2
RPC and Linked Sessions 2-2
2.2 Session Control in GemBuilder 2-3
Defining Session Parameters. 2-4
Defining Session Parameters Programmatically 2-5
2.3 The GemStone Session Browser. 2-7
Starting the Session Browser. 2-7
Supplying Session Parameters 2-8
Removing Session Parameters 2-9
2.4 Logging In To and Logging Out Of GemStone 2-9
Logging In To GemStone Programmatically. 2-10
The Current Session. 2-10
Logging Out of GemStone Programmatically 2-11
Session Management Using the Session Browser 2-12
Logging In 2-12
Setting the Current Session. 2-12
Logging Out of a GemStone Session With the Session Browser 2-13
2.5 Session Dependents 2-13

Chapter 3. Sharing Objects

3.1 Deciding Which Objects to Connect 3-2
Root Objects 3-2
3.2 Connectors 3-5
Session Connectors and Global Connectors 3-5
Kinds of Connectors 3-5
Connecting by Name 3-6
Connecting by Classes 3-7
Connecting by Class Variable Names. 3-7
Connecting by Class Instance Variable Names 3-8
Connecting by Identity: Fast Connectors 3-8
Verification of Connections 3-8
Initializing Connected Objects: The Postconnect Action 3-8
Initializing Class Connectors: Special Considerations 3-9
Connector Nilling 3-10
3.3 The Connector Browser 3-10
The Connector Group Pane 3-11

The Connector List Pane . . . 3-12
The Connector Control Panel . . . 3-12
Postconnect Action . . . 3-13
3.4 Managing Connectors Programmatically . . . 3-15
Creating Connectors . . . 3-16
Setting the Postconnect Action . . . 3-16
Adding Connectors to a Connector List . . . 3-17
Connecting and Disconnecting . . . 3-18
Examples of Session Control Methods . . . 3-18

Chapter 4. Managing Replicates and Forwarders

4.1 Working with Replicates . . . 4-2
4.2 Instance Variable Mapping Between GemStone and Smalltalk . . . 4-2
Suppressing Replication of Individual Instance Variables . . . 4-2
Nonstandard Instance Variable Mapping . . . 4-3
4.3 Object Synchronization . . . 4-4
Faulting Objects Into Smalltalk . . . 4-5
Object Stubs . . . 4-5
Faulting Root Objects . . . 4-6
Explicit Stubbing . . . 4-8
Faulting Modified GemStone Objects . . . 4-8
Controlling Replication of Instance Variables . . . 4-9
Modifying Instance Variables During Faulting . . . 4-10
Defunct Object Stubs . . . 4-12
Flushing Objects to GemStone . . . 4-12
Automatic Marking of Modified Objects . . . 4-13
Modifying Instance Variables During Flushing . . . 4-15
4.4 Class Mapping Between GemStone and Smalltalk . . . 4-17
Predefined Class Connectors . . . 4-17
Automatic Generation of Classes . . . 4-17
User-Created Classes With Different Formats . . . 4-18
4.5 Working with Forwarders . . . 4-19
Sending Messages to Forwarders . . . 4-19
Result Objects . . . 4-20
Replication of Client Smalltalk BlockClosures . . . 4-20
Workarounds for Block Limitations . . . 4-21
Defunct Forwarders . . . 4-21

Chapter 5. Managing Transactions

5.1 Transaction Management: an Overview 5-2
5.2 Operating Inside a Transaction 5-3
Committing a Transaction 5-4
Aborting a Transaction 5-5
Handling Commit Failures 5-5
5.3 Operating Outside a Transaction 5-6
Being Signaled to Abort 5-7
5.4 Transaction Modes 5-8
Automatic Transaction Mode 5-8
Manual Transaction Mode 5-9
Choosing Which Mode to Use 5-9
Switching Between Modes 5-9
5.5 Managing Concurrent Transactions 5-10
Read and Write Operations 5-10
Optimistic and Pessimistic Concurrency Control 5-11
Setting the Concurrency Mode 5-12
Setting Locks 5-13
Releasing Locks Upon Aborting or Committing 5-15
5.6 Reduced-Conflict Classes 5-16
5.7 Changed Object Notification 5-17
Gem-to-Gem Notification 5-18

Chapter 6. Security and Access to Objects

6.1 Object-Level Security 6-2
Requiring Login Authorization 6-2
Controlling Visibility of Objects 6-2
Protecting Methods 6-2
Using GemStone's Authorization Mechanisms 6-2
6.2 Classes for Controlling Access to Objects 6-4
Repository 6-4
Segment 6-4
UserProfile 6-5
6.3 Sharing Access to Objects 6-6
Group Authorization and Object Sharing 6-7

Using Segments for Authorization. 6-7
Making Objects Accessible Through Symbol Lists 6-9
6.4 GemStone Administration Tools 6-10
The Segment Tool 6-10
Segment Definition Area 6-11
Group Definition Area 6-12
Segment Tool Menus 6-13
Using the Segment Tool. 6-16
The Symbol List Browser 6-18
The Clipboard 6-19
Symbol List Browser Menus 6-20
User Account Management Tools 6-22
GemStone User List 6-22
GemStone User Dialog 6-23

Chapter 7. Schema Modification and Coordination

7.1 Schema Modification 7-2
Instance Migration Within GemStone. 7-2
Setting the Migration Destination 7-3
Migrating Objects 7-3
Things to Watch Out For 7-4
Instance Variable Mapping in Migration 7-4
7.2 Schema Coordination. 7-6
7.3 The Class Version Browser. 7-7
Menus in the Class Version Browser 7-7

Chapter 8. Using the GemStone Programming Tools

8.1 The GemStone Launcher 8-2
GemStone Menus 8-3
8.2 GemStone Browsers and Programming Tools. 8-4
The GemStone Workspace 8-5
GemStone Inspectors 8-5
The GemStone Classes Browser 8-7
Symbol List Pane. 8-9
Class Pane 8-12
Method Category Pane 8-14
Method Pane 8-15

Source Code Pane 8-16
GemStone Class, Dictionary, and Method List Browsers. 8-18
8.3 Creating Classes and Methods 8-20
What You Need to Know About GemStone Classes 8-20
Instance Variables Can Be Constrained 8-20
Constraints Are Inherited by Subclasses 8-20
Instances Can Be Made Invariant 8-21
Defining a New Class. 8-21
Private Instance Variables 8-24
Subclass Creation Methods. 8-24
Modifying an Existing Class 8-25
Defining Methods. 8-28
Saving Class and Method Definitions in Files 8-28
Handling Errors While Filing In 8-31
8.4 Using the GemStone Debugging Tools. 8-31
Step Points and Breakpoints 8-31
Breakpoints for Primitive and Special Methods 8-33
Debugging Commands in the GemStone Browser 8-35
The GemStone Debugger 8-38
The Stack Pane. 8-39
Source Code Pane in GemStone Debugger 8-40
Inspectors in the GemStone Debugger 8-41
The GemStone Breakpoint Browser. 8-42
Break Pane in Breakpoint Browser 8-42
Source Code Pane in the Breakpoint Browser 8-43
8.5 Interrupting GemStone Execution 8-44

Chapter 9. Performance Tuning

9.1 Selecting the Locus of Control 9-2
Locus of Execution 9-2
Relative Platform Speeds 9-3
Cost of Data Management 9-3
GemStone Optimization 9-3
Locus of Transaction Control 9-4
9.2 Profiling and Debugging 9-4
Profiling Client Smalltalk Execution 9-4
Watching Stub Activity. 9-4
Using Verbose Mode 9-5

9.3 Configuring GemBuilder 9-5
Configuration Parameters Available in GemBuilder 9-5
Using Configuration Parameters to Tune Your Application 9-7
The Settings Browser 9-11
Opening the Settings Browser 9-11
Parameter Notebook 9-13
9.4 Replication Tuning 9-14
Controlling the Level of Replication 9-14
Preventing Transient Stubs 9-15
Setting the TraversalBufferSize 9-15
9.5 Improving Performance by Local Caching 9-16
9.6 Optimizing Space Management 9-16
Explicit Stubbing 9-16
Using Forwarders 9-18
Not Caching Selected Objects 9-18
9.7 Managing Dirty Object Marking 9-19
9.8 Using Primitives 9-21
9.9 Changing the Initial Cache Size 9-21
9.10 Multi-threaded Applications 9-21
Thread-Safe Transparency Caches 9-22
Blocking and Nonblocking Protocol 9-22
One Thread Per Session 9-22
Multiple Threads per Session 9-22
Coordinating Transaction Boundaries 9-23
Coordinating Flushing 9-24
Coordinating Faulting 9-24
Using the VisualWorks Application Model 9-24

Chapter 10. Nontransparent Access to GemStone Objects

10.1 Nontransparency: General Principles 10-2
Flushing and Faulting Nontransparent Objects 10-2
Public vs. Private Classes and Methods 10-4
Specifying a Session 10-4
10.2 GbsObject Proxies 10-5
Sending Messages Through GbsObject Proxies 10-6
Special Treatment of Binary Selectors 10-8
Sending Code to Gemstone for Execution 10-8

Converting GSObjects to Replicates 10-10
10.3 Structural Access to GemStone Objects 10-11
10.4 Nontransparent Replication 10-13
10.5 Executing GemStone Host File Access Methods 10-14

Chapter 11. Exception Handling

11.1 Error Handling and Recovery 11-1
11.2 User-Defined Errors 11-3
11.3 GemBuilder's Smalltalk Emergency Handler 11-4

Appendix A. GemBuilder Classes and Proxies

A.1 Special GemBuilder Classes A-1
Class for Raising Errors A-1
Classes for Connecting Objects A-1
Class for Forwarding Messages A-2
Class for Providing Structural Access A-2
A.2 Reserved OOPs A-3

Appendix B. Network Resource String Syntax

B.1 Overview B-1
B.2 Defaults B-2
B.3 Notation B-3
B.4 Syntax B-4

Appendix C. ParcPlace Smalltalk and GemStone Smalltalk

List of Figures

Figure 1.1. The GemStone Object Server 1-2
Figure 2.1. RPC and Linked Gem Processes 2-3
Figure 2.2. The Session Browser Icon 2-7
Figure 2.3. The GemStone Session Browser 2-7
Figure 2.4. The Login Editor 2-8
Figure 2.5. Session Browser, One Session Defined 2-9
Figure 2.6. Session Browser, One Session Logged In 2-12
Figure 2.7. Committing with Approval From a Session Dependent 2-17
Figure 3.1. Connecting Application Roots 3-3
Figure 3.2. Connecting Application Roots 3-4
Figure 3.3. Creating a Name Connector 3-7
Figure 3.4. The Connector Browser Icon 3-10
Figure 3.5. The Connector Browser 3-11
Figure 3.6. Connector Class Hierarchy 3-15
Figure 4.1. Two-level Fault of an Object 4-6
Figure 4.2. Two-level Fault After Sending a Message to Stub 4-7
Figure 5.1. GemBuilder Application Workspace 5-3
Figure 6.1. GemStone's Object-Level Security Mechanism 6-5

Figure 6.2. The Segment Tool 6-11
Figure 6.3. The Symbol List Browser. 6-19
Figure 6.4. User Account Manager Icon 6-22
Figure 6.5. GemStone User List. 6-23
Figure 6.6. GemStone User Dialog 6-24
Figure 6.7. Privileges Dialog in GemStone User Window 6-25
Figure 7.1. The Class Version Browser 7-7
Figure 8.1. The GemStone Launcher 8-2
Figure 8.2. GemStone Workspace Icon 8-5
Figure 8.3. Inspecting a Nonsequenceable Collection in GemStone 8-6
Figure 8.4. GemStone Proxy Inspector. 8-7
Figure 8.5. Classes Browser Icon 8-8
Figure 8.6. The GemStone Classes Browser 8-8
Figure 8.7. Other GemStone Browsers. 8-19
Figure 8.8. Creating a New Version of a Class 8-27
Figure 8.9. Classes Browser with Highlighted Breakpoint 8-36
Figure 8.10. Breakpoint Notifier 8-36
Figure 8.11. The GemStone Debugger 8-38
Figure 8.12. Breakpoint Browser 8-42
Figure 9.1. The Settings Browser 9-12
Figure 9.2. Employee Set Faulted into the Client Smalltalk 9-17
Figure 10.1. Transparent Object 10-3
Figure 10.2. Nontransparent Objects. 10-3

List of Tables

Table 3.1. Connector Types 3-6
Table 3.2. Group List Menu in the Connector Browser 3-12
Table 3.3. Connectors Menu in the Connector Browser 3-12
Table 3.4. Options in the Connector Browser's Control Panel. 3-13
Table 3.5. Postconnect Action Options in the Connector Browser 3-13
Table 4.1. Instance Variable Replication Specifications 4-10
Table 5.1. GbsSession Methods for Running Outside of a Transaction. 5-7
Table 6.1. File Menu in the Segment Tool 6-14
Table 6.2. Segment Menu in the Segment Tool 6-14
Table 6.3. Group Menu in the Segment Tool 6-15
Table 6.4. Member Menu in the Segment Tool 6-15
Table 6.5. Report Menu in the Segment Tool 6-16
Table 6.6. File Menu in the Symbol List Browser 6-20
Table 6.7. Edit Menu in the Symbol List Browser. 6-21
Table 6.8. Object Menu in the Symbol List Browser 6-21
Table 6.9. GemStone User List: File Menu 6-23
Table 6.10. Buttons in the GemStone User Window 6-24
Table 6.11. Privileges. 6-25

Table 7.1. Class Menu in Class Version Browser 7-8
Table 8.1. GemStone Workspace Menu Commands 8-5
Table 8.2. GemStone Inspector Menu Commands 8-6
Table 8.3. Symbol List Menu in GemStone Browser 8-10
Table 8.4. Class Menu in GemStone Browser 8-13
Table 8.5. Method Category Menu in GemStone Browser. 8-14
Table 8.6. Method Menu in GemStone Browser 8-15
Table 8.7. Browser's Source Code Pane Menu Commands 8-17
Table 8.8. Breakpoint Notifier Commands 8-37
Table 8.9. Debugger's Stack Pane Menu Commands 8-39
Table 8.10. Debugger's Source Code Pane Menu Commands 8-40
Table 8.11. Breakpoint Browser Menu Commands 8-43
Table 9.1. Configuration Parameters for GemBuilder 9-5
Table 9.2. Notebook Control Buttons and Their Combo Box Menus. 9-13
Table 9.3. Parameter Page Control Buttons 9-14

Chapter

1 Basic Concepts

This chapter describes the overall design of a GemBuilder application and presents the fundamental concepts required to understand the interface between client Smalltalk and the GemStone object server.

The GemStone Object Server
introduces GemStone and its architecture and explains the part each component plays in the system.

GemBuilder for Smalltalk
outlines the basic features of GemBuilder that allow you to access GemStone objects from your Smalltalk application, and describes the basic programming functions that GemBuilder provides.

Designing a GemStone Application: an Overview
outlines the basic steps involved in producing a Gembuilder application.

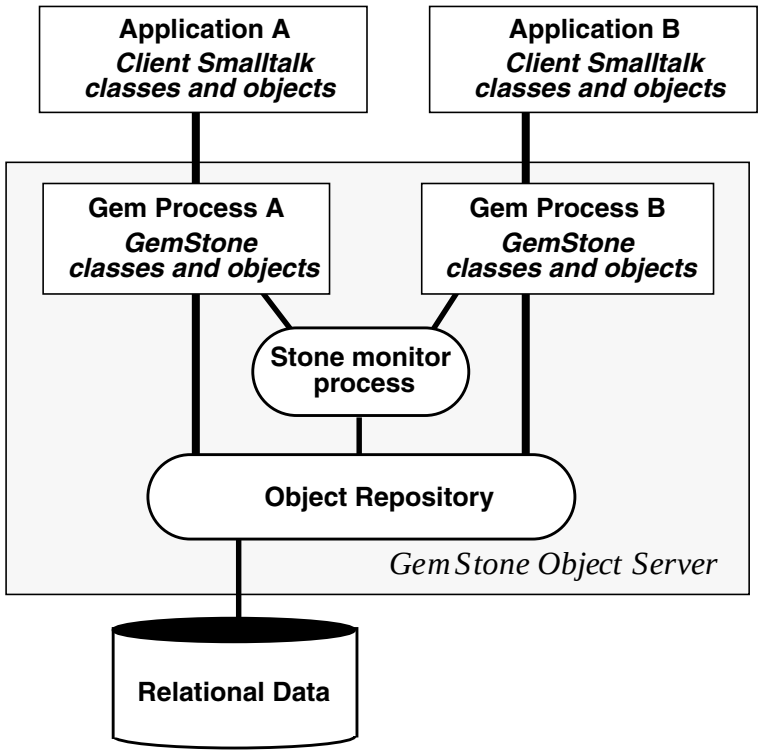
1.1 The GemStone Object Server

The GemStone object server supports multiple concurrent users of a large repository of objects. GemStone provides efficient storage and retrieval of large sets of objects, resiliency to hardware and software failures, protection for object integrity, and a rich set of security mechanisms.

The GemStone object server consists of three main components: a *repository* for storing persistent, shared objects; a monitor process called the *Stone*, and one or more user processes, called *Gems*.

Figure 1.1 shows how the object server supports clients in a Smalltalk application environment.

Figure 1.1 The GemStone Object Server



The *object repository* is a multiuser, disk-based Smalltalk image containing shared application objects and GemStone kernel classes. It is composed of files

(known to GemStone as *extents*) that can reside on a single machine or can be distributed among several networked hosts. The repository can also include GemConnect objects representing data stored in third-party relational databases.

Your Smalltalk application program treats the repository as a single unit, regardless of where its elements physically reside.

A *Gem* is an executable process that your application creates when it begins a GemStone session. A Gem acts as an object server for one session, providing a single-user view of the multiuser GemStone repository. A Gem reads objects from the repository, executes GemStone Smalltalk methods, and updates the repository.

Each Gem represents a single session. An application can create more than one session, each representing an internally-consistent single view of the repository. When a Gem *commits a transaction*, it modifies the shared repository and updates its own view of the repository.

The *Stone* monitor process handles locking and controls concurrent access to objects in the repository, ensuring integrity of the stored objects. Each repository is monitored by a single Stone.

Despite its central role in coordinating the work of all individual Gems, the Stone is surprisingly unintrusive. To optimize throughput for all users, most processing is handled by the Gems, which often interact directly with the repository. The Stone intervenes only when required to ensure the integrity of the multiuser repository.

1.2 GemBuilder for Smalltalk

GemBuilder for Smalltalk is a set of classes and primitives that can be installed in a Smalltalk image. With the functionality provided by GemBuilder, you can:

- store your client Smalltalk application objects in GemStone;
- import GemStone objects into Smalltalk as Smalltalk objects;
- allow your application objects to be transparently replicated and maintained both in Smalltalk and in GemStone, or allow some objects to reside only in GemStone but appear as Smalltalk objects;
- arrange for messages sent to client Smalltalk objects to be forwarded and executed in GemStone by corresponding GemStone objects;

- use GemStone's programming tools to develop GemStone classes and methods to operate on your application objects; and
- perform certain system functions, such as committing transactions and starting or ending GemStone sessions.

The Programming Interface

Your client Smalltalk application creates a GemStone session by using GemBuilder to log in to GemStone, creating a Gem process to serve your application. Many Gem processes can actively communicate with a single repository at the same time.

As Figure 1.1 illustrates, several applications can work concurrently with a single repository, with each application viewing the repository as its own. GemStone coordinates transactions between each of the applications and the repository.

Transparent Access to GemStone

The interface between your client Smalltalk application and GemStone can be relatively seamless.

Many of the classes in the base Smalltalk image are mapped to comparable GemStone classes, and additional class mappings can be created either automatically or explicitly. GemBuilder is also able to automatically generate GemStone classes from client Smalltalk classes, and vice versa, as necessary. Your Smalltalk objects can be replicated in GemStone, and GemStone objects can be replicated in Smalltalk.

The most common way to make a Smalltalk object persistent—that is, to store it in GemStone—is to define a *connector* for the object. A *connector* is an object that knows how to resolve a client Smalltalk object and a GemStone object and how to establish a relationship between them when a session logs into GemStone. Once you've defined a connector for the two objects, the GemBuilder interface maintains the relationship between the shared GemStone object and the private client Smalltalk object, updating values from the repository to your application and vice versa, as necessary. The connector ensures that if a shared GemStone object is modified, the application's Smalltalk counterpart is updated automatically.

Your client Smalltalk application updates shared objects in the repository by sending a `commit` message to a session. With a successful commit, changes to objects in the current session are propagated to the shared object repository in GemStone. Once you have committed a transaction, your application objects are updated with the most recently saved state of the repository, incorporating changes made by other users.

While, for the most part, GemBuilder will automatically manage objects in both the client Smalltalk and in GemStone, you can exert as much control as you want over how objects are stored and used. GemBuilder provides tools that let you specify customized policies for translating between your client Smalltalk and GemStone objects.

Chapter 3 describes GemBuilder's mechanisms for making your client Smalltalk objects persistent, and Chapter 9 explains how to tune the system to minimize maintenance overhead and optimize performance.

GemStone's Smalltalk Language

GemStone provides a version of Smalltalk that supports multiple concurrent users of the shared object repository through such features as session management, reduced-conflict collection classes, querying, transaction management, and persistence.

GemStone Smalltalk is like single-user client Smalltalk in both its organization and syntax. Objects are defined by classes based on common structure and protocol and classes are organized into an *is-a* hierarchy, rooted at class Object. The class hierarchy is extensible; new classes can be added as required to model an application. The behavior of common classes conforms to the ANSI draft standard for Smalltalk. GemStone's class hierarchy is discussed in the introductory chapter to the *GemStone Programming Guide*.

The most significant difference between GemStone Smalltalk and Smalltalk lies in GemStone's support for a multiuser environment in which persistent objects can be shared among many users.

As an object server, GemStone must address the same key issues as conventional information storage systems that support multiple concurrent users. For this reason, GemStone's Smalltalk includes classes for:

- managing concurrent access to information,
- protecting information from unauthorized access, and
- keeping stored information secure and restoring it in the event of a failure.

You should be aware of a few differences between GemStone Smalltalk and client Smalltalk in syntax and in the behavior of some of the classes. A summary of these differences can be found in Appendix C.

The GemBuilder Tools

GemBuilder's programming environment provides tools for programming in GemStone Smalltalk. The following tools are described in detail in subsequent chapters of this manual:

- A *GemStone System Browser* lets you examine, modify, and create GemStone classes and methods.
- A *GemStone System Workspace* provides easy access to commonly used GemStone Smalltalk code.
- *GemStone Inspectors* let you examine and modify the state of GemStone objects.
- A *GemStone Breakpoint Browser* and a *Debugger* let you examine and dynamically modify the state of a running GemStone application.
- A *Session Browser* allows you to manage sessions and transactions.
- A *Connector Browser* allows you to manage the connectors that establish relationships between Smalltalk and GemStone objects.
- A *Class Version Browser* can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.
- A *Symbol List Browser* allows you to examine the GemStone Symbol Lists associated with UserProfiles, add and delete dictionaries from these lists, and manipulate the entries in those dictionaries.
- A *Settings Browser* allows you to examine and set the configuration parameters for GemBuilder.
- A *User Account Management Tool* allows you to create new user accounts, change account passwords, and assign group membership.
- A *Segment Tool* facilitates managing GemStone authorization at the object level by controlling how objects are assigned to segments.

1.3 Designing a GemStone Application: an Overview

Many GemStone users start with an application they have already written in Smalltalk. Their mission is to transform that application into one that makes meaningful use of GemStone's features: persistence, multiuser access, security, integrity, and the ability to store and manage large quantities of information.

As a GemStone programmer, your application design and porting efforts involve the following tasks:

- choosing the objects that should be stored and shared,
- deciding which objects need to be secured,
- establishing connections between root objects in the client and the server,
- deciding when to commit transactions and how to handle concurrency control, and
- tuning your application for optimal performance.

This section gives you an overview of these steps and points you to the chapters that discuss these topics in detail.

Which objects should be stored and shared?

Your application will typically have two kinds of objects: those that must persist across images and GemStone sessions and be shared among users, and those that represent a transient state. Your first task is to identify the objects that make up your application and decide which ones need to be stored and shared. Making objects persistent unnecessarily can degrade performance and complicate programming.

Use GemStone to store those objects that need to exist between sessions and must be shared with other users. For example, objects representing information in your application such as financial statements, employee health records, or library book cards would certainly require storage in GemStone. Some methods for manipulating the persistent data can also be usefully coded in GemStone Smalltalk and stored in GemStone for improved efficiency.

You don't need to store in GemStone transient session objects that no one else will ever need; such objects can remain in Smalltalk. For example, suppose you generate a report from the financial statements stored in GemStone. Once you view or print the report it has served its purpose; the next time you need a report you will generate a new one. The report and its component objects can exist simply as Smalltalk objects; they don't need to be stored in GemStone. However,

you might want the classes and methods used to build the report to be stored in GemStone so that others can use them.

Certain objects can be considered your organization's *business objects*. Business objects contain the data that give your organization its strategic, competitive advantage; their instance variables contain information about the business process that they model, and their methods represent actions involved in conducting business. Keeping your business objects centralized and stored separately from the applications that access them allows your organization to serve the needs of all users, while still enforcing consistency and maintaining control of critical information.

Which objects should be secured?

What security challenges does the application pose? Determine the strategy you will use to handle those challenges. Does access to certain objects need to be restricted to only certain authorized users? Many of your business objects may fall into this category. If so, who should be authorized to access them, and how? Do your users fall into groups with different access needs? Will anyone need to execute privileged methods? The earlier you lay the groundwork for your security needs, the easier they will be to implement. Security is discussed in detail in Chapter 6.

Which objects should be connected?

Once you've decided how to partition your application objects, you will want to set up connections between the objects that will reside on the client and those that will reside on the server so that GemBuilder can automatically manage changes to them and understand how to update them properly. This connection is established by making sure a connector is defined for those objects.

A connector connects not only the immediate object but also all those objects that it references, so you don't need to define a connector for every object in your application that you want to store in GemStone. Instead, you should begin by identifying the subsystems in your application that define persistent objects, and then identifying a root object in each subsystem to target with a connector. You can find further discussion of connectors in Chapter 3.

How should transactions be handled?

Another decision you need to make involves transactions: when to commit and how to handle the occasional failure to commit. Do you want to use locks to ensure a successful commit? If so, identify the places in your application where you must acquire the locks. (Concurrency control and locking are discussed in more detail in Chapter 5.)

How can performance be improved?

If— after you have built your application— you find that its performance does not meet your expectations, you have a variety of ways to improve matters.

One of the most powerful single things you can do to improve performance is to move some of the behavior from Smalltalk into GemStone and let the GemStone execution engine do the work. This approach reduces network traffic, which is a prime cause of slow performance.

Which methods might best be executed in GemStone? GemStone already contains behavior for many of the common Smalltalk kernel classes and, as mentioned earlier, the syntax and class hierarchy of GemStone's Smalltalk language are so similar to those of Smalltalk that the porting effort is likely to be relatively simple. Performance issues in general are discussed in Chapter 9. Moving execution into GemStone is discussed in the section entitled "Locus of Execution" on page 9-2.

That chapter also discusses the configuration parameters that can be altered to improve GemBuilder's performance. GemBuilder's configuration parameters are described in the section called "Configuring GemBuilder" on page 9-5. Chapter 9 also explains how to use GemBuilder's Settings Browser to tune your system for maximum performance, given the details of your application and environment.

Finally, you can configure the GemStone object server for maximum performance, given the details of your application and environment. GemStone configuration parameters are discussed in detail in the *GemStone System Administration Guide*; in addition, the *GemStone Programming Guide* gives a variety of tips in the chapter entitled on performance.

Chapter 2

Communicating with the GemStone Object Server

When you install GemBuilder, your Smalltalk image becomes "GemStone-enabled," meaning that your image is equipped with additional classes and methods that allow it to work with shared, persistent objects through a multi-user GemStone object server. Your Smalltalk image remains a single-user application, however, until you connect to the object server. To do so, your application must log in to a GemStone object server in much the same way that you log in to a user account in order to work on a networked computer system.

This chapter explains how to communicate with the GemStone object server by initiating and managing GemStone sessions.

GemStone Sessions
introduces sessions and explains the difference between RPC and linked sessions.

Session Control in GemBuilder
explains how to use the classes GbsSession, GbsSessionManager, and GbsSessionParameters to manage GemBuilder sessions.

The GemStone Session Browser
describes the GemStone Session Browser.

Logging In To and Logging Out Of GemStone
how to log in and out of GemStone sessions programmatically and by using the Session Browser.

Session Dependents
explains how to use the Smalltalk dependency mechanism to coordinate the effects of session management actions on multiple application components.

2.1 GemStone Sessions

An application connects to the GemStone object server by logging in to GemStone and disconnects by logging out. Each logged-in connection is known as a *session* and is supported by one Gem process. The Gem reads objects from the repository, executes GemStone Smalltalk methods, and propagates changes from the application to the repository.

Each session presents a single-user view of a multiuser GemStone repository. An application can create multiple sessions, one of which is the *current session* at any given time. You can manage GemStone sessions either through your application code or through the Session Browser.

RPC and Linked Sessions

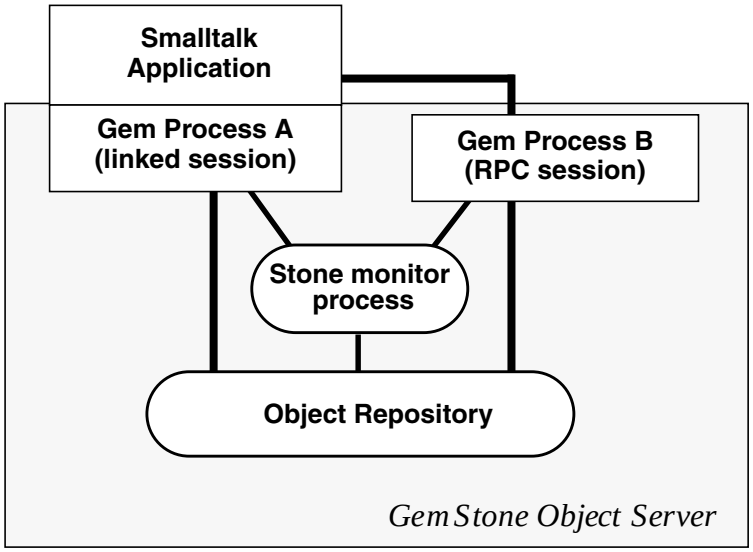
A Gem can run as a separate process and respond to Remote Procedure Calls (RPCs) from its application, in which case the session it supports is called an *RPC* session.

On platforms that host the GemStone object server and its runtime libraries, one Gem can be integrated with the application into a single process. That gem is called a *linked* session. A linked session provides significantly faster performance than does an RPC session. With the linked version, however, an application and its Gem must run on the same machine and the runtime code requires additional memory.

An RPC session, though less efficient, offers more flexibility because the application and its Gem are separate processes that can run on different hosts in a network. Any GemBuilder application can create RPC sessions. Where a linked session is supported, an application can create multiple sessions, but only one can be linked. (To suppress linked sessions, forcing all Gems to run as RPC processes, you can set the configuration parameter **loginLinkedIfAvailable** to **false**.)

Figure 2.1 shows an application with two logged-in sessions. Gem process A supports a linked session, while Gem process B supports an RPC session.

Figure 2.1 RPC and Linked Gem Processes



2.2 Session Control in GemBuilder

Managing GemStone sessions involves many of the same activities required to manage user sessions on a multi-user computer network. To manage GemStone sessions you need to:

- Identify the object server to which you wish to connect
- Identify the user account to which you wish to connect
- Log in and log out
- List active sessions
- Designate a current session
- Send messages to specific sessions

Three GemBuilder classes provide these session control capabilities: GbsSession, GbsSessionParameters, and GbsSessionManager.

GbsSession

An instance of GbsSession represents a GemStone session connection. A

successful login returns a new instance of GbsSession. You can send messages to an active GbsSession to execute GemStone code, control GemStone transactions, compile GemStone methods, and perform low-level structured access to groups of objects.

GbsSessionParameters
Instances of GbsSessionParameters store information needed to log in to GemStone. This information includes the Stone name, your user name, passwords, and the set of connectors to be connected at login.

GbsSessionManager
There is a single instance of GbsSessionManager, named GBSM. Its job is to manage all known GbsSessions, support the notion of a current session (explained in the following section), and handle other miscellaneous GemBuilder matters. Whenever a new GbsSession is created, it is registered with GBSM. GBSM shuts down any GemStone connections before Smalltalk quits.

Defining Session Parameters

To initiate a GemStone session, you must first identify the object server and user account to which you wish to connect. This information is stored in an instance of GbsSessionParameters and added to a list maintained by GBSM. You can provide the information through window-based tools or programmatically. Both methods are described in later sections. In either case, you must supply these items:

- **The name of the GemStone monitor**
  For a Stone running on a remote server, be sure to include the server's hostname in Network Resource String (NRS) format. For instance, for a Stone named "gemserver50" on a node named "mozart", specify an NRS string of the form:

  ```
  !@mozart!gemserver50
  ```

  (Appendix B describes NRS syntax in detail.)

- **GemStone user name and GemStone password**
  This user name and password combination must already have been defined in GemStone by your GemStone data curator or system administrator. (GemBuilder provides a set of tools for managing user accounts—see "User Account Management Tools" on page 6-22.) Because GemStone comes equipped with a data curator account, we show the DataCurator user name in many of our examples.

- **Host username and Host password** (not required for a linked session)
  This user name and password combination specifies a valid login on the Gem's host machine (the network node specified in the Gem service name, described below). Do not confuse these values with your GemStone username and password. You do not need to supply a host user name and host password if you are starting a linked Gem process. However, an application that must control more than one GemStone session can use a linked interface for only one session. Other sessions must use the RPC interface.
- **Gem service** (not required for a linked session)
  The name of the Gem service on the host computer (that is, the Gem process to which your GemBuilder session will be connected). For most installations, the Gem service name is `gemnetobjcsh` (if you use the C shell) or `gemnetobject` (if your host login shell is the Bourne shell). Both service names are accepted on Windows NT installations.

  You can specify that the gem is to run on a remote node by using an NRS for the Gem service name For example:

  ```
  !@mozart!gemnetobjcsh
  ```

  You do not need to supply a Gem Service name if you are starting a linked Gem process.

Once defined, an instance of GbsSessionParameters can be used for more than one session. Thus, a session description that includes the RPC-required parameters can be used for both linked and RPC logins.

Defining Session Parameters Programmatically

The instance creation method for a full set of RPC parameters is:

```
GbsSessionParameters newWithGemStoneName:    aGemStoneName
 username:   aUsername
 password:   aPassword
 hostUsername:   aHostUsername
 hostPassword:   aHostPassword
 gemService:   aGemServiceName
```

For a shorter set of parameters that supports only linked logins, you can use a shorter creation method:

```
GbsSessionParameters newWithGemStoneName:    aGemStoneName
 username:   aUsername
 password:   aPassword
```

Storing Session Parameters for Later Use

If you want the GemBuilder session manager to retain a copy of your newly-created session description for future use, you must register it with GBSM:

```
GBSM addParameters:  aGbsSessionParameters
```

Once registered with GBSM and saved with the image, the parameters are available for use in future invocations of the image. In addition, the Session Browser and other login prompters make use of GBSM's list of session parameters.

Executing the expression `GBSM knownParameters` returns an array of all GbsSessionParameters instances registered with GBSM.

To delete a registered session parameters object, send `removeParameters:` to GBSM:

```
GBSM removeParameters:  aGbsSessionParameters
```

Password Security

You can control the degree of security that GemBuilder applies to the passwords in a session parameters object. For example, when you create the parameters object, you can specify the passwords as empty strings. When the parameters object is subsequently used in a login message, GemBuilder will prompt the user for the passwords.

For example:

```
mySessionParameters := GbsSessionParameters
   newWithGemStoneName: '!@mozart!gemserver50'
   username:      'DataCurator'
   password:      ''
   hostUsername:    'daveb'
   hostPassword:    ''
   gemService:     '!@mozart!gemnetobjcsh'
```

If convenience is more important than security, you can fill in the passwords and then instruct the parameters object to retain the password information for future use:

```
mySessionParameters rememberPassword: true;
          rememberHostPassword: true
```

The default "remember" setting for both passwords is `false`, which causes the stored passwords to be erased after login.

2.3 The GemStone Session Browser

The GemStone Session Browser streamlines logging in and logging out of GemStone and managing sessions and transactions. This section explains how to invoke the Session Browser, and how to use it to define session parameters and to log in and out of GemStone.

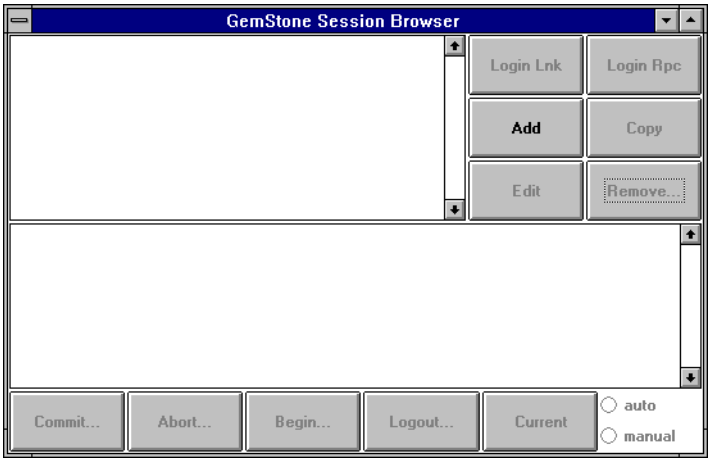
Starting the Session Browser

1. Start Smalltalk with GemBuilder installed.
2. Select **Tools** > **Session Browser** from the GemStone Launcher to open a Session Browser, or click on the Session Browser icon (Figure 2.2).

Figure 2.2 The Session Browser Icon

Figure 2.3 shows the Session Browser.

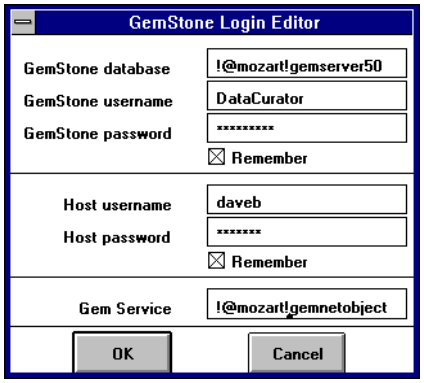
Figure 2.3 The GemStone Session Browser



Supplying Session Parameters

Select the **Add** button to define a set of session parameters. A Login Editor appears, as shown in Figure 2.4.

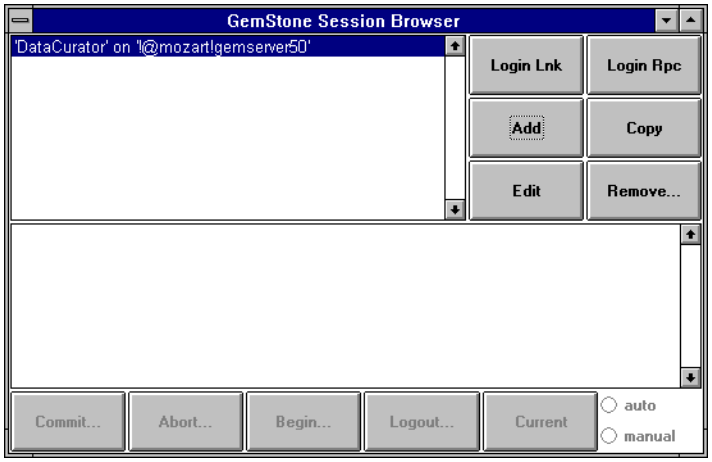
Figure 2.4 The Login Editor



Use the **Tab** key or the mouse to move through the fields in the login dialog, and the **Return** key to accept input or changes in the login dialog. Provide the session parameters described previously (see "Defining Session Parameters" on page 2-4). For maximum password security, leave the **Password** and **Host Password** fields empty, and the **Remember** boxes unselected.

When you click on **OK**, GemBuilder creates an instance of GbsSessionParameters and registers it with GBSM. The new session description is added to the Session Browser (Figure 2.5).

Figure 2.5 Session Browser, One Session Defined



To change a session parameters object, select the name of the parameters object in the upper left pane of the Session Browser and use the browser's **Edit** button to open a Login Editor. Use the Login Editor to change existing session parameters; clicking on **OK** causes your changes to take effect.

Removing Session Parameters

To remove a GemStone session parameters object from the Session Browser, select the session parameters defining the session in the upper left pane of the Session Browser and click on **Remove**.

2.4 Logging In To and Logging Out Of GemStone

Before you can start a GemStone session, you need to have a Stone process and, for an RPC session, a NetLDI (network long distance interface) process running. See your System Administrator if the transcript indicates that these processes aren't active.

Depending on your version of GemStone and the terms of your GemStone license, you can have many sessions logged in at once through your GemStone Smalltalk Interface. These sessions can all be attached to the same GemStone repository, or they can be attached to different repositories.

Logging In To GemStone Programmatically

The protocol for logging in is understood both by GBSM and by instances of GbsSessionParameters. To log in using a specific session parameters object, send a `login` message to the parameters object itself:

aGbsSessionParameters `login`

To start multiple sessions with the same parameters, simply repeat these login messages.

An application can also send a generic login message to GBSM:

`GBSM login`

This message invokes an interactive utility that allows you to select among known GbsSessionParameters or to create a new session parameters object using the Login Editor.

A successful login returns a unique instance of GbsSession. (An unsuccessful login attempt returns `nil`.) Each instance of GbsSession maintains a reference to that session's parameters, which you can retrieve by sending:

aGbsSession `parameters`

GBSM maintains a collection of currently logged in GbsSessions. You can determine if any sessions are logged in with `GBSM isLoggedIn` and you can execute `GBSM loggedInSessions` to return an array of currently logged in GbsSessions.

The Current Session

When a new GbsSession is created, it is registered with GBSM, which maintains a variable that represents the *current* session. When a session logs in, it becomes the current session. If you execute code in a GemStone tool, the code is evaluated in the session that was current when you opened that tool. If you send a message to GBSM that is intended for a session, the message is forwarded to the current session.

Sending the message `GBSM currentSession` returns the current GbsSession. You can change the current session in a workspace by executing an expression of the following form:

`GBSM currentSession:` *aGbsSession .*

You can also send a message directly to a logged-in GbsSession even when it is not the current session. If you send a specific session a message executing code, that

code is evaluated in the receiving session, regardless of whether it is the current session.

Your application can make another session the current session by executing code like that shown in Example 2.1:

Example 2.1

```
|s1 s2|
 s1 := GBSM login.
 s2 := GBSM login.
GBSM currentSession: s1  .       "Make s1 current"
 .
  .                               "Do some work"
 .
GBSM currentSession: s2  .       "Make s2 current"
```

Each GemStone browser, inspector, debugger, and breakpoint browser is attached to the instance of GbsSession that was the current session when it opened. For example, you can have two browsers open in two different sessions, such that operations performed in each browser are applied only to the session to which that browser is attached.

Workspaces, however, are not session-specific. Code executed in a workspace defaults to the current session, unless another session is specified.

Logging Out of GemStone Programmatically

To instruct a session to log itself out, send logout to the session object:

aGbsSession `logout`

Or, you can execute the more generic instruction:

`GBSM logout`

This message prompts you with a list of currently logged-in sessions from which to choose.

Before logging out, GemBuilder prompts you to commit your changes. If you log out after performing work and do not commit it to the permanent repository, the uncommitted work you have done will be lost.

If you have been working in several sessions, be sure to commit only those sessions whose changes you wish to save.

Session Management Using the Session Browser

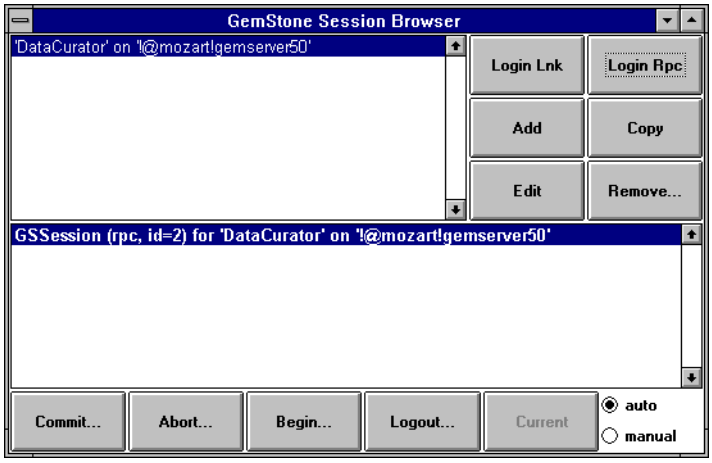
You can use the Session Browser to perform the same session management tasks that you can perform programmatically: log in to GemStone, view current sessions, set the current session, and log out of GemStone.

Logging In

To log into GemStone with the Session Browser, select the name of the session parameters object in the upper left pane, and click on either **Login Lnk** or **Login Rpc.**

When you are logged in, the Session Browser displays the session description in its lower pane (Figure 2.6).

Figure 2.6 Session Browser, One Session Logged In



If your login is not successful, make sure you entered the correct parameters and that the necessary underlying processes are running.

Setting the Current Session

The Session Browser's upper pane shows all of the known parameters that are registered with GBSM. The lower pane shows all sessions currently logged in.

