GemBuilder™ for Smalltalk/VW User's Guide

Version 8.4

June 2019

GemTalk™ SYSTEMS

INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemTalk Systems LLC assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

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

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemTalk Systems.

This software is provided by GemTalk Systems LLC and contributors "as is" and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemTalk Systems LLC or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

COPYRIGHTS

PATENTS

GemStone software is or has been covered by U.S. Patent Number 6,256,637 "Transactional virtual machine architecture" (1998-2018), Patent Number 6,360,219 "Object queues with concurrent updating" (1998-2018), Patent Number 6,567,905 "Generational garbage collector with persistent object cache" (2001-2021), and Patent Number 6,681,226 "Selective pessimistic locking for a concurrently updateable database" (2001-2021).

TRADEMARKS

GemTalk, **GemStone**, **GemBuilder**, **GemConnect**, and the GemTalk logo are trademarks of GemTalk Systems LLC, or of VMware, Inc., previously of GemStone Systems, Inc., in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Solaris, **Java**, and **Oracle** are trademarks or registered trademarks of Oracle and/or its affiliates. **SPARC** is a registered trademark of SPARC International, Inc.

Intel and **Pentium** are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, **Windows**, and **Windows Server** are registered trademarks of Microsoft Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds and others.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

Ubuntu is a registered trademark of Canonical Ltd., Inc., in the U.S. and other countries.

SUSE is a registered trademark of Novell, Inc. in the United States and other countries.

AIX, **POWER6**, **POWER7**, and **POWER8** and **VisualAge** are trademarks or registered trademarks of International Business Machines Corporation.

Apple, **Mac**, **MacOS**, and **Macintosh** are trademarks of Apple Inc., in the United States and other countries.

CINCOM, **Cincom Smalltalk**, and **VisualWorks** are trademarks or registered trademarks of Cincom Systems, Inc.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. GemTalk Systems cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemTalk Systems LLC
15220 NW Greenbrier Parkway
Suite 240
Beaverton, OR 97006

Preface

About This Manual

This manual describes GemBuilder™ for Smalltalk, an environment for developing GemStone™ applications using Cincom VisualWorks Smalltalk.

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

The programming interface provides facilities for managing the relationship between objects on the GemStone server and in client Smalltalk, allowing objects to be available on the client and updated on the shared GemStone server.

The GemBuilder programming environment provides a set of integrated tools for programming in GemStone's version of Smalltalk.

GemBuilder supports multiple versions of the GemStone server, and there are minor differences between the features that will be available. Please consult the documentation for the server product and version you are using for specific details on that product.

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/S 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 VisualWorks Smalltalk language and programming environment as described in the VisualWorks Smalltalk product manuals.

Finally, you should have the GemStone system installed correctly on your host computer, as described in the *GemStone/S 64 Bit Installation Guide* for your platform, and have client Smalltalk and GemBuilder for Smalltalk installed on the client computer, as described in the *GemBuilder for Smalltalk Installation Guide*.

Terminology Conventions

The term "GemStone" is used to refer to the server products GemStone/S 64 Bit and GemStone/S; the GemStone Smalltalk programming language; and may also be used to refer to the company, previously GemStone Systems, Inc., now a division of VMware, Inc.

Likewise, GemStone/S may refer to either the 32-bit GemStone/S product, or to the GemStone/S 64 Bit product.

GemStone Server Documentation

GemBuilder for Smalltalk provides an interface to the GemStone/S server. To get full advantage it is helpful to understand the special features of the GemStone Smalltalk language, and how the GemStone server operates.

The documentation for GemStone/S can be found on the GemTalk website, at https://gemtalksystems.com/techsupport/resources.

Technical Support

Website

http://gemtalksystems.com

GemTalk's website provides a variety of resources to help you use GemStone products:

- **Documentation** for the current and for previous released versions of all GemTalk products, in PDF form.
- **Product download** for the current and selected recent versions of GemTalk software.
- **Bugnotes**, identifying performance issues or error conditions that you may encounter when using a GemTalk product.
- **Supplemental Documentation** and **TechTips**, providing information and instructions that are not in the regular documentation.
- **Compatibility matrices**, listing supported platforms for GemTalk product versions.

We recommend checking this site on a regular basis for the latest updates.

Help Requests

GemTalk Technical Support is limited to customers with current support contracts. Requests for technical assistance may be submitted online (including by email), or by telephone. We recommend you use telephone contact only for urgent requests that require immediate evaluation, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: techsupport.gemtalksystems.com

Email: techsupport@gemtalksystems.com

Telephone: (800) 243-4772 or (503) 766-4702

Please include the following, in addition to a description of the issue:

- The versions of GBS and of all related GemTalk products, and of any other related products, such as client Smalltalk products, and the operating system and version you are using.
- Exact error message received, if any, including log files and statmonitor data if appropriate.

Technical Support is available from 8am to 5pm Pacific Time, Monday through Friday, excluding GemTalk holidays.

24x7 Emergency Technical Support

GemTalk 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, for issues impacting a production system. For more details, contact GemTalk Support Renewals.

Training and Consulting

GemTalk Professional Services provide consulting to help you succeed with GemStone products. Training for GemTalk products is available at your location, and training courses are offered periodically at our offices in Beaverton, Oregon. Contact GemTalk Professional Services for more details or to obtain consulting services.

Table of Contents

Chapter 1. Basic Concepts

Chapter 2. Communicating with the GemStone Object Server

Chapter 3. Sharing Objects

Chapter 4. Connectors

Chapter 5. Managing Transactions

Chapter 6. Security and Object Access

Chapter 7. Exception Handling

Chapter 8. Schema Modification and Coordination

Chapter 9. Performance Tuning

Chapter 10. GemBuilder Configuration Parameters

Chapter 11. The GemStone Tools: an Overview

Chapter 12. Using the GemStone Programming Tools

Chapter 13. Inspecting and Debugging in GemStone

Chapter 14. Using the GemStone Administration Tools

Appendix A. Application Deployment

Appendix B. Client Smalltalk and GemStone Smalltalk

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 client/server application with GemBuilder.

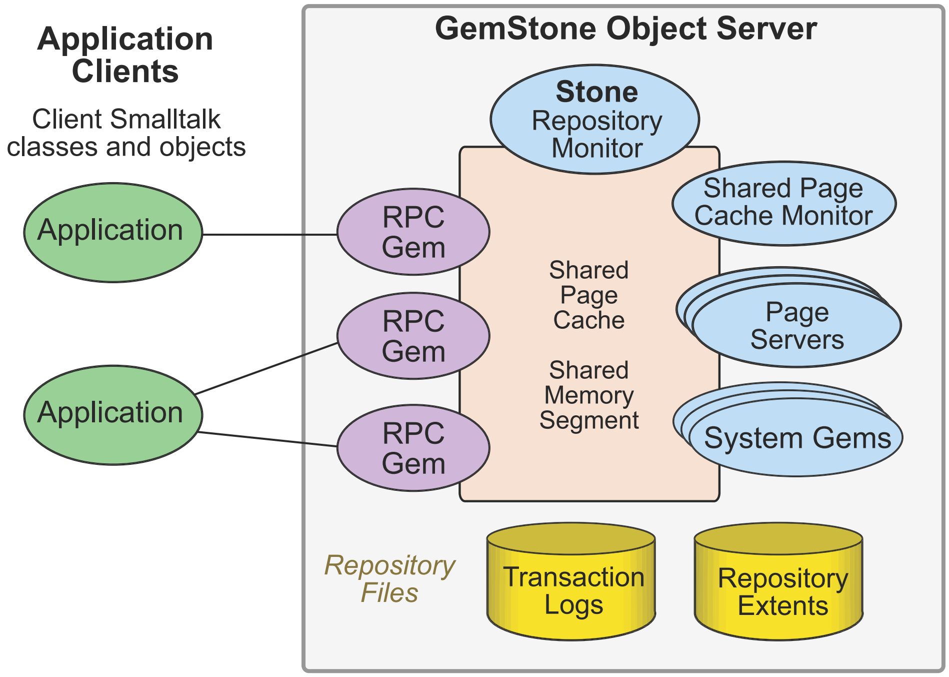
1.1 The GemStone Object Server

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

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

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

Figure 1.1 The GemStone Object Server



The *object repository* is a multiuser, disk-based Smalltalk image containing shared application objects and GemStone kernel classes. It is composed of files (known to GemStone as *extents*) that can reside on a single machine or can be distributed among several networked hosts. The repository can also include GemConnect objects representing data stored in third-party relational databases.

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

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

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

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

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

1.2 GemBuilder for Smalltalk

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

- store your client Smalltalk application objects in the GemStone server;
- import GemStone objects into client Smalltalk as client Smalltalk objects;
- allow your application objects to be transparently replicated and maintained both in the client and in the server, or allow some objects to reside only in the server but be accessible on the client;
- arrange for messages sent to client Smalltalk objects to be forwarded and executed in the GemStone server by corresponding server objects;
- use GemStone's programming tools to develop GemStone server 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 one or more Gem process to serve your application.

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

Only server objects can become persistent in the GemStone object repository. To make a client Smalltalk object persistent, it must be mapped to a server object. This mapping is a relationship between a client object and a server object, whereby each represents the other. Once objects are mapped, GemBuilder maintains the relationship between the shared GemStone server object and the private client Smalltalk object, updating values from the repository to your application and vice versa, as necessary, as well as forwarding messages between the objects. Chapter 3 describes the replication of state and forwarding of messages between client and server objects.

Your client Smalltalk application updates shared objects in the repository by sending a `commitTransaction` 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 the GemStone server, 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 server objects.

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 *GemStone Programming Guide*.

The most significant difference between GemStone Smalltalk and client 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 B.

The GemBuilder Tools

GemBuilder’s programming environment provides tools for programming in GemStone Smalltalk. The following tools are described in detail in Part 2 of this manual:

- The *GemStone Launcher* provides access to the various browsers, and allows you to create and manage sessions and transactions.
- A GemStone *System Browser* and related browsers let you examine, modify, and create GemStone classes and methods.
- The *System Workspace* provides easy access to commonly used GemStone Smalltalk expressions.
- GemStone *Inspectors* let you examine and modify the state of GemStone objects.
- The *Breakpoint Browser* and a *Debugger* let you examine and dynamically modify the state of a running GemStone application.
- The *Connector Browser* allows you to manage the connectors that establish relationships between client Smalltalk and GemStone server objects.
- A *Class Version Browser* can be used for examining the history of a GemStone class, migrating instances, and deleting old versions.
- A *User List*, *User Editor*, and *Privileges Dialog* allow you to create new user accounts, change account passwords, and assign group membership.
- A *Security Policy Browser* facilitates managing GemStone authorization at the object level by controlling how objects are governed by security policies.

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 between 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 the GemStone server 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 the server. 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 transient session objects that no one else will ever need on the server; such objects can remain in the client. 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 client Smalltalk objects; they don't need to be stored in GemStone. However, you might want the classes and methods used to build the report to be stored in GemStone so that others can use them.

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

Which objects should be secured?

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

Which objects should be connected?

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

A connector connects not only the immediate object but also all those objects that it references, so you don’t need to define a connector for every object in your application that you want to store in the GemStone server. 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 5.

How can performance be improved?

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

Network speed can have a singificant impact. Reducing the number of round trips to the GemStone server is a key way to improve performance. This can be done by moving some of the behavior from the client to the GemStone server and let the GemStone Smalltalk execution engine do the work.

Which methods might best be executed on the GemStone server? 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 other Smalltalks that the porting effort is likely to be relatively simple. Performance issues in general are discussed in Chapter 9. Moving execution to the GemStone server is discussed in the section entitled “Locus of Execution” on page 112.

Finally, you can configure the GemStone object server for maximum performance, given the details of your application and environment. GemStone server configuration parameters are discussed in detail in the *System Administration Guide* for GemStone/S 64 Bit. In addition, the *Programming Guide* for GemStone/S 64 Bit gives a variety of tips in the chapter entitled “Tuning Performance.”

Public and Private Classes and Methods

GemBuilder adds many classes and methods to your client Smalltalk image. Some of these are public, which means that they are designed to be referenced directly from your applications. GemStone avoids changing public classes and methods from release to release. Many GemBuilder classes and methods are considered private; they are used to implement the internal workings of GemBuilder and are not designed to be referenced directly from applications. Avoid using private classes and methods; they may have undocumented side effects, and they are subject to change from release to release.

A GemBuilder class is private if its name begins with the prefix `Gbx`.All methods on private GemBuilder classes are considered private, unless the class comment indicates otherwise.

A GemBuilder method on a public class can be marked private in any of several ways:

- The selectors 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 in a method category marked "private".

In general, we encourage you to use in your applications only GemBuilder classes and methods that are documented in this User's Guide, which describes the preferred way to accomplish tasks. Other public classes or methods may be obsolete but kept for backward compatibility.

Reserved prefix

In your code, do not define methods starting with "gb". Methods with this prefix are reserved for GBS.

Likewise, do not use process environment keys starting with "gb".

Deprecated features

GemBuilder may include features that are deprecated in the current release. This means that although the feature is still functional, it may be removed in a future release. In your applications, you should eliminate any dependencies on deprecated GemBuilder features.

You can find all deprecated features of GemBuilder by browsing the senders of `#deprecated:`.

To help you determine whether your application uses any deprecated features, GemBuilder by default raises a `GemStone.Gbs.GbsInterfaceError` whenever a deprecated feature is used. The description of the `GbsInterfaceError` gives information about the deprecated feature that was being used.

If your application does not have an active handler for GbsInterfaceError when a deprecated feature is used, a walkback dialog will open. Once you've noted information about your application's use of the deprecated feature, you may proceed from the walkback or debugger, and your application will continue. This is a useful technique for a developer testing for use of deprecated features.

Once you've noted a particular use of a deprecated feature, you can avoid further walkbacks from that use by adding a handler for the `GbsInterfaceError` that resumes the exception, at the point in your application that the deprecated feature is used.

You may also set the GemBuilder configuration parameter `deprecationWarnings` to false, as described in Chapter 10. When `deprecationWarnings` is false, any code in the client image may use any deprecated GemBuilder feature without raising the error.

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.

Client Libraries
explains how to setup to use the correct client shared libraries.

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.

Logging In to and Logging Out of GemStone
describes how to log in and out of GemStone sessions programmatically.

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

2.1 Client Libraries

Before you can log in to a GemStone object server, in addition to having GBS loaded in your image, you must have the client libraries available for loading into your image. The client libraries are provided as part of the GemStone object server product release distribution. You must use the correct client libraries for the particular version of the object server you wish to connect to, and for the platform that the client Smalltalk image is running on. If you update to a new version of GBS or VisualWorks on the same client platform, but continue to use the same version of the GemStone server, the same client libraries will be used.

There are a number of options for where to put your client libraries to they can be accessed by GBS, and options on how to set the library names in GBS. These options, as well as the correct client library name to use for your GemStone/S server product and version, are described in the *GemBuilder for Smalltalk Installation Guide*.

The current client library name is stored in the GbsConfiguration setting "libraryName". This can be viewed or set using the Settings dialog, or you may execute:

```
GbsConfiguration current libraryName
```

```
GbsConfiguration current libraryName: 'libraryName'
```

For more information on this setting, see Chapter 10, "GemBuilder Configuration Parameters".

2.2 GemStone Sessions

An application connects to the GemStone object server by logging in to the server 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. Most applications use a single session per client; but an application can create multiple sessions from the same client, one of which is the *current session* at any given time. You can manage GemStone sessions either through your application code or through the GemStone Launcher.

RPC and Linked Sessions

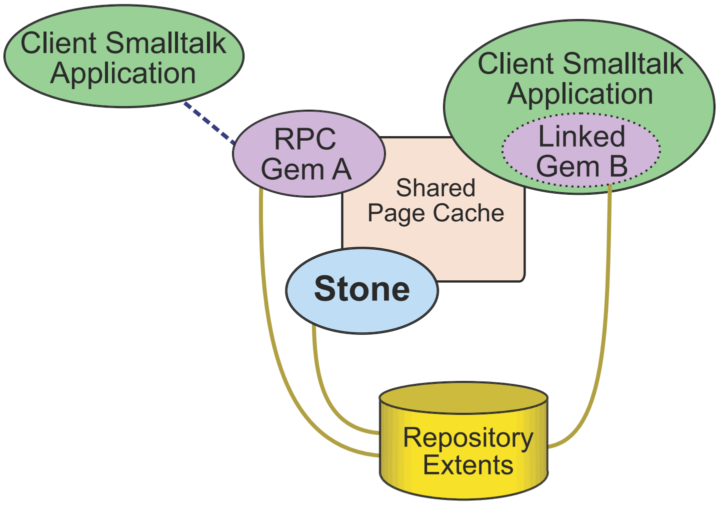
A Gem can run as a separate operating system process and respond to Remote Procedure Calls (RPCs) from its client, in which case the session it supports is called an *RPC* session.

On platforms that host the GemStone object server and its runtime libraries, one Gem can be integrated with the application into a single operating system 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 client 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 load the RPC-only version of the shared libraries.

Figure 2.1 shows an application with two logged-in sessions. Gem A is an RPC session running as a separate process, while Gem B is a linked session, sharing the client Smalltalk application's process space.

Figure 2.1 RPC and Linked Gem Processes



2.3 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 access named server 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 GbsSessions logged in from this client, 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 server connections before the client Smalltalk quits.

Session Parameters

To initiate a GemStone session, you must first provide the details of the object server (repository) that you want to connect to. 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 interfaces are described in later sections.

You may need to contact your GemStone server administrator for the specific details to supply. In distributed configurations, you will need to specify some information in GemStone's Network Resource String (NRS) format. NRS format is described in the *System Administration Guide*, which also describes NetLDI security requirements.

The information needed includes the following:

- **The specification of the GemStone repository**
  In its simplest form, the name of the Stone repository monitor. In a distributed configuration with non-default names, you may need to include the server's hostname and/or NetLDI name or port in NRS format.

  For example, for a Stone named "gs64stone" on a host named "pelican", where the NetLDI is listening on port 54321, specify an NRS string of the form:

  ```
  !@pelican#netldi:54321!gs64stone
  ```

- **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 190.) 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** (if required to start a Gem)
  The host user name and password, which must specify a valid UNIX login account for the Gem's host machine (the machine specified in the Gem service, described below).

  These parameters are not needed for a linked session, since no separate Gem process is started up, nor if the NetLDI is running in guest mode.

- **The specification for the Gem service** (for RPC logins only)
  In its simplest form, the Gem service is `gemnetobject`. In a distributed configuration with non-default names, you may need to include the server's hostname and/or NetLDI name or port, in NRS format. For example:

  ```
  !@pelican#netldi:54321!gemnetobject
  ```

  The gem service may include additional command-line arguments that are passed to gemnetobject, as described in the *System Administration Guide*.

  You should not specify a Gem Service if you are starting a linked Gem process, since no separate Gem process is started up.

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: aStoneSpecification
   username: aUsername
   password: aPassword
   hostUsername: aHostUsername
   hostPassword: aHostPassword
   gemService: aGemServiceSpecification
```

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

```
GbsSessionParameters newWithGemStoneName: aStoneSpecification
   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 GemStone Launcher 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: '!@pelican!gs64stone'
   username:            'DataCurator'
   password:            ''
   hostUsername:        'lisam'
   hostPassword:        ''
   gemService:          '!@pelican!gemnetobject'
```

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.

2.4 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 information) process running.

Depending on the terms of your GemStone license, you can have many sessions logged in at once from the same GemBuilder client. These sessions can all be attached to the same GemStone repository, or they can be attached to different repositories, provided they are all the same version of the GemStone server.

Logging In to GemStone

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:

```
mySession := 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:

```
mySession := 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 Session Parameters 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:

```
myGbsSessionParameters := 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 current session, or 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.

You can send a message directly to any logged-in GbsSession, even when it is not the current session. If you send a message to another session that executes code, that code is evaluated in the receiving session, regardless of whether it is the current session.

Most applications have only one session logged in at a time. In this case, that session will always be the current session, and it is safe to send messages to GBSM for forwarding to the session.

However, if your application concurrently logs in more than one session, your application should send messages directly to each session. If your application client uses multiple Smalltalk processes it is very difficult to accurately coordinate the changing of 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*`.`

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. Executing a **GS-Do it** in a workspace will execute in the current session.

Logging Out of GemStone

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 the GbsConfiguration setting confirm is true (it is true by default). 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 development environment, for example, you can commit by clicking on a button in the GemStone Launcher. 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: [ 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.

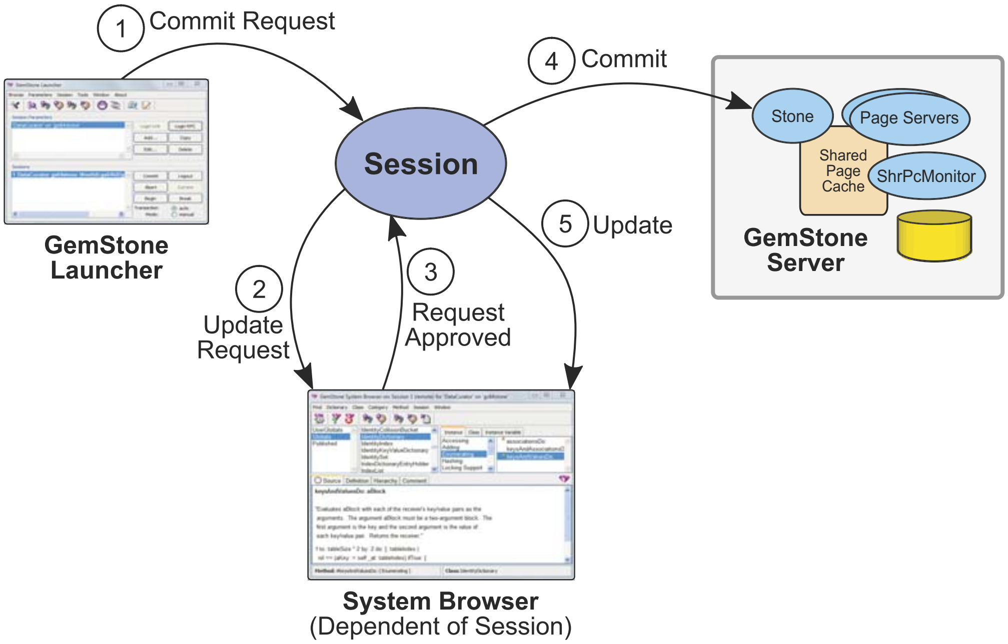
Each registered dependent should be prepared to receive an `update:` message with one of the above aspect symbols as its argument. Example 2.4 shows how a session dependent might implement an `update:` method.

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 2.2 summarizes the sequence of events that occurs when a session queries a dependent before committing. In the figure, the GemStone Launcher 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: #committed` to each (5).

Figure 2.2 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, and introducing forwarders, replicates, and stubs.

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 server 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.

Precedence of Replication Controls
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 virtual machine—acting on one object space—your image. With 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.

You have two basic decisions in making use of the additional abilities provided by the GemStone server:

- Which object 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 resides 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 root objects across the two object spaces.

Three kinds of objects help a GemBuilder client and a GemStone server repository share state and execution: forwarders, stubs, and replicates.

Forwarder—is a *proxy:* a simple object that knows only which object in the other space 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. Forwarders can be on the client, for server master objects, or on the server for client master objects.

Replicate—is an object associated with a particular object in the other object space. The replicate copies some or all of the other object's state, which it synchronizes at appropriate times. It implements all messages it expects to receive. By default, the replicate executes locally. However, you can use `performOnGsServer:` to forward a message to the server.

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

Connector—associates a root client object with a root server object, typically resolving objects by name, although there are other ways. When connected, they synchronize data or pass messages in either direction or take no action at all, as specified. For more information on connectors, see Chapter 4.

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.

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 during the initial connection.

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

- both client and server must define the class, and
- the two classes must be mapped by name or by using a *class connector*.

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

Unlike connectors for replicates or forwarders, class connectors by default do not update at connect time. 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 71.

If there is no connector for a class, and a mapping for that class is required, GemBuilder will attempt to map the client and server classes with the same name. By default, it will also create a connector for those classes. If the configuration parameter generateClassConnectors is false, GemBuilder will still map the classes by name, but will not create a connector. The difference is that without a connector, the mapping only lasts until the session logs out, and any other sessions logged in will not have that mapping. If a connector is created, it is associated with the session parameters object, and any session logged in using that session parameters object will have that class mapping created at login time.

Automatic Class Generation and Mapping

You can configure GemBuilder to generate class definitions and connectors automatically. When so configured, if GemBuilder requires the GemStone server to replicate an instance of a client class that is not already defined on the server, then at the first access, GemBuilder generates a server 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 that is 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 **`generateServerClasses`** and **`generateClientClasses`** (described starting on page 126). These settings are global to your image.

- If you have disabled automatic generation of GemStone classes by setting **`generateServerClasses`** to `false` (the default), situations that would otherwise generate a server class instead raise the exception `GbsClassGenerationError`.
- If you have disabled automatic generation of client Smalltalk classes by setting **`generateClientClasses`** to `false` (the default), situations that would otherwise generate a client Smalltalk class instead raise the exception `GbsClassGenerationError`.
- You can disable class connector generation by setting **`generateClassConnectors`** to `false`. When classes are generated or mapped by name, no connector is generated.

GemBuilder deposits automatically generated GemStone server classes in the GemStone symbol dictionary UserClasses, which it creates if necessary. Automatically generated client Smalltalk classes are deposited in the current package or parcel.

Automatic class generation is primarily useful as a development-time convenience. In an application runtime environment, we recommend having all necessary classes predefined in both object spaces, and having a connector defined for each class before logging in. This can improve performance by avoiding unnecessary work when the class is first accessed.

Schema Mapping

By default, when you map a client class to a GemStone server class, 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

When GemBuilder generates classes automatically, it only copies the definition of the class, not the methods of the class.

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 tools to implement the same 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 chapter entitled "Class Versions and Instance Migration" in the *GemStone Programming Guide*.)

If you change a class definition on the client or server, and 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 server class regenerates the client class and recompiles its methods.
- Updating a GemStone server class from a client Smalltalk class creates a new 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 are not affected by a connector connected to another class version.*
>
> *Migration can affect this issue. See Chapter 8, "Schema Modification and Coordination".*

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, generally resident on the GemStone server, 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, an instance of GbsFowarder, that knows only which GemStone server object it represents. It responds to all messages by passing them to the appropriate server object, where its associated state 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 server object that knows only which client Smalltalk object it represents. It responds to all messages by passing them to its associated client Smalltalk object, where state resides and behavior is implemented.

