GemStone®

GemBuilder for Smalltalk

For use with IBM VisualAge Smalltalk Enterprise

December 2001

GemStone/S

GemBuilder for Smalltalk Version 5.2

IMPORTANT NOTICE

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 GemBuilder for Smalltalk will function on all Smalltalk images.

Trademarks

Preface

About This Manual

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

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.

Prerequisites

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 VisualAge Smalltalk language and programming environment as described in the VisualAge Smalltalk 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, describes the various mechanisms GemBuilder can use to coordinate your application's local objects with objects in the GemStone repository, thus making them persistent and sharable.

Chapter 4, Connectors, explains how to connect your application's local objects to objects in the GemStone repository in order to implement object-sharing and allow your application to manipulate objects in the repository.

Chapter 5, 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 6, 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 7, Security and Object Access, 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 8, Schema Modification and Coordination, explains how GemStone supports schema modification by maintaining versions of classes in class histories. It describes the Class Version Browser and explains how to use it. It also explains how to synchronize schema modifications between the client image and GemStone.

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 low-level approaches to tuning GemBuilder applications.

Chapter 11, Error-handling, discusses errors: how to handle them and how to recover from them.

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

Appendix B, Packaging Runtime Applications, provides brief instructions for packaging runtime applications.

Appendix C, 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 D, Client 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 GemStone Smalltalk and discusses managing common operations.
- A description of the behavior of each GemStone kernel class is available in the class comments in the GemStone Smalltalk image.
- 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/S provides several sources for product information and support. GemStone/S product manuals provide extensive documentation, and should always be your first source of information. GemStone/S 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/S 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/S account manager.

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

- Your name, company name, and GemStone/S license number,
- the GemStone/S 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/S 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 web form, email, 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.

World Wide Web: **http://support.gemstone.com**
The preferred method of contact. The Help Request link is at the top right corner of the home page—please use this to submit help requests. 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.

Email: **support@gemstone.com**

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.

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) 533-3503**

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/S offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 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/S 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 2-7
Removing Session Parameters 2-8
2.4 Logging In To and Logging Out Of GemStone 2-8
Logging In To GemStone Programmatically. 2-9
The Current Session. 2-9
Logging Out of GemStone Programmatically 2-10
Session Management Using the Session Browser 2-11
Logging In 2-11
Setting the Current Session. 2-11
Logging Out of a GemStone Session With the Session Browser 2-11
2.5 Session Dependents 2-12

Chapter 3. Sharing Objects

3.1 Which Objects to Share? 3-2
Connect Systems at the Root. 3-4
3.2 Class Mapping 3-6
Automatic Class Generation 3-7
Schema Mapping 3-8
Behavior Mapping 3-8
Mapping and Class Versions 3-8
3.3 Forwarders 3-9
Sending Messages. 3-10
Arguments 3-10
Results 3-10
Defunct Forwarders 3-11
3.4 Replicates 3-12
Synchronizing State. 3-12
Faulting. 3-13
Flushing 3-14
Minimizing Replication Cost 3-14
Instance Variable Mapping. 3-14
Stubbing 3-17
Replication Specifications 3-21

Customized Flushing and Faulting 3-27
Modifying Instance Variables During Faulting. 3-27
Modifying Instance Variables During Flushing 3-29
Mapping Classes With Different Formats. 3-31
Limits on Replication 3-31
Replicating Client Smalltalk Blocks 3-32
Block Callbacks 3-35
Replicating ScaledDecimals 3-36
Client Copies. 3-37
3.5 Precedence of Replication Mechanisms 3-38
3.6 Converting Between Forms 3-39

Chapter 4. Connectors

4.1 Connecting and Disconnecting 4-2
Scope 4-2
Verifying Connections 4-2
Initializing 4-3
Updating Class Definitions 4-4
4.2 Kinds of Connectors 4-4
Connection Order 4-5
Lookup 4-5
Connecting by Identity: Fast Connectors 4-6
4.3 Making and Managing Connectors 4-7
Making Connectors Programmatically 4-7
Creating Connectors. 4-8
Setting the Postconnect Action 4-8
Adding Connectors to a Connector List. 4-9
Session Control. 4-10
The Connector Browser. 4-13
The Group Pane 4-14
The Connector Pane 4-14
The Control Panel 4-14
Postconnect Action 4-15

Chapter 5. Using the GemStone Programming Tools

5.1 GemStone Menu 5-2

5.2 Browsing Code 5-4
The File Menu 5-6
The GemStone Menu 5-7
The Classes Menu. 5-7
Pop-up Text Pane Menu 5-8
5.3 Other GemStone Tools 5-9
GemStone Workspaces 5-10
The System Workspace. 5-10
Inspectors 5-11
Inspecting Nonsequenceable Collections 5-11
5.4 Coding 5-12
About GemStone Smalltalk Classes. 5-12
Defining a New Class. 5-13
Subclass Creation Methods. 5-16
Private Instance Variables 5-17
Modifying an Existing Class 5-17
Defining Methods. 5-18
Public and Private Methods 5-18
Reserved and Optimized Selectors 5-19
Saving Class and Method Definitions in Files 5-19
Handling Errors While Filing In 5-22
5.5 Debugging. 5-22
Breakpoints 5-23
Breakpoints for Primitive Methods 5-25
Breakpoints for Optimized Methods 5-25
Tools 5-26
The Breakpoint Browser 5-26
The Debugger 5-27
Getting a Stack Trace Without a Debugger 5-28

Chapter 6. Managing Transactions

6.1 Transaction Management: an Overview 6-2
6.2 Operating Inside a Transaction 6-3
Committing a Transaction 6-4
Aborting a Transaction. 6-5
Avoiding or Handling Commit Failures 6-5
6.3 Operating Outside a Transaction 6-6

Being Signaled to Abort 6-7
6.4 Transaction Modes 6-8
Automatic Transaction Mode 6-8
Manual Transaction Mode 6-9
Choosing Which Mode to Use 6-9
Switching Between Modes 6-10
6.5 Managing Concurrent Transactions 6-10
Read and Write Operations 6-11
Optimistic and Pessimistic Concurrency Control 6-11
Setting the Concurrency Mode. 6-12
Setting Locks. 6-13
Releasing Locks Upon Aborting or Committing. 6-15
6.6 Reduced-conflict Classes. 6-16
6.7 Changed Object Notification. 6-17
Gem-to-Gem Notification 6-18

Chapter 7. Security and Object Access

7.1 Object-Level Security 7-2
Requiring Login Authorization 7-2
Controlling Visibility of Objects 7-2
Protecting Methods 7-2
Using GemStone’s Authorization Mechanisms 7-2
7.2 Classes for Controlling Access to Objects 7-4
Repository 7-4
Segment 7-4
UserProfile 7-5
7.3 Sharing Access to Objects 7-6
Group Authorization and Object-sharing 7-7
Using Segments for Authorization. 7-7
Making Objects Accessible Through Symbol Lists 7-9
7.4 GemStone Administration Tools 7-10
The Segment Tool 7-10
Segment Definition Area 7-11
Group Definition Area 7-13
Segment Tool Menus 7-14
Using the Segment Tool. 7-17
The Symbol List Browser 7-20

The Clipboard 7-21
Symbol List Browser Menus 7-21
User Account Management Tools. 7-23
GemStone User List 7-23
GemStone User Dialog 7-25

Chapter 8. Schema Modification and Coordination

8.1 Schema Modification. 8-2
Instance Migration Within GemStone 8-2
Setting the Migration Destination 8-3
Migrating Objects 8-3
Things to Watch Out For 8-4
Instance Variable Mapping in Migration 8-4
8.2 Schema Coordination 8-6
8.3 The Class Version Browser 8-7
Menus in the Class Version Browser 8-7

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
9.2 Profiling 9-4
Profiling Client Smalltalk Execution 9-4
Watching Stub Activity. 9-4
Using Verbose Mode 9-4
9.3 Configuring GemBuilder 9-5
GemBuilder Configuration Parameters 9-5
Using Configuration Parameters to Tune Your Application 9-6
The Settings Browser 9-11
Opening the Settings Browser 9-11
Parameter Notebook 9-13
9.4 Replication Tuning 9-14
Controlling the Fault Level 9-15
Preventing Transient Stubs 9-15

Setting theTraversal Buffer Size 9-15
9.5 Optimizing Space Management 9-16
Explicit Stubbing 9-16
Using Forwarders 9-17
Not Caching Selected Objects 9-17
9.6 Using Primitives 9-18
9.7 Changing the Initial Cache Size 9-19
9.8 Multiprocess Applications 9-19
Process-safe Transparency Caches 9-19
Blocking and Nonblocking Protocol. 9-19
One Process per Session 9-20
Multiple Processes per Session. 9-20
Coordinating Transaction Boundaries. 9-21
Coordinating Flushing 9-21
Coordinating Faulting. 9-22

Chapter 10. Nontransparent Access to GemStone Objects

10.1 Nontransparency: General Principles 10-2
Flushing and Faulting Nontransparent Objects 10-2
Public and Private Classes and Methods 10-4
Specifying a Session. 10-4
10.2 Delegate Objects 10-5
Sending Messages Through GbsObject Delegates 10-6
Special Treatment of Binary Selectors. 10-8
Sending Code to Gemstone for Execution 10-8
Converting GbsObjects to Replicates 10-10
10.3 Structural Access to GemStone Objects 10-11
10.4 Executing GemStone Host File Access Methods 10-13

Chapter 11. Error-handling

11.1 Error-handling and Recovery 11-1
Stack-based Error-handling 11-2
Session-based Error-handling 11-2
User-defined Errors 11-3
11.2 Detecting GemStone Interrupts 11-4

Appendix A. GemBuilder Classes and GbsObjects

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. Packaging Runtime Applications

B.1 Prerequisites B-1
Names B-1
Replicating Blocks B-2
Defunct Stubs and Forwarders B-2
Shared Libraries. B-2
B.2 Packaging B-2

Appendix C. Network Resource String Syntax

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

Appendix D. Client Smalltalk and GemStone Smalltalk

Index

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 GemStone Session Browser 2-7
Figure 2.3. The Login Editor 2-8
Figure 2.4. Committing with Approval From a Session Dependent 2-15
Figure 3.1. Connecting Application Roots 3-5
Figure 3.2. Root Objects 3-6
Figure 3.3. Two-level Fault of an Object 3-18
Figure 3.4. A Stub Responds to a Message 3-19
Figure 4.1. Connecting a Name Connector 4-6
Figure 4.2. Connector Class Hierarchy 4-7
Figure 4.3. The Connector Browser 4-13
Figure 5.1. GemStone Classes Browser 5-4
Figure 5.2. Menus in the GemStone Browser 5-6
Figure 5.3. GemStone System Workspace 5-10
Figure 5.4. GemStone Delegate Inspector 5-11
Figure 5.5. GemStone Breakpoint Browser with a Breakpoint 5-27
Figure 6.1. GemBuilder Application Workspace 6-3

Figure 7.1. GemStone’s Object-Level Security Mechanism 7-5
Figure 7.2. The Segment Tool. 7-11
Figure 7.3. The Symbol List Browser. 7-20
Figure 7.4. GemStone User List. 7-24
Figure 7.5. GemStone User Dialog 7-25
Figure 7.6. Privileges Dialog in GemStone User Window. 7-26
Figure 8.1. The Class Version Browser 8-7
Figure 9.1. The Settings Browser 9-12
Figure 9.2. Employee Set Faulted into the Client Smalltalk 9-16
Figure 10.1. Transparent Object 10-3
Figure 10.2. Nontransparent Objects. 10-3

List of Tables

Table 3.1. Delegate Conversion Protocol. 3-39
Table 3.2. Forwarder (to the Server) Conversion Protocol 3-39
Table 3.3. Replicate Conversion Protocol 3-40
Table 3.4. Stub Conversion Protocol 3-40
Table 3.5. Conversion Protocol for Unshared Client Objects 3-41
Table 4.1. Group List Menu in the Connector Browser 4-14
Table 4.2. Connectors Menu in the Connector Browser 4-14
Table 4.3. Options in the Control Panel 4-15
Table 4.4. Postconnect Action Options in the Connector Browser 4-15
Table 5.1. The GemStone Menu. 5-2
Table 5.2. File Menu in the GemStone Browser 5-6
Table 5.3. GemStone menu in the GemStone Browser 5-7
Table 5.4. GemStone Browser’s Classes Menu .5-8
Table 5.5. Pop-up Menu in GemStone Browser’s Text Pane. 5-9
Table 5.6. Additional GemStone Menu Items .5-9
Table 5.7. Commands in GemStone Inspector 5-11
Table 5.8. Commands for Inspecting NSCs 5-11
Table 6.1. GbsSession Methods for Running Outside of a Transaction. 6-7

Table 7.1. File Menu in the Segment Tool 7-14
Table 7.2. Segment Menu in the Segment Tool 7-15
Table 7.3. Group Menu in the Segment Tool 7-16
Table 7.4. Member Menu in the Segment Tool 7-16
Table 7.5. Report Menu in the Segment Tool 7-17
Table 7.6. File Menu in the Symbol List Browser 7-21
Table 7.7. Dictionary Menu in the Symbol List Browser. 7-22
Table 7.8. Entry Menu in the Symbol List Browser. 7-22
Table 7.9. GemStone User List: File Menu. 7-24
Table 7.10. GemStone User List: Users Menu 7-24
Table 7.11. Buttons in the GemStone User Window 7-25
Table 7.12. Privileges 7-27
Table 8.1. Classes Menu in Class Version Browser. 8-8
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.

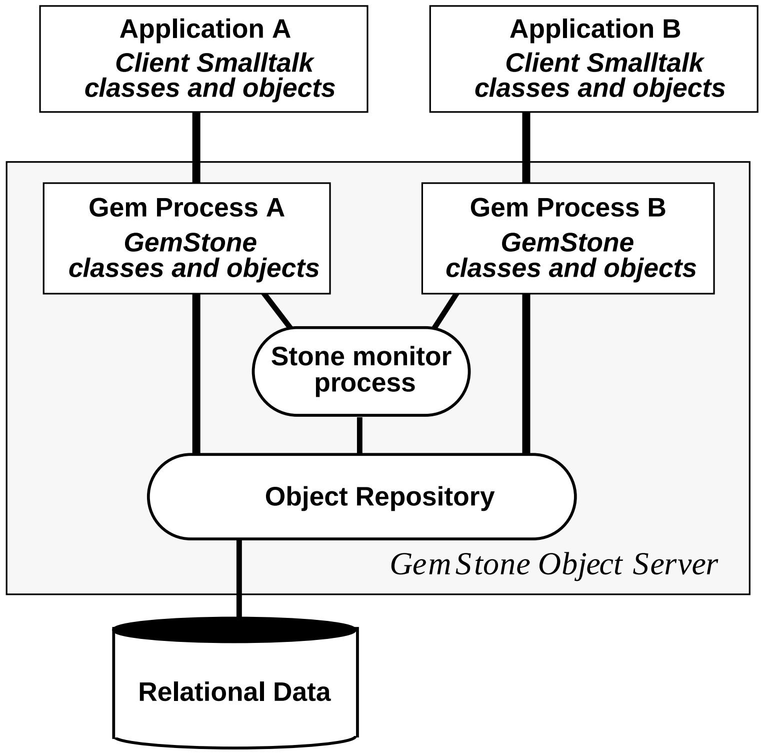
1.1 The GemStone Object Server

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


Application A
Client Smalltalk classes and objects
Application B
Client Smalltalk classes and objects
Gem Process A
GemStone classes and objects
Gem Process B
GemStone classes and objects
Stone monitor process
Object Repository
GemStone Object Server
Relational Data

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 4 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 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 D.

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 expressions.
- *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 7.

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 4.

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 6.

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 "Tuning 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
describes how to log in and out of GemStone sessions programmatically and 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

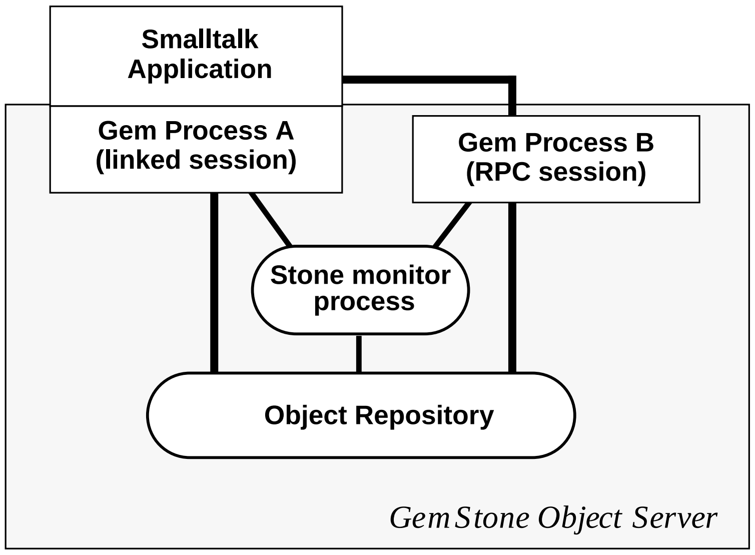
A Gem can run as a separate process and respond to Remote Procedure Calls (RPCs) from its application, in which case the session it supports is called an *RPC* session.

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. When running linked, an application and its Gem must run on the same machine and the runtime code requires additional memory.

An RPC session 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 do various operations:

- 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 "gemserver60" on a node named "mozart", specify an NRS string of the form:

  ```
  !@mozart!gemserver60
  ```

  (Appendix C 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 7-23.) 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!gemserver60'
   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

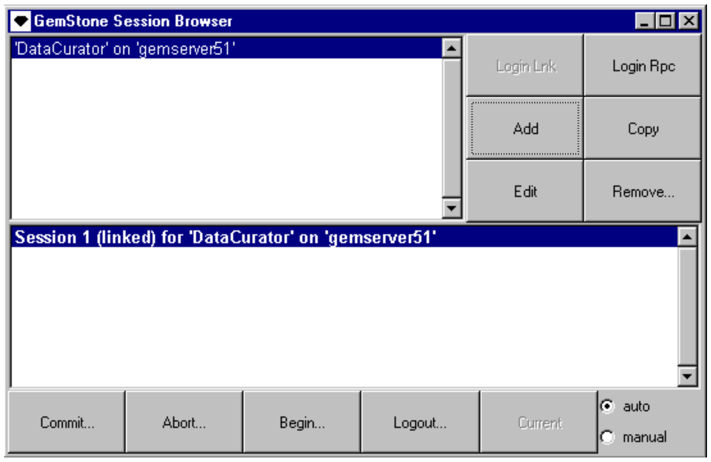
The GemStone Session Browser streamlines logging in and logging out of GemStone and managing sessions and transactions. This section explains how to invoke the Session Browser, and how to use it to define session parameters and to log in and out of GemStone.

Starting the Session Browser

1. Start your GemBuilder for Smalltalk image.
2. Select **Sessions** from the GemStone menu to open a Session Browser.

   Figure 2.2 shows the Session Browser.

Figure 2.2 The GemStone Session Browser



'DataCurator' on 'gemserver51'

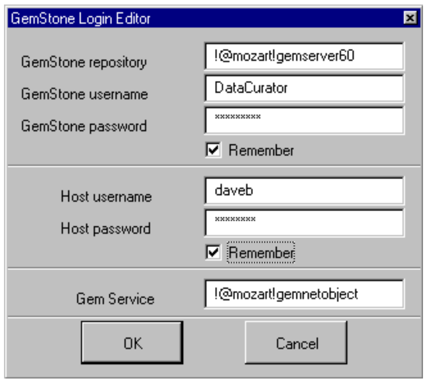
Session 1 (linked) for 'DataCurator' on 'gemserver51'

(● auto ○ manual)

Supplying Session Parameters

Select the **Add** button to define a set of session parameters. A Login Editor appears, as shown in Figure 2.3.

Figure 2.3 The Login Editor



GemStone repository	!@mozart!gemserver60
GemStone username	DataCurator
GemStone password	*********
	☑ Remember
Host username	daveb
Host password	********
	☑ Remember
Gem Service	!@mozart!gemnetobject

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.

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.

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 ]
   ifFalse: [
             "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.