To change the current session, select a logged-in session in the lower pane and click **Current**.

Logging Out of a GemStone Session With the Session Browser

To log out of GemStone from the Session Browser, select the session in the browser's lower pane and click on **Logout** in the row of buttons at the bottom of the browser.

Before logging out, GemBuilder prompts you to commit your changes. If you log out after performing work and do not commit it to the permanent repository, the uncommitted work you have done will be lost.

If you have been working in several sessions, be sure to commit only those sessions whose changes you wish to save.

2.5 Session Dependents

An application can create several related components during a single GemBuilder session. When one of the components commits, aborts, or logs out, the other components can be affected and so may need to coordinate their responses with each other. In the GemBuilder environment, for example, you can commit by selecting a button in the Session Browser. But before the commit takes place, all other session-dependent components are notified that a commit is about to occur. So a related application component, such as an open browser containing modified text, prompts you for permission to discard its changes before allowing the commit to proceed.

Through the Smalltalk dependency mechanism, any object can be registered as a dependent of a session. In practice, a session dependent is often a user-visible application component, such as a browser or a workspace. When one application component asks to abort, commit, or log out, the session asks all of its registered dependents to approve before it performs the operation. If any registered dependent vetos the operation, the operation is not performed and the method (`commitTransaction`, `abortTransaction`, etc.) returns `nil`.

To make an object a dependent of a GbsSession, send:

mySession `addDependent:` *myObj*

To remove an object from the list of dependents, send the following message:

mySession `removeDependent:` *myObj*

So, for example, a browser object might include code similar to Example 2.2 in its initialization method:

Example 2.2

```
| mySession |
mySession := self session.
"Add this browser to the sessions dependents list"
(session dependents includes: self)
      ifFalse: [session addDependent: self]
...
```

When a session receives a commit, abort, or logout request, it sends an `updateRequest:` message to each of its dependents, with an argument describing the nature of the request. Each registered object should be prepared to receive the `updateRequest:` message with any one of the following aspect symbols as its argument:

#queryCommit
The session with which this object is registered has received a request to commit. Return `true` to allow the commit to take place or `false` to prevent it.

#queryAbort
The session with which this object is registered has received a request to abort. Return `true` to allow the abort to take place or `false` to prevent it.

#queryEndSession
The session with which this object is registered has received a request to terminate the session. Return `true` to allow the logout to take place or false to prevent it.

Example 2.3 shows how a session dependent might implement an `updateRequest:` method.

Example 2.3

```
updateRequest: aspect

"The session I am attached to wants to do something.
 Return a boolean granting or denying the request."

(#(queryAbort queryCommit queryEndSession)
 includes: aspect)
  ifTrue: [
       "My session wants to commit or abort.
        OK unless user doesn't want to."
  ^self askUserForPermission ].

"Let any other action occur."
^true
```

After the action is performed, the session sends `self changed:` with a parameter indicating the type of action performed. This causes the session to send an `update:` message to each of the registered dependents with one of the following aspect symbols:

#committed
All registered objects have approved the request to commit, and the transaction has been successfully committed.

#aborted
All registered objects have approved the request to abort, and the transaction has been aborted.

#sessionTerminated
The request to log out has been approved and the session has logged out.

Each registered dependent should be prepared to receive an `update:` message with one of the above aspect symbols as its argument. Example 2.4 shows how a session dependent might implement an `update:` method.

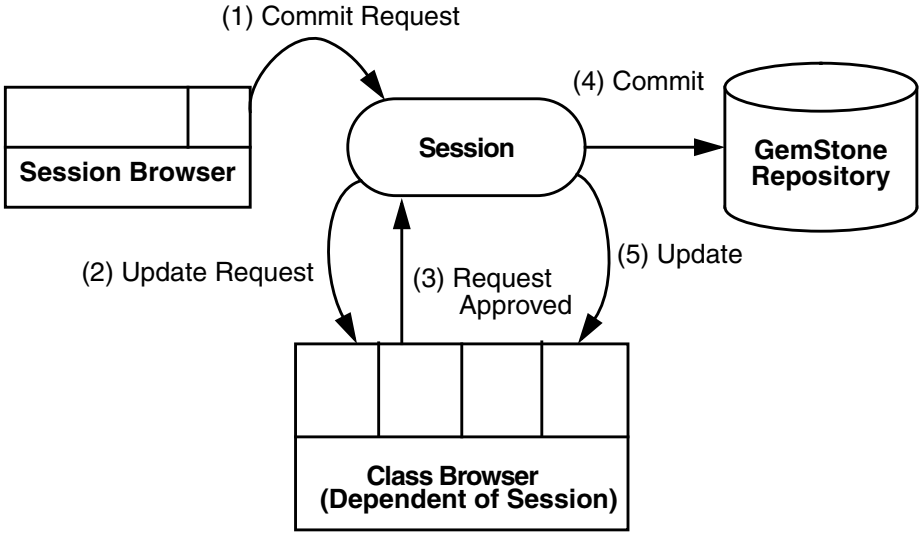
Example 2.4

```
update: aSymbol
"The session I am attached to just did something.
 I might need to respond."

(aSymbol = #sessionTerminated) ifTrue: [
"The session this tool is attached to has logged out
 - close ourself."
self builder notNil ifTrue:
      [self closeWindow]]
```

Figure 2.7 summarizes the sequence of events that occurs when a session queries a dependent before committing. In the figure, the Session Browser sends a commit request (`commitTransaction`) to a session (1). The session sends `updateRequest: #queryCommit` to each of its dependents (2). If every dependent approves (returns **true**), the commit proceeds (4). Following a successful commit, the session notifies its dependents that the action has occurred by sending `update: #commited` to each (5).

Figure 2.7 Committing with Approval From a Session Dependent



Chapter

3 Sharing Objects

This chapter explains how to set up connections between your application and GemStone that make your application's objects persistent and sharable and that allow your application to manipulate objects in the GemStone object server's shared object repository.

Deciding Which Objects to Connect
describes how to decide which objects to share.

Connectors
describes the kinds of connectors available.

The Connector Browser
explains how to use the Connector Browser to create and modify connectors.

Managing Connectors Programmatically
explains how to create and modify connectors using Smalltalk code.

3.1 Deciding Which Objects to Connect

Your Smalltalk image probably contains several subsystems of objects. Some of these objects are useful only in the client Smalltalk context of your image; others represent data that should be shared with other users through the GemStone object server. The first step in integrating GemStone into your application is to identify the application objects to be stored in GemStone.

GemBuilder defines *connectors* (instances of GbsConnector and its subclasses) that establish relationships between your client Smalltalk objects and the shared objects stored in GemStone.

Root Objects

A connector does more than simply connect one Smalltalk object to one GemStone object. It also connects all other objects to which the two connected objects refer, such as their instance variables. And because their instance variables are connected, *their* instance variables are also connected, and so on, until you reach objects that refer to no others, or atomic objects such as characters, integers, strings, booleans, or `nil`. The entire network of related objects forms a pair of tree structures whose roots are the two objects originally connected and whose leaves are the final objects reached—those objects that refer to no other objects.

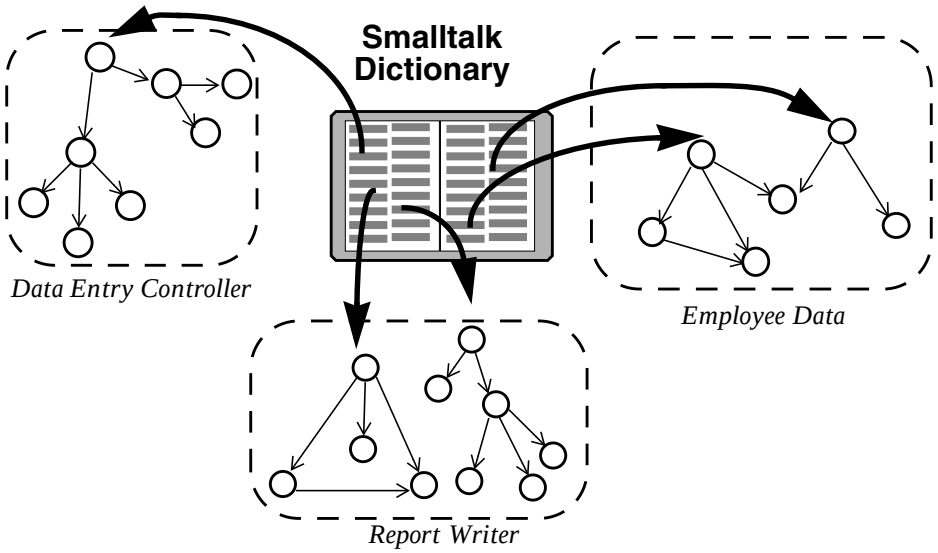
Because of this hierarchical structure, you need not define a connector for every shared object in your application; you can instead identify the subsystems in your application that define shared, persistent objects, and then identify a root object in each subsystem. Creating a connector for the root object makes the entire subsystem storable in GemStone.

If your application never modifies any persistent objects, but only reads them (for example, to generate reports), then you have finished your task simply by defining connectors between a few of your application's root objects and the GemStone objects they read. However, if your application modifies shared objects or creates new ones, then you must ensure that your modifications are propagated to and from the repository.

When a persistent object changes in the repository, GemBuilder (because it was designed as a multi-user object server) automatically updates the corresponding client Smalltalk object. However, you must arrange for modifications you make in Smalltalk (essentially a single-user development environment) to be propagated to GemStone. To manage this task, GemBuilder provides several mechanisms that can be customized to suit the specifics of your application as well as to enhance performance.

Figure 3.1 shows an application in which several connected objects are accessed through the Smalltalk dictionary. One system represents an employee database. Another system represents a data entry application for creating and modifying objects. A third system represents a report writer for these objects. Dotted lines in the figure group the logically related subsystems.

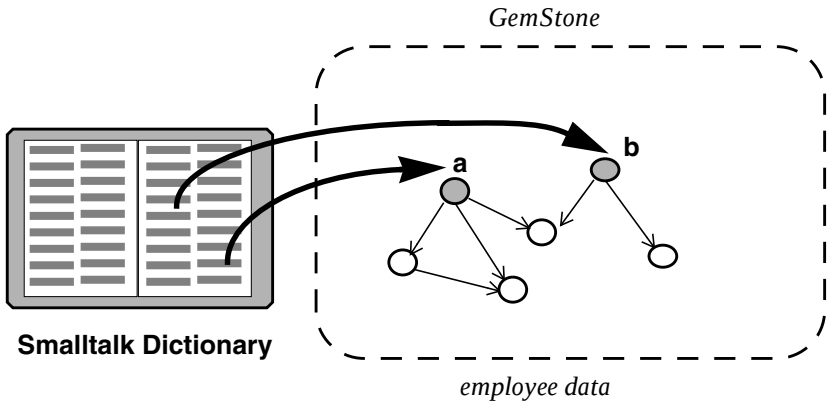
Figure 3.1 Connecting Application Roots



The data entry application and the report writer would reside in Smalltalk; however, the employee database should probably be stored in GemStone, as it defines persistent data that other users may need to share, data that would benefit from the stability and protections provided by GemStone. Also, because the employee database could be quite large, it could benefit from GemStone's large capacity and accelerated search capabilities.

Figure 3.2 shows the state of the employee data when stored in GemStone. (Later we will discuss how the GemStone objects are created from client Smalltalk objects.)

Figure 3.2 Connecting Application Roots



In Figure 3.2, objects **a** and **b** are *root* objects.

The root objects of an application are the persistent objects from which all other persistent objects can be reached by *transitive closure*; that is, either by direct reference or indirectly through any number of layers of references.

The most common kinds of root objects are:

- global variables in the Smalltalk dictionary,
- class variables, and
- class instance variables.

Once you have defined connectors for the application's roots, GemBuilder automatically manages all the objects referenced from these roots.

3.2 Connectors

A *connector* is an object that defines how to resolve a client Smalltalk object and a GemStone object. A connector establishes the relationship between its objects when a session logs in to GemStone.

Session Connectors and Global Connectors

A connector defined to connect two objects only when a specific session is logged in is called a *session connector*. A connector defined to connect two objects whenever any session is logged in is called a *global connector*. Session connectors allow individual applications to customize the connection list by using separate session parameters for each application, while global connectors allow a standard set of connectors to be maintained common to all applications in the client Smalltalk image.

Session connectors offer a degree of encapsulation that makes them the safer, more flexible choice for most uses. You can create session connectors for any session whose parameters you have defined. The objects will be connected only when that specific session is logged in; when other sessions are logged in, the connectors are not connected.

Two connectors are considered equal if they resolve to the same client Smalltalk object. Connectors are saved in client Smalltalk sets. Because Smalltalk sets eliminate duplicates based on equality, adding a global or session connector that resolves to the same object as an existing connector removes the existing connector. (The global connectors and each session's connectors are stored in separate sets, so duplicate session connectors are not removed if they are stored in different sessions.)

Kinds of Connectors

GemBuilder provides five types of connectors to resolve client Smalltalk and GemStone objects. These connectors use different mechanisms to verify aspects of the connection and to set the initial states of the connected objects. You can create connectors with the Connector Browser or in your application's initialization code.

Table 3.1 shows the types of connectors that are available.

Table 3.1 Connector Types

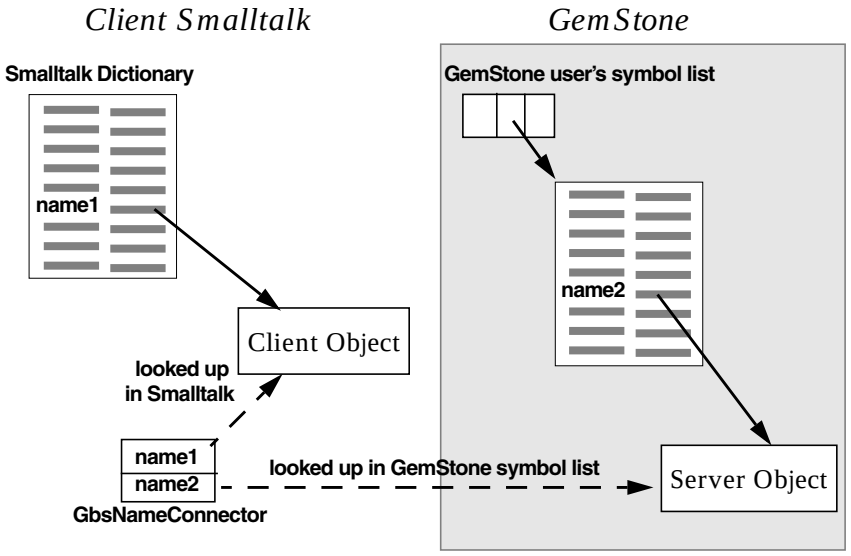
Type of connector	Function
Name connector	Connects client Smalltalk and GemStone objects identified by name.
Class connector	Connects (by name) a client Smalltalk and a GemStone class.
Class variable connector	Connects (by name) a client Smalltalk class variable to a GemStone class variable.
Class instance variable connector	Connects (by name) a client Smalltalk class instance variable to a GemStone class instance variable.
Fast connector	Bypasses name lookup in the interest of speed and connects client Smalltalk and GemStone objects using direct references. (Use with caution!)

Connecting by Name

Name connectors connect objects by their names. In the client Smalltalk application, objects included in the Smalltalk dictionary are visible throughout the single name space of the Smalltalk image. In GemStone, objects are "named" by being included in a dictionary in the symbol list of the user who owns the session.

Figure 3.3 illustrates how a name connector connects a Smalltalk object (Client Object) to a GemStone object (Server Object). The name is looked up when the connection is established—that is, when the session first logs in.

Figure 3.3 Creating a Name Connector



Connecting by Classes

A class connector resolves names in much the same way as a name connector, except that the connected objects must be classes.

Class connectors are created automatically by GemBuilder whenever automatic class generation occurs. (Automatic class generation is discussed in "Class Mapping Between GemStone and Smalltalk" on page 4-17.) The automatic creation of these connectors can be disabled by setting **generateClassConnectors** to **false** in the Settings Browser.

Class connectors should not be initialized in the same way as other connectors—in nearly all cases, they should specify a postconnect action of "none". See "Initializing Class Connectors: Special Considerations" on page 3-9 for details.

Connecting by Class Variable Names

A class variable connector initially resolves the named objects representing the classes, then looks for a class variable in each class with the specified name and connects those objects.

Connecting by Class Instance Variable Names

A class instance variable connector initially resolves the named objects representing the classes, then looks for a class instance variable in each class with the specified name and connects those objects.

Connecting by Identity: Fast Connectors

Name lookup in both client Smalltalk and GemStone Smalltalk can be slow if you are using a lot of connectors. You can bypass the name lookup by using a fast connector, which saves direct references to the client Smalltalk objects and the object IDs of the GemStone objects that are connected.

Using fast connectors can be risky, however. If the GemStone object is renamed or redefined, a fast connector will continue to point to the old object: the one with the same object identifier. When the identity of an object changes (for example, if it is a variable that you assign to a new object), a fast connector becomes incorrect. An out-of-date fast connector may cause an "object does not exist" error, or it may silently continue to pass messages to an old object.

Because using object identity is not always an appropriate way to resolve an object, we recommend that you generally use standard connectors instead of fast connectors, especially during early development stages. You can always use the Connector Browser to change a connector type later, when you are certain that your application can rely on named objects to have a constant identity.

Verification of Connections

GemBuilder provides a configuration parameter, **`connectVerification`**, that, when set to **`true`**, causes connectors to verify at login that they are not redefining a connector that already exists. When `connectVerification` is enabled, class connectors verify that the two classes they are connecting have compatible structures. When a connector fails verification, GemBuilder issues a notifier (if the **verbose** configuration parameter is **true**) or raises and exception (if **verbose** is **false**).

This parameter can be set in the Connector Browser or in the Settings Browser.

Initializing Connected Objects: The Postconnect Action

When you define a connector for two objects and then log in to GemStone, you connect an object in a single-user image to an object in a multiuser global space: the repository. The value of either of these objects could have been modified since the last time you logged in. The question arises: whose value is valid and should

therefore be visible? Part of the definition of a connector is to answer this question—to specify which object must be updated to match the other. This specification is called the *postconnect action*.

Postconnect actions apply only at the time the objects are initially connected. After the initial connection has been made, changes propagate in either direction as needed, depending upon behavior in your image, in GemStone, or in other GemStone sessions. See "Object Synchronization" on page 4-4.

A connector can initialize its objects in any one of the following four ways:

Update Smalltalk
Initializes the client Smalltalk object using the current state of the GemStone object. This initialization is the default for all connector types except class connectors.

Update GemStone
Initializes the GemStone object using the current state of the client Smalltalk object.

Create a forwarder
Makes the client Smalltalk object a forwarder to the GemStone object. A *forwarder* is a client Smalltalk object that responds to messages by passing them to its associated GemStone object. Forwarders are discussed in "Working with Forwarders" on page 4-19.

No initialization
Leaves the client Smalltalk object and the GemStone object "as is" after their initial connection. This initialization is the default for class connectors (see "Initializing Class Connectors: Special Considerations").

Initializing Class Connectors: Special Considerations

Unlike other kinds of connectors, class connectors have no default update direction, because it is usually desirable to prevent client Smalltalk and GemStone class definitions from updating each other. If you choose to assign an update direction to a class connector, do so with care. Updating a GemStone class from a client Smalltalk class creates a new GemStone version of the class. Updating a client Smalltalk class from a GemStone class regenerates the Smalltalk class and recompiles its methods.

For similar reasons, a class connector should generally not be instructed to treat its client Smalltalk class as a forwarder. In fact, the forwarder postconnect action is disabled for GemBuilder classes, GemStone kernel classes, and other critical classes.

Connector Nilling

When a session disconnects, GemBuilder sets certain connected variables to `nil`. This action, known as "connector nilling," reduces the risk of encountering defunct stub and defunct forwarder errors by clearing connectors that depend on being attached to GemStone objects. Connectors that represent variables (that is, name connectors, class variable connectors, and class instance variable connectors) and whose postconnect action is **updateST** or **forwarder**, are nilled. Fast connectors, class connectors, and connectors whose postconnect action is **updateGS** or **none** are not nilled.

For more information on stubs and forwarders and the possible causes of defunct stub and defunct forwarder errors, refer to Chapter 4, "Managing Replicates and Forwarders".

3.3 The Connector Browser

You can use GemBuilder's Connector Browser to manage connectors easily. Select **Tools** > **Connector Browser** from the GemStone Launcher to open a Session Browser, or click on the Connector Browser icon (Figure 3.4).

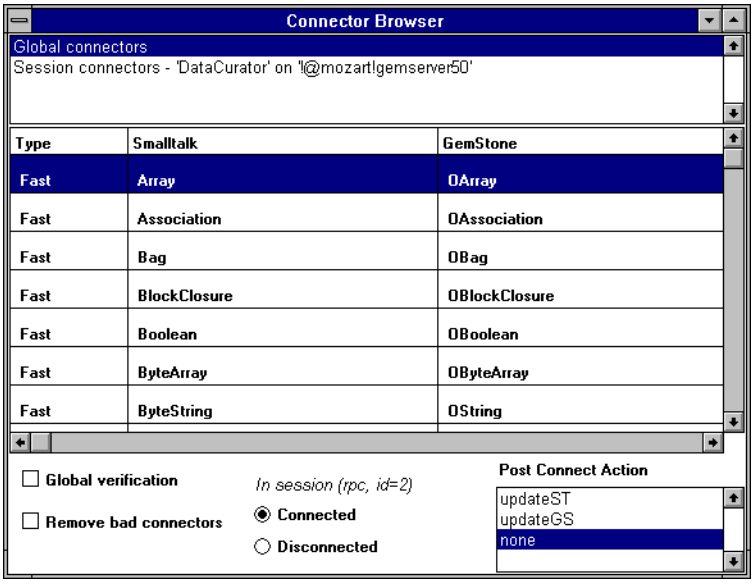
Figure 3.4 The Connector Browser Icon

This Browser allows you to:

- examine, create, and remove global or session-based connectors;
- inspect the client Smalltalk or GemStone object to which a connector resolves;
- determine whether a specified connection is currently connected;
- connect or disconnect a connector; and
- examine or modify the postconnect action associated with a connector.

Figure 3.5 shows the Connector Browser.

Figure 3.5 The Connector Browser



The Connector Group Pane

The Connector Browser's top (Group List) pane, allows you to select either **Global connectors** or an individual session. When you select a session, the connectors you have defined for that session appear in the middle (Connector List) pane.

If you select **Global connectors**, a list of connectors appears in the middle list pane; these connectors connect the GemStone kernel classes to their client Smalltalk counterparts. Kernel class connectors are all fast connectors; they connect a client Smalltalk object to a kernel object identifier in GemStone.

In the Group List pane, the middle-button menu provides the following options:

Table 3.2 Group List Menu in the Connector Browser

update	Refreshes the views and updates the browser; this option is useful if you have made changes in other windows and need to synchronize the browser with them.
initialize	*(available only when Global Connectors is selected)* Allows you to remove all connectors except those that connect kernel classes.

The Connector List Pane

The Connector List pane lists the connectors, their types and their descriptions in both client Smalltalk and GemStone. In the Connector List pane, the middle-button menu offers the following choices:

Table 3.3 Connectors Menu in the Connector Browser

inspect ST	Resolves and inspects the client Smalltalk object for the selected connector.
inspect GS	Resolves and inspects the GemStone object for the selected connector.
add...	Adds a new connector. You will be prompted for relevant information.
remove...	Removes a connector. You will be asked for confirmation before doing so.
change type...	Changes a connector to a different type of connector. You will be prompted for the type.

The Connector Control Panel

The Connector Browser's bottom pane is a control panel that allows you to change the **connectVerification** and **removeInvalidConnectors** configuration parameters, connect or disconnect objects, and modify a connector's postconnect action.

Table 3.4 Options in the Connector Browser's Control Panel

Global verification	When enabled, connectors (other than class connectors) will verify that they are not redefining an object connection. Class connectors will, upon connection, verify that the class structures are of the same storage type.
Remove bad connectors	When enabled, connectors that fail to resolve at login are automatically removed from the connector collections.
Connected / Disconnected	Connects or disconnects the GemStone and client Smalltalk objects described by the connector. Applies to the selected session, or to the current session if global connectors are selected.

Having connector verification turned on can slow down login. We suggest that you turn on verification during development and turn it off when you move into production.

Postconnect Action

The postconnect action determines how GemBuilder sets the initial state of connected objects. The selections in the postconnect action list are:

Table 3.5 Postconnect Action Options in the Connector Browser

updateST	Initializes the client Smalltalk object using the current state of the GemStone object.
updateGS	Initializes the GemStone object using the current state of the client Smalltalk object.
forwarder	Makes the client Smalltalk object a forwarder to the GemStone object.
none	Leaves the client Smalltalk object and the GemStone object "as is" after their initial connection.

To create a new connector:

1. Place the cursor in the middle pane of the Connector browser.
2. Select **add** from the Connector List menu.

3. When prompted, specify the type of connector.
4. When prompted, specify the names of the client Smalltalk and GemStone objects.
5. When prompted, specify the name of the dictionary for the GemStone object.
6. Specify the postconnect action.

To create a forwarder:

1. Create a connector as described above.
2. Select **forwarder** as the desired postconnect action. After connection, the client Smalltalk object will be a "dummy" object that responds to any message it receives by forwarding the message to its connected GemStone object. (Forwarders are discussed in "Working with Forwarders" on page 4-19.)

To change the postconnect action:

1. Select the appropriate setting.
2. Disconnect the objects by clicking on the **Disconnected** button.
3. Reconnect the objects by clicking on the **Connected** button.

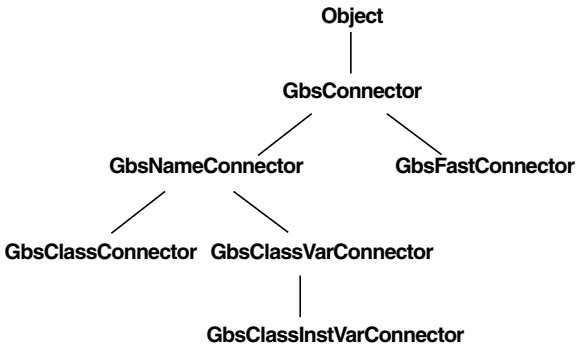
If your application initially stores its data in client Smalltalk, and you intend to store the data in GemStone but have not done so yet:

1. Create a connector or connectors for the root object(s) in the data set.
2. Temporarily identify the client Smalltalk data as master by selecting **updateGS** as the postconnect action for these connectors.
3. Log into GemStone so that GemBuilder can create the GemStone objects that replicate the client Smalltalk data. You may want to then inspect the GemStone objects to be sure they have the values you intended.
4. After the first connection, select the connectors and change their postconnect actions to **updateST** so that future sessions will begin by using the stored GemStone data.

3.4 Managing Connectors Programmatically

The GbsConnector class is an abstract superclass for the hierarchy of connector classes. Connector classes have instance variables for naming GemStone objects and the client Smalltalk objects to be connected and methods for connecting them. Figure 3.6 shows the Connector class hierarchy.

Figure 3.6 Connector Class Hierarchy



An application should create its GemStone session parameters object and its connectors in an initialization method. This section describes connector creation methods and provides some code samples to show how the methods can be used. (Creation methods for session parameters objects were described in “Defining Session Parameters” on page 2-4.)

To create a connector programmatically, the sequence of steps is:

1. Create a connector.
2. Set the connector’s postconnect action, if other than the default.
3. Add the connector to a session parameters connector list, or add it to the global connector list.

Creating Connectors

Each connector class offers a variety of instance creation methods, which you can view with a Smalltalk Class Browser. For example, a simple creation method for a name connector requires only the names of the two objects to be connected:

```
GbsNameConnector stName:    stName
                 gsName:    gsName
```

The above method assumes that the GemStone object already exists. If GemBuilder must create the object, choose an instance creation method that specifies the GemStone dictionary in which the new GemStone object should be created:

```
GbsNameConnector stName:    stName
                 gsName:    gsName
         dictionaryName:    gsDictionary
```

The first method can also be used to create a class connector:

```
GbsClassConnector stName:   stName
                  gsName:   gsName
```

Both classes must exist before they can be successfully connected.

A class variable connector can be created by a program as follows:

```
GbsClassVarConnector
    stName: # ClassName
    gsName: # ClassName
    cvarName: # ClassVarName
```

Similarly, a class instance variable connector can be created by a program as follows:

```
GbsClassInstVarConnector
    stName: # ClassName
    gsName: # ClassName
    cvarName: # ClassInstVarName
```

Setting the Postconnect Action

The symbolic names for postconnect actions are `#updateST`, `#updateGS`, `#forwarder`, and `#none`. All connectors default to using `#updateST` except class connectors, which default to `#none`.

To cause a GemStone object to take its initial values at login from its Smalltalk counterpart, send `postConnectAction: #updateGS` to the connector. This is occasionally useful for loading data into GemStone from the client Smalltalk.

Adding Connectors to a Connector List

After you create a connector programmatically, you must choose whether it is to be managed by an individual session or globally by adding it to a list of connectors. Each session maintains its own list of session connectors, and GBSM maintains a list of global connectors. (Failure to assign a connector to one of these managing entities leaves the connector "unmanaged": it will not be connected and disconnected at appropriate times and can hurt performance in object retrieval operations.)

A newly-created session connector becomes effective when the session next logs in. A newly-created global connector becomes effective the next time any session logs in.

To add a connector to a session's GemStone parameters object, execute:

ThisApplicationParameters `addConnector:` *aConnector*

If you want a connector to be invoked whenever any GbsSession logs in, put it in the global connectors collection:

`GBSM addGlobalConnector:` *aConnector*

For example, suppose your system roots are a global variable called `MyGlobal` and a class variable in `MyClass` called `MyClassVar`. An initialization method for your application might contain the code shown in Example 3.1:

Example 3.1

```
GBSM addGlobalConnector: (GbsNameConnector
            stName: #MyGlobal
            gsName: #MyGlobal);
      addGlobalConnector: (GbsClassVarConnector
            stName: #MyClass
            gsName: #MyClass
            cvarName: #MyClassVar)
```

Connecting and Disconnecting

Initialization code like that shown in Example 3.1 needs to be executed only once. From then on, every time you log into GemStone, `MyGlobal` and `MyClassVar` (and all the objects they reference) will be connected.

When a session logs in, its session connectors and all global connectors (if not already connected) are automatically connected. When a session logs out, its session connectors are disconnected. If the session is the last in the application to log out, it disconnects the global connectors.

Examples of Session Control Methods

The example methods in this section illustrate one approach to managing GemBuilder sessions and connectors. Imagine an application class that implements a help request system. One component of the application might be a session control class that defines these methods.

An instance of the session control class could be stored in the application object as a class variable, in which case the session information would be the same for all instances of the application, or it could be stored in the application as an instance variable, in which case each instance of the application would get its own copy, which it can change as needed. In either case, the methods that create the session parameters object and its connectors might follow these patterns.

The method session (Example 3.2) returns the application's logged-in session. If the session is not logged in, the method requests an RPC login and returns the resulting session. If the login attempt fails, the method returns `nil`.

Example 3.2

```
session
        "self session"
        (session isNil or: [session isLoggedIn not]) ifTrue: [
                session := self sessionParameters loginRpc.
                session isNil ifTrue: [^nil]].
        ^session
```

Example 3.3 shows a method that initializes a set of session parameters. For security, you may choose to leave passwords empty and prompt for them when they are needed.

Example 3.3

```
sessionParameters
        | params |
        sessionParameters isNil ifTrue: [
                params := GbsSessionParameters new.
                params gemStoneName: 'gemserver50'.
                params username: 'DataCurator'.
                params password: 'swordfish'.
                params gemService: 'gemnetobject'.
                params rememberPassword: true.
                params rememberHostPassword: true.
                self addConnectorsTo: params.
                sessionParameters := params.
                GBSM addParameters: params].
        ^sessionParameters
```

The method shown in Example 3.4 adds connectors to the session parameters object by calling lower-level methods to individual types of connectors.

Example 3.4

```
addConnectorsTo: aParams
        self addClassConnectorsTo: aParams.
        self addClassVarConnectorsTo: aParams
```

Example 3.5 shows a method that creates class connectors and adds them to the session parameters connector list.

Example 3.5

```
addClassConnectorsTo: aParams
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Action
                  gsName: #GST_Action).
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Customer
                  gsName: #GST_Customer).
      aParams addConnector:
            (GbsClassConnector
                  stName: #GST_Engineer
                  gsName: #GST_Engineer).
```

Example 3.6 shows a method that creates class variable connectors and adds them to the session parameters connector list.

Example 3.6

```
addClassVarConnectorsTo: aParams
      | aConnector |
      aParams addConnector:
            (aConnector := GbsClassVarConnector
                  stName: #GST_HelpRequest
                  gsName: #GST_HelpRequest
                  cvarName: #AllRequests).
            aConnector postConnectAction: #forwarder.
      aParams addConnector:
            (GbsClassVarConnector
                  stName: #GST_Company
                  gsName: #GST_Company
                  cvarName: #AllCompanies)
```

You can create methods similar to those shown in examples 3.5 and 3.6 to create name connectors and global connectors for your application, as well.

Chapter

4

Managing Replicates and Forwarders

This chapter describes how GemBuilder coordinates your application's local objects with shared objects in the GemStone repository.

Working with Replicates
gives an overview of replicates and their role in a GemBuilder application.

Instance Variable Mapping Between GemStone and Smalltalk
explains the mechanisms provided by GemBuilder for bringing GemStone objects into the client Smalltalk or client Smalltalk objects into GemStone.

Object Synchronization
describes the processes of propagating changes to GemStone objects into the client Smalltalk and changes to client Smalltalk objects into GemStone.

Class Mapping Between GemStone and Smalltalk
explains in what ways classes are treated differently than instances.

Working with Forwarders
explains how to use forwarders to store an object's state and behavior in GemStone.

4.1 Working with Replicates

A connector designates a shared object by connecting a Smalltalk object to a GemStone object. A shared object can either be *replicated* in both object spaces, or the client Smalltalk object can become a *forwarder* to the GemStone object. A *replicate* is a copy of a GemStone object in client Smalltalk. A *forwarder* is a client Smalltalk object that acts as a placeholder for a GemStone object; its data and behavior actually exist only in GemStone. The forwarder knows which GemStone object it represents, and responds to all messages by passing them to the appropriate GemStone object. In this chapter we talk first about replicates; forwarders are discussed in "Working with Forwarders" on page 4-19.

After a connection has been established between client Smalltalk and GemStone objects, GemBuilder automatically updates one of the objects when the corresponding object changes. To do this, GemBuilder must know about the structure of the two objects and the mapping between those structures.

Mapping is managed in GemBuilder on a class basis. Two connected objects have classes that are also connected. While GemBuilder can handle many commonly-used class mappings automatically with no intervention by the programmer, some nonstandard mappings may require that you override certain instance and class methods from class Object's GemStone support protocol.

4.2 Instance Variable Mapping Between GemStone and Smalltalk

By default, GemBuilder automatically maps instance variables between connected classes in the client Smalltalk and GemStone by matching their names. This allows the mapping to occur even though the instance variables of a GemStone object may be stored in a different order than their client Smalltalk counterparts. It also allows GemBuilder to automatically recompute the mapping information to keep it synchronized with changes made during a session to a class definition.

Suppressing Replication of Individual Instance Variables

You can suppress the mapping of an individual client Smalltalk instance variable by omitting its name from the corresponding GemStone class definition and vice versa. No special mapping code is required for these common cases.

When a client Smalltalk object contains a named instance variable that does not exist in its GemStone counterpart, the value of that variable is simply not mapped to GemStone, so its value is not stored when the rest of the object is stored in the

repository. When the GemStone object is faulted into the client application, the instance variable for which GemStone provides no mapping is left unchanged.

You may choose to take advantage of this behavior when a client Smalltalk object has instance variables that are relevant only in the client Smalltalk environment of the current session (for example, a reference to a window object). Such data is transient and doesn't need to be stored in GemStone.

Situations can also arise where the GemStone class has an instance variable that you do not want replicated in the client Smalltalk. In such cases, you can omit that instance variable from the client Smalltalk class definition. When the object is retrieved from GemStone, the value of that instance variable has no mapping and so is not transferred into the client Smalltalk application. When the object is stored in the repository, the unmapped instance variable in GemStone is left unchanged.

Nonstandard Instance Variable Mapping

In some cases, you may want to explicitly specify an instance variable mapping between GemStone and the client Smalltalk. For example, you might want to:

- map two instance variables to one another, even though their names don't match, or
- prevent mapping two instance variables to one another, even though their names *do* match.

You can control instance variable mapping between GemStone and your client Smalltalk by implementing a class method named `instVarMap`.

For example, a class called TestObject might provide the implementation shown in Example 4.1:

Example 4.1

```
TestObject class>> instVarMap
      ^super instVarMap ,
            #(    (stName gsName) )
```

The first component of the return value should always be a call to `super instVarMap`. This declares that all instance variable mappings established in superclasses will be in effect.

Appended to the inherited instance variable map in the above example is an array containing a pair of instance variable names to be mapped: the instance variable `stName` in Smalltalk maps to the instance variable `gsName` in GemStone.

You don't need to explicitly name the instance variable names that already match between Smalltalk and GemStone. However, if you want to suppress mapping between instance variables of the same name, you can specify that the instance variable be mapped to `nil` when moving objects into the client Smalltalk or when storing objects to GemStone, as is shown in Example 4.2.

Example 4.2

```
TestObject class>> instVarMap
      ^super instVarMap ,
            #(  (varName1 nil)
               (nil varName1)  )
```

In this example, the first pair of names (`varName1 nil`) specifies that the client object pointed to by instance variable varName1 will not be stored into GemStone.

The pair (`nil varName1`) specifies that the GemStone object pointed to by the instance variable `varName1` will not be faulted into the client Smalltalk.

4.3 Object Synchronization

Once a relationship has been established between a client Smalltalk object and a GemStone object, GemBuilder keeps their states synchronized by propagating changes as necessary.

There are two situations in which a Smalltalk / GemStone object relationship needs to be synchronized:

- A client Smalltalk object is modified, leaving its GemStone counterpart out of date. This makes the client Smalltalk object "dirty."
- A GemStone object is modified, leaving its client Smalltalk counterpart out of date. This makes the GemStone object "dirty."

The first of these conditions is the more common one if most of the application behavior exists in the client Smalltalk; the latter is more common if most of the application behavior is in GemStone or if other sessions commit modified objects.

Changed GemStone objects are automatically marked dirty by the GemStone object manager. Any commit, abort, or execution of GemStone code can cause or reveal GemStone object changes. This includes GemStone objects that have been changed and committed by *other* sessions.

Client Smalltalk objects can be marked dirty automatically, or you can choose to mark them dirty in your application code.

The term *faulting* refers to copying modified GemStone objects into the client Smalltalk, either creating a client Smalltalk replicate or updating an existing replicate. The term *flushing* refers to copying modified client Smalltalk objects into GemStone. GemBuilder manages the timing of this faulting and flushing.

Faulting Objects Into Smalltalk

Automatic faulting of GemStone objects into the client Smalltalk occurs in the following situations:

- at the beginning of a GemStone session (determined by connectors),
- when a message is sent to an object stub (see "Object Stubs" on page 4-5),
- after a `commit` or `abort` or a `continueTransaction` that reveals object modifications committed by other sessions, and
- after execution of GemStone Smalltalk code that modifies the GemStone state of an object cached in your image.

  Note: GemStone Smalltalk execution occurs when messages are sent to forwarders or when you use GbsSession >> execute:, GbsObject >> remotePerform:, or any of their variants.

Object Stubs

If a GemStone object being faulted into the client Smalltalk refers to other GemStone objects that do not have replicates in the client Smalltalk, those GemStone objects are also faulted. In the following discussion, following one reference is referred to as faulting *one level*. If those objects in turn refer to another level of objects that are faulted, this faults *two levels*. The number of levels referred to is the number of direct references that have been followed.

Because GemStone can handle much larger data sets than most client Smalltalk environments, it is often impossible to replicate an entire GemStone data set in the client Smalltalk. Furthermore, it is not necessary to incur the overhead of faulting a complete GemStone data set into the client Smalltalk if only a small number of objects are needed for the current operation.

To optimize performance, GemBuilder provides the concept of object *stubs*. A stub is an empty placeholder in the client Smalltalk that knows only which object it represents in the GemStone repository. As soon as a stub receives a message, the GemStone object is immediately faulted into the client Smalltalk. The client

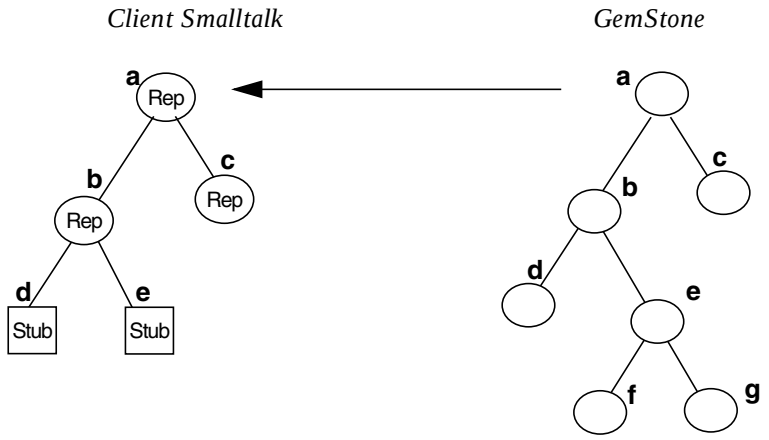
Smalltalk object can then respond to the message. After a certain number of levels of objects have been replicated in the client Smalltalk, GemBuilder stops faulting and creates object stubs rather than full replicates of objects.

Faulting Root Objects

A client Smalltalk GemBuilder image that is not logged into GemStone contains no client Smalltalk replicates of GemStone objects. The set of connectors associated with a session is not connected until the session is logged in and a replicate is created for each connector defined with `updateSTOnConnect`.

Figure 4.1 illustrates an object (object **a**) being faulted from GemStone into Smalltalk with a stub level of 2.

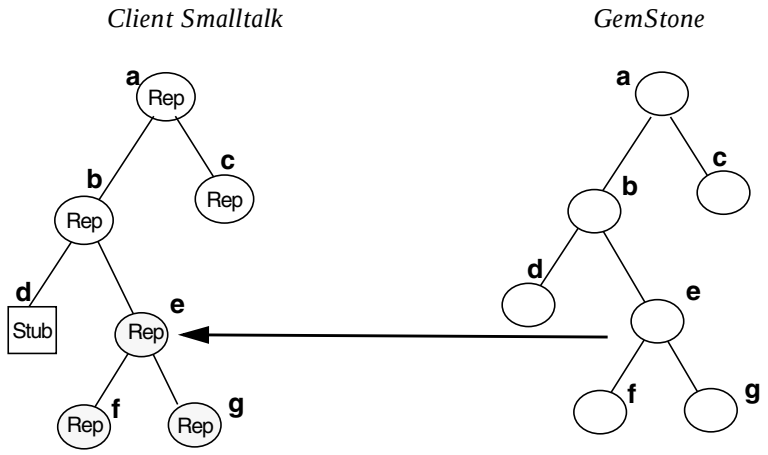
Figure 4.1 Two-level Fault of an Object



The figure shows objects **a**, **b**, and **c** being faulted into the client Smalltalk as replicates. Objects **d** and **e** are created in the client Smalltalk as stubs. Objects **f** and **g** are ignored in the fault process.

Stubbed objects are faulted in to the client Smalltalk on demand. When a stub receives a message, it immediately replicates the GemStone object so the client Smalltalk object can respond to the message. Continuing the above example, if object **e** receives a message, the stub for object **e** will be replaced with a replicate in the client Smalltalk, as shown in Figure 4.2.

Figure 4.2 Two-level Fault After Sending a Message to Stub



The previous examples used a stubbing level of 2; when an object is faulted, two levels of GemStone objects are fetched from GemStone to the client Smalltalk—the immediate object and any objects, such as instance variables, that it refers to directly. Controlling this level can be a key to optimizing a GemBuilder application.

To change the default number of levels for an object faulted in from GemStone, use the Settings Browser to set the value of `faultLevelRpc` or `faultLevelLnk`. You can also specify the fault level on an instance variable basis within a replication specification (see "Controlling Replication of Instance Variables" on page 4-9.)

These configuration parameters indicate the default number of levels to replicate when updating an object from GemStone to the client Smalltalk. A fault level of 2 means faulting the root object and each object it references directly; indirect references are represented as stubs. A fault level of 0 causes all objects referenced by the root object to be replicated in the client Smalltalk.

Preventing Stubbing

Sometimes stubbing is not desirable, either for performance reasons or for correctness. Performance considerations are discussed later, in Chapter 9.

The only correctness consideration related to stubs has to do with primitives. Primitives cannot accept stubs as parameters if the primitive accesses

the value of the parameter. If an object is used as an argument to a primitive, the application must either prevent that object from becoming a stub or ensure that the object is unstubbed before the primitive is executed.

To ensure that an object is not a stub, send it the message `fault`. If it is a stub, this message converts it to a replicate. If it is already a replicate, the message has no effect.

Explicit Stubbing

If you have a replicate in the client Smalltalk and want it to become a stub, you can send that object the message `stubYourself`. This can be useful for controlling the amount of memory required by the client Smalltalk image. Explicit control of stubs is discussed in "Optimizing Space Management" on page 9-16.