You can create forwarders in several ways:

- Create a connector with a postconnect action of #forwarder or #clientForwarder. For example, connect the server 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 on the client as a forwarder to the server (using a replication specification, discussed starting on page 47). For example, a client class might implement a specification that declares the instance variable `inventory` as a forwarder to the server.
- Prefix `fw` to a method name to return a forwarder from any message-send to the server. For example, to return a forwarder from a GemStone server name lookup, send the GbsSession `fwat:` or `fwat:ifAbsent:` instead of `at:` or `at:ifAbsent:`.
- Override all these by implementing a class method `instancesAreForwarders` on the client class to return `true`, and all instances of that class and its subclasses will be 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 on the server and return a forwarder to that instance.

Sending Messages

On the client, when a forwarder to the server receives a message, it sends the message to its counterpart on the GemStone server—presumably an instance that can respond meaningfully. The target server object's response is then returned to the forwarder on the client, which then returns the result.

When a forwarder to the client receives a message on the server, it forwards the message to the full-fledged client object to which it is connected. This object's response is returned to the client forwarder, which returns the result represented as a server object.

Arguments

Before a message is forwarded to the GemStone server, arguments are translated to server objects. As a message is forwarded to the client, its 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 BlockClosures" on page 55.

Results

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

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

To ensure a forwarder result, prefix 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 server 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:

```
conn := GbsNameConnector
      clientName: #BigDictionary
      serverName: #BigDictionary.
conn beForwarderOnConnect.
GBSM addGlobalConnector: conn
```

When a GemBuilder session logs into the GemStone server, BigDictionary becomes a valid forwarder to the current server BigDictionary. But when no session is logged into the server, 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 applies only to session-based name, class variable, or class instance variable connectors that have a postconnect action of `#updateST` or `#forwarder` (see "Connectors" on page 65). 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 server object that the replicate represents. Unlike a forwarder, replicates also hold (some) state and implement (some) behavior. Replicates synchronize their state with that of their associated server object.

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: each replicate must be an instance of a class whose definition is mapped to the definition of the corresponding class in the server object space. GemBuilder handles many obvious cases automatically, but nonstandard mappings require you to implement certain instance and class methods. Nonstandard mappings are discussed starting on page 42.

Synchronizing State

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

When an object changes in the server, GemBuilder automatically updates the corresponding client Smalltalk replicate. By default, GemBuilder also detects changes to client Smalltalk replicates and automatically updates the corresponding server object.

The stages and terminology of this synchronization are as follows:

- When an object is modified in the client, leaving its server counterpart out of date, the client object is now referred to as *dirty*.
- When the state of dirty client objects is transferred to their corresponding server objects, this is called *flushing*.
- When a server object is modified in the server, leaving its client counterpart out of date, the server object is now *dirty*. This can occur during execution of server Smalltalk, or at a transaction boundary when changes committed by other sessions become visible to your session.
- When the state of dirty server objects is transferred to their corresponding client objects, this is called *faulting*.

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

Faulting

GemBuilder faults objects automatically when required. Faulting is required when a stub receives a message, requesting it to turn itself into a replicate. (see stubbing on page 43)

Faulting may also be required 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. Faulting may or may not occur upon connection, depending on the post-connect action specified for the connector.
- A server object that has been replicated to the client is modified on the server. This can happen in two cases:
    1. GemStone Smalltalk execution in your session modifies the state of the object. GemStone Smalltalk execution occurs when a forwarder receives a message, or in response to any variant of GbsSession >> evaluate:.
    2. Your session starts, commits, aborts, or continues a transaction—passes a transaction boundary—which refreshes your session's private view of the repository. If the server object has been changed by some other concurrent session, and that change was committed, the object's new state will be visible when your session refreshes its view.

  In both of these cases, the replicate's state is now out of date, and cannot be used until updated by faulting. Depending on the replicate's faultPolicy (see page 46) the new state will either be faulted immediately, or the replicate becomes a stub, and will be faulted the next time it receives a message.

Flushing

GemBuilder flushes dirty client objects to the GemStone server 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 part of 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.

For GemBuilder to flush a changed object to the server, that object must be **marked dirty**, that is, GemBuilder must be made aware that the object has changed. Objects are, by default, marked dirty automatically. In addition, you can explicitly mark objects dirty.

Marking Modified Objects Dirty Automatically

By default, GemBuilder uses VisualWorks automatic mark dirty capability. This detects modifications to replicates on the client so that modified replicates can be automatically marked dirty. This mechanism is fast, reliable, and does not affect client objects that are not replicates. Thus, we recommend always using automatic dirty-marking. Automatic dirty-marking is enabled by default.

To disable automatic dirty-marking, execute:

```
GbsConfiguration current autoMarkDirty: false
```

or use the Settings Tool to turn off the configuration parameter `autoMarkDirty`. It is enabled or disabled globally for the client; you cannot enable automatic dirty-marking for only some classes or objects in the client virtual machine. If you disable automatic dirty-marking, your application must manually mark modified client replicates dirty as described in the next section.

Marking Modified Objects Dirty Manually

Generally, we recommend you use the automatic mechanisms. You can instead, if you wish, mark objects dirty explicitly in your code. The automatic mechanism is faster and much more reliable—if you miss even one place where a shared object is modified, your application will misbehave.

To manually mark a replicate dirty, send `markDirty` to the replicate immediately after each time your application modifies it. If a replicate is modified on the client but not marked dirty, the modification will be lost eventually. The object could be overwritten with its GemStone server state after the application has executed code on the server, or at the next transaction boundary. Even if the client object is never overwritten, the modification will never be sent to the server.

Minimizing Replication Cost

Replicating the full state of a large and complex object graph 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 in the other object space. 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—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 35, 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 the copying of certain instance variables, or by explicitly specifying a mapping between nonmatching names.

Suppressing Instance Variables

Some client Smalltalk objects 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 visible to the GemStone server. Situations can also arise in which the server 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 server counterpart, the value of that variable is not replicated in the server. When GemBuilder faults the server object into the client, the client's suppressed instance variable remains unchanged.
- Likewise, if a server 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 GemBuilder flushes the object into the server, the server object's suppressed instance variable remains unchanged.

You can also suppress instance variable mappings by implementing the client class method `instVarMap`. For example:

```
TestObject class>>instVarMap
      ^super instVarMap ,
            #(    (nil serverName)
                  (clientName 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 serverName`) specifies that the server instance variable `serverName` will never be replicated in the client. The second pair (`clientName nil`) specifies that the client instance variable `clientName` will never be replicated in the server.

Nonmatching Names

You can also specify an explicit instance variable mapping between the server 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 mappings from one instance variable name to another, specify each name in the mapping array. For example:

```
TestObject class>>instVarMap
      ^super instVarMap ,
            #(    (clientName  serverName) )
```

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

One implementation can both prune irrelevancy and accommodate differing schemas. for example,

```
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 server-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 replicated in the server.

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

Stubbing

Often 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 the GemStone server can. 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 server object. Unlike a forwarder, however, when a stub receives a message, it does not send the message across to the other object space. Instead, it faults its server counterpart into the client image. The client Smalltalk replicate then responds to the message.

When GemBuilder faults automatically, it replicates the object hierarchy to a certain level, then creates stubs for objects on the next level deeper than that. The number of levels that are replicated each time is the *fault level.*

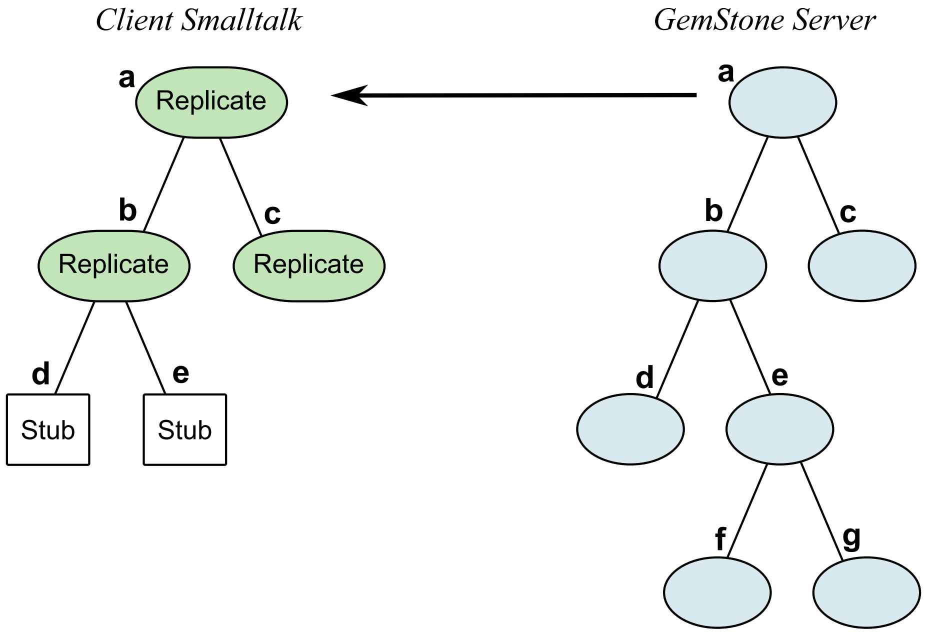
The fault level of 1 follows an object's immediate references and faults those in; the fault level of 2 follows one more layer of references and replicates those objects, too.

Figure 3.1 illustrates an application with where object **a** has a fault level of 1.

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 3.1 Two-level Fault of an Object



Faulting in Response to a Message

When object **e**, which is a stub, receives a message, it faults in a replicate of its counterpart GemStone server 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:

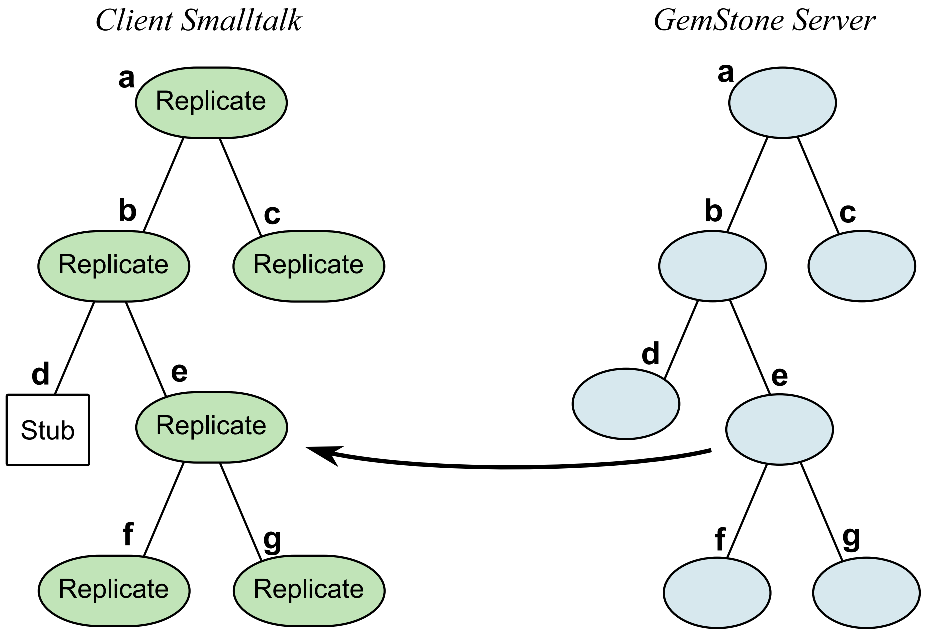
```
demonstrateProblem
   | firstTemp secondTemp |
   firstTemp := size.  "Size is an inst var of the receiver.
   FirstTemp now has a valid value."
   self stubYourself.  "self is now a stub, and has no
                        instance variable values"
   secondTemp := size. "Since this access is not a message
                        send, it does not unstub self.
                        SecondTemp now contains an invalid
                        value, most likely nil."
   ^Array with: firstTemp with: secondTemp.
```

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

```
self size. "This is a message, and faults the stub."
```

e is now a replicate, as shown in Figure 3.2. The new replicate responds to the message.

Figure 3.2 A Stub Responds to a Message



Again, two levels are replicated, object **e** and its immediate instance variable: a fault level applies to each instance of that class.

Faulting in Changes From Other Sessions

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

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 client 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.

The way in which these changes depends on the server product, and with 32-Bit GemStone/S, on a GemBuilder configuration setting.

GemStone/S 64 Bit

Because network overhead is minimal, all objects are immediate-faulted. When GemBuilder detects a change in a repository object, it updates the replicate immediately.

GemStone/S (32 bit)

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 may be able to access the gem with lower latency, 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 47.

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 114.

Sometimes stubbing is not desirable, either for performance reasons or for correctness. For example, primitives cannot accept stubs as arguments if the primitive accesses the instance variables of the argument. 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 is 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.

Defunct Stubs

Faulting in a stub relies on the existence of a valid GemStone server object to replicate or forward to. If an object is stubbed and the session logs out, a message to that stub raises an error complaining that it is *defunct.* For example, suppose MyGlobal is modified in a 32-bit server, 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 client 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 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 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:

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

To ensure that replication specs established in superclasses remain in effect, Example 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 `replicate` unless to restore behavior overridden in a superclass.

```
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) )
```

Replication Specifications and Class Versions

As explained in "Mapping and Class Versions" on page 36, client Smalltalk classes connect not simply to GemStone Smalltalk classes, but to specific server class versions. A class connector connects to only one server class 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 the server until its class history lists three versions. Your client Smalltalk class is connected to Version 2. Class X's replication spec will affect server instances of Class X, Version 2. If the server 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. The selector of the replicationSpec acts as the named of the replication spec. Using this message to specify the replicationSpec to use to perform a particular operation, 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, specifying the selector symbol of the new replicationSpec method.

Step 5. Immediately after the operation completes, send `replicationSpecSet:` *#replicationSpec* to the GbsSession to restore replication. If the session could be addressed from more than one client Smalltalk process, your application should use a semaphore to control access to the session.

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 GemBuilder replicates an object, it also replicates those objects it depends on.

Hashed collection classes that wish to replicate instances between client and server should answer true to the message `#gbsMustDeferElements`. This is the recommended approach.

When an object whose class answers true to `#gbsMustDeferElements` is faulted to the client, the elements are not added to the collection until the replication of those elements is complete. This ensures that all of the information necessary to compute the hash of the element is present before adding it to the collection; if added earlier, its hash might change as its replication continued, corrupting the collection.

There is one exception to this requirement. Hashed collections that compute hash purely on the identity hash of their elements may answer false to `#gbsMustDeferElements`, since their hash values are computed strictly on the identity of the elements themselves, which is always present.

NOTE

If you do not use `#gbsMustDeferElements` *(the recommended approach), you must independently address the issues described in the following paragraphs.*

For example, in order to create a Dictionary when replicating it from the server, 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 the server as they are in the client). So, if GemStone replicates a Dictionary, it must also at a minimum replicate the association and the key in the association, so it can compute the hash. 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 replicated whenever one of these dictionaries is replicated, 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, so as not 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 replicated. 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 encountered: 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.

Forwarding Messages to Server Objects Through Replicates and Stubs

Most messages received by a client replicate execute their behavior locally on the client. However, it is possible to make a replicate or a stub forward a message to its server counterpart, somewhat like a forwarder does. This is done with the following message:

`performOnGsServer:` *selector* `withArguments:` *argumentArray*

For messages with no arguments, you may use

`performOnGsServer:` *selector*

The server object will be sent a message with the given selector and arguments, and the result will be replicated to the client. This gives you a great deal of flexibility as to which behaviors are executed on the server and which on the client.

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 the server 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 server.

Modifying Instance Variables During Faulting

You can customize object retrieval by using buffers for the client counterparts of GemStone server 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 "The current server didn't 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:

```
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 the server 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 the server.

`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 the server.

Implementations of `namedValues` must return an object with the appropriate number of named instance variable slots. In Example 3.1, a clone of the positionable stream is returned that increments the `position` instance variable by 1 as needed when mapped into the server:

Example 3.1

```
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 3.2, for class Set, an Array is returned, because the indexable fields of a Smalltalk set are a sparse list of the actual elements.

Example 3.2

```
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 server. 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 collections with instance variables can present special problems, discussed below.

Replicating Client Smalltalk BlockClosures

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 58.

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.

Two workarounds are possible:

1. Leave the compiler and decompiler in the image. For example, the VisualWorks Image Maker tool offers a "Remove Compiler" option which you can deselect to leave the compiler and decompiler in the image.
2. 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)
         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 a 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 ensure 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 Collections with Instance Variables

If you create a subclass of a Collection and give it instance variables, you must reimplement the `copyEmpty:` method to ensure that added instance variables are included in the copy operation. Failure to reimplement `copyEmpty:` results in data loss.

For example, consider a Collection subclass called MyCollection that defines the additional instance variable `name`, with methods `name` and `name:` that retrieve and assign its value, respectively. MyCollection might reimplement `copyEmpty:` like this:

```
MyCollection >> copyEmpty: size
^(super copyEmpty: size) name: name
```

This reimplementation of `copyEmpty:` preserves the copying behavior of the superclass and assures that the added instance variable is also copied.

3.5 Precedence of Replication Controls

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

1. If the class methods `instVarMap` (for replicates) or `instancesAreForwarders` (for forwarders) are implemented, they take precedence over all others and are always respected.
2. Otherwise, if the class method `replicationSpec` is implemented, or if an application sends `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.
3. In the absence of a replication spec, the instance method `faultToLevel:`, if called, is respected for replicates. Forwarders, of course, do not fault.
4. For classes that use no other mechanism, the configuration parameters `faultLevelLnk` and `faultLevelRpc` are respected.

3.6 Evaluating Smalltalk Code on the GemStone server

In addition to sending messages to forwarders, GemBuilder provides mechanisms to execute ad-hoc Smalltalk code on the server.

Using the development environment Workspace, you can type in and select Smalltalk code and use the menu option "GS-Do it", "GS-Inspect it" or "GS-Print it" to execute the selected text on the GemStone server, and return a replicate of the results.

You can also do this on the client by sending the string to a session for execution. The expression:

`aGbsSession evaluate:` *aString*

when executed on the client, tells GemBuilder to have the server compile and execute the GemStone Smalltalk code contained in *aString*, and answer a client replicate of the result of that execution. If, rather than a replicate, you would like the result as a forwarder, use the expression

`aGbsSession fwevaluate:` *aString*

The code in *aString* may be any arbitrary GemStone Smalltalk code that would be a valid method body (see Appendix A of the *GemStone Programming Guide* for GemStone Smalltalk syntax), with the exceptions that the code:

- cannot take any arguments
- must not refer to the variables self or super
- must not refer to any instance variable of any class

Example 3.3 shows how to use `evaluate:` to execute code.

Example 3.3

```
resultReplicate := GBSM currentSession
   evaluate: '
      | result |
      result := Array new: 3.
      result
         at: 1 put: ''Pear'';
         at: 2 put: #unripe;
         at: 3 put: 42.
      ^ result'
```

You can avoid some of these restrictions by passing in a context object using:

`aGbsSession evaluate:` *aString* `context:` *aServerObject*

or

`aGbsSession fwevaluate:` *aString* `context:` *aServerObject*

The context argument, *aServerObject*, can be any replicate of or forwarder to a GemStone server object. If the code in *aString* refers to the variables self or super, these will be bound to the context object. The code in *aString* can also refer to any instance variables of the context object.

```
aGbsSession
    evaluate: 'self at: 2 put: #ripe'
    context: resultReplicate.
```

The advantage of the `evaluate:` family of messages is that they allow you to execute arbitrary ad-hoc code on the server without previously defining a method.

However, this isn't always the best way to execute server code. The `evaluate:` messages invoke the GemStone Smalltalk compiler upon each execution, and so have extra overhead. Also, the inability to pass arguments rules out the evaluate: messages for some uses.

Message sends through forwarders are the most common means of initiating execution of GemStone Smalltalk code on the server. However, a message passed through a forwarder will fail if the server object that receives the message does not understand that message. Forwarder sends require previous definition of an appropriate GemStone method on the server.

The two forms of execution complement each other. The `evaluate:` messages do not require prior method definition, but cannot take arguments. Forwarder sends require prior method definition, but can take arguments.

3.7 Converting Between Forms

A variety of messages exist to convert between delegates, forwarders, replicates, stubs, and unconnected client objects. The following tables list the results of sending any of several conversion messages to these objects.

A delegate is an instance of GbxDelegate. Delegates are used internally by GemBuilder. An application doesn't normally need to use delegates directly, but you may see them when debugging. We recommend against using delegate protocol, as in Table 3.1, in customer applications.

NOTE

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

Table 3.1 Delegate Conversion Protocol

Message	Return Value
`copy`	a shallow copy of delegate
`asLocalObject`	a replicate
`asGSObject`	`self`
`asForwarder`	undefined *(not recommended)*
`beReplicate`	undefined *(not recommended)*
`fault`	undefined *(not recommended)*
`stubYourself`	undefined *(not recommended)*

Table 3.2 Forwarder (to the Server) Conversion Protocol

Message	Return Value
`copy`	copies associated server object and returns a replicate of the copy
`asLocalObject`	undefined *(not recommended)*
`asGSObject`	the associated delegate (not recommended for customer applications)
`asForwarder`	`self`
`beReplicate`	`self`, which has become a replicate
`fault`	`self` (use `beReplicate` to make a replicate)
`stubYourself`	`self`

Table 3.3 Replicate Conversion Protocol

Message	Return Value
`copy`	shallow copy of delegate, not associated with any server object
`asLocalObject`	undefined *(not recommended)*
`asGSObject`	the associated delegate (not recommended for customer applications)
`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 *(not recommended)*
`asGSObject`	the associated delegate (not recommended for customer applications)
`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`	a shallow copy
`asLocalObject`	undefined *(not recommended)*
`asGSObject`	a new delegate; this creates a new associated server object (not recommended for customer applications)
`asForwarder`	`self`, which has become a forwarder; this creates new associated server object
`beReplicate`	`self`
`fault`	`self`
`stubYourself`	`self`

Chapter

4

Connectors

This chapter describes *connectors*, which allow an application developer to explicitly declare an association between a root client object and a root server object.

- Connectors connect at login. After that, you must explicitly disconnect and reconnect them to effect any changes.
- There are different kinds of connectors for different types of objects.
- At connect time, connectors may update either connected object, depending on how they are set up.
- Connectors exist either in a given set of session parameters, or globally—in every session your image defines.

Connecting Root Objects
explains which objects to associate using connectors.

Connecting and Disconnecting
describes what connectors do and when they do it.

Kinds of Connectors
describes the available kinds of connectors and the differences between them.

4.1 Connecting Root Objects

Every replicate and forwarder in the client is connected to an object in the server. You do not, however, need a connector for every replicate or forwarder. A typical application only needs connectors for a small number of root objects.

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 through 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.

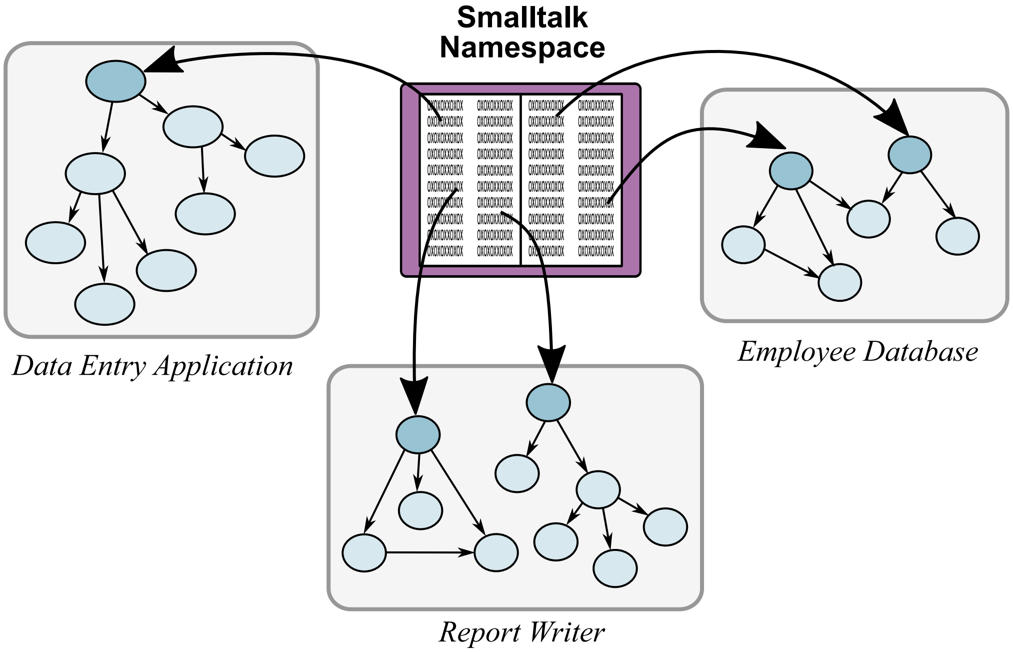
You can take advantage of this hierarchical structure to minimize application overhead. Identify the object at the root of each subsystem of shared objects, and then connect only these root objects. Depending on how you've defined configuration parameters and related matters, you can synchronize entire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

Root objects are often:

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

Figure 4.1 shows an application in which several connected objects are accessed through global or shared variables in client Smalltalk. 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.

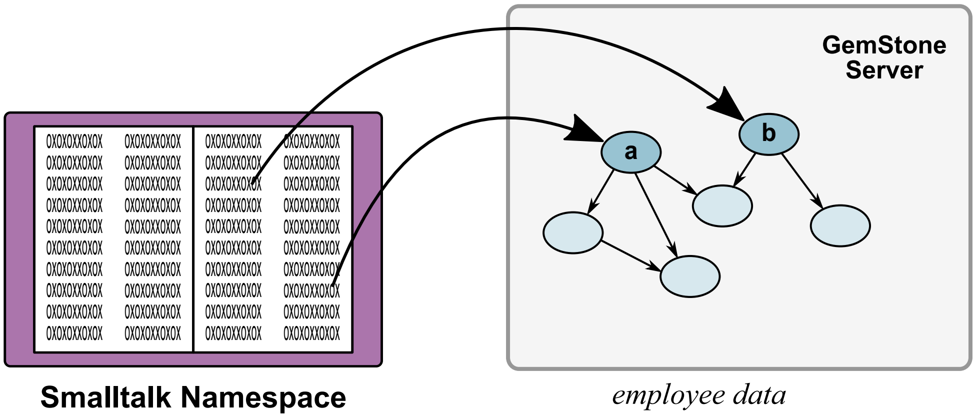
Figure 4.1 Connecting Application Roots



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

Figure 4.2 shows the state of the employee data when stored on the server:

Figure 4.2 Root Objects



In Figure 4.2, 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.

Scope

Some connectors connect their objects whenever any session logs in. Some do so only when logging in using a specific session parameters object:

- *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, the connectors of its session parameters and all global connectors connect automatically. When a session logs out, its connectors disconnect.

Verifying Connections

Connectors are saved in client Smalltalk sets, separate ones for global connectors and each session parameters object. 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 session parameters.

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.

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 server object.

Update GemStone
initializes the GemStone server 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 37.

No initialization
leaves the client object and GemStone server 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 39, or they do not propagate at all, as is normally the case with class connectors, as described in "Class Mapping" on page 35.

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 cannot specify that the client class is a forwarder—the forwarder and clientForwarder postconnect action are unavailable for class connectors.

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 168, provides convenient buttons for the purpose.

NOTE

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

4.2 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.

Connectors with a post-connect action of `#clientForwarder` cannot be explicitly disconnected. These connectors only disconnect at logout.

At logout, GemBuilder sets the instance variables of connectors to nil, if the GemBuilder configuration parameter **connectorNilling** is set to true (the default). This reduces the risk of defunct stub or forwarder errors, replacing them with `nil doesNotUnderstand` errors.

Only connectors whose values are set from the server on login are cleared when **connectorNilling** is true. Session-based name, class variable, or class instance variable connectors that have a postconnect action of `#updateST` or `#forwarder` are cleared. Fast connectors, class connectors, or connectors whose postconnect action is `#updateGS` or `#none` do not have instance variables set to nil.

connectorNilling can be set for individual sessions, if desired. See “Setting Configuration Parameters” on page 119 for details on setting session-specific configuration parameters. The detailed description for this configuration parameter is on page 123.

4.3 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 35, 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 server objects identified by name. Figure 4.3 illustrates how a name connector connects a client object to a server object.

Class variable connector—first resolves the named objects representing the classes, then looks for a class variable in the GemStone class, and a Class or Shared Variable in the client Smalltalk 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. The kernel classes to which they point will not change identity during the course of a session. The GemStone kernel class connectors are predefined, and GemBuilder relies on them. Applications should not define fast connectors.

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

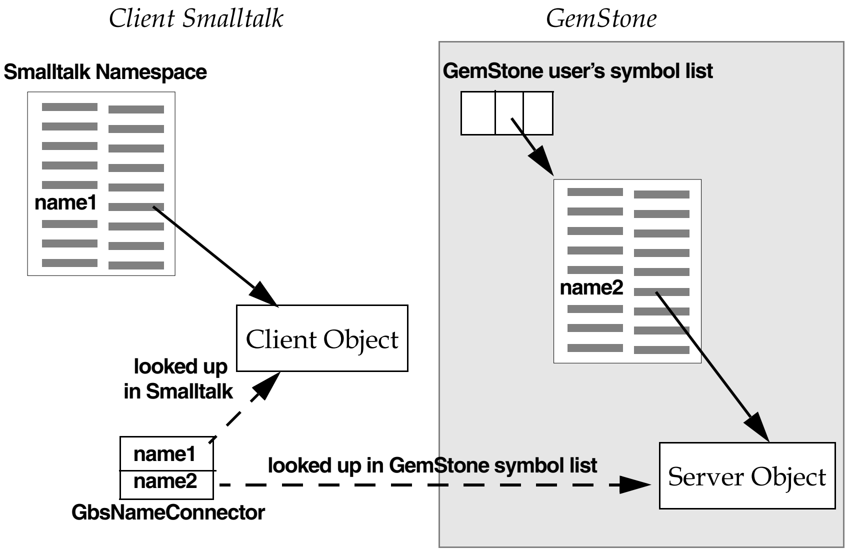
Connecting by Name

Except for fast connectors (discussed in the following section), all kinds of connectors find the objects to connect through a name lookup. Names must be found in namespaces. GemBuilder looks in the namespace "Smalltalk"; a fully-qualified name can also be used.

In the client, 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 connector connects, usually when the session first logs in.

Figure 4.3 Connecting a Name Connector



Connecting by Identity: Fast Connectors

You can bypass name lookup by using a fast connector, which saves direct references to the client Smalltalk objects and the object IDs of the GemStone server objects that are connected.

NOTE
The name "fast connector" is historic. These connectors are not necessarily faster than other connectors.

Using fast connectors can be risky. If the GemStone server 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 do not use fast connectors.

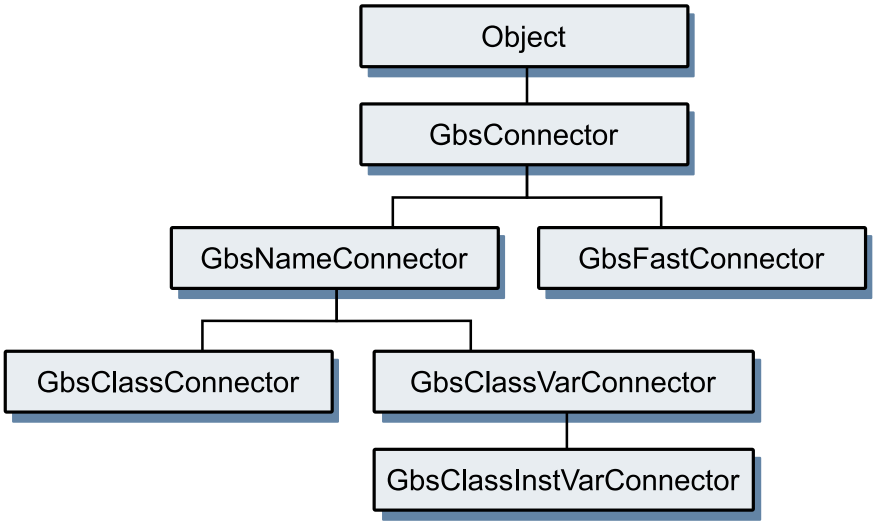
4.4 Making and Managing Connectors

To make and manage connectors interactively, see "Connector Browser" on page 168. 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 4.4 shows the hierarchy.

Figure 4.4 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 "Session Parameters" on page 26.)

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

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

To create a class variable connector:

```
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`, `#clientForwarder`, and `#none`. All connectors default to using `#updateST` except class connectors, which default to `#none`.

To cause a GemStone server 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 parameters object 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 a session logs in using those parameters.

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: 'gs64stone'.
      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:

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 server object represented by the connector, you'll receive an error message stating that the connector failed to connect.

Chapter

5

Managing Transactions

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

Transactions
explains the transaction model, committing, and aborting.

Transaction Modes and Auto-Commit
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.

Gem-to-Gem Notification
describes the mechanisms for inter-gem communications.

Asynchronous Event Error Handling
explains how to handle errors asynchronously.

5.1 Transaction Management: an Overview

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

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

Transactions

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 can arise and cause some commit attempts to fail. *Aborting* the transaction discards any changes to persistent objects and refreshes the session's view of the repository in preparation for further work.

Sessions may also be out of 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 server objects. A session operating outside a transaction can, at any time, begin a transaction.

Transaction modes

The *transaction mode* controls if the session is always in a transaction, or must explicitly begin a transaction. There are three transaction modes: *automatic transaction mode*, *manual transaction mode*, and *transactionless*. In automatic transaction mode, a session is always in transaction—a new transaction is started whenever a transaction completes. In manual and transactionless mode, a transaction must be explicitly started.

Avoiding transaction conflicts

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. GemStone also offers *reduced-conflict classes* that are similar to familiar Smalltalk collections, but are especially designed for the demands of multiuser applications.

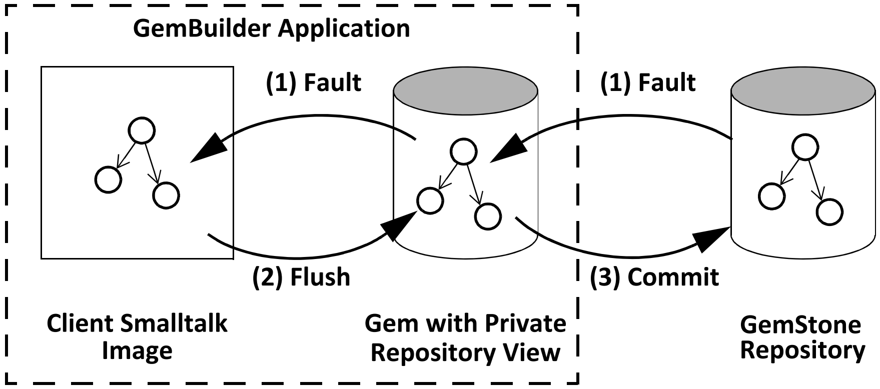
This chapter explains each of the topics mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced conflict classes. Be sure to refer to the related topics in the *GemStone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

5.2 Transactions

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

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

Figure 5.1 GemBuilder Application Workspace



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

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

Operating in a Transaction

Beginning a Transaction

Before the application starts collecting object changes that it intends to committed, it must begin a transaction. In automatic transaction mode, this occurs automatically following login, commit, or abort, but a begin transaction must be explicitly done in other transaction modes.

To begin a transaction from the client, send the message:

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

or:

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

or, in the GemStone Launcher, select the session and click on the **Begin** button; or in any GBS browser, use the **Session** menu item **Begin Transaction**, or the toolbar icon.

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 from the client, send the message:

aGbsSession `commitTransaction` (to commit a specific session)

or:

`GBSM commitTransaction` (to commit the current session)

or, in the GemStone Launcher, select the session and click on the **Commit** button; or in any GBS browser, use the **Session** menu item **Commit Transaction**, or the toolbar icon.

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 to match the current state of the repository, 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 30.)

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

When a session *aborts* its transaction, it discards any uncommitted changes to persistent objects and refreshes its view of the shared object repository. 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 GemStone Launcher, select the session and click on the **Abort** button; or in any GBS browser, use the **Session** menu item **Abort Transaction**, or the toolbar icon.

Aborting has these effects:

- Any changes to persistent objects are discarded.
- 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 server objects remain unchanged.
- Unmodified GemStone server 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 85 for more discussion of the kinds of conflicts that can arise.)

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, and must be kept track of.

To avoid tying up server resources, an application may configure a session to operate *outside a transaction*. When a session is operating outside a transaction, the Stone may request that the session abort, to free up resources.

Being Signaled to Abort by the Stone

When you are in a transaction, GemStone must keep references to every object that was in your view when you started that transaction, including objects that you are actually not using and that are no longer referenced by any other sessions. These obsolete objects cannot be reclaimed until you commit or abort.

When you are running outside of a transaction, 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 that storage. When this happens, you must respond within the time period specified in the STN_GEM_ABORT_TIMEOUT parameter in the Stone's configuration file. If you do not, GemStone either terminates the Gem or forces an abort, depending on the value of the related GemStone server 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 should generally 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 automatically abort when it receives a signal to abort.

5.3 Transaction Modes and Auto-Commit

The GemStone server has three transaction modes, automatic, manual begin, and transactionless. Transactionless is intended for idle sessions, and is not intended for use when actively working, since the view may be updated at any time. In combination with GemBuilder's auto-commit, there are three modes you may use when working in GemBuilder:

- **Automatic transaction mode with manual commits**—starting transactions is automatic, committing must be done manually. This is the default.
- **Automatic transaction mode with auto-commit** —starting transactions is automatic; code changes in GBS browsers are automatically committed.
- **Manual transaction mode**—both starting and committing transactions must be done manually.

On login, the GemBuilder session transaction mode depends on how this is configured in the GemStone server; by default, automatic transaction mode. The mode can be changed after login.

GemBuilder by default has auto-commit off. This can be enabled after login using the toolbar toggle, or menu items.

Automatic Transaction Mode

In *automatic transaction mode*, committing or aborting a transaction automatically starts a new transaction. This is the default, although it can be configured on the GemStone server. In automatic transaction mode, the session operates within a transaction the entire time it is logged into GemStone.

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, and must be maintained in the shared repository.

Manual Transaction Mode

In *manual transaction mode*, the session remains outside a transaction until you choose to begin a transaction. When you abort, or commit successfully, the session goes out of transaction and remains outside a transaction until another begin transaction. You may abort in this mode, which updates your view of the repository, but you may not commit.

When you change the transaction mode from automatic to manual, the current transaction is aborted and the session is left outside a transaction.

When a new transaction is begun, it automatically performs an abort to refresh the session's private view of the repository. Objects that have never been committed are not affected by abort, and can be carried into the new transaction where they can be committed (subject to the usual constraints of conflict-checking).

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

Transactionless

Transactionless mode is similar to manual transaction mode, but in transactionless mode, the session does not maintain a consistent view of the repository. Transactionless sessions may be automatically aborted. Since active work depends on a stable view of the repository, transactionless mode is intended for idle sessions, and there is minimal support in GemBuilder.

Auto-commit

In addition to GemStone server transaction modes, you may enable auto-commit in GemBuilder. Auto-commit is only available for sessions in automatic transaction mode. Auto-commit applies specifically to code changes, not to all changes in object state.

To enable auto-commit, use the toolbar option on any code browser, or a code browser's **Session** menu item **Auto-Commit**; it is also available in the Launcher **Session** menu item **Auto-Commit in Browsers**.

In auto-commit mode, every code change made in the browsers is automatically committed to the repository, which also starts a new transaction. Changes made in inspectors or debuggers do not trigger the commit, although any changes that have been made are included in the next commit.

Auto-commit avoids the risk of losing work, but each method change updates your transactional view of the server. This may result in unexpected changes in a multi-user system in which other users may commit changes to the same server objects you are using.

Choosing Which Mode to Use

During code development, where you may unexpectedly error and lose your session, running in automatic transaction mode with auto-commit ensures that code changes are never lost. However, any code changes you do make are permanent.

If you will be committing changes, but do not want each individual code change to be immediately committed, automatic transaction mode without auto-commit avoids the need to start a transaction before making changes.

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.

Methods supporting transactions

Table 5.1 shows GbsSession methods that support GemStone transactions :

Table 5.1 GbsSession Methods supporting Transactions

`beginTransaction`	Aborts the current transaction, updates the view with any changed server objects, and begins a transaction.
`commitTransaction`	Commits the current transaction, if possible, and updates the view with any changed server objects. If in automatic transaction mode, begins a new transaction.
`abortTransaction`	Aborts the current transaction and updates the view with any changed server objects. If in automatic transaction mode, begins a new transaction.
`transactionMode`	Returns `#autoBegin`, `#manualBegin`, or `#transactionless`.
`transactionMode:`*newMode*	Sets `#autoBegin`, `#manualBegin`, or `#transactionless`.
`inTransaction`	Returns `true` if the session is currently in a transaction.
`signaledAbortAction:` *aBlock*	Executes *aBlock* when a signal to abort is received (see "Being Signaled to Abort by the Stone" on page 82).

5.4 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 may check 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 or write 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.

For more details on transaction management, see the “Transactions and Concurrency Control” chapter of the *GemStone/S 64 Bit Programming Guide*.

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*`.`

The above messages request a particular type of lock on *anObject*. Lock types are described in the *GemStone Programming Guide*. If the lock is granted, the method returns the receiver. 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 a 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*

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*.

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*

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

Example 5.1

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

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

The following methods are used to remove specific locks.

aGbsSession `removeLock:` *anObject*.

aGbsSession `removeLockAll:` *aCollection*.

aGbsSession `removeLocksForSession.`

The following methods answer various lock inquiries:

aGbsSession `sessionLocks.`

aGbsSession `systemLocks.`

aGbsSession `lockOwners:` *anObject*.

aGbsSession `lockKind:` *anObject.*

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 GemStone Smalltalk statements remove all objects from the set of objects whose locks are to be released upon the next commit or abort. These methods are executed using **GS-Do it**:

```
System clearCommitReleaseLocksSet
```

```
System clearCommitOrAbortReleaseLocksSet
```

The statement *aGbsSession* `commitAndReleaseLocks` attempts to commit the current transaction, and clears all locks for the session if the transaction was successfully committed.

5.5 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. For details, refer to the "Transactions and Concurrency Control" chapter of the *GemStone/S 64 Bit Programming Guide*.

5.6 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 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 evaluates a client Smalltalk block, sending a collection of the changed objects as an argument to the block. By examining the argument, the session can determine exactly which objects triggered the signal. (The block must have been previously defined by sending `notificationAction:` to the session, with the block as the argument.)

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 116.)

Example 5.2 demonstrates notification in GemBuilder.

Example 5.2

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

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

5.7 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 string.

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*

Example 5.3

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

"Now, send the signal. This example sends a signal to the same
session"
GBSM currentSession
   sendSignal: 15
   to: (GBSM evaluate: 'GsCurrentSession currentSession serialNumber')
   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.

5.8 Asynchronous Event Error Handling

For each session, there is a background thread that detects events from the server such as sigAbort, lostOTroot, gem to gem signals, and changed object notifications, and other events that are handled internally. If a non-fatal error occurs in processing these events, by default a walkback is opened.

To avoid an end-user experiencing a walkback, you may set a handler block for an unexpected error in this event detection, using the method:

`GbsSession >> eventDetectorErrorHandler:` *aOneArgBlock*

If the eventDetectorErrorHandler is set, and if the exception is not already handled by another handler that is set up for the application, this handler block will be executed for the exception caught by the event detection thread.

Chapter

6

Security and Object Access

This chapter discusses security and access at the object level.

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

6.1 GemStone 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.

GemStone provides security at several levels:

- Login authorization keeps unauthorized users from gaining access to the repository;
- Privileges limit ability to execute special methods affecting the basic functioning of the system; and
- Object level security allows specific groups of users access to individual objects in the repository.

Complete details on GemStone security mechanisms are found in the *System Administration Guide* for your GemStone server product and version. A brief overview is included here.

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.

The UserProfile

Each instance of UserProfile is created by the system administrator. The UserProfile contains information about you as an individual user, such the UserId and password, your SymbolList, any groups you belong to, and your privileges. This information is used to provide system and object level security, including object visibility.

Controlling Visibility of Objects with SymbolLists

One way to control access is to hide certain objects from users. Each GemStone user has a SymbolList, containing a collection of SymbolDictionaries to which they have been given access. Objects, such as Classes, that are not found in a search of the user's SymbolLists are not accessible. Because it is difficult 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 avoids the accidental or inadvertent use of these objects, and provides a small measure of security. It is possible, however, for users to find ways to circumvent this, since it's difficult to ensure that all indirect paths to an object are eliminated.

NOTE

For performance reasons, GbsSession uses transient copies of your symbol lists. 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 synchronized, log out and log back in again.

System Privileges

Users require CodeModification privilege before they can make changes to GemStone server classes and methods, and a number of other GemStone Smalltalk methods are also limited to 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.

Specific privileges and the privileged messages are described in the image, and their use is discussed in the *System Administration Guidee*.

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.

Object-level Security

Object Security Policies

Instances of GemStone's GSObjectSecurityPolicy Class provide read and write authorization control to individual objects. When someone tries to read or write an object that is governed by an object security policy for which he or she lacks the proper authorization, GemStone raises an authorization error and does not permit the requested operation.

Objects may be associated with an object security policy or not. If not, no object authorization is done and any user can read and write the objects.

NOTE

In the 32-bit GemStone/S server product, and in GemStone/S 64 Bit 2.x, object security policies are known as Segments. In 32-bit GemStone/S, nil Segments are disallowed.

All objects associated with a particular object security policy have exactly the same protection; that is, if you can read or write one object with that security policy, you can read or write them all. Each security policy is owned by a specific single user, and may have authorizations for owner, groups, or world for read-only, read-write, or no access.

Groups provide a way to allow a number of GemStone users to share the same level of access to set of objects in the repository.

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

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

6.2 X509-Secured GemStone

Starting with GemStone/S 64 Bit v3.5, the GemStone server supports X509-Secured GemStone, in which all interprocess communication requires mutual X.509-certificate based authentication.

This mode provides many other security features. See the *GemStone/S 64 Bit X509-Secured GemStone System Administration Guide* for more details.

X509-Secured Gemstone requires a different set of session parameters for login. GemBuilder provides parallel support for X509 and regular session parameter logins.

Chapter

7

Exception Handling

This chapter discusses exceptions: how to handle them and how to recover from them, and how to define your own GemStone errors.

GemStone Errors and Exception Classes
describes how GbsErrors and Exceptions are created and used.

Handling Exceptions
explains how to handle exceptions, and how to define and signal your own errors.

Interrupting GemStone Execution
explains how to interrupt GemStone Execution.

7.1 GemStone Errors and Exception Classes

When an error occurs in the GemStone server and is not handled by server Smalltalk execution, an instance of GbsError is created on the client that contains detailed information about the error, and an exception is signaled in GemBuilder. You may set up exception handlers to catch the exception, and perform the desired client Smalltalk exception handling. If no exception handler is set up for the particular exception that occurred, the default handler opens a notifier, from which you can open a debugger.

GBS and GemStone server exceptions are signaled in client Smalltalk as instances of exception classes.

In GemStone/S 64 Bit 3.0 or later, you can replicate exception instances to the client, along with their instance variables. (See the discussion that begins on page 96.) This behavior may be desirable if you wish to more fully bring information about the server exception to the client. To replicate exception instances to the client, you must set the configuration parameter `replicateExceptions` to true. By default, `replicateExceptions` is false and the following behavior applies.

With the GemStone/S 64 Bit 2.x and GemStone/S 32-bit server products, and GemStone/S 64 Bit 3.0 (when `replicateExceptions` is false), exception classes can be found in the application/package GbsExceptions. In VisualWorks, these exception classes are defined in the namespace GemStone.Gbs.

On the GemStone server, there is a dictionary of error names, called ErrorSymbols. This dictionary lists all the errors that can occur when communicating with a GemStone session. For each of these errors, there is a corresponding client exception class. The name of that class is derived by making the first character of the error symbol uppercase, and prepending "Gbs". So, for instance, the server error `#rtErrBadArgKind` corresponds to the client exception class GbsRtErrBadArgKind.

The GemBuilder exception classes fit into the VisualWorks exception hierarchy as shown below:

```
GenericException
   ControlInterrupt
      GbxAbstractControlInterrupt
         <various specific error classes related to pauses, breaks, or breakpoints>
   Exception
      Error
         GbxAbstractException
            GbsGemStoneError
               GbsAbortingError
                  <various specific error classes that cause aborts>
               GbsCompilerError
                  <various specific compile-related error classes>
               GbsEventError
                  <various specific signal related error classes>
               GbsFatalError
                  <various specific fatal error classes>
               GbsInterpreterError
                  <various specific other error classes>
            GbsInterfaceError
               GbsAssertionError
               GbsBlockReplicationError
               GbsClassGenerationError
               GbsConnectorError
               GbsDeprecatedError
               GbsLinkedLoginError
               GbsMethodRemovedError
               GbsNotCachedError
               GbsPrintTimeoutError
               GbsUnsupportedFloatError
               GbsWSAEventSelect
```

The subclasses of **`GbxAbstractControlInterrupt`** are exceptions raised by the GemStone server that are normally used to invoke a debugger. Applications do not typically define handlers for these exceptions.

The subclasses of **`GbsGemStoneError`** are exceptions that are raised by the GemStone server.

The subclasses of **`GbsInterfaceError`** represent client side only GBS errors, that is, errors that are detected by the GBS client.

GemStone/S 64 Bit 3.0 or later (replicateExceptions=true)

For each server exception class, there is a corresponding client exception class. With GemStone/S 64 Bit 3.0 and later, when **`replicateExceptions`** is true, these client exception classes have the same name as the corresponding server classes, and are defined in the namespace GemStone.Gbs.Exceptions.

Before being signaled on the client, the actual exception instance on the server is replicated to the client, along with its instance variables.

7.2 Handling Exceptions

You can use the `on:do:` method to install error handlers to anticipate specific GemStone errors. For example, this shows how to use the exception classes to handle a GemStone server error:

Example 7.1

```
[ GBSM currentSession evaluate: '5 / 0' ]
   on: GbsNumErrIntDivisionByZero
   do: [ :ex | ex proceedWith: 'oops' ]
```

To handle an entire set of errors, rather than an individual error, set up the handler for the common superclass of the errors you want to handle. For example, to handle all GemStone fatal errors, set up a handler for GbsFatalError, something like this:

```
[    ]
       on: GbsFatalError
       do: [ :ex |    ]
```

Each exception detected by the GemStone server also has an associated instance of the class GbsError. This is primarily for internal use by GemBuilder, but you can examine the GbsError by sending #originator to the exception instance. For example:

Example 7.2

```
[ GBSM evaluate: '#( 1 2 3 ) at: 4']
   on: GbsObjErrBadOffsetIncomplete
   do: [ :ex | ex originator inspect ]
```

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.*

This section describes the behavior when **`replicateExceptions`** is false (the default), and for GemStone/S 64 Bit 2.x and GemStone/S 32-bit server products.

- If you're using GemStone/S 64 Bit 3.0 or later (with **`replicateExceptions`** set to true), see the discussion on page 99.

In a GemBuilder application, if you want to define a client Smalltalk exception handler for a user-defined error, you will first need to associate the GemStone error number with a client Smalltalk exception class. To do this, use the method `GbsError class>>defineErrorNumber:class:`. This only needs to be done once for each user defined error.

Note that user-defined error numbers must be unique, and the numbers 1000-6999 are reserved for use by GemStone.

For example, suppose you have created a GemStone user-defined error as follows:

Example 7.3

```
"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 ).
```

The following client Smalltalk code signals your newly created error in GemStone:

```
GBSM evaluate: 'System signal: 10
      args: #[ 46 ] signalDictionary: MyErrors'
```

A generic signal-handler for all GemStone errors would trap this signal. This code sets up the exception handler and causes the exception to be signaled:

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

To explicitly handle your new error in client Smalltalk, you first need to associate the GemStone error number with a client exception class. Create a new class, which should inherit from GbsGemStoneError. In this example, the class MyNewError has been created, and this code associates this class with the GemStone error number:

```
GbsError
      defineErrorNumber: 10
      class: MyNewError.
```

Then to explicitly handle your new error from client Smalltalk:

Example 7.4

```
^[ GBSM evaluate: 'System
   signal: 10
   args: #[ 46 ]
   signalDictionary: MyErrors' ]
      on: MyNewError
      do: [ :ex | ex return: #handled ]
```

You can obtain the exception's error description string by sending it #description. For example:

```
[GBSM execute: '#a at: 2']
      on: GbsObjErrBadOffsetIncomplete
      do: [ :ex | ex return: ex description ]
```

You can obtain the array of server exception arguments by sending #serverArguments to the client exception. This array contains client replicates of the server error arguments.