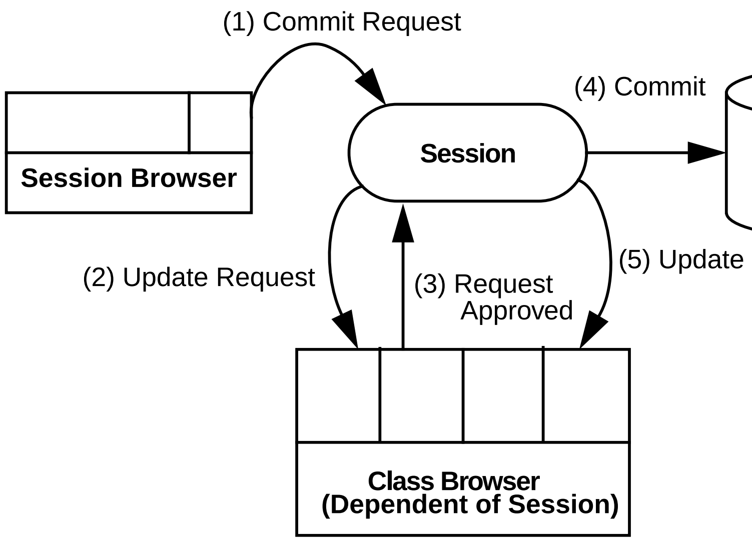
Example 2.4

```
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 2.4 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 2.4 Committing with Approval From a Session Dependent



Chapter 3 Sharing Objects

This chapter describes how GemBuilder shares objects with the GemStone/S object repository.

Which Objects to Share?
is an overview of the process of determining how to make good use of GemBuilder's resources, introducing the available mechanisms: forwarders, replicates, and local copies; and finally describing connectors, GemBuilder's way to associate pairs of objects in the two object spaces.

Class Mapping
explains how classes are defined and how forwarders, stubs, and replicates depend on them.

Forwarders
explains how to use forwarders to store all an object's state and behavior in one object space.

Replicates
explains replicating GemStone objects in client Smalltalk, or vice-versa; describes the processes of propagating changes to keep objects synchronized; presents various mechanisms to minimize performance costs; presents further details; last, discusses local copies.

Precedence of Replication Mechanisms
discusses the various ways replication mechanisms interact, and describes how to determine whether an application object becomes a forwarder, stub, or replicate.

Converting Between Forms
lists protocol for converting from and to delegates, forwarders, stubs, replicates, and unshared client objects.

3.1 Which Objects to Share?

Working with your client Smalltalk, you had one execution engine—the compiler—acting on one object space—your image. Now that you've installed GemBuilder, you have *two* execution engines and *two* object spaces, one of which is a full-fledged object repository for multiuser concurrent access, with transaction control, security protections, backups and logging.

What's the best way to make use of these new resources?

Objects represent both state and behavior. Therefore, you have two basic decisions:

- Which state should reside on the client, which on the server, and which in both object spaces?
- Which behavior should reside on the client, which on the server, and which in both object spaces?

Ultimately, the answer is dictated by the unique logic of your specific problem and solution, but these common patterns emerge:

Client presents user interface only; state (domain objects) and application logic reside on server; server executes all but user interface code. A web-based application that uses the client merely to manage the browser needs little functionality on the client, and what it does need is cleanly delimited.

State reside on both client and server; client manages most execution; server is used mainly as a database. A Department of Motor Vehicles could use a repository of driver and vehicle information, properly defined, for a bevy of fairly straightforward client applications to manage driver's licenses, parking permits, commercial licenses, hauling permits, taxation, and fines.

Execution occurs, and therefore state resides, on both client and server. At specified intervals, clients of a nationwide ticket-booking network download the current state of specific theaters on specific dates. Clients book seats and update their local copies of theaters until they next connect to the repository. To resolve conflicts, server and client engage in a complex negotiation.

For these and other solutions, GemBuilder provides several kinds of client- and server-side objects, and a mechanism—*a connector*—for describing the association between pairs of objects across the two object spaces.

Four kinds of objects help a GemBuilder client and a GemStoneS repository share state and execution: forwarders, stubs, replicates, and local copies.

Forwarder — is a *proxy:* a simple client object that knows only which GemStone object it is associated with. It responds to a message by passing it to its associated master object in the other object space, where state is stored and execution occurs remotely.

Replicate — is a copy associated with a particular object in the other object space. It copies some or all of the other object's state, which it synchronizes at appropriate times; it implements all messages it expects to receive; it executes locally.

Stub — is a proxy that responds to a message by bringing across a replicate of its counterpart object. Stubbing is a way to minimize memory use and network traffic by bringing only what is needed when it is needed.

Local Copy — is a quickly made copy of a specified object suitable for read-once access, with no knowledge of the other object space.

To help implement the functionality embodied in forwarders, stubs, and replicates, GemBuilder uses *delegates:*

Delegate — An instance of the class GbsObject that holds the object identifier (OOP) of the server object with which a client object is associated. Each forwarder is associated with one and only one delegate per GemBuilder image; each replicate or stub is associated with one and only one delegate per session. The delegate holds a session identifier, if necessary, and various flags indicating characteristics of the associated objects, such as whether one is a forwarder. Delegates also help keep track of the details required for garbage collection.

A *connector* is the mechanism by which an object in one object space refers to another in the other object space:

Connector — associates a client object with a server object, resolving objects by name, by class, or other ways. When connected, they synchronize data or pass messages in either direction or take no action at all, as specified.

Whatever combination of these elements your application requires, subsystems of objects will probably reside on both the client and the server. Some subset of these subsystems will need state or behavior on both sides: some objects will be shared.

Connect Systems at the Root

A connector connects more than the specified client object to the specified server object: through transitive reference, a connector connects whole networks of objects. Most objects (except atomic objects—characters, booleans, small integers, `nil`) refer to others, for example, their instance variables. And their instance variables refer to *their* instance variables, and so on, branch and twig, until you reach the leaves of a large network of objects with a treelike structure.

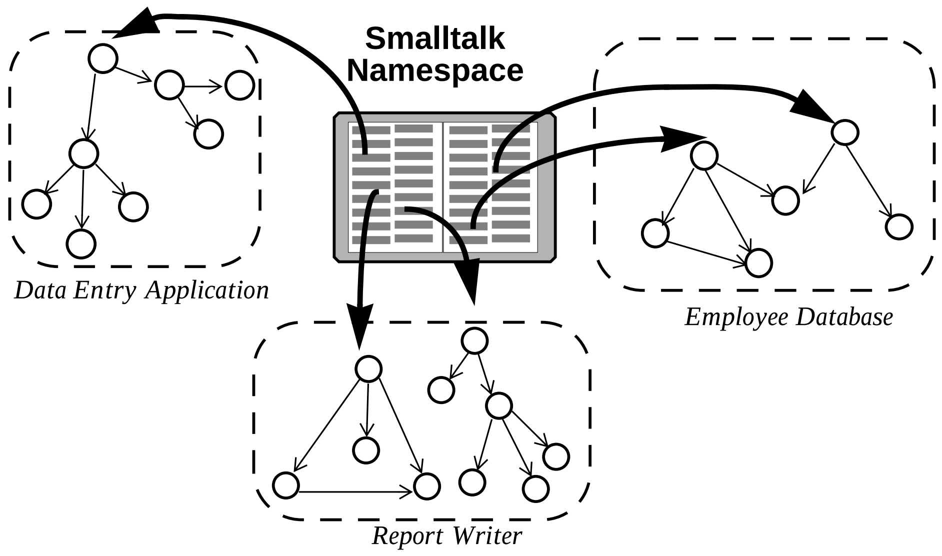
You can take advantage of this hierarchical structure to minimize application overhead. Identify the object at the root of each subsystem of shared objects, and then connect only these root objects. Depending on how you've defined configuration parameters and related matters, you can synchronize entire subsystems in GemStone/S this way. After you've connected the application's roots, GemBuilder automatically manages all the objects referenced from these roots according to particulars you specify when you define the connector.

Root objects are often:

- global variables,
- class variables, or
- class instance variables.

Figure 3.1 shows an application in which several connected objects are accessed through global variables in the Smalltalk namespace. 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.

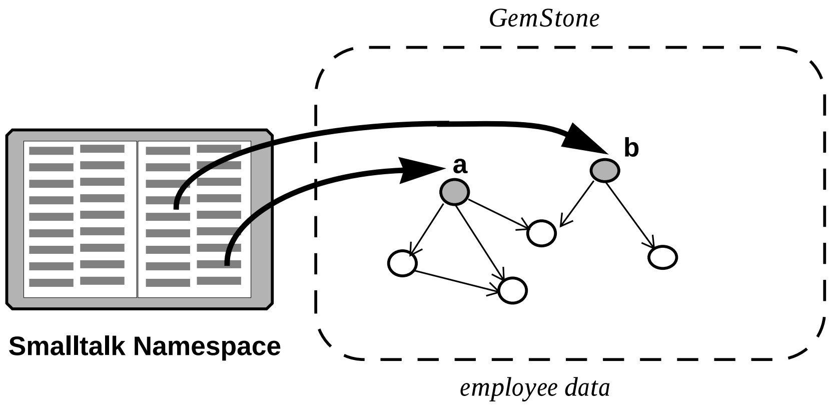
Figure 3.1 Connecting Application Roots



The data entry application and the report writer reside in the Smalltalk image; however, the employee database is stored in GemStone, as it defines a large amount of persistent data that other users may need to share, data that benefits from GemStone’s capacity, stability, robustness, and fast searches.

Figure 3.2 shows the state of the employee data when stored in GemStone:

Figure 3.2 Root Objects



In Figure 3.2, objects **a** and **b** are *root* objects: those objects from which all others can be reached by *transitive closure*: by direct reference, or by indirect reference through any number of layers.

The above discussion has focused on shared instances from your applications, but in order to share instances in any way, GemBuilder and GemStone must first share definitions for each class of shared instance.

3.2 Class Mapping

Before GemBuilder can replicate an object, it must know the respective structures of client and repository object and the mapping between them. Although not strictly necessary for forwarders, this knowledge improves forwarding performance, saving GemBuilder an extra network round-trip.

GemBuilder uses class definitions to determine object structure. To replicate an object:

- both client and repository must define the class, and
- the two classes must be connected using a *class connector*.

GemBuilder uses this mapping for all replication, whether at login or later.

Unlike connectors for replicates or forwarders, class connectors have no default update direction. If class definitions differ on the client and the server, it is usually for a good reason; you probably don't want to update GemStone with the client Smalltalk class definition, or vice-versa.

GemBuilder predefines special connectors, called *fast connectors,* for the GemStone kernel classes. For more information about fast connectors, see "Connecting by Identity: Fast Connectors" on page 4-6.

Automatic Class Generation

By default, GemBuilder generates class definitions and connectors automatically as necessary. If GemBuilder requires GemStone to replicate an instance of a Smalltalk class not already defined in GemStone, then at first access, GemBuilder generates a GemStone class having the same schema and position in the hierarchy, and a class connector connecting it to the appropriate client class. Conversely, if the client must replicate an instance of a GemStone class not already defined in client Smalltalk, GemBuilder generates the client Smalltalk class and the appropriate class connector. If superclasses are also undefined, GemBuilder generates the complete superclass hierarchy, as necessary.

You can control automatic class generation with the configuration parameters **`generateGSClasses`** and **`generateSTClasses`** (described on page 9-9). These settings are global to your image.

- If you disable automatic generation of GemStone classes by setting **`generateGSClasses`** to `false`, situations that would otherwise generate a GemStone class instead raise the signal `GbsError gbsClassGenerationFailed`.
- If you disable automatic generation of client Smalltalk classes by setting **`generateSTClasses`** to `false`, situations that would otherwise generate a client Smalltalk class instead raise an exception.
- You can disable class connector generation by setting **`generateClassConnectors`** to `false`. If you do so, GemBuilder generates classes, but not their corresponding connectors.

GemBuilder deposits automatically generated GemStone classes in the GemStone symbol dictionary UserClasses, which it creates if necessary. Automatically generated client Smalltalk classes are deposited in an application named UserClasses.

NOTE
To avoid undesirable results, do not rely on automatic class generation to replicate instances of classes that you have customized with replication specifications (described on page 3-21). Instead, make sure these classes

are connected before your application tries to access the corresponding server objects, because automatic class generation ignores replication specifications.

Schema Mapping

By default, when you connect a client class with a GemStone class using a class connector, GemBuilder automatically maps all instance variables whose names match, regardless of the order in which they are stored. (You can change this default mapping to accommodate nonstandard situations.)

If you later change either of the mapped class definitions, GemBuilder automatically remaps identically named instance variables.

Behavior Mapping

Connected classes define structure, not behavior: replicated instances depend on methods implemented in the object space in which they execute. During development, it may be simplest to use GemBuilder's programming conveniences to implement behavior in both spaces. For reliability and ease of maintenance, however, some decide to remove unnecessary duplication from production systems and to define behavior only where it executes.

Mapping and Class Versions

Unlike the client Smalltalk language, GemStone Smalltalk defines *class versions:* when you change a class definition, you make a new version of the class, which is added to an associated class history. (For details, see the the chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide*.)

If you decide to update one class definition with the other, the result depends on the direction of the update:

- Updating a client Smalltalk class from a GemStone class regenerates the Smalltalk class and recompiles its methods.
- Updating a GemStone class from a client Smalltalk class creates a new GemStone version of the class.

NOTE
A class connector connects to a specific GemStone class version, the version that was in effect when the connector was connected. Instances of a given class version will not receive any messages sent by means of a connector connected to another class version.

3.3 Forwarders

The simplest way to share objects is with f*orwarders,* simple objects that know just one thing: to whom to forward a message. A forwarder is a proxy that responds to messages by forwarding them to its counterpart in the other object space.

Forwarders are particularly useful for large collections, frequent GemStone residents, whose size makes them expensive to replicate and cumbersome to handle in a client image.

Forwarders are of two kinds:

- The most common kind of forwarder is a *forwarder to the server:* a client Smalltalk object that knows only which GemStone object it represents. It responds to all messages by passing them to the appropriate GemStone object, where data resides and behavior is implemented. (For historical reasons, this is the kind of forwarder usually meant when a discussion merely says "forwarder." This kind of forwarder is also called a server forwarder.)
- A *forwarder to the client* is a GemStone object that knows only which client Smalltalk object it represents. It responds to all messages by passing them to the appropriate client Smalltalk object, where data resides and behavior is implemented.
- You can create forwarders in several ways:Declare a connector as a forwarder upon login. For example, connect the GemStone global variable BigDictionary as a forwarder to the server so that it isn't replicated in the client.
- Specify that a given instance variable must always appear in the other object space as a forwarder to the server (using a replication specification, discussed starting on page 3-21). For example, a reference application might implement a specification that declares the class variable Atlas as a forwarder to the server.
- Prefix `fw` to a method name to return a forwarder from any message-send to GemStone. For example, to return a forwarder from a GemStone name lookup, send the GbsSession method `fwat:` or `fwat:ifAbsent:` instead of `at:` or `at:ifAbsent:`.
- Create a forwarder to the server explicitly using the message `#asForwarder` to any instance of GbsObject. For example:

  `(GBSM execute: '`*someCode*`') asForwarder`

- Override all these by implementing a class method `instancesAreForwarders` to return `true`, and all instances of a given class are forwarders to the server.Subclasses of GbsServerClass already respond `true` to this message; GbsServerClass is an abstract class, and all instances that inherit from it become forwarders to the server. When sent to a class that inherits from GbsServerClass, the instance creation methods `new` and `new:` create a new instance of the class in GemStone and return a forwarder to that instance.

Sending Messages

When a forwarder to the server receives a message, it sends the message to its associated delegate, which in turn sends it to the GemStone counterpart whose object identifier it holds—presumably an instance that can respond meaningfully. The target object's response is then returned to the delegate, and through the delegate to the forwarder, which then returns the result.

When a forwarder to the client receives a message, it forwards the message to the full-fledged client object to which it is connected, returning the result to the client forwarder, which stores it in GemStone.

Arguments

Before a message is forwarded to GemStone, arguments are translated to GemStone objects. As a message is forwarded to the client, arguments are translated to client Smalltalk objects.

When an argument is a block of executable code, special care is required: for details, see "Replicating Client Smalltalk Blocks" on page 3-32 .

Results

The result of a message to a client forwarder is a GemStone Smalltalk object stored in the GemStone repository.

The result of a message to a server forwarder is the client Smalltalk object connected to the GemStone object returned by GemStone—usually a replicate, although a forwarder might be desirable under certain circumstances.

To enforce a forwarder result, prefixing the message to the forwarder with the characters `fw`. For example:

- `aForwarder at: 1` returns a *replicate* of the object at index 1.
- `aForwarder fwat: 1` returns a *forwarder* to the object at index 1.

Defunct Forwarders

A forwarder contains no state or behavior in one object space, relying on the existence of a valid instance in the other. When a session logs out of the server, communication between the two spaces is interrupted. Forwarders that relied on objects in that session can no longer function properly. If they receive a message, GemBuilder raises an error complaining of either an invalid session identifier or a defunct forwarder.

You cannot proceed from either of these errors; an operation that encounters one must restart (presumably after determining the cause and resolving the problem).

GemBuilder cannot safely assume that a given object will retain the same object identifier (OOP) from one session to the next. Therefore, you can't fix a defunct forwarder error simply by logging back in.

(If a connector has been defined for that object or for its root, then logging back in will indeed fix the error, because logging back in will connect the variables. But in that case, it's the connector, not the forwarder, that repairs damaged communications.)

Consider the following forwarder for the global BigDictionary:

Example 3.1

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

When a GemBuilder session logs into GemStone, BigDictionary becomes a valid forwarder to the current GemStone BigDictionary. But when no session is logged into GemStone, sending a message to BigDictionary results in a defunct forwarder error.

GemBuilder's configuration parameter **connectorNilling**, when true, assigns each connector's variables to `nil` on logout. This usually prevents defunct stub and forwarder errors, replacing them with `nil doesNotUnderstand` errors.

3.4 Replicates

Sometimes it's undesirable to dispatch a message to the other object space for execution—sometimes local execution is desirable, even necessary, for example, to reduce network traffic. When local state and behavior is required, share objects using replicates instead of forwarders. Replicates are particularly useful for small objects, objects having visual representations, and objects that are accessed often or in computationally intensive ways.

Like a forwarder, a *replicate* is a client Smalltalk object associated with a delegate that knows which GemStone object the replicate represents. Unlike a forwarder, replicates also hold (some) state and implement (some) behavior. Replicates have available a variety of mechanisms for synchronizing their state with their associated GemStone object.

For example, replicates must declare one of two default update directions: either the client image is presumed valid and updates the GemStone object, or GemStone is presumed valid and updates the client object. While connected, GemBuilder automatically updates the specified object at transaction boundaries when its replicate has changed.

To do so, GemBuilder must know about the structure of the two objects and the mapping between those structures. GemBuilder manages this mapping on a class basis: replicates must be instances of classes whose definitions are connected, by means of a class connector, to definitions of the corresponding class in the other object space. GemBuilder handles many obvious cases automatically, but nonstandard mappings require you to override certain instance and class methods from class Object's GemStone support protocol. Nonstandard mappings are discussed starting on page 3-14.

Synchronizing State

After a relationship has been established between a client object and a GemStone object, GemBuilder keeps their states synchronized by propagating changes as necessary—this is one of the jobs that delegates handle for replicates.

Either of two situations can require synchronization:

- A client object is modified in the Smalltalk image, leaving its GemStone repository counterpart out of date. The client object is now referred to as *dirty*.
- A GemStone object is modified in the repository, leaving its client counterpart out of date. The GemStone object is now dirty.

An object is no longer dirty after it updates its counterpart in the other object space.

The former situation is more common if most application behavior occurs in the client image; the latter is more common if most application behavior occurs in GemStone, or if other concurrent sessions commit object changes of their own.

The direction in which update occurs is critical to the correctness of an application; we therefore use two different terms to distinguish the direction of update:

- *Faulting* refers to copying modified GemStone objects from the repository into the client Smalltalk image. Faulting either updates an existing replicate, or, if necessary, creates a new one.
- *Flushing* refers to copying modified client objects from the Smalltalk image into the GemStone repository.

Together, GemBuilder and GemStone manage the timing of faulting and flushing.

Faulting

GemStone manages faulting automatically. When shared objects change in the repository, the GemStone object manager marks them as dirty. In the most common scenario, when your session starts, commits, aborts, or continues a transaction—at a *transaction boundary*—GemStone faults in dirty objects. Committing or aborting thus refreshes your session's private view of the repository.

In addition to refreshing a session's view at transaction boundaries, faulting also occurs automatically when:

- Connectors connect: this typically occurs at login, the beginning of a GemStone session, but you can connect and disconnect connectors explicitly during the course of a session using either code or the Connector Browser.
- A stub receives a message.
- GemStone Smalltalk executes, and the repository modifies the state of a shared object that GemBuilder has already retrieved into the image.

  GemStone Smalltalk execution occurs when:

  - a forwarder receives a message, or
  - in response to any variants of:

    ```
    GbsSession >> execute:
    ```

    ```
    GbsObject >> remotePerform:
    ```

Flushing

GemBuilder flushes objects into GemStone at transaction boundaries, immediately before any GemStone Smalltalk execution, or before faulting a stub.

Flushing is not the same as committing. When GemBuilder flushes an object, the change becomes visible to the session's private view of the GemStone repository, but it doesn't become part of the shared repository until your session commits—only then are your changes accessible to other users.

GemBuilder automatically detects modifications to connected client objects. You can disable this feature, however, if you wish to mark objects dirty explicitly in your code.

To disable automatic dirty-marking for performance or debugging, execute:

```
GBSM autoMarkDirty: false
```

Minimizing Replication Cost

Replicating the full state of a large and complex collection can demand too much memory or network bandwidth. Optimize your application by controlling the degree and timing of replication; GemBuilder provides three ways to help:

Instance Variable Mapping — Modify the default class map to specify how *widely* through each object to replicate—which instance variables to connect and which to prune as never being of interest to an application. You can also specify the details of an association between two classes whose structures do not match.

Stubbing — Specify how *deeply* through the network to replicate, how many layers of references to follow when faulting occurs.

Replication Specifications — Another way to specify how *widely or deeply* through each object to replicate—of a class's mapped instance variables, which to replicate and which to stub.

Instance Variable Mapping

As discussed in "Class Mapping" on page 3-6, before GemBuilder can replicate objects, it must know their respective structures and the mapping between them. By default GemBuilder maps instance variables by name. You can override this default either by suppressing certain instance variables, thereby rendering them invisible to an application, or by explicitly specifying a mapping between nonmatching names.

Suppressing Instance Variables

Some client Smalltalk objects, however, must define instance variables that are relevant only in the client environment—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 in which the GemStone class defines instance variables that a given application will never need; many applications can share repository objects without necessarily sharing the same concerns. Mapping allows your application to prune parts of an object.

Suppress the replication of an individual instance variable simply by omitting its name from its counterpart's class definition:

- If a client object contains a named instance variable that does not exist in its GemStone counterpart, the value of that variable is not replicated in GemStone. When the rest of the object is stored in the repository, its value is omitted; when GemBuilder faults the GemStone object into the client, the client's suppressed instance variable remains unchanged.
- Likewise, if a GemStone object contains a named instance variable that does not exist in its client counterpart, the value of that variable is not replicated in the client. When the application replicates the GemStone object in the client, its value is not transferred; when the application flushes the object into the repository, GemStone's suppressed instance variable remains unchanged.

You can also suppress instance variable mappings by implementing the client class method `instVarMap`. Example 3.2 shows a simple implementation:

Example 3.2

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

The first component of the return value, a call to `super instVarMap`, ensures that all instance variable mappings established in superclasses remain in effect.

Appended to the inherited instance variable map, an array contains the pairs of instance variable names to map. The first pair (`nil gsName`) specifies that the GemStone instance variable `gsName` will never be replicated in the client. The second pair (`stName nil`) specifies that the client instance variable `stName` will never be replicated in GemStone.

Nonmatching Names

You can also specify an explicit instance variable mapping between GemStone and the client:

- to map two instance variables whose names don't match, or
- to prevent the mapping of two instance variables whose names *do* match.

In this way your application can accommodate differing schemas.

To specify nonstandard instance variable mappings, use the same class method `instVarMap`, as in Example 3.3:

Example 3.3

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

Appended to the inherited instance variable map, a single pair declares that the instance variable `stName` in the client maps to the instance variable `gsName` in GemStone.

One implementation can both prune irrelevancy and accommodate differing schemas, as the instance variable mapping for the class Book shows in Example 3.4:

Example 3.4

```
Book class>>instVarMap
      ^super instVarMap ,
            #(    (title title)
                  (author author)
                  (nil pages)
                  (publisher nil)
                  (copyright publicationDate) )
```

The first two pairs of instance variables change nothing: they explicitly state what would happen without this method, but are included for completeness.

`(nil pages)` specifies that the client application does not need to know a books page count and therefore this repository-side instance variable is not replicated in the client.

`(publisher nil)` specifies that the client application needs (and presumably assigns) the instance variable `publisher`, which is never stored in the repository.

`(copyright publicationDate)` maps the client class Book's instance variable `copyright` to the GemStone class Book's instance variable `publicationDate`.

Stubbing

Often, however, an application has need of certain instance variables, but not all at once. For example, it's impractical to replicate the entire hierarchy of BigDictionary at login: users will experience unacceptable network delays, and the client Smalltalk image can't handle data sets as large as GemStone's. Furthermore, it's unnecessary: only a small number of objects will be needed for the current task. To help prevent this kind of over-replication, GemBuilder provides stubs.

A *stub,* like a forwarder, is also a proxy associated with a delegate (an instance of GbsObject that knows which server object it's associated with). Unlike a forwarder, however, when a stub receives a message, it does not send the message across to the other object space. Instead, it fetches its delegate, which responds by faulting the GemStone counterpart into the client image. The client Smalltalk replicate then responds to the message.

GemBuilder faults automatically:

- when connectors connect,
- when a stub receives a message, and
- sometime after another session commits a change to a shared object.

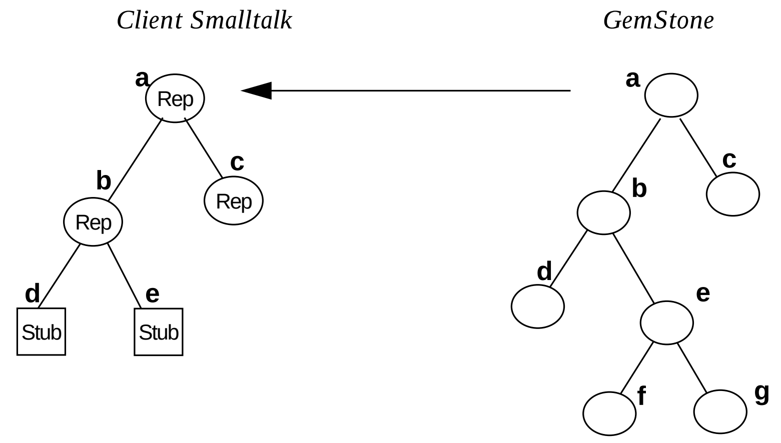
A stub can respond to any of these events. Each time, GemBuilder replicates the object hierarchy to a certain level, then creates stubs for objects one level deeper. The number of levels that are replicated each time is the *fault level.*

A fault level of 1 follows an object's immediate references and faults those in. A fault level of 2 follows one more layer of references and replicates those objects, too. Figure 3.3 illustrates an application with a fault level of 2.

Faulting at Login

At login, the connectors connect, and objects **a**, **b**, and **c** are replicated; objects **d** and **e** are stubbed; objects **f** and **g** are ignored.

Figure 3.3 Two-level Fault of an Object



Faulting in Response to a Message

When object **e**, a stub, receives a message, it faults in a replicate of its counterpart GemStone object.

A stub faults in a replicate in response to a message. Therefore, direct references to instance variables can cause problems. Direct access is not a message-send; the stub will not fault in its replicate, because it receives no message; neither can it supply the requested value. To avoid this problem, use accessor methods to get or set instance variables.

The following sequence demonstrates the problem. The object starts as a replicate in client Smalltalk:

```
myVar := ’abc’.
```

Next, it’s stubbed. There are several ways it could become a stub; for this example, we’ll assume that it became a stub by locking (locking an object stubs it).

```
self mySession acquireWriteLockFor: self.
```

The object, now a stub, has no knowledge of the replicate’s instance variables. Therefore, executing the code below causes an error:

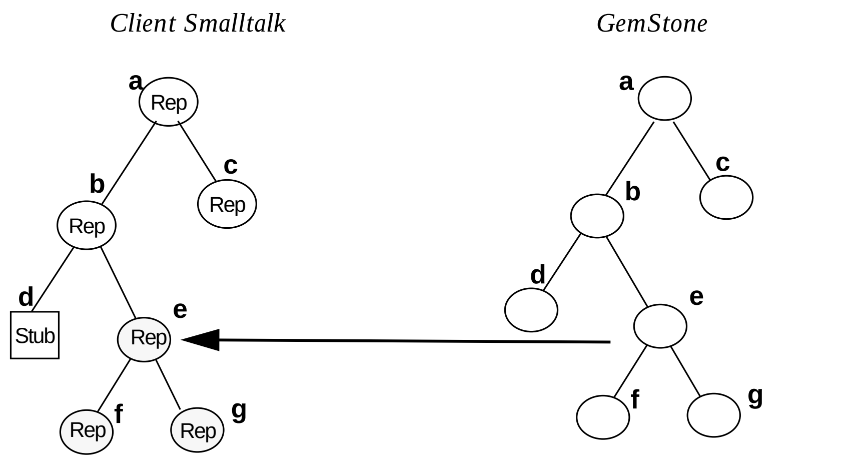
```
myVar := ’bcd’
```

Using an accessor method, on the other hand, causes the stub to be faulted in and yields the correct result:

```
self myVar: 'bcd'
```

e is now a replicate, as shown in Figure 3.4. The new replicate responds to the message.

Figure 3.4 A Stub Responds to a Message



Again, two levels are replicated, object **e** and its immediate instance variable: a fault level is a global parameter.

Linked sessions can often tolerate higher fault levels than remote sessions because they are less sensitive to bandwidth limitations; you can set their defaults with the configuration parameters `faultLevelLnk` and `faultLevelRpc`.

Now, suppose another session commits a change to **b**?

Faulting in Changes From Other Sessions

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—that is, you can see your own changes to the repository—or the Gem can change your view at transaction boundaries or after a session has executed GemStone Smalltalk.

A Gem maintains a list of repository objects that have changed and notifies GemBuilder of any changes to objects it has replicated. If it finds any changed counterparts, it updates the client object with the new GemStone value.

Replicates and stubs respond to the message `faultPolicy`. The default implementation returns the value of GemBuilder's configuration parameter `defaultFaultPolicy`: either `#lazy` or `#immediate`.

- A *lazy* fault policy means that, when GemBuilder detects a change in a repository object, it turns the client counterpart from a replicate into a stub. The object will remain a stub until it next receives a message.
- An *immediate* fault policy means that, when GemBuilder detects a change in a repository object, it updates the replicate immediately.

If another session commits a change to **b**, and **b**'s fault policy is lazy, **b** becomes a stub. If **b**'s fault policy is immediate, **b** is updated.

The default fault policy is lazy, to minimize network traffic. For more information, see the description of `defaultFaultPolicy` in the Settings Browser. For examples, browse implementors of `faultPolicy` in the GemBuilder image.

Overriding Defaults

Because linked sessions are less sensitive to bandwidth limitations, GemBuilder ships with `faultLevelLnk` set to 2 and `faultLevelRpc` set to 4. In this way, linked sessions replicate less at login, faulting in objects as they are needed.

- You can override these defaults for specific instance variables of specific replicates.
- You can also stub or replicate certain objects explicitly.

To specify fault levels for all instance variables, implement a class method `replicationSpec` for the client class. Replication specifications are versatile mechanisms described starting on page 3-21.

To cause a replicate to become a stub, send it 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.

Sometimes stubbing is not desirable, either for performance reasons or for correctness. For example, primitives cannot accept stubs as parameters if the primitive accesses the value of the parameter. If your application uses an object as an argument to a primitive, you must either prevent that object from ever becoming a stub, or ensure that it's replicated before the primitive is executed.

To cause a stub to become a replicate, send it the message `fault`. Stubs respond to this message by replicating; replicates return `self`. The message `faultToLevel:` allows you to fault in several levels at once, as specified.

To prevent a replicate from ever being a stub, configure it as a replicate at login and set its `faultPolicy` to `#immediate`.

Defunct Stubs

Faulting in a stub relies on the existence of a valid GemStone object to replicate or forward to. If an object is stubbed, then the session logs out, a message to that stub raises an error complaining that it is *defunct*. For example, suppose MyGlobal is modified in GemStone, thereby stubbing it in your client session. If the session logs out before MyGlobal is faulted back in, the client Smalltalk dictionary contains a defunct stub.

Because GemBuilder cannot safely assume that a given object will retain the same object identifier from one session to the next, it cannot simply fix the problem at next login. That's the job of a connector: to reestablish at login the stub's relationship to GemStone. A connector can do so either directly, by connecting the stub itself, or transitively, by connecting some object that refers to the stub.

If you've defined a connector for MyGlobal, logging back into GemStone reconnects it.

Now, suppose an instance variable of MyGlobal becomes a stub shortly before a session logs out. Sending a message to this variable will produce a defunct stub error. At next login, MyGlobal's connector will fault in the variable. You can then retry the message, but only by means of a message sent to MyGlobal (or another connected object). If the application is maintaining a direct reference to the previous defunct stub, the error will persist.

NOTE
You cannot proceed from a defunct stub error. After you've encountered this error, determined the cause, and corrected the problem, you must restart the Smalltalk operation that encountered the defunct stub.

Replication Specifications

By default, when GemBuilder replicates an instance of a connected class, it replicates all that class's instance variables as well to the session's specified fault level. You can further refine faulting by class, however, with specific instructions for individual instance variables.

Each class replicates according to a replication specification (hereafter referred to as a *replication spec).* The replication spec allows you to fault in specified instance variables as forwarders, stubs, or replicates that will in turn replicate their instance variables to a specified level.

By default, a class inherits its replication spec from its superclass. If you haven't changed any of the replication specs in an inheritance chain, then the inherited behavior is to replicate all instance variables as specified by the session's configuration parameters `faultLevelLnk` and `faultLevelRpc`.

To modify a class's replication behavior in precise ways, implement the class method `replicationSpec`. For example, suppose you want class Employee's address instance variable always to fault in as a forwarder:

Example 3.5

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

To ensure that replication specs established in superclasses remain in effect, Example 3.5 appends its implementation to the result of:

```
super replicationSpec
```

Appended to the inherited replication spec are nested arrays, each of which pairs an instance variable with an expression specifying its treatment at faulting:

(*instVar*   *whenFaulted*)

instVar can be either:

- the client-side name of an instance variable, or
- the reserved identifier `indexable_part`, specifying an object's unnamed indexable instance variables, such as the elements of a collection.

whenFaulted is one of:

`stub` —	faults in the instance variable as a stub.
`forwarder` —	faults in the instance variable as a forwarder to the server.
`min` *n* —	faults in the instance variable and its referents as replicates to a minimum of *n* levels. `min 0` = replicate.
`max` *m* —	faults in the instance variable and its referents as replicates to a maximum of *m* levels. `max 0` = stub.

`replicate` — faults in the instance variable as a replicate whose behavior will be subject to the configuration parameters `faultlevelRpc` and `faultLevelLnk`, relative to the root object being faulted.

By default, an instance variable's behavior is `replicate`; your application needn't specify replicates unless to restore behavior overridden in a superclass.

Example 3.6

```
TestObject class>>replicationSpec
^super replicationSpec ,
      #(    (instVar1 stub)
            (instVar2 forwarder)
            (instVar3 max 0)
            (instVar4 min 0)
            (instVar5 max 2)
            (instVar6 min 2)
            (instVar7 replicate)
            (indexble_part min 1) )
```

NOTE

To ensure that your replication spec is respected, do not rely on automatic class generation to replicate instances of classes for which you have defined replication specs. Instead, make sure these classes are connected before your application tries to access the corresponding server objects. Automatic class generation ignores replication specifications.

Replication Specifications and Class Versions

As explained in "Mapping and Class Versions" on page 3-8, client Smalltalk classes connect not simply to GemStone Smalltalk classes, but to specific GemStone class versions. A class connector connects to at most one GemStone version.

A replication spec, therefore, affects only client instances connected to instances of the correct GemStone class version.

Suppose, for example, that you define and redefine class X in GemStone until its class history lists three versions. Your client Smalltalk class is connected to Version 2. Class X's replication spec will affect GemStone instances of Class X, Version 2. If the repository contains instances of Class X, Versions 1 or 3, the replication spec will not affect them.

Multiple Replication Specifications

It's not always possible to define one replication spec that works well for all operations in an application. Some queries or windows may require a different object profile than others in the same application and session; a replication spec crafted to optimize one set of operations can make others inefficient.

By default, the message `replicationSpec` returns the default replication spec. Change this by sending the message `replicationSpecSet:` *#someRepSpecSelector* to an instance of GbsSession. With this message, you can specify multiple replication specs, selecting one dynamically according to circumstances. The following procedure shows how:

Step 1. Decide on a new name, such as `replicationSpec2`.

Step 2. Implement `Object class >> replicationSpec2` to return `self replicationSpec`.

Step 3. Reimplement `replicationSpec2` as appropriate in those application classes that need it.

Step 4. Immediately before your application performs the query or screen fetch or other operation that requires the second replication spec, send `replicationSpecSet:` *#replicationSpec2* to the current GbsSession instance.

For example, suppose your application has a class Employee, with instance variables `firstName`, `lastName`, and `address`. `address` contains an instance of class Address. The application has one screen that displays the names from a list of employees, and another screen that displays the zip codes from a list of employee addresses. Here's how to replicate only what's needed:

Step 1. Define a new replication spec with the selector `empNamesRepSpec`.

Step 2. Implement `Object class >> empNamesRepSpec` as:

```
^self replicationSpec.
```

Step 3. Implement `Employee class >> empNamesRepSpec` as:

```
^#((firstName min 1) (lastName min 1) (address stub))
```

Step 4. Define another replication spec with the selector `empZipcodeRepSpec`.

Step 5. Implement `Object class >> empZipcodeRepSpec` as:

```
^self replicationSpec
```

Step 6. Define `Employee class >> empZipcodeRepSpec` as:

```
^#((firstName stub) (lastName stub) (address min 2))
```

and `Address class >> empZipcodeRepSpec` as:

```
^#((city stub) (state stub) (zip min 1))
```

Step 7. Before opening the employee names screen, send:

```
myGbsSession replicationSpecSet: #empNamesRepSpec
```

Restore it to `#replicationSpec` after opening the window.

Step 8. Before opening the zip code window, send:

```
myGbsSession replicationSpecSet: #empZipcodeRepSpec
```

Restore it to `#replicationSpec` after opening the window.

For each window, the procedure above reduces the number of objects retrieved to the minimum required. Other objects fault in as stubs; if subsequent input requires them, they are retrieved transparently.

Managing Interobject Dependencies

Replication specs are ordinarily an optimization mechanism. Some applications, however, require a replication spec to function correctly. If the structural initialization of an object depends on other objects, you must implement replication specs to ensure that, when GemStone traverses an object, it also traverses those objects it depends on.

For example, in order to create a Dictionary when replicating it from GemStone, we need to be able to send `hash` to each key to determine its location in the hash table (hash values aren't necessarily the same in GemStone as they are in the client Smalltalk image). So, if GemStone traverses a Dictionary, it must also traverse the association, and the key in the association. The default implementation for `Dictionary class >> replicationSpec` therefore contains `#(indexable_part min 1)`, and `Association class >> replicationSpec` contains `#(key min 1)`.

This works for Dictionaries with simple keys such as strings, symbols or integers. If an application has dictionaries with complex keys, though, additional replication specs can be required. For example, if you are storing Employees as keys in a dictionary, and you've implemented `=` and `hash` in Employee to consider the `firstName` and `lastName`, then you must ensure that when a dictionary containing Employees is traversed, so are the associations, the employees, and the `firstName` and `lastName`.

You could ensure this by implementing `Employee class >> replicationSpec` to include `#(firstName min 1)` and `#(lastName min 1)`. Or, if you had a special Dictionary class for Employees, you could include `#(indexable_part min 3)` in that dictionary class's replication spec. However, this could cause the entire Employee to be traversed whenever one of these dictionaries is traversed, rather than just the `firstName` and `lastName`.

We recommend that you use the default replication spec `#replicationSpec` as the base replication spec for all classes to reflect interobject dependencies. When defining other replication specs, make sure the default implementation in Object is:

```
^self replicationSpec
```

Ensure that subclass implementations of the new `replicationSpec` method do not stray from the default sonas to break interobject dependencies.

Precedence of Multiple Replication Specs

It's possible to implement replication specs that appear to contradict each other. Such apparent conflicts are resolved deterministically according to the order in which instance variables appear in a replication spec and the order in which objects are traversed. If a superclass specifies one way of handling an instance variable, and a subclass reimplements `replicationSpec` to handle the same variable in a different way, the last occurrence takes precedence.

For example, suppose the value returned from sending `replicationSpec` to the subclass is:

```
#((name min 1) (name max 2))
```

The last occurrence of the instance variable is `max 2`, and therefore takes precedence.

If subclass implementations of `replicationSpec` always append their results to `super replicationSpec`, the subclass will reliably override the superclass handling of a given instance variable. The recommended approach is:

```
^super replicationSpec, #((name max 2))
```

not:

```
^#((name max 2)), super replicationSpec.
```

Another apparent contradiction can arise between parent and child objects. For example, suppose Employee refers to an Address, which refers to a complex object County. The Employee `replicationSpec` includes `#(address min 5)`, specifying that several levels of the County object are to be replicated. But if Address includes `#(county max 1)`, it modifies Employee's handling of address.

Employee specifies, "Get at least 5 levels of address." Address specifies, "Whatever you do, don't get more than one level of county." The apparent contradiction is resolved by the order in which these specifications are enountered: because Address is encountered after Employee, Address takes precedence.

If your object network includes cycles, different replication specs could take effect at different times, depending on which object is the replication root at any given time. Given a specific root object, however, it's always possible to determine the exact effect of a set of replication specs.

Customized Flushing and Faulting

You can customize both flushing and faulting to change object structure arbitrarily, if your application requires it. You can even create a class in GemStone that maps to a client Smalltalk class with a different format—for example, a format of bytes on the client but pointers in the repository.

Modifying Instance Variables During Faulting

Customize object retrieval with buffers for the client counterparts of GemStone objects as they are faulted in. You can then process the contents of these buffers in any manner required.

To provide these buffers, reimplement the class methods:

```
namedValuesBuffer
indexableValuesBuffer
```

To unpack these buffers correctly, reimplement the class methods:

```
namedValues:
indexableValues:
namedValues:indexableValues:
```

By default, `namedValuesBuffer` returns `self`: new client objects are faulted directly into the named instance variable slots. Override this to supply either a different object of the same type, or an instance of GbsBuffer (a subclass of Array) of the required size.

By default, `indexableValuesBuffer` returns `self`. Override this to return an indexable buffer of the appropriate size.

The buffers you define in these methods are used during faulting. They are subsequently unpacked by the faulted object according to its implementation of the unpacking methods listed above.

Implement the unpacking methods to obtain the desired client representation by performing arbitrary computation on the buffer contents. Use the message `namedValues:indexableValues:` for cases in which computation must operate on indexable and named values together.

NOTE
The methods `namedValuesBuffer` *and* `namedValues:` *are a pair; so are* `indexableValuesBuffer` *and* `indexableValues:`*. To avoid replication errors, if you override one, you must also override the other.*

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 simply adds the element to the Set rather than assigning it to a specific slot.

NOTE
To avoid generating a “previous flush did not complete” error, if you override `namedValues:` *or* `indexableValues:`*, make sure you do not send messages to any stubs that would require a remote object to be faulted. Doing so causes an error as faulting is attempted while flushing. Adjust the* `replicationSpec` *and* `faultPolicy` *of the object to ensure that stubs won’t exist for special flush operations.*

You can override two other messages to control faulting initialization and postprocessing: `preFault` and `postFault`.

Implement `preFault` to initialize a newly created object prior to faulting its named and indexable values.

For example:

```
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 so that additions to the collection would be sorted properly during faulting.

Implement `postFault` to do any necessary postprocessing. For example, if the methods used to add to an OrderedCollection also marked the object dirty, the postprocessing could remove dirty-marking: by definition, faulting never results in a dirty object (assuming that GemStone's is the valid state):

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

Modifying Instance Variables During Flushing

To provide an arbitrary mapping of objects from the client to GemStone you can implement two class methods called `namedValues` and `indexableValues`.

`namedValues`
: Implement this 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 client object. The store operations then access this buffer for storing in GemStone.

`indexableValues`
: Implement this to return a list of the indexable instance variables in the client object. The store operations then access this list for storing in GemStone.

Implementations of `namedValues` must return an object with the appropriate number of named instance variable slots. In Example 3.7, a clone of the positionable stream is returned that increments the `position` instance variable by 1 as needed when mapped into GemStone:

Example 3.7

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

An alternative might return an instance of GbsBuffer (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, eliminating the need for a temporary array.

Implementations of `indexableValues` must return an indexable collection containing a sequential list of the elements in the collection. In Example 3.8, for class Set, an Array is returned, because the indexable fields of a Smalltalk set are a sparse list of the actual elements.

Example 3.8

```
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, eliminating the need for 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 for 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, you might need to reimplement the messages `indexableSize` and `namedSize` as well. For example, to match the implementation of `indexableValueAt:` above, OrderedCollection would have to implement `indexableSize` as shown below; otherwise, the object storage mechanisms would try to iterate over the entire list of indexable slots rather than those controlled by `firstIndex` and `lastIndex`:

```
indexableSize
      ^self size
```

Mapping Classes With Different Formats

You can create a class in GemStone that maps to a client Smalltalk class with a different format—for example, a format of bytes on the client but pointers in the repository. To do so, reimplement the class method `gsObjImpl` in the client Smalltalk to return a value specifying the GemStone implementation.

A `gsObjImpl` method must return a SmallInteger representing the GemStone class format. The following formats are valid:

Return	**Format**
0	pointers
1	bytes
2	nonsequenceable collection

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

Limits on Replication

Replicating blocks and scaled decimals can present special problems, discussed below.

Replicating Client Smalltalk Blocks

Forwarders are especially well-suited for managing large collections that reside in the object server. Collections are commonly sent messages that have blocks as arguments. When the collection is represented in client Smalltalk by a forwarder, these argument blocks are replicated in GemStone and executed in the server.

When a GemStone replicate for a client Smalltalk block is needed, GemBuilder sends the block to GemStone Smalltalk for recompilation and execution. If a block is used more than once, GemBuilder saves a reference to the replicated block to avoid redundant compilations.

For example, consider the use of `select:` to retrieve elements from a collection of Employees:

```
| fredEmps |
fredEmps := myEmployees select:
            [ :anEmployee | (anEmployee name) = 'Fred' ].
```

If `myEmployees` is a forwarder to a collection residing in the object server, then GemBuilder sends the parameter block's source code:

```
[ :anEmployee | (anEmployee name) = 'Fred' ].
```

to GemStone to be compiled and executed.

Replication of client Smalltalk blocks to GemStone Smalltalk is subject to certain limitations. When block replication violates one of these limitations, GemBuilder issues an error indicating that the attempted block replication has failed.

To avoid these limitations, consider using block callbacks instead. Block callbacks are discussed starting on page 3-35.

You can disable block replication completely using GemBuilder's configuration parameter **blockReplicationEnabled**. Block replication is enabled by default. Set this parameter to `false` to disable it, and GemBuilder raises an exception when block replication is attempted. This can be useful for determining if your application depends on block replication.

Image-stripping Limitations

Block replication relies on the client Smalltalk compiler and decompiler; if they've been removed from a deployed runtime environment, blocks cannot be replicated.

In a deployed image from which the compiler and decompiler have been removed, do not use block replication. Usually this requires implementing a cover method for the block in a GemStone method, and sending that message instead. For instance:

```
aForwarder select: [ :name | name = #Fred ]
```

—is instead coded:

```
aForwarder selectNameEquals: #Fred
```

...and in GemStone, `selectNameEquals:` is implemented as:

```
selectNameEquals: aName
  ^self select: [ :name | name = aName ]
```

When the block is encoded entirely in GemStone in this way, you can further optimize its operation by taking advantage of indexes and use an optimized selection block, as described in the *GemStone Programming Guide*.

Temporary Variable Reference Restrictions

A block is replicated in the form of its source code, without its surrounding context. Therefore, values drawn from outside the block's own scope cannot be relied upon to exist in both the client Smalltalk and in GemStone. Replication is not supported for blocks that reference instance variables, class variables, method arguments, or temporary variables declared external to the block's scope.

An exception is allowed in the case of global references, such as class names:

- Global variable references from inside a block must have the same name in both object spaces.

In the case of global variables containing data, it is the programmer's responsibility to ensure that the global identifier represents compatible values in both contexts.

Temporary variable reference restrictions disallow the following, because "tempName" is declared outside the block's scope:

```
| namedEmps tempName |
tempName := 'Fred'.
namedEmps := myEmployees select:
            [ :anEmployee | (anEmployee name) = tempName ].
```

As a workaround, implement a new Employees method in GemStone Smalltalk named `select:with:` that evaluates a two-argument block, in which the extra block argument is passed in as the `with:` parameter. For example:

```
select: aBlock with: extraArg
|result|

result := self speciesForSelect new.
self keysAndValuesDo: [ :aKey :aValue |
  (aBlock value: aValue value: extraArg) "two-value block"
    ifTrue: [result at: aKey put: aValue]
  ].

^ result.
```

You can then rewrite the application code to pass its temporary as the argument to the `with:` parameter without violating the scope of the block:

```
| namedEmps tempName |
tempName := 'Fred'.
namedEmps := myEmployees select:
            [ :anEmployee :extraArg |
                  (anEmployee name) = extraArg
            ] with: tempName.
```

Restriction on References to self or super

References to `self` and `super` are also context-sensitive and, therefore, disallowed:

- A replicated block cannot contain references to `self` or `super`.

For example, the following code cannot be forwarded to GemStone because the parameter block contains a reference to `self`:

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

References to `self` or `super` in forwarded code must occur outside the scope of the replicated block, where you can be sure of the context within which they occur. For example, you can rewrite the above code to return a result code, which can then be evaluated in the calling context, outside the scope of the replicated block:

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

Explicit Return Restriction

Because a block is replicated without its surrounding context, a return statement has no surrounding context to which to return. Therefore:

- A replicated block cannot contain an explicit return.

For example:

```
result := myDict at:#key ifAbsent:[ ^nil ]
```

is disallowed. The statement can be recoded to perform its return within the calling context:

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

Replicating GemStone Blocks in Client Smalltalk

Also supported, though less commonly used, is the replication of GemStone blocks in client Smalltalk. Similar restrictions apply with regard to external references and the need for compiler / decompiler support. Blocks most frequently passed from the server to the client are the sort blocks that accompany instances of SortedCollection and its subclasses. Sort blocks rarely have occasion to violate replicated block restrictions.

If restrictions hamper you, consider using block callbacks instead.

Block Callbacks

Block callbacks provide an alternate mechanism for representing a client block in GemStone that avoids the limitations of block replication by calling back into the client Smalltalk to evaluate the block.

Block callbacks have the following advantages over block replication:

- Block callbacks don't require a compiler or decompiler.
- Block callbacks don't suffer the context limitations of block replication. The block can reference self, super, instance variables, and non-local temporaries; it can also perform explicit returns. For example, the following expression works correctly as a block callback, but fails if you try to replicate the block:

  aForwarder `at: aKey ifAbsent: [ ^nil ] asBlockCallback`

Block callbacks have the following disadvantages:

- A block that is evaluated many times in GemStone will perform poorly as a block callback. For example, the following expression sends a message to the client forwarder for each element of the collection represented by *aForwarder*:

  aForwarder `select: [ :e | e isNil ] asBlockCallback`

You can determine whether, by default, blocks are replicated or call back to the client using GemBuilder's configuration parameter **blockReplicationPolicy**. Legal values `#replicate` and `#callback`. A value of `#replicate` causes a client block to be stored in GemStone as a GemStone block. A value of `#callback` causes a client block to be stored in GemStone as a client forwarder, so that sending `value` to the block in GemStone causes `value` to be forwarded to the client block; the result of that block evaluation is then passed back to the GemStone context that invoked the block.

To ensure a specific replication policy for a given block, use the methods `asBlockCallback` or `asBlockReplicate`. Send `asBlockCallback` to ensures= that the block always executes in the client, regardless of the default block replication policy set by the configuration parameter. Likewise, send `asBlockReplicate` to ensure that the block is executed local to the context that invokes it (either in GemStone or in the client).For example:

```
dictionaryForwarder
   at: #X
   ifAbsent: [ ^nil ] asBlockCallback

collectionForwarder do: [ :e | e check ] asBlockReplicate
```

Replicating ScaledDecimals

The ScaledDecimal representation in VisualAge is different from the one used in GemStone Smalltalk; therefore, arithmetic operations can return different results in VisualAge than in GemStone Smalltalk.

In GemStone Smalltalk, a ScaledDecimal is represented as a numerator and a denominator (to preserve complete accuracy of any rational number), along with a scale value used to determine the number of decimal digits to print.

In VisualAge, a ScaleDecimal is represented as a 31-digit number, with a field width and a scale indicating how many of the digits are to the right of the decimal point. When creating a VisualAge ScaledDecimal from a GemStone instance, any further digits in the fractional component are truncated.

The following examples demonstrate the difference that can result. In GemStone, Example 3.9 returns `1.00`, because `x` is represented as `1/3`, which preserves full accuracy:

Example 3.9

```
| x |
x := ScaledDecimal numerator: 1 denominator: 3 scale: 2.
^ x + x + x
```

In VisualAge, however, Example 3.10 returns `0.99d3.2`, because `x` is represented as `0.33`, thereby truncating the result:

Example 3.10

```
| x |
x := ScaledDecimal gbsNumerator: 1 denominator: 3 scale: 2.
^ x + x + x
```

Client Copies

It’s fast and simple to make a client copy of a GemStone object that maintains no reference to the repository. Because they have no knowledge of the GemStone object they were made from, such copies are not real replicates.

These copies are deep copies: they replicate a complete transitive closure of the GemStone object. Nothing is stubbed.

NOTE
Be careful not to replicate a GemStone object large enough to overflow the client image.

To make an unassociated copy of a GemStone object in the client object space, send:

```
aGbsObject asLocalObjectCopy
```

Because it is unrelated to the GemStone original, values are neither flushed nor faulted, nor is state synchronized; it is safe to assume the copy will be out of date, if not soon, then eventually.

Such copies are suitable for read-once applications that are pressed for resources.

To make a similar unassociated copy of a client object in GemStone, send:

```
aClientObject asGSObjectCopy
```

While replicates are almost always easier to use, it may sometimes be faster and simpler to copy the data, manipulate it, and then replicate it back in GemStone. This might be true, for example, if the ratio of execution to data set size is large.

To do this reliably:

- There must be just one root for the GemStone data, because the identity of internal objects will be lost with this technique.
- The data set must be small enough to fit in the client Smalltalk memory.

NOTE
Each server copy created with `asGSObjectCopy` *gets a new object identifier, even if the client object you're copying already has a server counterpart with its own object identifier. Therefore, copying client objects in this way can double your use of object identifiers.*

3.5 Precedence of Replication Mechanisms

Certain replication mechanisms can appear to contradict each other. The rules of precedence are:

- If the class methods `instVarMap` (for replicates) or `instancesAreForwarders` (for forwarders) are implemented, they take precedence over all others and are always respected.
- Otherwise, if the class method `replicationSpec` is implemented, or if an application calls or `replicationSpecSet:` to switch among several replication specs, those replication specs take precedence.

  In other words, if a class implements a replication spec, but it also implements `instancesAreForwarders` to return `true`, then instances of that class will be forwarders and the replication spec will be ignored.

  Or, if a class implements both `instVarMap` and `replicationSpec`, the `instVarMap` determines which instance variables will be visible to the replication spec.
- In the absence of a replication spec, the instance method `faultToLevel:`, if called, is respected for replicates. Forwarders, of course, do not fault.

- For classes that use no other mechanism, the configuration parameters `faultLevelLnk` and `faultLevelRpc` are respected.

3.6 Converting Between Forms

A variety of messages exist to convert between delegates, forwarders, replicates, stubs, and unconnected client objects. Table 3.1–Table 3.5 list the results of sending any of several conversion messages to these objects.

NOTE
To avoid unpredictable consequences and possible errors, do not use the expressions listed as producing undefined results.

Table 3.1 Delegate Conversion Protocol

Message	**Return Value**
`copy`	shallow copy of delegate
`asLocalObject`	replicate
`asLocalObjectCopy`	deep copy of replicate
`asGSObject`	`self`
`asForwarder`	undefined
`beReplicate`	undefined
`fault`	undefined
`stubYourself`	undefined

Table 3.2 Forwarder (to the Server) Conversion Protocol

Message	**Return Value**
`copy`	copies associated server object and returns replicate of copy
`asLocalObject`	undefined
`asLocalObjectCopy`	undefined
`asGSObject`	associated delegate
`asForwarder`	`self`
`beReplicate`	`self`, which has become a replicate
`fault`	`self` (use `beReplicate` to make a replicate)

Table 3.2 Forwarder (to the Server) Conversion Protocol

Message	Return Value
`stubYourself`	`self`

Table 3.3 Replicate Conversion Protocol

Message	Return Value
`copy`	shallow copy of delegate not associated with any server object
`asLocalObject`	undefined
`asLocalObjectCopy`	undefined
`asGSObject`	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self`
`fault`	`self`, whose instance variables are now also replicates to the configured fault level
`stubYourself`	`self`, which has become a stub

Table 3.4 Stub Conversion Protocol

Message	Return Value
`copy`	shallow copy; receiver becomes a replicate
`asLocalObject`	undefined
`asLocalObjectCopy`	undefined
`asGSObject`	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self` (use `fault` to become a replicate)
`fault`	`self`
`stubYourself`	`self`

Table 3.5 Conversion Protocol for Unshared Client Objects

Message	Return Value
`copy`	shallow copy
`asLocalObject`	undefined
`asLocalObjectCopy`	undefined
`asGSObject`	new delegate; creates new associated server object
`asForwarder`	`self`, which has become a forwarder; creates new associated server object
`beReplicate`	`self`
`fault`	`self`
`stubYourself`	`self`

Chapter 4

Connectors

GemBuilder associates a client and GemStone object with a *connector*.

- Connectors function at login. After that, you must manually disconnect and reconnect them to effect any functional changes.
- Connectors exist either in a given session or globally—in every session your image defines.
- Connectors can update either connected object.
- Different kinds of connectors use different lookup mechanisms.
- To avoid certain kinds of errors, you can verify connections at login.
- You create connectors with the Connector Browser or in code.

Connecting and Disconnecting
describes what happens at login and logout, which login and logout, and how to initialize instances.

Kinds of Connectors
describes how lookup occurs.

Making and Managing Connectors
explains how to make and manage connectors in code or using the Connector Browser.

4.1 Connecting and Disconnecting

At login, connectors connect objects according to their specifications; thereafter, they are inactive. Changes to instances that occur during the course of a session are replicated either because those instances are synchronized replicates that mark changes dirty, or because one is a forwarder to the other. Changes to class definitions or other unsynchronized changes must be propagated manually. To do so, use the **Disconnect** and **Connect** buttons in the Connector Browser to disconnect and reconnect the appropriate connector.

At logout, to reduce the risk of defunct stub or forwarder errors, GemBuilder sets connections to `nil` if, at login, they update the client or create a forwarder.

Scope

Some connectors connect their objects whenever any session logs in; some do so only when a specific session logs in:

- *Global connectors* allow you to maintain a standard set of connectors common to all applications in your GemBuilder image.
- *Session connectors* allow individual applications to customize connectors: you define unique session parameters for each application, and different sessions can connect different objects. When sessions of one kind log in, other sessions' connectors are defined but not connected.

When a session logs in, its session connectors and all global connectors (if not already connected) connect automatically.

When a session logs out, its session connectors disconnect. If the session is the last in the application to log out, it disconnects the global connectors.

Verifying Connections

Connectors are saved in client Smalltalk sets, separate ones for global connectors and each named kind of session (each uniquely defined set of session parameters). Two connectors are considered equal if they resolve to the same client object. Client Smalltalk sets eliminate duplicates based on equality. Therefore:

NOTE
Adding a global or session connector that points to the same object as an existing connector will remove the existing connector.

Duplicate session connectors are not removed if they are stored in different sessions.

GemBuilder provides a configuration parameter, **`connectVerification`**, that, when `true`, causes connectors to verify at login that they are not redefining a connector that already exists. In addition, class connectors verify that the two classes they are connecting have compatible structures.

If a connector fails verification, GemBuilder issues a notifier if **verbose** is also `true`, or raises an exception otherwise. You can set **`connectVerification`** in the Connector Browser or in the Settings Browser.

NOTE
Under certain circumstances, GemBuilder may verify that a connector still exists in a session after it has been explicitly removed from that session. This is because the order in which connectors are connected at login is not deterministic.For more information and a work-around, see "Connection Order" on page 4-5.

Initializing

At login, a connector associates an object in a single-user image with an object in a multiuser repository. The value of either could have changed since last login. Which value is valid?

Connectors can initialize either object by performing a specified *postconnect action:*

Update Smalltalk
default for all but class connectors, initializes the client object with the current state of the GemStone object.

Update GemStone
initializes the GemStone object with the current state of the client object.

Forward to the server or client
makes one object a forwarder to the other. Forwarders are discussed starting on page 3-9.

No initialization
leaves the client object and GemStone object unmodified after connection—default for class connectors.

As the name implies, postconnect actions execute only at initial connection. After that, changes propagate according to mark dirty specifications, as described in "Synchronizing State" on page 3-12, or they do not propagate at all, as is normally the case with class connectors, as described in "Class Mapping" on page 3-6.

Updating Class Definitions

By default, after login and initialization, class connectors do not propagate changes. If you've defined classes differently on the client and the server, you probably had good reason to do so; you probably don't want one object space to update the other with its own class definition. Therefore, to avoid updating class definitions, class connectors generally specify a postconnect action of *none*.

For similar reasons, class connectors seldom specify that the client class is a forwarder—in fact, the forwarder postconnect action is disabled for GemBuilder classes, GemStone kernel classes, and other critical classes. (To determine the full list, display the result of executing `GbsForwarder nonForwarding`.)

If you change either a client or GemStone class definition during a session, you must propagate the change yourself by disconnecting and reconnecting the connector. The Connector Browser, described starting on page 4-13, provides convenient buttons for the purpose.

NOTE
Remember to restore a postconnect action of none *after you complete the desired update.*

4.2 Kinds of Connectors

Five kinds of connectors use different ways of finding the two objects to connect. You have already encountered one kind:

Class connector — connects a client Smalltalk and GemStone class. As discussed in “Class Mapping” on page 3-6, to replicate an object, both client and repository must define the class, and the two classes must be connected using a class connector.

For replicating instances, however, we need ways to connect root objects:

Name connector — connects client and GemStone objects identified by name. Figure 4.1 illustrates how a name connector connects a client object to a GemStone object.

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

Class instance variable connector — first 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.

Fast connector — connects the GemStone kernel classes to their client Smalltalk counterparts. Fast connectors are predefined, implemented for speed. The kernel classes to which they point will not change identity during the course of a session; GemBuilder can take advantage of that to reduce overhead.

NOTE
Application objects can change identity during the course of a session. Applications should therefore not define fast connectors.

The GemStone kernel class connectors are predefined, and GemBuilder relies on them. You cannot convert them to forwarders.

Connection Order

At login, GemBuilder connects connectors in the following order:

1. First, predefined fast connectors for kernel classes;
2. next, class connectors whose postconnect action is anything other than **updateGS**; and finally
3. all other connectors, in no particular order.

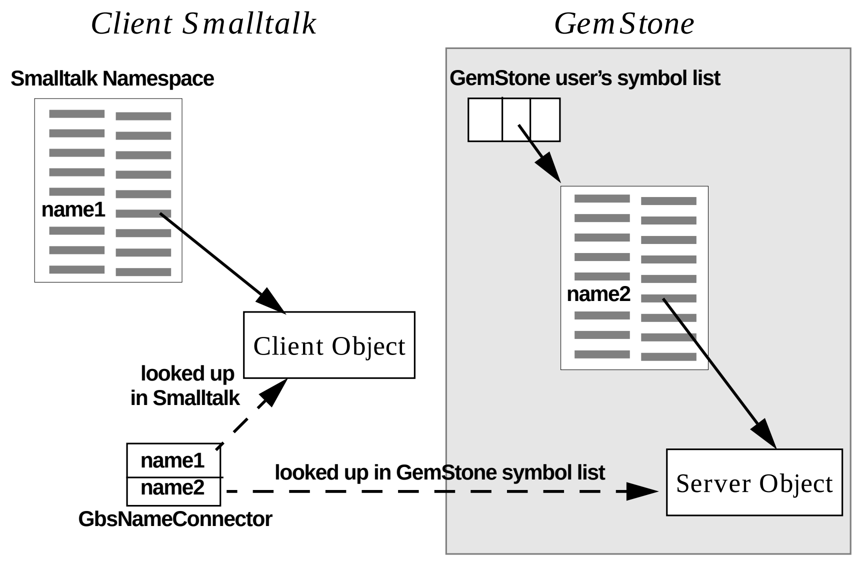
You can control the order in which connectors connect by connecting them explicitly in your code, instead of relying on GemBuilder's automatic mechanism to connect them for you at login.

Lookup

Names must be found in namespaces. In the client, objects included in a Smalltalk namespace can be visible throughout the Smalltalk image. GemStone implements namespaces with symbol dictionaries: if the symbol list of the session user includes the symbol dictionary defining object A, then object A is visible to that user.

Lookup occurs when the connection is established—that is, when the session first logs in.

Figure 4.1 Connecting a Name Connector



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.

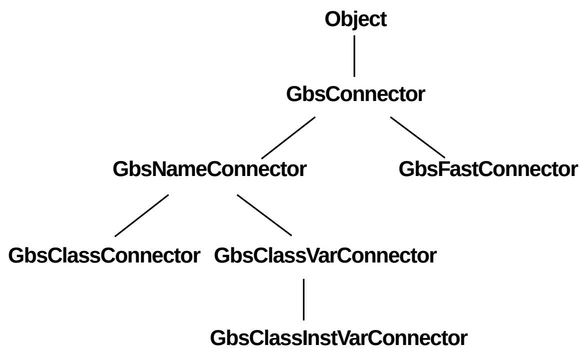
4.3 Making and Managing Connectors

To make and manage connectors interactively, see “The Connector Browser” on page 4-13. The next section describes making and managing connectors in code.

Making Connectors Programmatically

GbsConnector is the abstract superclass for the connector class hierarchy. These classes implement connection methods and define instance variables to refer to the associated GemStone and client objects. Figure 4.2 shows the hierarchy.

Figure 4.2 Connector Class Hierarchy



To create a connector programmatically:

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

Create the required GemStone session parameters and connectors in an initialization method.(Creation methods for session parameters are described in “Defining Session Parameters” on page 2-4.)

Creating Connectors

One simple creation method for a name connector requires only the names of the two objects to be connected:

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

You can create a class connector this way too:

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

The above methods require that the GemStone object already exist. If GemBuilder must create the object, choose an instance creation method that specifies the GemStone dictionary in which to place it:

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

To create a class variable connectors:

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

Similarly, a class instance variable connector:

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

For more, browse instance creation methods for each connector class.

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 image.

Adding Connectors to a Connector List

When you create a connector, you must decide whether it is to be managed by an individual session or globally. Leaving it unmanaged can have several adverse effects: it will not be connected and disconnected when required, and object retrieval may slow.

A connector is managed by adding it to the appropriate list of connectors.

If you want a connector in effect whenever any session logs in, put it in the global connectors collection:

`GBSM addGlobalConnector:` *aConnector*

A new global connector first takes effect the next time any session logs in.

Each session parameters object maintains its own list of session connectors. If you want a connector in effect whenever a session logs in using specific parameters, add a connector to the session parameters object:

ThisApplicationParameters `addConnector:` *aConnector*

A new session connector first takes effect the next time that session logs in.

To initialize a system with two roots, the global `BigDictionary`, and a class variable in `MyClass` called `MyClassVar`, your application might execute code such as that shown in Example 4.1:

Example 4.1

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

Initialization code such as that in Example 4.1 needs to execute only once. From then on, every time you log into GemStone, `MyGlobal` and `MyClassVar` (and all the objects they reference) connect; after that, replication and updating occur as specified.

Session Control

The following examples illustrate one approach to managing GemBuilder sessions and connectors: a session control class that defines these methods for, in this example, a help request system.

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 to change as needed. In either case, methods to create the session parameters object and its connectors might follow these patterns:

Example 4.2 shows the method `session`, which 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 login fails, the method returns `nil`.

Example 4.2

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

Example 4.3 shows a method that initializes a set of session parameters. (For security, you may choose to prompt for passwords instead.)

Example 4.3

```
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
```

Example 4.4 adds connectors to the session parameters object by calling lower-level methods to individual types of connectors:

Example 4.4

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

Example 4.5 shows a method that creates class connectors and adds them to the session parameters connector list:

Example 4.5

```
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 4.6 shows a method that creates class variable connectors and adds them to the session parameters connector list:

Example 4.6

```
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 4.5 and 4.6 to create name connectors and global connectors for your application, as well.

NOTE

If more than one session is logged into GemStone using the same session parameters object, and you add a connector to one of those sessions, *GemBuilder will try to connect that connector for all sessions sharing the same parameters. If any fail to reference the GemStone object represented by the connector, you'll receive an error message stating that the connector failed to connect.*

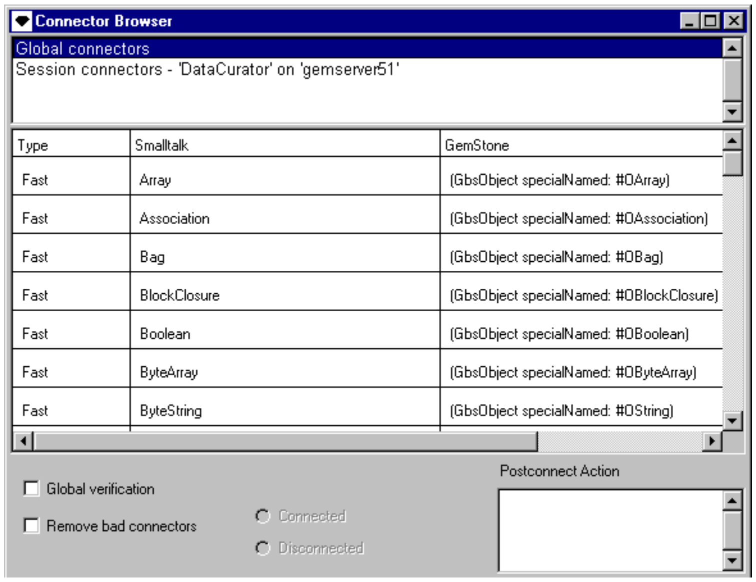
The Connector Browser

You can use GemBuilder's Connector Browser to make and manage connectors interactively. To open a Connector Browser, select **Connectors** from the GemStone menu. With this browser, you can:

- 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 4.3 shows the Connector Browser.

Figure 4.3 The Connector Browser



Connector Browser

Global connectors
Session connectors - 'DataCurator' on 'gemserver51'

Type	Smalltalk	GemStone
Fast	Array	(GbsObject specialNamed: #OArray)
Fast	Association	(GbsObject specialNamed: #OAssociation)
Fast	Bag	(GbsObject specialNamed: #OBag)
Fast	BlockClosure	(GbsObject specialNamed: #OBlockClosure)
Fast	Boolean	(GbsObject specialNamed: #OBoolean)
Fast	ByteArray	(GbsObject specialNamed: #OByteArray)
Fast	ByteString	(GbsObject specialNamed: #OString)

☐ Global verification
☐ Remove bad connectors
○ Connected
○ Disconnected
Postconnect Action

The Group Pane

The top pane is the Group pane; it allows you to select either global connectors or those associated with an individual session. Global connectors are predefined to connect the GemStone kernel classes with their client Smalltalk counterparts. When you select an item in this pane, the connectors defined for the selected item appear in the middle pane.

In the Group pane, the popup menu provides the following items:

Table 4.1 Group List Menu in the Connector Browser

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

The Connector Pane

The middle pane is the Connector pane; it lists the connectors, their types, and descriptions in both the client and GemStone Smalltalks. In the Connector pane, the popup menu offers the following items:

Table 4.2 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, prompting for required information.
remove...	Removes a connector, after confirmation.
change type...	Changes a connector to a different type, prompting you for the type.

The Control Panel

The bottom pane is a control panel that allows you to change the **connectVerification** and **removeInvalidConnectors** configuration parameters and connect or disconnect objects. setting a connector's postconnect action is described in the section that follows.

Table 4.3 Options in the Control Panel

Global verification	When enabled, connectors (other than class connectors) verify that they are not redefining an object connection before connecting. Class connectors, upon connection, verify that the structures of the two connected classes 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.

Enabling connector verification can slow login: we recommend that you turn on verification during development and turn it off for production systems.

Postconnect Action

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

Table 4.4 Postconnect Action Options in the Connector Browser

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

To create a new connector:

1. Place the cursor in the Connector pane.
2. Select **add** from the menu.
3. When prompted, specify the type of connector.
4. When prompted, specify the names of the client 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.

To change the postconnect action:

1. Disconnect the objects by clicking on the **Disconnected** button.
2. Change the postconnect action as required.
3. Reconnect the objects by clicking on the **Connected** button.

If your application initially stores its data in the client , 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. Select **updateGS** as the postconnect action for these connectors.
3. Log into GemStone so that GemBuilder can create the GemStone replicates for the client Smalltalk data.
4. Inspect the GemStone objects to be sure they have the intendedvalues.
5. Commit the transaction and log out.
6. Select the connectors and change their postconnect actions to **updateST** so that future sessions will begin by using the stored GemStone data.

Chapter 5 Using the GemStone Programming Tools

After you install GemBuilder, many menus in your Smalltalk image contain additional commands for executing GemStone Smalltalk code and accessing GemBuilder programming tools. These tools are in many ways similar to those of the client Smalltalk, but with important differences; this chapter describes those differences.

GemStone Menu
introduces the tools and options available from the GemStone menu.

Browsing Code
describes the GemStone Classes Browser and other code browsers.

Other GemStone Tools
describes GemStone workspaces, including the System Workspace, and various GemStone inspectors.

Coding
explains how to use the GemBuilder tools to create classes and methods in GemStone Smalltalk for execution and storage on the server.

Debugging
describes setting breakpoints, using the Breakpoint Browser, and using GemBuilder's enhanced debugger.

5.1 GemStone Menu

The **GemStone** menu accessible from various tools, gives you access to the GemStone Smalltalk compiler and the GemBuilder programming tools. Many of these functions are also available from pop-up menus in the browsers and tools.

As shown in Table 5.1, the **GemStone** menu provides commands for executing GemStone Smalltalk code and accessing the GemStone programming tools.

Table 5.1 The GemStone Menu

Sessions	Opens a GemStone Session Browser, allowing you to log into or out of GemStone and manage transactions. The Session Browser is described in Chapter 2.	
Connectors	Opens a GemStone Connector Browser, allowing you to manage the connections between GemStone and Smalltalk objects. The Connector Browser is described in Chapter 4.	
Browse	Produces a submenu with the following options:	
	All Classes	Opens a GemStone Browser, comparable to the client Smalltalk System or Classes Browser. The GemStone Browser is described in “Browsing Code” on page 5-4.
	Namespace...	Prompts for the name of a symbol dictionary, then opens a browser focused on that dictionary.
	Class...	Prompts for the name of a class, then opens a browser focused on that class.
	Senders of...	Prompts for the name of a message selector, then opens a method browser showing senders of that message.
	Implementors of...	Prompts for the name of a message selector, then opens a method browser showing implementors of that message.
	References to...	Prompts for the name of a variable, then opens a method browser showing all methods that refer to that variable.
	Methods with substring...	Prompts for a string, then opens a method browser showing all methods whose source contains that string.
Admin	Produces a submenu with the following options:	

Table 5.1 The GemStone Menu (Continued)

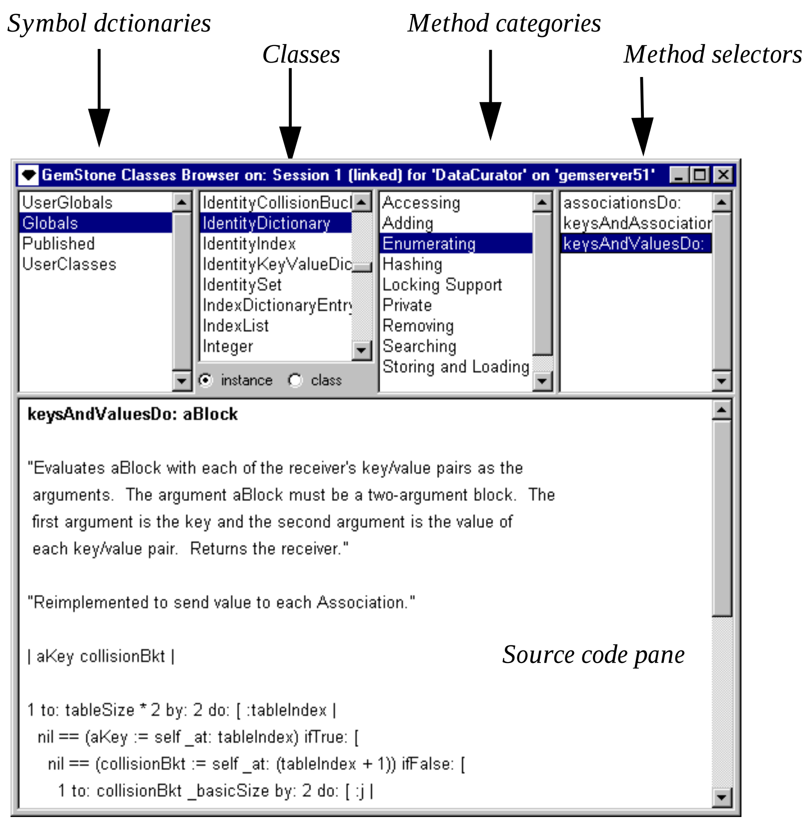
	Users	Opens a GemStone User Account Manager, allowing you to create new users, assign attributes to them, and manage user accounts, provided you have the privileges to do so. The User Account Manager is described in Chapter 7.
	Namespaces	Opens a Symbol List Browser, allowing you to examine and modify symbol dictionaries and their entries. The Symbol List Browser is described in Chapter 7.
	Segments	Opens a Segment Tool, allowing you to control authorization at the object level by assigning objects to segments. The Segment Tool is described in Chapter 7.
Tools	Produces a submenu with the following options:	
	New GS Workspace	Opens a GemStone Workspace.
	Open GS Workspace...	Prompts for a file name, then opens the selected saved workspace file in a GemStone workspace.
	GS File in	Files the selected GemStone Smalltalk code into GemStone.
	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.
	Breakpoints	Opens a Breakpoint Browser, allowing you to set and clear breakpoints in GemStone Smalltalk code. The Breakpoint Browser is described on page 5-26.
	System Workspace	Opens the GemStone System Workspace, a workspace containing a variety of useful GemStone Smalltalk and client Smalltalk expressions.
About GemBuilder	Opens a window providing the GemBuilder version and copyright information.	

5.2 Browsing Code

After logging in to GemStone, open a GemStone Classes Browser by choosing **GemStone > Browse > All Classes**.

The GemStone Classes Browser allows you access source and other information about each of the kernel classes and methods; you can also create GemStone Smalltalk classes and methods in the GemStone repository.

Figure 5.1 GemStone Classes Browser



```
keysAndValuesDo: aBlock

"Evaluates aBlock with each of the receiver's key/value pairs as the
 arguments.  The argument aBlock must be a two-argument block.  The
 first argument is the key and the second argument is the value of
 each key/value pair.  Returns the receiver."

"Reimplemented to send value to each Association."