Faulting Modified GemStone Objects

Each session maintains its own view of the GemStone object server's shared object repository. The session's private view can be changed by the Smalltalk application when it adds, removes, or modifies objects, or the view can be changed by the object server when the session commits or aborts. The Gem process that oversees the session's private view of the repository maintains a list of objects that have changed. GemBuilder examines this list after any action that can modify objects in the session's view (more precisely, after transaction boundaries and GemStone Smalltalk executions).

When GemBuilder detects that an object has been modified in GemStone, it checks to see if a replicate for that object exists in the client Smalltalk. If it does, the client Smalltalk object must be updated to represent the new GemStone value.

All client objects respond to the message faultPolicy. The default implementation returns the value of GemBuilder's configuration parameter named `defaultFaultPolicy` to control when and how the client Smalltalk object is updated. Its value can be set to either `#lazy` or `#immediate`.

If an object's `faultPolicy` is `#lazy`, a change initiated in GemStone will cause the client Smalltalk replicate to be turned into a stub. The new object will not be faulted into the client Smalltalk until a message is sent to it. If the object's `faultPolicy` is `#immediate`, a change initiated in GemStone will cause the client Smalltalk replicate to be updated immediately. By default, an object's `faultPolicy` is `#lazy`, so the extra work of unnecessarily refetching the object from GemStone is avoided if the object is not sent a message.

For more information, see the description of fault policy on page 9-8. For examples, browse implementors of `faultPolicy` in the client Smalltalk.

Controlling Replication of Instance Variables

When you replicate an object from GemStone into Smalltalk you can individually control the treatment of each of the object's instance variables.

The interface's default behavior is to replicate all of an object's instance variables. You can further refine faulting behavior by class, with (if desired) particular instructions for faulting individual instance variables.

Each class has its own replication specification (spec). By default, the replication spec of a class is inherited from its superclass. If you have not modified any of the replication specs in a chain of inheritance, then the default behavior is to replicate all instance variables in Smalltalk (subject to session configuration variables, such as `faultLevelLnk`, `faultLevelRpc`, and `noStubLevel`).

To modify a class's replication specification, reimplement the class method `replicationSpec`. The `replicationSpec` method, as shown in Example 4.3, returns a list of nested arrays.

Example 4.3

```
TestObject class>> replicationSpec
^super replicationSpec ,
      #(    (instVar1 stub)
            (instVar2 forwarder)
            (instVar3 max 0)
            (instVar4 min 0)
            (instVar5 max 2)
            (instVar6 min 2)
            (instVar7 replicate)  )
```

An implementation of `replicationSpec` should always be appended to the result of **`super replicationSpec`**, as shown in Example 4.3 to ensure that defaults established in the superclass will be carried forward.

Appended to the inherited replication spec are nested arrays, each of which specifies an instance variable and an expression specifying the treatment you want it to receive when replicated:

(*instVar* *repSpec*)

The left-hand value, *instVar*, in each nested array can be

- the **name of an instance variable** (always use the names as defined in the client Smalltalk), or

- the reserved identifier "**indexable_part**" (which, as its name implies, denotes the object's indexable, unnamed, instance variables).

The right-hand value (*repSpec*) in the nested array describes the treatment the instance variable should receive when it is faulted into the client Smalltalk. The repSpec overrides system- or class-wide defaults established by the configuration variables `faultLevelLnk` and `faultLevelRpc`.

Table 4.1 lists the *repSpec* choices available and describes their effects.

Table 4.1 Instance Variable Replication Specifications

(*instVar* **stub**)	Imports the instance variable as a stub.
(*instVar* **forwarder**)	Specifies that the instance variable should be become a forwarder for the appropriate GemStone object.
(*instVar* **max** *m*)	Specifies replication to a maximum of *m* levels. **max 0** is equivalent to **stub**.
(*instVar* **min** *n*)	Specifies replication to a minimum of *n* levels. **min 0** is equivalent to **replicate**.
(*instVar* **replicate**)	Imports the instance variable as a replicate, subject to constraints such as `faultLevelLnk`, and `faultLevelRpc`. This is the default behavior, and so can be omitted unless it is necessary to restore default behavior overridden by a modified superclass definition.

Modifying Instance Variables During Faulting

You can customize object retrieval by providing buffers for the Smalltalk counterparts of the GemStone objects being faulted into the image. The contents of these buffers can then be processed into a suitable form. You can reimplement two methods, `namedValuesBuffer` and `indexableValuesBuffer`, to supply buffers for the faulting operation.

`namedValuesBuffer`
By default, returns self. New client Smalltalk objects are written directly into the named slots of the object being faulted. It can also be overridden to either supply another object of the same type or an instance of GbsBuffer of the appropriate size.

`indexableValuesBuffer`
By default, returns self. Can be overridden to return an indexable buffer of the appropriate size.

These buffers are supplied to the faulting mechanisms and can subsequently be unpacked by the faulted object by implementing the methods `namedValues:`, `indexableValues:` or the combination `namedValues:indexableValues:`. The object can then do arbitrary computations on the buffer contents to obtain the actual Smalltalk representation. The message `namedValues:indexableValues:` can be overridden in cases where the arbitrary computations need to take into account indexable as well as named values.

You can also override the messages `indexableValueAt:put:` and `namedValueAt:put:` to process the values of the indexable and named slots of the object. For example, class Set might implement the former as:

```
Set>indexableValueAt: index  put: aValue
      self add: aValue
```

The method will simply add the element to the Set rather than assigning it to a specific slot.

Two additional messages can be overridden by developers to control initialization and postprocessing of the object being faulted. The first is `preFault` which can be implemented to initialize the newly created object prior to faulting the named and indexable values of the object. For example, OrderedCollection>>preFault might be implemented as:

```
OrderedCollection> preFault
 "Initialize <firstIndex> and <lastIndex> prior to
 adding elements."
      self setIndices
```

The method `indexableValueAt:put:` for OrderedCollection has an implementation similar to Set to add the indexable objects. As another example, a specialized type of SortedCollection could use `preFault` to assign the sortBlock instance variable so that additions to the collection would be sorted properly during faulting.

The second message is `postFault`, which can be implemented to do any necessary postprocessing. For example, if the methods used to add to an OrderedCollection also marked the object dirty, the postprocessing could make

sure it is marked “not dirty” since the faulting mechanisms should not result in a dirty object.

```
OrderedCollection> postFault
 "Additions to the OrderedCollection are due to the faulting
 mechanisms and should not result in a dirty object."
       self markNotDirty
```

Defunct Object Stubs

Faulting in an object stub relies on the existence of a valid GemStone object. If a stub is created, then the session to which it belongs logs out, a message sent to that stub raises a “defunct stub” error. Because GemBuilder cannot safely assume that a given object will retain the same identifier from one session to the next, simply logging back in will not fault in an object for the defunct stub unless a connector has been defined that reestablishes that object’s relationship with GemStone, either directly or transitively.

For example, suppose you have defined a global variable named MyGlobal, and that object is modified in GemStone, causing it to be stubbed in the client Smalltalk. If that session logs out before MyGlobal is faulted back into the client Smalltalk, you are left with a defunct stub in your client Smalltalk dictionary. If you have defined a name connector for this object, then logging back into GemStone will reconnect MyGlobal with the current state of the repository.

Now, suppose an element of MyGlobal becomes stubbed and then the session logs out. Sending a message to this element will result in a “defunct stub” error. In the next login, the connector for MyGlobal will transitively fault in the element, and the message can be retried as long as it takes a path through MyGlobal, or any other root object that has a connector defined for it, rather than maintaining a direct reference to the previous defunct object.

> *NOTE: The defunct object stub error is non-proceedable. That is, if you have encountered this error, determined the cause, and corrected the problem (for example, by logging into GemStone), you must restart the Smalltalk operation that encountered the problem.*

Flushing Objects to GemStone

The term *flushing* refers to moving modified client Smalltalk objects into GemStone. GemBuilder manages the timing of this flushing, but you, as the GemBuilder programmer, must be involved in indicating which client Smalltalk objects are dirty.

GemBuilder will handle dirty object marking automatically with some initial configuration by the programmer. Alternately, you can handle the job yourself if there appears to be an advantage in doing so.

When an object is flushed, it is propagated to the session's private view of the GemStone repository. When the session commits, the modified object moves to the shared repository where it becomes accessible to other users.

Automatic Marking of Modified Objects

If you choose to let GemBuilder mark client Smalltalk objects dirty automatically, you have several choices as to when and how they are marked dirty. You can choose to have objects marked dirty on assignment of instance variables, on receiving `at:put:` messages, or in both of these situations.

GemBuilder handles dirty marking on a class-by-class basis. Some classes can have automatic dirty management enabled while others have it disabled.

To enable automatic dirty management, GemBuilder changes the default compiler for the class to send the `markDirty` message to `self` after every instance variable assignment.

You won't see this message-send by examining your source code, but (in a non-ENVY image) you can see it by looking at the decompiled version of a method. To do so, hold the shift key down while selecting the method selector in a browser. Similarly, during debugging you will notice the cursor jump to the top left of the code pane when stepping past such an embedded `markDirty` message.

It also modifies `at:put:` and `basicAt:put:` for that class to also send `markDirty`. Therefore, all named and indexable variable assignments are caught.

Named instance variable and indexable instance variable updates are controlled by two separate messages sent to client Smalltalk classes: `markDirtyOnInstvarAssign` and `markDirtyOnAtPut`. You may prefer to use `markDirtyOnInstvarAssign` because `markDirtyOnAtPut` is relatively expensive.

Note that sending a class `markDirtyOnInstvarAssign` affects only instance variable assignments in methods of that class and its subclasses. Consider the following example, in which class B is a subclass of class A.

Suppose class A has an instance variable `a`, and class B has an instance variable `b`, and B is sent `markDirtyOnInstvarAssign`. If a message is then sent to an instance of B for which the method is inherited from A, and that method modifies instance variable `a`, the object won't get marked dirty.

There are two things you can do in this case.

- You can implement the method

  ```
  B>a: newValue
        a := newValue
  ```

  in class B, and have all modification to instance variable `a` be through sending the message `a:`. This ensures that instances of A don't get charged an unnecessary `markDirty` overhead, and instances of B always get marked dirty when any instance variable is assigned.

  — or —

- You can simply enable automatic dirty management on the superclass of B (in this case, class A). Of course, this will affect all instances of A, not merely those created through class B; in some scenarios, this may be desirable.

A special method, `makeGSTransparent`, is provided that sends both `markDirtyOnAtPut` and `markDirtyOnInstvarAssign` to a class.

To remove any transparency mechanisms added by `makeGsTransparent`, you can use `removeMakeGSTransparent`. This method reverses `markDirtyOnInstvarAssign` and `markDirtyOnAtPut`.

Special Considerations for Kernel Classes

Because some kernel classes are heavily used by the client Smalltalk system, you might choose to not make them fully transparent. While the overhead for `markDirtyOnInstvarAssig` n is usually not noticeable, configuring a kernel class such as Array with `markDirtyOnAtPut` can slow the system dramatically.

An alternative to making a kernel class transparent is to create a subclass of the kernel class and send `makeGSTransparent` to it, then use instances of this class in your application instead of the kernel class.

Explicit Marking of Modified Objects

You can choose to mark client Smalltalk objects dirty explicitly. This is done by sending `markDirty` to objects when they are modified. You can usually do this more efficiently than the automatic mechanism. The danger is that you might miss places where `markDirty` should be sent.

If an object is modified in the client Smalltalk but not marked dirty, the modification will be lost eventually, probably at the start of the next session when the GemStone version of the object is fetched to refresh the client Smalltalk version.

Manually marking objects dirty is discussed further in "Managing Dirty Object Marking" on page 9-19.

Modifying Instance Variables During Flushing

By default, the instance variable mapping mechanisms operate directly on the named and indexable slots of the Smalltalk objects and their associated GemStone objects. To provide an arbitrary mapping of objects from the client Smalltalk to GemStone you can implement two methods called `namedValues` and `indexableValues`.

`namedValues`
: can be implemented to return a copy of the object being stored or an instance of GbsBuffer sized to match the number of named instance variables in the Smalltalk object. The store operations will then access this buffer for storage into GemStone.

`indexableValues`
: can be implemented to return a list of the indexable instance variables in the Smalltalk object. The store operations will then access this list for storage into GemStone.

Implementations of `namedValues` should always return an object with the appropriate number of named instance variable slots. In Example 4.4, a clone of the positionable stream is returned that increments the `position` instance variable by 1 as needed when mapped into GemStone.

Example 4.4

```
PositionableStream>> namedValues
        | aClone |
        aClone := self copy.
        aClone instVarAt: 1 put: self contents.
        aClone instVarAt: 2 put: position + 1.
        ^aClone
```

An alternative could return an instance of GbsBuffer (which is a subclass of Array) of the appropriate size. A special buffer class is necessary to distinguish between trying to store an array and trying to store the named values of an object residing in a buffer.

The default implementation of `namedValues` is to return `self`. In this case, the instance variables are processed directly from the object being stored. This eliminates the need to create a temporary array.

Implementations of indexableValues should always return an indexable collection containing a sequential list of the elements in the collection. In Example 4.5 for class Set, an Array is returned since the indexable fields of a Smalltalk set are a sparse list of the actual elements.

Example 4.5

```
Set>>indexableValues
        | values index |
        values := Array new: self size.
        index := 1.
        self elementsDo: [:each |
                values at: index put: each.
                index := index + 1].
        ^values
```

The default implementation of indexableValues is to return self. In this case, the indexable slots are processed directly from the object being stored. This eliminates the need to create a temporary array.

You can also override the messages indexableValueAt: and namedValueAt: to return processed values rather than the actual values in the indexable and named slots of the object. For example, OrderedCollection might implement indexableValueAt: as:

```
OrderedCollection>indexableValueAt: index
        ^self at: index
```

This lets OrderedCollection control the fact that its underlying indexable slots are being managed by the firstIndex and lastIndex instance variables (that is, the first actual indexable slot of the object may not necessarily be the first logical element).

In conjunction with these two methods, the messages indexableSize and namedSize might need to be reimplemented as well. For example, OrderedCollection should implement indexableSize as:

```
indexableSize
        ^self size
```

to match the implementation of indexableValueAt:. Otherwise, the object storage mechanisms would try to iterate over the entire list of indexable slots rather than those controlled by firstIndex and lastIndex.

4.4 Class Mapping Between GemStone and Smalltalk

GemBuilder does not propagate class definition changes between the client Smalltalk and GemStone during the course of a logged-in session. Although classes are generated automatically by GemBuilder when they do not exist in the client Smalltalk or GemStone, and class connectors can cause the client Smalltalk or GemStone class definitions to be updated at login time, you must manually coordinate changes to existing classes during a session.

This can be done easily with the Connector Browser's **updateGS** and **updateST** commands. Simply open the Connector Browser and select the class name. If you have made changes in the client Smalltalk, select **updateGS**; if you have made changes in GemStone, select **updateST**. To propagate your changes, disconnect the classes and reconnect them by selecting **Disconnected** then selecting **Connected**.

IMPORTANT:
Remember always to restore the correct default setting (usually "none") in the Connector Browser after you temporarily change the postconnect action.

Predefined Class Connectors

Certain connectors are predefined by GemBuilder. In particular, GemBuilder provides predefined global connectors for the GemStone kernel classes. These classes are implemented as fast connectors. Because the kernel classes to which they refer will never change identity during the course of a session, GemBuilder can take advantage of the reduced overhead that fast connectors provide when connecting to kernel classes. GemStone kernel class connectors cannot be converted to forwarders.

Automatic Generation of Classes

GemBuilder is able to generate GemStone classes from client Smalltalk classes, and vice-versa, as necessary. This means that if a Smalltalk object needs to be replicated in GemStone, but it belongs to a class that doesn't already exist in GemStone, a GemStone class with the same structure and position in the hierarchy is generated automatically. Conversely, if a GemStone class does not exist in Smalltalk, a corresponding client Smalltalk class is automatically generated when the need for it arises.

When a class is automatically generated, either in the client Smalltalk or in GemStone, a complete superclass hierarchy is created as necessary. The structure

or definition of the class is replicated, but the class's behavior is not. In other words, class and instance variables are created, but not methods. Methods for those classes must be updated through the tools in the GemBuilder programming environment.

Classes generated by GemBuilder in GemStone are deposited in a GemStone symbol list dictionary called UserClasses. Classes generated by GemBuilder in the client Smalltalk are deposited in a client Smalltalk class category named UserClasses. In ENVY they are placed in an application called UserClasses.

You can use the GemStone Settings Browser to control the behavior of automatic class generation. Use the **`generateGSClasses`** and **`generateSTClasses`** parameters, as explained on page 9-9.

- If you disable automatic generation of GemStone classes by setting **`generateGSClasses`** to `false`, situations that would normally generate a GemStone class will instead raise the signal `GbsError gsiClassGenerationFailed` .
- If you disable automatic generation of client Smalltalk classes by setting **`generateSTClasses`** to `false`, situations that would normally generate a client Smalltalk class will instead present it as a forwarder to the GemStone class.
- You can also choose to disable class connector generation by setting **`generateClassConnectors`** to `false`. If you do this, GemBuilder will still generate classes, but not connectors.

User-Created Classes With Different Formats

If you're creating a class in GemStone that maps to a client Smalltalk class whose format is different from the GemStone class (for example, the Smalltalk format is pointers but the GemStone format is bytes), you will need to reimplement the class method `gsObjImpl` in the client Smalltalk to return the value describing the GemStone implementation.

A `gsObjImpl` method must return a SmallInteger representing the GemStone implementation of a class as being one of the following formats:

- Pointers (0)
- Bytes (1)
- NSC (2)

Symbolic names for these values are stored in the pool dictionary SpecialGemStoneObjects.

4.5 Working with Forwarders

So far, this manual has not described how messages can be forwarded to GemStone for execution. We have assumed that all GemStone objects are replicated in Smalltalk and handle their messages locally. However, some GemStone objects—collections, for example—can be too large to be handled easily in Smalltalk. And some behavior might be more efficiently executed in GemStone. Therefore, GemStone allows you to define certain client Smalltalk objects as forwarders.

A *forwarder* is a client Smalltalk object whose data and behavior are actually in GemStone. The forwarder knows which GemStone object it represents, and responds to all messages by passing them to the appropriate GemStone object.

Forwarders can be declared in the following ways:

- Any connector can be initialized to declare the client Smalltalk object to be a forwarder upon login. For example, if you have a GbsNameConnector for an object named MyBigDictionary, and you want that object to be a forwarder to avoid replicating it into the client Smalltalk, you can define its postconnect action to be **forwarder**, or send it the message `postConnectAction: #forwarder`.
- To specify an instance variable forwarder, you can implement `replicationSpec` as described beginning on page 4-9. For example, if you want the `address` instance variable of class Employee to always be faulted into the client Smalltalk as a forwarder, implement:

  ```
  Employee >> replicationSpec
      ^ super replicationSpec, #(( address forwarder ))
  ```

- To return a forwarder from a GemStone name lookup, send the GbsSession method `fwat:` or `fwat:ifAbsent:` instead of `at:` or `at:ifAbsent:`.
- To explicitly create a forwarder from an instance of GbsObject, send it the message `#asForwarder`.
- You can specify that all instances of a given class are to be forwarders, by implementing a class method `instancesAreForwarders` to return `true`.

Sending Messages to Forwarders

Messages sent to forwarders (other than those intended for the forwarder itself) are forwarded to their GemStone counterpart objects. No prefix is required on the message selector. Arguments sent with a message to a forwarder are translated to GemStone objects before the message is forwarded to GemStone.

Result Objects

The result of a message sent to a forwarder is a client Smalltalk object that represents the GemStone object returned from GemStone. This will usually be a replicate, although it could be a forwarder under special conditions—for example, if it is an instance of a class whose instances are forwarders or if it's an object that already exists in the cache as a forwarder.

In some cases, you may prefer to have the result of a message to a forwarder be a forwarder itself. You can enforce such a result by prefixing the message to the forwarder with the letters `fw`. For example, the following expression will return a client Smalltalk replicate of the object at index 1:

```
aForwarder at: 1
```

The following expression, however, returns a forwarder for the object at index 1:

```
aForwarder fwat: 1
```

Replication of Client Smalltalk BlockClosures

Because forwarders are often used for large collections and collections are commonly sent messages that have blocks passed as arguments, these blocks will be replicated in GemStone.

When a GemStone replicate for a client Smalltalk Block is needed, GemBuilder will compile a new GemStone block for the receiver. If a Block is used multiple times, GemBuilder saves the reference and avoids multiple compilations.

Replicating client Smalltalk Blocks to GemStone Smalltalk and replicating GemStone Smalltalk Blocks to the client Smalltalk has the following limitations:

- Block replication is not supported in deployed runtime environments in which the GemStone Smalltalk compiler has been stripped from the image.
- Global variable references from inside a block must have the same name in both object spaces.
- Replication is not supported for blocks that reference instance variables, class variables, method arguments, or temporary variables declared external to the block's scope.
- Replicated blocks cannot contain a return or reference to `self` or `super`.

Workarounds for Block Limitations

Temp and variable reference restrictions disallow the following:

```
myDict select:
    [ :blkArg | blkArg =     aTempOrVariable ]
```

Workaround: Implement a new Dictionary method in GemStone Smalltalk named `select:with:` and rewrite as follows:

```
myDict select:
[:blkArg :extraArg | blkArg = extraArg ]        with: aTempOrVariable.
```

Restriction on references to `self` or `super` disallows:

```
myDict at:#key ifAbsent:[^self]
```

Workaround: Rewrite as follows:

```
result := myDict at:#key ifAbsent:[#absent].
result = #absent ifTrue: [ ^self ]
```

Defunct Forwarders

Because a forwarder contains no state or behavior in the client Smalltalk, it relies on the existence of a valid GemStone object. Therefore, when you log out of GemStone, any forwarders that relied on objects in that session can no longer function properly. If a message is sent to such a forwarder, GemBuilder raises a "defunct forwarder" error. Because GemBuilder cannot safely assume that a given object will retain the same identifier from one session to the next, simply logging back in does not fix a defunct forwarder unless a connector has been defined for that object or for its root.

For example, consider a forwarder to a persistent global object named BigDictionary, created programmatically as shown in Example 4.6:

Example 4.6

```
conn := GbsNameConnector
       stName: #BigDictionary
       gsName: #BigDictionary.
conn beForwarderOnConnect.
GBSM addGlobalConnector: conn
```

Each time you log into GemStone, BigDictionary becomes a valid forwarder to the current persistent BigDictionary. When you are not logged into GemStone, sending a message to BigDictionary results in a "defunct forwarder" error.

If the forwarder itself is not a root object (that is, if it is not a global variable or class variable), and you are encountering "defunct forwarder" errors, you must determine the path being taken to reach the forwarder. All persistent objects should be reachable from some small set of root objects, and these root objects should each have a connector defined for them.

For instance, suppose you have declared the address instance variable of Employee to be a forwarder, and you have a root object that is a class variable of Employee, called AllEmployees. You must then define a class variable connector for AllEmployees. The path taken to access Employee Bob's address would then be:

1. The application logs into GemStone, and the connector for AllEmployees causes the current state of the GemStone object to be faulted into the client Smalltalk.
2. A lookup into AllEmployees finds the Employee named Bob.
3. A message send to Employee Bob returns his address (a forwarder).

If, in a situation like this, you are still encountering "defunct forwarder" errors, make sure that the application has not inadvertently saved persistent objects that were looked up from an earlier GemStone session. All access paths to persistent objects should be traceable to some root object for which a connector is defined.

> *NOTE: The defunct forwarder error is non-proceedable. That is, if you have encountered this error, determined the cause, and corrected the problem (for example, by logging into GemStone), you must restart the Smalltalk operation that encountered the problem.*

GemBuilder's configuration parameter **connectorNilling**, when true, has the effect of assigning connectors' variables to nil on logout. This usually prevents defunct stub and forwarder errors, replacing them with "nil does not understand (some message)" errors.

Chapter 5

Managing Transactions

The GemStone object server's fundamental mechanism for maintaining the integrity of shared objects in a multiuser environment is the *transaction*. This chapter describes transactions and how to use them.

Transaction Management: an Overview
introduces the concepts to be explained later in the chapter.

Operating Inside a Transaction
explains the transaction model, committing, and aborting.

Operating Outside a Transaction
discusses a lower-overhead alternative for read-only views of the shared repository.

Transaction Modes
explains the difference between automatic and manual transaction modes.

Managing Concurrent Transactions
discusses concurrency conflicts and ways to minimize them, such as locks.

Reduced-Conflict Classes
describes specialized GemStone collections that minimize conflicts without locking.

Changed Object Notification
explains a mechanism for coordinating the activities of multiple sessions.

5.1 Transaction Management: an Overview

The GemStone object server provides an environment in which many users can share the same persistent objects. The object server maintains a central repository of shared objects. When a GemBuilder application needs to view or modify shared objects, it logs in to the GemStone object server, starting a session as described in Chapter 2.

A GemBuilder session creates a private view of the GemStone repository containing views of shared objects for the application's use. The application can perform computations, retrieve objects, and modify objects, as though it were a single-user Smalltalk image working with private objects. When appropriate, the application propagates its changes to the shared repository so those changes become visible to other users.

In order to maintain consistency in the repository, GemBuilder encapsulates a session's operations (computations, fetches, and modifications) in units called *transactions*. Any work done while operating in a transaction can be submitted to the object server for incorporation into the shared object repository. This is called *committing* the transaction.

During the course of a logged-in session an application can submit many transactions to the GemStone object server. In a multiuser environment, concurrency conflicts will arise that can cause some commit attempts to fail. *Aborting* the transaction refreshes the session's view of the repository in preparation for further work.

In order to reduce its operating overhead, a session can run *outside a transaction*, but to do so the session must temporarily relinquish its ability to commit. A session running outside a transaction must operate in *manual transaction mode*, in contrast to the system default *automatic transaction mode*.

GemBuilder provides ways of avoiding the concurrency conflicts that can cause a commit to fail. *Optimistic concurrency control* risks higher rates of commit failure in exchange for reduced transaction overhead, while *pessimistic concurrency control* uses locks of various kinds to improve a transaction's chances of successfully committing. GemBuilder also offers *reduced-conflict classes* that are similar to familiar Smalltalk collections, but are especially designed for the demands of multiuser applications.

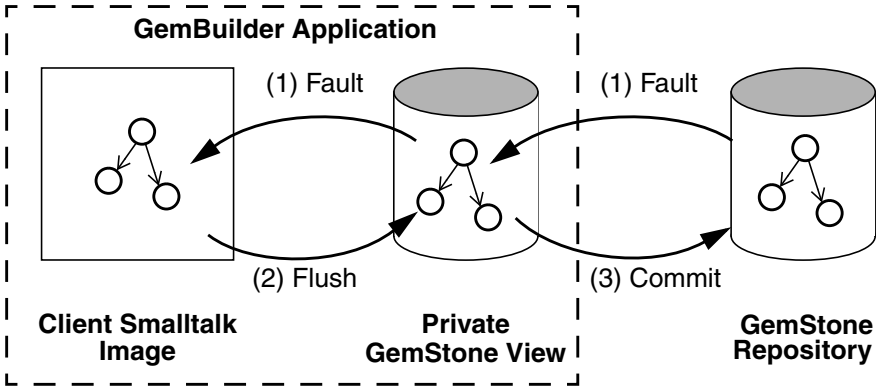
This chapter explains each of the topics mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced-conflict classes. Be sure to refer to the related topics in the *GemStone Programming Guide* for a full understanding of these transaction management concepts.

5.2 Operating Inside a Transaction

While a session is logged in to the GemStone object server, GemBuilder maintains a private view of the shared object repository for that session. To prevent conflicts that can arise from operations occurring simultaneously in different sessions in the multiuser environment, GemBuilder encapsulates each session's operations in a *transaction*. Only when the session commits its transaction does GemStone try to merge the modified objects in that session's view with the main, shared repository.

Figure 5.1 shows a client image and its repository, along with a common sequence of operations: (1) faulting in an object from the shared repository to Smalltalk, (2) flushing an object to the private GemStone view, and (3) committing the object's changes to the shared repository.

Figure 5.1 GemBuilder Application Workspace



The private GemStone view starts each transaction as a snapshot of the current state of the repository. As the application creates and modifies shared objects, GemBuilder updates the private GemStone view to reflect the application's changes. When your application commits a transaction, the repository is updated with the changes held in your application's private GemStone view.

For efficiency, GemBuilder does not replicate the entire contents of the repository. It contains only those objects that have been replicated from the repository or created by your application for sharing with the object server. Objects are replicated only when modified. This minimizes the amount of data that moves across the boundary from the repository to the Smalltalk application.

Committing a Transaction

When an application submits a transaction to the object server for inclusion in the shared repository, it is said to *commit* the transaction. To commit a transaction, send the message:

aGbsSession `commitTransaction` (to commit a specific session)

or:

`GBSM commitTransaction` (to commit the current session)

or, in the Session Browser, select a logged-in session and click on the **Commit...** button.

When the commit succeeds, the method returns `true`. Successfully committing a transaction has two effects:

- It copies the application's new and changed objects to the shared object repository, where they are visible to other users.
- It refreshes the application's private GemStone view by making visible any new or modified objects that have been committed by other users.

A commit request can be unsuccessful in two ways:

- A commit *fails* if the object server detects a concurrency conflict with the work of other users. When the commit fails the `commitTransaction` method returns `false`.
- A commit is *not attempted* if a related application component is not ready to commit. When the commit is not attempted, the `commitTransaction` method returns `nil`. (See "Session Dependents" on page 2-13.)

In order to commit, the session must be operating within a transaction. An attempt to commit while outside a transaction raises an exception.

Aborting a Transaction

A session refreshes its view of the shared object repository by aborting its transaction. Despite the terminology, a session need not be operating inside a transaction in order to abort. To abort, send the message:

aGbsSession `abortTransaction` (to abort a specific session)

or:

`GBSM abortTransaction` (to abort the current session)

or, in the Session Browser, select a logged-in session and click on the **Abort...** button.

Aborting has these effects:

- The transaction (if any) ends. If the session's transaction mode is automatic, GemBuilder starts a new transaction. If the session's transaction mode is manual, the session is left outside of a transaction.
- Temporary Smalltalk objects remain unchanged.
- The session's private view of the GemStone shared object repository is updated to match the current state of the repository.

Handling Commit Failures

If an attempt to commit fails because of a concurrency conflict, the `commitTransaction` method returns `false`.

Following a commit failure, your Smalltalk view of persistent objects may differ from its pre-commit state:

- The current transaction is still in effect. However, you must end the transaction and start a new one before you can successfully commit.
- Temporary Smalltalk objects remain unchanged.
- Modified GemStone objects remain unchanged.
- Unmodified GemStone objects are updated with new values from the shared repository.

Following a commit failure, your session must refresh its view of the repository by aborting the current transaction. The uncommitted transaction remains in effect so you can save some of its contents, if necessary, before aborting.

A common strategy for handling such a failure is to abort, then reinvoke the method in which the commit occurred. Depending on your application, you may simply choose to discard the transaction and move on, or you may choose to remedy the specific transaction conflict that caused the failure, then initiate a new transaction and commit.

If you want to know why a transaction failed to commit, you can send the message:

aGbsSession `transactionConflicts`

This expression returns a symbol dictionary whose keys indicate the kind of conflict detected and whose values identify the objects that incurred each kind of conflict. (See “Managing Concurrent Transactions” on page 5-10 for more discussion of the kinds of conflicts that can arise.)

5.3 Operating Outside a Transaction

A session must be *inside a transaction* in order to commit. While operating within a transaction, every change the session makes and every new object it creates can be a candidate for propagation to the shared repository. GemBuilder monitors the operations that occur within the transaction, gathering all the necessary information required to prepare the transaction to be committed.

For efficiency, an application may configure a session to operate *outside a transaction*. When operating outside a transaction, a session can view the repository, browse the objects it contains, and even make computations based upon their values, but it cannot commit any new or changed GemStone objects. While operating outside a transaction, a session saves some of the overhead of tracking changes, which may be significant in some applications. A session operating outside a transaction can, at any time, begin a transaction.

No session is overhead-free: even a session operating outside a transaction uses GemStone resources to manage its objects and its view of the repository. For best system performance, all sessions, even those running outside a transaction, must periodically refresh their views of the repository by committing or aborting.

Table 5.1 shows GbsSession methods that support running outside of a GemStone transaction:

Table 5.1 GbsSession Methods for Running Outside of a Transaction

`beginTransaction`	Aborts and begins a transaction.
`transactionMode`	Returns `#autoBegin` or `#manualBegin`
`transactionMode :`*newMode*	Sets `#autoBegin` or `#manualBegin`
`inTransaction`	Returns `true` if the session is currently in a transaction.
`signaledAbortAction:` *aBlock*	Executes *aBlock* when a signal to abort is received (see below).

To begin a transaction, send the message:

aGbsSession `beginTransaction`
(to begin a transaction for a specific session)

or:

`GBSM beginTransaction`
(to begin a transaction for the current session)

or, in the Session Browser, select a logged-in session and click on the **Begin...** button.

This message gives you a fresh view of the repository and starts a transaction. When you abort or successfully commit this new transaction, you will again be outside of a transaction until you either explicitly begin a new one or change transaction modes.

If you are not currently in a transaction, but still want a fresh view of the repository, you can send the message *aGbsSession* `abortTransaction`. This aborts your current view of the repository and gives you a fresh view, but does not start a new transaction.

Being Signaled to Abort

When you are in a transaction, GemStone waits until you commit or abort to reclaim storage for objects that have been made obsolete by your changes. When you are running outside of a transaction, however, you are implicitly giving GemStone permission to send your Gem session a signal requesting that you abort your current view so that GemStone can reclaim storage when necessary. When this happens, you must respond within the time period specified in the

STN_GEM_ABORT_TIMEOUT parameter in your configuration file. If you do not, GemStone forces an abort and sends your session an `abortErrLostOtRoot` signal, which means that your view of the repository was lost, and any objects that your application had been holding may no longer be valid. When you receive `abortErrLostOtRoot` , the state of your GemBuilder cache is reinitialized as though you just logged in.

You can avoid `abortErrLostOtRoot` and control what happens when you receive a signal to abort with the `signaledAbortAction` : *aBlock* message. For example:

```
aGbsSession signaledAbortAction:
    [aGbsSession abortTransaction].
```

This causes your GemBuilder session to abort when it receives a signal to abort.

5.4 Transaction Modes

A GemBuilder session always initiates a transaction when it logs in. After login, the session can operate in either of two transaction modes: automatic or manual.

Automatic Transaction Mode

In *automatic transaction mode*, committing or aborting a transaction automatically starts a new transaction. This is GemBuilder's default transaction mode: in this mode, the session operates within a transaction the entire time it is logged into GemStone.

However, being in a transaction incurs certain costs related to maintaining a consistent view of the repository at all times for all sessions. Objects that the repository contained when you started the transaction are preserved in your view, even if you are not using them and other users' actions have rendered them meaningless or obsolete.

Depending upon the characteristics of your particular installation (such as the number of users, the frequency of transactions, and the extent of object sharing), this burden can be trivial or significant. If it is significant at your site, you may want to reduce overhead by using sessions that run outside transactions. To run outside a transaction, a session must switch to manual transaction mode.

Manual Transaction Mode

In *manual transaction mode*, the session remains outside a transaction until you begin a transaction. When you change the transaction mode from automatic (its initial setting) to manual, the current transaction is aborted and the session is left outside a transaction. In manual transaction mode, a transaction begins only as a result of an explicit request. When you abort or commit successfully, the session remains outside a transaction until a new transaction is initiated.

To begin a transaction, send the message

aGbsSession `beginTransaction`

or select the **Begin...** button on the Session Browser.

A new transaction always begins with an abort to refresh the session's private view of the repository. Local objects that customarily survive an abort operation, such as temporary results you have computed while outside a transaction, can be carried into the new transaction where they can be committed, subject to the usual constraints of conflict-checking. If you begin a new transaction while already inside a transaction, the effect is the same as an abort.

In manual transaction mode, as in automatic mode, an unsuccessful commit leaves the session in the current transaction until you take steps to end the transaction by aborting.

Choosing Which Mode to Use

You should use automatic transaction mode if the work you are doing requires committing to the repository frequently, because you can make permanent changes to the repository only when you are in a transaction.

Use manual transaction mode if the work you are doing requires looking at objects in the repository, but only seldom requires committing changes to the repository. You will have to start a transaction manually before you can commit your changes to the repository, but the system will be able to run with less overhead.

Switching Between Modes

To find out if you are currently in a transaction, execute *aGbsSession* `inTransaction`. This returns `true` if you are in a transaction and `false` if you are not.

To change from manual to automatic transaction mode, execute the expression:

aGbsSession `transactionMode: #autoBegin`

This message automatically aborts the transaction, if any, changes the transaction mode, and starts a new transaction.

To change from automatic to manual transaction mode, execute the expression:

aGbsSession `transactionMode: #manualBegin`

This message automatically aborts the current transaction and changes the transaction mode to manual. It does not start a new transaction, but it does provide a fresh view of the repository.

5.5 Managing Concurrent Transactions

When you tell GemStone to commit your transaction, it checks to see if doing so presents a conflict with the activities of any other users.

1. It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have also modified during your transaction. If they have, then the resulting modified objects can be inconsistent with each other.
2. It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have read during your transaction, while at the same time you have modified an object that the other session has read.
3. It checks for locks set by other sessions that indicate the intention to modify objects that you have read or to read objects you have modified in your view.

If it finds no such conflicts, GemStone commits the transaction, and your work becomes part of the permanent, shared repository. Your view of the repository is refreshed and any new or modified objects that other users have recently committed become visible in any dictionaries that you share with them.

Read and Write Operations

It is customary to consider the operations that take place within a transaction as *reading* or *writing* objects. Any operation that accesses any instance variable of an object *reads* that object, as do operations that fetch an object's size, class, or other descriptive information about that object. An object also is read in the process of being stored into another object.

An operation that stores a value in one of an object's instance variables *writes* the object. While you can read without writing, writing an object always implies reading it, because GemStone must read the internal state of an object in order to store a value in it.

In order to detect conflict among concurrent users, GemStone maintains two logical sets for each session: a set containing objects read during a transaction and a set containing objects written. These sets are called the *read set* and the *write set*. Because writing implies reading, the read set is always a superset of the write set.

The following conditions signal a possible concurrency conflict:

- An object in your write set is also in ananother transaction's write set (a *write/write* conflict).
- An object in your write set is in another transaction's read set *and* an object in your read set is in that transaction's write set (a *read/writ*e conflict).

Optimistic and Pessimistic Concurrency Control

GemStone provides two approaches to managing concurrent transactions: optimistic and pessimistic. An application can use either or both approaches, as needed.

Optimistic concurrency control means that you simply read and write objects as if you were the only session, letting GemStone detect conflicts with other sessions only when you try to commit a transaction.

Pessimistic concurrency control means that you act as early as possible to prevent conflicts by explicitly requesting locks on objects before you modify them. When an object is locked, other users are unable to lock the object or commit changes to it.

Optimistic concurrency control is easy to implement in an application, but you run the risk of having to re-do the work you've done if conflicts are detected and you're unable to commit. When GemStone looks for conflicts only at commit time, your chances of being in conflict with other users increase with the time between commits and the size of your read and write sets. Under optimistic concurrency control, GemStone detects conflict by comparing your read and write sets with those of all other transactions committed since your transaction began.

Running under optimistic concurrency control is the most convenient and efficient mode of operation when:

- you are not sharing data with other sessions, *or*
- you are reading data but not writing, *or*

- you are writing a limited amount of shared data *and* you can tolerate not being able to commit your work sometimes.

If you take a pessimistic approach, you act as early as possible to prevent conflicts by explicitly requesting locks on objects before you modify them. When an object is locked, other people are unable to lock the object, and they cannot optimistically commit changes to the object. Also, when you encounter an object that someone else has locked, you can abort the transaction immediately instead of wasting time on work that can't be committed.

Locking improves one user's chances of committing, but at the expense of other users, so you should use locks sparingly to prevent an overall degradation of system performance. Still, if there is a lot of competition for shared objects in your application, or if you can't tolerate even an occasional inability to commit, then using locks might be your best choice.

Locks do not prevent read-only access to objects, so read-only query transactions are not affected by modification transactions.

Setting the Concurrency Mode

Any shared object that is not explicitly locked is treated optimistically. For objects under optimistic concurrency control, GemStone's level of checking for concurrency conflicts is configurable. You can set the level of checking for concurrency conflicts by specifying one of the following values for the CONCURRENCY_MODE configuration parameter in your application's configuration file. There are two levels:

- FULL_CHECKS (the default mode), which checks for both *write/write* and *read/write* conflicts. If either type of conflict is detected your transaction cannot commit.
- NO_RW_CHECKS, which performs *write/write* checking only.

Locking methods override the configured optimistic CONCURRENCY_MODE by stating explicitly the kind of pessimistic control they implement.

Setting Locks

GemBuilder provides locking protocol that allows application developers to write client Smalltalk code to lock objects and specify client Smalltalk code to be executed if locking fails.

A GbsSession is the receiver of all lock requests. Locks can be requested on a single object or on a collection of objects. Single lock requests are made with the following statements:

aGbsSession `readLock:` *anObject*.
aGbsSession `writeLock:` *anObject*.
aGbsSession `exclusiveLock:` *anObject*.

The above messages request a particular type of lock on *anObject*. If the lock is granted, the method returns the receiver. (Lock types are described in the *GemStone Programming Guide*.) If you don't have the proper authorization, or if another session already has a conflicting lock, an error will be generated.

When you request an exclusive lock, an error will be generated if another session has committed a change to *anObject* since the beginning of the current transaction. In this case, the lock is granted despite the error, but it is seen as "dirty." A session holding a dirty lock cannot commit its transaction, but must abort to obtain an up-to-date value for *anObject*. The lock will remain, however, after the transaction is aborted.

Another version of the lock request allows these possible error conditions to be detected and acted on.

aGbsSession `readLock:` *anObject* `ifDenied:` *block1* `ifChanged:` *block2*
aGbsSession `writeLock:` *anObject* `ifDenied:` *block1* `ifChanged:` *block2*
aGbsSession `exclusiveLock:` *anObject* `ifDenied:` *block1* `ifChanged:` *block2*

If another session has committed a change to *anObject* since the beginning of the current transaction, the lock is granted but dirty, and the method returns the value of the zero-argument block *block2*.

The following statements request locks on each element in the three different collections.

aGbsSession `readLockAll:` *aCollection*.
aGbsSession `writeLockAll:` *aCollection*.
aGbsSession `exclusiveLockAll:` *aCollection*.

The following statements request locks on a collection, acquiring locks on as many objects in *aCollection* as possible. If you do not have the proper authorization for any object in the collection, an error is generated and no locks are granted.

aGbsSession `readLockAll:` *aCollection* `ifIncomplete:` *block1*
aGbsSession `writeLockAll:` *aCollection* `ifIncomplete:` *block1*
aGbsSession `exclusiveLockAll:` *aCollection* `ifIncomplete:` *block1*

Example 5.1 shows how error handling might be implemented for the collection locking methods:

Example 5.1

```
getWriteLocksOn:aCollection
 "This method attempts to set write locks on the elements
 of a Collection."
aGbsSession
 writeLockAll: aCollection
 ifIncomplete: [ :result |
                  (result at: 1)isEmpty ifFalse:
                        [self handleDenialOn: denied].
                  (result at: 2)isEmpty ifFalse:
                        [aGbsSession abortTransaction].
                  (result at: 3)isEmpty ifFalse:
                        [aGbsSession abortTransaction].
              ].
```

Once you lock an object, it normally remains locked until you either log out or explicitly remove the lock; unless you specify otherwise, locks persist through aborts and commits. In general, you should remove a lock on an object when you have used the object, committed the resulting values to the repository, and no longer anticipate a need to maintain control of the object.

The following methods are used to remove specific locks.

aGbsSession `removeLock:` *anObject*`.`
aGbsSession `removeLockAll:` *aCollection*`.`
aGbsSession `removeLocksForSession.`

The following methods answer various lock inquiries:

```
aGbsSession sessionLocks.
aGbsSession systemLocks.
aGbsSession lockOwners: anObject.
aGbsSession lockKind: anObject.
aGbsSession lockStatus: anObject.
```

Releasing Locks Upon Aborting or Committing

The following statements add a locked object or the locked elements of a collection to the set of objects whose locks are to be released upon the next commit or abort:

```
aGbsSession addToCommitReleaseLocksSet : aLockedObject
aGbsSession addToCommitOrAbortReleaseLocksSet : aLockedObject
aGbsSession addAllToCommitReleaseLocksSet : aLockedCollection
aGbsSession addAllToCommitOrAbortReleaseLocksSet : aLockedCollection
```

If you add an object to one of these sets and then request a fresh lock on it, the object is removed from the set.

You can remove objects from these sets without removing the lock on the object. The following statements show how to do this:

```
aGbsSession removeFromCommitReleaseLocksSet: aLockedObject
aGbsSession removeFromCommitOrAbortReleaseLocksSet: aLockedObject
aGbsSession removeAllFromCommitReleaseLocksSet: aLockedCollection
aGbsSession removeAllFromCommitOrAbortReleaseLocksSet: aLockedCollection
```

The following statements remove all objects from the set of objects whose locks are to be released upon the next commit or abort:

```
System clearCommitReleaseLocksSet
System clearCommitOrAbortReleaseLocksSet
```

The statement `System commitAndReleaseLocks` clears all locks for the session if the transaction was successfully committed.