For example:

```
[GBSM execute: '#a at: 2']
      on: GbsObjErrBadOffsetIncomplete
      do: [ :ex | ex return: ex serverArguments ]
```

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.

GemStone/S 64 Bit 3.0 or later (replicateExceptions=true)

With GemStone/S 64 Bit 3.0 or later, when `replicateExceptions` is true, you can define and signal your own errors by performing the following steps:

1. Create a subclass of an exception class on the server.
2. Create a subclass of the corresponding exception class on the client.
3. Define a connector to connect these two classes.

Subsequently, any exceptions signaled and not handled on the server will be replicated to the client and signaled there.

7.3 Interrupting GemStone Execution

When executing GemStone server Smalltalk in a non-blocking RPC session, it’s possible to interrupt GemStone execution by sending the session #softBreak. GemStone responds by raising GbsRtErrSoftBreak, which is a continuable exception.

GBS automatically sends #softBreak when a client Smalltalk user interrupt occurs in a Process executing GemStone server code (in a non-blocking RPC session). If VisualWorks can’t determine which Process to interrupt when control-Y is entered, try executing GBSM softBreak, or aGbsSession softBreak (if you have more than one GemStone session).

In addition to being continuable, soft break can also be handled on the server by an exception handler. There is another mechanism called hard break which is not continuable, and cannot be handled on the server. Sending an executing GbsSession the message #hardBreak stops server execution, aborts the current GemStone transaction, and raises GbsRtErrHardBreak on the client.

In GBS, in situations where a client Smalltalk user interrupt sends a soft break, a hard break will be sent after the server has not responded to three soft breaks.

Chapter

8

Schema Modification and Coordination

No matter how carefully your schema was designed, sooner or later changes in your application requirements 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.

8.1 Schema Modification

Client Smalltalk and GemStone Smalltalk both have schema modification support.

Client Smalltalk supports only a single version of a class. When a class definition is modified, migration of instances to the new class definition occurs immediately.

Because GemStone stores persistent objects, schema modification is a more complex issue.

GemStone Smalltalk supports schema modification and allows a more flexible migration by allowing you to define different versions of classes.These versions are associated by a class history object. When you redefine a class (or create a new class with the same name as an existing class, within the same SymbolDictionary), it automatically creates a new version of the class, leaving the old class definition also present in the repository; both versions refer to the same classHistory, which contains both class definitions.

It is not necessary that all versions of a class have the same name or other shared attribute; versions of a class are identified by being part of a single class history.

When you create a class definition, GBS will ask if you wish to commit your change and migrate all instances. This will migrate all instance of the old class definition to the new class definition. While convenient, this may be time-consuming for large repositories.

GemStone server smalltalk supports highly configurable instance migration. For details about schema modification, how GemStone handles class versioning, and the options for instance migration, see the "Class Creation, Versions, and Instance Migration" chapter in the *Programming Guide* for your GemStone/S server.

8.2 Schema Coordination

GemBuilder's goal in supporting schema migration is to provide an interaction between the client Smalltalk and the GemStone server 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 server class, resolved at login time (if a connector is defined) by its name. If, while faulting an object into client Smalltalk, GemBuilder discovers that the server object is an instance of a class that is a different version of the class that is in client Smalltalk, and the class version of the server object is earlier than the class version that is already mapped to the client class, 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 the server, the server instance will be of the newer class version. This will lazily-migrate forward server instances to newer class versions.

For example, suppose you have a class named `Foo` on the GemStone server, and there are two versions of it: `Foo [1]` and `Foo [2]`. Suppose that client Smalltalk has a representation of `Foo [2]`. Instances of `Foo [2]` are replicated back and forth between client and server, as usual. If GemBuilder attempts to fault an instance of `Foo [1]`, however, GemBuilder will discover that there is no class mapping for `Foo [1]`. GemBuilder will then do the following:

1. It will fetch the name of the GemStone server class and discover that there is a client Smalltalk class by the same name that is already mapped to a server class.
2. It will verify that the two server classes are in the same class history, and that the version of `Foo [1]` is earlier than the version of `Foo [2]`.
3. It will then ask GemStone to make a copy of the `Foo [1]` instance, migrate the copy to `Foo [2]`, and replicate that migrated copy to the client. The client replicate is mapped to the original `Foo [1]` server instance.
4. If the client replicate is later modified in client Smalltalk, marked dirty, and flushed to the server, it will not be migrated back, and the server object will become an instance of `Foo [2]`.

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 server 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 server class version corresponding to your client Smalltalk class.

Chapter 9

Performance Tuning

This chapter discusses ways that you can tune your GemBuilder application to optimize performance and minimize maintenance overhead.

Profiling
explains ways you can examine your program's execution.

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.

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.

Multiprocess Applications
discusses nonblocking protocol and process-safe transparency caches.

See Chapter 10, "GemBuilder Configuration Parameters" for GemBuilder configuration parameters that can used to tune performance.

For further information, see the *Programming Guide* for your GemStone/S server 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 Profiling

Before you can optimize the performance of your application, you must find out where most of the execution time is being spent. There are client Smalltalk tools available for profiling client 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, within GemStone server execution. See the chapter on performance in the *GemStone Programming Guide* for details.

Profiling Client Smalltalk Execution

GemBuilder can be configured to collect statistics describing the performance of its internal operation. These statistics are archived to a file (a statistics archive file), which can be viewed by GemTalk's VSD (Visual Stat Display) tool (see "The VSD tool" on page 111). Statistics tracking introduces minimal overhead into GBS. A VisualWorks process named "GBS Stat Monitor" samples and archives the statistics at a regular, configurable time interval.

GBS provides several types of statistics:

- Session manager statistics.
- Statistics for each logged in session.
- Object memory statistics for VisualWorks.
- Cache inventory statistics, providing the count and size of objects in the cache.

To enable the tracking of all types of GBS statistics in an image, and start the statistics monitor, execute the following:

```
GBSM statsEnabled: true
```

It can be disabled and the monitor turned off by passing `false` to the above method.

You may also choose to monitor the Main statistics, excluding the cache inventory. To monitor GbsSessionManager, GbsSession, and ObjectMemory statistics only, and to start the statistics monitor, execute:

```
GBSM mainStatsEnabled: true.
GBSM statMonitorRunning: true.
```

Unlike the all-in-one "statsEnabled:" method, this method doesn't start the statistics monitor, so you need to explicitly start the monitor.

The default sample interval for the above methods is 2 seconds (2000 milliseconds). To specific another statistics archiving interval in milliseconds, execute:

`GBSM statSampleInterval:` *milliseconds*

Main Statistics

The session manager statistics are:

Table 9.1 Session Manager Main Statistics

Statistic	Description
cacheStatSampleTime	The amount of time spent sampling cache statistics
clientMapSize	The number of entries in the client map (stObjectCache)
gciCallProtectInvocations	The number of accesses to the GciCallProtect semaphore (a GbxCInterface shared variable)
numSessions	The number of logged in GbsSessions
mainStatSampleTime	The amount of time spent sampling manager and session main statistics
sessionListProtectInvocations	The number of accesses to the sessionListProtect semaphore

The statistics associated with each logged in session are:

Table 9.2 Session Main Statistics

Statistic	Description
bytesSentByStoreTraversal	The number of bytes sent cumulatively by store traversal calls
bytesTraversed	The cumulative number of bytes returned by traversal calls
gciCallsToGem	The number of gci calls made that communicate with the gem
gciCallsToGemTime	The amount of time spent in gci calls that communicate with the gem
objectsStoredByTraversal	The total number of objects stored by store traversal calls
objectsTraversed	The total number of objects (with or without a value buffer) received by traversal call
sigAborts	The number of signaled aborts received
storeTraversals	The number of store traversal calls made
traversalUnpackingTime	The total number of milliseconds spent unpacking traversal buffers
traverseCalls	The number of traversal calls (all types, including more traversal)
traverseCallTime	The amount of time spent in traversal calls

Table 9.2 Session Main Statistics (Continued)

Statistic	Description
changedObjNotifications	The number of changed object notifications received
freeOopsFetched	The number of free oops fetched
gciErrors	The number of errors reported by GciErr()
lostOtRoots	The number of lostOTRoot signals received
nbEndResultNoProgress	The number of times GciNbEnd() was called when the result wasn't ready and no progress was made
nbEndResultProgressed	The number of times GciNbEnd() was called and progress was made
nbEndResultReady	The number of times GciNbEnd() was called and a result was ready
serverMapSize	The number of entries in this session's server map (gsObjectCache)
sessionProtectInvocations	The number of times the sessionProtect semaphore has been invoked
sessionSignals	The number of gem-to-gem signals received

The statistics associated with VisualWorks object memory are:

Table 9.3 Object Memory Main Statistics

Statistic	Description
allocFailures	The number of OldSpace allocations that failed to find a sufficiently large data chunk on the threaded free list; hence the allocation was probably satisfied by eating into the contiguous free space between the OT and the data heap
allocMatches	The number of OldSpace allocations that succeeded with an exact match.
allocProbes	The number of threaded data chunks examined by OldSpace allocations.
allocSplits	Number of OldSpace allocations that succeeded by splitting a data chunk on the free list.
availableContiguousFixedSpace	The size (in bytes) of the largest chunk of contiguous free space in FixedSpace.

Table 9.3 Object Memory Main Statistics (Continued)

Statistic	Description
availableContiguousOldSpace	The size (in bytes) of the largest chunk of contiguous free space in OldSpace after subtracting off the space reserved for use by the virtual machine. This figure does not include the free space on the threaded free lists. Contiguous space is consumed when an object will not fit in an entry on the threaded free lists.
availableContiguousSpace	The size (in bytes) of the largest chunk of contiguous free space in old and fixed space after subtracting off the space reserved for use by the virtual machine. This figure does not include the free space on the threaded free lists. Contiguous space is consumed when an object will not fit in an entry on the threaded free lists.
availableFreeBytes	The number of free memory bytes available in the heap.
availableFreeEdenBytes	The number of free memory bytes available in eden.
availableFreeFixedSpaceBytes	The number of free FixedSpace bytes available for use (i.e., threadedDataBytes + contiguousFreeBytes.
availableFreeLargeSpaceBytes	The number of free LargeSpace bytes available for use (i.e., threadedDataBytes + contiguousFreeBytes.
availableFreeOldSpaceBytes	The number of free OldSpace bytes available for use (i.e., threadedDataBytes + contiguousFreeBytes - reservedContiguousFreeBytes.
availableFreeOldSpaceBytesLimit	The LowSpaceSemaphore is signalled by the virtual machine when the availableFreeOld SpaceBytes drops below this limit.
compCodeCacheBytes	The size of the compiledCodeCache in bytes.
contiguousFreeOldBytes	The total number of contiguous free bytes in OldSpace. This figure does not include any space on the threaded free lists. Some of this space is reserved for use by the virtual machine.
dynamicallyAllocatedFootprint	The total size (in bytes) of the dynamically allocated footprint.
edenBytes	The size of Eden in bytes.
edenUsedBytes	The number of used bytes in Eden.
edenUsedBytesScavengeThreshold	The scavenger will be invoked when edenUsedBytes exceeds this threshold.

Table 9.3 Object Memory Main Statistics (Continued)

Statistic	Description
enumerationCallsPerMillisecond	How many objects per millisecond were processed in the most recent invocation of allInstancesWeakly:. This can be used to estimate the systems object scanning performance for tasks like incremental marking.
fixedBytes	The size of FixedSpace in bytes.
fixedSegments	The number of FixedSpace segments.
fixedUsedBytes	The usage of FixedSpace in bytes.
fixedUsedObjects	The usage of FixedSpace in objects.
freePermBytes	The number of free PermSpace bytes.
incAbortedCount	The number of times the incremental GC aborted its mark phase.
incGCState	The current state of the incremental garbage collector. 0 = resting, 1 = marking, 2 = aborting, 3 = sweeping.
incMarkedBytes	The total bytes of data in the marked objects.
incMarkedObjects	The number of objects currently marked.
incMarkedWeakBytes	The total bytes of data in the marked objects that are weak.
incMarkedWeakObjects	The number of marked objects that are weak.
incMarkStackOverflows	The number of times the incremental GCs mark stack overflowed.
incNilledBytes	The number of bytes in the weak objects examined so far.
incNilledObjects	The number of weak objects examined so far (to see if they contain dead references that need nilling out).
incReclaimedBytes	The bytes of data reclaimed so far in this collection.
incReclaimedObjects	The number of objects reclaimed so far in this collection.
incSweepAllocatedBytes	The bytes of data in the OldSpace objects allocated since the start of the incremental sweep phase.
incSweepAllocatedObjects	The number of OldSpace objects allocated since the start of the incremental sweep phase.
incSweptObjects	The number of objects swept so far in this collection.
incUnmarkedObjects	The number of objects unmarked since aborting the incremental GC.

Table 9.3 Object Memory Main Statistics (Continued)

Statistic	Description
largeBytes	The size of LargeSpace in bytes.
largeFreeBytesTenuringThreshold	The scavenger will tenure LargeSpace objects when the free bytes in LargeSpace drops below this threshold.
largeUsedBytes	The the number of used bytes in LargeSpace.
largeUsedObjects	The number of objects housed in LargeSpace.
maximalFreeOldBytes	The size in bytes of the largest chunk of contiguous free bytes in OldSpace.
nativeStackSpills	The number of times the native stack was not large enough to hold all the internal representation of Smalltalk process frames, thus forcing a spill of context objects into new space. This number is also incremented every time a new Smalltalk process is forked. Excessively high values should be taken as a hint to either increase the native stack space size via ObjectMemory class>>sizesAtStartup:, or to adopt an implementation strategy that forks Smalltalk processes less often. See also ObjectMemory class>>sizesAtStartup for more information.
newBytesAvailableForStorage	The number of bytes that can be used to house objects in new space.
numCompactNMethods	The number of times the compiled code cache was compacted due to lack of space.
numDataCompactions	The number of OldSpace data compactions that have been performed since the start of the VM.
numGCs	The number of compacting GCs that have been performed since the start of the VM.
numGlobalGCs	The number of global GCs that have been performed since the start of the VM.
numIncGCs	The number of incremental GCs that have been performed since the start of the VM.
numMarkStackOverflows	The number of times the GC mark stack overflowed. This should be avoided because it makes GC considerably slower. To address this issue, increase the size of new space (eden + survivor).
numScavenges	The number of new space scavenges that VW has performed since the launch of the VM.

Table 9.3 Object Memory Main Statistics (Continued)

Statistic	Description
numWeakObjectListOverflows	The number of times the weak object list overflowed during GC. This should be avoided because then not all weak objects are processed in a single GC. To address this issue, increase the size of compiled code cache space.
oldBytes	The size of OldSpace in bytes.
oldDataBytes	The total data bytes in OldSpace, including the threaded free data.
oldRtBytes	The size of the old remembered table in bytes.
oldRtEntries	The size of the old remembered table in terms of the total number of entries.
oldRtUsedEntries	The number of actual entries in the old remembered table.
oldSegments	The number of OldSpace segments.
oopsLeft	Estimate of the number of additional objects that can be housed in object memory.
permBytes	The size of PermSpace in bytes.
permDataBytes	The total data bytes in PermSpace.
permOTEs	The number of entries in the PermSpace OT.
reservedContiguousFreeBytes	The bytes of contiguous free space reserved for use by the virtual machine.
rtBytes	The size of the remembered table in bytes
rtEntries	The size of the remembered table in terms of the total number of entries.
rtUsedEntries	The number of actual entries in the remembered table.
stackBytes	The size of StackSpace in bytes.
survBytes	The size of a single SurvivorSpace in bytes.
survUsedBytes	The number of used bytes in the occupied SurvivorSpace.
survUsedBytesTenuringThreshold	The scavenger will start to tenure objects when survUsedBytes exceeds this threshold.
threadedDataBytes	Number of bytes of data in OldSpace's free list.
threadedDataEntries	Number of data chunks in OldSpace's free list.
threadedOTEntries	Number of OTEs in OldSpaces free list.

Cache Inventory Statistics

GBS provides cache inventory statistics, which show the number of instances of, and bytes consumed by, each class of object found in the clientMap (formerly the stObjectCache). When viewing statistics in VSD, each class will be listed in the process list, with the cache statistics numInstances and gbsBytesCached.

To enable cache inventory statistics without enabling main statistics, execute:

```
GBSM cacheStatsEnabled: true
GBSM statMonitorRunning: true
```

Cache inventory statistics are more expensive to sample and archive than the main GBS statistics. Because of this, cache statistics are not sampled and archived every time the statistics monitor performs sampling and archiving of the main statistics. By default, the cache statistics are sampled and archived every 5th time. This value is configurable by sending:

`GBSM cacheSampleIntervalMultiplier:` *anInteger*

This value times statSampleInterval is the interval between two cache statistics samples. For example, with the default cacheSampleIntervalMultiplier of 5 and the default statSampleInterval of 2000 milliseconds, the cache statistics will be sampled and archived every 10000 milliseconds (or once every 10 seconds). A cacheSampleIntervalMultiplier of 1 would mean that cache statistics will be sampled and archived every time the main statistics are sampled.

The VSD tool

The Visual Statistics Display tool, VSD, is provided with the GemStone server product, and can be found in the directory $GEMSTONE/bin/vsd.

VSD does not require the GemStone server; you can download and install VSD onto another machine to perform the analysis. The latest versions of VSD for all supported platforms are provided on the GemTalk website:

https://gemtalksystems.com/products/vsd/

To view one or more GBS statistics files, invoke vsd with the statistic files as arguments, or load the statistics file after starting vsd. See the *VSD User's Guide* for more information on installing and using VSD.

9.2 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 may 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 server 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 server 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 server execution. In particular, repository searching and sorting can be done much more quickly on the GemStone server 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.

9.3 Replication Tuning

The faulting of GemStone server objects into the client Smalltalk is described in Chapter 3. As described there, a GemStone server 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. Communication overhead is important. When fetching an employee object, it is wasteful to stub the name and then immediately fetch the name from GemStone. 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, in the GemStone Launcher choose **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 the default `faultLevelLnk` or `faultLevelRpc` is the only mechanism 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 47.

Setting the Traversal 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

`GbsConfiguration current traversalBufferSize:` *smallIntegerBytes.*

You can also change this value by using the Settings Browser: from the GemStone Launcher, select the Settings icon on the toolbar, or choose **Tools > Settings** and select the **Server Communications** category in the resulting Settings Browser.

This does not affect currently logged-in sessions; it takes effect for sessions that login following the change.

9.4 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.

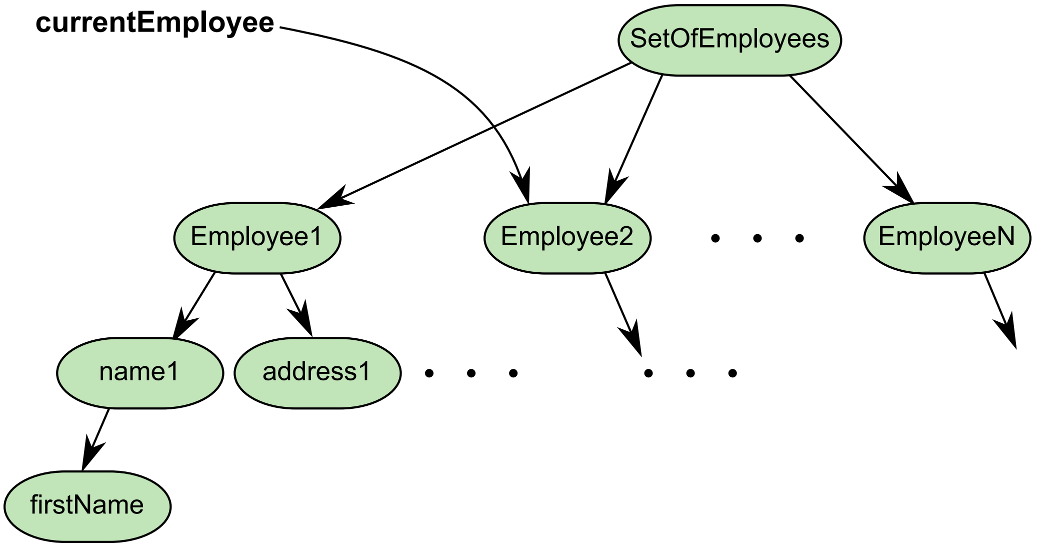
Measures you can take to control the size of the client Smalltalk object cache include explicit stubbing, using forwarders, and not caching certain objects.

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 9.1 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.

If you send `stubYourself` to an object within a method, be careful not to read instance variables of the unstubbed class later in the same method, since the stub will not have these variables and you may observe incorrect `nil` values.

Using Forwarders

Another solution is to declare the `setOfEmployees` as a forwarder. For more information, see "Forwarders" on page 37.

9.5 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* and *C Callouts*. User Actions allow you to access Smalltalk objects using C code, while C Callouts allow you to invoke C libraries using C data types.

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. C Callouts are described in the *Programming Guide* for your GemStone/S server; User Actions are described in the *GemBuilder for C* manual.

9.6 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.

Blocking and Nonblocking Protocol

In a linked GemBuilder session, GemStone operations execute synchronously: the entire client Smalltalk VM 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 (other than the process interacting with GemStone) 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 entire client Smalltalk VM must wait for a GemStone call to return before proceeding.

One Process per Session

Applications that limit themselves to one client Smalltalk 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. 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 with 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:
GbsSession >> gemSignalAction:
GbsSession >> 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 means 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 GemBuilder Configuration Parameters

GemBuilder provides configuration settings that allow GemBuilder to operate differently for development or debugging, control details of the user interface, and tune your program for performance.

This appendix describes the GemBuilder configuration parameters, their default and legal values, and their significance.

10.1 Setting Configuration Parameters

Some configuration parameters control fundamental features of GemBuilder, and must remain the same while the image is running. Other parameters can be modified while GemBuilder is running, and may take effect immediately or at a later point, or can be set individually for a session to override the global behavior. The configuration parameter descriptions starting on page 120 provide specific details for each parameter.

Global settings

When sessions log in, they obtain an initial set of configuration parameters based on the configuration settings in the current global GbsConfiguration.

To determine the current global setting of a parameter, send the parameter name as a message to the global instance of GbsConfiguration, GbsConfiguration current. For example, the following expression returns the setting of the `connectVerification` parameter:

```
GbsConfiguration current connectVerification
false
```

To globally set a parameter, append a colon to the parameter name and send it as a message to the GbsConfiguration instance, with the desired value as the argument. For example, to set the **`connectVerification`** parameter, send:

```
GbsConfiguration current connectVerification: true
```

You may also use the Settings Browser to view and change the settings of these parameters. (see “Settings Browser” on page 140.)

Session-specific settings

While many configuration parameters apply to the image as a whole, other parameters may be modified for specific sessions.

For these parameters, the value in GbsConfiguration current is used at login. Subsequently, you may send the GbsConfiguration messages to the session's configuration (acquired by sending #configuration to the session) to determine or modify the value for that session only.

For example, if the current session requires a larger traversal buffer, an expression such as the following will increase the size for this session, while leaving the global setting for new sessions unchanged.

```
GBSM currentSession configuration traversalBufferSize: 500000.
```

10.2 GemBuilder Configuration Parameters

The following table summarizes GemBuilder configuration parameters. Each parameter is described in detail following the table.

alwaysUseGemCursor

Used to reduce the number of conditions under which GBS switches cursors. When true, GBS changes the cursor to the "gem" cursor during all interactions with the server. When false, the cursor is only changed during some server operations by the GBS tools.

Legal values: true/false
Default: true
Settings Tool tab: User Interface
Scope: Global

assertionChecks

This parameter is for internal use and as directed by GemTalk technical support.

Legal values: true/false
Default: false
Settings Tool tab: Debugging
Scope: Global

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. Do not change this setting while sessions are logged in from this client process.

Legal values: true/false
Default: true
Settings Tool tab: Replication
Scope: Global

backupPolling

The waiting socket interface normally detects events from the server without needing to poll. When backupPolling is false, polling is rare, only every 10 minutes. When backupPolling is set to true, GemBuilder will poll the server every eventPollingFrequency milliseconds, rather than waiting a full 10 minutes.

The default is false on Windows, but since the waiting interface may occasionally miss signals on Unix platforms, the default is true on other platforms.

The default is set when GBS parcels are loaded, so if you move images containing GBS between platforms, you may wish to set this option explicitly.

Legal values: true/false
Default: false on Windows, true on Unix
Settings Tool tab: Signals and Events
Scope: Global

blockingProtocolRpc

Determines whether to use blocking or nonblocking protocol for RPC sessions (linked sessions cannot be non-blocking). When `false`, nonblocking protocol is used, enabling other threads to execute in the image while one or more threads are waiting for a GemStone call to complete. When `true`, GemBuilder must wait for a GemStone call to complete before proceeding with the thread that called it. Should not be changed for sessions that are already logged in.

Legal values: true/false
Default: false
Settings Tool tab: Server Communication
Scope: Session-specific

blockReplicationEnabled

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

Legal values: true/false
Default: true
Settings Tool tab: Replication
Scope: Global

blockReplicationPolicy

Block replication requires decompiling and compiling the source code for blocks at runtime. Since it is usually not possible to include the Smalltalk compiler in a runtime image, block replication may cause problems in runtime applications. 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.

Legal values: `#replicate` or `#callback`
Default: `#replicate`
Settings Tool tab: Replication
Scope: Global

bulkLoad

This parameter has no effect when logged into a GemStone/S 64 Bit server.

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.

Legal values: true/false
Default: false
Settings Tool tab: Cache Tuning
Scope: Global

clampByCallback

Specifies whether replication clamps can be applied by server class callback or not. Support for clampByCallback requires use of the callback feature in recent GemStone/S 64 Bit versions, and is intended to be implemented under the guidance of GemTalk Engineering.

Legal values: true/false
Default: false
Settings Tool tab: Replication
Scope: Session-specific

clientMapCapacity

The minimum capacity in objects of the client object map. The client object map is used to map client replicates, stubs, and forwarders to their corresponding server objects. The map will grow in capacity if it runs out of room, and may shrink in capacity if it has an excess of free space. The map will never shrink its capacity below **`clientMapCapacity`**. This value is also used as the initial capacity of the map, which is initialized only upon the first login after loading GBS into a clean image. Thus, changing **`clientMapCapacity`** will only affect the initial cache size if changed before the first login after GBS load. After that time, however, changing **`clientMapCapacity`** will limit the shrinkage of the map. Growing and shrinking the map take time, so performance-critical applications that replicate many objects may wish to have a larger map capacity. Check statistics for map size and grow/shrink events to see whether your map capacity is sufficient.