| aKey collisionBkt |

1 to: tableSize * 2 by: 2 do: [ :tableIndex |
  nil == (aKey := self _at: tableIndex) ifTrue: [
    nil == (collisionBkt := self _at: (tableIndex + 1)) ifFalse: [
      1 to: collisionBkt _basicSize by: 2 do: [ :j |
```

The GemStone Classes Browser is similar to the client Smalltalk System or Classes Browser, but a few differences exist: for example, the upper left pane contains a list of symbol dictionaries, GemStone's mechanism for implementing namespaces. This facilitates finding and sharing objects efficiently. The symbol dictionaries that you can access are listed in the GemStone Browser's symbol list pane.

When you select a symbol dictionary in the Symbol List pane, all classes defined in that dictionary appear in the Classes pane to the right. (Symbols other than classes can be viewed by opening an inspector on the symbol dictionary in question.)

GemStone Smalltalk categorizes methods by function to make them easier to browse. When you select a class in the Classes pane, a list of its method categories appears in the Method Categories pane to the right.

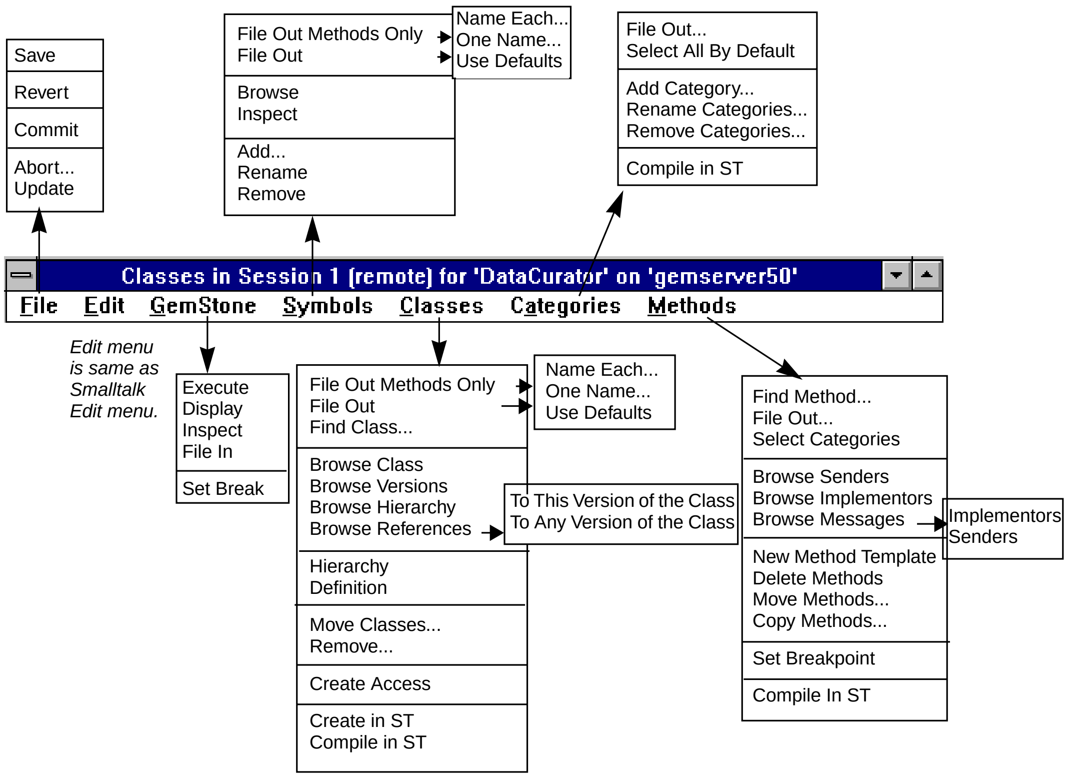
When you select a method category, all the message selectors in that category appear in the rightmost Method Selectors pane.

As in the comparable client Smalltalk browsers, you can switch focus between **instance** or **class** methods using the toggle provided.

Also as in the comparable client Smalltalk browsers, when you select a method, its source code is displayed in the lower portion of the browser—the source pane. In this pane, you can edit and recompile the method, set breakpoints in it, or execute fragments of GemStone Smalltalk code as in a workspace.

Each pane of the GemStone Browser also has pop-up menus accessible with the *operate* mouse button. The pop-up menus in the Symbol List, Class, Categories, and Methods panes are identical to the **Symbols, Classes, Categories,** and **Methods** menus available from the menu bar, except that the file-out options are not present in the pop-up menus. Figure 5.2 shows all the menus available from the GemStone Browser's menu bar.

Figure 5.2 Menus in the GemStone Browser



The following sections describe gemStone-specific commands in the Symbols, Classes, Category, and Methods menus.

The File Menu

The following GemStone-specific commands are available from the File menu.

Table 5.2 File Menu in the GemStone Browser

Commit	Attempts to save to the GemStone repository all modifications that occurred during the current GemStone transaction.
Abort...	Undoes all changes that you have made in the GemStone

The GemStone Menu

The following commands are available from the **GemStone** menu.

Table 5.3 GemStone menu in the GemStone Browser

Execute	Compiles and executes the selected GemStone Smalltalk code.
Display	Compiles and executes the selected GemStone Smalltalk code, and displays a textual representation of the result.
Inspect	Compiles and executes the selected GemStone Smalltalk code, then opens a GemStone inspector on the result.
File In	Files the selected code into GemStone.
Set Break	Sets a message breakpoint on the selected method, causing the virtual machine to halt when that selector is sent to an instance of the current class or a subclass. You can then open a GemStone debugger to examine the current execution context.

CAUTION
Do not remove the Globals dictionary; it defines the GemStone kernel classes.

The Classes Menu

Table 5.4 describes the GemStone-specific commands available from the **Classes** menu. A later section, "Defining a New Class" on page 5-13, explains how to define a new GemStone Smalltalk class to add to the currently selected symbol dictionary.

CAUTION
To avoid inadvertently removing or modifying a GemStone kernel class, use the DataCurator account for all system administration functions except those that require SystemUser privileges, such as upgrading or restoring the GemStone repository.

Table 5.4 GemStone Browser's Classes Menu

File Out Methods Only	Allows you to file out only the methods of a given class, so that you can file them into a client class without changing the client class's structure.
Browse Versions	Spawns a Class Version Browser that shows how many versions of the selected class exist, and allows you to access each.
Create Access...	Creates basic methods for accessing and updating the instance variables of the selected class. A dialog allows you to specify which variables to include.
Create In ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, if one doesn't already exist. If it does exist, executing this menu item has no effect.
Compile In ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, and compiles all currently defined methods for the class. If necessary, a notifier lists any methods that cannot be compiled in client Smalltalk.

Pop-up Text Pane Menu

A pop-up menu appears in any text pane when you press the *operate* mouse button. This menu provides the same commands as the corresponding menu in the client Smalltalk browser's text pane. In addition, it contains menus for displaying, executing, inspecting, and filing in GemStone Smalltalk code and for using breakpoints in GemStone Smalltalk code.

The GemStone-specific commands available from a text area pane are shown in Table 5.5.

Table 5.5 Pop-up Menu in GemStone Browser's Text Pane

GS-execute	Executes the code in GemStone.
GS-display	Executes the code in GemStone and displays the result in the text area.
GS-inspect	Executes the code in GemStone and opens an inspector on the result.
GS-file in	Files the selected text into GemStone.
Set Break	Sets a breakpoint at the step point nearest the cursor location. If the cursor is not exactly at a step point, scans the method from the current cursor location on and sets a breakpoint at the next step point. See "Debugging" on page 5-22 for a full discussion of using breakpoints.

GemBuilder also adds the following items to the appropriate menus in the client Smalltalk browser:

Table 5.6 Additional GemStone Menu Items

create in GS	Creates a GemStone Smalltalk class having the same name and structure as the selected client Smalltalk class, if one doesn't already exist. If it does exist but you've changed its structure, executing this menu item creates a new version of the class.
compile in GS	Creates a GemStone Smalltalk class having the same name and structure as the selected client Smalltalk class, and compiles all currently defined methods for the class in GemStone. If necessary, a notifier lists any methods that cannot be compiled.

5.3 Other GemStone Tools

GemStone workspaces, the GemStone System Workspace, and GemStone inspectors are described briefly in the following sections.

GemStone Workspaces

To open a GemStone workspace, choose **GemStone > Tools > New GS Workspace**. In a GemStone workspace, you can execute GemStone Smalltalk as well as client Smalltalk.

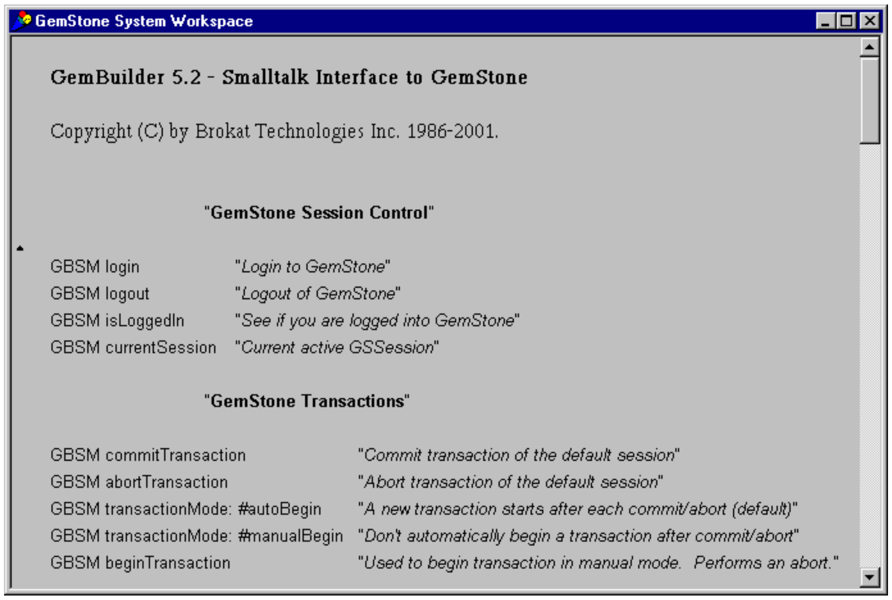
The GemStone workspace menu offers the same GemStone-specific commands listed in Table 5.5.

The System Workspace

The GemStone System Workspace is a workspace containing templates for many useful GemStone Smalltalk and client Smalltalk expressions. Browse it to familiarize yourself with its contents.

To open a GemStone System Workspace (Figure 5.3), choose **GemStone > Tools > System Workspace** from the **GemStone** menu.

Figure 5.3 GemStone System Workspace



GemBuilder 5.2 - Smalltalk Interface to GemStone

Copyright (C) by Brokat Technologies Inc. 1986-2001.

"GemStone Session Control"

```
GBSM login              "Login to GemStone"
GBSM logout             "Logout of GemStone"
GBSM isLoggedIn         "See if you are logged into GemStone"
GBSM currentSession     "Current active GSSession"
```

"GemStone Transactions"

```
GBSM commitTransaction               "Commit transaction of the default session"
GBSM abortTransaction                "Abort transaction of the default session"
GBSM transactionMode: #autoBegin     "A new transaction starts after each commit/abort (default)"
GBSM transactionMode: #manualBegin   "Don't automatically begin a transaction after commit/abort"
GBSM beginTransaction                "Used to begin transaction in manual mode.  Performs an abort."
```

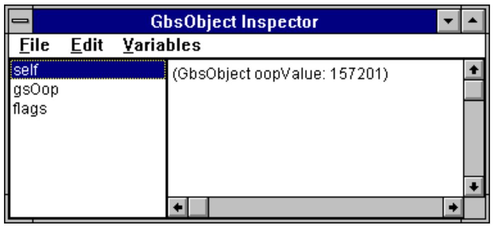
Inspectors

GemStone Inspectors, like client Smalltalk inspectors, let you examine the values of a variety of GemStone objects, and modify them when appropriate. When you select a GemStone Smalltalk expression and execute **GS-inspect**, a GemStone inspector opens instead of a client Smalltalk inspector. The GemStone inspector (Figure 5.4) is similar to a client Smalltalk inspector; it has comparable panes and functionality. The inspector contain the following GemStone-specific commands:

Table 5.7 Commands in GemStone Inspector

Basic Inspect	Opens an inspector on the delegate object, an instance of GbsObject (see Figure 5.4).

Figure 5.4 GemStone Delegate Inspector



(GbsObject oopValue: 157201)

Inspecting Nonsequenceable Collections

When you're inspecting an instance of any nonsequenceable collection, the following additional menu items are available.

Table 5.8 Commands for Inspecting NSCs

Add	Prompts you for the name of the object to add to the nonsequenceable collection. To a Dictionary, an Association with the given key and a value of `nil` is added.
Remove	If you've selected an index variable, removes the corresponding element from the collection. If you've selected a key, removes the corresponding association from the dictionary.

5.4 Coding

This section explains how to define new GemStone classes and methods, and describes aspects of coding unique to GemStone Smalltalk.

About GemStone Smalltalk Classes

The following discussion summarizes the main differences between GemStone Smalltalk and client Smalltalks. For complete information about programming in GemStone Smalltalk, refer to the *GemStone Programming Guide*.

Instance variables can be constrained. To speed GemStone Smalltalk's indexed associative access for efficient querying, you can constrain the value of an instance variable to contain only specified kinds of objects. Constraining a variable means that its value is always either an instance of the specified class, a subclass thereof, or `nil`.

Constraints can be circular: you can constrain an instance variable to be an instance of its own class, or you can also constrain instance variables of two classes to each hold instances of the other.

Constraints are inherited: when you define a subclass, its inherited instance variables by default bear the same constraints as those specified in their superclass. However, inherited instance variables can be further constrained in a subclass. In this case, the instance variable's new constraint must be a subclass of that specified by the inherited constraint.

To further constrain inherited instance variables, specify the name of the inherited variable and its new constraint in the argument to the `constraints`: keyword in the class definition template. For example, suppose you have defined a class Employee with instance variables named `jobTitle` and `department` that are constrained to be Strings. You can now create a subclass of Employee named FormerEmployee and constrain the inherited variables `jobTitle` and `department` to be InvariantStrings. FormerEmployee's new instance variables can be constrained or not, as you require, and its other inherited instance variables retain whatever constraints were set in the superclass that defined them, if any.

Instances can be invariant. A class definition can specify that all instances are invariant, meaning they can be modified only during the transaction in which it is created. After the transaction is committed, you can no longer modify its instance variables, nor the size or class of the object.

Specify invariance for a class by providing the argument *true* to the `instancesInvariant:` keyword in the class definition template.

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. Any object can be made invariant by sending it the message `immediateInvariant`. This mechanism protects objects from being modified and can be useful for maintaining the integrity of your repository. After `immediateInvariant` is sent to an object, you can no longer modify its instance variables, nor the size or class of the object. The effect of the `immediateInvariant` message is not reversible.

The message `isInvariant` returns true if the receiver is invariant; false otherwise.

Defining a New Class

To define a new GemStone class:

Step 1. Open a GemStone Browser if one is not already open.

Step 2. In the Symbol List pane, select the dictionary in which you wish to refer to the new class. Make sure no class is selected in the class list.

The browser displays the class definition 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.

Step 3. Replace *NameOfSuperclass* with the name of your new class’s immediate superclass.

Step 4. Replace *NameOfClass* with the name of the new class. By convention, the first letter of each GemStone class name is capitalized.

Step 5. Replace *instVarName* with the names of any instance variables, or delete all the text within the parentheses if your new class has no instance variables.

A class can define up to 255 named instance variables.

Step 6. Replace *classVarNames* with the names of any class variables, or delete all the text within the parentheses if your new class has no class variables.

Step 7. Replace *classInstVarNames* with the names of any class instance variables, or delete all the text within the parentheses if your new class has no class instance variables.

Step 8. Fill in the brackets after the `poolDictionaries:` keyword with any pool dictionaries that you want the class to access. Pool dictionaries are special-purpose storage structures that enable any arbitrary group of classes and their instances to share information. When classes share a pool dictionary, methods defined in those classes can refer to the variables defined in the pool dictionary.

Step 9. After the `inDictionary:` keyword, the name of the selected symbol dictionary is inserted in the template. This is the symbol dictionary that will allow you to refer to your class by name. Unless you replace the inserted text with the name of another symbol dictionary to which you have access, your new class is defined in the selected symbol dictionary.

Step 10. Fill in the brackets after the `constraints:` keyword with any constraints you wish to specify for one or more of the instance variables.

To constrain the elements of a collection, type the name of the constraint class inside the brackets. For example, to constrain the Bag subclass BagOfEmployees to contain only instances of the class Employee, type:

```
constraints: #[Employee]
```

To constrain a named instance variable, type the name of the variable and the constraint class as a pair separated by a comma, each within its own set of square brackets, also separated by commas. Preface the instance variable name with a #. For example, to constrain the instance variables `name` and `address` of the class Employee to be a String and an instance of the class Address, respectively, type:

```
constraints: #[ #[ #name, String], #[ #address, Address] ]
```

Step 11. After the `instancesInvariant:`keyword, specify whether instances of the class are modifiable. The default is `false`—change this to `true` if you wish instances to be invariant.

Step 12. After the `isModifiable:` keyword, specify whether the structure of the class can be modified. The default value is `false`—change this to `true` if you wish class to be invariant.

Step 13. Save your changes and commit your transaction to make the class part of the repository.

NOTE
You cannot subclass certain GemStone kernel classes. To determine which ones, execute the method `subclassesDisallowed` *against the class Object. The method returns* `true` *for any class that you cannot subclass.*

For example, consider the following definition of a class named Employee:

Example 5.1

```
Object subclass: 'Employee'
      instVarNames: #( 'name' 'employeeNum' 'jobTitle' 'department'
                       'address')
      classVars: #()
      classInstVars #()
      poolDictionaries: #[]
      inDictionary: UserGlobals
```

Employee resides in the developer's UserGlobals dictionary.

```
      constraints: #[ #[#name, String],
                      #[#employeeNum, SmallInteger],
                      #[#jobTitle, String],
                      #[#department, Symbol],
                      #[#address, Address] ]
```

For efficient access, constraints have been placed on each instance variable: `name` must be an instance of String, `employeeNum` must be an instance of SmallInteger, `jobTitle` must be an instance of String, `department` must be an instance of Symbol, and `address` must be an instance of Address.

```
      instancesInvariant: false
```

When instances of the class are created, their values will be modifiable even after they've been committed to the repository.

`isModifiable: true`

This class is modifiable; instance variables can still be added, removed, and constrained, and class or class instance variables can be added. However, as long as the class itself remains modifiable, no instances of it can be created.

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, 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 discussion of class storage formats in the chapter entitled “Advanced Class Protocol” 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 the class definition template or 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.

If you do not include the `newVersionOf:` keyword, the compiler checks to see if another class having the same name already exists. If it does, the new class is compiled as a new version of the other class and shares its class history. If it does not, the new class is created with a new class history.

For more discussion of class versions and histories, see the chapter entitled “Class Versions and Instance Migration” in the *GemStone Programming Guide.*

Private Instance Variables

Some GemStone kernel classes have private instance variables. For example, the GemStone Bag class defines four, used by the object manager and primitives to implement features of nonsequenceable collections, such as adding indexing structures for efficient querying. Private instance variable names begin with an underscore (_). When defining subclasses, private instance variables cannot be modified or constrained.

Modifying an Existing Class

If you select an existing GemStone Smalltalk class, then modify and save the class definition, you create a new version of that 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 migrate its instances; they're still instances of the old class. You can migrate some or all instances of one version of a class to another version explicitly.

For more information on migrating instances, see the chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide.*

NOTE
You can modify only classes for which you have write authorization

To create a new version of a class:

Step 1. Select the class in the browser to bring up its definition in the source pane.

Step 2. Edit the definition as required.

Step 3. Select **Save** from the pop-up menu.

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 have the old class's structure and access the old class's methods, but they can be migrated to the new class.

Let’s assume that you have a class named Employee with instance variables for `name`, `employeeNum`, `jobTitle`, `department`, and `address`, and that the class is defined as shown in Figure 5.1. Two of Employee’s instance variables are constrained to be instances of class String, and one (`employeeNum`) is constrained to be a SmallInteger. Suppose that you decide that the class needs an additional instance variable named `salary` to represent the Employee’s salary.

To do this, you can define a new version of the class Employee 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.

After you compile the class definition, the new class is named Employee, 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 8-2.)

Defining Methods

You can modify only 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.

Public and Private Methods

GemStone has both public and private methods. *Public* GemStone methods are supported. *Private* GemStone methods are those implemented to support the public protocol—they are not supported and are subject to change.

Private GemStone methods are those whose seelctor is prefixed with an underscore (_). They appear in the browsers along with the public methods, and you can display the source for them.

CAUTION
Private methods are subject to change at any time. Do not depend on the presence or specific implementation of any private method when creating your own classes and methods.

Reserved and Optimized Selectors

The following selectors are reserved for the sole use of the GemStone Smalltalk kernel classes. Those selectors are:

```
ifTrue:              untilFalse        timesRepeat:
ifFalse:             untilTrue         isNil
ifTrue:ifFalse:      whileFalse:       notNil
ifFalse:ifTrue:      whileTrue:        ==
_or:                 to:do:            ~~
_and:                to:by:do:         _class
isKindOf:            _isInteger        _isSmallInteger
_isSymbol            includesIdentical
```

Redefining a reserved selector has no effect; the same primitive method is called and your redefinition is ignored.

In addition, the following methods are optimized in the class SmallInteger:

```
+        -      *      >=     =
```

You can redefine the optimized methods above in your application classes, but redefinitions in the class SmallInteger are ignored.

Saving Class and Method Definitions in Files

It’s often useful to store the GemStone Smalltalk source code in text 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.

To save GemStone code in a file, use any of the GemStone browser’s **File Out** menu items. To read and compile a saved file, use any of the **Gs-File in** or **GS-File it in** menu items.

Saved GemStone files are written as sequences of Topaz commands. Example 5.2 shows a class definition in Topaz format:

Example 5.2

```
!
! From GEMSTONE: 5.0, Tue Jun  4 18:36:18 US/Pacific 1996;
!
!
! Class 'Address'
!
run
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
%
```

GemStone's filing out and filing 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 custom 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, refer to the *Topaz Programming Environment*.

The file-in mechanism cannot execute the full set of Topaz commands; it's limited to the following subset:

```
category:         method
classmethod       method:
classmethod:      printit
commit            removeAllMethods
doit              removeAllClassMethods
```

The GemStone file-in mechanism acknowledges the presence of the following commands by adding notes to the System Transcript, but it does not execute them:

```
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, either. Instead, it 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, GemStone displays a compilation error notifier that contains the erroneous module in a text editor. If you correct the error and then choose **Save**, GemStone recompiles the module and then processes the rest of the file.

In the case of authorization problems, commands that the file-in mechanism doesn't recognize, or other errors, GemStone displays a simple error notifier without an editor and stops processing the file.

5.5 Debugging

In addition to the basic code development tools described previously in this chapter, GemStone also provides GemStone Smalltalk debugging facilities similar to the debugging aids supplied by the client Smalltalk. These facilities enable you to perform the following operations:

- You can step through execution of a method, examining the values of arguments, temporaries, and instance variables after each step.
- You can set, clear, and examine GemStone Smalltalk breakpoints. When a breakpoint is encountered during normal execution, a notifier appears and you can open a debugger with which you can interactively explore the contexts in the stack at the time execution halted.

- You can inspect or change the values of arguments, temporaries, and receivers in any context (stack frame) on the virtual machine call stack, then continue execution from the top of the stack. This means that you can find out what the system was doing at the time a breakpoint, or an error interrupted execution.
- You can execute a message expression within the scope of a given context.

Breakpoints

For the purpose of determining exactly where a step will go during debugging, a GemStone Smalltalk method is composed of step points. You can set breakpoints at any step point.

Generally, step points correspond to the message selector and, within the method, message-sends, assignments, and returns of nonatomic objects. However, compiler optimizations may occasionally result in a different, nonintuitive step point, particularly in a loop.

Example 5.3 indicates step points with numbered carets.

Example 5.3

```
includesValue: value
^1

"Return true if the receiver contains an object of the same
value as the argument. Return false otherwise."

| found index size|

found := false.
      ^2
index := 0.
      ^3
size := self size.
     ^5      ^4
[ found not & (index < size)] whileTrue: [
        ^6  ^8       ^7       ^9
      index := index + 1.
            ^11      ^10
      found := value = (self at: index)
            ^14      ^13     ^12
      ].
^found
^15
```

If you use the GemStone debugger (described starting on page 5-27) to step through this method, the first step takes you to the point where `includesValue:` is about to be sent. Stepping again sends that message and halts the virtual machine at the point where `found` is assigned. Another step sends that message and halts the virtual machine just before the result is assigned to `index`, and so on.

When the GemStone Smalltalk virtual machine encounters an enabled breakpoint during normal execution, GemStone displays a GemStone Debugger. In this window, you can interactively explore the context in which execution halted.

Special considerations apply in setting breakpoints for primitive and special methods.

Breakpoints for Primitive 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 machine 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 execute the message-send at the bottom of the method and thus encounter the breakpoint.

Breakpoints for Optimized Methods

Certain simple methods are optimized by the GemStone Smalltalk compiler in such a way that they contain no step points. Naturally, you cannot set a method 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 class or pool variable or a variable defined in a symbol dictionary.

A method that performs 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 methods listed above, a handful of specific kernel class methods are specially optimized so that they cannot take breakpoints. Those methods are:

```
ifTrue:              untilFalse       timesRepeat:
ifFalse:             untilTrue        isNil
ifTrue:ifFalse:      whileFalse:      notNil
ifFalse:ifTrue:      whileTrue:       ==
_or:                 to:do:           ~~
_and:                to:by:do:        _class
isKindOf:            _isInteger       _isSmallInteger
_isSymbol            includesIdentical
```

Tools

To set breakpoints, GemBuilder provides the Breakpoint Browser. To debug the resulting stack, GemBuilder enhances the client Smalltalk debugger.

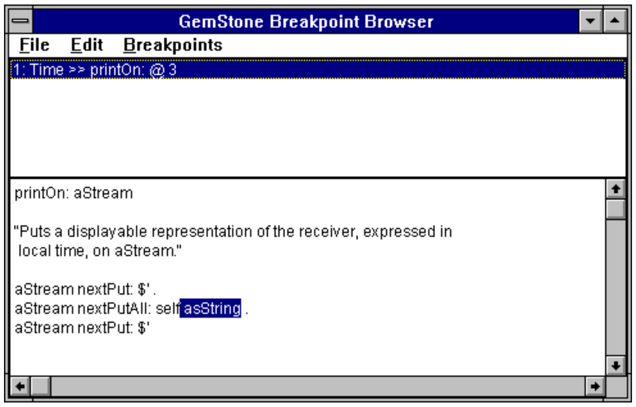
The Breakpoint Browser

You can set breakpoints in the source code pane of any browser, using the **set break** menu item described in Table 5.5 on page 5-9. You can also use the breakpoint browser, which lets you set, clear, and examine breakpoints for all classes and methods.

After you've set a breakpoint, for further convenience, you can use the menu items to disable or re-enable all breakpoints, or just selected ones.

A breakpoint browser has two panes: the list of break points on top, and the source code associated with the selected breakpoint on the bottom. Figure 5.5 shows an example:

Figure 5.5 GemStone Breakpoint Browser with a Breakpoint



```
1: Time >> printOn: @ 3
```

```
printOn: aStream

"Puts a displayable representation of the receiver, expressed in
 local time, on aStream."

aStream nextPut: $' .
aStream nextPutAll: self asString .
aStream nextPut: $'
```

The Break Pane

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 method break is set at step point 5 within the method `taxOwed` defined by class Employee.

The Source Pane

If you have selected a breakpoint in the break pane, the text area displays the source code for that method. This pane is similar to the GemStone Browser text area, but you cannot recompile an edited method by executing **Save**.

The Debugger

The GemStone Debugger is integrated with the client Smalltalk debugger, allowing you to:

- view GemStone Smalltalk and client Smalltalk contexts together in one stack,
- set and clear breakpoints without modifying source code,
- select a context from among those active on the virtual machine stack,

- examine and modify objects and code within that context, and
- continue execution either normally or in single steps.

The Debugger's stack pane displays the active call stack and allows 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 can 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.

If you select a GemStone Smalltalk context in the stack pane, the popup menu contains the item **show glue**. This lets you reveal additional GemBuilder stack that is normally of no interest. If this stack is already revealed, the menu item becomes **hide glue**.

Like other GemBuilder text areas,the debugger source code pane provides commands to execute GemStone Smalltalk, and it lets you examine step points and set breakpoints.

Getting a Stack Trace Without a Debugger

In a situation in which you encounter an error but cannot use the Smalltalk debugger (such as in a runtime application, or when the result is a virtual machine crash), you can write a stack trace to a text file. To do so, execute:

```
GbsConfiguration dumpAllProcessStacks
```

In response, all processes in the image write their contexts to a file named `stacksAt`*x*`.txt` in the current working directory, where *x* is a 10-digit number derived from a time stamp.

Chapter 6 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. For further information, see the chapter in the *GemStone Programming Guide* entitled "Transactions and Concurrency Control."

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.

6.1 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.

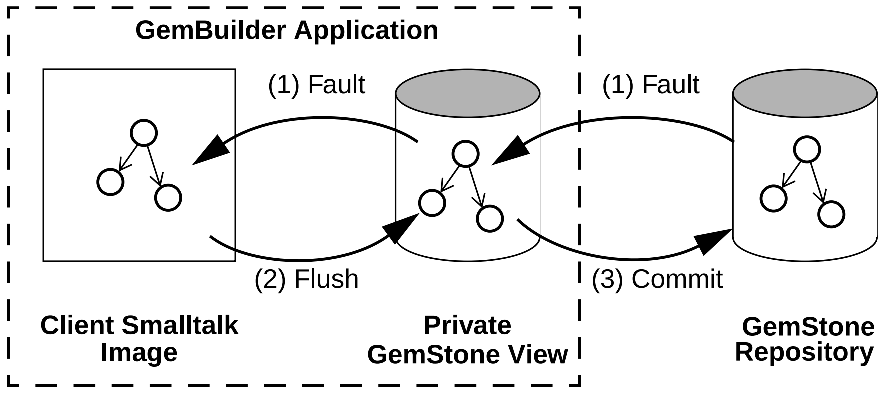
This chapter explains each of the topics mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced conflict classes. Be sure to refer to the related topics in the *GemStone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

6.2 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 6.1 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 6.1 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 2-12.)