5.6 Reduced-Conflict Classes

At times GemStone will perceive a conflict when two users are accessing the same object, when what the users are doing actually presents no problem. For example, GemStone may perceive a write / write conflict when two users are simultaneously trying to add an object to a Bag that they both have access to because this is seen as modifying the Bag.

GemStone provides some reduced-conflict classes that can be used instead of their regular counterparts in applications that might otherwise experience too many unnecessary conflicts. These classes include RcCounter, RcIdentityBag, RcKeyValueDictionary, and RcQueue.

Use of these classes can improve performance by allowing a greater number of transactions to commit successfully without locks, but they do carry some overhead.

For one thing, they use more storage than their ordinary counterparts. Also, you may find that your application takes longer to commit transactions when you use instances of these classes. Finally, you should be aware that under certain circumstances, instances of these classes can hide conflicts from you that you indeed need to know about. Because of the way these classes are implemented, GemBuilder creates instances of these classes as forwarders, rather than replicates.

Here are brief descriptions of the reduced-conflict classes. For details about these classes and their usage, see the *GemStone Programming Guide* and the *GemStone Kernel Reference*.

RcCounter
RcCounter maintains an integral value that can be incremented or decremented. A single instance of RcCounter can be shared among multiple concurrent sessions without conflict.

RcIdentityBag
RcIdentityBag provides the same functionality as IdentityBag, except that no conflict occurs on instances of RcIdentityBag when a number of users read objects in the bag or add objects to the bag at the same time. Nor is there a conflict when one user removes an object from the bag while other users are adding objects, or when a number of users remove objects from the bag at the same time, so long as no more than one of them tries to remove the last occurrence of an object.

RcKeyValueDictionary
This class provides the same functionality as KeyValueDictionary except that no conflict occurs on instances of RcKeyValueDictionary when users read

values in the dictionary or add keys and values to it (unless one tries to add a key that already exists) or when users remove keys from the dictionary at the same time (unless more than one user tries to remove the same key at the same time).

Conflict occurs only when more than one user tries to modify or remove the same key from the dictionary at the same time.

RcQueue

The class RcQueue represents a first-in-first-out (FIFO) queue. No conflict occurs on instances of RcQueue when multiple users read objects in or add objects to the queue at the same time, or when one user removes an object from the queue while other users are adding objects. However, if more than one user removes objects from the queue, they are likely to experience a write / write conflict.

5.7 Changed Object Notification

A *notifier* is an optional signal that is activated when an object's committed state changes. Notifiers allow sessions to monitor the status of designated shared application objects. A program that monitors stock prices, for example, could use notifiers to detect changes in the prices of certain stocks.

In order to be notified that an object has changed, a session must register that object with the system by adding it to the session's *notify set*.

Notify sets are virtual but persist through transactions, living as long as the GemStone session in which they were created. When the session ends, the notify set is no longer in effect. If you need it for your next session, you must recreate it. However, you need not recreate it from one transaction to the next.

Class GbsSession provides the following two methods for adding objects to notifySets:

`addToNotifySet:`
adds one object to the notify set

`addAllToNotifySet:`
adds the contents of a collection to the notify set

When an object in the notify set appears in the write set of any committing transaction, the system executes a previously defined client Smalltalk block, sending a collection of the objects signaled as its argument. By examining the argument, the session can determine exactly which object triggered the signal.

Because these events are not initiated by your session but cause code to run within your session, this code is run asynchronously in a separate Smalltalk process. Depending on what else is occurring in your application at that time, using this feature might introduce multi-threading into your application, requiring you to take some additional precautions. (See "Multi-threaded Applications" on page 9-21.)

Example 5.2 demonstrates notification in GemBuilder.

Example 5.2

```
"First, set up notifying objects and notification action"
| notifier |
GBSM currentSession abortTransaction; clearNotifySet.
notifier := Array new: 1.
GBSM currentSession at: #Notifier put: notifier.
GBSM currentSession commitTransaction.
GBSM currentSession addToNotifySet: notifier.
GBSM currentSession notificationAction: [ :objs |
      Transcript cr; show: 'Notification received' ]

"Now, from any session logged into the same stone with
visibility to the object 'notifier' - to initiate
notification"
GBSM currentSession abortTransaction;
      execute: 'Notifier at: 1 put: Object new';
      commitTransaction
```

Gem-to-Gem Notification

Sessions can send general purpose *signals* to other GemStone sessions. These signals allow the transmission of the sender's `sessionId` along with a numerical signal value and an associated message.

A client image can handle gem-to-gem signals using the method GbsSession >> `gemSignalAction:` *aBlock*, where *aBlock* is a three-argument block that will be passed the sessionId of the signalling session, an arbitrary signal number, and an arbitrary message string.

One GemStone session can send a signal to another session, by sending:

aGbsSession `sendSignal:` *aSignal* `to:` *aSessionId* `withMessage:` *aString*

Here is an example of how to use gem-to-gem signalling:

Example 5.3

```
"First, set up a signal receiving action"
GBSM currentSession gemSignalAction: [ :sessId :sigNum :message |
       Transcript cr; show: 'Signal ', sigNum printString,
              ' received from session ', sessId
                printString, ': ', message ].
"Now, from any session logged into the same stone, send a
signal.(This example uses the same session)"
GBSM currentSession
       sendSignal: 15
       to: (GBSM evaluate: 'System session')
       withMessage: 'This is the signal'
```

See your *GemStone Programming Guide* for details on using the error mechanism for change notification.

Chapter

6 Security and Access to Objects

Once objects have been successfully committed to GemStone, they can be damaged or destroyed only by mishaps that damage or erase the disk files containing your repository. GemStone provides several mechanisms for safeguarding the objects in your GemStone repository. These mechanisms are discussed in the chapter on creating and restoring backups in the *GemStone System Administration Guide.*

This chapter discusses security and access at the object level.

Object-Level Security
highlights the mechanisms GemStone provides for keeping your stored objects secure.

Classes for Controlling Access to Objects
describes the three key classes —Repository, Segment, UserProfile—that provide object-level security.

Sharing Access to Objects
explains how you can use GemStone's group authorization mechanism and the individual users' UserProfiles to ensure that users have appropriate access to the objects they need.

GemStone Administration Tools
describes the visual tools that you can use to manage access to objects by

multiple users: the Segment Tool, the Symbol List Browser, and the User Account Manager.

6.1 Object-Level Security

GemStone provides for blocking access to certain objects as well as sharing them. Applications can take advantage of several security mechanisms to prevent unauthorized access to, or modification of, sensitive code and data. These mechanisms are listed below, and you can choose to use any or all of them.

Requiring Login Authorization

GemStone's first line of protection is to control login authorization. When someone tries to log in to GemStone, GemStone requires a user name and password. If the user name and password match the user name and password of someone authorized to use the system, GemStone allows interaction to proceed; if not, the connection is severed.

The GemStone system administrator controls login authorization by establishing user names and passwords when he or she creates UserProfiles.

Controlling Visibility of Objects

You can also control access by hiding certain objects from users. Because it is difficult, if not impossible, for users to refer to objects that are not defined somewhere in their symbol lists, simply omitting off-limits objects from a user's symbol list provides a certain amount of security. It is possible, however, for users to find ways to circumvent this, because it's difficult to ensure that all indirect paths to an object are eliminated.

Protecting Methods

Another choice is to implement procedural protection. If your program accesses its objects only through methods, you can control the use of those objects by including user identity checks in the accessing methods.

Using GemStone's Authorization Mechanisms

The easiest and most reliable way to secure objects, however, is to use GemStone's authorization and privilege mechanisms.

Segments

GemStone's authorization mechanism uses a class called Segment to protect objects from access by users who have not been explicitly given permission to use them. Every user can use the authorization mechanism to protect both data and code objects selectively.

Segment objects have an owner, and settings for read and write access for the owner, groups of users, and everyone else. When someone tries to read or write an object that references a segment to which he or she lacks the proper access, GemStone raises an authorization error and does not permit the requested operation.

Segments are not meant to organize objects for retrieval; GemStone uses Symbol Lists for that. Moreover, segments don't have any relationship to the physical location of objects on disk; they are merely security objects.

Segments are discussed in more detail in subsequent sections: "GemStone Administration Tools" on page 6-10, and "The Segment Tool" on page 6-10.

Privileges

A few GemStone Smalltalk methods can be executed only by those who have explicitly been given the necessary *privileges*. The privilege mechanism is entirely independent of the authorization mechanism. This mechanism allows the system administrator to control who can send certain powerful messages, such as those that halt the system or change passwords. Privileges are associated with only a few methods and cannot be extended to others.

For more information about security in general and about the above mechanisms in particular, see the relevant chapter of the *GemStone Programming Guide*. For more specific information about privileged methods, see the chapter of the *GemStone System Administration Guide* that discusses common system operations.

6.2 Classes for Controlling Access to Objects

There are three key classes that cooperate in providing access control at the object level: Repository, Segment, and UserProfile. This section describes how these classes interact to maintain control of access to objects in an application.

Repository

All disk space used by GemStone—that is, your entire object store—is represented as a single instance of class Repository. Committing an object consigns it to that repository as a member of a segment.

When your system is first delivered, GemStone's repository maps to a single file called **`extent0.dbf`**, whose name and physical location can be controlled by the GemStone system administrator, with operating system commands and configuration files. In GemStone, the name of the initial repository object is SystemRepository. The repository's name can be changed by the system administrator or anyone with equivalent authorization.

Segment

The SystemRepository object initially has three instances of class Segment associated with it:

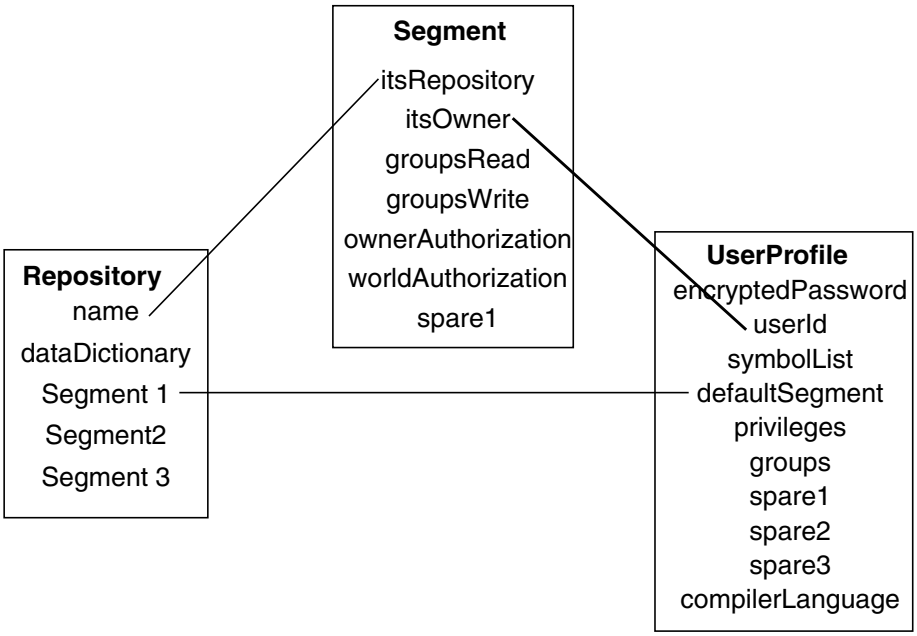
- the SystemSegment (owned by the SystemUser),
- the DataCuratorSegment (owned by the DataCurator), and
- the GcUser's Segment (owned by the GcUser).

A segment has no physical basis; it is not a location. It is merely a logical entity that serves as a means of controlling ownership of, and access to, objects. New segments can be added to the SystemRepository when new users are added.

Each segment has a single owner and stores a reference to the owner's UserProfile.

Figure 6.1 shows the relationship between the classes Repository, Segment, and UserProfile.

Figure 6.1 GemStone's Object-Level Security Mechanism



Each segment associated with the SystemRepository contains instance variables that refer to its repository, its owner, the groups that are authorized to read and / or write objects that are assigned to it, and the level of authorization for the segment's owner and for the world. Note that segments do not know which objects are assigned to them, nor are they meant to organize objects for easy listing and retrieval; that is the role of symbol lists.

UserProfile

When you become a GemStone user you are assigned a UserProfile object. Your UserProfile stores information about you as an individual user, such as your name, your password, and a list of symbol dictionaries that the compiler can consult to find the objects named in your applications. Your UserProfile also stores a reference to a segment that serves as your *default segment*. When you create objects, GemStone assigns them to your default segment, unless you specify otherwise. You can be the owner of your default segment, or the system administrator may have assigned ownership to someone else.

You can use your default segment for all the objects you create, but if you have the proper privileges, you can exert more control over access to your objects by creating additional segments. For example, you can create the classes and other objects for an application in a private segment and then reassign them to segments that other users can access.

Your *current segment* is the same as your default segment when you log in, but you can designate a different segment to be your current segment, so that subsequent new objects will be assigned to it instead of to your default segment.

As a segment's owner, you have control over the access that you and others have to the objects assigned to it, and you can authorize access separately for:

- *owner* — You can also alter your own access rights at any time, even forbidding yourself to read or write objects assigned to the segment.
- *groups* — You can authorize groups of users (by name).
- *world* — You can provide or restrict access to all GemStone users.

Note that these categories can overlap.

If you lose write authorization to your current segment, your default segment becomes your current segment as soon as the transaction that changed authorization is committed. If you lose write authorization to your default segment, GemStone terminates your session with an error, because GemStone execution cannot continue without permission to create temporary objects in some segment.

You can find out information about your default segment by executing `System myUserProfile defaultSegment` with a **GS-print.** The system will return something like this:

```
Segment, Number 1 in Repository SystemRepository
Owner SystemUser write
World read
```

6.3 Sharing Access to Objects

Two conditions must be present for a group of users to share access to an object:

- The object must be assigned to a segment that authorizes all the users to access it.
- The object must also be accessible from each user's Symbol List.

Group Authorization and Object Sharing

When you have a group of users working with the same GemStone application, you need to arrange for everyone in the group to have access to the objects that need to be shared, such as the application classes. You might also want to limit access to certain data objects to some specified users.

While an application's developer may require full read and write access to all its objects, the end users may need to see them all, but only modify a few. GemStone users generally fall into three categories:

- *developers*, who define classes and methods;
- *updaters*, who create and modify instances; and
- *reporters*, who read and output information.

GemStone's segment mechanism allows you to provide the appropriate level of access for each type of user by organizing users who have common interests or needs into designated groups. Like segment owners, groups can be given the authorizations `#write`, `#read`, or `#none`.

When a GemStone user tries to read or write an object assigned to a particular segment, GemStone compares the group authorizations stored by the segment with the group memberships stored in that user's UserProfile. If there is a group in common, and if the authorization for that group permits what the user is trying to do, the operation is allowed to continue; if not, barring other relevant authorizations, the operation is halted.

If your default segment is associated with a group, any user whose UserProfile also includes that group has the right to read from your default segment. Note that you cannot always add group authorizations for a segment you own, and you might need to ask your data curator or system administrator for help in adding authorization for a new group.

Using Segments for Authorization

As explained in the previous section, *segments* define authorization attributes for all the objects assigned to them. All objects assigned to a given segment have the same protection; if you can read or write one object assigned to a segment, you can read or write all of them. Each segment is owned by one user, and that user authorizes read access, write access, or no access at all to objects assigned to that segment for the owner, a number of named groups of users, and the world—that is, all users of that GemStone repository.

A segment can be used to provide or restrict access to objects that are associated with it. Segments are merely authorization objects; they are not storage locations. Whenever you create an object it is associated with a Segment, and the characteristics of the segment determine who has access to that object.

Each segment has a single owner who can determine the level of access that the various groups of GemStone users have to that segment and to the objects associated with it.

A segment knows who its owner is and which repository it is associated with. A segment also knows if any groups are associated with it.

However, a segment *doesn't* know who the members of these groups are; it knows only what type of access (read, write, or none) these groups have to the objects that reference it. A segment also doesn't know which objects it controls; instead, each GemStone object knows to which segment it has been assigned

In other words, while segments know their authorization attributes, they do not store references to the objects that are assigned to them. That information is stored in the objects.

For example, suppose a segment specifies that only its owner has write access to objects, and everyone else is limited to read access. When the segment's owner creates an object associated with that segment, only the owner will be able to modify that object; everyone else will have read-only access.

Whenever a program tries to read or write an object, GemStone compares the authorization characteristics of the segment with the characteristics of the user who is attempting to do the reading or writing. If those characteristics match, the operation proceeds. If not, GemStone returns an error notification.

Because each object has separate authorization, each object must be assigned separately. This per-object authorization is useful during multiuser development, because there might be some objects that you want to share and other objects you don't want to share. For example, you could choose to make a collection shared, but keep the existing elements private, allowing other developers to add elements, but not modify the elements you have already created.

GemStone's use of segments to control authorization is an efficient way to maintain flexibility and simplicity in managing object access. It allows you to change authorization by changing the segment, rather than having to make changes to individual objects.

Making Objects Accessible Through Symbol Lists

In setting up a UserProfile, the data curator initially includes in each user's symbol list the dictionaries that define the names of all the objects he or she believes that user would need. Initially, each user's symbol list generally includes at least:

- a *Globals* dictionary that defines the GemStone kernel classes and any other global objects,
- a *Published* dictionary for globally-visible shared objects,
- a private *UserGlobals* dictionary in which the user can store objects defined for his or her own use.

Your symbol list tells the GemStone Smalltalk compiler which of many possible GemStone dictionaries to search through to find an object named in your program and determines the order in which to search them. Unless a variable is local or is defined in a method's class, the GemStone Smalltalk compiler can resolve that reference only if it is in one of the dictionaries named in your symbol list.

The GemStone Smalltalk compiler searches for names in the following order:

1. It first checks to see if the variable is local—that is, a temporary variable or argument.
2. If the variable is not local, the compiler checks to see if the variable is defined by the class that defines the current method or one of its superclasses.
3. If it still cannot resolve the reference, the compiler searches the symbol list in your UserProfile sequentially—that is, from top to bottom.

   This means that if some name— for example, `#Supplier` —is defined in the first dictionary and in the last dictionary in your symbol list, the compiler will find only the first definition.

Because you can use GemStone's symbol resolution mechanism to arrange to share—or not to share—any specific object with other GemStone users, it is necessary for you to be aware of what is in your symbol list and to understand how to use symbol list dictionaries for sharing objects. You can use GemBuilder's Symbol List Browser to do this. (See "The Symbol List Browser" on page 6-18).

All you have to do to set up other users to share access to specified objects is to name the objects in question in a specific symbol list dictionary, then make sure that all the relevant users include that dictionary in the symbol list of their UserProfiles.

When you examine your symbol list dictionaries you may notice that most, if not all, of the dictionaries refer to themselves. For example, the dictionary named

UserGlobals contains an Association for which the key is UserGlobals and the value is the dictionary itself. Symbol list dictionaries define their own names so that you can refer to them conveniently in your own applications.

You can add the references to existing dictionaries, or you may prefer to create a special-purpose dictionary for each application, adding the specialized dictionaries to symbol lists as needed. Your system's authorization mechanism is probably set up to prohibit you from doing this yourself, so you will probably need the cooperation of your GemStone data curator. The *GemStone System Administration Guide* provides more information on this subject.

For a complete discussion of symbol resolution and object sharing, see the relevant chapters of the *GemStone Programming Guide.*

6.4 GemStone Administration Tools

The following sections describe the GemStone tools that are provided to allow you to easily manage the object sharing and protection issues discussed in the previous section.

- **The Segment Tool** is a tool for examining and changing GemStone user authorization.
- **The Symbol List Browser** is a tool you can use for examining the GemStone SymbolLists associated with UserProfiles. You can use it to add and delete dictionaries from these lists, as well as to add, delete and inspect the entries in the dictionaries.
- **The User Account Manager** is a tool that allows you to create new user accounts, change account passwords, and assign group membership.

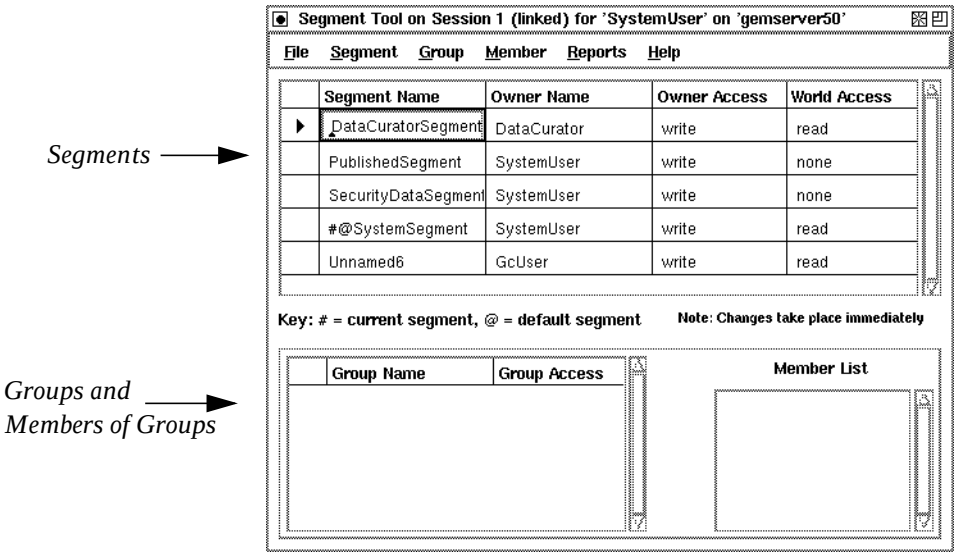
The Segment Tool

The Segment Tool allows you to inspect and change the authorization that GemStone users have at the object level. As explained in the section entitled "Group Authorization and Object Sharing" beginning on page 6-7, each object in GemStone is assigned separately to a segment. The only users authorized to read or modify an object are those who are granted read or write authorization by the segment it is assigned to. The Segment Tool also allows you to examine and change group membership.

Some of the operations supported by the Segment Tool involve privileged methods. If your account does not have the needed privileges, ask your system administrator to set up your segments for you.

You can open a Segment Tool by selecting **Segment Tool** from the GemStone **Tools** menu or through the User Account Manager's **Show Segments** button. Figure 6.2 shows a Segment Tool.

Figure 6.2 The Segment Tool



The Segment Tool is divided into two main sections: one displays segments; the other displays groups and their members.

Segment Definition Area

The segment definition area at the top of the dialog displays the segments in the SystemRepository to which the current user has read authorization.

You will notice that some segments are named, some are unnamed, and some segment names are preceded by symbols. Named segments are segments that are referenced in a dictionary or symbol list; unnamed segments are those that are not referenced in any dictionary or symbol list

When a segment name is prefixed by one of the special characters **#** and **@**:

- **#** means this is your current segment, the segment to which GemStone will assign any objects you create now.

- **@** means this is your default segment, the home segment that is your current segment when you log into GemStone.

When you change a segment name, you can put one or both of these characters at the beginning of the name to designate that segment to be the current segment, the default segment, or both.

In addition to the segments displayed in the Segment Tool, all users also have read and write authorization to GsIndexing segment. Because authorization changes should not be made to that segment, however, it is not included in the tool.

IMPORTANT:

You should be aware that any changes made to cells in the tables will be accepted automatically as soon as you either press Return, make a selection in a combo box associated with the cell, or simply move the focus to another cell or field by moving the mouse. If you enter an invalid value in a cell results in a warning, and the cell reverts to the original value.

In the segment definition area of the Tool you can change the following:

Segment Name — You can edit the names of named segments.

Owner Name — You can enter any valid user name that already exists in the system. To change an owner name, type a valid owner name into the cell.

Owner Access and **World Access —** To change owner and world access, type one of the following values into their cells:

- **read** means that a user can read any of the segment's objects, but can't modify (write) them or add new ones
- **write** means that a user can read and modify any of the segment's objects and create new objects associated with the segment
- **none** means that a user can neither read nor write any of the segment's objects

NOTE:

Be careful when changing the authorizations on any segment that a user may be using as a current segment or a default segment.

Group Definition Area

The bottom of the dialog is the group definition area. In this area you can assign authorizations to groups of users instead of individuals. Groups are typically organized as categories of users who have common interests or needs.

When you select a segment at the top of the dialog, the group definition area displays the groups that have access to the segment. When you select one of the groups, its members appear.

In the group definition area you can change the following:

Group Name — You can change the group name, but you should be aware that when you edit a group name, you are not just renaming the group; you are actually replacing the group with a new one. The old group's members are not copied to the new one, so you need to add them again. If the name of the group entered is a group that does not exist, you will be asked if you want to create it.

Group Access — Group access can be changed in the same way as owner and world access. To change group access, type either **read** or **write** into the cell, as outlined for owner and world access on page 6-12.

NOTE:
Be careful when changing the authorizations on any segment that a user may be using as a current segment or a default segment.

If you want to add group access to a segment, select **add...** from the pop-up menu in a Group Name cell. Similarly, to remove group access from a segment, select **remove...** from the pop-up menu.

In addition, you can select groups and users here to be the receiver of actions on the menus.

Segment Tool Menus

The following sections describe the menus that are available in the Segment Tool.

The File Menu

Use the **File** menu to commit work done in the Segment Tool, to abort the transaction, to update the tool's view of segments, groups, and users in the current session, and to close the Segment Tool.

Table 6.1 File Menu in the Segment Tool

Commit	Commits all the work executed in GemStone during the current transaction. After you commit, you are given a new, updated view of the repository, and you can continue your work.
Abort...	Cancels all changes that you have made anywhere in GemStone since your last commit. After you abort the transaction, you are given a new, updated view of the repository, and you can continue your work.
Update	Updates the information in the Segment Tool and gives you a new, updated view of segments, groups, and users that reflects the most recent version of the repository, and you can continue your work.

Segment Menu

Use the **Segment** menu to create new segments and to manipulate existing segments.

Table 6.2 Segment Menu in the Segment Tool

Create...	Creates a new segment. You must have the SegmentCreation privilege to use this option. In the Create Segment dialog, enter a name for the segment and a symbol dictionary to store it in. Private segments are typically kept in UserGlobals. Segments for large groups of users are typically kept in Globals.
Grab	Grabs a reference to the selected segment and places it on the clipboard. This can be used to add a reference to a user's symbol list or for changing the default segment of a user in the User Account Manager.
Make Current	Makes the selected segment your current segment. When you create an object, GemStone assigns it to your current segment.
Make Default	Makes the selected segment your default segment. This is the home segment that is your current segment when you log into GemStone.

Group Menu

Use the **Group** menu to add and remove groups.

Table 6.3 Group Menu in the Segment Tool

Add...	Adds a new group. In the Add Group dialog, enter a name for the group and choose **OK** or **Apply**.
Remove...	Removes authorization for the selected group. This does not delete the group from GemStone. It only means that the current segment no longer stores access information for that group. Users may still be able to access other objects because of their membership in the group, but they will not have access to the objects assigned to this segment unless it has been provided by the segment's owner or world access.

Member Menu

Use the **Member** menu to add users to and remove users from groups.

Table 6.4 Member Menu in the Segment Tool

Add...	Adds a user to the group. Enter any valid user name in the Add Member dialog and choose **OK** or **Apply**. The user must already exist in the system. You can use the User Account Manager to create new users.
Remove...	Removes the selected user from the group. (This does not delete the user from GemStone.)

Reports Menu

Use the **Reports** menu to bring up a window displaying information about the segments, users, and groups in your view of the repository. Use the window's **Print** button to print a report, and use the **Cancel** button to close the window.

Table 6.5 Report Menu in the Segment Tool

Group Report	Produces a list of all groups in GemStone and the users in each group.
Segment Report	Produces a list of segments the user has read authorization for and displays information about each one as to • its owner, • the groups for which it contains access information, and • the access it grants to the owner, groups, and world. This report includes the GsIndexing segment, for which all users have read and write authorization.
User Report	Produces a list of all GemStone users and shows each user's group memberships.

Segments that appear as Unnamed are not in your symbol list. Thus, their names and dictionaries are unknown.

Help Menu

The **Help** menu contains one item, **Session Info**, which provides information about the session for the Segment Tool window and about the current session.

Using the Segment Tool

If you are a segment's owner, you can determine who has access to objects assigned to that segment. For more information, see the chapter on administering user accounts and segments in the *GemStone System Administration Guide.*

Checking Segment Authorization

Anyone who has read authorization for a segment can use the Segment Tool to find out who is authorized to read or write that segment by doing the following:

1. Bring up the Segment Tool by selecting **GemStone** >**Admin Tools> Browse Segments** or by choosing **Show Segments** in a GemStone User dialog.
2. In the Segment Tool, choose **Reports > Segment Report**. The resulting list contains all segments.
3. To view the members of each group, choose **Reports > Group Report**. To view the groups to which each user belongs, choose **Reports > User Report**.

Changing Segment Authorization

Assuming that you either have SegmentProtection privileges or are the segment's owner, you can use the Segment Tool to change the authorization of a segment.

The top half of the Segment Tool shows the owner, the owner's access, and world access for each segment in the repository. To change owner or world access, select the existing permission you want to change. Then enter a new permission ("read", "write", or "none").

The new authorization will take effect when you commit the current transaction.

CAUTION:
Be careful to check whether a user is logged in before you remove write authorization. A user will be unable to commit changes if write authorization is removed from the current segment, and if it is the user's default segment, the user's session will be terminated and the user will be unable to log in again.

Controlling Group Access to a Segment

If you are authorized to set up or change group access to a segment, you can add or remove groups to that segment's authorization list.

- Make sure the segment is selected in the top half of the tool.
- To add a group to the authorization list for the selected segment, choose **Add...** from the **Group** menu. Enter the group name in the dialog box that appears. If the group does not exist in the repository, you will be asked whether to create it.
- To remove a group from the authorization list, first select the group by clicking in the first column of the groups list. Then choose **Remove...** from the **Group** menu. You will be asked to confirm the action.
- To change the type of access for a particular group, first select that group in the groups list and select the existing permission. Then enter the new permission ("read" or "write").
- To add a member to a group that has access to this segment, first select that group in the groups list. Then choose **Add...** from the **Member** menu. Enter the UserId and choose **OK**. (A UserProfile with that UserId must already exist in the repository.)
- To remove a member from a group that has access to this segment, select the UserId in the member list and choose **Remove...** from the **Member** menu. You will be asked to confirm the action.

Remember to commit your transaction before logging out. A convenient way to do that is by choosing **Commit** from this tool's **File** menu.

Changing a User's Default Segment

You must either have DefaultSegment privileges to change your own default segment, or have write authorization in the DataCurator Segment to change another user's default segment.

To change your own default segment, select the desired segment by clicking in its first column. Then choose **Segment > Make Default**. (You can also do this by typing the @ symbol in front of the segment name.)

To change someone else's default segment, in the GemStone User dialog, choose **Show Segments**.

1. In the Segment Tool, select the desired segment by clicking in its first column. Choose **Segment > Grab**.
2. In the GemStone User dialog, choose **Paste To Default Segment**.
3. Choose **OK** or **Apply**.

NOTE:
Changes to a segment's authorization do not take effect until the current transaction is committed. This means that if you change any user's default segment (including your own) to a segment for which that user lacks write authorization, and you subsequently commit the transaction, the affected user will no longer be able to log in to GemStone.

The Symbol List Browser

The Symbol List Browser is a tool for examining the GemStone SymbolLists associated with UserProfiles, adding and deleting dictionaries from these lists, examining the entries in those dictionaries and adding, deleting and inspecting the entries. References to dictionaries and dictionary entries can be copied between GemStone user accounts, subject to authorization and segment restrictions, to allow users to share application objects and name spaces developed by other users, and to publish them to other users.

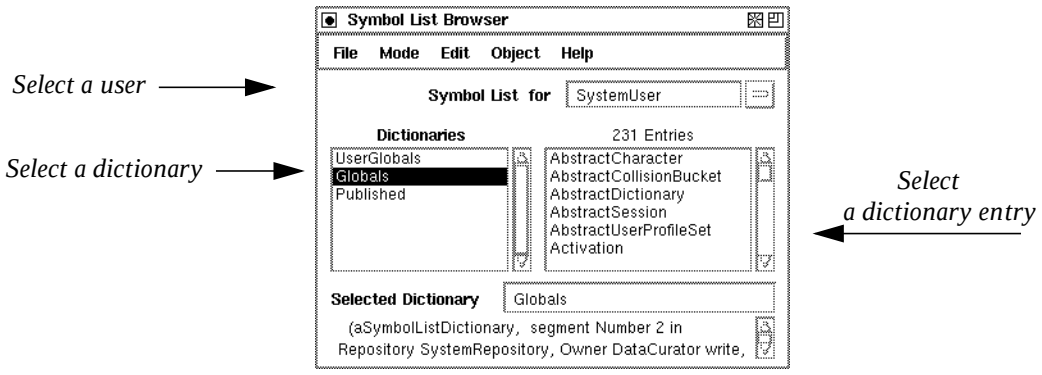
To open a Symbol List Browser, select **Browse Symbol Lists** from the **Gemstone > Admin Tools** menu or click on the **Show Symbol List** button on a GemStone User window.

Like the other GemStone tools, the Symbol List Browser opens on a particular login session. When a Symbol List Browser instance is created, it is attached to the

current GemStone session and remains attached to that session until the browser is closed.

Figure 6.3 shows the Symbol List Browser.

Figure 6.3 The Symbol List Browser



The field labeled **Symbol List for** contains a list of all the GemStone users that are visible to the session to which the browser is attached. When you select a GemStone user name, a list of the dictionaries in that user's SymbolList is displayed in the **Dictionaries** pane. GemStone permissions are observed; any dictionaries in that SymbolList that are not normally accessible to the browser's session will not be visible in the list.

When a dictionary is selected, the keys of the entries in the dictionary are displayed in the **Entries** pane on the right.

Whenever a dictionary or an entry is selected, information about that object is displayed at the bottom of the browser.

The Clipboard

Within the Symbol List Browser you can delete, move, and copy objects to and from SymbolLists and the Dictionaries in those SymbolLists. For each session to which a Symbol List Browser is attached, there is a "clipboard" onto which GemStone objects can be cut and copied and from which objects can be pasted into another Symbol List Browser that is also attached to that session.

Symbol List Browser Menus

The menus in the symbol list browser allow you to examine, add, and delete SymbolLists, dictionaries, and dictionary entries. You can use this browser to copy references to dictionaries and dictionary entries among user accounts so application objects can be shared by other users.

File Menu

The **File** menu contains items for operating on the window itself and for committing and aborting transactions from the Symbol List Browser.

Table 6.6 File Menu in the Symbol List Browser

Commit	Makes all changes in the current transaction permanent.
Abort	Aborts the current transactions.
Update	Updates the browser's view of the GemStone objects it shows. The browser is automatically updated if the attached session aborts a transaction.

Mode Menu

The Mode menu allows you to switch from dictionary mode to entry mode. In dictionary mode, you can select entries and dictionaries from the lists. In entry mode, you can edit or enter new text in the Symbol List and Selected Entry fields.

Edit Menu

In Dictionary Mode, the Edit menu allows you to rearrange dictionaries by cutting, copying, or pasting. In Entry Mode, the Edit menu allows you to rearrange entries by cutting, copying, or pasting.

Table 6.7 Edit Menu in the Symbol List Browser

Cut Dict **Cut Entry**	*In Dictionary mode:* Removes the selected dictionary from the user's symbol list and places it in the session's clipboard. *In Entry mode:* Removes the selected entry from the selected Dictionary and places it in the session's clipboard.
Copy Dict **Copy Entry**	Copies a reference to the selected item (a dictionary or an entry, depending which mode is in effect) into the session's clipboard.
Paste Dict **Paste Entry**	*In Dictionary mode:* Causes the reference to the dictionary object in the clipboard to be added to the SymbolList in the pane, with the name it had when it was put in the clipboard. *In Entry mode:* Causes the reference to the entry in the clipboard to be added to the selected dictionary, with the name it had when it was put in the clipboard. *In both modes:* If the clipboard item's name is already in use in the destination list, a Confirmer will pop up to allow replacing the old item, or to abort the paste operation.

Object Menu

The Object Menu allows you add a new dictionary, open an inspector on a dictionary entry, and open a browser on a class that is contained in a dictionary. Its menu items are: **Add Dict, Inspect Dict,** and **Browse Class.**

Table 6.8 Object Menu in the Symbol List Browser

Add Dict **Add Entry**	Prompts for name of a new item to be added to the Dictionary or Entry list.
Inspect Dict **Inspect Entry**	Opens a GemStone inspector on the selected item.
Browse Class	If the selected entry is a class, opens a GemStone class browser on that entry. Performs the same operation in either Dictionary or Entry mode.

Help Menu

The **Help** menu contains one item, **Session Info**, which provides information about the session for the Symbol List Browser and about the current session.

User Account Management Tools

The User Account Management tools are a set of three tools that allow the GemStone System Administrator to create new user accounts, change account passwords, and assign group membership.

This section describes the three User Account Management tools: the GemStone User List, the GemStone User Dialog, and the Privileges Dialog.

Note that you must either be DataCurator or have certain privileges to perform most of the system administration functions described in this section. If you are responsible for GemStone system administration, you should refer the chapter on administering user accounts and segments in the *GemStone System Administration Guide* for specific information on user account management. That chapter discusses the privileges you need to manage user accounts and explains how to add and remove users, set up user environments, change passwords and user privileges, and how to add and remove users from groups.

GemStone User List

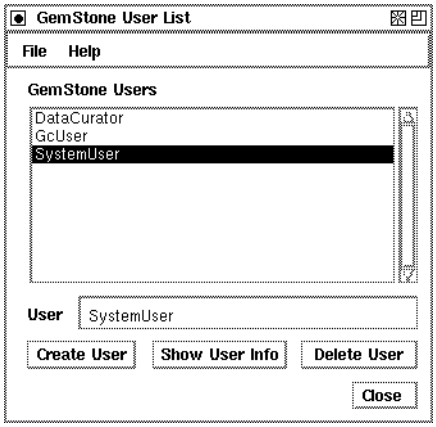
The GemStone User List window contains a list of all user accounts known to the current repository. The administrator can use this window to delete users and as a starting point to add new users and to change the attributes of GemStone users.

To bring up the User Account Manager, select **Admin > GemStone Users** from the GemStone launcher or click on the User Account Manager Icon (Figure 6.4).

Figure 6.4 User Account Manager Icon

Figure 6.5 shows the GemStone User List.

Figure 6.5 GemStone User List



The GemStone Users List window has two menus: **File** and **Help**.

The **File** menu contains the following items.

Table 6.9 GemStone User List: File Menu

Commit	Makes all changes in the current transaction permanent.
Abort	Aborts the current transaction.
Update	Causes the browser to update its view of the GemStone users it shows. The browser will automatically be updated if the attached session aborts a transaction.

The **Help** menu contains one item, **Session Info**, which provides information about the session for the GemStone User List and about the current session.

GemStone User Dialog

The GemStone User Dialog displays the attributes of a particular GemStone user. The GemStone administrator can examine and change the user's privileges or default segment and can control the user's group membership. The administrator can also change the name available in the user's symbol list.

The GemStone User Dialog is shown in Figure 6.6.

Figure 6.6 GemStone User Dialog

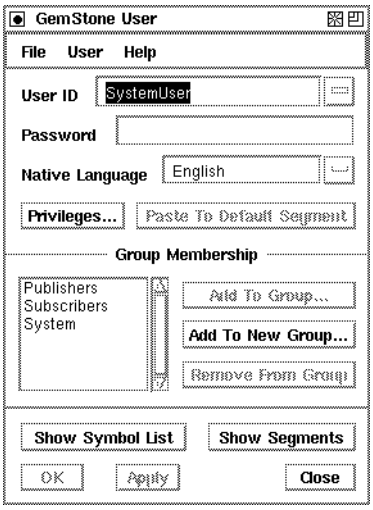


Table 6.10 shows the operations that are available in this dialog.

Table 6.10 Buttons in the GemStone User Window

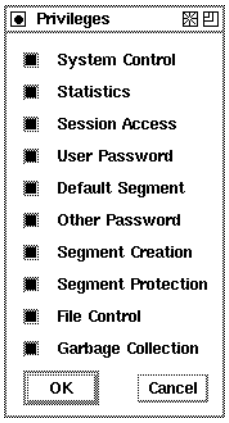
Privileges...	Brings up a Privileges dialog, in which you can select privileges for this user.
Paste To Default Segment	Makes the grabbed segment from the Segment Tool the default segment for this user.
Add to Group	Allows you to select a group name from a menu of known groups and adds this user to that group.
Add To New Group	Prompts you for a new group name and adds this user to the group.
Remove From Group	Removes this user from the selected group.
Show Symbol List	Brings up a Symbol List Browser for the designated user.
Show Segments	Brings up a Segment Tool.

The Privileges Dialog

Certain system functions are customarily performed by the DataCurator; for example, many of the messages to System require explicit privilege to use. The privileges dialog displays the privileges an individual user possesses. You can use this dialog to examine a user's privileges, and—if you have the authority to do so—to select privileges for a user. For more information on privileges, see the chapter on Administering User accounts and Segments in the *GemStone System Administration Guide.*

The Privileges Dialog is shown in Figure 6.7.

Figure 6.7 Privileges Dialog in GemStone User Window



The privileges and the methods that are associated with them are shown in Table 6.11.

Table 6.11 Privileges

Type of Privilege	Privileged Methods	In Class

System Control	`resumeLogins`
	`shutDown`
	`stopOtherSessions`
	`stopSession:`
	`suspendLogins`
		System

Statistics	`stoneStatistics`	System

Table 6.11 Privileges

Type of Privilege	Privileged Methods	In Class

Session Access	`concurrencyMode:`
	`currentSessionNames`
	`currentSessions`
	`descriptionOfSession:`
	`stopOtherSessions`
	`userProfileForSession:`
		System

User Password	`oldPassword: newPassword:`	UserProfile
Default Segment	`defaultSegment:`	UserProfile
Other Password	`password:`	UserProfile
Segment Creation	`newInRepository:`	Segment

Segment Protection	`group:authorization:`
	`ownerAuthorization:`
	`worldAuthorization:`
		Segment

File Control	`abortRestore`
	`addTransactionLog:replicate:size:`
	`commitRestore`
	`continueFullBackupTo:MBytes:`
	`createExtent:`
	`createExtent: withMaxSize:`
	`createReplicateOf:named:`
	`disposeReplicate:`
	`fullBackupTo:`
	`fullBackupTo:MBytes:`
	`restoreFromBackup:`
	`restoreFromBackups:`
	`restoreFromCurrentLogs`
	`restoreFromLog:`
	`shrinkExtents`
	`startNewLog`
		Repository

Garbage Collection	`auditWithLimit:`
	`findDisconnectedObjects`
	`markForCollection`
	`pagesWithPercentFree:`
	`repairWithLimit:`
	`scavengePagesWithPercentFree:`
		Repository

Chapter 7

Schema Modification and Coordination

No matter how elegantly your schema was designed, sooner or later changes in your application requirements or even changes in the world around your application will probably make it necessary to make changes to classes that are already instantiated and in use. When this happens, you will want the process of propagating your changes to be smooth and to impact your work as little as possible.

This chapter discusses the mechanisms GemStone and GemBuilder provide to help you accomplish this.

Schema Modification
explains how GemStone supports schema modification by maintaining versions of classes in class history objects. It shows you how to migrate some or all instances from one version of a class to another while retaining the data that these instances hold.

Schema Coordination
explains how to synchronize schema modifications between GemStone and the client Smalltalk.

The Class Version Browser
describes a specialized Class Browser that can be used for examining a class

history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

7.1 Schema Modification

Client Smalltalk and GemStone Smalltalk both have schema modification support. Client Smalltalk supports only a single instance of a class; when a class is modified, instance migration occurs immediately. Because GemStone stores persistent objects, schema modification is a more complex issue.

GemStone Smalltalk supports schema modification and protects the integrity of your stored data by allowing you to define different versions of classes. It keeps track of these versions in a class history object.

Every class in GemStone Smalltalk has a class history instance variable. A class history is an object that maintains a list of all versions of the class. Every GemStone class is listed in exactly one class history. You can define any number of different versions of a class and declare that the different versions belong to the same class history. You can also migrate some or all instances of one version of a class to another version when you need to. By default, migration of an object from one class version to another will preserve the values of unnamed instance variables and instance variables that have the same name in both classes.

It is not necessary for different versions of a class to have a similar structure or a similar implementation. The classes don't even need to have the same name, although it is probably less confusing if they do or if you establish and adhere to some naming convention.

The section entitled "Modifying an Existing Class" on page 8-25 explains how to create different versions of a class in GemBuilder.

Instance Migration Within GemStone

The migration operation in GemStone is flexible and configurable.

- Instances of any class can migrate to any other, as long as they share a class history. The two classes need not be similarly named or have anything else in common.
- Migration can occur whenever you want it to.
- You don't have to migrate all instances of a class at once; you can migrate only certain instances as needed.

- You can choose which values of the old instance variables are used to initialize values of the new instance variables. overriding the default mapping mechanism as necessary.

Setting the Migration Destination

You can use the message `migrateTo:` to set a migration destination in the class that you need to migrate from as follows:

OldClass `migrateTo:` *NewClass*

This message merely lets the class know its migration destination; it does not cause migration to occur. Migration takes place only when the class receives one of the `migrateInstances` messages described in the section "Migrating Objects."

It is not necessary to set a migration destination ahead of time; you can specify the destination class when you decide to migrate instances. It is also possible to set a migration destination and then migrate the instances of the old class to a completely different class by specifying a different migration destination as part of the message that performs the migration.