The legal value is any positive Integer; GBS will set the actual capacity to an appropriate value somewhat larger than specified.

Legal values: any positive Integer.
Default: 30000
Settings Tool tab: Cache Tuning
Scope: Global

clientMapFinalizerPriority

The process priority at which garbage-collected objects are finalized from the client object map. These must be finalized before the server can garbage-collect the corresponding server objects. By default, this finalization is done at a priority below the normal application priority. This allows finalization to run at times when the main application is waiting for user input or responses from the server. However, if your application seldom waits, and creates a lot of garbage replicates, it is possible that the finalizer might not get enough CPU cycles to keep up. If the unfinalized objects have no remaining references on the server, this will cause increased memory usage in the gem. If this is a problem, you may need to increase this setting to a priority above that of your application. Changing this value will change the finalizer's priority at the time of the next server interaction.

Legal values: an Integer between 1 and 99, inclusive
Default: 30
Settings Tool tab: Cache Tuning
Scope: Global

confirm

When `true`, you are prompted to confirm various GemBuilder actions. Leave set to `true` during application development; deployed applications may set to `false`.

Legal values: true/false
Default: true
Settings Tool tab: User Interface
Scope: Global

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 server 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.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send `#connectorNilling:` to the session's configuration to change the value for that session only. The session's current value will be used at logout.'

Legal values: true/false
Default: true
Settings Tool tab: Connectors
Scope: Session-specific

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.

Connector verification can slow down the login process. Set to `true` during development unless logging in becomes too slow, or your connector definitions are stable. Applications in production should normally set this to false.

In addition to setting this parameter via the Settings Dialog or via message sends, the Connector Browser check box “Global verification”: sets this parameter. See “Connector Browser” on page 168.

Legal values: true/false
Default: false

Settings Tool tab: Connectors
Scope: Global

defaultFaultPolicy

This parameter has no effect when logged into a GemStone/S 64 Bit server, which is always `#immediate`.

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 server 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 server object by updating the client Smalltalk replicate immediately. The defaultFaultPolicy is implemented by Object >> `faultPolicy`. Subclasses can override this method for specific cases.

Legal values: **`#immediate`**/**`#lazy`**
Default: **`#lazy`**
Settings Tool tab: Replication
Scope: Global

deprecationWarnings

When true, any attempt to use a deprecated feature of GemBuilder for Smalltalk causes a proceedable exception to be raised. Deprecated features may be removed in a future release. When false, deprecated features may be used with no warning.

Legal values: true/false
Default: true
Settings Tool tab: Debugging
Scope: Global

eventPollingFrequency

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

Legal values: any Positive Integer
Default: 1000
Settings Tool tab: Signals And Events
Scope: Global

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.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #eventPriority: to the session's configuration to change the value for that session only. The priority will not change immediately, but the new value will be used the next time an action block is set and the event detection process is restarted.

Legal values: an Integer between 1and 99, inclusive

Default: 50
Settings Tool tab: Signals And Events
Scope: Session-specific

faultLevelLnk

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

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #faultLevelLnk: to the session's configuration to change the value for that session only.

Legal values: any Integer
Default: 2
Settings Tool tab: Replication
Scope: Session-specific

## faultLevelRpc

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

The value in GbsConfiguration current is used at login. Subsequently, you may send #faultLevelRpc: to the session's configuration to change the value for that session only.

Legal values: any Integer
Default: 4
Settings Tool tab: Replication
Scope: Session-specific

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 server object.

Legal values: true/false
Default: false
Settings Tool tab: Debugging
Scope: Global

## 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 outdated 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).

Legal values: true/false
Default: true
Settings Tool tab: Replication
Scope: Global

fullCompression

When true, GemStone compresses all communication between the client and the server, reducing the amount of data sent across a network connection to an RPC gem. Has no effect on linked sessions. For network connections with low throughput, compression may improve overall performance. For fast enough network connections, compression may decrease overall performance due to the CPU time required to do compression and decompression.

This setting only takes effect at the time that a library is loaded (see **`libraryName`** below). If a library is loaded you will need to save your image, quit, and restart for a new **`fullCompression`** value to take effect.

Legal values: true/false
Default: false
Settings Tool tab: Server Communication
Scope: Global

gcedObjBufferSize

The initial size in objects of the buffer that holds server object IDs for objects which have been garbage collected in the client. The buffer is enlarged when necessary, but performance-sensitive applications that release many replicates at once may want to avoid this. The IDs in this buffer are sent to the server with each server interaction.

Legal values: any Integer
Default: 2000
Settings Tool tab: Cache Tuning
Scope: Global

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 `generateClientClasses` and `generateServerClasses`. When `false`, session connectors are not automatically created.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateClassConnectors: to the session's configuration to change the value for that session only.

See “Class Mapping” on page 35.

Legal values: true/false
Default: true
Settings Tool tab: Class Generation
Scope: Session-specific

generateClientClasses

When `true`, if a GemStone server 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.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateClientClasses: to the session's configuration to change the value for that session only.

See "Class Mapping" on page 35.

Legal values: true/false
Default: false
Settings Tool tab: Class Generation
Scope: Session-specific

generateServerClasses

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.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #generateServerClasses: to the session's configuration to change the value for that session only.

See "Class Mapping" on page 35.

Legal values: true/false
Default: false
Settings Tool tab: Class Generation
Scope: Session-specific

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.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #initialDirtyPoolSize: to the session's configuration to change the value for that session only. The new value will take effect after the next server operation.

Legal values: any positive Integer. GBS will select a prime size greater than this value.
Default: 10
Settings Tool tab: Cache Tuning
Scope: Session-specific

libraryName

The name of the DLL or shared library to use to contact the server. If this is set to an empty string, GBS loads the first found library with a default name. It tries loading linked libraries first (which support both linked and RPC logins), then RPC-only libraries. If a libraryName is specified, that exact library name is loaded. If the library is not found, an error is reported. On Unix or Linux, the library name may be specified as an absolute file path to the library file, or as a simple name (e.g. libgcirpc.so). On Windows, use a simple name. If a simple name is used, the library is found in the (platform-specific) standard directories for libraries. This setting does not affect any library that is already loaded. If a library is already loaded you will need to save your image, quit, and restart for a new libraryName to take effect.

Legal values: any String
Default: empty String
Settings Tool tab: Server Communication
Scope: Global

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 failed connector does not attempt to resolve again at the next login.

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

When `false`, invalid connectors are ignored.

In addition to setting this parameter via the Settings Dialog or via message sends, the Connector Browser check box “Remove Bad Connectors”: sets this parameter. See “Connector Browser” on page 168.

Legal values: true/false
Default: false
Settings Tool tab: Connectors
Scope: Global

replicateExceptions

When `true` and connected to a GemStone/S 64-bit 3.0 or later server, server exceptions that are not caught on the server will be replicated to the client and signaled on the client stack. This allows objects referenced from exception instance variables to also be replicated to the client.

When `false`, or when connected to an earlier server product or version, server errors are associated with a client error class by number.

Legal values: true/false
Default: false
Settings Tool tab: Replication
Scope: Global

serverMapLeafCapacity

The lower bound in objects of the capacity of each leaf in the server map. The server map maps the IDs of server objects to their corresponding client replicates, forwarders, and stubs. The server map is structured as a shallow fixed-depth tree. Each node in the tree is identified by the upper bits of the object ID. Each leaf node is a hashed collection indexed by the lower bits of the object ID. The upper limit of the capacity for each leaf depends on the server product and version, ranging from 2^{22} to 2^{24} object IDs. Leaves are created on demand; only those leaves that actually contain objects exist. This setting controls the initial capacity of each leaf. Leaves will grow and shrink as necessary, but will not shrink below this setting. Growing and shrinking take some time, so performance-sensitive applications may want to adjust this value. Using a larger value decreases the time spent growing and shrinking each leaf, but increases memory use, and also increases the time spent initializing each leaf.

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #serverMapLeafCapacity: to the session's configuration to change the value for that session only. From that point on, new leaf creation and all leaf shrinkage will be subject to the new value.

Legal values: any positive Integer. GBS will select a prime table size greater than this value, but not exceeding 2^{24}.
Default: 400
Settings Tool tab: Cache Tuning
Scope: Session Specific

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.

Legal values: true/false
Default: false
Settings Tool tab: Debugging
Scope: Global

traversalBufferSize

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

This setting can be different from session to session. The value in GbsConfiguration current is used at login. Subsequently, you may send #traversalBufferSize: to the session's configuration to change the value for that session only. An increase in size will take effect immediately, but a decrease may not.

Legal values: any positive Integer. The actual setting must be a multiple of 8, larger than 2048. If an illegal number is entered, it will be replaced with the nearest legal number.
Default: 250000
Settings Tool tab: Server Communication
Scope: Session-specific

useGbsMemoryPolicy

When true, GemBuilder will install a memory policy that defers some garbage collection activities until the current replication is finished, when more garbage is likely to be available for collection. In conditions of heavy replication, this can both control memory growth and save time running unproductive garbage collection.

When false, a normal VisualWorks MemoryPolicy is used. You may tune the GBS memory policy the same way you would a VisualWorks MemoryPolicy. Tuning parameters are preserved when changing the state of this parameter.

Legal values: true/false
Default: true
Settings Tool tab: Cache Tuning
Scope: Global

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.

Legal values: true/false
Default: true
Settings Tool tab: User Interface
Scope: Global

Chapter 11 The GemStone Tools: an Overview

This part of the manual introduces you to the GemBuilder visual programming environment. This and the following chapters describe the tools for managing logins to the GemStone server, programming in GemStone Smalltalk, and the GemBuilder tools for performing some GemStone Administrative tasks.

GemStone Launcher
describes the GemStone Launcher, which provides session management and access to the GemStone tools.

Settings Browser
describes how to examine and set GemBuilder configuration parameters with the Settings Browser.

System Workspace
describes the System Workspace.

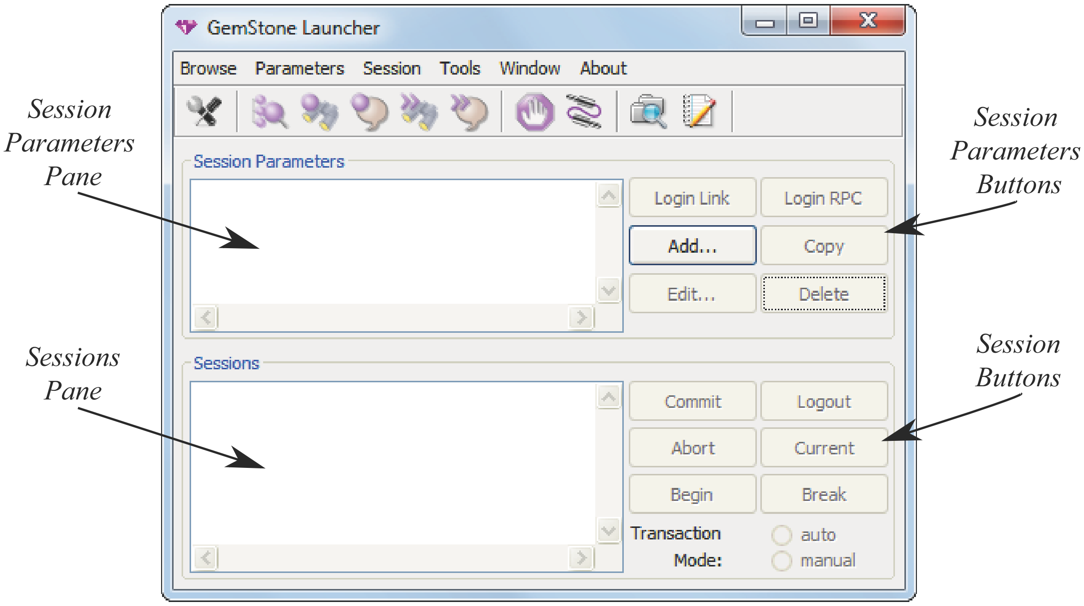
11.1 GemStone Launcher

The GemStone Launcher provides the starting place for your interactions with GemBuilder and the GemStone server. It streamlines logging in and logging out of GemStone and managing sessions and transactions, as well as providing menus and toolbar shortcuts that allow you to access all the GemBuilder programming and administrative functions.

This section describes the tools for managing Session Parameters and Sessions and handling logins. For a more detailed discussion, and the underlying programmatic login process, see the discussion under “Session Parameters” on page 26.

To access the GemStone Launcher, use the VisualWorks Launcher menu item **GemStone > GemStone Launcher**, or use the equivalent Toolbar icon.

Figure 11.1 GemStone Launcher



Session Parameters

To log in to a GemStone repository using the GemStone Launcher, you need to have a Session Parameters. The GemStone Launcher allows you to create new Session Parameters.

The **Add...** button allows you to create a regular Session Parameters; using the Menu item Parameters, you may also create X509 Session Parameters, which allow to specify certificates needed to login to X509-secured GemStone.

The list of previously registered session parameters is displayed in the **Session Parameters Pane**, the upper left part of the window. The buttons in this pane perform operations on Session Parameters. These operations are also found in the **Parameters** menu and the Session Parameters Pane popup menu.

You may drag and drop Session Parameters with the list, to reorder them for convenience. The order is not relevant.

Sessions

Once you select a session parameters and log in successfully, your session, along with any other logged in sessions, is displayed in the **Sessions Pane**, the lower left part of the window. The Sessions Buttons operate on the selected session in the Sessions Pane. You may login multiple sessions from a single session parameters, or multiple sessions based on different parameters.

The operations provided by the Sessions Buttons, as well as additional operations, are available on the **Session** menu and the Sessions Pane popup menu.

Current Session vs. Selected Session

Selecting a session in the Sessions Pane highlights the session by coloring the background of the text. The Sessions Buttons and Session menu items, and other GemStone Launcher menu items such as those on the Browse menu, operate on the selected session or are associated with the selected session, and are disabled if no session is selected.

The current session is in bold text. This is the session used when executing code using system-wide GemStone code execution such as the Workspace or client Smalltalk browser menu items "GS-Do it", etc., or in the debugger. If any sessions are logged in, one of these is always the current session; it may or may not be the selected session.

To make a session the current session, select that session and use the **Current** button, or the menu item **Session > Be Current Session**.

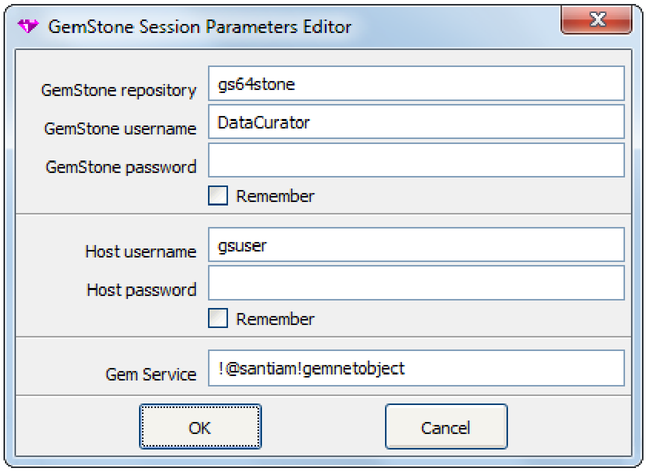
Session Parameters Editor

The Session Parameters Editor is used to create a new session parameters, or to edit an existing one. You must define a Session Parameters before you can login.

Creating a new Session Parameters

To create a new (regular) Session Parameters, select the menu item **Parameters > Add...**, or the **Add..** button, to open a dialog allowing you to define a set of session parameters.

Figure 11.2 Session Parameters Editor



GemStone repository	gs64stone
GemStone username	DataCurator
GemStone password	
	☐ Remember
Host username	gsuser
Host password	
	☐ Remember
Gem Service	!@santiam!gemnetobject

The first time this is done after starting the client Smalltalk image, the server-specific client libraries are loaded. Any problems in the client library configuration will show up now. For more information on these libraries, see "Client Libraries" on page 23, and the *Installation Guide*.

In the Session Parameters Editor, specify the following session parameters:

- **GemStone repository**
  For a Stone running on a host other than the Gem host (described under Gem service

below), you must include the server's hostname in Network Resource String (NRS) format. (NRS format is described in an appendix to the *System Administration Guide*)

- **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. 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, or if netldi is running in guest mode)
  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.

  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.
- **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 `gemnetobject`.

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

  ```
  !@pelican!gemnetobject
  ```

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

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 GemStone Launcher.

Editing Existing Session Parameters

To change a session parameters object, select the session parameters in the GemStone Launcher, and select the menu item **Parameters > Edit**, or the **Edit..** button. This opens a Session Parameters Editor displaying the parameters. Clicking on **OK** causes your changes to take effect.

For more details on the available operations for a session parameters, see "Session Parameters Buttons and Parameters Menu" on page 138.

X509 Session Parameters Editor

In addition to regular sessions that are logged in using Session Parameters, GemBuilder supports X509 logins, which require a different set of Session Parameters.

To edit X509 Session Parameters, you can use the X509 Session Parameters Editor. This dialog is accessed by the menu item **Parameters > Add X509...**.

Login using X509 Session Parameters is limited to GemStone/S 64 Bit v3.5 that has apprproate licence, and with the X509-secured GemStone environment configured and started up. See the *GemStone/S 64 Bit X509-Secured GemStone System Administration Guide* for more on X509 logins from GemBuilder.

Logging into and logging out of GemStone

The session parameters specify a Stone by name, on a the same or a different node, and for an RPC session, includes the name or port of the NetLDI (network long distance information). Before you can login, the specified Stone must be running on that node, and if needed, the specified NetLDI must be running. The versions of the stone and NetLDI must also match the client shared library that is (or will be) loaded into GBS.

Depending on the terms of your GemStone license, you can have many sessions logged in at once from the same GemBuilder client. These sessions can all be attached to the same GemStone repository, or they can be attached to different repositories, provided they are running the same version of the GemStone server.

Logging in to GemStone

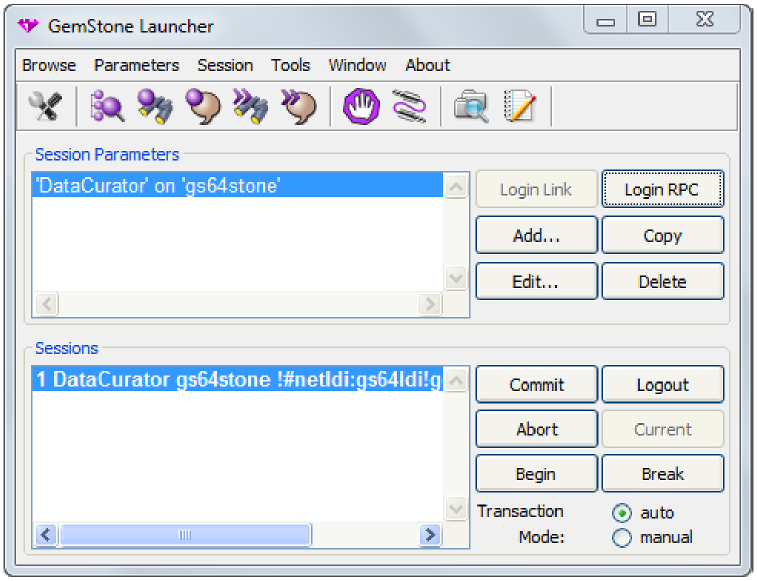
To log into the GemStone server with the GemStone Launcher, select the session parameters object in the Session Parameters pane, and click on either **Login Link** or **Login RPC**. You may also use the equivalent menu items on the **Parameters** menu.

Note that linked logins are not always available. In order to login linked, you must use the ”lnk” version of the shared libraries, be running the appropriate bit size (32-bit or 64-bit) version of the client VisualWorks executables, and have the appropriate GemStone server components available on the client.

When you are logged in, the GemStone Launcher displays the session description in the Sessions Pane.

On login, sessions are in automatic transaction mode. If you want to be in manual begin transaction mode, for example if you are only viewing code and do not want to consume resources on the GemStone server, select the radio button or use the **Session > Automatic Begin Mode** menu item.

Figure 11.3 GemStone Launcher with session logged in



'DataCurator' on 'gs64stone'

1 DataCurator gs64stone !#netldi:gs64ldi!g

Transaction Mode: ● auto ○ manual

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

Logging Out of GemStone

To log out of GemStone from the GemStone Launcher, select the session in the lower pane and click the **Logout** button.

Before logging out, GemBuilder prompts you to commit your changes, if the GbsConfiguration setting confirm is true (it is true by default). 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.

For more details on the operations on each session, see “Sessions Buttons and Sessions Menu” on page 139.

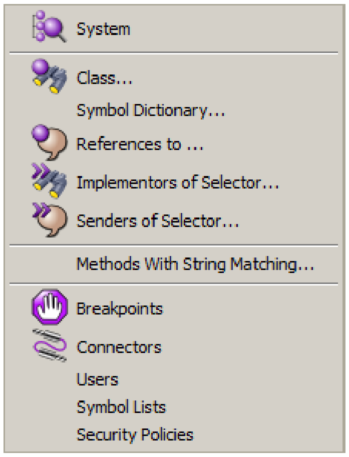
11.2 GemStone Launcher Menus and Toolbar shortcuts

The GemStone launcher provides direct access to most commonly used GemBuilder functions. The Toolbar provides access to a subset of the operations that are available in the menu bar menus and popup-menus

Browse Menu

The Browse menu provides access to the programming tools, such as the System Browser, and the administrative tools. The most commonly accessed of these tools also have icons on the ToolBar.

The following items are under the menu bar **Browse** menu.



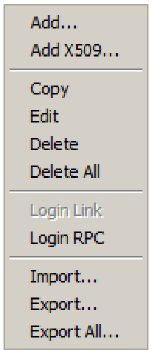
System	Opens a System Browser, comparable to the client Smalltalk System Browser. The System Browser is described starting on page 146. Available as a ToolBar shortcut.
Class...	Prompts for the name of a class and opens a System Browser focused on that class. Available as a ToolBar shortcut.
Symbol Dictionary...	Prompts for the name of a symbol dictionary, then opens a System Browser focused on that dictionary.
References to...	Prompts for the name of class or class variable name, then opens a References Method List Browser listing all methods that refer to that variable. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used. Available as a ToolBar shortcut.
Implementors of Selector...	Prompts for the name of a message selector, then opens an Implementors Method List Browser showing all methods that implement that selector. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used. Available as a ToolBar shortcut.
Senders of Selector...	Prompts for the name of a message selector, then opens a Senders Method List Browser showing all methods that send that selector. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used. Available as a ToolBar shortcut.
Methods with String Matching...	Prompts for a string, then opens a browser listing all methods whose source code contains the specified string.
Breakpoints	Opens a Breakpoint Browser, allowing you to view and manipulate breakpoints in GemStone Smalltalk code. The Breakpoint Browser is described on page 178. Available as a ToolBar shortcut.

Connectors	Opens a Connector Browser, allowing you to view and manipulate connectors between Client and GemStone server objects. The Connector Browser is described on page 168. Available as a ToolBar shortcut.
Users	Opens the User Account Management Tools, 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 Management Tools are described on page 190.
Symbol Lists	Opens a Symbol List Browser, allowing you to examine and modify symbol dictionaries and their entries. The Symbol List Browser is described on page 187.
Security Policies	Opens a Security Policy Tool, allowing you to control authorization at the object level by assigning objects to security policies. The Security Policy Tool is described on page 182.

Session Parameters Buttons and Parameters Menu

The Parameters menu, and the popup menu in the Session Parameters Pane, include operations that are performed on the Session Parameters. Many of these functions are also provided as Session Parameters Buttons.

The following items are on the parameters menu:



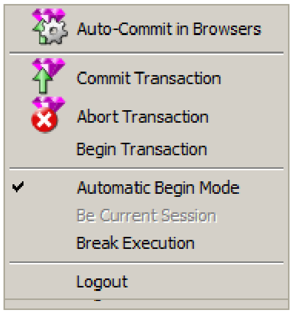
Add...	Open an empty Session Parameters Editor. Available as a Session Parameters Button.
Add X509...	Open an empty X509 Session Parameters Editor. This creates Parameters that can only be used to login to X509-Secured GemStone, a server feature in GemStone/S 64 Bit v3.5 and later.
Copy	Make a copy of the selected session parameters, and add it to the list. Available as a Session Parameters Button.
Edit	Open a Session Parameters Editor on the selected session parameters. Available as a Session Parameters Button.
Login Link	Log in linked, using the selected session parameters. Available as a Session Parameters Button.
Login RPC	Log in RPC, using the selected session parameters. Available as a Session Parameters Button.
Remove...	Remove the selected session parameters. Available as a Session Parameters Button.
Import...	Prompt for a filename, and read a previously exported list of Session Parameters from a text file.
Export...	Prompt for a filename, and write out the selected Session Parameters to a text file. The exported information is in plain text. If you have included passwords in your session parameters, these will be visible.

Export All...	Prompt for a filename, and write out the complete list of Session Parameters to a text file. The exported information is in plain text. If you have included passwords in your session parameters, these will be visible.

Sessions Buttons and Sessions Menu

The Sessions menu, and the popup menu in the Sessions Pane, include operations that are performed on the selected Session. Many of these functions are also provided by the lower right set of buttons.

The following items are on the Sessions menu:

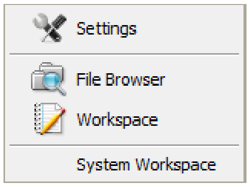


Auto-Commit in Browsers	Turns on auto-commit, so saved changes in the code browsers are automatically committed. Only code changes in code browsers are committed via auto-commit, although auto-commit does commit all changes in your transaction.
Commit Transaction	Commit all changes in the selected session. Available as a Session Button.
Abort Transaction	Abort in the selected session. Available as a Session Button.
Begin Transaction	Abort in the selected session and start a new transaction. Available as a Session Button.
Automatic Begin Mode	Toggles the transaction mode between Automatic begin and Manual begin. When this is checked, the session is in automatic transaction mode, when unchecked, in manual transaction mode. For more information on transaction modes, see the details starting on page 83. Available as radio button options with the Session Buttons.
Be Current Session	Makes the selected session into the current session. This is described in more detail on page 133. Available as a Session Button.
Break Execution	Sends a soft break to the selected session, interrupting any currently executing server Smalltalk code, and open a debugger. Available as a Session Button.
Logout	Logs out the selected session. Available as a Session Button.

Tools Menu

The Tools menu provide access to operations whose functionality is primarily provided by the client Smalltalk environment.