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.

Avoiding or Handling Commit Failures

You can use the GemBuilder method GbsSession >> `hasConflicts` to determine if any concurrency conflicts exist that would cause a subsequent commit operation to fail. It returns `false` if it finds no conflicts with other concurrent transactions, `true` otherwise. You can then determine how best to proceed.

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

Following a commit failure, the client's view of persistent objects may differ from their precommit 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 6-10 for more discussion of the kinds of conflicts that can arise.)

6.3 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 6.1 shows GbsSession methods that support running outside of a GemStone transaction:

Table 6.1 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 either terminates the Gem or forces an abort, depending on the value of the related configuration parameter STN_GEM_LOSTOT_TIMEOUT. The Stone forces an abort by sending 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 your application receives `abortErrLostOtRoot`, the application must log out of GemStone and log back in, thus rereading all of its data in order to resynchronize its snapshot of the current state of the GemStone repository.

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.

An application modal dialog or a suspended user interface process prevents GemBuilder from handling the `abortErrLostOtRoot` signal until the dialog box is dismissed, or until the process resumes.

6.4 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.

6.5 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 6.1 shows how error handling might be implemented for the collection locking methods:

Example 6.1

```
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.

6.6 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 are 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 chapter in the *GemStone Programming Guide* entitled “Transactions and Concurrency Control.”

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.

6.7 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 “Multiprocess Applications” on page 9-19.)

Example 6.2 demonstrates notification in GemBuilder.

Example 6.2

```
"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, allowing the transmission of the sender’s session, a numerical signal value, and an associated message.

One Gem can handle a signal from another using the method GbsSession >> `sessionSignalAction:` *aBlock*, where *aBlock* is a one-argument block that will be passed a forwarder to the instance of GsInterSessionSignal that was received. From the GsInterSessionSignal instance, you can extract the session, signal value, and string.

One GemStone session sends a signal to another with:

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

For example:

Example 6.3

```
"First, set up the signal-receiving action"
GBSM currentSession sessionSignalAction: [ :giss |
      nil gbsMessenger
            comment: 'Signal %1 received from session %2: %3.'
            with: giss signal
            with: giss session
            with: giss 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'.
```

If the signal is received during GemStone execution, the signal is processed and execution continues. If *aBlock* is `nil`, any previously installed signal action is deinstalled.

NOTE

The method `sessionSignalAction:` *and the mechanism described above supersede the old mechanism that used the method* `gemSignalAction:`*. Do not use both this method and* `gemSignalAction:` *during the same session; only the last defined signal handler will remain in effect.*

See the chapter entitled “Error-handling” in your *GemStone Programming Guide* for details on using the error mechanism for change notification.

Chapter

7

Security and Object Access

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.

7.1 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 7-10, and "The Segment Tool" on page 7-10.

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.

7.2 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:

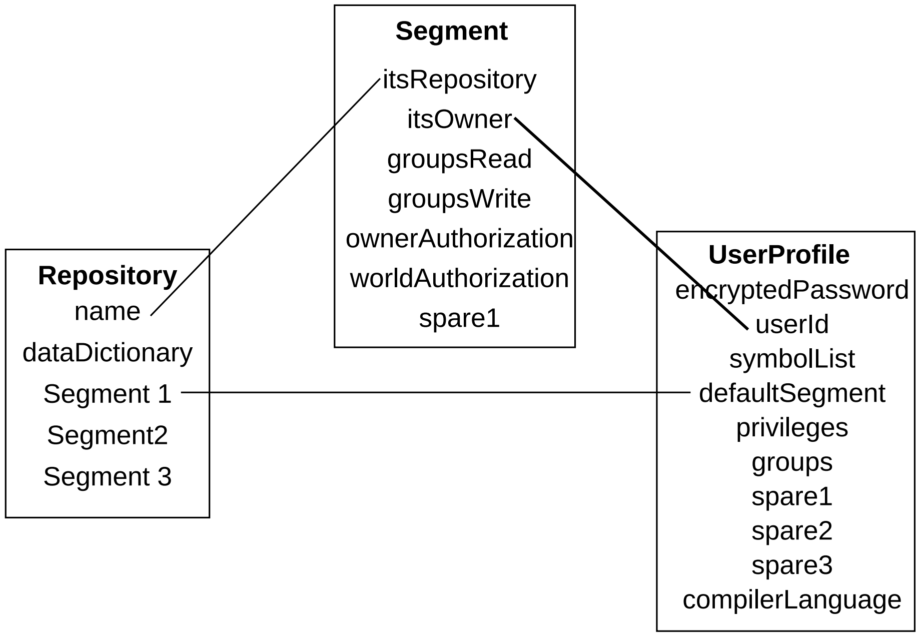
- the SystemSegment (owned by the SystemUser),
- the DataCuratorSegment (owned by the DataCurator), and
- the GcUser’s Segment (owned by the GcUser).

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 7.1 shows the relationship between the classes Repository, Segment, and UserProfile.

Figure 7.1 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
```

7.3 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 7-20).

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 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.

NOTE
For performance reasons, GbsSession uses a transient copies of your symbol list. If you change this transient copy programmatically, the changes are not immediately reflected in the permanent GemStone object. Also, changes to the permanent GemStone symbol list are not reflected in the GbsSession's transient symbol list until a transaction boundary. If you must be absolutely certain that the two copies are symchronized, log out and log back in again.

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

7.4 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

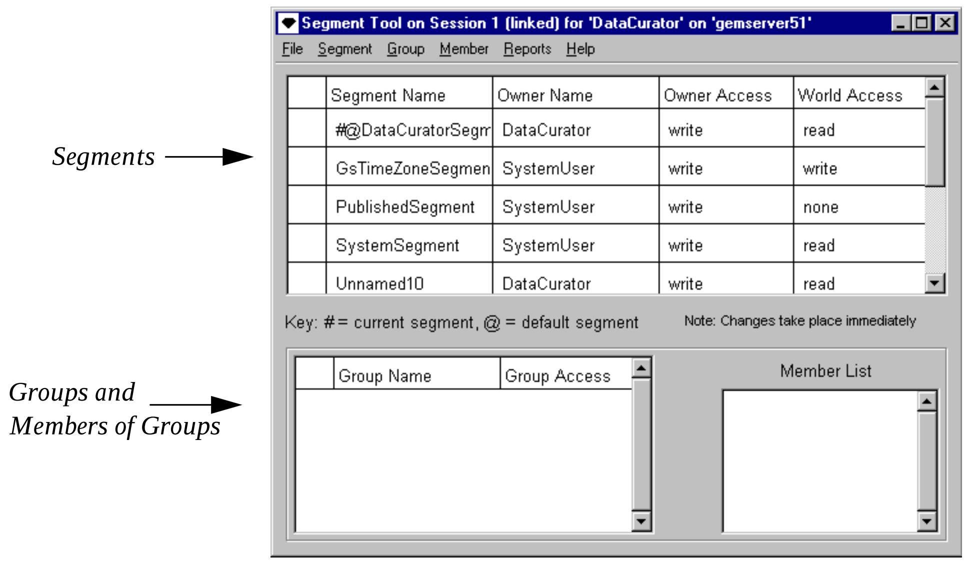
The Segment Tool allows you to inspect and change the authorization that GemStone users have at the object level. As explained in the section entitled

“Group Authorization and Object-sharing” beginning on page 7-7, 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.

To open a Segment Tool, select **Admin > Segments** from the GemStone menu or through the User Account Manager’s **Show Segments** button. Figure 7.2 shows a Segment Tool.

Figure 7.2 The Segment Tool


Segments
Groups and Members of Groups
Segment Tool on Session 1 (linked) for 'DataCurator' on 'gemserver51'
File Segment Group Member Reports Help
Member List

	Segment Name	Owner Name	Owner Access	World Access
	#@DataCuratorSegm	DataCurator	write	read
	GsTimeZoneSegmen	SystemUser	write	write
	PublishedSegment	SystemUser	write	none
	SystemSegment	SystemUser	write	read
	Unnamed10	DataCurator	write	read

Key: # = current segment, @ = default segment

Note: Changes take place immediately

	Group Name	Group Access

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.

NOTE

Changes made to cells in the tables are 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 7-13.

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 7.1 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 7.2 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 7.3 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 7.4 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 7.5 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

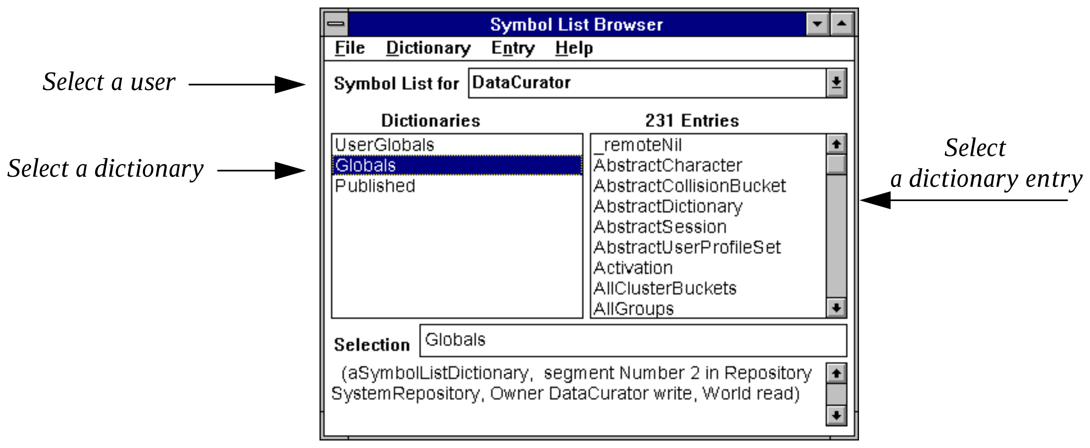
The Symbol List Browser is a tool for examining the GemStone SymbolLists associated with UserProfiles, adding and deleting dictionaries from these lists, examining the entries in those dictionaries and adding, deleting and inspecting the entries. References to dictionaries and dictionary entries can 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 **Admin > Namespaces** from the Gemstone 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 7.3 shows the Symbol List Browser.

Figure 7.3 The Symbol List Browser



Symbol List for: DataCurator

Dictionaries	231 Entries
UserGlobals	_remoteNil
Globals	AbstractCharacter
Published	AbstractCollisionBucket
	AbstractDictionary
	AbstractSession
	AbstractUserProfileSet
	Activation
	AllClusterBuckets
	AllGroups

Selection: Globals

(aSymbolListDictionary, segment Number 2 in Repository SystemRepository, Owner DataCurator write, World read)

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 7.6 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.

Dictionary Menu

The **Dictionary** menu allows you to rearrange dictionaries by cutting, copying, or pasting.

Table 7.7 Dictionary Menu in the Symbol List Browser

Cut	Removes the selected dictionary from the user's symbol list and places it in the session's clipboard.
Copy	Copies a reference to the selected dictionary into the session's clipboard.
Paste	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. If the name of the dictionary in the clipboard 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.
Add...	Prompts for name of a new Dictionary to be added to the symbol list.
Inspect	Opens a GemStone inspector on the selected Dictionary.

Entry Menu

The Entry Menu allows you to edit dictionary entries by cutting, copying, or pasting.

Table 7.8 Entry Menu in the Symbol List Browser

Cut	Removes the selected entry from its dictionary and places it in the session's clipboard.
Copy	Copies a reference to the selected entry into the session's clipboard.
Paste	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. If the clipboard entry'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.

Table 7.8 Entry Menu in the Symbol List Browser (Continued)

Add...	Prompts for name of a new entry to be added to the selected Dictionary.
Inspect	Opens a GemStone inspector on the selected entry.
Browse Class	If the selected entry is a class, opens a GemStone class browser on that entry.

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.

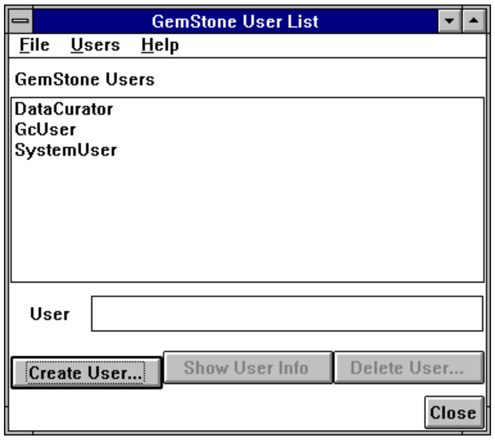
Note that you must either be DataCurator or have certain privileges to perform most of the system administration functions described in this section. If you are responsible for GemStone system administration, you should refer the chapter on administering user accounts and segments in the *GemStone System 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 GemStone User List, from the **GemStone** menu select **Admin Tools > Browse GemStone Users**. Figure 7.4 shows the GemStone User List.

Figure 7.4 GemStone User List



DataCurator
GcUser
SystemUser

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

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

Table 7.9 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 **Users** menu allows you to create a new user, display information about an individual user, and remove a user name. Its as buttons at the bottom of the window.

Table 7.10 GemStone User List: Users Menu

Create User...	Brings up a GemStone User dialog in which you can define a new user.
Show User Info...	Brings up a GemStone User window displaying privilege and group membership information on a selected user.
Delete User...	Allows you to remove a user name.

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 7.5.

Figure 7.5 GemStone User Dialog

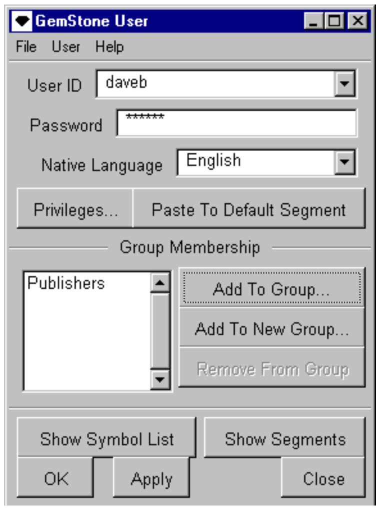


Table 7.11 shows the operations that are available in this dialog.

Table 7.11 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.

Table 7.11 Buttons in the GemStone User Window

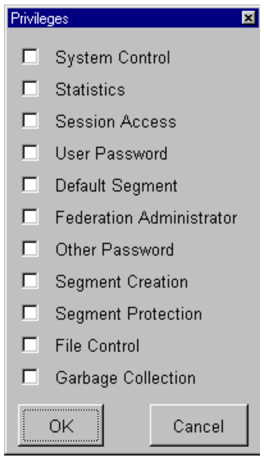
Add To New Group	Prompts you for a new group name and adds this user to the group.
Remove From Group	Removes this user from the selected group.
Show Symbol List	Brings up a Symbol List Browser for the designated user.
Show Segments	Brings up a Segment Tool.

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 7.6.

Figure 7.6 Privileges Dialog in GemStone User Window



The privileges and the methods that are associated with them are shown in Table 7.12.

Table 7.12 Privileges

Type of Privilege	Privileged Methods	In Class

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

Statistics	`stoneStatistics`	System

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

Table 7.12 Privileges

Type of Privilege	Privileged Methods	In Class

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

Chapter

8

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.

8.1 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 5-17 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 3, 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*.)

8.2 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.

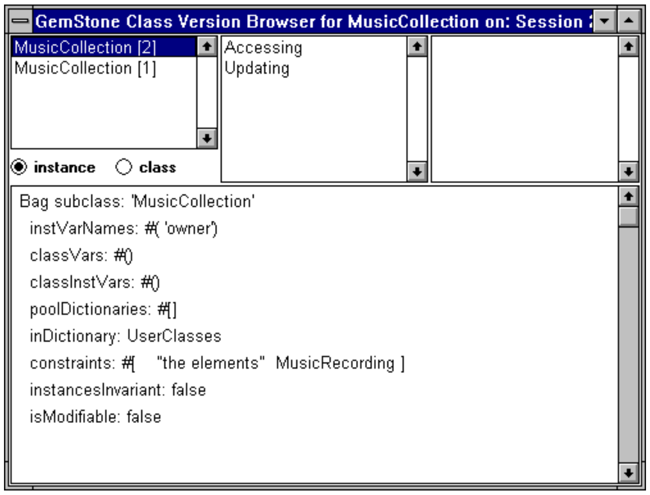
8.3 The Class Version Browser

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 **Browse 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 8.1.

Figure 8.1 The Class Version Browser



```
Bag subclass: 'MusicCollection'
  instVarNames: #( 'owner')
  classVars: #()
  classInstVars: #()
  poolDictionaries: #[]
  inDictionary: UserClasses
  constraints: #[    "the elements"  MusicRecording ]
  instancesInvariant: false
  isModifiable: false
```

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...** 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.

You can simultaneously inspect multiple class versions by holding down the shift key as you make your selections. Similarly, you can make a multiple selection to migrate the instances of several class versions to another version. Whenever more than one version is selected, only two menu items are accessible in the class pane: **Inspect Instances** and **Migrate Instances...**.

The commands available in the Class Version Browser are shown in Table 8.1.

Table 8.1 Classes Menu in Class Version Browser

File Out Methods Only	Writes GemStone Smalltalk code defining the selected class's versions methods to be written to a file in Topaz format. See "Saving Class and Method Definitions in Files" on page 5-19.	
	Name Each...	Prompts for a separate file name for each class.
	One Name...	Prompts for a single file name.
	Use Defaults	Uses a default name based on the class name.
File Out	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 read in 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 5-19.	
	Name Each...	Prompts for a separate file name for each class.
	One Name...	Prompts for a single file name.
	Use Defaults	Uses a default name based on the class name.
Browse Class	Opens a Browser on the selected class.	
Browse Versions	Spawns a Class Version Browser on the selected class.	
Browse Hierarchy	Spawns a GemStone Browser on the superclass hierarchy of the selected class.	

Table 8.1 Classes Menu in Class Version Browser (Continued)

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 of the selected class in the Text Area.
Move Classes...	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...	Removes 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.
Create Access	Creates methods for accessing and updating the instance variables of the currently selected class version.
Create in ST	Generates 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.
Inspect Instances	Opens an inspector on instances of the selected version.
Migrate Instances	Migrate all instances of the selected versions. Prompts you to select which version to migrate to.

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
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

Optimizing Space Management
explains how you can reclaim space from unneeded replicates.

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.

Multiprocess Applications
discusses nonblocking protocol and process-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.

9.2 Profiling

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 on performance in the *GemStone Programming Guide* 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 stub debugging support on (`stubDebugging: true`) or off (`stubDebugging: false`). When stub debugging is `true`, 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.

Another effect of setting `stubDebugging: true` is that operations involving stubs are recorded in the System Transcript. Turning on stub debugging and watching the faulting activity can help you evaluate your tuning parameters.

Notice that applications that rely on these methods might get incorrect results when stub debugging 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.

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.

GemBuilder Configuration Parameters

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
`autoMarkDirty`	true / false	true
`blockingProtocolRpc`	true / false	false
`blockReplicationEnabled`	true / false	true
`blockReplicationPolicy`	#replicate / #callback	#replicate
`bulkLoad`	true / false	false
`confirm`	true / false	true
`connectorNilling`	true / false	true
`connectVerification`	true / false	false
`defaultFaultPolicy`	#immediate / #lazy	#lazy
`eventPollingFrequency`	any integer	5000
`eventPriority`	any integer	3
`faultLevelLnk`	any integer	2
`faultLevelRpc`	any integer	4
`forwarderDebugging`	true / false	false
`freeSlotsOnStubbing`	true / false	true

Table 9.1 Configuration Parameters for GemBuilder

Parameter	Legal values	Default
`generateClassConnectors`	true / false	true
`generateGSClasses`	true / false	true
`generateSTClasses`	true / false	true
`initialCacheSize`	any integer	5003
`initialDirtyPoolSize`	any integer	157
`loginLinkedIfAvailable`	true / false	true
`removeInvalidConnectors`	true / false	false
`stubDebugging`	true / false	false
`processSafeCaches`	true / false	false
`traversalBufferSize`	any integer	250000
`traversalCompression`	true / false	false
`verbose`	true / false	false

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.

autoMarkDirty
Defines whether modifications to client objects are automatically detected. When `false`, the application must explicitly send `markDirty` to a client object after it has been modified, so GemBuilder will know to update the object in GemStone.

blockingProtocolRpc
Determines whether to use blocking or nonblocking protocol for RPC sessions. When `false`, nonblocking protocol is used, enabling other processes to execute in the image while one or more processes are waiting for a GemStone call to complete. When `true`, GemBuilder must wait for a GemStone call to complete before proceeding with the execution process that called it.

blockReplicationEnabled
When false, GemBuilder raises an exception when block replication is attempted—useful in determining if your application depends on block replication.

blockReplicationPolicy
Legal values are `#replicate` or `#callback`. Block replication requires decompiling and compiling the source code for blocks at runtime, which sometimes can cause probblems due to limitations on block replication. Block callbacks use client forwarders to evaluate the block in the client. Block callbacks escape the documented limitations of block replication, but do not perform well for blocks invoked repeatedly from GemStone.

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 improves performance for bulk data loading. When `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 set to `false`.

connectorNilling
When `true`, GemBuilder nils the Smalltalk object for certain session-based connectors after logout: all name, class variable, or class instance variable

connectors whose postconnect action is **#updateST** or **#forwarder**. When the last session logs out, the Smalltalk object references of global connectors are also set to `nil`. Fast connectors, class connectors, and connectors whose postconnect action is **#updateGS** or **#none** are not set to `nil`. Clearing connectors that depend on being attached to GemStone objects helps prevent defunct stub and forwarder errors.

When false, the logout sequence leaves the state of persistent objects in the image as it was.

`connectVerification`

When `true`, 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 `false`, these things are not checked. Set to `true` during development unless logging in becomes too slow, or your connector definitions are stable.

See “The Connector Browser” on page 4-13.

`defaultFaultPolicy`

Specifies GemBuilder’s default approach to updating client Smalltalk objects whose GemStone counterparts have changed. When **`#lazy`**, GemBuilder responds to a change in a GemStone object by turning its client Smalltalk replicate into a stub. The new GemStone value is faulted in the next time the stub is sent a message. When **`#immediate`**, GemBuilder responds to a change in a GemStone object by updating the client Smalltalk replicate immediately. The defaultFaultPolicy is implemented by Object >> `faultPolicy`. Subclasses can override this method for specific cases.

`eventPollingFrequency`

How often, in milliseconds, that GemBuilder polls for GemStone events such as changed object notification or Gem-to-Gem signaling.

`eventPriority`

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, or when GemStone signals you to abort so that it can reclaim storage, respectively.

`faultLevelLnk`

The default number of levels to replicate an object from GemStone to client Smalltalk in a linked session.

`faultLevelRpc`

The default number of levels to replicate an object from GemStone to client Smalltalk in a remote session.

`forwarderDebugging`

When `true`, forwarders support debugging by responding to some basic messages locally, such as `printOn:, instVarAt:,` and `class`, which returns GbsForwarder. When `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`, allowing stubs and their potentially outdates instance variables to be garbage collected if they become eligible. When false, GemBuilder does not alter instance variable values. To override this behavior on a class-by-class basis, reimplement `#freeSlotsOnStubbing` (inherited from Object).

`generateClassConnectors`

When `true`, 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`**. When `false`, session connectors are not automatically created.

See “Class Mapping” on page 3-6.

`generateGSClasses`

When `true`, 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. When `false`, GemBuilder raises an error.

See “Class Mapping” on page 3-6.

`generateSTClasses`

When `true`, 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. When `false`, behavior is defined by the client Smalltalk image.

See “Class Mapping” on page 3-6.

`initialCacheSize`

The size in bytes of the initial cache for each GemStone session. For best performance, make this a prime number.

See “Changing the Initial Cache Size” on page 9-19.

`initialDirtyPoolSize`

Initial size of the GbsSession dirtyPool identity set. For bulk loading, increasing this value reduces the number of times the set needs to grow. For applications that flush a small number of objects, decreasing this value (while keeping it larger than the number of objects being flushed) improves flushing performance.

`loginLinkedIfAvailable`

When `true`, the result of executing `GBSM login` is a linked GemStone session, which provides faster repository access, unless GemBuilder cannot start a linked session (as is the case on some platforms) or another session is already running linked. When false, the result of executing GBSM login is always a remote GemStone session.

`removeInvalidConnectors`

When `true` and **confirm** is `false`, if a connector fails to resolve at login, it is removed from the connector collections so that the issue does not arise again at next login.

When `true` and **confirm** is `true`, you are prompted to remove invalid connectors during login.

When `false`, invalid connectors are ignored.

See “The Connector Browser” on page 4-13.

`stubDebugging`

When `true`, stubs support debugging by responding to some basic messages locally, such as `printOn:, instVarAt:,` and `class`, which returns GbxObjectStub. When `false`, these messages cause the stub to fault into the client image from GemStone.

`processSafeCaches`

When `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. Nonblocking RPC sessions always use process-safe caches regardless of this parameter setting.

`traversalBufferSize`

Sets the size, in bytes, of the buffer used in traversal replication.

`traversalCompression`

When `true`, GemStone compresses the traversal buffer contents for transmission to the client, and the client decompresses the traversal buffer contents upon receipt, thus reducing the amount of data sent across a network connection to an RPC Gem. If the network connection between the client and server transmits less than one million bits per second (slower than a T1 line), compression will probably improve transmission speeds, assuming that the client and server machines are equivalent to 75 MHz 486 CPUs or faster.

`verbose`

When `true`, GemBuilder prints messages to the Transcript when certain events occur, such as logging a session in or out, or committing or aborting a transaction. When `false`, these messages are not printed.

The Settings Browser

The Settings Browser makes it easy to examine and set the configuration parameters for GemBuilder.

Opening the Settings Browser

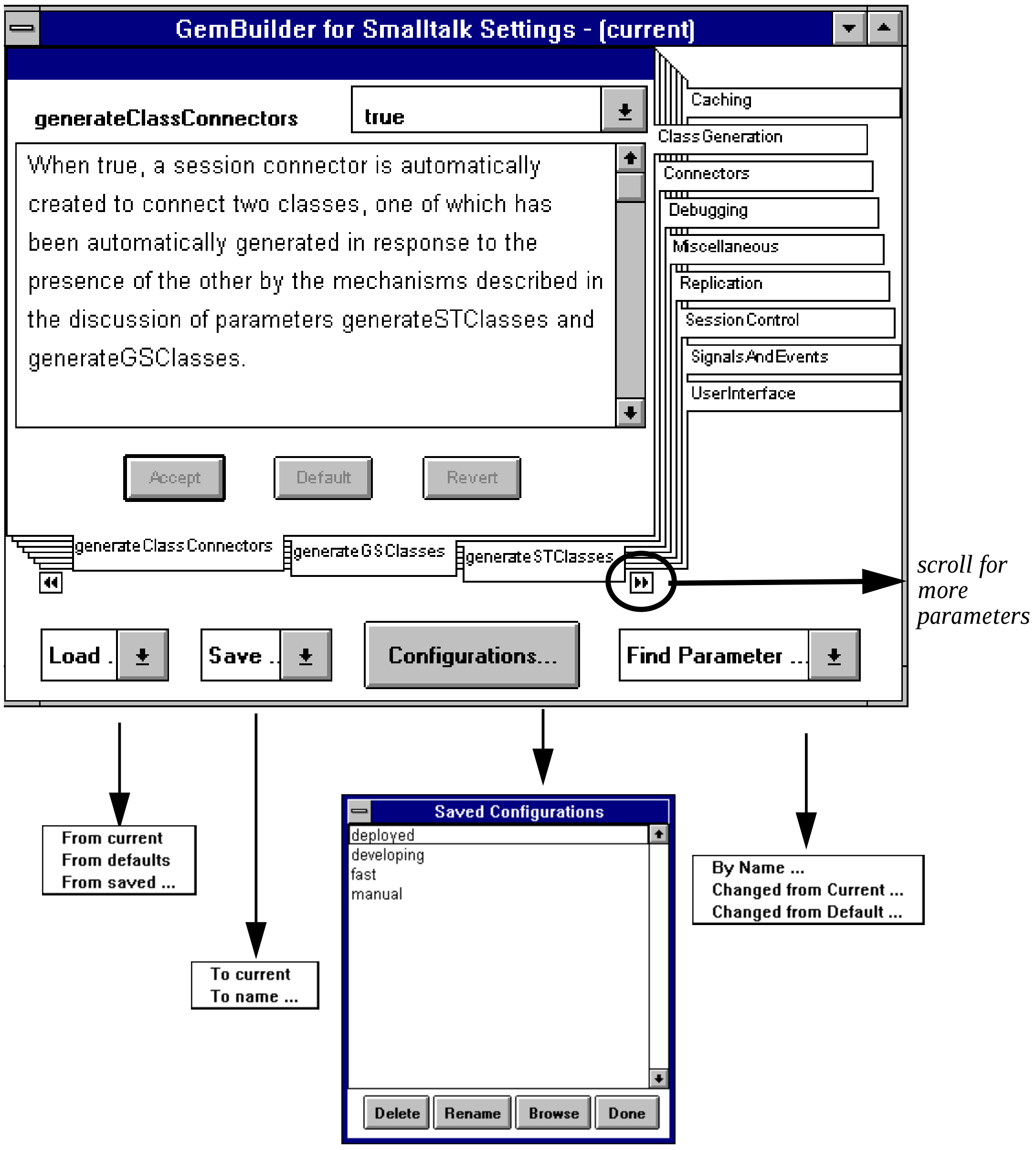
To open the Settings Browser, select **Tools > Settings** from the GemStone menu.

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



GemBuilder for Smalltalk Settings - (current)

generateClassConnectors true

When true, 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.

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 3. 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 Fault Level

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 `replicationSpec` class method for that class. For details, see “Replication Specifications” on page 3-21.

Setting theTraversal Buffer Size

The traversal buffer is an internal buffer that GemBuilder uses when retrieving objects from GemStone. The larger the traversal buffer size, 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.
```

You can also change this value by using the Settings Browser: choose **GemStone > Browse > Settings** and select the **Replication** tab in the resulting Settings Browser.