You can erase the migration destination for a class by sending it the message `cancelMigration`, and you can query the migration destination by sending `migrationDestination` to the class.

Migrating Objects

A number of mechanisms are available to allow you to migrate one instance or a specified set of instances to a previously specified migration destination or to another explicitly specified destination.

You can execute the following expression to identify instances that may need to be migrated:

`SystemRepository listInstances:` *anArrayOfClasses*`.`

The `listInstances:` message takes as its argument an array of classes and returns an array of sets. The contents of each set consists of all instances whose class is equal to the corresponding element in the argument *anArrayOfClasses*. Instances to which you lack read authorization are omitted without notification.

The simplest way to migrate an instance of an older class is to send the message `migrate` to the instance. If the object is an instance of a class for which a migration destination has already been defined, the object becomes an instance of the specified version of the class. If no destination has been defined, no change occurs.

You can bypass the migration destination or migrate instances of classes for which no migration destination has been specified by specifying the destination directly in the message that performs the migration.

The following messages (defined in class Class) specify a one-time-only operation that ignores any preset migration destination class.

`migrateInstances:` *aCollectionOfInstances* `to:` *DestinationClass*

`migrateInstancesTo:` *DestinationClass*

The `migrateInstances:to:` message migrates specified instances to a class; the `migrateInstancesTo:` migrates all instances of the receiver to a class.

Things to Watch Out For

There are a few things that you should be aware of when migrating objects.

- You cannot send a `migrate` message to `self`. Attempting to do so generates an error that reports "The object you are trying to migrate was already on the stack."
- You cannot migrate instances that you are not authorized to read or write.
- You need to be aware that the instance variable map used in migrating instances from one GemStone class to another is not the same as the instance variable map described in Chapter 4, whose purpose is to map instance variables from GemStone to Smalltalk.

Instance Variable Mapping in Migration

GemStone supports instance migration between two classes that belong to the same class history. For simple migrations, such as the addition or removal of an instance variable, GemStone provides a default migration mechanism that copies data from each instance variable of the old object to the instance variable of the same name in the new object (if one exists). You can write methods to customize this migration on a class-by-class basis.

When an object is migrated, it refers to the class and class instance variables that have been defined for the new version of the class. These variables have whatever values have been assigned to them in the class object.

The simplest way to retain the data held in instance variables is to use instance variables having the same names. If two versions of a class have instance variables with the same name, the values of those variables are automatically retained when the instances migrate from one class to the other.

However, the structure of the two classes may be different, and a one-to-one mapping may not be possible. For example, if the new class has an instance variable for which no corresponding variable exists in the old class, that instance variable is initialized to *nil* upon migration. Similarly, if the old class has an instance variable for which no corresponding variable exists in the new class, the value of the old variable is dropped and the data it represents is no longer accessible from that object.

You may encounter situations in which you want to initialize a variable having one name with the value of a variable having a different name. This requires providing an explicit mapping from the instance variable names of the older class to the instance variable names of the migration destination. To do this you will need to override the default mapping strategy by reimplementing a class method named `instVarMappingTo:` in your destination class. This method is defined in Class to return an instance variable mapping from the receiver's named instance variables to those in the other class, but it can be customized in the new class to explicitly map the two different names.

There also may be times when you need to perform a specific operation on the value of a given variable before initializing the corresponding variable in the class to which the object is migrating.

For example, suppose that you have a class named Point, which defines two instance variables: *x* and *y*. These instance variables define the position of the point in Cartesian two-dimensional coordinate space. Now suppose that you define a class named NewPoint to use polar coordinates. The class has two instance variables named *radius* and *angle*. The default mapping strategy would cause Point objects to completely lose their position because the old and new classes have no instance variables in common.

This can be handled, however, by overriding a migration method in NewPoint by defining it to include an operation that transforms the values of *x* and *y* into values that can properly be assigned to *radius* and *angle*. In this case, the appropriate method to override is `migrateFrom:instVarMap:` . Then, when you migrate an instance of Point to an instance of NewPoint, the migration code that calls `migrateFrom:instVarMap:` executes the method in NewPoint instead of the one in Object that defines the default behavior. (This example is explained in detail in the *GemStone Programming Guide*.)

7.2 Schema Coordination

GemBuilder's goal in supporting schema migration is to provide an interaction between the client Smalltalk and GemStone that provides as much of GemStone's capabilities as possible, while minimizing the impact on the client Smalltalk system.

GemBuilder preserves the behavior of having only a single version of a given class in client Smalltalk at one time. That client Smalltalk class will be mapped to a specific version of a GemStone class, resolved at login time by its name. If, while faulting an object into the client Smalltalk, GemBuilder discovers that the object is an instance of a class that is a different version of the class that is in client Smalltalk, it will be faulted in in the format of the class in client Smalltalk and flagged so that if it is modified and written back to GemStone, it can be written out in the appropriate format.

For example, suppose you have a class named **C** in GemStone, and there are two versions of it: $\mathbf{C_1}$ and $\mathbf{C_2}$. Suppose that client Smalltalk has a representation of $\mathbf{C_2}$. Instances of $\mathbf{C_2}$ are replicated back and forth between client Smalltalk and GemStone, as usual.

If it attempts to replicate an instance of $\mathbf{C_1}$, however, GemBuilder will discover that there is no class mapping for $\mathbf{C_1}$. GemBuilder will then do the following:

1. It will fetch the name of the GemStone class and discover that there is a client Smalltalk class by the same name that is already mapped to a GemStone class.
2. It will verify that the two GemStone classes are in the same class history.
3. It will then ask GemStone to make a migrated copy of the object in $\mathbf{C_2}$ format and to replicate that migrated copy into client Smalltalk. The proxy associated with that client Smalltalk object will be flagged to indicate that the client Smalltalk object is a migrated representation of the GemStone object. If that object is later modified in client Smalltalk and subsequently needs to be written to GemStone, GemBuilder will first flush the object from client Smalltalk to GemStone as an instance of $\mathbf{C_2}$, then have GemStone migrate the object back to an instance of $\mathbf{C_1}$.

This process is fairly expensive. If you are running GemBuilder in verbose mode, the discovery of an client Smalltalk class that is mapped to an old version of a GemStone class (a version that is not the migration destination) will be logged to the transcript. If you see this happening frequently, you should consider migrating your instances to the GemStone class version corresponding to your client Smalltalk class.

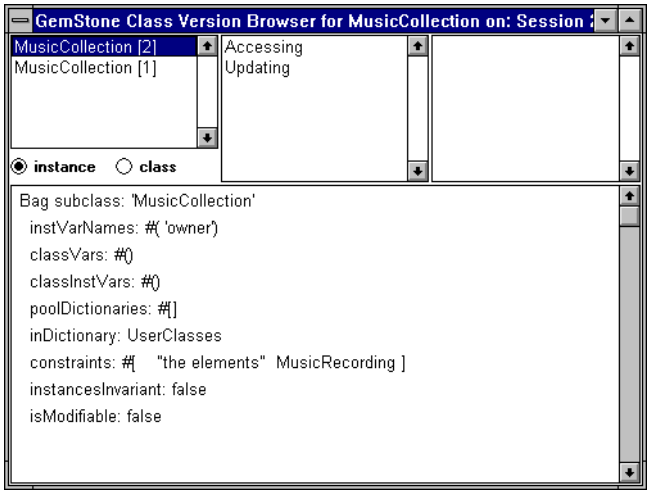
7.3 The Class Version Browser

The Class Version Browser is a specialized Class Browser that can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

To open a Class Version Browser, select a class in a browser and choose **spawn versions** from the Classes menu. If more than one version of a class has been created, the class list in the spawned browser displays the version number next to the class name.

A Class Version Browser is shown in Figure 7.1.

Figure 7.1 The Class Version Browser



Menus in the Class Version Browser

For the most part, the Class Version Browser's menus are the same as the menus in the Class Browser. However, the Class Version Browser's Classes menu contains the additional items **Inspect Instances** and **Migrate Instances**. Also, note that the Classes menu items **move to...** and **remove...** behave differently in this browser.

The layout of the browser is similar to the Class Browser. The Method Category and Message menus are the same as in a spawned Class Browser. The Classes menu, however, has additional functionality.

The commands available in the Class Version Browser are shown in Table 7.1

Table 7.1 Class Menu in Class Version Browser

file out as...	Writes GemStone Smalltalk code defining the selected class version and all of its methods to be written to a file in Topaz format. The class and its methods can later be re-created (read from the file and recompiled) by means of a command given from the File List Browser. See "Saving Class and Method Definitions in Files" on page 8-28.
file out methods as...	Writes GemStone Smalltalk code defining the selected class version's methods to be written to a file in Topaz format. See "Saving Class and Method Definitions in Files" on page 8-28 for more information on filing out.
spawn	Spawns a browser that includes only the selected class version.
spawn hierarchy	Spawns a browser that includes superclasses and subclasses of the selected class version.
spawn versions	Spawns another Class Version Browser.

hierarchy	Lists the superclasses of the current class. For example, if the current class is WriteStream, the hierarchy list will appear as follows:
	`Object`
	`Stream`
	`PositionableStream`
	`('` *itsCollection*`' '` *position*`')`
	`WriteStream`
	Any instance variable names declared in a class appear in the hierarchy list in parentheses. The preceding example shows PositionableStream to have two instance variables.

definition	Displays the definition (that is, the subclass creation message) of the currently selected GemStone class version.

Table 7.1 Class Menu in Class Version Browser (Continued)

move to...	Moves the selected class version to another class history. Prompt for a target class, adds the selected version to the target class's class history, and updates the browser. The class name of the selected version is changed to that of the target class.
remove...	Remove the selected class version from the class history. Upon confirmation to proceed, asks if the user wants to migrate instances. If yes, prompts for the migration target, migrates the instances and updates the browser.
inspect instances	Open an inspector on instances on the selected version.
migrate instances	Migrate all instances of the selected versions. Prompts you to select which version to migrate to. The user can only migrate to another version of the same class history, so if all versions are selected there is no migration destination and the item should be grayed out. Otherwise, prompt for the version to migrate to by popping up a list of versions not selected. Allow the user to cancel the operation by clicking a cancel button.
create access	Creates methods for accessing and updating the instance variables of the currently selected class version.
create in ST	Generate the selected class in client Smalltalk, if a mapping doesn't already exist. If it does exist, executing this menu item has no effect.
compile in ST	Attempts to compile all methods (instance and class) of selected class version in corresponding client Smalltalk class.

Chapter

8 Using the GemStone Programming Tools

This chapter introduces and describes the tools available in the GemBuilder programming environment.

The GemStone Launcher
introduces the tools and options available from the GemStone Launcher.

GemStone Browsers and Programming Tools
describes the various GemStone browsers, inspectors, and workspaces.

Creating Classes and Methods
explains how to use the GemStone tools to create GemStone classes and methods.

Using the GemStone Debugging Tools
describes the GemStone Debugger, setting breakpoints, and using the Breakpoint Browser.

Interrupting GemStone Execution
describes how to interrupt and halt execution of GemStone Smalltalk code.

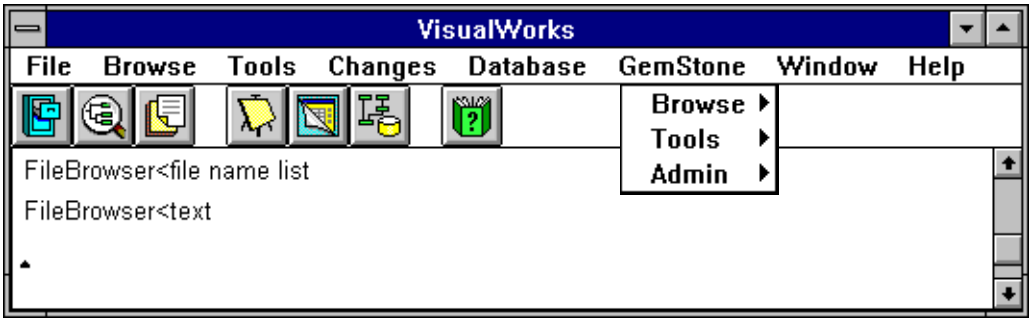
8.1 The GemStone Launcher

Depending on how your image was configured when GemBuilder was installed into your VisualWorks image, you will either have a modified VisualWorks Launcher that contains additional selections for bringing up GemStone tools and performing GemStone operations, or you will have both a VisualWorks Launcher and a stand-alone GemStone Launcher. In either case, you will have access to the various GemStone browsers and tools that are described in this chapter.

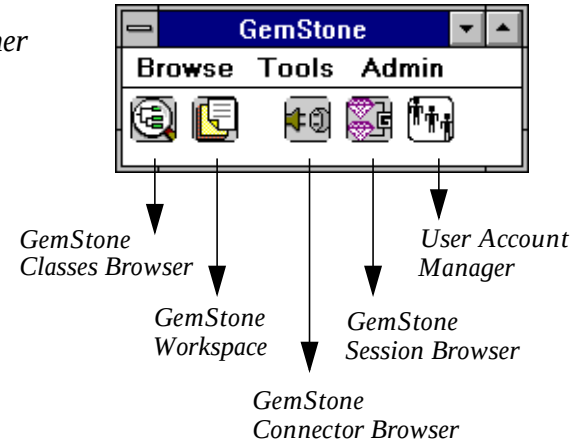
The GemStone commands available from your Launcher are shown in Figure 8.1.

Figure 8.1 The GemStone Launcher

VisualWorks Launcher with GemStone Menu



Standalone GemStone Launcher



GemStone Menus

The GemStone Launcher has three basic menus: **Browse**, **Tools**, and **Admin**.

Selecting **Browse** gives you access to the following options:

All Classes — Opens a GemStone Classes Browser. As in VisualWorks, the Classes Browser is GemStone's principal programming tool. It allows you to browse through the system's GemStone Smalltalk code, and it allows you to edit, copy, paste, and compile selected portions of it.

Dictionary... — Prompts you to select a dictionary from your Symbol List, then opens a Dictionary Browser, which is a subset of the Classes Brower showing a designated dictionary from your symbol list.

Class Named... — Opens a Browser focused on a particular class.

Senders Of... — Opens a Method List Browser listing all methods that send a specified message. See Figure 8.7 on page 8-19.

Implementors Of... — Opens a Method List Browser listing of all the classes that implement a specified method. Select the class to see its implementation of the method. See Figure 8.7 on page 8-19.

Breakpoints — Opens a Breakpoint Browser, which is a debugging tool that lets you set, clear, and examine breakpoints in GemStone methods. The Breakpoint Browser is described in this chapter.

Settings — Opens a Settings Browser in which you can examine, change, and store parameters for configuring GemBuilder. The Settings Browser is described in Chapter 9.

Selecting **Tools** gives you access to the following tools:

Connector Browser — Opens a Connector Browser, which is a tool you can use to examine, create, and remove connectors and the objects that they resolve to. Connectors are explained and the Connector Browser is described in Chapter 3.

Session Browser — Opens a Browser that lets you control your Gem Sessions. You can use the Session Browser to log in and out of GemStone, edit, copy, or remove a session's parameters, make a session the current session, and commit and abort transactions. The Session Browser is described in Chapter 2.

Segment Tool — Opens a Segment Tool that allows you to control authorization at the object level by assigning objects to segments. The Segment Tool is described in Chapter 6.

Workspace — Opens a GemStone Workspace, in which you can send GemStone Smalltalk code to GemStone for execution or execute client Smalltalk code.

System Workspace — Opens a GemStone workspace that contains examples of and templates for useful GemStone Smalltalk and client Smalltalk expressions.

Selecting **Admin** provides the following menu options:

GemStone Users — Opens a User Account Manager, which is a list of GemStone users. This tool allows you to create new users, assign attributes to them, and manage user accounts, provided you have the privileges to do so. The User Account Manager is described in Chapter 6.

Symbol Lists — Opens a Symbol List Browser, which allows you to examine and modify symbol dictionaries and their entries. The Symbol List Browser is described in Chapter 6.

This chapter explains how to use these applications and the other interactive GemStone programming environment tools, including the GemStone Debugger and the GemStone inspector. The GemStone Debugger is a window that allows you to examine and change the state of objects when GemStone execution has halted because of errors or breakpoints. You can continue execution either normally or by stepping through message-sends. The GemStone inspector is a window that lets you examine and modify components of GemStone objects.

8.2 GemStone Browsers and Programming Tools

This section describes the browsers and programming tools that are available within the Gemstone programming environment and explains the operations that are available in their respective menus. The GemStone debugging tools are discussed later in this chapter, in “Using the GemStone Debugging Tools” on page 8-31.

The GemStone Workspace

To bring up a GemStone workspace, select **Workspace** from the GemStone **Tools** menu or click on the GemStone Workspace icon (Figure 8.2).

Figure 8.2 GemStone Workspace Icon

The GemStone workspace provides the same commands as the VisualWorks workspace. In addition, the GemStone workspace menu offers the commands listed in Table 8.1.

Table 8.1 GemStone Workspace Menu Commands

GS-do it	Sends the selected GemStone Smalltalk expression (or sequence of expressions) to GemStone to be compiled and executed. Unlike **GS-print it,** the execution result is not displayed. This command is similar to **do it** in client Smalltalk.
GS-print it	Sends the selected GemStone Smalltalk expression (or sequence of expressions) to GemStone to be compiled and executed and displays a textual representation of the result. This command is similar to **print it** in client Smalltalk.
GS-inspect	Executes the selected GemStone Smalltalk expression (or sequence of expressions) in GemStone, then opens a GemStone inspector on the result.

GemStone Inspectors

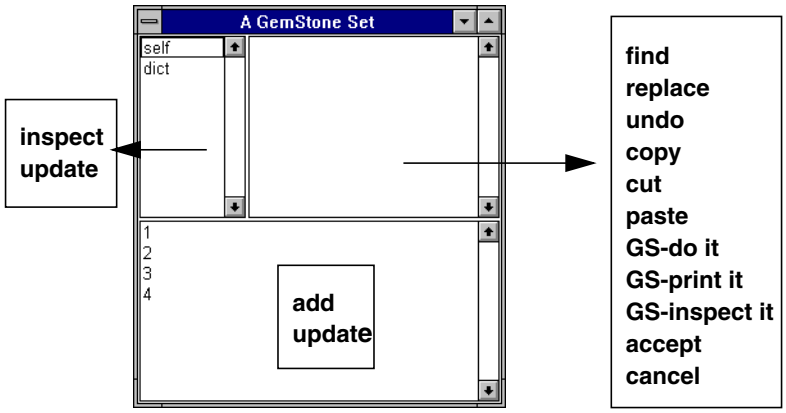
When you select a GemStone Smalltalk expression in a GemStone workspace and execute the **GS-inspect** command, the GemStone programming environment opens a GemStone inspector, a window whose panes allow you to examine and modify the values of instance variables in the GemStone object returned as the expression's result. As in the client Smalltalk inspector, the pane on the left lists `self`, which refers to the inspected object, together with the names or indexes of the object's instance variables. (If the object is a dictionary, the pane on the left lists the keys). When you select an item in the pane on the left, the value of that item is displayed in the pane on the right.

In the right-hand pane, you can execute GemStone Smalltalk expressions that contain the names that appear in the left pane. For example, if the object under inspection had an instance variable named `count` with a value of 5, then the expression `count * 5` executed in the right-hand pane of the inspector would return the value 25.

To replace the value of an item shown in the left pane, select the item, then type a GemStone Smalltalk expression in the right pane. Select **accept** from the *operate* menu. The selected item in the left pane will be given the value of the expression. Note that you cannot replace `self` in this way.

When you inspect a Bag, Set, or other nonsequenceable collection in GemStone, your inspector has three panes. The elements of the nonsequenceable collection are displayed in the bottom pane, as shown in Figure 8.3.

Figure 8.3 Inspecting a Nonsequenceable Collection in GemStone



The menu in the left pane of the standard inspector contains the following items:

Table 8.2 GemStone Inspector Menu Commands

inspect	Opens a GemStone inspector on the selected item. If that item is a Dictionary key, **inspect** opens an inspector on the object associated with that key. If the item is an instance variable name or index, it opens an inspector on the corresponding object.
update	Updates the inspector to include any changes made from within another window.

When you're inspecting a nonsequenceable collection, the bottom pane's pop-up menu displays the following additional commands:

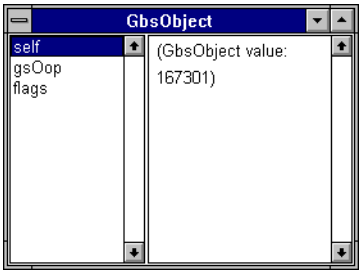
remove	*Appears only when an item is selected.* If you've selected an indexed variable, removes that element from the collection. If you've selected a key, removes the association with that key from the dictionary.
add	For a dictionary, prompts you for a new key, then adds an association with the given key and nil value to the dictionary. For a Bag or Set, your response is simply added to the collection.

If an object being inspected has more than 100 indexed instance variables, it will exceed the default initial display and **more** *appears as a menu option.*

more	Doubles the number of elements displayed in the bottom pane. Each successive selection doubles again until all indexed instance variables are shown.

Holding down a shift key while opening an inspector on a GemStone object causes a VisualWorks inspector to be opened on the proxy itself (Figure 8.4).

Figure 8.4 GemStone Proxy Inspector



The GemStone Classes Browser

The GemStone Classes Browser is very similar to the client Smalltalk classes browser in its appearance and functionality. You can use this browser to add classes and methods to the repository. The *GemStone Programming Guide* provides

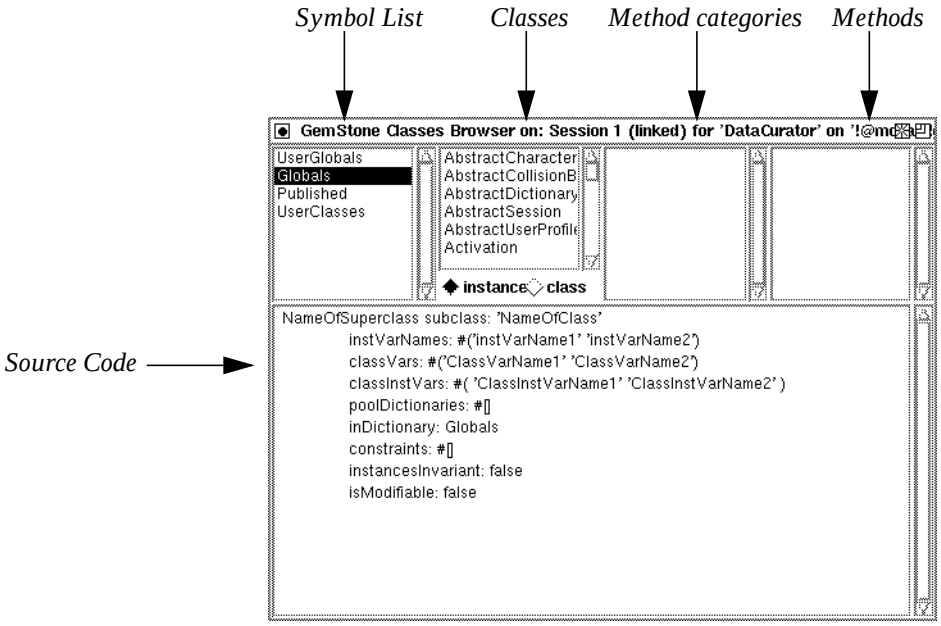
full information about each of the GemStone kernel classes that you can examine through the GemStone browser.

Select **Browse > All Classes** from the GemStone menu to open a GemStone Classes Browser, or click on the GemStone launcher's Classes Browser icon (Figure 8.5).

Figure 8.5 Classes Browser Icon

Figure 8.6 shows the GemStone Classes Browser.

Figure 8.6 The GemStone Classes Browser



As explained in previous chapters, GemStone allows concurrent access to shared objects by allowing users to share dictionaries that contain classes and global

variables. Each GemStone user has an object called a UserProfile, and each UserProfile contains a list of dictionaries called a *symbol list*. The **Symbol List pane** appears in the GemStone Browser's upper left corner and displays the names of all symbol dictionaries you can access. When you select a symbol dictionary, all classes defined in that dictionary appear in the adjacent Classes list. This pane corresponds to the VisualWorks browser's Class Category pane.

A Globals dictionary is present in every user's symbol list. This enables all users to have access to the GemStone kernel classes.

The **Classes pane** is to the right of the symbol list. When you select a class, all categories of that class's instance methods (or class methods) are listed. This pane corresponds to the VisualWorks browser's Class pane.

As in the VisualWorks browser, you can choose between a list of **instance** method categories or **class** method categories. By default, **instance** is selected.

The **Method Category pane** is to the right of the Classes pane. When you select a category from this list, all methods within that category appear in the Method list. This corresponds to the VisualWorks browser's Protocol pane.

The **Methods pane** is to the right of the Method Category list. When you select a method, its method is displayed in the lower portion of the window. This corresponds to the VisualWorks browser's Method Selectors pane.

The GemStone Smalltalk code itself is displayed in the **Source Code pane**, which occupies the lower half of the GemStone browser window. In this pane, you can edit and recompile the text of the displayed method, and set breakpoints in the method, or execute fragments of GemStone Smalltalk code as in a GemStone workspace.

NOTE:
You can only modify methods for which you have write authorization — for example, methods that you have written for your own classes. ***You cannot modify any GemStone kernel class method*** *— that is, any method that is defined for one of the predefined classes supplied with the GemStone system.*

The following pages describe the commands available from each of the GemStone Classes Browser's pane menus.

Symbol List Pane

This section explains how to define a new GemStone class and describes each of the menu commands available in the Symbol List pane.

In VisualWorks, classes are categorized in a System Organizer. The collections within the VisualWorks System Organizer contain only class objects. In GemStone, however, classes and other objects are collected into and given symbolic names by *symbol dictionaries*. These symbol dictionaries are used as a mechanism for both sharing and protecting objects in a multiuser environment, and, as such, they classify classes and objects in the GemStone Browser. (For more information about symbol dictionaries in GemStone see the *GemStone Programming Guide*.

To add the definition of a new GemStone class to a symbol dictionary, select the dictionary in which the definition of the new class will be stored. (Make sure that no classes are selected in the Class list.)

The browser responds by displaying the following template:

```
NameOfSuperclass subclass: 'NameOfClass'
  instVarNames: #('instVarName1' 'instVarName2')
  classVars: #('ClassVarName1' 'ClassVarName2')
  classInstVars #('ClassInstVarName1' 'ClassInstVarName2')
  poolDictionaries: #[]
  inDictionary: aDictionary
  constraints: #[]
  instancesInvariant: false
  isModifiable:false
```

For instructions on how to complete this template and add the new class definition to your symbol list, see "Adding a New Class" in the discussion of the Class menu, later in this chapter.

The menu options available in the Symbol List pane shown in Table 8.3.

Table 8.3 Symbol List Menu in GemStone Browser

file out as...	*Appears only when a dictionary is selected.* Writes the classes in the selected symbol dictionary to a file in GemStone fileout format. See "Saving Class and Method Definitions in Files" on page 8-28 for more information on filing out.
file out methods as...	*Appears only when a dictionary is selected.* Writes GemStone Smalltalk code defining all the methods in all classes in the selected symbol dictionary. The information is written to a file in GemStone fileout format. See "Saving Class and Method Definitions in Files" on page 8-28 for more information on filing out.

Table 8.3 Symbol List Menu in GemStone Browser (Continued)

spawn	*Appears only when a dictionary is selected.* Creates a browser for the selected dictionary.
inspect	*Appears only when a dictionary is selected.* Allows you to inspect the objects in your dictionaries.
add...	Allows you to create a new symbol dictionary and add it to your symbol list. You will be prompted for the name of the new symbol dictionary. If an existing entry is selected, the new symbol dictionary is added above the selection. If no entry is selected, the new symbol dictionary is added at the end of the list.
rename as...	*Appears only when a dictionary is selected.* Changes the name of the currently selected symbol dictionary. Prompts for new dictionary name. A notifier will inform you if there is already an object with that name in the repository.
remove...	*Appears only when a dictionary is selected.* Removes the currently selected symbol dictionary from your symbol list. This may cause objects contained in that dictionary to become unreachable. If the dictionary contains any objects other than class objects, a prompter will require you to confirm their removal as well.

CAUTION:
Do not attempt to remove the Globals dictionary. That dictionary contains the definitions of all kernel classes.

update	Refreshes the GemStone browser. You can execute this command to resynchronize the GemStone browser with your GemStone transaction. This command is useful when you have modified the repository from a GemStone workspace or inspector, or from other tools that use GemBuilder.

Table 8.3 Symbol List Menu in GemStone Browser (Continued)

commit	Attempts to commit modifications to the repository that occurred during the current GemStone transaction (that is, since the last transaction commit, abort, or login). After you have committed the transaction, you will be connected to the most recently committed version of the repository and you can continue your work in the GemStone browser. This command can also be executed from the Session Browser by clicking on the button labeled **Commit...**.

NOTE:
Commit your work before logging out. If you have performed work without committing it to the permanent repository, the uncommitted work is lost when you exit client Smalltalk.

abort	Undoes all changes that you have made in the repository since the beginning of the transaction. You are asked to confirm this choice. This command can also be executed by clicking on the button labeled **Abort...** in the Session Browser. After you have aborted the transaction, your GemBuilder session views the most recent version of the repository, and you can continue your work. You can choose **abort** at any time to ensure that you are connected to the most recent version of the repository.
find class...	Prompts for a class to search for. This command accepts a wild card character (*).

Class Pane

The menu commands available in the GemStone Browser's Class Pane are listed in Table 8.4. This section describes each of those commands, and subsequent sections discuss the procedure required to add the definition of a new GemStone class to the currently selected symbol dictionary.

A Class pane menu appears only when a class has been selected.

Table 8.4 Class Menu in GemStone Browser

file out as...	Writes GemStone Smalltalk code defining the selected class and all of its methods to be written to a file in GemStone fileout format. The class and its methods can later be re-created (read from the file and recompiled) by means of a command given from the File List Browser. See "Saving Class and Method Definitions in Files" on page 8-28.
file out methods as...	Writes GemStone Smalltalk code defining the selected class's methods to be written to a file in GemStone fileout format. See "Saving Class and Method Definitions in Files" on page 8-28 for more information on filing out.
spawn	Spawns a browser that includes only the selected class.
spawn hierarchy	Spawns a browser that includes superclasses and subclasses of the selected class.
spawn versions	Spawns a class version browser that shows all versions of the selected class.

hierarchy	Lists the superclasses of the current class. For example, if the current class is WriteStream, the hierarchy list in the text pane will appear as follows:
	`Object`
	`  Stream`
	`  PositionableStream`
	`  ('` *itsCollection*`' '` *position*`')`
	`  WriteStream`
	Any instance variable names declared in a class appear in the hierarchy list in parentheses. The preceding example shows PositionableStream to have two instance variables.

definition	Displays the definition (that is, the subclass creation message) of the currently selected GemStone class in the text pane.
comment	Displays the comment text associated with the currently selected GemStone class in the text pane.

Table 8.4 Class Menu in GemStone Browser (Continued)

move to...	Moves the selected class definition to another dictionary in your symbol list. A list of dictionaries appears. Select the one you wish to move the class into, or cancel the move by selecting **Cancel**.
remove...	Removes the current class, along with all its categories and methods, from the dictionary where the class is stored. (However, instances of the class are not removed from the repository.) A prompter will require you to confirm this choice.
create access	Creates methods for accessing and updating the instance variables of the currently selected class.
create in ST	Generate the selected class in client Smalltalk, if a mapping doesn't already exist. If it does exist, executing this menu item has no effect.
compile in ST	Attempts to compile all methods (instance and class) of the current class in the corresponding client Smalltalk class. If an error occurs during compilation, a browser containing the failed method is displayed for correction.

Method Category Pane

The browser's Method Category menu commands allow you to add, file out, remove, or rename method categories. It also provides an option for compiling GemStone methods in client Smalltalk. The following section describes these commands.

Until you've selected a method category, **add...** and **find method** are the only menu items. The Method Category menu commands are shown in Table 8.5.

Table 8.5 Method Category Menu in GemStone Browser

file out as...	Writes the source code for methods in the selected category to a file in GemStone fileout format. The methods can later be filed in (read from the file and recompiled) via a command given from the File List browser.
add...	Prompts for the name of a new method category.
rename as...	Prompts for a new name for the selected category.

Table 8.5 Method Category Menu in GemStone Browser (Continued)

remove...	Removes the selected method from the current class. A prompter requires you to confirm this choice.
compile in ST	Attempts to compile the selected method in the corresponding client Smalltalk class. If the class does not exist in client Smalltalk, it is created.
compile in ST	*Enabled only when a method category is selected.* Attempts to compile all methods in the selected category in the corresponding client Smalltalk class. If the client Smalltalk class does not exist, it is created.

Method Pane

The Method menu appears only when a method has been selected. This menu allows you to file out a single method, reorganize methods into different categories, remove a method, set a breakpoint within a method, and compile a method in client Smalltalk.

The commands available from the Method menu are shown in Table 8.6.

Table 8.6 Method Menu in GemStone Browser

file out as...	Writes the source code for the selected method to a file in GemStone fileout format. The method can later be filed in (read from the file and recompiled) via a command given from the File List browser. See "Saving Class and Method Definitions in Files" on page 8-28 for a more thorough discussion.
spawn	Opens a method list browser on the selected method.
senders	Opens a method list browser on all methods in all classes that send the message whose selector is that of the currently selected method.
implementors	Opens a method list browser on all methods of any class that implement the currently selected message selector.
messages...	Offers a menu of all messages used in the source code for the currently selected method. Select one to open a method list browser on all implementors of that message selector.
move to...	Moves the selected method to another category. You will be prompted for the name of the destination category. If that method category does not yet exist, it will be created for you.

Table 8.6 Method Menu in GemStone Browser (Continued)

remove...	Removes the selected category and all of its methods from the repository. A prompter requires you to confirm this choice.
find method...	Initiates a search for a method by presenting a scrollable list of selectors known to the currently selected class. To search for a method that does not appear on the list, use the Browse Implementors tool from the main GemStone menu.
compile in ST	Attempts to compile all methods in the selected category in the corresponding client Smalltalk class. If the class does not exist in client Smalltalk, it is created. If an error occurs during compilation, a browser containing the failed method is displayed for correction.

Private Methods

Most of the methods for GemStone kernel classes are *public*; that is, they are documented in the *GemStone Kernel Reference* and are available for you to use. In addition, system developers have implemented *private* methods to support the public protocol. Unlike the public protocol, private methods are not supported by GemStone Systems, Inc.; they are implementation-dependent and are thus subject to change.

By convention, the selectors of all private methods for the GemStone kernel classes begin with an underscore character (_). For example, see Objects's instance method `_alias` (in the "Accessing" category).

Because GemStone is an open system, private methods are listed along with the public methods for the associated category. As with public methods, you can display the text of private methods; however, while it may be interesting to review the definitions of private methods, remember that they are subject to change at any time.

NOTE:
Do not depend on the implementation or existence of private methods when creating your own methods.

Source Code Pane

Like the client Smalltalk browser, the GemStone Browser's Source Code pane has a pop-up menu that contains commands for editing, executing expressions, compiling code, and inspecting objects.

In the text pane of a GemStone browser, when executing GemStone Smalltalk code self refers to the currently selected class, or nil if no class is selected.

The operations available from this menu are described in Table 8.7

Table 8.7 Browser's Source Code Pane Menu Commands

find...	Finds the next occurrence of the string typed into the prompter.
replace...	Brings up a dialog in which you can specify a string to search for and a string with which to replace it.
undo	Reverses the most recent editing operation.
copy	Copies the selected text to the clipboard.
cut	Deletes the selected text and places it on the clipboard.
paste	If text is selected, it is replaced by the text on the clipboard. If no text is selected, the contents of the clipboard are inserted at the location of the cursor.
do it	Executes the selected code in client Smalltalk.
print it	Executes the selected code and displays the result.
inspect	Executes the selected client Smalltalk code in client Smalltalk and opens an inspector on the result.
accept	Compiles the contents of the pane, in GemStone, as a new method or class (depending on which is displayed).
cancel	Restores the text in the Source Code pane to the state it was in when it was last compiled.
GS-do it	Sends the selected code to GemStone, where it is compiled and executed. Unlike **GS-print it,** the result is not displayed. This command is similar to **do it** in Smalltalk.
GS-print it	Sends the selected code to GemStone, where it is compiled and executed. Displays a textual representation of the result. This command is similar to **print it** in client Smalltalk.
GS-inspect	Executes the selected GemStone Smalltalk code in GemStone, then opens a GemStone inspector on the result. This command is similar to **inspect** in the client Smalltalk.
set break	Sets a breakpoint for debugging. See"Debugging Commands in the GemStone Browser" on page 8-35

GemStone Class, Dictionary, and Method List Browsers

GemStone also provides Browsers that focus on a particular class, a dictionary in your symbol list, and on the location of methods that implement particular methods. These Browsers are subsets of the GemStone Classes Browser and are shown in Figure 8.7.

These browsers are selections from the **GemStone > Browse** menu:

Browse > Dictionary...
Displays a list of known dictionaries, then opens a dictionary browser on your selection.

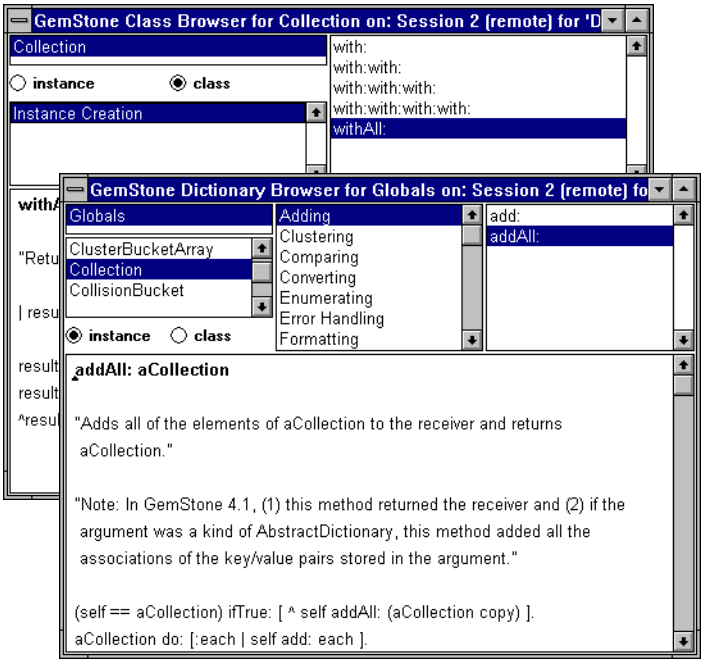
Browse > Class Named...
Prompts for a class name, then opens a class browser on your selection. The string you enter may contain wild card characters. If more than one class matches your string, GemBuilder presents a list of classes from which to choose.

Browse > Senders Of...
Prompts for a message selector, then opens a two-paned method browser. The upper pane shows methods that send the message you specified. When you select one of these methods, its source code is displayed in the browser's lower pane.

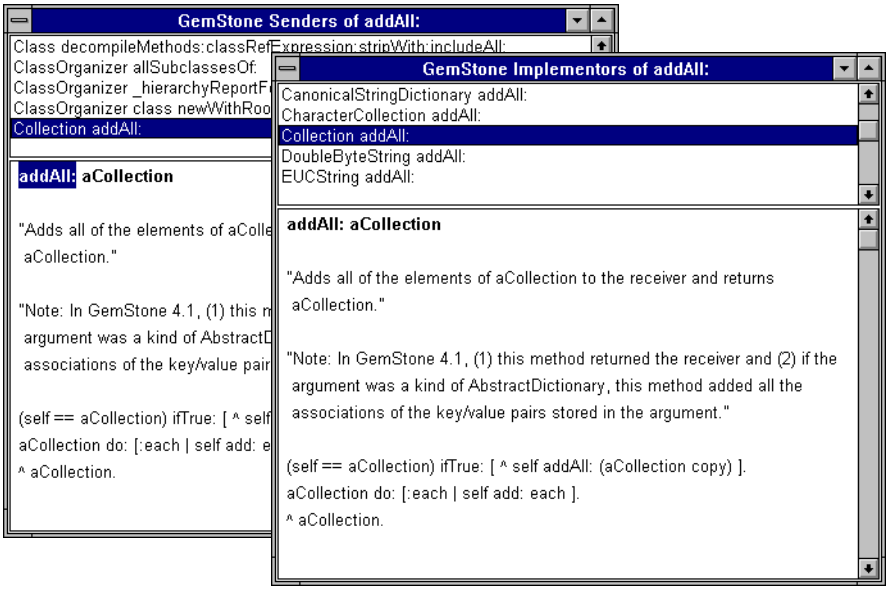
Browse > Implementors Of...
Prompts for a message selector, then opens a two-paned method browser. The upper pane shows classes that implement the message you specified. When you select one of these classes, the browser's lower pane shows the source code for that class's implementation of the specified method.

Figure 8.7 Other GemStone Browsers

Class and Dictionary Browsers



Method Browsers on senders (references to) and implementors of a selector



8.3 Creating Classes and Methods

Using GemBuilder to create classes and methods is very similar to using the client Smalltalk browsers and tools. However, because GemStone classes are persistent and shared, you need to take into consideration a few more things when creating them.

What You Need to Know About GemStone Classes

This section covers what you need to remember as you begin to create classes in GemStone Smalltalk and add them to the GemStone system; however you should refer to the *GemStone Programming Guide* for a detailed explanation of class creation.

Instance Variables Can Be Constrained

To speed up GemStone's indexed associative access for efficient querying, you can constrain certain instance variables to contain only specified kinds of objects.

Constraining a variable's kind ensures that the variable will always contain either `nil` or instances of a certain class or that class's subclasses. This lets GemStone know what kind of object to expect. When you constrain an instance variable to be a kind of array, you are guaranteed that it will always be an instance of Array, a subclass of Array, or `nil`.

Constraints can be *circular*. This means that you can constrain an instance variable to hold instances of its own class. You can also constrain instance variables of two classes to each hold instances of each other.

Constraints Are Inherited by Subclasses

Constraints, like instance variables, are inherited by subclasses. This means that if you create a subclass of a class, its inherited instance variables will, by default, bear the same constraints as specified in the parent class.

Inherited instance variables can be further constrained — that is, made more restrictive — in a subclass. In this case, the instance variable's new constraint must be a subclass of the inherited constraint.

To make inherited constraints more restrictive in a new subclass, name the inherited instance variables and their new constraints in the `constraints:` argument. For example, you can create a subclass of Employee named FormerEmployee, add an instance variable named `termDate`, and constrain your subclass's inherited `jobTitle` and `department` instance variables (constrained in Employee to be Strings) to be InvariantStrings. FormerEmployee's other

inherited instance variables will retain the constraints that were set in their parent class, Employee.

Instances Can Be Made Invariant

The definition of the class can specify that all instances of the class are invariant by setting `instancesInvariant:` to `true`. An invariant object can be modified only during the transaction in which it is created. When the transaction is committed, no further modifications can be made to any of its instance variables, nor can the size or class of the object be changed. Class-level invariance is useful for supporting literals in methods and in other limited situations, but it is generally more cumbersome than object-level invariance, which is preferred.

Any object can be made invariant by sending it the message `immediateInvariant`. This mechanism essentially write-protects certain objects and is useful for maintaining the integrity of your repository. Once `immediateInvariant` is sent to an object, no modifications can be made to any of the object's instance variables, nor can the size or class of the object be changed. You cannot reverse the effect of the `immediateInvariant` message. The message `isInvariant` returns `true` if the receiver is invariant; `false` otherwise.

Defining a New Class

To define a new GemStone class, go to the Symbol List pane and select the dictionary in which you want the definition of the new class to be stored. Make sure no classes are selected in the class list when you do this.

The browser responds by displaying the following template:

```
NameOfSuperclass subclass: 'NameOfClass'
  instVarNames: #('instVarName1' 'instVarName2')
  classVars: #('ClassVarName1' 'ClassVarName2')
  classInstVars #('ClassInstVarName1' 'ClassInstVarName2')
  poolDictionaries: #[]
  inDictionary: aDictionary
  constraints: #[]
  instancesInvariant: false
  isModifiable:false
```

This is the basic form of the subclass creation message in GemStone Smalltalk. To create a new class, you must provide the following:

- *NameOfSuperclass* — Replace this with the name of the class's immediate superclass. You cannot create subclasses of certain GemStone kernel

classes. A note to that effect is included in the descriptions of those classes in the *GemStone Kernel Reference*.

- *NameOfClass* — Replace this with the name of the new class. By convention, the first character in each GemStone class name is capitalized.
- *instVarNames*—names of any instance variables (`instVarNames:`). A class can define instance variables to which you can refer by name in the code for that class. A named instance variables is a variable whose name is shared by all instances of a class and all instances of its subclasses. Each instance, however, holds a distinct value for the variable. Instance methods of a class and its subclasses can refer to its named instance variables, but class methods cannot. Remove the placeholders if you are not specifying instance variables for the new class.
- *classVarNames*—names of any class variables. A class variable is a variable whose name and value are shared by a class, all of its instances, its subclasses, and all of their instances. Both class and instance methods of the class and its subclasses can refer to the variable. Remove the placeholders if you are not specifying class variables.
- *classInstVarNames*—names of any class instance variables. A class instance variable is a variable whose name is shared by a class and all of its instances. Subclasses and all of their instances inherit the variable's name but not its value. Only class methods of a class and its subclasses can refer to class instance variables. Class instance variables are useful when a class and its subclasses need to share the same structure, but not the same value, for a variable. Remember to remove the placeholders if you are not specifying class variables.
- `poolDictionaries:` — These are special-purpose storage structures that enable any arbitrary group of classes and their instances to share information.
  Classes can share a pool dictionary so that methods defined in those classes can refer to the variables defined in the pool dictionaries. List any pool dictionaries that you want the class to access.
- `inDictionary:` —the name of the dictionary where the new class is to be stored. The currently selected dictionary is inserted in the template, and unless you specify otherwise, the new class will be stored in that dictionary.
- `constraints:` — List any instance variable constraints, if appropriate. Constraining a variable's kind guarantees that it will always contain either `nil` or instances of the constraint class or that class's subclasses.