The following items are on the Tools menu:



Settings	Opens a Settings Browser in which you can examine, change, and store parameters for configuring GemBuilder. The Settings Browser is described on page 140. Available as a ToolBar shortcut.
File Browser	Opens a client Smalltalk File Browser. Available as a ToolBar shortcut.
Workspace	Opens a client Smalltalk Workspace. Available as a ToolBar shortcut.
System Workspace	Opens the GemStone System Workspace, a workspace containing a variety of useful GemStone Smalltalk and client Smalltalk expressions. The System Workspace is described on page 143.

Window Menu

The window menu allows you to collapse and re-expand most GemStone Windows, other than the GemStone Launcher itself.

Raise GemStone Windows	Expands all minimized GemBuilder for Smalltalk windows.
Collapse GemStone Windows	Minimizes all GemBuilder for Smalltalk windows, except the GemStone Launcher.

About Menu

The About menu contains two items:

About GemBuilder®...	Opens a window providing the GemBuilder version and copyright information.
About Gemstone®Server...	Opens a window providing the GemStone server product and version, and login information.

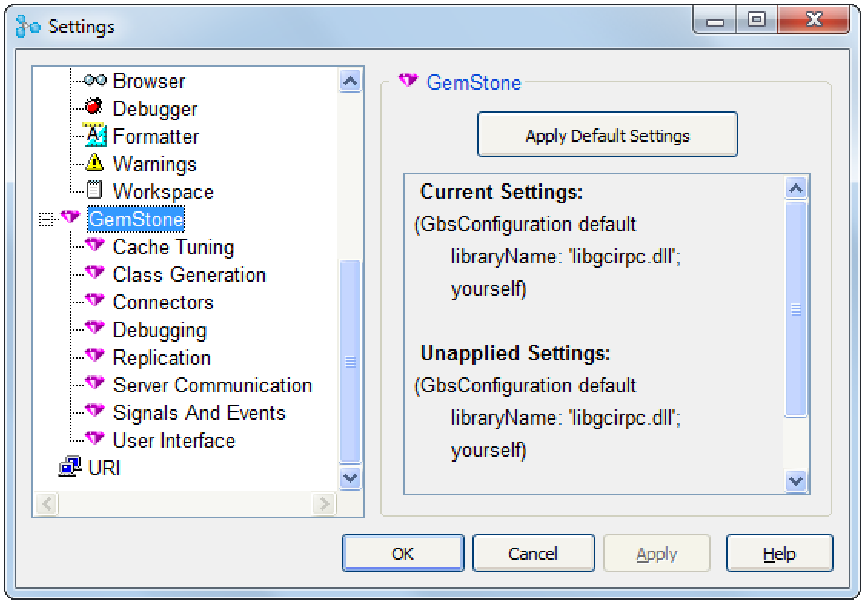
11.3 Settings Browser

The Settings Browser allows you to examine and set the configuration parameters for GemBuilder. The Settings Browser is integrated with the client Smalltalk settings.

Opening the Settings Browser

To open the Settings Browser, select **Tools > Settings** from the GemStone Launcher; or select **System > Settings** from the VisualWorks Launcher and scroll down to see the GemStone configuration settings categories.

Figure 11.4 The Settings Browser Summary



Current Settings:

```
(GbsConfiguration default
    libraryName: 'libgcirpc.dll';
    yourself)
```

Unapplied Settings:

```
(GbsConfiguration default
    libraryName: 'libgcirpc.dll';
    yourself)
```

Buttons on the Settings Browser

Apply Default Settings	(At the highest level only) Reset all GemStone settings to their default value.
OK	Close the Settings Browser, applying all changes to the current configuration.
Cancel	Close the Settings Browser, cancelling all unapplied changes.
Apply	Apply all unapplied changes to the current configuration.
Help	Open a dialog with help for the configurations in the categories that you are currently viewing.

Parameter Categorization

The Settings Browser categories the parameters under headings. Selecting each heading allows you to update a set of related configuration parameters.

Table 11.1 Settings Browser Categorization

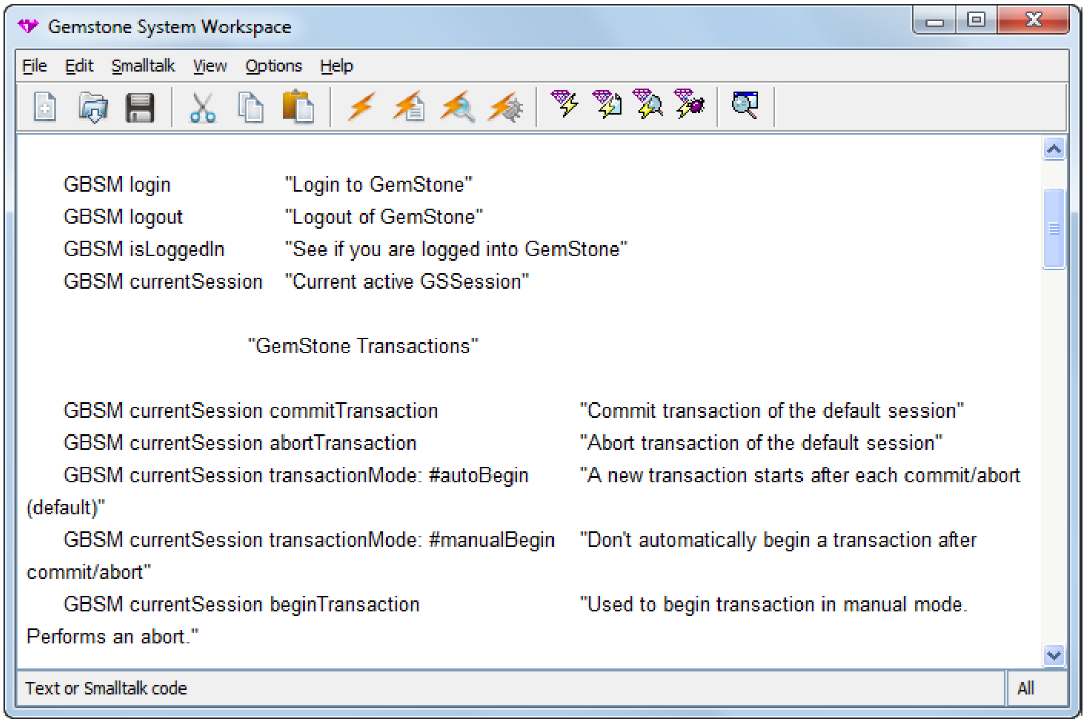
Cache Tuning	bulkLoad clientMapCapacity clientMapFinalizerPriority gcedObjBufferSize initialDirtyPoolSize serverMapLeafCapacity useGbsMemoryPolicy
Class Generation	generateClassConnectors generateServerClasses generateClientClasses
Connectors	connectorNilling connectVerification removeInvalidConnectors
Debugging	assertionChecks deprecationWarnings forwarderDebugging stubDebugging
Replication	autoMarkDirty blockReplicationEnabled blockReplicationPolicy clampByCallback defaultFaultPolicy faultLevelLnk faultLevelRpc freeSlotsOnStubbing replicateExceptions
Server Communication	blockingProtocolRpc fullCompression libraryName traversalBufferSize
Signals And Events	backupPolling eventPollingFrequency eventPriority
User Interface	alwaysUseGemCursor confirm verbose

11.4 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 11.5),From the GemStone Launcher choose **Tools > System Workspace.**

Figure 11.5 System Workspace



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

Chapter

12 Using the GemStone Programming Tools

GemBuilder provides a set of code browsers that allow you to browse and edit code residing on the GemStone server. Writing code in GemStone Smalltalk is similar to writing code in client Smalltalk, but there are some important differences. Some of these differences are mentioned in this chapter; for more detail on programming in GemStone Smalltalk, see the *GemStone/S Programming Guide*.

In addition to the browsers and related tools provided by GBS, menus in client Smalltalk tools will have additional menu options to execute GemStone Smalltalk code.

Browsing Code
describes the GemStone System Browser and other code browsers, and the available menu items.

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

Fileout and Filein of Class and Method Definitions
describes how to create text files containing your code.

Connectors
describes the connector browser, and how to use it to setup connectors between your client classes and objects and server classes and objects.

12.1 Browsing Code

GemBuilder includes a number of Code Browsers, allowing you to browse all code in the System, or browse smaller subsets of code. These browsers are similar to the client Smalltalk browsers, allowing you access source code and other information about each of the kernel classes and methods, and create classes and methods in the GemStone repository.

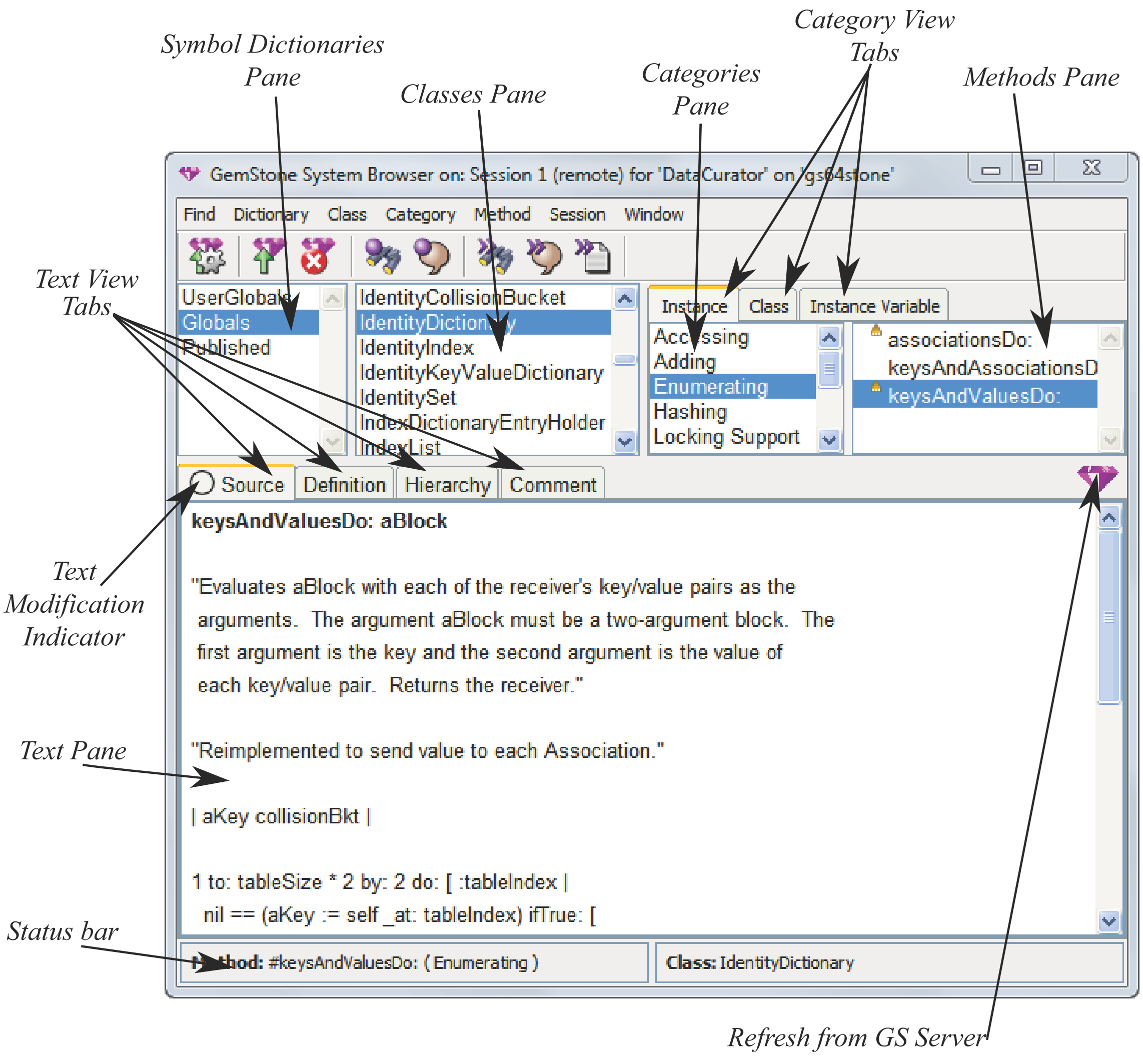
System Browser

When logged in to GemStone, you can open a GemStone System Browser by choosing **Browse > System**, or by selecting the icon on the GemStone Launcher Toolbar.

The System Browser is similar to the client Smalltalk System Browser. The GemStone System Browser allows you access source code and other information about each of the kernel classes and methods, and create classes and methods in the GemStone repository.

Each pane of the each of the GemStone Browsers also has pop-up menus that are specific to that pane; these menus are also available from the appropriate menu on the menu bar. Some of these operations are also available via shortcuts on the ToolBar.

Figure 12.1 GemStone System 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: [
```

Method: #keysAndValuesDo: (Enumerating)
Class: IdentityDictionary

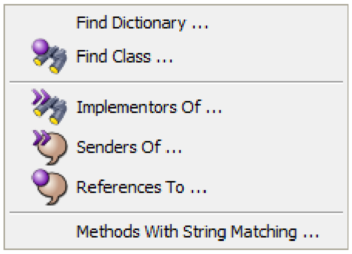
Most of the features of the system browser are likely to be familiar from client Smalltalk browsers, although there are some features and menus that are unique.

Note these browser features:

- **Text Modification Indicator**. This icon on the Text View tabs **Source**, **Definition** or **Comment** becomes red if you have modified the code in the text pane. You can use the Accept or Cancel menu options to save or discard the changes.
- **Refresh From GS Server**. Clicking on this pink diamond refreshes the view in the current browser from the GemStone server.

Find Menu

The **Find** menu allows you to find Symbol Dictionaries, Classes, and Methods within GemStone.



The options available on this menu are:

Find Dictionary...	Prompts for the name of a SymbolDictionary, and opens a SymbolDictionary Browser on the result.
Find Class...	Opens a Select Class dialog, which allows you to select the name of a class from a list of available classes. Navigates to the selected class and Symbol dictionary, or opens a Hierarchy Browser on the selection, depending on the Browser.
References to...	Prompts for the name of a variable, then opens a References Method List Browser listing all methods that refer to that variable. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used.
Implementors of...	Prompts for the name of a message selector, then opens an Implementors Method List Browser showing all methods that implement that selector. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used.
Senders of...	Prompts for the name of a message selector, then opens a Senders Method List Browser showing all methods that send that selector. Search accepts wildcard characters * and ?, and is case-insensitive unless any uppercase characters are used.
Methods with String Matching...	Prompts for a string, then opens a browser listing all methods whose source code contains the specified string. The search is case-sensitive.

Symbol Dictionaries Pane and the Dictionary menu

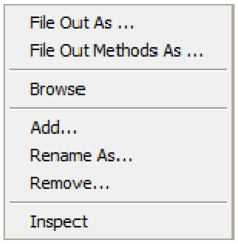
The Symbol Dictionaries Pane displays the current GemStone user's SymbolList, which is a collection of Symbol Dictionaries. Symbol Dictionaries are comparable to Namespaces; they contain a collection of classes, and their ordering controls the resolution of Class and variables names. For more on how Symbol Dictionaries resolve symbols, refer to *GemStone/S Programming Guide*.

You can reorder Symbol Dictionaries by dragging and dropping within the Symbol Dictionaries pane.

Each user normally has a different set of symbol dictionaries. Symbol dictionary access is managed using tools that are described in Chapter 14.

When you select a symbol dictionary in the Symbol Dictionaries 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, or by selecting **Browse > Symbol Lists** from the GemStone Launcher).

The following menu items are in this pane's pop-up menu and under the menu bar **Dictionary** menu.



File Out As...	Prompts a file name to save all the class and method definitions for all the classes in the selected SymbolDictionary.
File Out Methods As...	Prompts a file name to save all the method definitions for all the classes in the selected SymbolDictionary.
Browse	Open a SymbolDictionary Browser on the selected SymbolDictionary.
Add...	Add a new Symbol Dictionary
Rename As...	Rename the selected SymbolDictionary.
Remove...	Remove the selected SymbolDictionary (does not apply to Globals).
Inspect	Open an inspector on the selected SymbolDictionary. This allows you to see Globals in the SymbolDictionary as well as the classes.

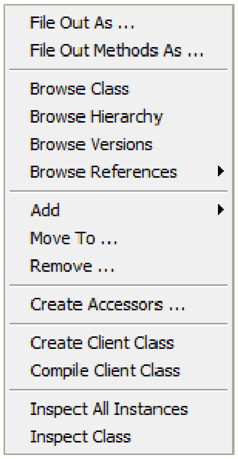
Classes Pane and the Class menu

The Class pane displays a list of the classes in the selected symbol dictionary.

Selecting a class displays the list of categories in this class in the Category pane, all methods of that class in the methods pane, and sets the Text View Tabs to the Definition tab so the class definition is shown in the Text pane.

When you have a class selected, the Text View Tabs on the Text pane can be used to control what is viewed: the class definition, hierarchy, or class comment.

The following menu items are in this pane’s pop-up menu and under the menu bar **Class** menu.



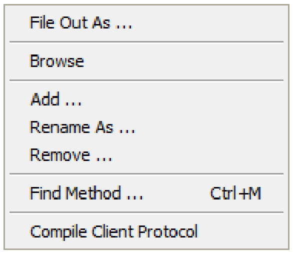
File Out As...	Prompts for a file name under which to save the class definition and all methods of the selected class.
File Out Methods As...	Prompts for a file name under which to save all methods of the selected class, but not the class definition itself.
Browse Class	Open a Class Browser on the selected class.
Browse Hierarchy	Open a Hierarchy Browser on the selected class.
Browse Versions	Open a Class Version Browser on the selected class.
Browse References	Open a References Browser on all references to an instance variable, a class variable, to the current version of the class, or to all versions of the class.
Add	Open a dialog allowing you to create a new class, or to add an instance variable to the currently selected class.
Move to...	Prompt for another SymbolDictionary to which to move the selected class.
Remove...	Remove the selected class.
Create Accessors...	Opens a dialog allowing you to create setter and getter methods for instance variables of the selected class.
Create Client Class	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. The client class will not have any methods.
Compile Client Class	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.
Inspect All Instances	Performs a server allInstances operation, and opens a browser on the results. Prompts for confirmation.
Inspect Class	Open an inspector on the class.

Categories Pane and the Category Menu

The Categories pane displays a list of the method categories that are in the selected class. Instance and Class tabs control whether the instance method categories or class method categories are displayed. Selecting a category displays the methods in that category in the Methods pane.

In addition to the Instance and Class tabs, there is an Instance Variable tab that allows you to view a list of the instance variables in that class, rather than Categories. Selecting an instance variable displays the methods that directly reference that instance variable.

The following menu items are in this pane’s pop-up menu and under the menu bar **Category** menu.



File Out As...	Prompts you for a file name under which to save all methods of the selected category.
Browse	Open a Category Browser on the selected Category.
Add...	Open a dialog allowing you to create a new category.
Rename As...	Open a dialog to enter a new name for the selected category.
Remove...	Remove the selected category and all methods contained in it.
Find Method...	Open a dialog allowing you to find a method within the class of the category.
Compile Client Protocol	Compile all methods in the selected category on the client class that is connected to the currently selected class.

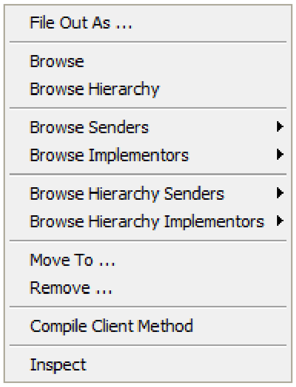
Methods Pane and the Method Menu

The Methods pane displays a list of methods in the selected category, or if the Instance Variable tab is selected, the list of methods that reference that instance variable.

When you select a method, its source code is displayed in the lower portion of the browser, the text 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.

You may select multiple methods, or drag a selected method or methods to another category.

The following menu items are available on the methods pane pop-up menu and on the **Method** menu.



File Out As...	Prompts you for a file name under which to save the selected method or methods.
Browse	Open a Method Browser on the selected method.
Browse Hierarchy	Open a Hierarchy Browser on the class of the selected method.
Browse Senders	Open a Senders Method List Browser on all methods that send the selector of the method or any of the messages it sends.
Browse Implementors	Open a Implementors Method List Browser on all methods that implement the selector of the method or any of the messages it sends.
Browse Hierarchy Senders	Open a Hierarchy Senders Method List Browser on all methods within the hierarchy of the selected class that send the selector of the method or any of the messages it sends.
Browse Hierarchy Implementors	Open a Hierarchy Implementors Method List Browser on all methods within the hierarchy of the selected class that implement the selector of the method or any of the messages it sends.
Move To...	Open a dialog to select the name of an existing category to which to move the selected method or methods, or allowing you enter a new category.
Remove...	Remove the selected method or methods.
Compile Client Method	Compile the selected method on the client class that is connected to the currently selected class.
Inspect	Open an inspector on the method or methods.

Text Pane

The Text Pane displays source code or class information, depending on the selection of the Class View Tabs.

If any changes are made in the text in the Text pane, the Code Modification Indicator turns on (becomes red), to alert you to the unsaved changes. Changing tabs or navigating off of a pane with Code Modification Indicator on prompts you to save or discard your work.

Text View Tabs

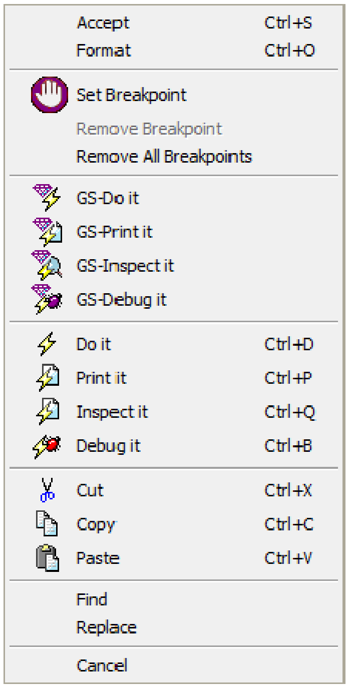
When a method is selected, the Source tab is selected and the method source is displayed. You can use the other tabs to access other information about the class. When no method is selected and a class is selected, the class definition will be displayed.

The available tabs display:

Source	Displays source code for a selected method, or the method template if no method is selected. If any changes are made to this method source, the Text Modification Indicator on that tab becomes red.
Definition	Displays the class definition for the class. If any changes are made to this definition text, the Text Modification Indicator on that tab becomes red.
Hierarchy	Displays the hierarchy of the class
Comment	Displays the class comment, if any. Accept saves the class comment according to the rules for the particular server version you are logged into. If any changes are made to this comment, source, the Text Modification Indicator on that tab becomes red.

The Text pane has a pop-up menu, which provides commands similar to that of the corresponding menu in the client Smalltalk browser's text pane; i.e. **Accept** and **Format** have the same behavior as in the client Smalltalk, as does **Cut**, **Copy**, and **Paste**, and so on.

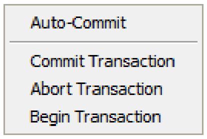
The following GemStone-specific menu items are also available on the text pane pop-up menu:



Set Breakpoint	Insert 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 "Breakpoints" on page 176 for a further discussion of using breakpoints, and the Breakpoint Browser.
Remove Breakpoint	Remove selected breakpoint.
Remove All Breakpoints	Remove all breakpoints in the method.
GS-Do it	Executes the selected code in GemStone.
GS-Print it	Executes the selected code in GemStone, and displays the result in the text pane.
GS-Inspect it	Executes the selected code in GemStone and opens an inspector on the result.
GS-Debug it	Opens a GemStone Debugger on the selected code, allowing you to step through it in the Debugger.

Session Menu

The **Session** menu provides convenience methods for managing the session that opened this browser. The operations on this menu apply to all browsers open for that session, not just the current browser.

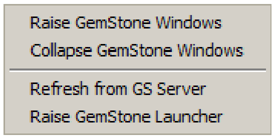


Auto-Commit	Turn on or turn off auto-commit for this session.
Commit Transaction	Attempts to commit modifications to the repository that occurred during the current GemStone transaction.
Abort Transaction	Undoes all changes that you have made in the repository since the beginning of the current transaction. You are asked to confirm.
Begin Transaction	Undoes all changes that you have made in the repository since the beginning of the current transaction, and begins a new Transaction.

Window Menu

The Window menu provides options that operate on the given browser, or on all GemStone windows.

The following items are on the Windows menu.



Raise GemStone Windows

Expands all minimized GemBuilder for Smalltalk windows.

Collapse GemStone Windows

Minimizes all GemBuilder for Smalltalk windows, except the GemStone Launcher.

Refresh From GS Server

Update the current window with any changes to be consistent with the current view from the GemStone Server. This does not commit or abort, so no new committed changes from the GS Server are visible. This is valuable when you are modifying code in multiple browsers.

Raise GemStone Launcher

Opens the GemStone Launcher if it is not already open, and brings it to the front.

Status Bar

The status bar, along the bottom of the window, keeps you updated on the specific Symbol Dictionary, Class, Protocol and Method you are viewing. The specific display will depend on the selections in your browser.

The Status Bar is particularly useful for browsers other than the System Browser that allow you to browse individual classes, protocols, methods or lists of methods. In these browsers, described in the next sections, the class or protocol may not be otherwise visible in the browser.

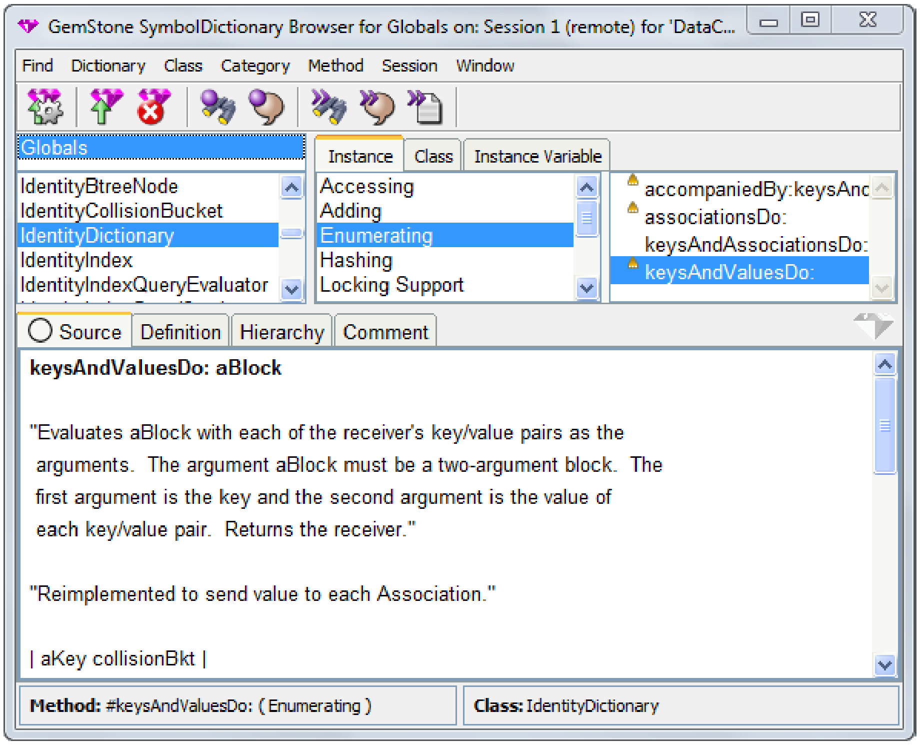
Other Code Browsers

There are a number of other browsers available that allow you to browse a subset of the code that is visible in the System Browser. These behavior of these browsers and the available menu operations are the same as the equivalent pane or panes in the System Browser; refer to the earlier sections in this chapter for details.

SymbolDictionary Browser

The SymbolDictionary Browser allows you to browse all classes and code within a single SymbolDictionary. Menu items and behavior are similar to the System Browser when a SymbolDictionary is selected.

Figure 12.1 SymbolDictionary 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 |
```

Method: #keysAndValuesDo: (Enumerating)

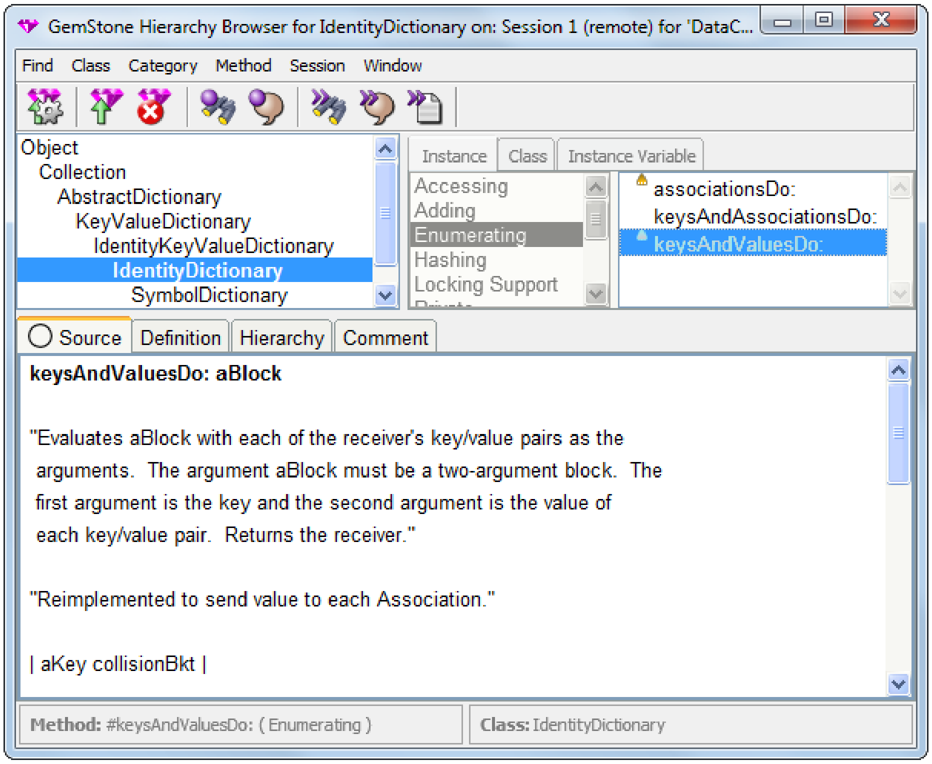
Class: IdentityDictionary

Hierarchy Browser

The Hierarchy Browser lets you browse a class and its super and subclasses.