9.5 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.

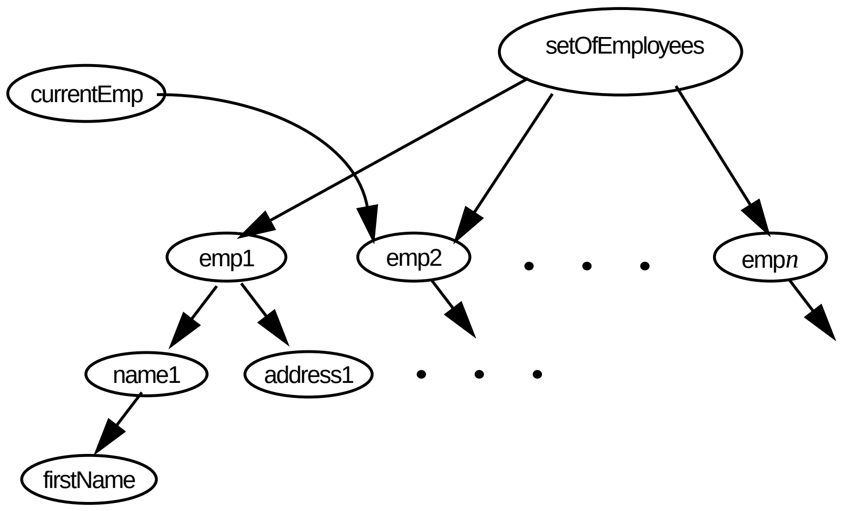
Measures you can take to control the size of the client Smalltalk object cache include explicit stubbing, using forwarders, and not caching certain objects.

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.

Also, be aware of the 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 in as a replicate ("unstubbed") before the assignment can occur. The access method causes the stub to be faulted in and yields the correct result. Direct assignment, however, does not cause the stub to be faulted in and can cause errors:

```
self stubYourself.
self instVarX: 5.             (reliable)

self stubYourself.
instVarX := 5.                (unreliable)
```

Using Forwarders

Another solution is to declare the `setOfEmployees` as a forwarder. See "Forwarders" on page 3-9.

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.

9.6 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 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, 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.

To load the user action in client Smalltalk, execute:

```
GBSM loadUserActionLibrary: userAction
```

9.7 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.

9.8 Multiprocess Applications

Some applications support multiple Smalltalk processes running concurrently in a single image. In addition, some applications enter into a multiprocess state occasionally when they make use of signalling and notification. Multiprocess GemBuilder applications must exercise some precautions in order to preserve expected behavior and data integrity among their concurrent processes.

Process-safe Transparency Caches

By default, GemBuilder uses transparency cache dictionaries that are not process-safe. To use transparency cache dictionaries that are protected for use with multiprocess client Smalltalk applications, you must set the GemBuilder configuration parameter **processSafeCaches** to `true` by changing its setting in the Settings Browser or by sending the message:

```
aGbsSession processSafeCaches: true
```

When **processSafeCaches** is `true`, subsequent logins use transparency cache dictionaries that are protected. However, 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 execution process 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 processes 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 execution process must wait for a GemStone call to return before proceeding.

One Process per Session

Applications that limit themselves to one process per GemStone session are relatively easy to design because each process has its own view of the repository. Each process can rely on GemStone to coordinate its modifications to shared objects with modifications performed by other processes, each of which has its own session and own view of the repository. For such applications, setting **processSafeCaches** to true is the only additional precaution required. If at all possible, try to limit your application to one process per GemStone session.

Multiple Processes per Session

Applications that have multiple processes running against a single GemStone session must take additional precautions.

You may not have designed your application to run multiple processes under a single GemStone session. However, if your application uses signals and notifiers, chances are it is occasionally running two processes 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 process. Unless your main execution process is idle when these events occur, you need to take the same precautions as any other application running multiple processes against a single session.

Applications that have multiple processes 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 process 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 processes need to be in agreement before a commit or abort occurs. For example, suppose two processes share a single GemStone session. If one process is in the process of modifying a set of persistent objects and a second process 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 process the transaction controller for a session, and require that all other processes sharing that session request that process for a transaction state change. The controller process can then be blocked from performing that change until all other processes using that session have relinquished control by menas of 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 process modifies persistent objects, it must protect against other processes 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 processes send a message to a stub at roughly the same time, one of the processes 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.

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.

Delegate Objects
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.

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.

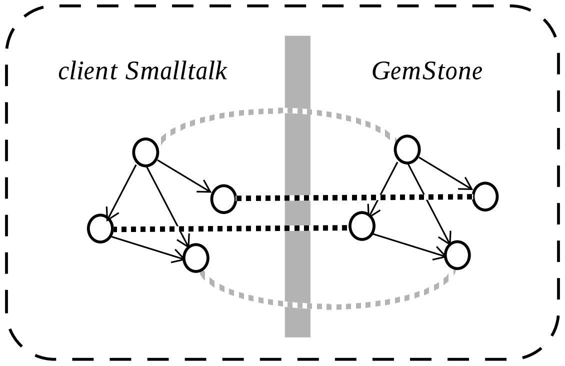
GemBuilder's transparency features provide both convenience for the developer and data integrity for the application, but they do incur some overhead. The optimizations in this chapter can offer some efficiency gains, but you, as the developer, must give up some convenience and take responsibility for 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

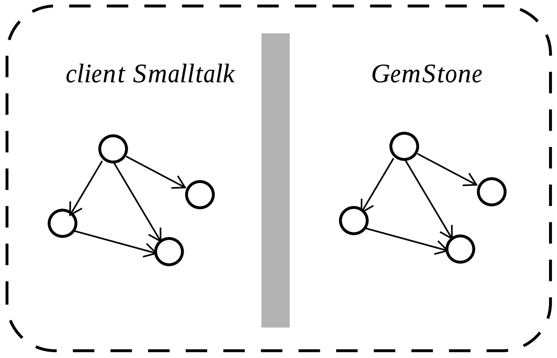
Chapter 4 discussed how a replicate created in client Smalltalk can maintain a link to a GemStone Smalltalk object so that changes in either the GemStone Smalltalk or the client Smalltalk object could propagate to the other. Figure 10.1 illustrates this.

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 client 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 and 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*; 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 the prefix `Gbx`. GemBuilder methods can be marked private in any of several ways:

- The names of private methods in base class extensions begin with the prefix `gbx`.
- Some methods specify they are private in the method comment.
- Other methods are categorized as private either in a method category, or an ENVY category, marked “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 Delegate Objects

Instances of `GbsObject` are a way for your client Smalltalk application to refer to local objects associated with objects in the GemStone repository. Through these instances of GbsObject, the client Smalltalk objects can send messages to GemStone objects, bypassing any local forwarders, stubs, or replicates.

Get a `GbsObject` by:

- sending `asGSObject` to an existing replicate of a GemStone object;
- using a predefined `GbsObject` for kernel objects; or
- 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 for the sought-after GemStone objects. For example, the following expression finds the object named `UserGlobals` in the user's symbol list and return a `GbsObject` for it:

  ```
  aDelegateObjectForUserGlobals :=
     aGbsSession resolveSymbol: #UserGlobals
  ```

  Although you can use `resolveSymbol:` to obtain a local object corresponding to 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` that 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 local delegate for a 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 the local delegate for the GemStone Dictionary class object, `Onil` refers to the local delegate for GemStone `nil`, and `Otrue` refers to the local delegate 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 represent the same GemStone object.)

Appendix A of this manual, "GemBuilder Classes and GbsObjects," lists the GemStone objects that are defined in SpecialGemStoneObjects.

Sending Messages Through GbsObject Delegates

GemBuilder provides two mechanisms for sending messages to GemStone objects through their `GbsObject` delegates: `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

```
| myGSArray mySession |
mySession := GBSM currentSession.
myGSArray :=
  (mySession resolveSymbol: 'Array') remotePerform: #new.
myGSArray
```

Actually, because GemBuilder provides a predefined delegate for each GemStone kernel object, you can simplify the expression by using the predefined OArray surrogate, instead of creating a new delegate with the `resolveSymbol:` message (Example 10.2).

Example 10.2

```
   myGSArray := 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 delegate object does not understand a message, it checks for the `gs` prefix. If the delegate 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

```
| myGSArray |
myGSArray := OArray gsnew.
```

The following two expressions are also equivalent:

```
myGSArray remotePerform: #at:put: with: 1 with: 10
myGSArray gsat: 1 put: 10
```

In both styles of message passing, the selector (`remotePerform:` or `gsat:put:`) is a local client Smalltalk Symbol object and the arguments can be either 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 is 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 delegates 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 delegate 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 delegates for both the collection and the selection block, then instruct GemStone to perform the operation and return its results, also as a delegate:

Example 10.4

```
| empSetDelegate aSelBlockDelegate selResultDelegate mySession |

mySession := GBSM currentSession.

"Get a delegate for the set of Employees"
empSetDelegate := mySession resolveSymbol: #MyEmps.

"Make a GemStone SelectionBlock and get a delegate for it"
aSelBlockDelegate := mySession execute:
            '{:each | each.name.first = ''Joebob''}'.

"Execute the query"
selResultDelegate := empSetDelegate gsselect: aSelBlockDelegate.
selResultDelegate
```

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

```
| empSetDelegate aSelBlockDelegate selResultDelegate nameString
      queryString mySession |

mySession := GBSM currentSession.

"Get a delegate for the set of Employees"
empSetDelegate := mySession 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 delegate for it"
aSelBlockDelegate := mySession execute: queryString.

"Execute the query"
^selResultDelegate := empSetDelegate gsselect: aSelBlockDelegate.
```

Converting GbsObjects to Replicates

A `GbsObject` 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` to execute GemStone Smalltalk code in GemStone, the result is returned as a `GbsObject`.

To create a replicate from a `GbsObject` and continue local execution of messages, you can send `asLocalObject` to the instance of GbsObject.

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

```
| myGSArray mySession |

mySession := GBSM currentSession.

"Make a new GemStone Array"
myGSArray := mySession execute: 'Array new: 30'.

"Fetch the object at position 1"
myGSArray fetchVaryingOOPAt: 1.

"Fetch 2 objects starting at position 1"
myGSArray fetch: 2 idxOOPsAt: 1.

"Store the GemStone object nil at position 30"
myGSArray storeIdxOOP: (ONil) at: 30.
```

Each of the structural access fetching methods either returns an instance of `GbsObject` or an Array of `GbsObjects`.

There are similar methods for fetching from and storing in named and anonymous (unordered) pointer instance variables and for accessing byte objects such as Strings.

Example 10.7

```
| aGbsString aGbsIDBag mySession |

mySession := GBSM currentSession.

aGbsString := mySession execute: 'String new: 100'.
aGbsIDBag := mySession 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.

NOTE

When you add elements to a GemStone UnorderedCollection (a Bag, Set, or Dictionary) using a GemBuilder for Smalltalk structural access call such as `addOOPs:`*, the OOPs are not immediately added to the collection. To get around this problem, after using one or more GemBuilder for Smalltalk structural access calls such as GbsObject >>*

`addOOPs:` *or* `replaceOOPs:` *to store into a GemStone UnorderedCollection, you must send* `processDeferredUpdates` *to the GbsSession to which those objects belong. Before sending* `processDeferredUpdates`*, you must ensure that none of the objects stored into those collections are forward references—they must be fully-created, initialized objects.*

Changes to UnorderedCollections made using structural access calls are not visible until after `processDeferredUpdates` *has been called. If you commit the session without calling* `processDeferredUpdates`*, it is called for you automatically.*

You can also make nontransparent copies, as described in “Client Copies” on page 3-37.

10.4 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 Error-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.

11.1 Error-handling and Recovery

An instance of GbsError is created when GemBuilder encounters a GemStone error. Each GbsError can represent itself as an exception. Your application can use these exceptions to perform client Smalltalk exception-handling. When an error is detected, GemBuilder creates an instance of GbsError and raises its signal.

Error-handlers in your application are typically stack-based, but you may wish to install a session-based error-handler instead of, or in addition to, stack-based error handlers. Finally, if no handler is defined, the default handler opens adebugger.

Stack-based Error-handling

You can use the `on:do:` method to install error handlers to anticipate specific GemStone errors, as shown in Example 11.1.

Example 11.1

```
[ GBSM execute: '#( 1 2 3 ) at: 4']
   on: (GbsError signalFor: #objErrBadOffsetIncomplete)
   do: [ :sig |
         sig halt: 'proceed to inspect bad offset error.'.
         sig originator inspect ]
```

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:

```
[. . . ]
  on: (GbsError signalFor: #compilerErrorSignal)
  do: [:ex|. . . ]
```

You can also create a handler to check for multiple errors:

```
[ . . . ]
      on: (GbsError signalFor:#interpreterErrorSignal),
          (GbsError signalFor: #rtErrAbortTrans)
do: [:ex| . . .]
```

Session-based Error-handling

You can define an error-handler that is global to your entire session instead of being installed in an active context. For example:

Example 11.2

```
GBSM currentSession
      onEventSignal: (GbsError signalFor: #objErrBadOffsetIncomplete)
      handle: [ :sig |
         sig halt: 'proceed to inspect bad offset error.'.
         sig originator inspect ]
      raiseException: true
```

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 11.3

```
"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:

```
^[GBSM execute: 'System signal: 10
               args: #[ 46 ]
               signalDictionary: MyErrors']
   on: GbsError errorSignal
   do: [ :ex | ex return: #handled ].
```

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 newChild.
```

So now, to explicitly handle your new error from client Smalltalk:

Example 11.4

```
^[ GBSM execute: 'System signal: 10 args:  #[ 46 ]
         signalDictionary: MyErrors' ]
      on: (GbsError signalFor: #myNewError)
      do: [ :ex | ex return: #handled ]
```

For information on how to create GemStone error dictionaries and how to handle GemStone errors (predefined and user-defined) within the GemStone environment, see the chapter entitled “Handling Errors” in the *GemStone Programming Guide*.

For more information about defining error handlers in the client Smalltalk, refer to your client Smalltalk documentation on exception-handling.

11.2 Detecting GemStone Interrupts

Interrupt detection allows a soft break after one hard-break character (formerly Control-c), and a hard break after three. GemBuilder uses the native client Smalltalk handler for such interrupts, which can be detected only in RPC nonblocking mode.

Appendix **A**

GemBuilder Classes and GbsObjects

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 4. 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              OClassSet
OAbstractCollisionBucket        OClientForwarder
OAbstractDictionary             OClusterBucket
OAbstractUserProfileSet         OClusterBucketArray
OAllClusterBuckets              OCollection
OArray                          OComplexBlock
OAssociation                    OComplexVCBlock
OAutoComplete                   ODatabaseConversion
OBag                            ODate
OBasicSortNode                  ODateTime
OBehavior                       ODecimalFloat
OBlockClosure                   ODictionary
OBoolean                        ODoubleByteString
OByteArray                      ODoubleByteSymbol
OCanonicalStringDictionary      OEUCString
OCharacter                      OEUCSymbol
OCharacterCollection            OEmptyInvariantArray
OClampSpecification             OEmptyInvariantString
OClass                          OException
OClassHistory                   OExecutableBlock
OClassOrganizer                 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 Packaging Runtime Applications

Use the following guidelines when packaging a client Smalltalk application that uses GemBuilder to access GemStone.

B.1 Prerequisites

In addition to code required by your application, the packaged image must contain the application or parcel GbsRuntime, which contains the system code modified for GemBuilder.

In order to ensure that your image initializes correctly, your application must specify GbsRuntime as a prerequisite.

Do not include the application or parcel GbsTools. These are subclasses of classes that will be deleted during the packaging process.

Names

Ensure that your image is packaged to include class pool dictionaries and instance variable names and does not remove them.

Replicating Blocks

To ensure that your application behaves in the same manner as it did in the development environment, we recommend that you include the compiler.

Defunct Stubs and Forwarders

Defunct stubs and forwarders cause problems during packaging. To avoid these problems, start with new client image as shipped from your client Smalltalk vendor.

Shared Libraries

A deployed runtime application that uses GemBuilder needs to contain all the shared libraries from the GemBuilder `/bin` directory, as well as `englis`*`xx`*`.err` `(`where *`xx`* is the release number).

If you are logging in only remote sessions, set the GemBuilder configuration parameter `loginLinkedIfAvailable` to *false* in your image; you can then omit the file `gcilw`*`xx`*`.dll` (Windows-based systems) or `gcilw`*`xx`*`.so` (Solaris) or `gcilw`*`xx`*`.sl` (HP-UX).

B.2 Packaging

Step 1. Open a new client image as shipped from your client Smalltalk vendor.

Step 2. Ensure that you have satisfied the prerequisites given above.

Step 3. Load your application code.

Step 4. Follow the packaging instructions given by your Smalltalk vendor.

Appendix C 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.

C.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 "gemserver60" running on node "handel", you would specify two NRS strings:

```
topaz> set gemstone !@handel!gemserver60
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\!gemserver60 -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!gemserver60
```

```
% waitstone "!tcp@oboe#auth user@password!gemserver60"
```

C.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 C-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.

C.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**)

C.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 `netldi60`. (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 D — Client Smalltalk and GemStone Smalltalk

This appendix outlines the few general and syntactical differences between the IBM VisualAge 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]
```

Block Temporaries

IBM Smalltalk supports block temporaries. It does not, however, permit the declaration of a temporary variable in an inner block if a temporary variable of the same name has been declared in an outer scope.

One-way become:

In GemStone's Smalltalk, `become:` swaps the identities of the receiver and the argument.

IBM Smalltalk implements a one-way `become:`. The following code returns true in IBM Smalltalk, and false in GemStone:

```
| a b |
a := Object new.
b := Object new.
a become: b.
a == b
```

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 VisualAge is accomplished by sending messages such as `whenExceptionDo:`, `when:do:`, or `when:do:when:do:`. In VisualAge the argument to `when:` is a predefined ExceptionalEvent. From within the handler block, messages can be sent to the ExceptionalEvent to cause the flow of control to resume, exit the `when:do:` block, or restart the block.

Index

A

abort
  (GbsSession) 6-4, 6-5
  (GbsSessionManager) 6-5, 6-7
abort request from GemStone 6-7
abortErrLostOtRoot signal 6-8
abortTransaction (GbsSession) 6-8
accessing files on host 10-13
activation exceptions 11-2
addDependent: 2-12
adding connector to session or global list 4-9
addParameters (GbsSession) 2-6
addToCommitOrAbortReleaseLocksSet: (System) 6-15
addToCommitReleaseLocksSet: (System) 6-15
application design 1-7–1-9, 3-2–3-6
argument in message to forwarder 3-10
array constructors in GemStone Smalltalk D-2
asForwarder 3-9
asGSObject 10-5
asGSObjectCopy 3-38
asLocalObjectCopy 3-37
assertionChecks configuration parameter 9-5, 9-7
assigning a migration destination 8-3
authorization 7-6, 7-7
  and migration 8-4
autoMarkDirty configuration parameter 9-5, 9-7
automatic class generation 3-7–3-8, 9-9
  disabling 3-7
  interactions with replication specifications 3-7, 3-23
automatic transaction mode 6-8, 6-9
  defined 6-8

B

behavior, replicating 3-8
binary selectors, special treatment of 10-8

block
  callback 3-35
  replicating 3-32, 9-7
blockingProtocolRpc configuration parameter 9-5, 9-7
blockReplicationEnabled configuration parameter 3-32, 9-5, 9-7
blockReplicationPolicy configuration parameter 9-5, 9-7
break
  hard 11-4
  soft 11-4
breakpoints 5-22, 5-26
  and primitive methods 5-25
  and special methods 5-25
  methods that cannot have 5-25
bulkLoad configuration parameter 9-5, 9-7
business objects 1-8

C

cache
  making process-safe 9-10
  size, changing 9-19
  size, initial 9-10
  space management 9-16
callback for blocks 3-35
changed object notification 6-17, A-1, B-1
changing
  cache size 9-19
  connector initialization 4-16
  initial cache size 9-19
  internal structure of GemStone objects, with methods 10-11
  postconnect action 4-16
  schema 8-2
    class versions and 8-6
  shared data 3-12, 6-10
choosing the locus of execution 9-3
circular constraints 5-12
class 3-6
  connector 3-6, 4-4, 9-9
    connection order 4-5
  connector, updating 3-6
  customizing faulting 3-21
  filing out 5-19
  generating automatically 3-7–3-8, 9-9
  mapping 1-4, 8-6
  mapping to one with a different storage format 3-31
  migrating instances to a new version 8-2
  nonforwarding 4-4
  private 10-4
  reduced-conflict 6-16
  structure, matching on client and server 3-6
  updating definitions with connectors 4-4
  versions and replication specifications 3-23
  versions of 8-2
class instance variable connector 4-5
class variable connector 4-4
Classes menu
  in GemStone Browser 5-7
Classes pane
  in GemStone Browser 5-5
`clearCommitOrAbortReleaseLocksSet` (System) 6-15
`clearCommitReleaseLocksSet` (System) 6-15
client forwarder 3-9
code pane
  breakpoint browser 5-27
  menu for 5-8
`commit`
  (GbsSession) 6-4
`commitAndReleaseLocks` (System) 6-15
committing
  a transaction 1-4
    performance 6-16
  and flushing, compared 3-14
  changes to the repository 6-4

compile in ST command
  in Browser's Class menu 8-9
compiling
  a class definition 5-13
  GemStone Smalltalk 10-8
  in a runtime application B-2
  special treatment of binary selectors in client Smalltalk 10-8
compressing traversal data for transmission 9-11
concurrency
  modes, setting 6-12
  optimistic control 6-11
  pessimistic control 6-11, 6-12
concurrent transactions, managing 6-10
configuration parameters 9-5–9-11
  assertionChecks 9-5, 9-7
  autoMarkDirty 9-5, 9-7
  blockingProtocolRpc 9-5, 9-7
  blockReplicationEnabled 3-32, 9-7
  blockReplicationPolicy 9-5, 9-7
  blockRepliucationEnabled 9-5
  bulkLoad 9-5, 9-7
  confirm 9-5, 9-7
  connectorNilling 3-11, 9-5, 9-7
  connectVerification 9-5, 9-8
  defaultFaultPolicy 3-20, 9-5, 9-8
  eventPollingFrequency 9-5, 9-8
  eventPriority 9-5, 9-8
  faultLevelLnk 3-19, 9-5, 9-8
  faultLevelRpc 9-5, 9-9
  faultLeveRpc 3-19
  forwarderDebugging 9-5, 9-9
  freeSlotsOnStubbing 9-5, 9-9
  GemStone
    CONCURRENCY_MODE 6-12, 7-1
    STN_GEM_ABORT_TIMEOUT 6-8
  generateClassConnectors 3-7, 9-6, 9-9
  generateGSClasses 3-7, 9-6, 9-9
  generateSTClasses 3-7, 9-6, 9-9
  initialCacheSize 9-6, 9-10
  initialDirtyPoolSize 9-6, 9-10
  loginLinkedIfAvailable 2-2, 9-6, 9-10
  processSafeCaches 9-6
  removeInvalidConnectors 9-6, 9-10
  setting and examining 9-6
  stubDebugging 9-6, 9-10
  traversalBufferSize 9-6, 9-10, 9-15
  traversalCompression 9-6, 9-11
  verbose 9-6, 9-11
confirm configuration parameter 9-5, 9-7
conflicts in transactions
  reducing, and performance 6-16
  write / write, and RcQueue 6-17
Connected command in Connector Browser 4-2, 4-16
connected objects, synchronizing 4-15

connector 4-1–4-16
  adding to session or global list 4-9
  class 3-6, 4-4, 9-9
    class versions and 3-8
    connection order 4-5
    forwarders and 3-8
    update direction and 3-8
    updating 3-6
  class hierarchy 4-7
  class instance variable 4-5
  class variable 4-4
  connecting object networks 3-4
  connection order 4-5
  controlling 4-10
  creating automatically 9-9
  creating interactively 4-16
  creating programmatically 4-7
  defined 3-3, 4-1
  fast 4-6
  faulting upon connection 3-13
  for kernel classes 4-5
  global 4-2, 4-14
  initializing 4-3
  introduction to 1-4
  list of 4-9
  name 4-4
  nilling 3-11, 9-7
  postconnect action 3-9, 4-3
  removing duplicates 4-2
  removing invalid 9-10
  removing unresolved 4-15
  scope 4-2
  session 4-14
  setting postconnect action programmatically 4-8
  setup for initial storage of data in GemStone 4-16
  updateGS postconnect action 4-16
  updateST postconnect action 4-16
  updating class definitions and 4-4
  verifying 4-3, 4-15, 9-8

Connector Browser 4-13–4-16
  updateGS postconnect action 4-16
  updateST postconnect action 4-16
connectorNilling configuration parameter 3-11, 9-5, 9-7
connectVerification configuration parameter 9-5, 9-8
constraints
  circular 5-12
  on instance variables 5-12
contexts 5-23
Control-c interrupt-handling 11-4
controlling the size of the client Smalltalk object cache 9-16
converting among forwarders, stubs, replicates, and delegates 3-39
copying
  client objects 3-38
  GemStone objects 3-37
create access command
  in Browser's Class menu 8-9
create in ST command
  in Browser's Class menu 5-9, 8-9
creating
  connector interactively 4-16
  connector programmatically 4-7
  forwarder 3-9
  forwarder interactively 4-16
  linked session 2-5
  new instances of GemStone classes 10-11
  remote session 2-5
  subclasses 5-13
current segment 7-6
current session 2-9
  setting 2-11
  tools attached to 2-10

D

data
  cost of managing 9-3
  modifying shared 3-12, 6-10
  storage in GemStone 4-16

Database Users List (User Account Manager) 7-23
debugger 5-22, 5-27
debugging
  forwarders 9-9
  getting stack trace without debugger 5-28
  stubs 9-10
  support in GemBuilder 9-4
decimals, replicating 3-36
default directory, and host file access methods 10-13
default segment 7-5
defaultFaultPolicy configuration parameter 3-20, 9-5, 9-8
defining GemStone errors 11-3
defunct forwarder 3-11
  during packaging B-2
defunct stub 3-21
  during packaging B-2
delegate
  converting 3-39
  defined 3-3
  sending messages through 10-6
delegates 10-5–10-10
  sending messages through 10-6
dependencies between objects, managing with replication specififcations 3-25
dependents, session 2-12–2-15
  adding 2-12
  committing a transaction 2-12
  removing 2-12
dictionaries
  adding Associations to 5-11
  Globals 7-9, 7-10
  pool 5-14
  shared 7-9–7-10
  specifying for a new class 5-14
  UserGlobals 7-9, 7-10
dirty
  defined 3-12
disabling
  automatic class generation 3-7
  block replication 3-32

Disconnected command in Connector Browser 4-2, 4-16
domain objects 1-8
`dumpAllProcessStacks` (GbsConfiguration) 5-28

E

error, user-defined 11-3
error-handling
  during file in 5-22
  in client Smalltalk and in GemStone D-3
  session-based 11-2
  stack-based 11-2
event, polling for 9-8
eventPollingFrequency configuration parameter 9-5, 9-8
eventPriority configuration parameter 9-5, 9-8
examining the internal structure of a GemStone object 5-11, 10-11
exception-handling 11-1–11-4
  see error-handling
  session-based 11-2
  stack-based 11-2
`exclusiveLock:` (GbsSession) 6-13
`exclusiveLock:ifDenied:ifChanged:` (GbsSession) 6-13
`exclusiveLockAll:` (GbsSession) 6-13
`exclusiveLockAll:ifIncomplete:` (GbsSession) 6-14
`execute:` (GbsSession) 10-8
execution
  in GemStone 9-2
  in the client Smalltalk 9-2
  of GemStone host file access methods 10-13
  profiling 9-4
  tuning 9-2–9-3
explicit stubbing of objects to reclaim space 9-16
extents 1-3

F

False (predefined GemStone object) 10-6, A-3
fast connector 4-6
fault 3-21
fault control
  and replicates 9-14
  and stubs 9-14
fault level
  defined 3-17
  for linked vs. remote sessions 3-19
  performance and 9-14
  specifying with replication specification 3-20
fault policy, defined 3-20
faulting 3-13
  at login 3-17
  changes from other sessions 3-19
  cost of 9-14
  customized 3-27–3-31
  customizing a class 3-21
  default policy for 9-8
  defined 3-13
  dirty GemStone objects 9-3
  immediate 3-20
  inadequate, penalties of 9-14
  lazy 3-20
  minimizing for performance tuning 9-14
  when 3-13
  when a stub receives a message 3-18
  while flushing, error caused by 3-28
faultLevelLnk configuration parameter 3-19, 9-5, 9-8
faultLevelRpc configuration parameter 3-19, 9-5, 9-9
faultToLevel: 3-38
file
  host access 10-13
  writing class and method definitions to 5-19
file in, and error-handling 5-22
filing out classes and methods 5-19
finding objects in repository 10-5
floats, omitting from transparency caches 9-18
flushing 3-14
  and committing, compared 3-14
  customized 3-27–3-31
  defined 3-13
  improving performance of 9-10
  of dirty replicates 9-3
  when 3-14
  while faulting, error caused by 3-28
forwarder 3-9–3-11
  arguments to 3-10
  classes that cannot become 4-4
  converting 3-39
  creating 3-9
  creating interactively 4-16
  debugging 9-9
  declaring in replication specification 3-9
  defined 3-3
  defunct 3-11
  enforcing a return of 3-10
  for optimization 9-17
  return from 3-10
  sending messages to 3-10
  to client 3-9
  to server 3-9
  when to use 3-9
forwarderDebugging configuration parameter 9-5, 9-9
freeSlotsOnStubbing configuration parameter 9-5, 9-9
fwat: 3-10
fwat:ifAbsent: 3-10

G

GbsBuffer 3-27
GbsClassConnector A-2
GbsClassInstVarConnector 4-7
GbsClassVarConnector 4-7
GbsConfiguration 5-28

GbsConnector 4-7, A-2
GbsError A-1, B-2
GbsFastConnector 4-7, A-2
GbsForwarder A-2
GBSM global 2-6
GBSM, instance of GbsSessionManager 2-4, 6-4
GbsNameConnector 4-7, A-2
GbsObject instances 10-5–10-10
GbsObjectTraversal A-1
GbsRuntime B-1
GbsServerClass 3-10
GbsSession 2-4
  reference to parameters 2-9
GbsSessionManager 2-4
GbsSessionParameters 2-4
  instance creation 2-5
GbsSessionParameters class
  instance creation 2-5
GbsTools B-1
Gem
  service name 2-5
  signaling another Gem 6-18
  user process 1-2, 1-3
GemBuilder
  capabilities of iii
  classes added to client Smalltalk image A-1
  overview 1-3
GemBuilder tools
  breakpoint browser 5-26
  Classes Browser 5-4–5-9
  Connector Browser 4-13–4-16
  debugger 5-27
  GemStone inspectors 5-11
  GemStone menu 5-2
  GemStone workspace 5-10
  Login Editor 2-7
  overview iii, 1-6
  Segment Tool 7-10–7-19
  Session Browser 2-7–2-12
  Symbol List Browser 7-20–7-23, ??–7-28
  System Workspace 5-10
  User Account Manager 7-23–??
GemStone
  documentation vi
  kernel classes 10-6, A-3
  SpecialGemStoneObjects 10-6
GemStone Browser 5-4
  Classes menu 5-7
  Classes pane 5-5
  Method Categories pane 5-5
  pop-up menus 5-5
  Symbol List pane 5-5
GemStone error dictionaries, predefined 10-6
GemStone inspector 5-11
GemStone Smalltalk
  comparing with client Smalltalk D-1
  debugger 5-27
  executing in repository 10-8
  faulting upon execution of 3-13
  features of 1-5
  inspecting objects in 5-11
  interrupting 11-4
  when execution occurs 3-13
Gem-to-Gem notifiers 6-18
generateClassConnectors configuration parameter 3-7, 9-6, 9-9
generateGSClasses configuration parameter 3-7, 9-6, 9-9
generateSTClasses configuration parameter

3-7, 9-6, 9-9
global connectors 4-2
Globals dictionary 7-9, 7-10
`gs` prefix, and trap-door message passing to GemStone objects 10-6
GsInterSessionSignal 6-18
`gsObjImpl` 3-31

H

hard break 11-4

I

immediate fault policy 3-20
`indexableSize` 3-31
`indexableValueAt:` 3-30
`indexableValueAt:put:` method 3-28
`indexableValues` 3-29
`indexableValuesBuffer` 3-28
inheritance
  replication specification and 3-22
initial cache size, changing 9-19
InitialCacheSize class variable 9-19
initialCacheSize configuration parameter 9-6, 9-10
initialDirtyPoolSize configuration parameter 9-6, 9-10
initializing
  connectors 4-3
  connectors programmatically 4-8
Inspect 5-7
inspecting
  GemStone objects 5-11
  in a debugger 5-23
inspector 5-11
instance migration 8-2
instance variables
  constraining 5-12
  direct access causing stub errors 3-18, 9-17
  mapping 3-8, 3-14
    in migration 8-4
  mapping nonmatching names 3-16
  maximum number in a Class 5-13
  modifying while faulting 3-27
  modifying while flushing 3-29
  private 5-17
  suppressing replication of 3-15
`instancesAreForwarders` 3-9
`instVarMap` 3-15
internal structure of GemStone objects
  inspecting 5-11
  methods to examine and change 10-11
interrupt-handling 11-4
interrupting execution 11-4
invariant objects 5-13

K

kernel class
  connecting connectors for 4-5
  connecting instances of 4-5
  list of A-3
  SpecialGemStoneObjects and 10-6

L

lazy fault policy 3-20
linked application 2-2
linked session 2-2, 9-10
`listInstances:` (Repository) 8-3
locks 6-10
  logging out, effect of 6-14
  on objects 6-13
  releasing 6-15
  removing 6-14
  setting 6-13
  stubbing and 3-18
locus of execution 9-2

logging into GemStone 2-2
  interactively 2-11
  programmatically 2-9
logging out of GemStone
  effect on locks 6-14
  interactively 2-11
  programmatically 2-10
  to resynchronize application state 6-8
login
  authorization 7-2
  faulting at 3-17
Login Editor 2-7
`login` message 2-9
loginLinkedIfAvailable configuration parameter 2-2, 9-6, 9-10
`logout` message 2-10
lost OT root 6-8

M

managing
  concurrent transactions 6-10
  connectors 4-9
  space, and cache size 9-16
manual transaction mode 6-9
manual, organization of v
mapping 3-6
  automatic 3-8
  behavior 3-8
  class 1-4
  class versions and 3-8
  classes 3-6
  classes with different storage formats 3-31
  instance variables 3-14
  nonmatching names 3-16
  schema coordination 8-6
`markDirty` 3-14
marking dirty
  automatic vs. explicit, performance costs of 3-14
  automatic vs. explicit, reliability of 3-14
  automatically 9-7
maximum number of instance variables in a class 5-13
messages
  faulting when a stub receives 3-18
  sent to GemStone objects 10-6
Method Categories pane in GemStone Browser 5-5
methods
  breakpoints in 5-22
  filing out 5-19
  primitive, and breakpoints 5-25
  private 10-4
  protecting 7-2
  public 5-18
  special, and breakpoints 5-25
  without step points 5-25
migration
  authorization errors and 8-4
  destination 8-3
    ignoring 8-4
  instance variable mapping 8-4
  methods for
    `migrate` (Object) 8-3
    `migrateFrom:instVarMap:` (Object) 8-5
    `migrateInstances:to:` (Object) 8-4
    `migrateTo:` (Object) 8-3
  of instances 8-2
modal dialog, and application responsiveness 6-8
modifying the internal structure of a GemStone object 10-11
monitoring GemStone execution 9-4
move command
  in Browser's Class menu 8-9
moving data into GemStone 4-16
multiprocess applications 9-19

N

name connector 4-4

name of superclass, specifying 5-13
`namedValueAt:` 3-30
`namedValueAt:put:` 3-28
`namedValues` 3-29
`namedValues:indexableValues:` 3-27
`namedValuesBuffer` 3-27, 3-28
network
  node 2-5
  of objects, connecting 3-4
  resource string syntax C-1
`Nil`
  (predefined GemStone object) 10-6, A-3
`noFaultDebugging` message 9-4
nonblocking mode 11-4
notification, Gem-to-Gem 6-18
NRS (network resource string) syntax C-1

O

object
  business 1-8
  domain 1-8
  repository, overview 1-2
object-level invariance 5-13
`OFalse` (predefined object) A-3
`OIllegal`
  (predefined object) A-3
`ONil` (predefined object) A-3
optimization
  and multiprocess applications 9-19
  and traversal buffer size 9-15
  by explicit stubbing 9-16
  by using forwarders 9-17
  changing the initial cache size 9-19
  choosing execution platform 3-2
  choosing the execution platform 9-3
  controlling cache size 9-17
  controlling replication level and 3-17
  controlling the locus of execution 9-2
  controlling the replication level 9-15
  cost of data management 9-3
  explicit stubbing and 3-20
  fault levels and 3-19
  minimizing replication cost 3-14–3-27
  preventing transient stubs 9-15
  reduced-conflict classes and 6-16
  using forwarders 3-9, 9-17
  using GemStone Smalltalk for searching and sorting large objects 9-3
  using GemStone user actions and client Smalltalk primitives 9-18
  watching stub activity 9-4
order in which connectors are connected 4-5
`OTrue` (predefined object) A-3

P

packaging runtime applications B-1
parameter in message to forwarder 3-10
password
  GemStone 2-4, 2-6
  host 2-5

performance 1-9
  automatic vs. explicit dirty-marking 3-14
  changing the initial cache size and 9-19
  choosing execution platform 3-2
  choosing the execution platform and 9-3
  client Smalltalk primitives and 9-18
  controlling cache size and 9-17
  controlling fault level and 9-15
  controlling replication level and 3-17
  controlling the locus of execution and 9-2
  cost of data management and 9-3
  database searching and sorting 9-3
  determining bottlenecks 9-4
  enhancing replication 3-14
  explicit stubbing and 3-20, 9-16
  fault levels and 3-19, 9-14
  forwarders and 9-17
  GemStone Smalltalk user actions and 9-18
  minimizing faulting of dirty GemStone objects 9-14
  minimizing replication cost 3-14–3-27
  multiprocess applications and 9-19
  preventing transient stubs 9-15
  reduced-conflict classes and 6-16
  reducing conflict and 6-16
  traversal buffer size and 9-15
  using forwarders 3-9
  using GemStone Smalltalk for searching and sorting large objects 9-3
  watching stub activity 9-4
pool dictionaries 5-14
pool variables 5-14
pop-up menus
  in GemStone Browser 5-5
postconnect action 4-3
  changing 4-16
  setting programmatically 4-8
  updateGS 4-16
`postFault` 3-28
precedence
  of multiple replication specifications 3-26
  of replication mechanisms 3-38
predefined GemStone objects A-3
`preFault` 3-28
prerequisites iv
preventing transient stubs 9-15
primitives 9-18
  arguments to 3-20
  breakpoints and 5-25
private
  GemBuilder classes 10-4
  GemBuilder methods 10-4
  instance variables 5-17
privileges 7-3
processSafeCaches configuration parameter 9-6
profiling GemStone Smalltalk execution 9-4
ProfMonitor class 9-4
programming interface 1-4
protecting methods 7-2
public methods 5-18

R

RcBag 6-16
RcCounter 6-16
RcKeyValueDictionary 6-16
RcQueue 6-17
  write / write conflicts and 6-17
read lock messages
  `readLock:` (GbsSession) 6-13
  `readLock:ifDenied:ifChanged:` (GbsSession) 6-13
  `readLockAll:` (GbsSession) 6-13
  `readLockAll:ifIncomplete:` (GbsSession) 6-14
read operations 6-11
read set 6-10
read / write transaction conflicts 6-10–6-16

reduced-conflict classes 6-16–6-17
  performance and 6-16
  RcBag 6-16
  RcCounter 6-16
  RcKeyValueDictionary 6-16
  RcQueue 6-17
  storage and 6-16
reducing the number of objects in Smalltalk 9-16
registering a session 2-6
releasing locks 6-15
reliability of automatic vs. explicit dirty-marking 3-14
remote perform messages
  `remotePerform:` (GbsObject) 10-6
  `remotePerform:with:` (GbsObject) 10-6
  `remotePerform:with:with:` (GbsObject) 10-6
  `remotePerform:withArgs:` (GbsObject) 10-6
remote session 2-2
`removeDependent:` 2-12
`removeFromCommitOrAbortReleaseLocksSet:` (System) 6-15
`removeFromCommitReleaseLocksSet:` (System) 6-15
removeInvalidConnectors configuration parameter 9-6, 9-10
`removeLock: (GbsSession)` 6-14
`removeLockAll: (GbsSession)` 6-14
`removeLocksForSession: (GbsSession)` 6-14
`removeParameters` (GbsSession) 2-6
removing
  duplicate connectors 4-2
  locks 6-14
  unresolved connectors 4-15, 9-10
replicate 3-12–3-37
  as argument to primitive method 3-20
  converting 3-39
  customized faulting of 3-21
  defined 3-3
  fault control and 9-14
  flushing dirty 3-14, 9-3
  locking and stubbing 3-18
  preventing stubbing 3-21
  update direction 3-12
  when to use 3-12
replicating
  behavior 3-8
  blocks, avoiding 3-35
  client Smalltalk blocks 3-32, 9-7
  limits of 3-31–3-37
  minimizing costs of 3-14
  precedence of various mechanisms 3-38
  ScaledDecimals 3-36
  suppressing instance variables 3-15
replication specification 3-21–3-27
  class versions and 3-23
  declaring forwarder in 3-9
  inheritance and 3-22
  interactions with automatic class generation 3-7, 3-23
  managing dependencies between objects with 3-25
  precedence 3-26
  root object for 3-26
  specifying fault levels in 3-20
  switching among several 3-24
`replicationSpecSet:` 3-24
repository
  modifying 3-12, 6-10
  overview 1-2
Repository (class) 7-4
reserved OOPs A-3
reserved selectors 5-19
`resolveSymbol:, resolveSymbol:ifAbsent:` 10-5
return value from forwarder 3-10

root objects 3-4–3-6
  in replication specifications 3-26
RPC Gems
  using blocking protocol for 9-7
RPC session 2-2
RT_ERR_SIGNAL_ABORT signal 6-7
runtime applications B-1

S

saving
  class and method definitions 5-19
  login information 2-8
ScaledDecimal replication 3-36
schema
  coordinating 8-6
  matching, and instance variable mapping 3-8
  modification 8-2
    class versions and 8-6
scope of connectors 4-2
security 1-8
  login authorization 7-2
  privileges 7-3
  protecting methods 7-2
Segment class 7-4
Segment Tool 7-10, 7-18
  changing a default segment 7-19
  changing authorization 7-18
  displaying segments 7-11
  examining authorization 7-18
  File menu 7-14
  Group menu 7-16
  Help menu 7-17
  Member menu 7-16
  Report menu 7-17
  Segment menu 7-15
segments
  and authorization 7-6, 7-7
  changing authorization 7-18
  checking authorization 7-18
  group assignment 7-13
  Segment Tool 7-10
selection blocks in GemStone D-2
selector, reserved 5-19
sending messages to GemStone through delegates 10-6
session 2-1–2-15
  control 2-3
    classes for 2-3
  creating linked 2-5
  creating remote 2-5
  current 2-2, 2-9, 2-11
  dependents 2-12–2-15
    adding 2-12
    committing a transaction 2-12
    removing 2-12
  linked 2-2
    fault level 3-19
  logging in 2-2
    interactively 2-11
    programmatically 2-9
  logging out
    interactively 2-11
    programmatically 2-10
  managing connectors for 4-10
  multiple 2-2, 2-9
  persistence of notify set in 6-17
  registering with GBSM 2-6
  remote 2-2
    fault level 3-19
  removing 2-8
  RPC 2-2
  seeing others' changes 3-19
  session-based error-handling 11-2
  signaling between 6-18
  supplying parameters with Login Editor 2-7
  tools attached to current 2-10

Session Browser 2-7–2-12
  opening 2-7
  starting 2-7
session parameters 2-4–2-6
  adding connectors and 4-12
  adding new 2-7
  See also GbsSessionParameters
setting
  configuration parameters 9-6
  locks 6-13
shared dictionaries 7-9, 7-10
shared libraries required for runtime applications B-2
shared variables 5-14
sharing objects
  determining which 1-7, 3-2
  modifications and 6-10
`shouldBeCached` method 9-17
`signaledAbortAction:` (GbsSession) 6-8
signaling one Gem from another 6-18
`SmallInteger` (predefined object) 10-6
Smalltalk
  GemStone, features of 1-5
soft break 11-4
special
  methods, and breakpoints 5-25
  selectors 5-19
special GBSM classes A-1, B-1
SpecialGemStoneObjects dictionary 3-31, 10-6, A-3
stack
  examining in GemStone 5-28
  getting trace without debugger 5-28
stack-based error-handling 11-2
static exceptions 11-2
step points 5-23
  methods without 5-25
stepping 5-22
STN_GEM_ABORT_TIMEOUT GemStone configuration parameter 6-8
Stone
  name of 2-4
  repository monitor 1-2, 1-3
storage, and reduced-conflict classes 6-16
storing data in GemStone 4-16
structural access to GemStone objects 10-11–10-12
stub 3-17–3-21
  as argument to primitive method 3-20
  controlling the stub level 9-15
  converting 3-39
  debugging 9-10
  defined 3-3
  defunct 3-21
  explicit control of 9-16
  explicit creation 3-20
  explicit stubbing 9-16
  fault control and 9-14
  faulting upon message receipt 3-13, 3-18
  instance variable access and 9-17
  observing activity of 9-4
  preventing transient 9-15
  replicating 3-21
  sending messages to 3-17
  setting instance variables to `nil` 9-9
  watching activity of 9-4
stubDebugging configuration parameter 9-6, 9-10
`stubYourself` 3-20, 9-16
subclassing 5-13
superclass, specifying name of 5-13
suspended user interface process 6-8
symbol dictionaries 7-20
Symbol List Browser 7-9, 7-10, 7-20
  copying and pasting objects 7-21
  Dictionaries pane 7-20
  File menu 7-21
Symbol List pane in GemStone Browser 5-5
symbol lists 7-9

synchronizing
  client and GemStone objects 3-12–3-14
    performance costs 3-14
    reliability of automatic vs. explicit 3-14
  shared objects 4-15
SystemRepository, segments in 7-11

T

technical support vii
tools
  attached to current session 2-10
  overview 1-6
transaction 6-1–6-19
  aborting 6-5
  committing 1-4
    and performance 6-16
  committing, and session dependents 2-12
  faulting at boundaries 3-13
  managing 2-12, 6-2, 6-4
  modes 6-8–6-10
    automatic 6-8, 6-9
    automatic, defined 6-8
    manual 6-9
    manual, defined 6-9
    switching between 6-10
transient object stubs, preventing 9-15
transitive closure 3-6
transparency
  and access to GemStone 1-4
  avoiding 3-38
  caches, managing size 9-18
trap-door message passing to GemStone objects 10-6, 10-7
traversal buffer, compressing for transmission 9-11
traversalBufferSize (method) 9-15
traversalBufferSize configuration parameter 9-6, 9-10, 9-15
traversalCompression configuration parameter 9-6, 9-11
`True` (predefined GemStone object) 10-6, A-3

U

`updateRequest:` 2-13
updating
  class definitions 3-6
  replicate 3-12
User Account Manager 7-10, 7-23
  Database User dialog 7-25
  Database Users list 7-23
user actions 9-18
  and primitives 9-18
UserClasses
  client Smalltalk Browser category 3-7
  GemStone Browser symbol dictionary 3-7
user-defined errors 11-3
UserGlobals dictionary 7-9, 7-10
username
  GemStone 2-4
  host 2-5
UserProfile class 7-5

V

variables
  pool 5-14
  shared 5-14
verbose configuration parameter 9-6, 9-11
verifying connectors 4-15, 9-8
versions of classes 8-2, 8-6
  connecting and 3-8
  mapping and 3-8
  replication specifications and 3-23

W

write lock messages
  `writeLock:` (GbsSession) 6-13
  `writeLock:ifDenied:ifChanged:`

(GbsSession) 6-13
`writeLockAll:` (GbsSession) 6-13
`writeLockAll:ifIncomplete:`
(GbsSession) 6-14
write operations 6-11
write set 6-10
write / write transaction conflicts 6-10–6-16
RcQueue and 6-17