- `instancesInvariant: false` — This parameter determines whether instances of the class may be changed. The default value is false.
- `isModifiable: false` — This parameter determines whether the structure of the class can be modified. The default value is false.

For example, consider the following definition of a class named Employee.

Example 8.1

```
Object subclass: 'Employee'
```
Employee is a direct subclass of Object.

```
instVarNames: #( 'name' 'empNum' 'jobTitle' 'department')
```
The Employee class has four named instance variables: `name`, `empNum`, `jobTitle`, and `department`.

```
classVars: #()
```
This class has no class variables.

```
classInstVars #()
```
This class has no class instance variables.

```
poolDictionaries: #()
```
This class need not access any pool dictionaries.

```
inDictionary: UserGlobals
```
Employee resides in the user's UserGlobals dictionary.

```
constraints:  #[ #[#name, String],
                 #[#empNum, SmallInteger],
                 #[#jobTitle, String],
                 #[#department, Department],
                 #[#address, String] ]
```
For efficient access, constraints have been placed on each instance variable: `name` must be an instance of String, `empNum` must be an instance of SmallInteger, `jobTitle` must be an instance of String, `department` must be an instance of Department, and `address` must be an instance of String.

```
instancesInvariant: false
```
When instances of the class are created, their values will be modifiable after they have been committed to the repository.

`isModifiable: true`

This class is modifiable; instance variables can still be added, removed, and constrained. However, as long as the class itself remains modifiable, no instances of it can be created.

Private Instance Variables

In addition to the private methods discussed earlier, you also see private instance variables occasionally in the GemStone kernel classes. For example, the GemStone Bag class defines four private instance variables that are used by the object manager and primitives to implement NSC semantics and indexes. The names of private instance variables are symbols beginning with an underscore (_). Private instance variables cannot be modified or constrained when defining subclasses. An attempt to place a constraint on a private instance variable generates an error.

Subclass Creation Methods

You can choose from a variety of subclass creation messages, depending on the type of class you want to create. For example, if you wish to create a byte subclass, replace the initial keyword `subclass:` with the keyword `byteSubclass:`. If the superclass is not a subclass of String, instances of the new class store and return SmallIntegers in the range 0 - 255.

Similarly, if you wish to create an indexable subclass, replace the initial keyword `subclass:` with the keyword `indexableSubclass:`. Instances of the new class are represented as pointer objects.

For complete descriptions of the different kinds of classes, see the section describing class storage format in the *GemStone Programming Guide*.

If you wish to set the class history of your new class explicitly, you can include the keyword `newVersionOf:` in any subclass creation message, after `instancesInvariant:` and before `isModifiable:`. If the argument to this keyword is a class, this method creates the new class as a new version of that class, and the two classes share a class history. In this way, you can make one class a new version of another even if they do not have the same name.

If the argument to the `newVersionOf:` keyword is `nil`, the new class is created with a new class history.

For more discussion of class histories, see the section on class histories in the *GemStone Programming Guide*.

Compiling the New Class Definition

After you have edited the class template to your satisfaction, choose **accept** from the Source Code pane menu. When you do this, GemStone's compiler will evaluate the entire contents of the browser's Source Code pane as a subclass creation message. (For this reason, it's important not to have any extraneous text in the Source Code pane when you choose **accept**.)

If an Error Occurs

If the class definition does not compile properly, an error message is inserted in the browser's Source Code pane. Delete the error message, then edit your subclass creation message and try to compile the class definition again. Alternatively, you can display the unmodified definition of the currently displayed class by choosing **cancel** from the Code pane menu.

If the new class object cannot be created due to a runtime error, an error notifier window is displayed. Edit the class creation message and try again, or choose **cancel**, as described above.

As we discussed in Chapter 7, GemStone supports modification of the schema when you already have instances of invariant classes populating your repository and you discover that you must redefine some of your classes.

Modifying an Existing Class

If you select an existing class, then modify and accept the class definition, you are creating a new version of the class and all of its subclasses. The browser attempts to recompile all methods from the previous version into the new version. Methods that fail to recompile are presented in a method list browser, from which you can correct the errors. If the class has subclasses, they are also versioned and their methods recompiled.

Versioning a class does not cause its instances to be migrated to the new class. They are still instances of the old class. You can migrate some or all instances of one version of a class to another version when you need to.

For more information on migrating instances, see the *GemStone Programming Guide*.

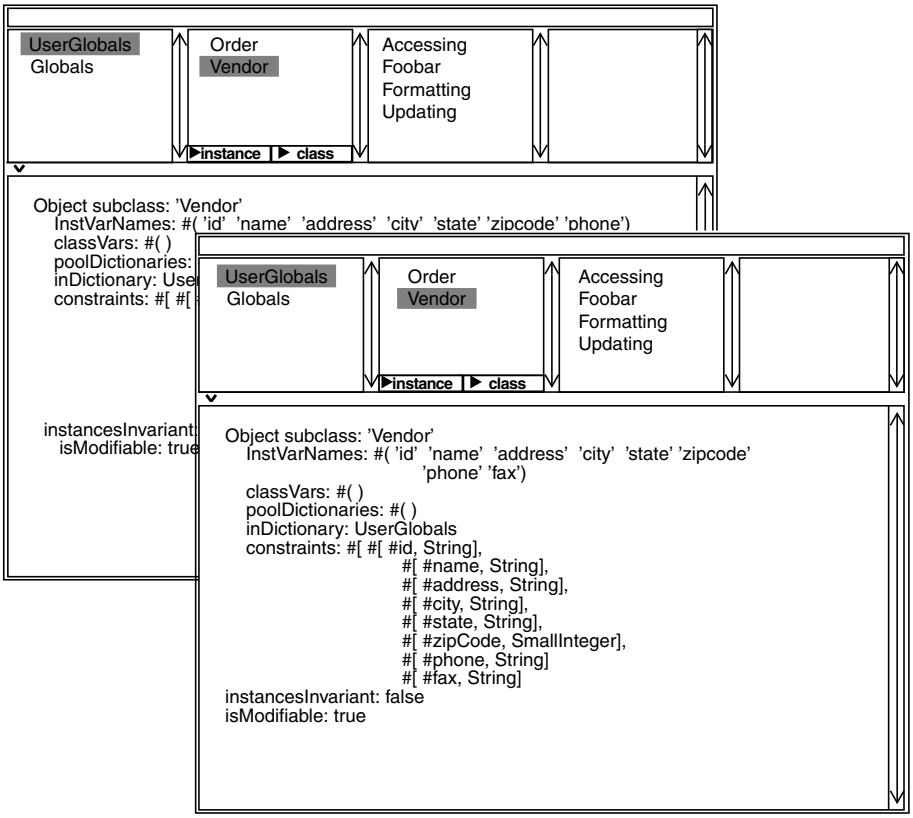
To create a new version of a class, select the class's name in the browser bring up its definition in the Source Code pane. Edit the definition and select **accept** from the pop-up menu in the Source Code pane. Whenever you create a class with the same name as a class that already exists in one of your symbol dictionaries, the new class is automatically created as the latest version of the existing class and it

automatically shares the same class history. Instances created after the redefinition have the new class's structure and access the new class's methods. Instances that were created earlier access the old class's methods, but they can be migrated to the new class.

Let's assume that you have a class named Vendor with instance variables for `id`, `name`, `address`, `city`, `state`, `zipCode`, and `phone`, and that the class is defined as shown in Figure 8.6. Five of Vendor's instance variables are constrained to be instances of class String, and one (zipcode) is constrained to be a SmallInteger. Suppose that you decide that the class needs an additional instance variable named `fax` to represent the Vendor's fax number.

To do this, you can define a new version of the class Vendor to include the new instance variable. Keeping the same name as the old class ensures that it shares the same class history as the previous version.

Figure 8.8 Creating a New Version of a Class



After you compile the class definition, the new class is named Vendor, and all of the original instance and class methods are copied to the new class. Any existing instances will still belong to the original class and may have to be migrated to the new class. (See “Instance Migration Within GemStone” on page 7-2.)

Defining Methods

To add a new method, select the category you want to put it in and select no methods. The following template is displayed for you:

```
message selector and argument names
"comment stating purpose of message"

| temporary variable names |
statements
```

Edit the template to fill in the suggested components. Argument names and temporary variables may not be needed. Comments are optional too, of course, but recommended.

After you have completed the template, choose **accept** from the Code pane menu. If the new method compiles successfully, it is inserted into the list of methods for the current category and class. Otherwise, the first compiler error encountered is inserted into the text at the point of error, and becomes the new text selection.

A small number of selectors are optimized by the GemStone Smalltalk compiler. Do not compile any method whose selector is one of the following:

```
and:                    notNil
class                   or:
downTo:by:do:           timesRepeat:
downTo:do:              to:by:do:
ifFalse:                to:do:
ifFalse:ifTrue:         untilFalse
ifTrue:                 untilTrue
ifTrue:ifFalse:         whileFalse:
isKindOf:               whileTrue:
isNil==
```

Saving Class and Method Definitions in Files

It's often useful to file out the GemStone Smalltalk source code for your classes and methods in order to store them in ordinary files. Such files make it easy to:

- transport your code to other GemStone systems,
- perform global edits and recompilations,
- produce paper copies of your work, and

- recover code that would otherwise be lost if you are unable to commit.

As noted earlier, the GemStone browser provides menu items for filing out methods and class definitions, while the File List browser provides the **GS-file in** menu item that you can use for reading and compiling (filing in) one of the resulting GemStone Smalltalk files.

Those files are written by GemBuilder as sequences of commands understood by Topaz, a command line-oriented GemStone programming environment. The following example shows a class definition in Topaz format.

Example 8.2

```
!
! From GEMSTONE: 5.0, Tue Jun  4 18:36:18 US/Pacific 1996;
!
! Class 'Address'
!
doit
Object subclass: 'Address'
             instVarNames: #( 'street' 'zip')
             classVars: #()
             poolDictionaries: #()
             inDictionary: UserGlobals
             constraints: #[#[#street, String],#[#zip,Integer]]
             isInvariant: false
%

!
! Instance Category 'Updating'
!
category: 'Updating'
method: Address
street: newValue
       "Modify the value of the instance variable 'street'."
       street:= newValue
%
method: Address
zip: newValue
       "Modify the value of the instance variable 'zip'."
       zip:= newValue
%
```

```
!
! Instance Category 'Accessing'
!
category: 'Accessing'
method: Address
street
      "Return the value of the instance variable 'street'."
      ^street
%

method: Address
zip
%
      "Return the value of the instance variable 'zip'."
      ^zip
%
```

GemBuilder's file out and file in facilities are intended mainly for saving and restoring classes and methods without manual intervention. If this is all you want to do, then you don't need to understand the Topaz commands involved. However, it is also possible to create customized files that include commands to commit transactions and to create and manipulate objects other than classes and methods. If you want to perform such tasks, you'll need a full description of the Topaz commands. See the *Topaz GemStone Programming Environment* for your system.

The file in mechanism can execute only the following Topaz commands:

```
category:          method
classmethod        method:
classmethod:       printit
commit             removeAllMethods
doit               removeAllClassMethods
```

GemBuilder acknowledges the presence of the following commands by writing to the System Transcript, but it cannot execute the commands:

```
display            omit
expectvalue        output
level              remark
limit              status
list               time
```

If GemBuilder encounters any other Topaz commands, it stops reading the file and displays an error notifier.

The file-in mechanism does not display execution results; instead, it just appends information to the System Transcript about the files it reads and the classes and categories for which it compiles methods.

Handling Errors While Filing In

If one of the modules (run commands or method definitions) that you're filing in contains a GemStone Smalltalk syntax error, GemBuilder displays a compilation error notifier that contains the erroneous module in a text editor. If you correct the error and then choose **accept**, GemBuilder performs the compilation again and then processes the remainder of the file.

In case of authorization problems, commands that the file-in mechanism doesn't recognize, or other errors, GemBuilder displays a simple error notifier (with no editor) and stops processing your file.

8.4 Using the GemStone Debugging Tools

In addition to the basic code development tools described previously in this chapter, GemBuilder also provides debugging facilities for GemStone Smalltalk code. These debugging facilities are provided by two special windows, the GemStone Breakpoint browser and the GemStone Debugger, and by some menu items in the GemStone browser code pane.

These debugging facilities allow you to:

- step through execution of a method examining the values of arguments, temporaries, and instance variables;
- set, clear, and examine GemStone Smalltalk breakpoints;
- inspect or change the values of arguments, temporaries, and receivers in any context on the virtual machine call stack, then continue execution from the top of the stack; and
- execute a GemStone Smalltalk expression within the scope of a given context.

Step Points and Breakpoints

For the purpose of determining exactly where a step will go during debugging, a GemStone Smalltalk method can be decomposed into *step points*. The locations of these step points determine where virtual machine breakpoints can be set.

Generally, step points correspond to message-sends, method returns and assignments.

In Example 8.4, step points are indicated by numbered carets.

Example 8.3

```
includesValue: anObject

"Return true if the receiver contains an object of the
same value as the argument, anObject.     Return false
otherwise.  (Compare with includes:, which is based on
identity.)"

| theSize |

theSize := self size.
^1      ^3      ^2
1 to: theSize do: [ :each |
  ^4
            ((self at: each) = anObject)
                   ^5       ^6
                     ifTrue: [ ^true ]
                        ^7         ^8
                   ].
^false
^9
```

Using the GemStone Debugger (described on page 8-38) to step through this method, the first step halts the virtual machine at the point where `size` is about to be sent to `self`. Another step causes that message to be sent and then halts the virtual machine just before the result was assigned to `theSize`.

Although the returns from this method are all explicit, there is always a step point before a method return even when the method does not include an explicit return statement. For example:

Example 8.4

```
add: firstNum to: secondNum
firstNum + secondNum.
      ^1
^2   "<----- Method return step point here"
```

As explained earlier, you can set a breakpoint at any step point. When a breakpoint is encountered during normal execution, a notifier appears.

The notifier's pop-up menu lets you choose **debug**, which opens a GemStone Debugger in which you can interactively explore the context in which execution halted. You can use the GemStone Breakpoint browser and the Debugger (described below) to set *breakpoints*. A breakpoint halts execution at a particular step point within a method. In general, you can choose to set a break before a message-send, an assignment, or a method return. The GemStone Browser, the GemStone Breakpoint Browser, and the GemStone Debugger can display the valid step points within a method for you.

Breakpoints for Primitive and Special Methods

Some special considerations apply in setting breakpoints for primitive and special methods.

If you set a breakpoint in a primitive method, the break is encountered only if the primitive fails. Consider the method below:

```
= aString

<primitive: 160>
self _primitiveFailed: #=
```

When this method is invoked, GemStone first executes the lower-level code in primitive 160. If that code executes successfully, the primitive is said to succeed, and the method returns a value. Because no GemStone Smalltalk code has yet been encountered, the virtual machine has not yet reached the first step point. Only if the primitive fails will the virtual machine send the message at the bottom of the method and thus possibly encounter a breakpoint.

Certain simple methods are optimized by the GemStone Smalltalk compiler and virtual machine in such a way that they contain no step points. Naturally, you cannot set a breakpoint if there are no step points. A method that performs only one of the following operations has no step points:

- return `true`,
- return `false`,
- return `nil`,
- return `self`,
- return the value of an instance variable,
- assign to an instance variable, or
- return a literal variable (a class or pool variable or one defined in a symbol dictionary such as UserGlobals).

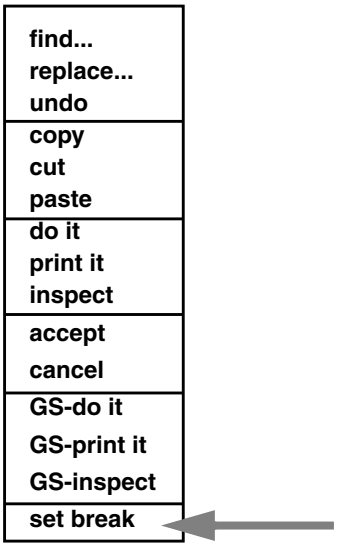
A method that does some computation and then performs one of the actions listed above contains step points; it is not a simple method for purposes of this discussion.

In addition to the special kinds of methods listed above, a handful of specific kernel class methods are specially optimized so that they cannot take breakpoints. Those methods are:

```
and:                    notNil
class                   or:
downTo:by:do:           timesRepeat:
downTo:do:              to:by:do:
ifFalse:                to:do:
ifFalse:ifTrue:         untilFalse
ifTrue:                 untilTrue
ifTrue:ifFalse:         whileFalse:
isKindOf:               whileTrue:
isNil                   ==
```

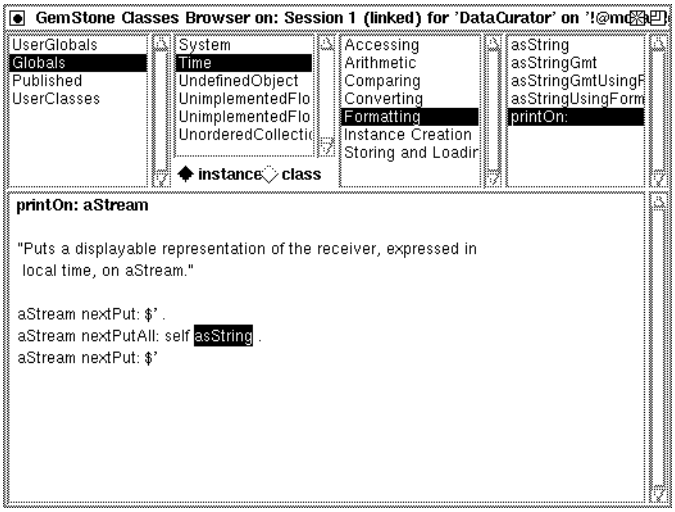
Debugging Commands in the GemStone Browser

Whenever a method that contains step points is being displayed, the Source Code pane's *operate* menu in the GemStone Browser includes the command **set break**.



The **set break** command causes a breakpoint to be set at the step point nearest the location of the cursor. If the cursor is not exactly at a step point, GemBuilder scans the method starting from the current cursor location, sets a breakpoint at the next step point it encounters, and highlights the breakpoint to indicate its location (Figure 8.9).

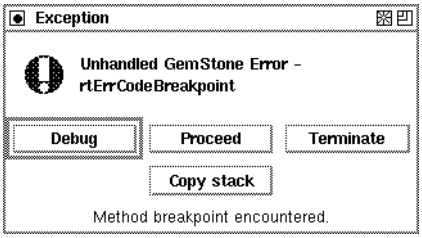
Figure 8.9 Classes Browser with Highlighted Breakpoint



If the code in the view has been altered but not recompiled, you are prompted to revert to the previously compiled version before executing **set break**.

When GemStone execution has been interrupted because of a run-time error, a breakpoint, an interrupt from the user, `Object>>halt`, or `Object>>pause`, GemBuilder displays a notifier (Figure 8.10).

Figure 8.10 Breakpoint Notifier



The notifier contains the buttons **Proceed, Debug, Terminate** and **Copy Stack**.

Table 8.8 Breakpoint Notifier Commands

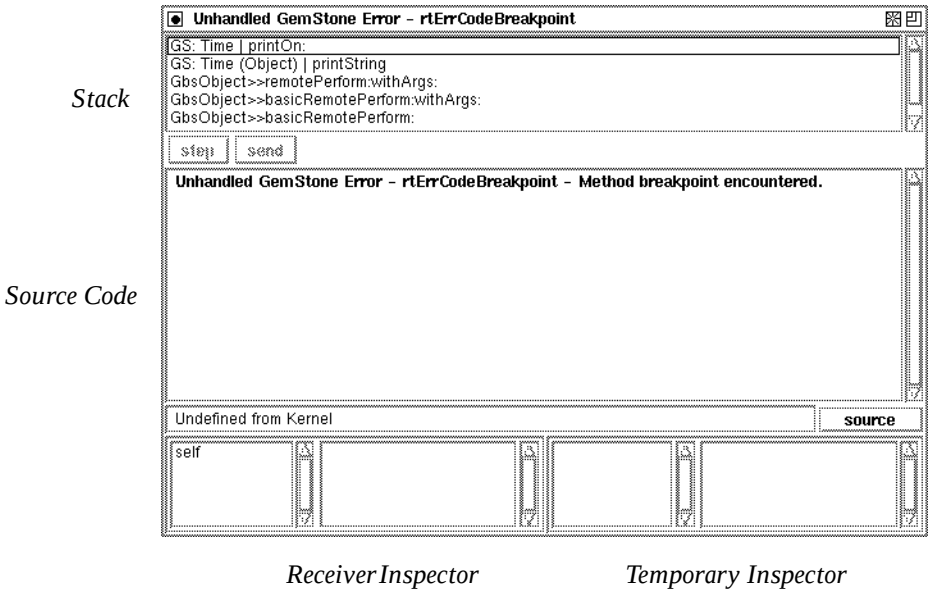
Proceed	If it is possible to continue execution, clicking this button closes the notifier and resumes execution from the point at which the notifier opened.
Debug	Opens a GemStone debugger, which provides facilities similar to those provided by the client Smalltalk debugger.
Terminate	Closes the notifier.
Copy Stack	Copies the contents of the stack to the paste buffer, from where you can paste it into a workspace or a file for later review or discussion with a colleague.

If you choose **Debug**, the GemStone Debugger allows you to:

- view GemStone Smalltalk and client Smalltalk contexts mixed together in the stack,
- set breakpoints without modifying source code,
- select a context from among those active on the virtual machine stack, and
- examine and modify objects and code within that context.

The GemStone Debugger

Figure 8.11 The GemStone Debugger



The GemStone Debugger consists of the following panes:

- *Stack pane.* Displays the active call stack and enables you to choose some context (stack frame) from that stack for manipulation in the window's other panes. The top contexts are GemStone Smalltalk contexts. If you scroll downward through the messages on the stack you will come to `GS: Executed Code` . This is where the GemStone and client Smalltalk worlds meet. The contexts below this point are client Smalltalk contexts.
- *Source pane.* When you select a context in the stack pane, the source pane displays the source code of the method being executed, highlighting text at the current point of execution. This view is an editor that allows you to set breakpoints and modify code.
- *Receiver inspector (two related panes).* An inspector on the receiver of the currently selected context in the stack pane.

- *Temporary inspector (two related panes).* An inspector on the temporary variables and arguments defined in the currently selected context in the stack pane.

The following paragraphs describe these panes in detail.

The Stack Pane

The stack pane is a list view that shows the contexts on the stack. By default, only the nine contexts at the top of the stack are displayed. You can double the number displayed by selecting **more stack** from the stack pane's pop-up menu.

You can select a context for examination and debugging with the left mouse button. When a client Smalltalk context is selected in the stack pane, the GemStone Debugger behaves in the same way as a client Smalltalk debugger. When a GemStone context is selected, the following commands are available.

Table 8.9 Debugger's Stack Pane Menu Commands

more stack	By default, the stack pane only displays the top nine contexts on the stack. Choosing **more stack** doubles the number until the entire stack is displayed.
proceed	*Appears only when it is possible to resume execution.* Tells the GemStone Smalltalk virtual machine to resume execution from the context at the top of the stack. If execution halts because of (for example) an authorization error, then the virtual machine cannot continue.
restart	*Appears only when it is possible to resume execution.* Sets the locus of execution back to the beginning of the method, just after dispatch, and proceeds execution.
senders	*Appears only when you have selected a context.* Opens a method list browser on all methods that send the message in the currently selected context.
implementors	*Appears only when you have selected a context.* Opens a method list browser on all methods that implement the selector in the currently selected context.
messages	*Appears only when you have selected a context.* You are offered a menu of all messages that are called in the source code for the method in the currently selected context. Opens a method list browser on all methods that implement the selector you choose.

Table 8.9 Debugger's Stack Pane Menu Commands (Continued)

skip to caret	*Appears only when you have selected a context.* Place the cursor at the position in the currently selected context where you would like execution to halt. Execution resumes until the virtual machine reaches step point nearest the cursor location.
step	*Appears only when you have selected a context*. Tells the virtual machine to advance to the next step point in the currently selected context and then halt. The GemStone Debugger displays the new virtual machine state.
send	*Appears only when you have selected the top context.* Tells the virtual machine to advance execution to the next step point in the currently selected context and then halt. If the next step point is a message-send, the virtual machine halts at the first step point within the method invoked by that message-send. The Debugger then displays the new context. This differs from **step** in that if the next message in the context contains step points itself, execution halts at the first of those step points. That is, the virtual machine steps into the new method instead of silently executing that method's instructions and halting after the method has completed. The next **step** or **send** command will take place within the context of the new method.

Source Code Pane in GemStone Debugger

When you select a GemStone context in the stack pane, this pane displays the source code of the method executing in that context. In some respects, this pane is similar to the browser's source code pane. That is, it provides **GS-do it**, **GS-print it**, and **GS-inspect** facilities and it lets you set breakpoints.

Table 8.10 shows the choices offered by the pop-up menu in the source code pane.

Table 8.10 Debugger's Source Code Pane Menu Commands

find... **replace...** **undo** **copy** **cut** **paste**	These selections invoke the familiar text editing functions.

Table 8.10 Debugger's Source Code Pane Menu Commands (Continued)

GS-do it **GS-print it** **GS-inspect**	Send the selected code to GemStone to be executed or inspected.
accept **cancel**	Compile the contents of the pane or restore the text in the Source Code pane to the state it was in when it was last compiled. If you accept new contents of a method, the context in which you performed the accept becomes the top context (except for blocks) and the current locus of execution is set to the beginning of the method.
set break	Set a breakpoint
hardcopy	Print the contents of the source pane using the VisualWorks conventions established in your image. Default is to write the text to the file "temp.prt" in the image's working directory.

Inspectors in the GemStone Debugger

The GemStone Debugger includes two inspectors: a receiver inspector at the bottom left and a temporary Inspector at the bottom right. They each consist of two interdependent panes.

The left pane lists the names of variables and arguments defined in the current context. When you select a variable name in the left pane, the right pane shows the value of that variable.

In the right pane, you can also execute expressions in the scope of the selected context. If, for example, the method under inspection had a temporary variable named `count` with a current value of 5, then the expression `count * 5` executed in the right inspector pane would return the value 25.

To replace the value of a variable selected in the left pane, type a GemStone Smalltalk expression in the right pane and select **accept** from that pane's pop-up menu. The selected variable is given the value of the expression. Only `self` cannot be replaced in this way.

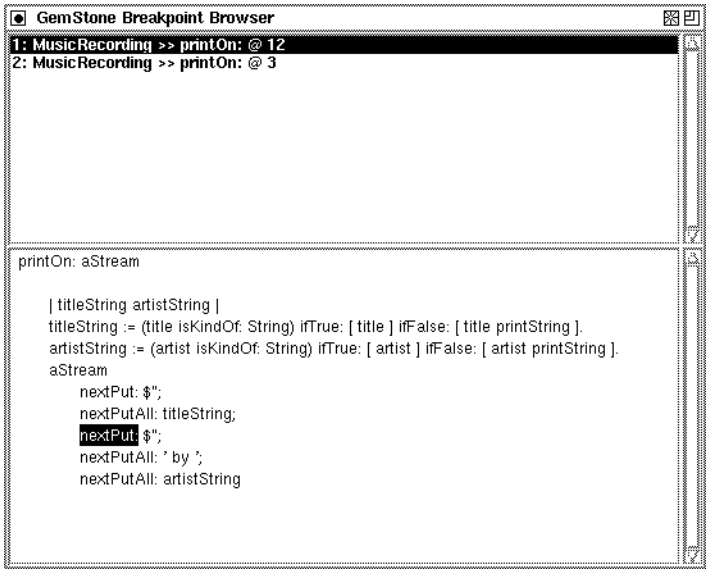
The left half of the receiver inspector pane has two menu items: **inspect** and **update**. (In the temporary inspector, only the **inspect** menu item is available.) Whenever you've selected a variable in this pane, choosing **inspect** opens a GemStone Inspector on the corresponding object; choosing **update** updates the inspector in case you have made changes to these objects from within another tool.

Thus, to examine the value of an instance variable in the current receiver (`self`) you need only select that variable in the receiver inspector and then choose **inspect**. You can then use the resulting GemStone inspector to further examine that variable.

The GemStone Breakpoint Browser

The GemStone menu includes a **Browse > Breakpoints** item; selecting this item opens a GemStone Breakpoint Browser window. The Breakpoint Browser enables you to set, clear, and examine breakpoints for all classes and methods. A Breakpoint Browser has a Break pane and a Source Code pane.

Figure 8.12 Breakpoint Browser



Break Pane in Breakpoint Browser

The break pane displays a scrollable list of the active breakpoints. The items in the list look like this:

```
1: Employee >> taxOwed @ 5
```

In this example, a break is set at step point 5 within the method `taxOwed` defined by class Employee.

This pane's *operate* menu includes the following commands**:**

Table 8.11 Breakpoint Browser Menu Commands

update	Updates the breakpoint list to include breaks that you may have set from within another window (a GemStone Browser, for example).

add break	Brings up a fill-in-the-blank dialog in which you can specify a new breakpoint to be added. The syntax for the specification is that used in the break pane display itself. In BNF form, the syntax for a breakpoint is:
	`<class> ["class"] ">>" <selector> [@ <step num>]`
	For example, the following breakpoint specification requests a breakpoint at step point 2 in the class method `fromStream:` defined by class Float:
	`Float class | fromStream: @ 2`
	The space after the class name is required. If you do not specify a step point number, the breakpoint is set at the first step point.

remove **remove all**	Removes the selected breakpoint or all of the breakpoints displayed in the Break pane.
enable **enable all** **disable** **disable all**	Enables or disables breakpoints without removing them. An enabled breakpoint will cause execution to stop if it is reached. A disabled breakpoint is ignored during execution.

Source Code Pane in the Breakpoint Browser

If you have selected a breakpoint in the break pane, the Source Code pane displays the source code for that method. It is exactly like a GemStone Browser source code pane, except that you cannot execute client Smalltalk code with **do it**, **print it**, or **inspect**. However, you can examine step points, set breakpoints, recompile code, and use **GS-do it**, **GS-print it**, and **GS-inspect** to execute GemStone Smalltalk code.

8.5 Interrupting GemStone Execution

Users of your application can terminate or interrupt GemStone execution in two ways. A soft break (*Control-C* on most platforms or *Command-.* on a Macintosh) interrupts the GemStone Smalltalk virtual machine in such a way that it can be restarted. GemBuilder's default behavior is to display a notifier that contains four buttons: **Proceed**, **Debug**, **Terminate**, and **Copy Stack**. Choosing **Proceed** closes the notifier and causes GemStone execution to resume. Choosing **Debug** opens a GemStone Debugger that enables you to explore the current execution context. Choosing **Copy Stack** copies the contents of the stack to the paste buffer, from where you can paste it into a workspace or a file for later review or discussion with a colleague. Selecting **Terminate** closes the notifier.

Note that GemStone detects soft breaks only when the virtual machine is running; a soft break issued while executing a primitive is not detected until the primitive finishes.

Chapter 9

Performance Tuning

This chapter discusses ways that you can tune your GemBuilder application to optimize performance and minimize maintenance overhead.

Selecting the Locus of Control
provides some rules of thumb for deciding when to have methods execute on the client and when to have them execute on the server.

Profiling and Debugging
explains ways you can examine your program's execution.

Configuring GemBuilder
describes the Settings Browser and GemBuilder configuration parameters, their default and legal values, and their significance.

Replication Tuning
explains the replication mechanism and how you can control the level of replication to optimize performance

Improving Performance by Local Caching
explains how you can speed up the mapping of client Smalltalk objects and their counterparts.

Optimizing Space Management
explains how you can reclaim space from unneeded replicates.

Managing Dirty Object Marking
explains how you can control dirty bit marking.

Using Primitives
introduces the use of methods written in lower-level languages such as C.

Changing the Initial Cache Size
shows how to change the initial cache size.

Multi-threaded Applications
discusses non-blocking protocol and thread-safe transparency caches.

For further information, see the *GemStone Programming Guide* for a discussion on how to optimize GemStone Smalltalk code for faster performance. That manual explains how to cluster objects for fast retrieval, how to profile your code to determine where to optimize, and discusses optimal cache sizes to improve performance.

9.1 Selecting the Locus of Control

By default, GemBuilder executes code in the client Smalltalk. Objects are stored in GemStone for persistence and sharing but are replicated in the client Smalltalk for manipulation. In general, this policy works well. There are times, however, when it is preferable or required to execute in GemStone.

One motivation for preferring execution in GemStone is to improve performance. Certain functions can be performed much more efficiently in GemStone. The following section discusses the trade-offs between client Smalltalk and server Smalltalk execution and how to choose one space over the other.

Beyond optimization, some functions can be performed *only* in GemStone. GemStone's System class, for example, cannot be replicated in the client Smalltalk; messages to System have to be sent in GemStone.

Locus of Execution

This section centers on controlling the locus of execution—in other words, determining whether certain parts of an application should execute in the client Smalltalk or in GemStone. Subsequent sections discuss other ways of tuning to increase execution speed.

Client Smalltalk and GemStone Smalltalk are very similar languages. Using GemBuilder, it is easy to define behavior in either client Smalltalk or GemStone to accomplish the same task. There are, however, performance implications in the

placement of the execution. This section discusses several factors to weigh when choosing the space in which to execute methods.

Relative Platform Speeds

One consideration when choosing the execution platform is the relative speed of the client Smalltalk and the server Smalltalk execution environments. Your client Smalltalk will often run faster than GemStone on the same machine. GemStone's database management functions and its ability to handle very large data sets add some overhead that the client Smalltalk environment doesn't have.

Cost of Data Management

Execution cannot complete until all objects required have been brought into the object space. When executing in the client Smalltalk, this means that all GemStone objects required by the message must be faulted from GemStone. When executing in GemStone, this means that dirty replicates must be flushed from the client Smalltalk. In general, it is impossible to tell exactly which objects will be required by a message send, so GemBuilder flushes all dirty replicates *before* a GemStone message send and faults all dirty GemStone objects *after* the send.

Clearly, data movement can be expensive. Although the client Smalltalk environment might be more efficient for some messages, faulting the object into the client Smalltalk might overwhelm the savings. If the objects are all already there, however, or if the objects will be reused for other messages, then the movement may be justified.

For example, consider searching a set of employees for a specific employee, giving her a raise, and then moving on to another unrelated operation. Although a brute force search may be faster in your client Smalltalk, the cost of moving the data to the client may exceed the savings. The search should probably be done in GemStone.

However, if additional operations are going to be done on the employee set, the cost of moving data is amortized and, as the number of operations increases, becomes less than the potential savings.

GemStone Optimization

Some optimizations are possible only using GemStone execution. In particular, repository searching and sorting can be done much more quickly in GemStone than in your client Smalltalk as data sets become large.

If you will be doing frequent searches of data sets such as the employee set in the previous example, using an index on the server Smalltalk set will speed execution.

The *GemStone Programming Guide* provides a complete discussion of indexes and optimized queries.

Locus of Transaction Control

Transactions can be committed or aborted in several ways. In your client Smalltalk, commit or abort messages can be sent either to the session manager, (GBSM) or to a specific session. In GemStone, transactions can be committed or aborted by sending messages to System.

Due to implementation details related to cache synchronization, the locus of the transaction control can noticeably affect performance. If possible, avoid controlling the transaction through GemStone. Commit or abort your transactions by sending client Smalltalk messages or by using the Session Browser.

9.2 Profiling and Debugging

Profiling Client Smalltalk Execution

A good starting point for optimizing the performance of an application is to find out where most of the execution time is being spent. There are tools available for profiling client Smalltalk code. GemStone also has a profiling tool in the class ProfMonitor. This class allows you to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method. See the chapter in the *GemStone Programming Guide* on Performance for details.

Watching Stub Activity

A switch in the stub class, GbxObjectStub, allows you to see stubs in a debugger and logs faulting activity involving stubs.

`GbxObjectStub stubDebugging:` *aBoolean*

This method turns debugging support on or off. When debugging is on, this class's superclass is GbsDebugStub, providing basic instance methods that allow the client Smalltalk debugger to operate among stubs without causing them to fault in the GemStone object. Notice that applications that rely on these methods might get incorrect results when `stubDebugging` is turned on. For example, sending `#class` to a GbxObjectStub normally causes a fault, returning the class of the replicated object, but when `stubDebugging` is on, the result of sending `#class` is GbxObjectStub.

Another effect of sending `stubDebugging:` is that operations involving stubs are recorded in the System Transcript.

Turning on `stubDebugging` and watching the faulting activity can help you evaluate your tuning parameters.

Using Verbose Mode

GbsSession has a class variable, Verbose (a Boolean), which, if true, causes sessions to write messages to the system transcript when special events occur (such as logout, login, commit, and abort.)

If your application sends `Block>>valueUninterruptably` , you may need to disable the GbsSession's logging of events to the transcript by sending `GBSM verbose: false.` Verbose mode uses `Transcript show:`, which eventually calls `Block>>valueUninterruptably.` If unstubbing occurs during execution of your application's `Block>>valueUninterruptably`, and the unstubbing activity triggers activity that is logged to the transcript, the client Smalltalk will fail.

9.3 Configuring GemBuilder

This section describes the Settings Browser and GemBuilder configuration parameters, their default and legal values, and their significance.

Configuration Parameters Available in GemBuilder

GemBuilder provides configuration switches that make it easy for you tune your program. These switches are listed in the following table, and are described in subsequent sections.

Table 9.1 Configuration Parameters for GemBuilder

Parameter	Legal values	Default
`assertionChecks`	true / false	false
`blockingProtocolRpc`	true / false	true
`bulkLoad`	true / false	false
`confirm`	true / false	true
`connectorNilling`	true / false	true
`connectVerification`	true / false	false

Table 9.1 Configuration Parameters for GemBuilder

Parameter	Legal values	Default
defaultFaultPolicy	#immediate / #lazy	#lazy
eventPollingFrequency	any integer	5000
eventPriority	any integer	50
faultLevelLnk	any integer	2
faultLevelRpc	any integer	4
forwarderDebugging	true / false	false
freeSlotsOnStubbing	true / false	false
generateClassConnectors	true / false	true
generateGSClasses	true / false	true
generateSTClasses	true / false	true
initialCacheSize	any integer	5003
initialDirtyPoolSize	any integer	157
loginLinkedIfAvailable	true / false	true
neglectReadSet	true / false	false
removeInvalidConnectors	true / false	false
stubDebugging	true / false	false
threadSafeCaches	true / false	false
traversalBufferSize	any integer	250000
verbose	true / false	true

To determine the current setting of a parameter, send the parameter name as a message to GBSM. For example, the following expression returns the current setting of the **connectVerification** parameter:

```
GBSM connectVerification
false
```

To set a parameter, append a colon to the parameter name and send it as a message to GBSM with the desired value as the argument. For example, to set the **connectVerification** parameter, send:

```
GBSM connectVerification: true
```

You will probably prefer to use the Settings Browser to view and change the settings of these parameters. (See “The Settings Browser” on page 9-11.)

Using Configuration Parameters to Tune Your Application

This section describes the configuration parameters and how their settings affect your program.

`assertionChecks`
This parameter is for the use of GemStone customer support.

`blockingProtocolRpc`
Determines whether to use blocking or nonblocking protocol for RPC sessions. When `false`, nonblocking protocol is used, enabling other threads to execute in the image while one or more threads are waiting for a GemStone call to complete. When `true`, GemBuilder must wait for a GemStone call to complete before proceeding with the thread of execution that called it.

`bulkLoad`
When `true`, newly created objects are stored in GemStone as permanent objects immediately, bypassing a step wherein they are temporary and eligible for storage reclamation by the GemStone garbage collector unless other objects refer to them (in which case they become permanent objects, as usual). Bypassing this step provides a performance gain in bulk load situations. When set to `false`, the temporary object step is not bypassed.

confirm
When `true`, you are prompted to confirm various GemBuilder actions. Leave set to `true` during application development; deployed applications may wish to set to `false`.

connectorNilling
When `true`, GemBuilder sets certain connectors to `nil` when a session disconnects. This action helps prevent defunct stub and defunct forwarder errors by clearing connectors that depend on being attached to GemStone objects. Connectors that represent variables (that is, name connectors, class variable connectors, and class instance variable connectors) and whose postconnect action is **updateST** or **forwarder**, are nilled. Fast connectors, class connectors, and connectors whose postconnect action is **updateGS** or **none** are not nilled.

When **connectorNilling** is `false`, the logout sequence leaves the state of persistent objects in the image “as is.”

connectVerification
When set to `true` (the default), connectors verify at login that they are not redefining a connector that already exists, and class connectors verify that the two classes they are connecting have compatible structures.

When this parameter is set to `false`, these things are not checked. Keep this parameter set to `true` during development unless logging in becomes too slow or your connector definitions are stable.

See "The Connector Browser" on page 3-10.

defaultFaultPolicy
Specifies GemBuilder's default approach to updating client Smalltalk objects when it becomes aware that their GemStone counterparts have changed. If set to **#lazy**, GemBuilder responds to a change in a GemStone object by turning its Smalltalk replicate into a stub. The new GemStone value will be faulted in when a message is next sent to the stub. If set to **#immediate**, GemBuilder responds to a change in a GemStone object by updating the client Smalltalk replicate immediately. The default is **#lazy**.

eventPollingFrequency
Set the frequency, in milliseconds, that GemBuilder polls for GemStone events - that is, the delay between polls.

eventPriority
Set the priority of the Smalltalk process that responds to GemStone events - that is, the priority at which the block will execute that was supplied as an argument to the keyword `gemSignalAction:`, `notificationAction:`, or `signaledAbortAction:`. These keywords occur in messages used by Gem-to-Gem signaling, changed object notification, and when GemStone signals you to abort so that it can reclaim storage.

faultLevelLnk
The number of levels to replicate an object from GemStone to Smalltalk in a linked session. The default is 2.

faultLevelRpc
The number of levels to replicate an object from GemStone to Smalltalk in a remote session. The default is 4.

forwarderDebugging
When this value is `true`, forwarders respond locally to some basic messages, such as `printOn:`, `instVarAt:`, and `class` (returns GbsForwarder) to support debugging. When this parameter is `false`, these messages are forwarded to the GemStone object.

freeSlotsOnStubbing
When `true`, stubbing an existing replicate causes all persistent named instance variables (that is, those that will be faulted in when the stub is unstubbed) and all indexable instance variables to be set to `nil`. This action allows stubs and their potentially out-of-date instance variables to be garbage collected if they become eligible. When false (the default), GemBuilder does not tamper with the instance variable values. This behavior can be overridden on a class-by-class basis by reimplementing the method `freeSlotsOnStubbing`.

generateClassConnectors
When set to `true` (the default), a session connector is automatically created to connect two classes, one of which has been automatically generated in response to the presence of the other by the mechanisms described in the discussion of parameters **generateSTClasses** and **generateGSClasses**.

See "Class Mapping Between GemStone and Smalltalk" on page 4-17.

generateGSClasses
When set to `true` (the default), if a client Smalltalk object is stored into GemStone and GemStone does not currently define the class of which it is an instance, a corresponding class is defined in GemStone Smalltalk.

See "Class Mapping Between GemStone and Smalltalk" on page 4-17.

generateSTClasses
When set to `true` (the default), if a GemStone object is fetched into the client Smalltalk image and the client Smalltalk image does not currently define the class of which it is an instance, a corresponding class is defined in the image.

See "Class Mapping Between GemStone and Smalltalk" on page 4-17.

initialCacheSize
Sets the size, in bytes, of the initial cache for each GemStone session. For best performance, make this a prime number. The default is 5003.

See "Changing the Initial Cache Size" on page 9-21.

initialDirtyPoolSize
Pregrow the size of GbsSession dirtyPool identity set. For bulk load situations, making this value larger will reduce the number of times this set needs to grow. For applications that perform flushes of a small number of objects, making this value smaller (but larger than the number of objects being flushed) will improve flushing performance.

loginLinkedIfAvailable
When `true`, the result of executing `GBSM login` is a linked GemStone

session, which provides faster database access, unless GemBuilder cannot start a linked session (as is the case on some platforms) or another session is already running linked.

neglectReadSet
Determines whether GemStone objects read by your session are flushed to GemStone's read set, which GemStone maintains to implement locking and detect concurrency conflicts. When you first log into GemStone, GemBuilder initializes a cache to holds all GemStone objects read or written by your session. The default value is `false`.

When this parameter is set to `true`, objects that are stored in GemBuilder's cache from previous transactions as having been read are never flushed to GemStone's read set—the only objects that are flushed to gemStone's read set are those which have been read in the current transaction.

removeInvalidConnectors
When set to `false` (the default), and **verbose** is also `false`, if a connector cannot resolve the objects it must connect at login, GemBuilder raises an error.

When this parameter is set to `false` and verbose is set to `true`, if a connector cannot resolve the objects it must connect at login, you are prompted to remove the invalid connector.