This browser is reached by double-clicking on a class in the System Browser, or by selecting a class and using the Class menu item "Browse Hierarchy" or the Method menu item "Browse Hierarchy". Menu items and behavior are similar to a System Browser when a Class is selected.

Figure 12.2 Hierarchy 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 |
```

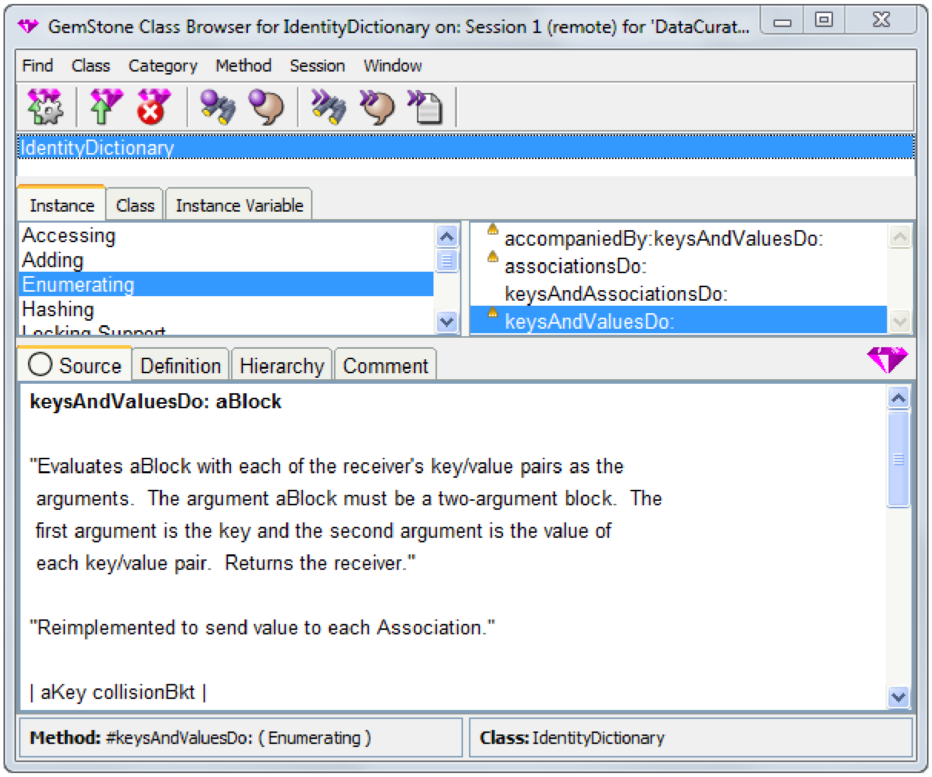
Method: #keysAndValuesDo: (Enumerating)

Class: IdentityDictionary

Class Browser

The Class Browser allows you to browse the class definition and all methods for a single class. This browser is reached by selecting a class and using the Class menu item "Browse Class". Menu items and behavior are similar to the System Browser when a Class is selected.

Figure 12.3 Class 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 |
```

Method: #keysAndValuesDo: (Enumerating)

Class: IdentityDictionary

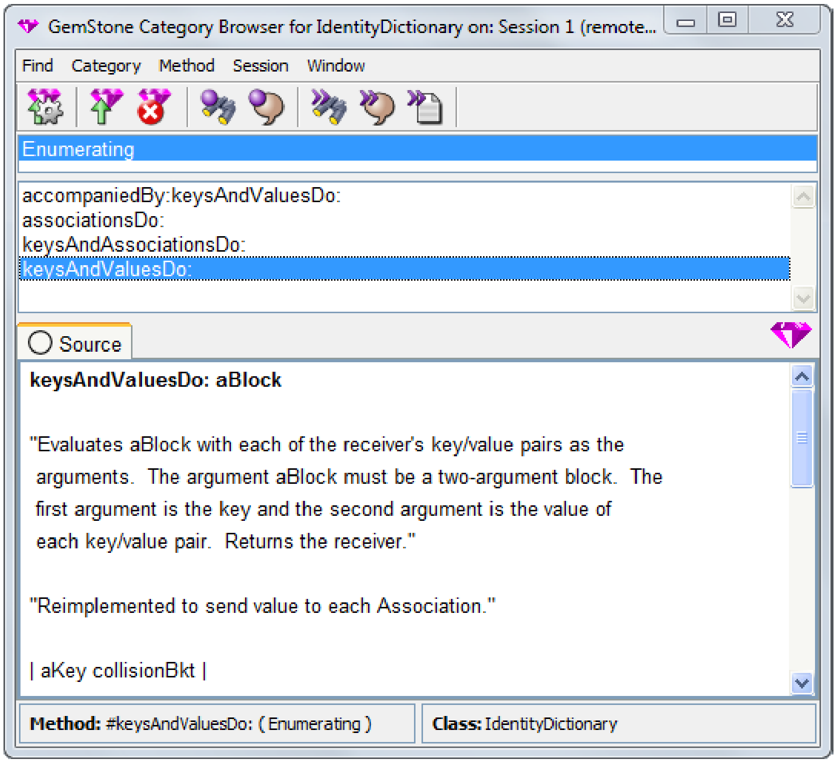
Category Browser

The Category browser allows you to browse all methods within a single method category.

The Category Browser is reached by double-clicking on a category in the System, Class, or Hierarchy Browser, or by using the **Category** menu item “Browse”.

Menu items and behavior are similar to the System Browser when a Category is selected.

Figure 12.4 Category Browser


GemStone Category Browser for IdentityDictionary on: Session 1 (remote...
Find
Category
Method
Session
Window
Source

Enumerating

accompaniedBy:keysAndValuesDo:
associationsDo:
keysAndAssociationsDo:
keysAndValuesDo:

```
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 |
```

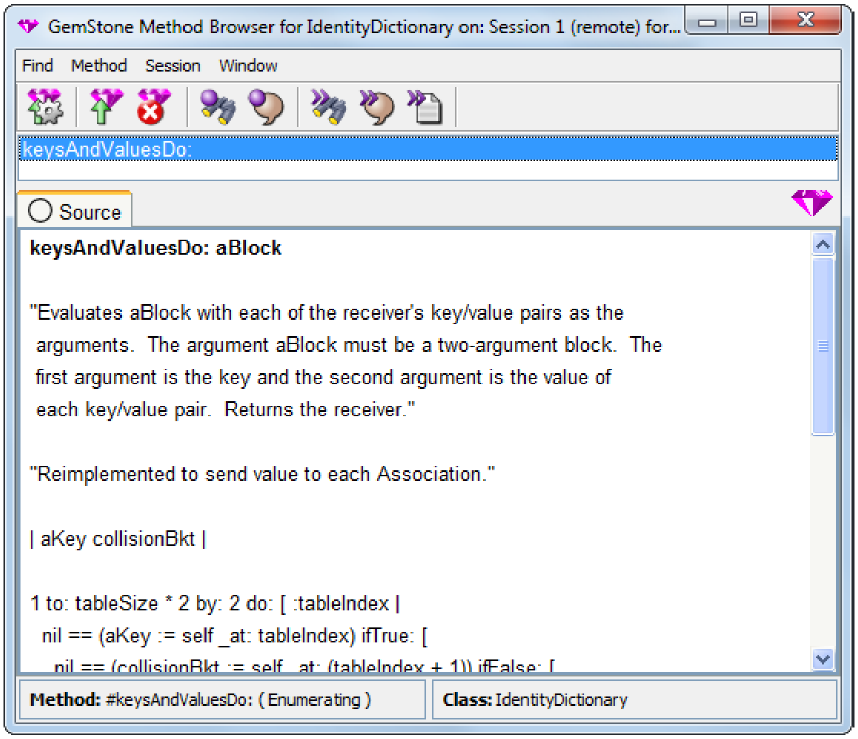
Method: #keysAndValuesDo: (Enumerating)

Class: IdentityDictionary

Method Browser

The Method Browser lets you view and edit a single method. Menu items and behavior are similar to the System Browser when a Method is selected.

Figure 12.5 Method Browser


GemStone Method Browser for IdentityDictionary on: Session 1 (remote) for...
Find
Method
Session
Window
keysAndValuesDo:
Source

```
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: [
```

Method: #keysAndValuesDo: (Enumerating)

Class: IdentityDictionary

12.2 Adding and Modifying Classes and Methods

This section explains how to define new GemStone classes and methods, and describes aspects of coding unique to GemStone Smalltalk. Only a brief summary is provided here; the detailed aspects of coding in GemStone Smalltalk are described in the *GemStone/S Programming Guide*.

Classes in GemStone Smalltalk

Classes in GemStone Smalltalk are similar to classes in client Smalltalk. There are a number of more advanced attributes for classes, provided as keywords during class creation. These are described in the *GemStone/S Programming Guide*.

Subclass creation messages

There are a variety of subclass creation messages, depending on the type of class you want to create. The set of subclass creation messages varies for different server products and versions. The class creation template displayed in the browsers will be appropriate for the currently logged in GemStone server version. For more details on subclass creation methods, see the instance methods on Class in the GemStone server, or refer to the *GemStone/S Programming Guide*.

Versioning

When you redefine a class in GemStone, it does not modify the class; instead a new version of the class is created, linked to the original class via a class history. This is described in more detail in “Modifying an Existing Class: Class Versions” on page 162.

Modifiable classes

Classes may be created as modifiable or not modifiable. Classes that are modifiable can be modified, e.g. you may add and remove instance, class, and class instance variables; but you cannot create instances of a modifiable class. Before you can create instances of a modifiable class, you must send the message `#immediateInvariant` to the class, to make it non-modifiable.

To make a class modifiable, use the `#modifiable` keyword in the `options:` array, or check the **Modifiable** checkbox in the Add GemStone Class dialog.

Defining a New Class

You may define a new GemStone class using the Add Class dialog or by using the template provided in the Code Browser Text pane.

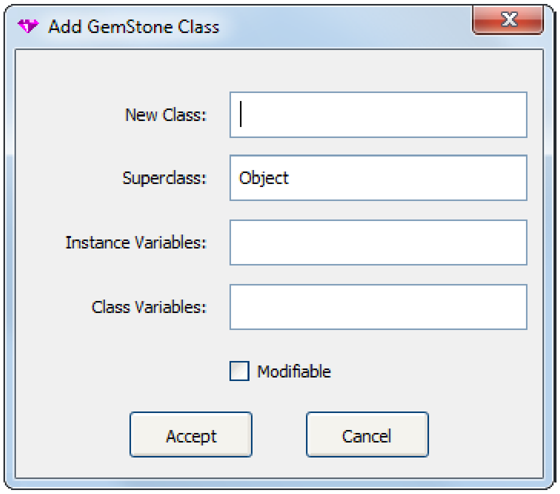
When you add a new class, if there is an existing class with the same name in the same SymbolDictionary, your new class will be a version of the existing class. In this case, a dialog will ask if you want to commit your changes and migrate instances.

If your new class has the same name as an existing class in a different SymbolDictionary, then both classes will exist, unrelated. References to the class’s name will resolve to the class in the SymbolDictionary that is higher in the SymbolList. Note that this resolution is at compile time, not runtime. For more information on symbol resolution, see the *GemStone/S Programming Guide*.

Add Class Dialog

The Add Class Dialog, accessed from the menu item **Class > Add**, allows you to create a class in the currently selected SymbolDictionary. It prompts you for the name of the new class, the superclass, and instance and class variables.

Figure 12.6 Add Class Dialog



Superclass: Object

Using the Template

The Class creation template is displayed in the System Browser when a SymbolDictionary is selected but no class is selected, or in the SymbolDictionary, Class, and Class Hierarchy Browsers when no class is selected.

The browser displays the class definition template, which will vary between server products and versions. For example, in GemStone/S 64 Bit 3.x:

```
NameOfSuperclass subclass: 'NameOfClass'
      instVarNames: #() "example: 'instVar1' 'instVar2' "
      classVars: #() "example: 'ClassVar1' 'ClassVar2' "
      classInstVars: #() "example: 'classIvar1' 'classIvar2' "
      poolDictionaries: {}
      inDictionary: SelectedSymbolDictionary
      options: #()
```

You may replace this template with another class creation method, if you prefer; for example, if you want to create an indexable or byte format class.

Edit the template to replace `NameOfSuperclass` and `NameOfClass` with your desired superclass and class names, and provide instance, class and class instance variable and pool dictionaries that you want your class to include.

SelectedSymbolDictionary determines which Symbol Dictionary will contain the new class; by default, the selected Symbol Dictionary.

The array following the `options:` keyword allows you to specify symbols to define specific features of the new subclass. This particular keyword is only available in class creation protocol in GemStone/S 64 Bit 3.x; in earlier products and versions, other class creation protocol is used to define the available class features. For details on these options, see the *Programming Guide* for your server product and version.

Modifying an Existing Class: Class Versions

If you select an existing GemStone Smalltalk class, then modify and save the class definition within the same symbol dictionary, you create a new version of that class and all of its subclasses. This is discussed in Chapter 8, “Schema Modification and Coordination”, starting on page 101. 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 are recompiled.

You can only modify classes for which you have write authorization. When running as a user other than SystemUser, you cannot modify GemStone kernel classes.

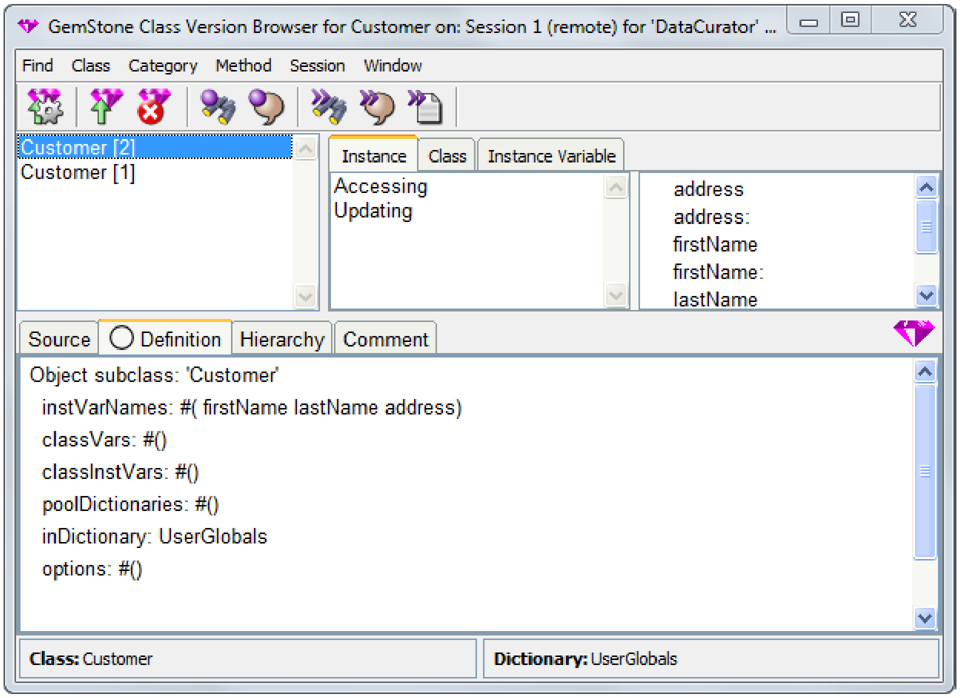
When you modify an existing class, in addition to recompiling the class and its subclasses, the tools will ask if you wish to commit the transaction and migrate all instances of the selected class to the new version of the class. If you choose not to do this, you can later explicitly migrate some or all instances of one version of a class to another version. Migration is described in detail in the *Programming Guide* for your server version and product.

Class Version Browser

Classes that are versions of each other are linked by a shared classHistory. Only the latest version is displayed in the browsers; the display includes the sequence number of the class within the class history, in brackets; for example if you have two versions of Customer class, the latest one will appear as `Customer [2]`.

You can examine and manage the versions of a class using the Class Version Browser. To open a Class Version Browser, select a class in a browser and choose **Browse Versions** from the **Class** menu. If more than one version of a class has been created, the Class List in the spawned Class Version Browser displays the version number next to the class name.

Figure 12.7 Class Version Browser



```
Object subclass: 'Customer'
  instVarNames: #( firstName lastName address)
  classVars: #()
  classInstVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  options: #()
```

The Class Version Browser's menus are similar to the menus for the Class Browser, with a few exceptions. In the Class Version Browser, the Class menu includes the additional items **Make Current** and **Migrate Instances**; the Classes menu items **Move To...** is not available, and **Remove...** has more limited scope.

Remove...	Remove the selected class version from this classes' class History.
Make Current	Makes the selected class version to be the current class version.
Migrate Instances	Migrate all instances of the selected versions. Prompts you to select which version to migrate to. The user can only migrate to another version of the same class history, so if all versions are selected there is no migration destination and the item should be grayed out. Otherwise, prompt for the version to migrate to by popping up a list of versions not selected. Allow the user to cancel the operation by clicking a cancel button.

Adding and Modifying Methods

To add a method, select the Class, then select or create a Category for the new method. This displays the add method template. If you do not select a Category, the new method will be placed in the "unknown" category, which will be created if necessary.

Adding a method to a class, or modifying a method for a class, requires that you have write authorization to that class.

When running as a user other than SystemUser, you cannot add methods nor modify methods for any of the GemStone kernel classes (the predefined classes supplied with the GemStone system).

Public and Private Methods

GemStone includes both public and private methods. *Public* GemStone methods are fully supported with the specified behavior. *Private* GemStone methods are those implemented to support the public protocol; they are not supported, and are subject to change without notice.

Private GemStone methods are those whose selector is prefixed with an underscore (_), that explicitly say they are private within the method comment, or that are in a category labeled Private. 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. 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 GemStone Smalltalk compiler optimizes certain frequently-used selectors. These selectors cannot be overridden in subclasses; the optimized code ignores any redefinitions. Some examples are `==`, `ifTrue:`, and `to:do:`.

The specific list of selectors will vary by GemStone server product and version, and can be found in the *GemStone Programming Guide* for that version, Appendix A.

12.3 Fileout and Filein of Class and Method Definitions

It is 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.

File Out

To save GemStone code in a file, use any of the GemStone browser's **File Out As...** or **File Out Methods As...** menu items.

You may file out all classes in a symbol dictionary, all methods without the class definitions within a symbol dictionary, a single class with its methods, just the methods of a single class, all methods within a single category, or a single method.

File In

To read and compile a saved file, use the File Browser menu item **Gs-File In** or **GS-File it in**, or a Workspace **GS-File it in** menu item.

Class definitions and methods that were filed out from a particular product and version may or may not file into a different GemStone product and version. You may need to edit the fileout text file if you wish to file in code from a different product or version.

Handling Errors While Filing In

If one of the modules 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.

Fileout text format

Saved GemStone files are written as sequences of Topaz commands. Example 12.1 shows a class definition in Topaz format. The details may vary according to the specific server version you are using.

Example 12.1

```
doit
Object subclass: 'Address'
  instVarNames: #( street zip)
  classVars: #()
  classInstVars: #()
  poolDictionaries: #()
  inDictionary: UserGlobals
  options: #()
%

! Remove existing behavior from Address
doit
Address removeAllMethods.
Address class removeAllMethods.
%
set compile_env: 0
! ------------------- Class methods for Address
! ------------------- Instance methods for Address
category: 'Accessing'
method: Address
street
   "Return the value of the instance variable 'street'."
   ^street
%
category: 'Updating'
method: Address
street: newValue
   "Modify the value of the instance variable 'street'."
   street := newValue
%
category: 'Accessing'
method: Address
zip
   "Return the value of the instance variable 'zip'."
   ^zip
%
category: 'Updating'
method: Address
zip: newValue
   "Modify the value of the instance variable 'zip'."
   zip := newValue
%
```

GemBuilder's filing out and filing in facilities are intended for saving and restoring classes and methods without manual intervention. The Topaz fileout format, however, has further options for scripting. It is possible to create custom files that include commands to commit transactions, perform initialization, and to create and manipulate objects other than classes and methods. If you want to perform such tasks, refer to the *Topaz Programming Environment* for your server product and version.

The GemBuilder filein mechanism cannot execute the full set of Topaz commands.

Filein is limited to the following subset:

- `category`
- `classmethod`
- `commit`
- `doit`
- `env`
- `method`
- `printit`
- `removeAllMethods`
- `removeAllClassMethods`
- `run`

The GemBuilder file-in mechanism accepts the following commands, but does not execute them. In some cases, it adds a note to the System Transcript that the command is ignored:

- `display`
- `expectvalue`
- `fileformat`
- `level`
- `limit`
- `list`
- `omit`
- `output`
- `remark`
- `status`
- `time`
- `set sourcestringclass String`

If GemBuilder encounters any other Topaz commands it stops reading the file and displays an error notifier.

The filein 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.

Filein/fileout and Characters outside the ASCII range

Fileout of server can be done conveniently from the GBS tools. The GemStone server also supports fileout directly, using topaz or by invoking the fileout methods on Behavior class.

When filing out server code using the GBS tools, VisualWorks normally will automatically encode the file as UTF-8, and interpret input files as UTF-8. Traditionally, fileout from the GemStone server is encoded in 8-bit ASCII, limited to Characters with codePoints in the range 0..255.

Code containing Characters with codepoints greater than 127, that is filed out from topaz as 8 bit ASCII, should not be filed in using GBS tools.

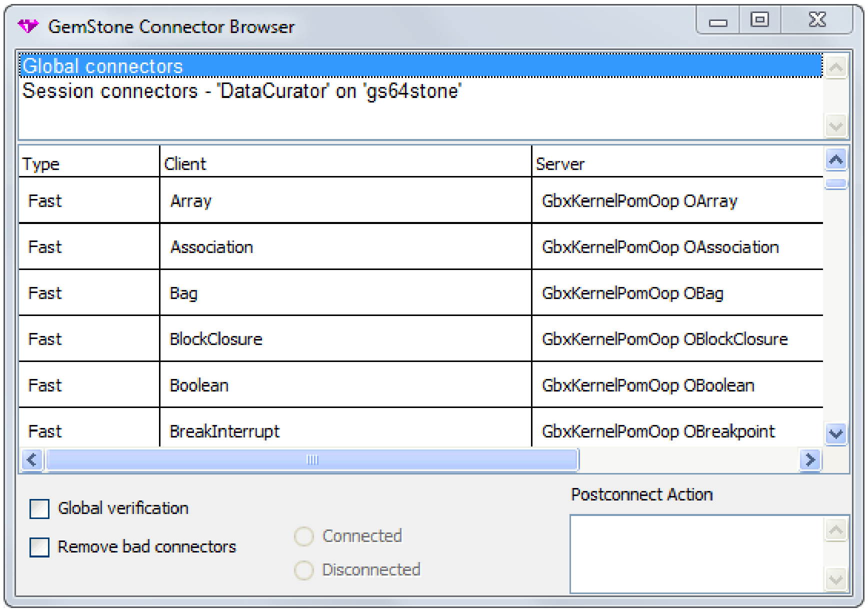
12.4 Connector Browser

Chapter 4 describes *connectors*, which allow an application developer to explicitly declare an association between a client class and a server class or between a root client object and a root server object. This section explains how to use GemBuilder’s Connector Browser to make and manage connectors interactively.