When this parameter is set to `true`, invalid connectors are removed from the connector collections so that the issue does not arise again at next login.

See "The Connector Browser" on page 3-10.

stubDebugging
When set to `true`, stubs respond to some basic messages locally, such as `printOn:`, `instVarAt:`, and `class` (returns GbxObjectStub) to support debugging. When this parameter is set to `false`, these messages cause the stub to fault into the client image from GemStone.

threadSafeCaches
When set to **true**, subsequent logins protect GemBuilder caches that map Smalltalk and GemStone objects, so that they can safely be accessed by more than one process at a time. Performance is slower. Leave this parameter set to false unless your applications use more than one process.

traversalBufferSize
Sets the size, in bytes, of the buffer used in traversal replication.

verbose
When set to `true` (the default), GemBuilder prints messages to the System

Transcript when certain events occur, such as logging a session in or out, or committing or aborting a transaction.

The Settings Browser

The Settings Browser makes it easy to examine and set the configuration parameters for GemBuilder.

Opening the Settings Browser

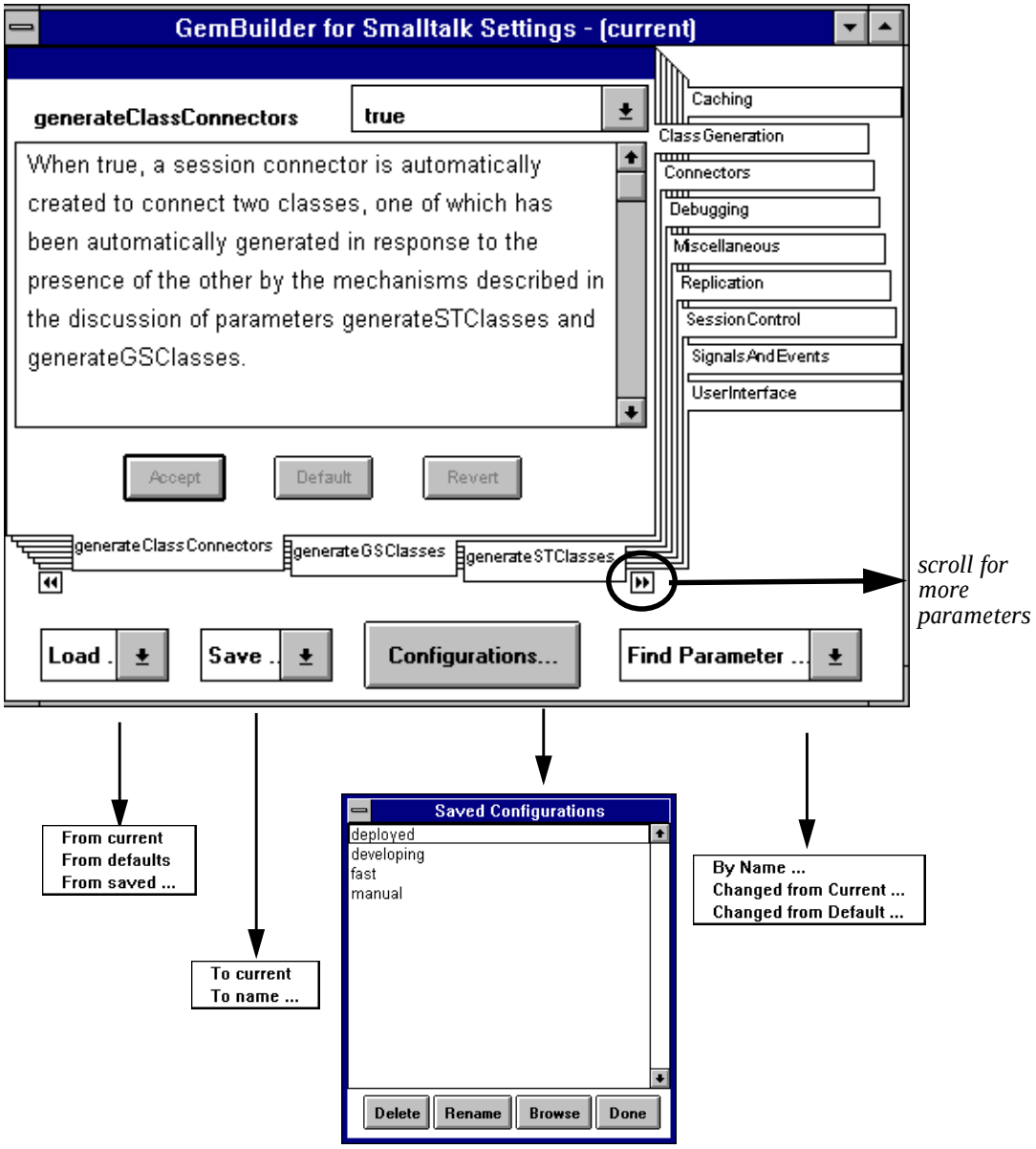
To open the Settings Browser, select **Browse > Settings** from the GemStone Launcher.

You can programmatically open a new Settings Browser by executing `GBSM ConfigurationTool new` in the client Smalltalk. The new tool will contain a copy of the values of the current configuration by default. To open the tool on its default configuration or on some other configuration use one of the following messages:

`open`	opens the tool on its current configuration
`openOn:` *aGbsConfiguration*	opens the tool on a copy of the given configuration
`openOnDefaults`	opens the tool on a copy of the default configuration
`openOnCurrent`	opens the tools on a copy of the currently-installed GemBuilder Configuration

The Settings Browser is shown in Figure 9.1.

Figure 9.1 The Settings Browser



Parameter Notebook

The Settings Browser uses a notebook metaphor to organize the various configuration parameters. A set of tabs on the right side of the notebook provides an index to categories of parameters.

Control buttons allow you to load and save the settings contained in the notebook and to specify the parameter to be displayed.

Table 9.2 Notebook Control Buttons and Their Combo Box Menus

Load...	Provides a menu for selecting a source configuration. Menu options are:	
	From current	Uses configuration values currently installed in GemBuilder.
	From defaults	Uses the default configuration values.
	From saved...	Brings up a dialog so you can enter the name of a saved configuration to use.
Save...	Provides a menu for select the destination of the save. Menu options are:	
	To current	Installs the specified configuration's values in GemBuilder.
	To name...	Brings up a dialog in which you can select an existing named configuration or enter a new name.
Configurations	Brings up a window that displays all named configurations and has buttons that allow you to delete or rename a setting and to open a Configuration Browser on a named setting.	
Find Parameter...	Provides a menu with the following choices:	
	Enter name...	Shows a selection list of all parameters.
	Changed from Current...	Shows all parameters whose values differ from the configuration currently installed in GemBuilder.
	Changed from Default...	Shows all parameters whose values differ from the default configuration values.

Each page of the notebook provides access to a single parameter, using the following fields:

- A label containing the name of the parameter.

- An editable field (a text entry field for Strings or Integers) or menu (when legal values have finite choices, e.g., true or false) that displays the current value of the parameter a pull-down menu if the legal values are choices from a finite set of enumerable values, e.g., true and false.
- A text field containing a description of the parameter and its legal values.

Table 9.3 Parameter Page Control Buttons

Accept	Installs a changed value in the entry field in the notebook's configuration.
Default	Copies the default value for that parameter into the entry field.
Revert	Copies the notebook's configuration value for that parameter into the entry field.

9.4 Replication Tuning

The faulting of GemStone objects into the client Smalltalk is described in Chapter 4. As described there, a GemStone object has a replicate in the client Smalltalk created for itself, and, recursively, for objects it contains to a certain level, at which point stubs instead of replicates are created.

Faulting objects to the proper number of levels can noticeably improve performance. Clearly, there is a cost for faulting objects into the client Smalltalk. This is made up of communication cost with GemStone, object accessing in GemStone, object creation and initialization in the client Smalltalk, and increased virtual machine requirements in the client Smalltalk as the number of objects grows. For this reason, you should try to minimize faulting and fault in to the client only those objects that will actually be used in the client.

On the other hand, inadequate faulting also has its penalties. In the RPC version of GemBuilder, communication overhead is important. When fetching an employee object, it is wasteful to stub the name and then immediately fetch the name from GemStone. Even in the linked version, it is better to avoid creating the stub and then invoking the fault mechanism when sending it a message.

Controlling the Level of Replication

By default, two levels of objects are faulted with the linked version of GemBuilder, and four levels are faulted for the RPC version. This reflects the cost of remote procedure calls and the judgment that it is better to risk fetching unneeded objects to avoid extra calls to GemStone.

It is possible to tune the levels of stubbing to a more optimal level with a knowledge of the application being programmed. You can set the configuration parameters `faultLevelRpc` and `faultLevelLnk` to a SmallInteger indicating the number of levels to replicate when updating an object from GemStone to the client Smalltalk. A level of 2 means to replicate the object and each object it references, stubbing objects beyond that level. A level of 0 indicates no limit; that is, entering 0 prevents any stubs from being created. The default for the linked version is 2; the default for the RPC version is 4. To examine or change this parameter, choose **GemStone > Browse > Settings** and select the **Replication** tab in the resulting Settings Browser.

NOTE:
Take care when using a level of 0 to control replication. GemStone can store more objects than can be replicated in a client Smalltalk object space.

Preventing Transient Stubs

If only the `defaultGStoSTLevel` mechanism is used to control fault levels, it is possible to create large numbers of stubs that are immediately unstubbed.

To prevent stubbing on a class basis, reimplement the `noStubLevel` class method for that class. This method returns an integer that indicates the number of levels of replicates that should always exist below instances of this class before stubbing is allowed.

Setting the TraversalBufferSize

TraversalBufferSize is a pool variable found in SpecialGemStoneObjects. This value describes the number of bytes which GemBuilder uses for its internal buffers when making low-level GemStone calls. The larger the TraversalBufferSize is, the more information GemBuilder is able to transfer in a single network call to GemStone. To change its value, send the message

`GBSM traversalBufferSize:` *aSmallInteger*.

9.5 Improving Performance by Local Caching

You can send `addGSCache` to a client Smalltalk class to add an instance variable called `gsObj` that serves as a local cache with associated cache accessing methods. Adding this instance variable greatly speeds the mapping of client Smalltalk objects and their GemStone counterparts. It should be added to all classes where it is possible.

Unfortunately, this is not possible if the receiver:

- is a class that has no instance variables;
- is a byte-implementation class;
- if the receiver already has a `gsObj` instance variable; or
- if the receiver is a class that is tightly bound to the client Smalltalk virtual machine.

9.6 Optimizing Space Management

In normal use of GemBuilder, objects are faulted from GemStone to the client Smalltalk on demand. In many ways, however, this is a one-way street, and the client Smalltalk object space can only grow. Advantages can be gained if client Smalltalk replicates can be discarded when they are no longer needed. A reduced number of objects on the client reduces the load on the virtual machine, garbage collection, and various other functions.

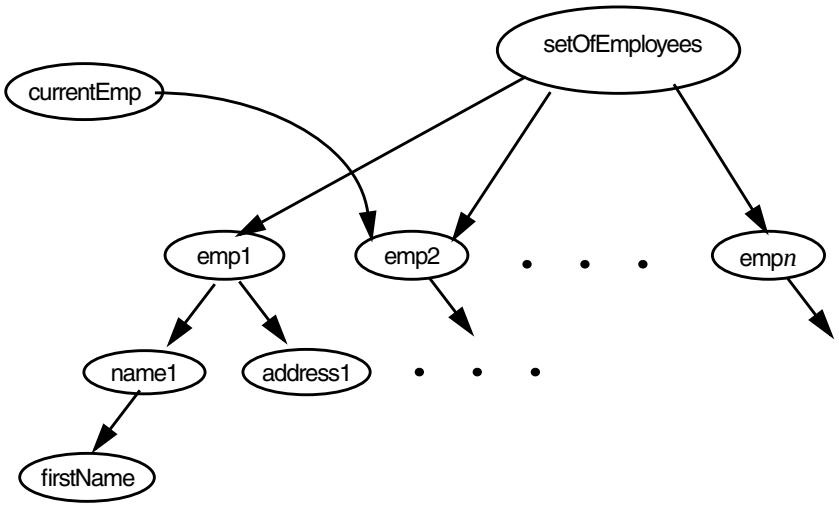
Measures you can take to control the size of the client Smalltalk object cache include explicit stubbing, using forwarders, and not caching certain objects.

Explicit Stubbing

If the application knows that a replicate is not going to be used for a period of time, the space taken by that object can be reclaimed by sending it the message `stubYourself`. More importantly, any objects it references become candidates for garbage collection in your client Smalltalk.

Consider having replicated a set of employees. After faulting in the set and the objects transitively referenced from that set, the objects in the client Smalltalk look something like this.

Figure 9.2 Employee Set Faulted into the Client Smalltalk



Clearly, there can be a large number of objects referenced transitively from the employee set. If the application's focus of interest changes from the set to, say, a specific employee, it may make sense to free the object space used by the employee set.

In this example, one solution is to send `stubYourself` to the `setOfEmployees`. All employees, except those referenced separately from the set, become candidates for garbage collection.

Of course, if the application will be referencing the `setOfEmployees` again in the near future, the advantage gained by stubbing could be offset by the increased cost of faulting later on.

You should also be careful of the subtle difference between two ways of modifying the value of an instance variable: by using an access method and by direct assignment.

For example, consider an object with an instance variable named instVarX. You can assign the value 5 to instVarX in two ways:

```
insVarX := 5            (direct assignment)
self instVarX: 5        (access method)
```

When the object is replicated in your Smalltalk workspace, each of these assigments yields the same result. When the object is represented in the Smalltalk workspace by a stub, however, the stub must be faulted inas a replicate

("unstubbed") before the assignement can occur. The access method causes the stub to be faulted in and yields the correct result. In many cases, however, the direct assignment does not cause the stub to be faulted in and will be ignored:

```
self stubYourself.
self instVarX: 5.          (reliable)

self stubYourself.
instVarX := 5.             (unreliable)
```

Using Forwarders

Another solution is to declare the `setOfEmployees` as a forwarder. See "Working with Forwarders" on page 4-19.

Not Caching Selected Objects

Finally, it is possible to specify classes whose instances should not be added to the transparency caches. You can reimplement the instance method `shouldBeCached` to cause GemBuilder to not add instances of that class to the transparency caches.

This can help control the size of the caches, but it would do so at the expense of giving up two-space referential integrity for those objects, and this might not be an acceptable side effect in certain applications.

For example, classes whose instances are modifiable should probably not return `false` to this message, because any modifications to the object could not be propagated back to the original GemStone object, as GemBuilder would have no way of knowing which object in the repository it came from. Nor should classes whose instances rely on their identity in any way return `false` to this message.

An example of a class that could be considered a candidate for returning `false` is Float. If floats are omitted from the transparency caches, consider the following subtle implications: If a float **F** is referenced by two other objects, **A** and **B**, then after replicating **A** and **B** into the client Smalltalk, there will be two distinct (but equal) copies of the object **F** in the client Smalltalk. If one or both of **A** and **B** are modified in Smalltalk and flushed back to GemStone, there will now be two distinct (but equal) copies of **F** in GemStone. Typically, referential integrity for floats isn't crucial, because comparison between floats is usually by equality rather than identity.

9.7 Managing Dirty Object Marking

The section entitled “Flushing Objects to GemStone” on page 4-12 described how to select between automatic and manual dirty object management. While automatic management is less error-prone, performance gains can sometimes be realized by disabling transparency and taking over this job manually.

GemBuilder implements automatic dirty object management by modifying the compiler. It inserts calls to `markDirty` at key points, as shown in Example 9.1.

Example 9.1

The method

```
exampleMethod
      instVar1 := 1.
      instVar2 := 2.
      instVar3 := 3.
```

is converted to:

```
exampleMethod
      instVar1 := 1.
      self markDirty.
      instVar2 := 2.
      self markDirty.
      instVar3 := 3.
      self markDirty.
```

If we handled this manually, we would write

```
exampleMethod
      instVar1 := 1.
      instVar2 := 2.
      instVar3 := 3.
      self markDirty.
```

The difference between manual and automatic marking dirty is more pronounced with indexable access. When a class is sent `markDirtyOnAtPut` , GemBuilder modifies methods with `at:put:` or `basicAt:put:` to send the receiver a `#markDirty` message whenever an `at:put:` or `basicAt:put:` is sent, so all named and indexed variable assignments are caught. If the class doesn’t contain

its own implementation of these methods, GemBuilder creates a new implementation of the `at:put:` method that includes a `markDirty` as follows:

```
at: arg1 put: arg2.
self markDirty.
^ super at: arg1 put: arg2
```

If the class *does* contain its own implementation of `at:put:` or `basicAt:put:`, GemBuilder creates the new implementation that includes a `markDirty` and renames the old `at:put:` or `basicAt:put:` method to be `origat:put:` or `origbasicAt:put:`.

> *Note: If you use the* `markDirtyOnAtPut` *mechanism, and subsequently want to modify the definition of* `at:put:` *in a class, be aware that you need to modify* `origat:put:`.

Reimplementing the `at:put:` method to include a `markDirty` each time it is used can make automatic dirty management expensive to use in cases such as Example 9.2, where the operation occurs repeatedly.

Example 9.2

```
1 to: 100 do: [:i | myObj at: i put: i // 2]
```

An optimization for such cases would be to not use `markDirtyOnAtPut`, and instead send a `markDirty` explicitly when using `at:put:` messages as in Example 9.3.

Example 9.3

```
1 to: 100 do: [:i | myObj at: i put: i // 2]
myObj markDirty
```

This eliminates 99 calls to `markDirty` and 100 sends to `super`. The drawback is that it is much easier to introduce subtle bugs using this technique.

9.8 Using Primitives

Sometimes there is an advantage to dropping out of Smalltalk programming and writing methods in a lower-level language such as C. Such methods are called *primitives* in Smalltalk; GemStone refers to them as *user actions*. There are serious concerns to consider when doing this. In general, such applications will be less portable and less maintainable. However, when used judiciously, there can be significant performance benefits.

In general, you should profile your code and find those methods that are heavily used to be candidates for primitives or user actions. The trick to proper use of primitives or user actions is to create as few as possible. Excess primitives or user actions make the system more difficult to understand and place a heavy burden on the maintainer.

For a description about adding primitives to your client Smalltalk, see the vendor's documentation. For adding user actions to GemStone, see the *GemBuilder for C* user manual.

9.9 Changing the Initial Cache Size

GbsSessionManager has a class variable named InitialCacheSize, which is an integer that represents the pregrown size of the object caches whenever the caches are initialized. The default is 5003.

For best cache performance, make InitialCacheSize a prime number.

You can change the value of InitialCacheSize by sending

```
GBSM initialCacheSize:  newValue
```

or by modifying that expression in the GemStone System Workspace.

9.10 Multi-threaded Applications

Some applications support multiple Smalltalk processes, or threads of execution, running concurrently in a single image. In addition, some applications enter into a multithreaded state occasionally when they make use of signalling and notification. Multithreaded GemBuilder applications must exercise some precautions in order to preserve expected behavior and data integrity among their concurrent processes.

Thread-Safe Transparency Caches

By default, GemBuilder uses transparency cache dictionaries that are not thread safe. To use transparency cache dictionaries that are protected for use with multi-threaded client Smalltalk applications, you must set the GemBuilder configuration parameter **threadSafeCaches** to `true` by changing its setting in the Settings Browser or by sending the message:

aGbsSession `threadSafeCaches: true`

When **threadSafeCaches** is `true`, subsequent logins use transparency cache dictionaries that are protected. Note, however that some operations take a bit longer when using protected dictionaries.

Blocking and Nonblocking Protocol

In a linked GemBuilder session, GemStone operations execute synchronously: the application must wait for a GemStone operation to complete before proceeding with the thread of execution that called it. Synchronous operation is known in GemBuilder as *blocking protocol*.

An RPC GemBuilder session can support asynchronous operation: *nonblocking protocol*. When the configuration parameter **blockingProtocolRpc** is `false` (the default setting in RPC sessions), client Smalltalk threads can proceed with execution during GemStone operations. A session, however, is permitted only one outstanding GemStone operation at a time.

When **blockingProtocolRpc** is `true`, behavior is the same as in a linked session: the thread of execution must wait for a GemStone call to return before proceeding.

One Thread Per Session

Applications that limit themselves to one thread per GemStone session are relatively easy to design because each thread has its own view of the repository. Each thread can rely on GemStone to coordinate its modifications to shared objects with modifications performed by other threads, each of which has its own session and own view of the repository. For such applications, setting **threadSafeCaches** to true is the only additional precaution required. If at all possible, try to limit your application to one thread per GemStone session.

Multiple Threads per Session

Applications that have multiple threads running against a single GemStone session must take additional precautions.

You may not have designed your application to run multiple threads under a single GemStone session. However, if your application uses signals and notifiers, chances are it is occasionally running two threads against a single GemStone session. Methods that create concurrent processes include:

```
GbsSession
    >>notificationAction:
    >>gemSignalAction:
    >>signaledAbortAction:
```

When the specified event occurs, the block you supply to these methods runs in a separate thread. Unless your main thread of execution is idle when these events occur, you need to take the same precautions as any other application running multiple threads against a single session.

Applications that have multiple threads running against a single GemStone session should take these additional precautions:

- coordinate transaction boundaries
- coordinate flushing
- coordinate faulting

GemBuilder provides a method, `GbsSession>>critical:` *aBlock*, that evaluates the supplied block under the protection of a semaphore that is unique to that session. The best approach to creating an application that must support more than one thread interacting with a single GemStone session is to organize its logical transactions into short operations that can be performed entirely within the protection of `GbsSession>>critical:`. All of that session's commits, aborts, executes, forwarder sends, flushes and faults should be performed within `GbsSession>>critical:` blocks.

For example, a block that implements a writing transaction will typically start with an abort, make object modifications, and then finish with a commit. A block that implements a reading transaction might start with an abort, perhaps perform a GemStone query, and then maybe display the result in the user interface.

Coordinating Transaction Boundaries

Multiple threads need to be in agreement before a commit or abort occurs. For example, suppose two threads share a single GemStone session. If one thread is in the process of modifying a set of persistent objects and a second thread performs a commit, the committed state of the repository will contain a logically inconsistent state of that set of objects.

The application must coordinate transaction boundaries. One way to do this is to make one thread the transaction controller for a session, and require that all other threads sharing that session request that thread for a transaction state change. The controller thread can then be blocked from performing that change until all other threads using that session have relinquished control via some semaphore protocol.

Coordinating Flushing

GemBuilder's transparency mechanism flushes dirty objects to GemStone whenever a commit, abort, GemStone execution or forwarder send occurs. Whenever a thread modifies persistent objects, it must protect against other threads performing operations that trigger flushing of dirty objects to GemStone. The risks are that a flush may catch a logically inconsistent state of a single object, or might cause GemBuilder to mark an object "not dirty" without really flushing it.

To control when flushing occurs, perform update operations within a block passed to `GbsSession>>critical:`.

Coordinating Faulting

If two threads send a message to a stub at roughly the same time, one of the threads can receive an incomplete view of the contents of the object. This results in doesNotUnderstand errors which cannot be explained by looking at them under a debugger, because by the time it is visible in the debugger, the object has been completely initialized. Unstubbing conflicts can be avoided by encapsulating potential unstubbing operations within the protection of a `GbsSession>>critical:` block.

Using the VisualWorks Application Model

As mentioned above, an application that otherwise restricts itself to one thread per GemStone session can occasionally find itself running multiple threads against a single session through the asynchronous operation of signals and notifiers.

One way to coordinate the activities of signal and notifier threads is to remap asynchronous GemStone events into synchrounous window events, as though they were received through the user interface. Implementing this strategy requires two steps. The first is to use some or all of the following methods to redirect GemStone signals and notifiers to the application model:

```
GbsSession
    >>sendGemSignalEventsTo: anApplicationModel
    >>sendNotificationEventsTo: anApplicationModel
    >>sendSignaledAbortEventsTo: anApplicationModel
```

For example, you could implement an initialize method in your application model as shown in Example 9.4.

Example 9.4

```
self session
      sendNotificationEventsTo: self;
      sendSignaledAbortEventsTo: self.
```

The second step is to reimplement `windowEvent:from:` in your ApplicationModel subclass to respond to the appropriate corresponding events `#gemSignalAction`, `#notificationAction`, or `#signaledAbortAction`, as shown in Example 9.5.

Example 9.5

```
windowEvent: anEvent from: aWindow
  super windowEvent: anEvent from: aWindow.
  (anEvent key == #signaledAbortAction) ifTrue: [
        Transcript cr;
             show: 'Aborting and refreshing the view.'.
             self session abortTransaction.
             self updateView ].
  (anEvent key == #notificationAction) ifTrue: [
        (anEvent value includes: someList) ifTrue: [
             Transcript cr;
               show: 'List modified by another session';
               cr; show: 'aborting and refreshing view.'.
             self session abortTransaction.
             self updateView ]]
```

Chapter

10 Nontransparent Access to GemStone Objects

In this chapter, we discuss some very low-level approaches to tuning GemBuilder applications. To varying degrees, each of these approaches ignores automatic GemBuilder transparency and bypasses the encapsulation provided by object-oriented programming. We do not recommend using these techniques until all other approaches have been evaluated and found lacking.

Nontransparency: General Principles
presents some concepts that are fundamental to other topics in the chapter, such as flushing and faulting, and public and private classes.

GbsObject Proxies
introduces one of GemBuilder's main mechanisms for controlling transparency.

Structural Access to GemStone Objects
discusses how to fetch bytes from, or store bytes into, GemStone objects directly.

Nontransparent Replication
explains how to create your own copy of a GemStone object local to the client Smalltalk.

Executing GemStone Host File Access Methods
lists certain GemStone Smalltalk methods that can be used to access the host operating system or the host file system.

There are several code examples in this chapter. If you want to experiment with them, we suggest you file the goodies **nontrans.gs** and **nontrans.st** into your image:

- First, file **nontrans.gs** into GemStone. This file contains classes and methods in support of the code examples in this chapter.
- Then file **nontrans.st** into your client Smalltalk to open a workspace that contains the code examples so you can execute and inspect them.

10.1 Nontransparency: General Principles

Under normal operating conditions, GemBuilder makes shared object access as transparent as possible. That is, shared objects from the object server appear for most purposes to be local to your application's Smalltalk image. Changes made to locally-visible shared objects are propagated automatically to the object server, and changes made by other users become visible to you with only minor intervention on your part.

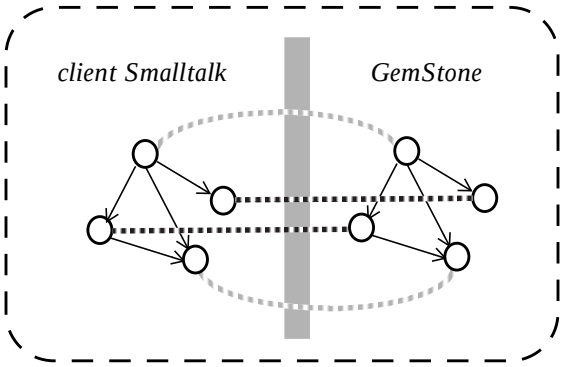
GemBuilder's transparency features provide both convenience for the developer and data integrity for the application, but they do incur some overhead. The optimizations in this chapter can offer some efficiency gains, but you, as the developer, must give up some convenience and take responsibility for some of the automatic object integrity features you choose to bypass.

We recommend that you confine such optimizations to a small number of performance-critical operations, implement them carefully, and comment your work clearly to avoid potential maintenance and upgrade problems in the future.

Flushing and Faulting Nontransparent Objects

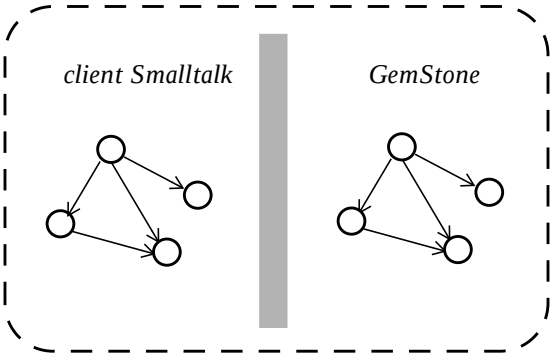
Chapter 3 discussed how a replicate created in client Smalltalk can maintain a link to a GemStone Smalltalk object so that changes in either the GemStone Smalltalk or the client Smalltalk object could propagate to the other. Figure 10.1 illustrates this.

Figure 10.1 Transparent Object



When transparency is bypassed, the replication is the same, but the links are gone, as shown in Figure 10.2.

Figure 10.2 Nontransparent Objects



When you work with nontransparent objects, you will need to take on the responsibility of triggering flushing and faulting. Before you access a GemStone object you will want to be sure that any local changes have been flushed. After accessing the object, you will want to update the local replicate if it changed.

In general, you should be sure to:

- flush objects before a fetch, and
- fault objects after a store.

To explicitly flush all dirty the client Smalltalk objects in a specific session, use:

aGbsSession `flushDirtyToGS`

This message can also be sent to the session manager, which passes it along to the current session:

`GBSM flushDirtyToGS`

To flush an individual object, send it the message `putInGS`. This message has no effect if the object is not dirty.

To fault an individual object from GemStone, use one of the following messages:

```
stubYourself
updateFromGS
```

The first, `stubYourself`, unconditionally converts the object to a stub that will be faulted the next time it receives a message. The second, `updateFromGS`, will either create a stub or update the object, based on its fault policy.

Public vs. Private Classes and Methods

GemBuilder adds many classes and methods to your client Smalltalk image. Most of these we consider *public*, which means that you are free to use them directly in your applications, knowing that GemStone will support them from release to release. Other classes and methods we consider *private*; you should avoid using private classes and methods because they may have undocumented side effects, and because they are subject to change from release to release.

A GemBuilder class is private if its name begins with a "Gbx" prefix or if it belongs to a class category containing the word "private." Similarly, a GemBuilder method is private if it begins with an underscore prefix (_) or if it belongs to a method category containing the word "private."

Specifying a Session

Many of the methods mentioned in this chapter are sent to instances of GbsSession. Like the `flushDirtyToGS` example shown earlier in this chapter, any message that can be sent to a specific session can also be sent to the GBSM session manager, which passes the message to the current session:

aGbsSession `flushDirtyToGS`	receiver is *aGbsSession*
`GBSM flushDirtyToGS`	receiver is current session

We recommend that you send messages, when possible, to specific sessions, rather than to the session manager. Sending messages to specific sessions is more reliable

(because the current session can change asynchronously under some circumstances) and more efficient (because there is one less level of indirection).

The examples in the rest of this chapter use specific sessions as the receivers of session messages. Most of them can be recoded to send messages to the GBSM session manager, if necessary.

10.2 GbsObject Proxies

Instances of `GbsObject` serve as "proxies" within your client Smalltalk application for corresponding objects in the GemStone repository. Through these proxies, the client Smalltalk objects can send messages to GemStone objects, bypassing any local replicates.

A `GbsObject` proxy can be obtained in one of the following ways:

- by sending `asGSObject` to an existing replicate of a GemStone object
- by searching the GemStone symbol list using the `resolveSymbol:` protocol

  A client Smalltalk program can gain access to those named GemStone objects by sending one of the following messages:

  aGbsSession `resolveSymbol:` *objectName*
  aGbsSession `resolveSymbol:` *objectName* `ifAbsent:` *exceptionBlock*

  These messages return instances of class `GbsObject` to act as proxies for the sought-after GemStone objects. For example, the following expression will find the object named `UserGlobals` in the user's symbol list and return a proxy `GbsObject` for it:

  ```
  aProxyForUserGlobals :=
      aGbsSession resolveSymbol: #UserGlobals
  ```

- by using a predefined `GbsObject` proxy for kernel objects

  Although you can use `resolveSymbol:` to obtain a proxy for any named GemStone object, it would be inefficient (not to mention inconvenient) if you had to make a network call to GemStone each time you wanted to refer to one of the well-known, unchanging GemStone kernel objects. Therefore, GemBuilder provides a dictionary called `SpecialGemStoneObjects` which contains instances of `GbsObject` representing all of the GemStone kernel classes in the GemStone repository, as well as the GemStone values `nil`, `true`, and `false`, and the GemStone error dictionaries.

Each proxy kernel object is named in SpecialGemStoneObjects by the name of the GemStone object it represents, prefixed by the letter "O." For example, `ODictionary` refers to a proxy for the GemStone Dictionary class object, `Onil` refers to a proxy for GemStone `nil`, and `Otrue` refers to a proxy for the GemStone value `true`.

You can use `SpecialGemStoneObjects` as a pool dictionary in your client Smalltalk classes to give their methods access by name to the set of basic GemStone objects. For your convenience during exploration and debugging, names defined in `SpecialGemStoneObjects` are also recognized in the GemStone Workspace. For example, if you had defined a GemStone object named "MyBoolean," then you could execute this expression in a GemStone Workspace:

```
(aGbsSession resolveSymbol: 'MyBoolean') = Otrue
```

(Two instances of `GbsObject` are equal if they are proxies for the same GemStone object.)

Appendix A of this manual, "GemBuilder Classes and Proxies," lists the GemStone objects that are defined in SpecialGemStoneObjects.

Sending Messages Through GbsObject Proxies

GemBuilder provides two mechanisms for sending messages to GemStone objects through their `GbsObject` proxies: `remotePerform:` messages and "trap-door" message-passing.

The `remotePerform:` message is used like the standard Smalltalk `perform:` message. There are four versions of this message:

```
remotePerform: aSelector
remotePerform: aSelector with: anArg
remotePerform: aSelector with: firstArg with: secondArg
remotePerform: aSelector withArgs: argArray
```

For example, you could send the GemStone Smalltalk message `new` to the class Array as shown in Example 10.1:

Example 10.1

```
| aGbsArray aGbsSession |
aGbsSession := GBSM currentSession.
aGbsArray :=
 (aGbsSession resolveSymbol: 'Array') remotePerform: #new.
aGbsArray
```

Actually, because GemBuilder provides a predefined proxy for each GemStone kernel object, you can simplify the expression by using the predefined OArray proxy, instead of creating a new proxy with the resolveSymbol: message (Example 10.2).

Example 10.2

```
aGbsArray := OArray remotePerform: #new.
```

You can also communicate with GemStone objects more naturally by sending "trap-door" messages. These are ordinary client Smalltalk messages that begin with the characters `gs`. When a proxy object does not understand a message, it checks for the `gs` prefix. If the proxy object finds that prefix, it removes the `gs` and passes the message along to its corresponding GemStone object for execution in GemStone.

Example 10.3 is equivalent to Example 10.1:

Example 10.3

```
| aGbsArray |
aGbsArray := OArray gsnew.
```

The following two expressions are also equivalent:

```
aGbsArray remotePerform: #at:put: with: 1 with: 10
aGbsArray gsat: 1 put: 10
```

In both styles of message passing, the selector (`remotePerform: orgsat:put:`) is a local client Smalltalk Symbol object and the arguments can be either GemStone proxies (instances of `GbsObject`) or client Smalltalk objects. If arguments are client Smalltalk objects, they are flushed to the repository before the message is sent. In either case, the result of the message will be a `GbsObject`.

Special Treatment of Binary Selectors

The client Smalltalk compiler does not allow any of the following binary selector characters in unary or keyword selectors:

```
!   @   &   *   -   +   |   \   /   <   >   ,   ~
```

This means that you must use `remotePerform:` to send binary GemStone Smalltalk messages rather than using trap door messages. For example, given two proxies representing instances of GemStone Number, the following expression asks whether one is less than the other:

```
aGbsInteger1 remotePerform: #< with: aGbsInteger2
```

The following is not legal in client Smalltalk:

```
aGbsInteger1 gs< aGbsInteger2
```

Sending Code to Gemstone for Execution

In addition to forwarding messages to be executed through proxy objects, your Smalltalk application can also send strings of GemStone Smalltalk code to GemStone for compilation and execution. The expression

aGbsSession `execute:` *aString*

tells GemStone to compile and execute the GemStone Smalltalk code contained in *aString*, and to return a `GbsObject` representing the result of that execution. The code in *aString* may be a message expression or a statement. For example, the following client Smalltalk code installs a new GemStone Set in the dictionary UserGlobals:

```
aGbsSession execute: 'UserGlobals at: #MySet put: Set new'.
```

In comparison with `remotePerform:`, the `execute:` method incurs additional overhead because it invokes the GemStone Smalltalk compiler. Nonetheless, it sometimes provides an attractive alternative to the remote message-passing mechanism without noticeably slowing your application.

The `execute:` message provides a useful way of getting around some limitations of the remote message-passing mechanism. Suppose, for example, that you wanted to trigger an indexed search of a GemStone Set. Because there is no client Smalltalk equivalent of a selection block, you could not simply build a literal selection block and use it as an argument to `gsselect:`. You can, however, create proxies for both the collection and the selection block, then instruct GemStone to perform the operation and return its results, also as a proxy:

Example 10.4

```
| empSetProxy aSelBlockProxy selResultProxy aGbsSession |

aGbsSession := GBSM currentSession.

"Get a proxy for the set of Employees"
empSetProxy := aGbsSession resolveSymbol: #MyEmps.

"Make a GemStone SelectionBlock and get a proxy for it"
aSelBlockProxy := aGbsSession execute:
            '{:each | each.name.first = ''Joebob''}'.

"Execute the query"
selResultProxy := empSetProxy gsselect: aSelBlockProxy.
selResultProxy
```

To perform a selection based on a client Smalltalk String provided by your application's user, you might build up the argument to `execute:` progressively, as in the method in Example 10.5.

Example 10.5

```
| empSetProxy aSelBlockProxy selResultProxy nameString
      queryString aGbsSession |

aGbsSession := GBSM currentSession.

"Get a proxy for the set of Employees"
empSetProxy := aGbsSession resolveSymbol: 'MyEmps'.
nameString := 'Joebob'.

"Build up a String representing a selection block, inserting
the nameString passed by our user into the appropriate place
in the predicate"
queryString := '{:each | each.name.first = '''     .
queryString := queryString, nameString, '''}'.

"Make a GemStone SelectionBlock and get a proxy for it"
aSelBlockProxy := aGbsSession execute: queryString.

"Execute the query"
^selResultProxy := empSetProxy gsselect: aSelBlockProxy.
```

Converting GSObjects to Replicates

A `GbsObject` proxy cannot be used as an ordinary replicate object can. It is useful only with the messages defined earlier in this section. After using a `GbsObject` proxy to execute GemStone Smalltalk code in GemStone, the result is returned as a `GbsObject proxy` .

To create a replicate from a `GbsObject` proxy and continue local execution of messages, you can send `asLocalObject` to the proxy.

10.3 Structural Access to GemStone Objects

GbsObject provides a set of *structural access* methods. These methods enable you to examine and modify the internal structures of GemStone objects without sending GemStone Smalltalk messages, and they allow you to create new instances of GemStone classes without executing any GemStone instance creation methods.

You may need to use structural access methods if speed is your primary concern. By calling on GemStone's internal object manager without invoking the GemStone Smalltalk interpreter, structural access methods provide the most efficient possible access to individual GemStone objects. However, use these methods only if you've determined that GemStone message-passing is too slow.

There are four groups of structural access methods. Each group of methods is specialized for fetching from or storing in different kinds of instance variables with different storage types (see the *GemStone Programming Guide* for details about storage types).

There is, for example, a group of methods you can use for fetching and storing indexable pointer instance variables. Example 10.6 uses several of those methods:

Example 10.6

```
| aGbsArray aGbsSession |

aGbsSession := GBSM currentSession.

"Make a new GemStone Array"
aGbsArray :=  aGbsSession execute: 'Array new: 30'.

"Fetch the object at position 1"
aGbsArray fetchVaryingOOPAt: 1.

"Fetch 2 objects starting at position 1"
aGbsArray fetch: 2 idxOOPsAt: 1.

"Store the GemStone object nil at position 30"
aGbsArray storeIdxOOP: (ONil) at: 30.
```

Each of the structural access fetching methods either returns an instance of GbsObject or an Array of GSObjects.

There are similar methods for fetching from and storing in named and anonymous (NSC) pointer instance variables and for accessing byte objects such as Strings.

Example 10.7

```
| aGbsString aGbsIDBag aGbsSession |

aGbsSession := GBSM currentSession.

aGbsString := aGbsSession execute: 'String new: 100'     .
aGbsIDBag := aGbsSession execute: 'IdentityBag new'.

"Fetch the first 3 characters of a String"
aGbsString fetch: 3 charsAt: 1.

"Store a new String in an existing String,overwriting the
 existing characters starting at position 50"
aGbsString storeChars: 'All''s well that ends in
H.G.Wells' at: 50.

"Since we can't refer to elements of IdentityBags and
 IdentitySets by indexes, we use methods that add or
 remove specific objects by identity"
aGbsIDBag addOOP: Otrue.
aGbsIDBag removeOOP: Otrue.
aGbsIDBag
```

The method `fetch:charsAt:` returns a client Smalltalk String representing the Characters fetched from GemStone. The method `storeChars:at:` translates the client Smalltalk String given as its argument into an equivalent GemStone String before doing the storage into the repository.

For a complete listing and descriptions of the structural access methods, use the Smalltalk browser to browse the `GbsObject` class.

10.4 Nontransparent Replication

As described earlier in this chapter, replication can be done nontransparently. In this case, a complete transitive closure of the GemStone object is replicated in the client Smalltalk; that is, stubs are never created when bypassing transparency. This forces the programmer to use care not to replicate a GemStone object that would overflow the client Smalltalk object space.

While transparency is almost always easier to use, it may sometimes be faster and easier to simply replicate the data nontransparently, manipulate it, and then replicate it in GemStone. To do this successfully, the following must be true:

- There must be just one root for the GemStone data. Identity of internal objects will be lost in this technique.
- The data set must be small enough to fit in the client Smalltalk memory.
- The ratio of execution to data set size must be large.

Example 10.8 illustrates the approach of nontransparent replication:

Example 10.8

```
local :=  aGbsSession at: #GemStoneDataSetName.
       .
       .
       .
```

(perform operations on local object)

```
aGbsSession at: #GemStoneDataSetName put: local
```

A similar approach can be taken without bypassing transparency. Sending `aGbsObject asLocalObjectCopy` makes a local copy unrelated to the GemStone original. Similarly, sending `aLocalObject asGSObjectCopy` makes an unrelated copy of the local object in GemStone.

10.5 Executing GemStone Host File Access Methods

If you execute an GemStone host file access method (as listed below) without supplying an explicit path specification as part of the method argument, the default directory for the GemStone method depends on the type of Gem that you are running. With a linked version, the default directory is the directory in which the client Smalltalk virtual machine was started. With a remote Gem, the default directory is the `$HOME` directory of the host user account.

Here is a list of the GemStone methods that are affected:

```
String | toServerTextFile:
String (C) | fromServerTextFile:
System (C) | contentsOfServerDirectory:
System (C) | deleteServerFile:
System (C) | performOnServer:
```

Chapter

11 Exception Handling

This chapter discusses errors: how to handle them and how to recover from them.

Error Handling and Recovery
explains how GbsError objects are created and used.

User-Defined Errors
explains how to define and signal your own errors.

GemBuilder's Smalltalk Emergency Handler
discusses GemBuilder's replacement for the default client Smalltalk emergency handler.

11.1 Error Handling and Recovery

An instance of GbsError is created when GemBuilder encounters a GemStone error. Each GbsError can represent itself as a predefined client Smalltalk signal. Your application can use these signals to perform client Smalltalk exception-handling. When an error is detected, GemBuilder creates an instance of GbsError and raises its signal.

If a handler is not set up, the emergency handler opens a notifier from which you can open a debugger. You can use the `handle:do:` method to install error handlers to anticipate specific GemStone errors, as shown in Example 11.1.

Example 11.1

```
(GbsError signalFor: #objErrBadOffsetIncomplete)
  handle: [ :ex |
    ex halt: 'proceed to inspect bad offset error.'.
    ex originator inspect ]
  do: [ GBSM execute: '#( 1 2 3 ) at: 4']
```

You can also create a handler to check for any GemStone error that falls in one of the following categories:

```
#compilerErrorSignal
#abortingErrorSignal
#interpreterErrorSignal
#fatalErrorSignal
#eventErrorSignal
```

For instance, this will handle any GemStone Smalltalk compiler error:

```
GbsError signalFor: #compilerErrorSignal
 handle: [ . . . ]
 do: [ . . . ]
```

You can also create a handler to check for multiple errors:

```
(SignalCollection
   with: (GbsError signalFor: #interpreterErrorSignal)
   with: (GbsError signalFor: #rtErrAbortTrans))
 handle: [ . . .]
 do: [ . . . ]
```

For more information, refer to the Smalltalk documentation on exception handling and to the chapter entitled “Handling Errors” in the *GemStone Programming Guide.*

11.2 User-Defined Errors

You can define and signal your own errors in GemStone. For more information on how to do this, see the *GemStone Programming Guide*.

In a GemBuilder application, you define a generic GemStone error handler by defining a standard client Smalltalk signal handler on the signal `GbsError errorSignal`. This handles any GemStone error, including user-defined errors.

If you want to define a client Smalltalk exception handler for a specific user-defined error, you will need to register an exception, GemStone error number, and a symbol representing that error with GbsError. To do this, send GbsError class>>`defineErrorNumber:name:signal:`.

For example, suppose you have created a GemStone user-defined error as follows:

Example 11.2

```
"In GemStone"
| myErrors |
myErrors := LanguageDictionary new.
UserGlobals at: #MyErrors put: myErrors.
myErrors at: #English put: (Array new: 10).
(myErrors at: #English)
      at: 10
      put: #( 'My new error with argument ' 1 ).
```

In Smalltalk, the following code would signal your newly created error:

```
GBSM execute: 'System signal: 10
          args: #[ 46 ] signalDictionary: MyErrors'
```

A generic signal-handler for all GemStone errors would trap this signal:

```
GbsError errorSignal
          handle: [ :ex | ^#handled ]
          do: [ GBSM execute: 'System
                signal: 10
                args: #[ 46 ]
                signalDictionary: MyErrors' ]
```

To explicitly handle your new error in client Smalltalk, you first need to define a name and signal for it. The new signal should inherit from GbsError errorSignal.

```
GbsError
   defineErrorNumber: 10
   name: #myNewError
   signal: GbsError errorSignal newSignal.
```

So now, to explicitly handle your new error from client Smalltalk:

Example 11.3

```
(GbsError signalFor: #myNewError)
      handle: [ :ex | ^#handled ]
      do: [ GBSM execute: 'System
            signal: 10
            args: #[ 46 ]
            signalDictionary: MyErrors' ]
```

For information on how to create GemStone error dictionaries and handle GemStone errors (predefined and user-defined) within the GemStone environment, see the *GemStone Programming Guide*.

11.3 GemBuilder's Smalltalk Emergency Handler

GemBuilder installs its own client Smalltalk emergency handler, replacing the default Smalltalk emergency handler, which is created by evaluating `Exception initialize`. The emergency handler opens a notifier from which you can open a debugger.

GemBuilder's handler is installed at installation time, in the method GbsError class `>>initEmergencyHandler`. Its purpose is to support debugging of GemStone behavior; besides this, it is not essential to the operation of GemBuilder.

In an image running a production application, you would probably want to create your own emergency handler.

Chapter

A

GemBuilder Classes and Proxies

A.1 Special GemBuilder Classes

Besides defining classes for the GemStone browsers and tools and managing sessions, GemBuilder adds a number of classes to the client Smalltalk hierarchy to allow your Smalltalk application to communicate with a GemStone repository. These classes are concerned with raising GemStone errors, connecting corresponding objects in the client Smalltalk and in GemStone, and providing direct access to the low-level structure of GemStone objects.

Class for Raising Errors

GemBuilder adds a client Smalltalk class named **GbsError** to raise errors.

An instance of class GbsError represents a GemStone error. Every GbsError is able to raise itself as a signal in the client Smalltalk. When a GemStone error is detected, GemBuilder creates an instance of GbsError and raises its signal.

Classes for Connecting Objects

GemBuilder adds a number of classes to the client Smalltalk class hierarchy that provide functionality for establishing connections between objects. **GbsConnector** is an abstract superclass for a hierarchy of classes whose

instances describe how to connect a GemStone and a client Smalltalk object. Connectors are described in detail in Chapter 3. The connector hierarchy is:

```
GbsConnector
      GbsFastConnector
      GbsNameConnector
            GbsClassConnector
            GbsClassVarConnector
                  GbsClassInstVarConnector
```

Class for Forwarding Messages

GemBuilder also provides a class that can minimize the overhead of replication by forwarding messages to be executed in a GemStone object.

GbsForwarder
A forwarder is a client Smalltalk object that responds to most messages by sending them on to its corresponding GemStone object. Results are returned to the forwarder, which then can return them as either client Smalltalk objects or other forwarders.

Class for Providing Structural Access

Instances of **GbsObject** act as proxies for GemStone objects. These proxies can be sent messages that perform structural access, traversal, GemStone message sends, or general inquiries. See Chapter 10 for a complete discussion.

A.2 Reserved OOPs

In order to allow your client Smalltalk application program to refer to predefined GemStone objects, GbsSessionManager's `initialize` method creates the pool dictionary SpecialGemStoneObjects, then adds the following objects to that dictionary:

- the values `Onil`, `Otrue`, and `Ofalse` (*nil*, *true*, and *false*)
- `OIllegal`, a value sometimes used for representing an illegal or inappropriate attempt to fetch an object.
- the GemStone kernel classes (*Oclassname*)
- the GemStone error dictionary (`OGemStoneErrorCategory`)

Illegal object — `OIllegal`

Nil (UndefinedObject) — `Onil`

Booleans— `Ofalse`, `Otrue`

GemStone Kernel Classes—

```
OAbstractCharacter
OAbstractCollisionBucket
OAbstractDictionary
OAbstractUserProfileSet
OAllClusterBuckets
OArray
OAssociation
OAutoComplete
OBag
OBasicSortNode
OBehavior
OBlockClosure
OBoolean
OByteArray
OCanonicalStringDictionary
OCharacter
OCharacterCollection
OClampSpecification
OClass
OClassHistory
OClassOrganizer
OClassSet
OClientForwarder
OClusterBucket
OClusterBucketArray
OCollection
OComplexBlock
OComplexVCBlock
ODatabaseConversion
ODate
ODateTime
ODecimalFloat
ODictionary
ODoubleByteString
ODoubleByteSymbol
OEUCString
OEUCSymbol
OEmptyInvariantArray
OEmptyInvariantString
OException
OExecutableBlock
Ofalse
```

OFloat
OFraction
OGsClassDocumentation
OGsCloneList
OGsCurrentSession
OGsDocText
OGsFile
OGsInterSessionSignal
OGsMethod
OGsMethodDictionary
OGsProcess
OGsRemoteSession
OGsSession
OGsSocket
OGsStackBuffer
OGsTransactionalSession
OISOLatin
OIdentityBag
OIdentityDictionary
OIdentityKeyValueDictionary
OIdentitySet
OIllegal
OInteger
OIntegerKeyValueDictionary
OInterval
OInvariantArray
OInvariantEUCString
OInvariantString
OKeyValueDictionary
OLanguageDictionary
OLargeNegativeInteger
OLargePositiveInteger
OMagnitude
OMetaclass
Onil
ONumber
OObject
OOrderedCollection
OPassiveObject
OPositionableStream
OProfMonitor
ORcCollisionBucket
ORcCounter
ORcIdentityBag
ORcKeyValueDictionary
ORcPipe
ORcPositiveCounter
ORcQueue
OReadStream
ORedoLog
ORepository
OSegment
OSegmentSet
OSelectBlock
OSequenceableCollection
OSet
OSimpleBlock
OSmallFloat
OSmallInteger
OSortNode
OSortedCollection
OStream
OString
OStringKeyValueDictionary
OStringPair
OStringPairSet
OSymbol
OSymbolAssociation
OSymbolDictionary
OSymbolKeyValueDictionary
OSymbolList
OSymbolSet
OSystem
OTime
Otrue
OUndefinedObject
OUnorderedCollection
OUserProfile
OUserProfileSet
OUserSecurityData
OVariableContext
OWriteStream
OEmptySymbol

Appendix

B Network Resource String Syntax

This appendix describes the syntax for network resource strings. A network resource string (NRS) provides a means for uniquely identifying a GemStone file or process by specifying its location on the network, its type, and authorization information. GemStone utilities use network resource strings to request services from a NetLDI.

B.1 Overview

One common application of NRS strings is the specification of login parameters for a remote process (RPC) GemStone application. An RPC login typically requires you to specify a GemStone repository monitor and a Gem service on a remote server, using NRS strings that include the remote server's hostname. For example, to log in from Topaz to a Stone process called "gemserver50" running on node "handel", you would specify two NRS strings:

```
topaz> set gemstone !@handel!gemserver50
topaz> set gemnetid !@handel!gemnetobject
```

Many GemStone processes use network resource strings, so the strings show up in places where command arguments are recorded, such as the GemStone log file. Looking at log messages will show you the way an NRS works. For example:

```
Opening transaction log file for read,
filename = !tcp@oboe#dbf!/user1/gemstone/data/tranlog0.dbf
```

An NRS can contain spaces and special characters. On heterogeneous network systems, you need to keep in mind that the various UNIX shells have their own rules for interpreting these characters. If you have a problem getting a command to work with an NRS as part of the command line, check the syntax of the NRS recorded in the log file. It may be that the shell didn't expand the string as you expected.

NOTE

Before you begin using network resource strings, make sure you understand the behavior of the software that will process the command.

See each operating system's documentation for a full discussion of its own rules. For example, under the UNIX C shell, you must escape an exclamation point (!) with a preceding backslash (\) character:

```
% waitstone \!tcp@oboe\!gemserver50 -1
```

If there is a space in the NRS, you can replace the space with a colon (:), or you can enclose the string in quotes (" "). For example, the following network resource strings are equivalent:

```
% waitstone !tcp@oboe#auth:user@password!gemserver50
% waitstone "!tcp@oboe#auth user@password!gemserver50"
```

B.2 Defaults

The following items uniquely identify a network resource:

- communications protocol— such as TCP/IP, DECnet, or SNA
- destination node—the host that has the resource
- authentication of the user—such as a system authorization code
- resource type—such as server, database extent, or task
- environment—such as a NetLDI, a directory, or the name of a log file
- resource name—the name of the specific resource being requested.

A network resource string can include some or all of this information. In most cases, you need not fill in all of the fields in a network resource string. The information required depends upon the nature of the utility being executed and the task to be accomplished. Most GemStone utilities provide some context-sensitive defaults. For example, the Topaz interface prefixes the name of a Stone process with the **#server** resource identifier.

When a utility needs a value for which it does not have a built-in default, it relies on the system-wide defaults described in the syntax productions in "Syntax" on page B-4. You can supply your own default values for NRS modifiers by defining an environment variable named GEMSTONE_NRS_ALL in the form of the *nrs-header* production described in the Syntax section. If GEMSTONE_NRS_ALL defines a value for the desired field, that value is used in place of the system default. (There can be no meaningful default value for "resource name.")

A GemStone utility picks up the value of GEMSTONE_NRS_ALL as it is defined when the utility is started. Subsequent changes to the environment variable are not reflected in the behavior of an already-running utility.

When a client utility submits a request to a NetLDI, the utility uses its own defaults and those gleaned from its environment to build the NRS. After the NRS is submitted to it, the NetLDI then applies additional defaults if needed. Values submitted by the client utility take precedence over those provided by the NetLDI.

B.3 Notation

Terminal symbols are printed in boldface. They appear in a network resource string as written:

#server

Nonterminal symbols are printed in italics. They are defined in terms of terminal symbols and other nonterminal symbols:

username ::= *nrs-identifier*

Items enclosed in square brackets are optional. When they appear, they can appear only one time:

address-modifier ::= [*protocol*] [**@** *node*]

Items enclosed in curly braces are also optional. When they appear, they can appear more than once:

nrs-header ::= **!** [*address-modifier*] {*keyword-modifier*} **!**

Parentheses and vertical bars denote multiple options. Any single item on the list can be chosen:

protocol ::= (**tcp** | **decnet** | **serial** | **default**)

B.4 Syntax

nrs ::= [*nrs-header*] *nrs-body*

where:

nrs-header ::= **!** [*address-modifier*] {*keyword-modifier*} [*resource-modifier*]**!**
All modifiers are optional, and defaults apply if a modifier is omitted. The value of an environment variable can be placed in an NRS by preceding the name of the variable with "$". If the name needs to be followed by alphanumeric text, then it can be bracketed by "{" and "}". If an environment variable named `foo` exists, then either of the following will cause it to be expanded: `$foo` or `${foo}`. Environment variables are only expanded in the *nrs-header*. The *nrs-body* is never parsed.

address-modifier ::= [*protocol*] [**@** *node*]
Specifies where the network resource is.

protocol ::= (**tcp** | **decnet** | **serial** | **default**)
Supports heterogeneous connections by predicating address on a network type. If no protocol is specified, GCI_NET_DEFAULT_PROTOCOL is used. On UNIX hosts, this default is **tcp**.

node ::= *nrs-identifier*
If no node is specified, the current machine's network node name is used. The identifier may also be an Internet-style numeric address. For example:

```
!tcp@120.0.0.4#server!cornerstone
```

nrs-identifier ::= *identifier*
Identifiers are runs of characters; the special characters !, #, $, @, ^ and white space (blank, tab, newline) must be preceded by a "^". Identifiers are words in the UNIX sense.

keyword-modifier ::= (*authorization-modifier* | *environment-modifier*)
Keyword modifiers may be given in any order. If a keyword modifier is specified more than once, the latter replaces the former. If a keyword modifier takes an argument, then the keyword may be separated from the argument by a space or a colon.

authorization-modifier ::= ((**#auth** | **#encrypted**) [**:**] *username* [**@** *password*] | **#krb**)

#auth specifies a valid user on the target network. A valid password is needed only if the resource type requires authentication. **#encrypted** is used by GemStone utilities. If no authentication information is specified, the system will try to get it from the `.netrc` file. This type of authorization is the default.

#krb specifies that kerberos authentication is to be used instead of a user name and password.

username ::= *nrs-identifier*

If no user name is specified, the default is the current user. (See the earlier discussion of *nrs-identifier*.)

password ::= *nrs-identifier*

If no password is specified, the system will try to obtain it from the user's `.netrc` file. (See the earlier discussion of *nrs-identifier*.)

environment-modifier ::= (**#netldi** | **#dir** | **#log**) [**:**] *nrs-identifier*

#netldi causes the named NetLDI to be used to service the request. If no NetLDI is specified, the default is `netldi50`. (See the earlier discussion of *nrs-identifier*.)

#dir sets the default directory of the network resource. It has no effect if the resource already exists. If a directory is not set, the pattern "`%H`" (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

#log sets the name of the log file of the network resource. It has no effect if the resource already exists. If the log name is a relative path, it is relative to the working directory. If a log name is not set, the pattern "`%N%P%M.log`" (defined below) is used. (See the earlier discussion of *nrs-identifier*.)

The argument to **#dir** or **#log** can contain patterns that are expanded in the context of the created resource. The following patterns are supported:

%H	home directory
%M	machine's network node name
%N	executable's base name
%P	process pid
%U	user name
%%	%

resource-modifier ::= (**#server** | **#spawn | #task** | **#dbf** | **#monitor** | **#file**)
Identifies the intended purpose of the string in the *nrs-body*. An NRS can contain only one resource modifier. The default resource modifier is context sensitive. For instance, if the system expects an NRS for a database file, then the default is **#dbf**.

#server directs the NetLDI to search for the network address of a server, such as a Stone or another NetLDI. If successful, it returns the address. The *nrs-body* is a network server name. A successful lookup means only that the service has been defined; it does not indicate whether the service is currently running. A new process will not be started. (Authorization is needed only if the NetLDI is on a remote node and is running in secure mode.)

#task starts a new Gem. The *nrs-body* is a NetLDI service name (such as "gemnetobject"), followed by arguments to the command line. The NetLDI creates the named service by looking first for an entry in `$GEMSTONE/bin/services.dat` , and then in the user's home directory for an executable having that name. The NetLDI returns the network address of the service. (Authorization is needed to create a new process unless the NetLDI is in guest mode.) The **#task** resource modifier is also used internally to create page servers.

#dbf is used to access a database file. The *nrs-body* is the file spec of a GemStone database file. The NetLDI creates a page server on the given node to access the database and returns the network address of the page server. (Authorization is needed unless the NetLDI is in guest mode).

#spawn is used internally to start the garbage-collection Gem process.

#monitor is used internally to start up a shared page cache monitor.

#file means the *nrs-body* is the file spec of a file on the given host (not currently implemented).

nrs-body ::= unformatted text, to end of string
The *nrs-body* is interpreted according to the context established by the *resource-modifier*. No extended identifier expansion is done in the *nrs-body*, and no special escapes are needed.

Appendix

C

ParcPlace Smalltalk and GemStone Smalltalk

This appendix outlines the few general and syntactical differences between the ParcPlace Smalltalk and GemStone Smalltalk languages.

GemStone Smalltalk and Client Smalltalk

GemStone's Smalltalk language is very similar to client Smalltalk in both its organization and its syntax. GemStone Smalltalk extends the Smalltalk language with classes and primitives to add multiuser features such as transaction support and persistence. The GemStone class hierarchy is extensible, and new classes can be added as required to model an application. The GemStone class hierarchy is described in the *GemStone Programming Guide*.

A quick look at the GemStone class hierarchy shows that it differs from the client Smalltalk class hierarchy in that classes for file access, communication, screen manipulation, and the client Smalltalk programming environment don't exist, and in that the GemStone Smalltalk hierarchy contains classes for transaction control, accounting, ownership, authorization, replication, user profiles, and index control.

GemStone Smalltalk also introduces constraints and optimized selection blocks.

As a Smalltalk programmer, you will feel quite at home with GemStone Smalltalk, but you should take note of the differences outlined in this appendix.

Selection Blocks

Selection blocks in GemStone Smalltalk and the use of dots for path notation have no counterparts in client Smalltalk.

```
myEmployees select: {:i | i.is.permanent}
```

Array Constructors

Array constructors do not exist in client Smalltalk. In GemStone, array constructors:

- use square brackets,
- use commas as separators, and
- can contain any valid GemStone Smalltalk expression as an element.

```
#['string one', #symbolOne, $c, 4, Object new]
```

Exception Handling

In client Smalltalk, exception handling is implemented with two classes: Signal and Exception. In GemStone it is implemented with a single class: Exception. An Exception in GemStone is an object that represents state to be invoked in the event of an exception.

There are two types of exceptions in GemStone. In order of precedence, they are: 1) context exceptions, and 2) static exceptions. Static exceptions remain from run to run. Context exceptions are active as long as the context to which the exception belongs is on your call stack when an exception is signaled.

Client Smalltalk exception handling is analogous to GemStone context exceptions.

All nonfatal errors can be trapped by a GemStone application.

Exception handling in ParcPlace Smalltalk is accomplished by sending messages such as `handle:do:`. In ParcPlace Smalltalk the argument to `handle:` is a predefined signal. From within the handler block, messages can be sent to the signal to cause the flow of control to resume, exit the `handle:do:` block, or restart the block.

Index

A

abort
(GbsSession) 5-4, 5-5
(GbsSessionManager) 5-5, 5-7

abort command in GemStone 8-12

add break command
in GemStone Breakpoint Browser 8-43

add command
in GemStone Browser's Symbol List menu 8-11
in GemStone Object Inspector 8-7
in GemStone System Browser's Symbol List pane 8-11

addGSCache message 9-16

adding
a new symbol dictionary 8-11
breakpoints 8-43

addParameters (GbsSession) 2-6

addToCommitOrAbortReleaseLocksSet: (System) 5-15

addToCommitReleaseLocksSet: (System) 5-15

application design 1-7–1-9

arguments
examining 8-41

array constructors
in GemStone Smalltalk C-2

asForwarder message 4-19

asGSObject 10-5

assertionChecksconfiguration parameter 9-7

assigning a migration destination 7-3

at: message 4-19

at:ifAbsent: message 4-19

at:put: message 4-13

authorization 6-6, 6-7
and migration 7-4

automatic class generation 4-17, 9-9
disabling 4-18

automatic dirty object management 9-19

automatic marking of modified client Smalltalk objects 4-13
and indexable access 9-19

automatic transaction mode 5-8, 5-9
defined 5-8

B

`basicAt:put:` message 4-13
binary selectors
special treatment of 10-8
blockingProtocolRpc configuration parameter 9-5, 9-7
Blocks
replication of 4-20
Breakpoint Browser
Source Code Pane 8-43
updating 8-43
breakpoints 8-31–8-37
adding 8-35, 8-38, 8-43
and primitives 8-33
and special methods 8-34
for special methods 8-33
methods that cannot have 8-34
setting 8-38, 8-43
bulkLoad configuration parameter 9-5, 9-7
business objects 1-8

C

cache management 4-4
cache size
changing 9-21
cache tuning 9-20
caching issues 9-5
changed object notification 5-17, A-1
changing
initial cache size 9-21
invariant objects 8-25
choosing the locus of execution 9-3
circular constraints 8-20
class connectors 3-6, 3-7, 9-9
class generation
automatic 4-17, 9-9
class generation, automatic 4-17
class instance variable connectors 3-6, 3-8
class instance variables
in GemStone 8-22, 8-23
Class List
in GemStone Browser 8-9
class mapping
between client Smalltalk and GemStone 7-6
nonstandard 4-2, 4-3
class variable connectors 3-6, 3-7
class variables
in GemStone 8-22, 8-23
classes
automatic generation 4-17
filing out 8-28
migrating instances to a new version 7-2
private 10-4
reduced-conflict 5-16
versions of 7-2
clear break command
in GemStone Breakpoint Browser 8-43
`clearCommitOrAbortReleaseLocksSet` (System) 5-15
`clearCommitReleaseLocksSet` (System) 5-15
client Smalltalk compiler
special treatment of binary selectors 10-8
client Smalltalk object cache
space management 9-16
`commit`
(GbsSession) 5-4
commit command 8-12
`commitAndReleaseLocks` (System) 5-15
committing
a transaction
performance 5-16
changes to the repository 5-4
compilation
errors
during **definition** command 8-25
of a class definition 8-21, 8-25
of GemStone Smalltalk code 10-8

compile in ST command
in Browser's Class menu 7-9
in GemStone Browser's Class menu 8-14
in GemStone Browser's Message Category menu 8-15
in GemStone Browser's Message menu 8-15, 8-16
concurrency
modes
setting 5-12
optimistic control 5-11
pessimistic control 5-11, 5-12
concurrent transactions
managing 5-10
configuration parameters 9-5–9-11
assertionChecks 9-5, 9-7
blockingProtocolRpc 9-7
bulkLoad 9-5, 9-7
confirm 9-5
connectorNilling 4-22
connectVerification 3-8, 9-5, 9-6, 9-8
defaultFaultPolicy 9-6
eventPollingFrequency 9-8
eventPriority 9-8
faultLevelLnk 9-8
faultLevelRpc 9-8
freeSlotsOnStubbing 9-6
GemStone
CONCURRENCY_MODE 5-12
CONCURRENCY_MODE 6-1
generateClassConnectors 9-6, 9-9
generateGSClasses 9-6, 9-9
generateSTClasses 9-6, 9-9
initialCacheSize 9-6, 9-9
initialDirtyPoolSize 9-6, 9-9
loginLinkedIfAvailable 2-2, 9-6, 9-9
neglectReadSet 9-6, 9-10
noFaultDebugging 9-6, 9-8
noForwardDebugging 9-6, 9-8
removeInvalidConnectors 9-6, 9-10
setting and examining 9-6
threadSafeCaches 9-6, 9-10
traversalBufferSize 9-6
verbose 9-6, 9-10
conflicts in transactions
reducing
and performance 5-16
write/write
and RcQueue class 5-17
Connected command in Connector Browser 3-14
connected objects
synchronizing 3-13, 3-14
connections
between application roots 4-20
Connector Browser 3-10–3-14
updateGS postconnect action 3-14
updateST postconnect action 3-14
connector nilling 4-22
connector verification 3-8, 9-8
connectorNilling configuration parameter 4-22
connectors 3-1–3-20
and verification 3-13, 9-8
class 3-6, 3-7, 9-9
class instance variable 3-6, 3-8
class variable 3-6, 3-7
class, and forwarders 3-9
class, and update direction 3-9
creating 3-13
creating programmatically 3-15, 3-16
defined 3-5
fast 3-6, 3-8
global 3-5, 3-11
hierarchy of 3-15
name 3-6
nilling 3-10
postconnect actions 3-8
predefined 4-17
session 3-5, 3-11
setup for initial storage of data in GemStone 3-14
updateGS postconnect action 3-14, 4-17
updateST postconnect action 3-14

connectVerification configuration parameter 3-8, 9-5, 9-8
constraints 8-22
constraints on instance variables 8-20, 8-22
  circular 8-20
  inherited by subclasses 8-20
contexts in the GemStone Debugger 8-31
controlling the size of the client Smalltalk object cache 9-16
converting proxies to replicates 10-10
Copy Stack command 8-36, 8-37, 8-44
create access command
  in Browser's Class menu 7-9, 8-14
create in ST command
  in Browser's Class menu 7-9, 8-14
creating
  a new version of a GemStone class 8-27
  new instances of GemStone classes 10-11
creating connectors 3-13
creating forwarders 3-14
current segment 6-6
current session 2-10

D

Database Users List (User Account Manager) 6-22
Debug command
  in GemStone Debugger 8-36, 8-37, 8-44
debugging GemStone code 8-31–8-44
debugging support in GemBuilder 9-4
default directory
  and GemStone host file access methods 10-14
default segment 6-5
defining
  methods in GemStone 8-28
  new classes in GemStone 8-21
  your own GemStone errors 11-1
definition command
  in Browser's Class menu 7-8, 8-13
defunct forwarders 4-21
defunct object stubs 4-8
dependents, session 2-13–2-17
dictionaries
  adding Associations to 8-7
  Globals 6-9, 6-10
  pool 8-22
  shared 6-9–6-10
  specifying for a new class 8-22
  UserGlobals 6-9, 6-10
dirty objects 4-4, 4-5, 4-13, 9-19
  automatic marking of 4-13
  explicit marking of 4-14
  managing for improved performance 9-19
  marking of 9-2
disabling
  automatic class generation 4-18
Disconnected command in Connector Browser 3-14

E

errors
  during **definition** command
    in Browser's Class menu 8-25
  during recompilation of class definition 8-25
  handling of
    and recovery 11-1
    during a GemStone file in 8-31
  runtime 8-25
  user-defined 11-1, 11-3
eventPollingFrequency configuration parameter 9-6, 9-8
eventPriority configuration parameter 9-6, 9-8
examining the internal structure of a GemStone object 10-11
exception handling 11-1–11-4
  in client Smalltalk and in GemStone C-2
`exclusiveLock:` (GbsSession) 5-13
`exclusiveLock:ifDenied:ifChanged:` (GbsSession) 5-13
`exclusiveLockAll:` (GbsSession) 5-13

exclusiveLockAll:ifIncomplete: (GbsSession) 5-14
execute: (GbsSession) 10-8
execution
in GemStone 9-2
in the client Smalltalk 9-2
of GemStone host file access methods 10-14
profiling 9-4
tuning 9-2–9-4
explict stubbing of objects to reclaim space 9-16

F

False (predefined GemStone object) 10-5, A-3
fast connectors 3-6, 3-8
fault control
and replicates 9-14
and stubs 9-14
fault levels
and performance tuning 9-14
fault message 4-8
faulting 4-5
cost of 9-14
dirty GemStone objects 4-5, 9-3
inadequate, penalties of 9-14
levels of 4-7
minimizing for performance tuning 9-14
faultLevellnk configuration parameter 9-6
faultLevelRpc configuration parameter 9-6
faultPolicy message 4-8
file in
and error handling 8-31
format 8-29
file out command
in Browser's Class menu 7-8, 8-13
in Browser's Message menu 8-14, 8-15
in Browser's Symbol List menu 8-10
file out methods command
in Browser's Symbol List menu 7-8, 8-10, 8-13
files
writing class and method definitions to 8-28
find class command
in Browser's Symbol List menu 8-12
finding objects
in the repository 10-5
floats
omitting from transparency caches 9-18
flushing
client Smalltalk objects into GemStone 4-5, 4-12
of dirty replicates 9-3
format for file in / out 8-29
forwarders 4-19–4-22
creating 3-14
defunct 4-21
sending messages to 4-19
using for optimization 9-18
fwat: message 4-19
fwat:ifAbsent: message 4-19

G

GbsBuffer 4-15
GbsClassConnector class A-2
GbsClassInstVarConnector class 3-15
GbsClassVarConnector class 3-15
GbsConnector class 3-2, 3-15, A-2
GbsError class A-1
GbsFastConnector class 3-8, 3-15, A-2
GbsForwarder class A-2
GBSM 2-6
GBSM, instance of GbsSessionManager 2-4, 2-10, 2-11, 5-4
GbsNameConnector 4-19
GbsNameConnector class A-2
GbsObject proxies 10-5–10-10
GbsObjectTraversal class A-1

GbsSession class 2-3, 2-10
GbsSessionManager class 2-4
GbsSessionParameters class 2-4
  instance creation 2-5
Gem
  service name 2-5
  user process 1-2, 1-3
GemBuilder tools 1-6
  Connector Browser 3-10–3-14
  Login Editor 2-8
  Session Browser 2-7–2-13
GemStone
  Debugger window 8-36
  kernel class objects and SpecialGemStoneObjects 10-5
GemStone **Admin** menu 8-4
GemStone Breakpoint Browser
  code pane 8-43
GemStone **Browse** menu 8-3
GemStone Browser 8-8, 8-35
  Class List pane 8-9, 8-12
  Message Category List pane 8-9, 8-14
  Message List pane 8-9
  Symbol List pane in System Browser 8-9
GemStone Debugger 8-4
  opening 8-37
  source code pane 8-40
  stack pane 8-39
    menu commands 8-39, 8-40
GemStone error dictionaries
  predefined objects available in client Smalltalk 10-5
GemStone file-in
  and Topaz commands 8-30
GemStone kernel classes
  predefined objects A-3
GemStone sessions
  current session 2-10
  multiple 2-11
  removing 2-9
GemStone sessions, multiple 2-2
GemStone Smalltalk 1-5
GemStone Smalltalk and client Smalltalk, compared C-1
GemStone Smalltalk code
  sending to GemStone 10-8
GemStone Smalltalk Interface
  classes added to client Smalltalk image A-1
GemStone technical support vi
GemStone **Tools** menu 8-3
GemStone Workspace 8-5
Gem-to-Gem notifiers 5-18
generateClassConnectorsconfiguration parameter 9-6, 9-9
generateGSClasses configuration parameter 9-6, 9-9
global connectors 3-5
Globals dictionary 6-9, 6-10
`gs` prefix
  and trap-door message passing to GemStone objects 10-6
GS-do it command 8-5, 8-17
GS-file in command 8-29
GS-inspect command 8-5, 8-17
GS-print it command 8-5, 8-17

I

implementors command
  in GemStone Browser's Message menu 8-15
  in GemStone Debugger 8-39
indexable access and automatic marking dirty 9-19
indexable instances 8-20
`indexableSize` message 4-16
`indexableValueAt:` message 4-16
`indexableValueAt:put:` method 4-11
`indexableValues` message 4-15, 4-16
`indexableValuesBuffer` message 4-11
inherited constraints 8-20
initial cache size
  changing 9-21

InitialCacheSize class variable 9-21
initialCacheSize configuration parameter 9-6, 9-9
initialDirtyPoolSize configuration parameter 9-6, 9-9
inspect command
  in GemStone Browser 8-11
  in GemStone Object Inspector 8-6, 8-41
inspecting
  objects 8-5, 8-6, 8-17, 8-31
  temporary variables 8-41
inspector
  receiver 8-38
  temporary 8-39
instance migration 7-2
instance variables 8-22
  constraining 8-20
  examining 8-42
  in GemStone
    named 8-22
    private 8-24
  inspecting 8-6
  mapping 4-16
    in migration 7-4
  mapping between GemStone and client Smalltalk 4-2
  maximum number in a Class 8-22
  modifying 4-15
  private 8-24
internal structure of a GemStone object
  examining and modifying 10-11
invariant objects 8-21
  changing 8-25

K

kernel class objects
  and SpecialGemStoneObjects 10-5
kernel classes
  and transparency 4-14
  predefined objects A-3
known GemStone objects
  making available to client Smalltalk classes 10-5

L

linked application 2-2
linked session 2-2
`listInstances:` (Repository) 7-3
local caching 9-16
locks 5-10
  logging out, effect of 5-14
  on objects 5-13
  releasing 5-15
  removing 5-14
  setting 5-13
locus of execution 9-2
locus of transaction control 9-4
logging in to a GemStone session 2-2
logging out of GemStone
  effect on locks 5-14
login authorization 6-2
Login Editor 2-8
loginLinkedIfAvailable configuration parameter 2-2, 9-6, 9-9

M

`makeGSTransparent` message 4-14
manual dirty object management 9-19
manual transaction mode 5-9
mapping
  classes 4-17, 4-18
  instance variables 4-2, 4-16
  nonstandard 4-2, 4-3
`markDirty` message 4-13, 9-19
`markDirtyOnAtPut` message 4-13
`markDirtyOnInstvarAssign` message 4-13
marking objects dirty 4-4, 4-5
  automatically 4-13

maximum number of
instance variables in a class 8-22
message category
filing out a 8-14, 8-15
Message Category List
in GemStone Browser 8-9, 8-14
Message List
in GemStone Browser 8-9
messages
sent to GemStone objects 10-6
messages command
in GemStone Browser's Message menu 8-15
in GemStone Debugger 8-39
methods
adding new 8-28
breakpoints in 8-31
filing out 8-28
moving a method to another category 8-15
primitive
and breakpoints 8-33
private 8-16, 10-4
protecting 6-2
public 8-16
removing 8-15, 8-16
special
and breakpoints 8-34
migration
authorization errors and 7-4
destination 7-3
ignoring 7-4
instance variable mapping 7-4
methods for
`migrate` (Object) 7-3
`migrateFrom:instVarMap:` (Object) 7-5
`migrateInstances:to:` (Object) 7-4
`migrateTo:` (Object) 7-3
of instances 7-2
modifying an existing class in GemStone 8-25
modifying the internal structure of a GemStone object 10-11
monitoring GemStone execution 9-4
more command in GemStone Object Inspector 8-7
more stack command
in GemStone Debugger window 8-39, 8-40
move command
in Browser's Class menu 7-9, 8-14
in Browser's Message menu 8-15
moving a method to another category 8-15
multithreaded applications 9-21

N

name
Gem service 2-5
name connectors 3-6
name of superclass
specifying 8-22
`namedValueAt:` message 4-16
`namedValueAt:put:` message 4-11
`namedValues` message 4-15
`namedValues:` message 4-11
`namedValues:indexableValues:` message 4-11
`namedValuesBuffer` message 4-10
names
of instance variables 8-22
neglectReadSet mode 9-10
network
node 2-4
resource string syntax B-1
`Nil`
(predefined GemStone object) 10-5, A-3
node
network 2-4
noFaultDebugging configuration parameter 9-6, 9-8
`noFaultDebugging` message 9-4
noForwardDebugging configuration parameter 9-6, 9-8

nonstandard class mapping 4-3
noStubLevel message 4-9
notification
  Gem-to-Gem 5-18
NRS (network resource string)
  syntax B-1

O

object
  stubs 4-5, 4-12
object repository 1-2
object stubs 4-5
  defunct 4-8
  explicit control of 4-8
  preventing 4-7
  sending messages to 4-5
object-level invariance 8-21
OFalse (predefined object) A-3
OIllegal
  (predefined object) A-3
ONil (predefined object) A-3
opening a GemStone Debugger 8-37
optimization
  and local caching 9-16
  and multithreaded applications 9-21
  and TraversalBufferSize 9-15
  by explicit stubbing 9-16
  by manual marking of dirty objects 9-19
  by using forwarders 9-18
  changing the initial cache size 9-21
  choosing the execution platform 9-3
  controlling cache size 9-18
  controlling the locus of execution 9-2
  controlling the replication level 9-14
  cost of data management 9-3
  locus of transaction control 9-4
  preventing transient stubs 9-15
  using forwarders 9-18
  using GemStone Smalltalk for searching and sorting large objects 9-3
  using GemStone user actions and client Smalltalk primitives 9-21
  watching stub activity 9-4
OTrue (predefined object) A-3

P

ParcPlace Smalltalk emergency handler in GemBuilder 11-4
password
  GemStone 2-4, 2-6
  host 2-5
performance
  determining bottlenecks 9-4
  implications in locus of execution 9-2
  reducing conflict and 5-16
  tuning
    and explicit stubbing 9-16
    and fault levels 9-14
    and local caching 9-16
    and multithreaded applications 9-21
    and TraversalBufferSize 9-15
    by manual marking of dirty objects 9-19
    caching issues 9-5
    changing the initial cache size 9-21
    choosing the execution platform 9-3
    controlling cache size 9-18
    controlling the locus of execution 9-2
    controlling the replication level 9-14
    cost of data management 9-3
    database searching and sorting 9-3
    locus of transaction control 9-4
    minimizing faulting of dirty GemStone objects 9-14
    preventing transient stubs 9-15
    using forwarders 9-18

using GemStone Smalltalk user actions and client Smalltalk primitives 9-21
watching stub activity 9-4
pointer objects 8-22
pool dictionaries in GemStone 8-22
pool variables in GemStone 8-22
postconnect action
changing 3-14
postconnect actions 3-8
updateGS 3-14, 4-17
`postFault` message 4-11
predefined GemStone objects A-3
`preFault` message 4-11
preventing stubbing of objects 4-7
preventing transient stubs 9-15
primitive methods and breakpoints 8-33
primitives and user actions 9-21
private classes in GemBuilder 10-4
private instance variables 8-24
defined 8-24
private methods
defined 8-16
private methods in GemBuilder 10-4
privileges 6-3
Proceed command
in GemStone Debugger 8-36, 8-37, 8-39, 8-44
proceeding
with GemStone execution 8-39
profiling
GemStone Smalltalk execution 9-4
ProfMonitor class 9-4
protecting methods 6-2
Proxies 10-6
proxies 10-5–10-10
converting to replicates 10-10
for GemStone objects 10-5
sending messages through 10-6
public methods
defined 8-16

R

RcBag class 5-16
RcCounter class 5-16
RcKeyValueDictionary class 5-16
RcQueue class 5-17
read lock messages
`readLock:` (GbsSession) 5-13
`readLock:ifDenied:ifChanged:` (GbsSession) 5-13
`readLockAll:` (GbsSession) 5-13
`readLockAll:ifIncomplete:` (GbsSession) 5-14
read operations 5-10
read set 5-10
maintenance
neglecting 9-10
read / write transaction conflicts 5-10–5-16
receiver inspector 8-38
reduced-conflict classes 5-16–5-17
performance and 5-16
RcBag 5-16
RcCounter 5-16
RcKeyValueDictionary 5-16
RcQueue 5-17
storage and 5-16
reducing the number of objects in Smalltalk 9-16
releasing locks 5-15
remote perform messages
`remotePerform:` (GbsObject) 10-6
`remotePerform:with:` (GbsObject) 10-6
`remotePerform:with:with:` (GbsObject) 10-6
`remotePerform:withArgs:` (GbsObject) 10-6

remove command
in Browser's Class menu 7-9, 8-14
in Browser's Message menu 8-15, 8-16
in Browser's Symbol List menu 8-11
in GemStone Object Inspector 8-7
in GemStone System Browser's Symbol List pane 8-11
`removeFromCommitOrAbortReleaseLocksSet:` (System) 5-15
`removeFromCommitReleaseLocksSet:` (System) 5-15
removeInvalidConnectors configuration parameter 9-6
`removeInvalidConnectors` message 9-10
`removeLock:(GbsSession)` 5-14
`removeLockAll:(GbsSession)` 5-14
`removeLocksForSession:(GbsSession)` 5-14
`removeMakeGSTransparent` message 4-14
`removeParameters` (GbsSession) 2-6
removing
a method 8-15, 8-16
locks 5-14
rename command
in Browser's Symbol List menu 8-11
renaming a dictionary 8-11
replicated objects 4-5
and fault control 9-14
flushing dirty 9-3
replicates 4-2–4-18
replication
nontransparent 10-13
of client Smalltalk Blocks 4-20
`replicationSpec` message 4-9, 4-19
repository 1-2
modifying 5-10
Repository class 6-4
reserved OOPs A-3
`resolveSymbol:`, `resolveSymbol:ifAbsent:` 10-5
resuming execution after a notifier 8-37
root objects 3-2–3-4
RPC session 2-2
RT_ERR_SIGNAL_ABORT 5-7
runtime errors
and notifiers 8-25
during **definition** command (Browser's Class menu) 8-25

S

saving
GemStone class and method definitions in files 8-28
saving login information 2-9
schema 7-2
coordination between the client Smalltalk and GemStone 7-6
modification
support for 7-2
schema modification
and versions of classes 7-2, 7-6
security mechanisms in GemStone
login authorization 6-2
privileges 6-3
protecting methods 6-2
Segment class 6-4
Segment Tool 6-10, 6-16
changing a default segment 6-18
changing authorization 6-17
displaying segments 6-11
examining authorization 6-16
File menu 6-13
Group menu 6-15
Help menu 6-16
Member menu 6-15
Report menu 6-15
Segment menu 6-14
segments
and authorization 6-6, 6-7
changing authorization 6-17
checking authorization 6-16
group assignment 6-12
Segment Tool 6-10

selection blocks
in GemStone C-2
send command
in GemStone Debuggerwindow 8-40
senders command
in GemStone Browser's Message menu 8-15
in GemStone Debugger 8-39
sending messages
in the GemStone Debugger 8-40
to GemStone objects 10-6
service name
Gem 2-5
session
control 2-3
classes for 2-3
current 2-10, 2-12
current session 2-2
dependents 2-13–2-17
linked vs. RPC 2-2
logging in 2-2, 2-10, 2-12
logging out 2-11, 2-13
persistence of notify set in 5-17
removing a 2-9
Session Browser 2-7–2-13
session connectors 3-5
session parameters 2-4–2-6, 2-8
See also GbsSessionParameters class
sessions 2-1–2-17
multiple 2-10
setting
breakpoints 8-43
configuration parameters 9-6
locks 5-13
shared data
modifying 5-10
shared dictionaries 6-9, 6-10
shared variables 8-22, 8-23
`shouldBeCached` method 9-18
skip to caret command
in GemStone Debugger 8-40
`SmallInteger` (predefined object) 10-5
Smalltalk
GemStone 1-5
spawn command
in Browser's Class menu 7-8
in GemStone Browser's Class menu 8-13
in GemStone System Browser's Symbol List menu 8-11, 8-15
spawn hierarchy command
in Browser's Class menu 7-8
in GemStone Browser's Class menu 8-13
spawn versions command
in GemStone Browser's Class menu 8-13
special GBSM classes A-1
special methods and breakpoints 8-34
SpecialGemStoneObjects dictionary 4-18, 10-5
SpecialGemstoneObjects dictionary A-3
stack
copying contents of 8-37
examining in GemStone 8-38
viewing GemStone Smallatlk contexts 8-39
viewing GemStone Smalltalk contexts 8-39
stack pane in GemStone Debugger 8-39
step command
in GemStone Debugger 8-40
step points 8-31
examining 8-38
methods that have no 8-34
stepping through code 8-31, 8-40
Stone
repository monitor 1-2, 1-3
storage
reduced-conflict classes and 5-16
structural access to GemStone objects 10-11–10-12
stubbing of objects 4-5, 4-12
controlling the stub level 9-14
explicit 9-16
explicit control of 4-8
preventing 4-7

stubs
and fault control 9-14
defunct 4-8
explicit control of 9-16
observing activity of 9-4
preventing transient 9-15
watching activity of 9-4
`stubYourself` message 9-16
subclass creation methods in GemStone 8-21, 8-24
support vi
symbol dictionaries 6-19
changing 8-11
Symbol List Browser 6-9, 6-10, 6-18
copying and pasting objects 6-19
Dictionaries pane 6-19
File menu 6-20
text entry fields 6-22
Symbol List pane in GemStone System Browser 8-9
symbol lists 6-9
synchronization
of client Smalltalk and GemStone objects 4-4
SystemRepository
segments in 6-11

T

technical support vi
temporary variables
inspecting 8-39, 8-41
Terminate command
in GemStone Debugger 8-36, 8-37, 8-44
threadSafeCaches configuration parameter 9-6, 9-10
tools, GemBuilder 1-6
Topaz commands supported by GemStone's file in mechanism 8-30
Topaz format (used for file out) 8-29
transaction
automatic mode
defined 5-8
transactions 5-1–5-19
aborting 5-5
committing
and performance 5-16
management of 2-9, 5-2
managing 5-4
modes 5-8–5-10
automatic 5-8, 5-9
manual 5-9
switching between 5-9
transient object stubs
preventing 9-15
transitive closure 3-3
transparency 1-4
and kernel classes 4-14
caches
controlling size of 9-18
trap-door message passing to GemStone objects 10-6, 10-7
TraversalBufferSize (configuration parameter) 9-15
traversalBufferSize (method) 9-10, 9-15
traversalBufferSize configuration parameter 9-6
`True`
(predefined GemStone object) 10-5
`True` (predefined GemStone object) A-3

U

update command
in Browser's Symbol List menu 8-11
in GemStone Breakpoint Browser 8-43
in GemStone Object Inspector 8-6
in GemStone System Browser's Symbol List pane. 8-11
updateGS postconnect action 4-17
updating
the Breakpoint Browser 8-43
User Account Manager 6-10, 6-22
Database User dialog 6-23, 6-24
Database Users list 6-22

user actions 9-21
and primitives 9-21
UserClasses
client Smalltalk Browser category 4-18
GemStone Browser symbol list dictionary 4-18
user-defined errors 11-1, 11-3
UserGlobals dictionary 6-9, 6-10
username
GemStone 2-4
host 2-5
UserProfile class 6-5
using forwarders 9-18

V

variables
class 8-23
class instance 8-23
inspecting in GemStone Debugger 8-41
pool 8-22
shared 8-22, 8-23
verbose configuration parameter 9-6, 9-10
verification by connectors 9-8
verification of connectors 3-13
versions of classes 7-2, 7-6

W

write lock messages
`writeLock:` (GbsSession) 5-13
`writeLock:ifDenied:ifChanged:` (GbsSession) 5-13
`writeLockAll:` (GbsSession) 5-13
`writeLockAll:ifIncomplete:` (GbsSession) 5-14
write operations 5-10
write set 5-10
write / write transaction conflicts 5-10–5-16
and RcQueue class 5-17