To open a Connector Browser, select **Browse > Connectors** from the GemStone Launcher. With this browser, you can:

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

Figure 12.8 Connector Browser



Global connectors
Session connectors - 'DataCurator' on 'gs64stone'

Type	Client	Server
Fast	Array	GbxKernelPomOop OArray
Fast	Association	GbxKernelPomOop OAssociation
Fast	Bag	GbxKernelPomOop OBag
Fast	BlockClosure	GbxKernelPomOop OBlockClosure
Fast	Boolean	GbxKernelPomOop OBoolean
Fast	BreakInterrupt	GbxKernelPomOop OBreakpoint

☐ Global verification
☐ Remove bad connectors
○ Connected
○ Disconnected

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 parameters. Global connectors are predefined to connect the GemStone server 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 pop-up menu provides the following items:

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 server Smalltalks. In the Connector pane, the popup menu offers the following items. Note options other than add... are only available when a connector is selected.

inspect Client	Resolves and inspects the client Smalltalk object for the selected connector.
inspect Server	Resolves and inspects the GemStone server object for the selected connector.
add...	Adds a new connector, prompting for required information.
remove...	Removes a connector, after confirmation.

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.

The control panel has the following check boxes, radio buttons and selection lists:

Global verification	Checking this sets the configuration parameter connectVerification to true. See “connectVerification” on page 123 for details. We recommend that you turn on verification during development and turn it off for production systems.
Remove bad connectors	Checking this sets the configuration parameter removeInvalidConnectors to true. See “removeInvalidConnectors” on page 128 for details.
Connected / Disconnected	Displays the connected status, and allows you to connect or disconnect the GemStone and client Smalltalk objects described by the connector. Applies to the current session.
Postconnect Action	The postconnect action determines how GemBuilder sets the initial state of connected objects. Options are: **updateST** initializes the client object using the current state of the GemStone server object. **updateGS** initializes the GemStone server object using the current state of the client object.

forwarder makes the client object a forwarder to the GemStone server object.

clientForwarder makes the GemStone server object a forwarder to the client object.

none leaves the client object and the GemStone server object unchanged after their initial connection. This is the default and recommended setting for class connectors.

Connector Operations

Creating a new connector

1. Select a Global or a session parameters in the Group pane.
2. In the Connector (center) Pane, select **add** from the pop-up menu.
3. When prompted, specify the type of connector.
4. When prompted, specify the names of the client and server objects.
5. When prompted, specify the name of the dictionary for the server object.
6. Specify the postconnect action.

Creating a forwarder

1. Create a connector as described above.
2. Select **forwarder** as the desired postconnect action.

Changing 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.

Transfer data storage from client to server

If your application initially stores its data in the client, and you intend to store the data on the GemStone server 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 the GemStone server so that GemBuilder can create the GemStone server replicates for the client Smalltalk data.
4. Inspect the GemStone server objects to be sure they have the intended values.
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 server data.

Chapter 13

Inspecting and Debugging in GemStone

The GemStone Smalltalk Debugging Tools allow you to inspect objects in both the GemStone and client object spaces, and to set breakpoints in and debug GemStone Smalltalk code, similarly to how you debug client Smalltalk code. GemBuilder's Debugger and Inspectors are integrated with the client Smalltalk debugger to allow seamless debugging of code in both object spaces.

This chapter describe the differences and additions that GemBuilder makes to the client Smalltalk debugging tools.

Inspectors
describes how to view and modify the instance variables of server objects

Debugger
describes GemBuilder's enhanced debugger

Breakpoints
describes step points and breakpoints, and the Breakpoint Browser

Stack Traces
describes GbsStackDumper, GemBuilder's enhanced stack dumping facility

13.1 Inspectors

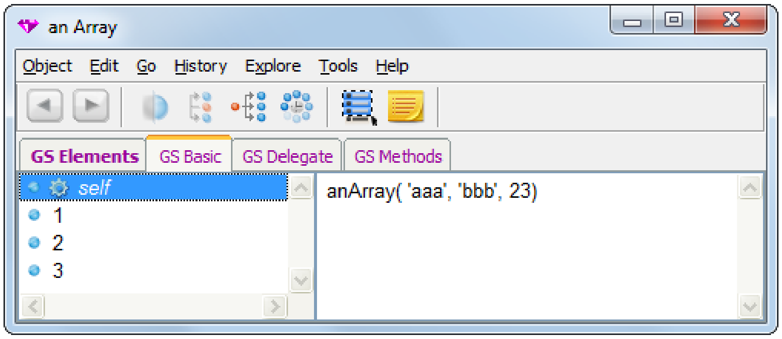
To allow you to examine the values of GemStone server objects and modify them when appropriate, GemBuilder provides enhancements to the client Smalltalk inspectors. When you select a GemStone Smalltalk expression and execute **GS-Inspect it**, an enhanced client Smalltalk inspector opens on the GemStone server object.

For example, if you select and GS-Inspect it on:

```
Array with: 'aaa' with: 'bbb' with: 23
```

The resulting inspector displays as shown in Figure 13.1.

Figure 13.1 GS Basic Tab

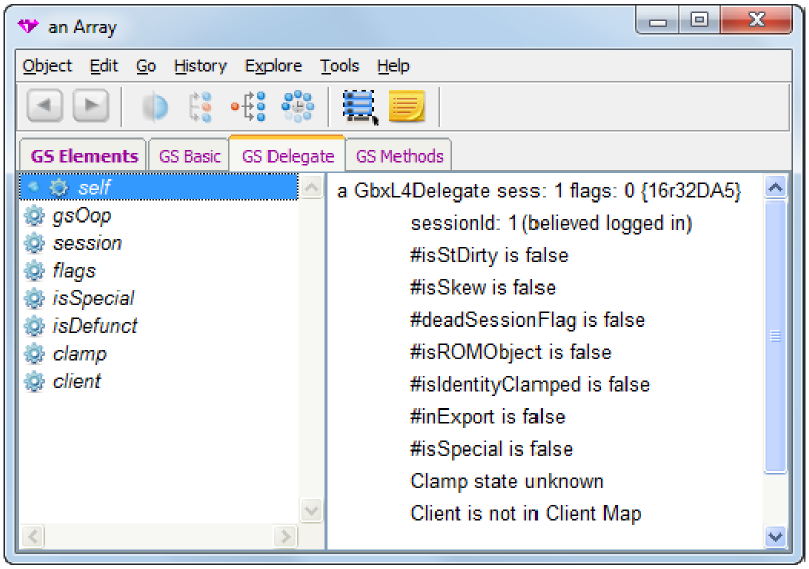


```
anArray( 'aaa', 'bbb', 23)
```

When inspecting a GemStone server object, the inspector provides additional tabs with the GS prefix, indicating that these apply to a GemStone server object.

In addition, there is a tab labeled "GS Delegate", which allows you to examine the internal state of the object's delegate in GemBuilder.

Figure 13.2 GS Delegate Tab



```
a GbxL4Delegate sess: 1 flags: 0 {16r32DA5}
        sessionId: 1 (believed logged in)
        #isStDirty is false
        #isSkew is false
        #deadSessionFlag is false
        #isROMObject is false
        #isIdentityClamped is false
        #inExport is false
        #isSpecial is false
        Clamp state unknown
        Client is not in Client Map
```

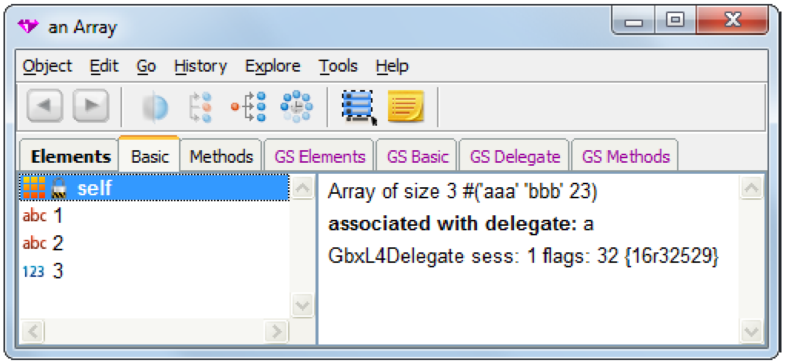
When inspecting a client object that has a corresponding GemStone server object (or vice versa), the inspector allows you to view both the client object and the server object in the same inspector. Client objects with corresponding server objects includes replicates of server objects, forwarders to server objects, stubs, and objects that are always mapped such as nil, true, false, SmallIntegers, and Characters. When you are inspecting one of these objects, you will see two sets of tabs: one that allows you to examine the client object, and a second similar set of tabs with the GS prefix, to examine the associated GemStone server object.

For example, if you select the following text and Inspect it:

```
GBSM evaluate: 'Array with: ''aaa'' with: ''bbb'' with: 23'
```

The following inspector displays both the client and server object:

Figure 13.3 Inspector for Objects that exist on both client and server


an Array
Object Edit Go History Explore Tools Help
Elements Basic Methods GS Elements GS Basic GS Delegate GS Methods

self
abc 1
abc 2
123 3

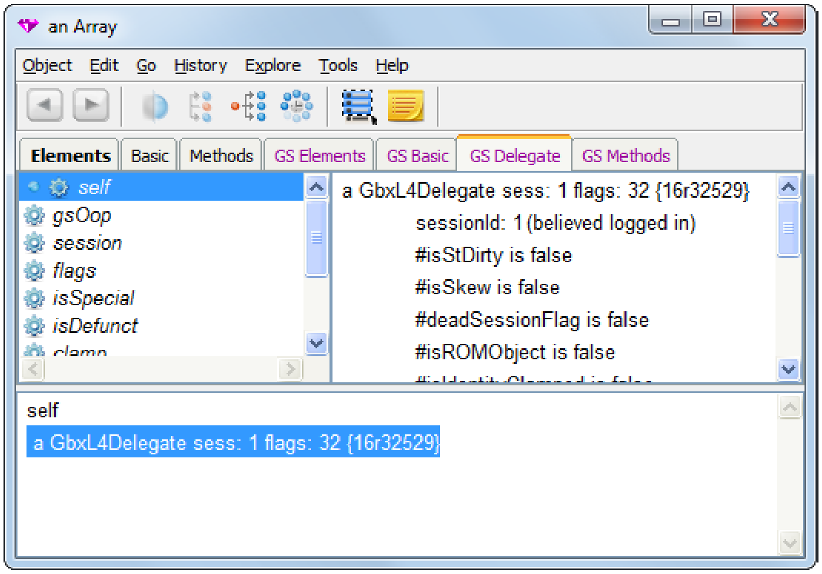
Array of size 3 #('aaa' 'bbb' 23)
associated with delegate: a
GbxL4Delegate sess: 1 flags: 32 {16r32529}

Executing code in the inspector

In the evaluation pane of the inspector, or any other pane, the use of “self” is resolved according to the following rules:

- Do it, Print it, Inspect it and Debug it resolve self to the client object.
- GS-Do it, GS-Print it, GS-Inspect it, and GS-Debug it resolve self to the GemStone server object.
- When the GS Delegate tab is selected, Do it, Print it, Inspect it and Debug it resolve self to the delegate object, as shown in Figure 13.4.

Figure 13.4 Inspector Evaluation Pane.


an Array
Object Edit Go History Explore Tools Help
Elements Basic Methods GS Elements GS Basic GS Delegate GS Methods

self
gsOop
session
flags
isSpecial
isDefunct
clamp

a GbxL4Delegate sess: 1 flags: 32 {16r32529}
sessionId: 1 (believed logged in)
#isStDirty is false
#isSkew is false
#deadSessionFlag is false
#isROMObject is false

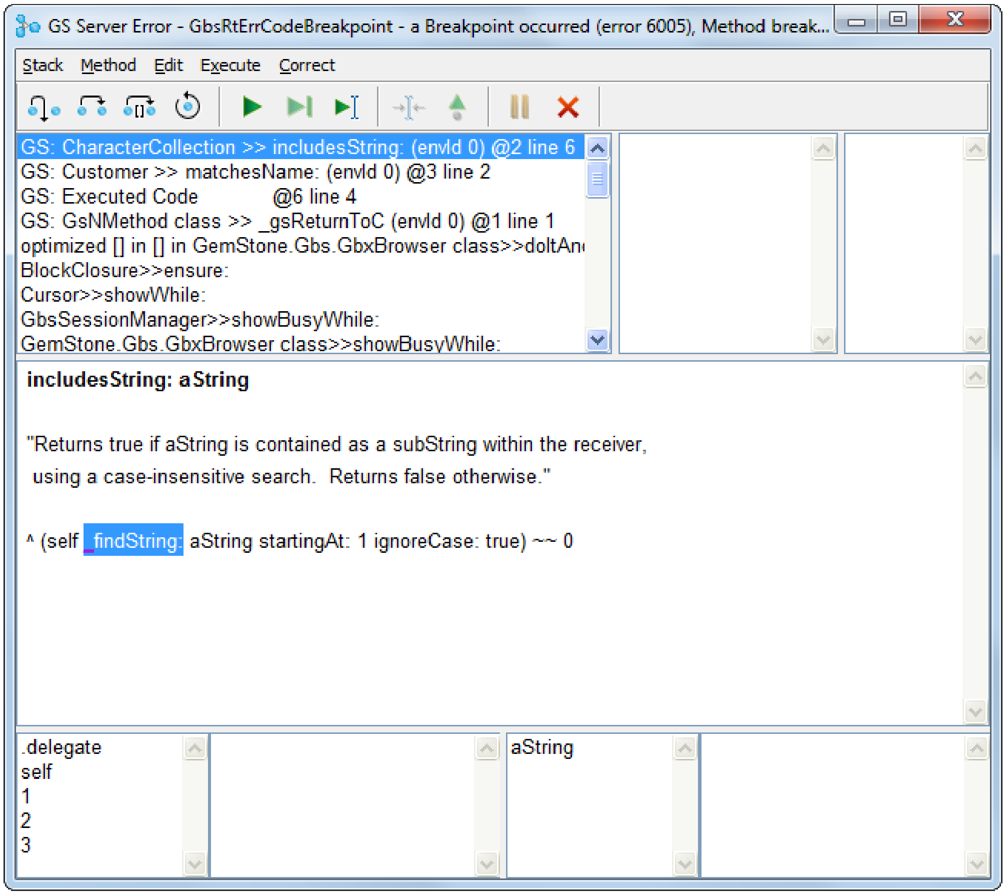
self
a GbxL4Delegate sess: 1 flags: 32 {16r32529}

13.2 Debugger

The GemStone Debugger is integrated with the client Smalltalk debugger, allowing you to step through client Smalltalk code and GemStone server code in the same tool. Using the GemStone debugger, you can:

- view GemStone Smalltalk and client Smalltalk contexts together in one stack
- select a GemStone Smalltalk or client Smalltalk context from among those active on the virtual machine stack
- examine and modify objects and code within that context
- continue execution either normally or in single steps

Figure 13.5 Debugger



GS Server Error - GbsRtErrCodeBreakpoint - a Breakpoint occurred (error 6005), Method break...

```
GS: CharacterCollection >> includesString: (envId 0) @2 line 6
GS: Customer >> matchesName: (envId 0) @3 line 2
GS: Executed Code          @6 line 4
GS: GsNMethod class >> _gsReturnToC (envId 0) @1 line 1
optimized [] in [] in GemStone.Gbs.GbxBrowser class>>doItAn
BlockClosure>>ensure:
Cursor>>showWhile:
GbsSessionManager>>showBusyWhile:
GemStone.Gbs.GbxBrowser class>>showBusyWhile:
```

```
includesString: aString

"Returns true if aString is contained as a subString within the receiver,
 using a case-insensitive search.  Returns false otherwise."

^ (self _findString: aString startingAt: 1 ignoreCase: true) ~~ 0
```

When GemStone Smalltalk execution is interrupted, it either directly opens the Debugger, or a notifier that includes a **Debug** button. Selecting the **Debug** button opens the Debugger. A runtime error opens a notifier, while a breakpoint, user interrupt, or an `Object >> pause` opens a debugger. You may also use the menu item GS Debug-it to open a Debugger on a code snippet, which opens a debugger on that code.

The Debugger's stack pane displays the active call stack and allows you to choose a context (stack frame) from that stack for manipulation in the window's other panes. Both GemStone server and client contacts are listed. GemStone server contexts begin with "`GS`".

The Debugger is much like the client Smalltalk debugger, with the menus enhanced with GemStone options. Like other GemBuilder text areas, the debugger source code pane provides commands to execute GemStone Smalltalk.

Colored contexts

You may configure VisualWorks to color the GemStone server contexts, to distinguish them from client contexts. To do this, select the Settings button on the Launcher toolbar, or go to **System > Settings**, and select **Debugger** (under **Tools**). Select the button to edit the Context List Presentation. If no Patterns exist, create a new Pattern for Everything with the

Pattern String *. Then, create a new Pattern with the Pattern String `$L$GS*`, and select a color for the GemStone server contexts text. This new Pattern must be above the Pattern String *.

The following example demonstrates doing the same action using a script.

Example 13.1 Colored Debugger Contexts using a script

```
| patterns contextPattern|
patterns := OrderedCollection new.
contextPattern := (ClientContextPatternSpec new)
   color: (ColorValue red: 0.2 green: 0.0 blue: 0.5);
   name: 'GemStone';
   pattern: '*$L$GS*';
   yourself.
patterns add: contextPattern.
contextPattern := (ClientContextPatternSpec new)
   color: ColorValue black;
   name: 'Everything';
   pattern: '*';
   yourself.
patterns add: contextPattern.
ContextListPolicyEditor.Patterns := patterns.
```

Note that for Windows-based VisualWorks Look and Feel settings, the selection background color may leave selected context text unreadable when colored contexts are enabled.

Disabling the Debugger

In some cases, you may want to disable the GBS debugger. You can disable and enable the debugger using the following expressions:

```
GBSM enableGbsDebugger
GBSM disableGbsDebugger
```

Disabling the GBS debugger restores the base VisualWorks debugger.

13.3 Breakpoints

Setting a breakpoint in your code allows your code to run up to that point, so you can start debugging there.

Step Points

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.

More detail on step points within GemStone Smalltalk methods is provided in the *Topaz Programming Environment*, Chapter 2.

Example 13.2 indicates step points with numbered carets, as is output in the topaz environment.

Example 13.2 GemStone Server method step points

```
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 to step through this method, the first step takes you to step point 1, the point where `includesValue:` is about to be sent. Stepping again sends that message and halts the virtual machine at step point 2, 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.

Breakpoints

You can set a breakpoint in any method by selecting the position in the source text and menu item **Set Breakpoint**. The step point nearest your cursor position is used for the breakpoint, and the first letter of the text following the breakpoint will be highlighted (colored and underscored).

You can remove the breakpoint by selecting the specific break and using the menu item **Remove Breakpoint**, or by using **Remove All Breakpoints**. Breakpoints can also be removed using the Breakpoint Browser, described in the next section. This tool also allows you to disable breakpoints, but leave them in your methods.

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

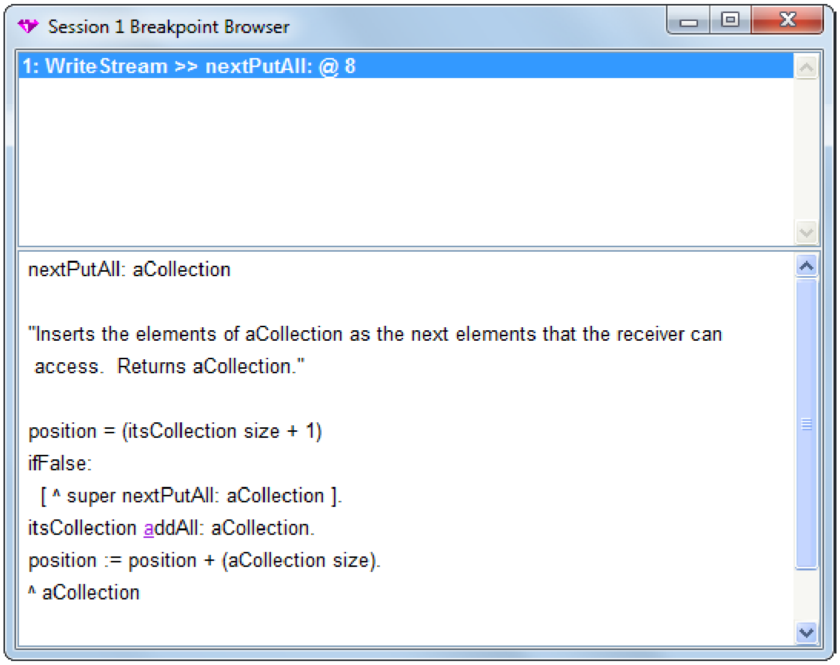
Breakpoint Browser

To see a complete list of the breakpoints you have set anywhere in your GemStone server code, use the Breakpoint Browser. You can open a Break point Browser using the **Browse > Breakpoints** menu item on the GemStone Launcher.

In addition to allowing you to examine, remove and add breakpoints, the Breakpoint Browser allows you to disable breakpoints without removing them, and later re-enable these breakpoints. These operations can be performed on a single selected breakpoint or on all breakpoints.

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 13.6 Breakpoint Browser



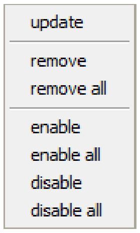
1: WriteStream >> nextPutAll: @ 8

```
nextPutAll: aCollection

"Inserts the elements of aCollection as the next elements that the receiver can
 access.  Returns aCollection."

position = (itsCollection size + 1)
ifFalse:
  [ ^ super nextPutAll: aCollection ].
itsCollection addAll: aCollection.
position := position + (aCollection size).
^ aCollection
```

Breakpoint Pane

The breakpoint pane displays a scrollable list of the active breakpoints. The items in the list include the Class and method name, and the step point within the method. In this example above, a method break is set at step point 8 within the method `nextPutAll:` defined by class WriteStream.

The Breakpoint Pane has the following menu items:



update	Update the list of breakpoints with any recently added breakpoints
remove	Remove this breakpoint.
removeAll	Remove all breakpoints
enable	Enable this breakpoint. Has no effect if the breakpoint was not previously disabled.
enableAll	Enable all breakpoints. Has no effect if no breakpoints were previously disabled.
disable	Disable this breakpoint, but do not remove it. A disabled breakpoint will not cause a break, and displays with (disabled) in the breakpoint browser
disableAll	Disable all breakpoints.

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 has a more limited set of options. You cannot edit and recompile methods in this pane.

13.4 Stack Traces

In some situations it is easier to extract complete stack traces for later analysis, rather than debugging interactively. In addition, you may need a stack trace to provide to GemStone Technical Support. GemBuilder includes facilities to dump the complete stack, with more information than provided in the standard stack, including information on GemStone server contexts and "glue" contexts.

To extract a complete stack, execute

```
GbsStackDumper dumpAllProcessStacks
```

In response, all processes in the image write their complete contexts to a file named `stacksAt`*x*`.txt` in the current working directory, where *x* is a string containing a timestamp. For example, `StacksAtFebruary-22-2013-11.00.20-AM.txt`.

To dump the stacks to a particular file location:

```
GbsStackDumper dumpAllProcessStacksToFileNamed: aString
```

These methods do not require the debugger, and can be used in runtime applications.

13.5 Call Tracing

GemBuilder relies on the GemStone server for much functionality, which it accesses by making GCI calls. You can configure GemBuilder to log each of these calls, which allows debugging of some intractable problems.

Call tracing produces a record of the low-level calls made, and are intended for use by experts who are familiar with the internal operations of the GCI calls. Normally, you would only use Call Tracing when advised by GemTalk Technical Support to diagnose specific problems.

Call tracing may produce a very large amount of output. Call tracing can be done to a file, or to a ring buffer in memory.

The following methods are available to turn on and off call tracing:

`GemStone.Gbs.GbsCallTracing >> startCallTracingToFile`
Turns on call tracing and records the traces to file with a default name, gbstrace-timestamp.log, in the current working directory. If call tracing was already on, a new log file is started.

`GemStone.Gbs.GbsCallTracing >> startCallTracingToFileNamed:` *aString*
Turns on call tracing and records the traces to the specified file. If the file exists, further tracing will be appended to this file.

`GemStone.Gbs.GbsCallTracing >> startCallTracingToMemoryMegabytes:` *aNumber*
Turns on call tracing and records the traces in a ring buffer of the specified size. The contents of the ring buffer can be written to a default file on request.

`GemStone.Gbs.GbsCallTracing >> startCallTracingToMemoryMegabytes:` *aNumber* `fileName:` *aString*
Turns on call tracing and records the traces in a ring buffer of the specified size. The contents of the ring buffer can be written to the specified file on request.

`GemStone.Gbs.GbsCallTracing >> stopCallTracing`
Turns off call tracing, if currently on.

`GemStone.Gbs.GbsCallTracing >> writeToFile`
Writes the current log to its file, if not already done. This is only useful when using a memory logger. It is safe to send this message regardless of the state of call tracing.

`GemStone.Gbs.GbsCallTracing >> logMessage:` *messageString*
If call tracing is active, the message is recorded, either to the log file or the ring buffer.

`GemStone.Gbs.GbsCallTracing >> isCallTracingActive`
Answers whether call tracing has been started and remains active.

`GemStone.Gbs.GbsCallTracing >> isFileCallTracingActive`
Answers whether call tracing has been started to a file, and remains active.

Chapter 14

Using the GemStone Administration Tools

This chapter describes the GemStone tools that are provided to allow you to manage the object sharing and protection issues discussed elsewhere in this manual.

Many of these tasks can also be accomplished by executing code within GemStone, which is more efficient for large repositories. In addition, there are some administration operations that are not supported by the tools, and must be done by executing GemStone server code. See the *System Administration Guide* for the GemStone/S server, for details on both the full set of administrative options and instructions on performing these tasks programmatically.

Many administrative operations involve privileged methods, and may include the requirement to have write authorization to DataCuratorObjectSecurityPolicy. These operations are most often done while logged in as DataCurator. If you do not have authorization to log in as DataCurator, and need to before performing administrative tasks, ask your system administrator to set up your account with the appropriate groups and privileges.

Security Policy Browser

describes a tool for examining and changing GemStone user authorization. Security policies (referred to as Segments in earlier versions) provide the means for managing GemStone authorization at the object level by assigning objects to security policies that have appropriate authorization characteristics.

Symbol List Browser

describes a tool that 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 those dictionaries.

User Account Management Tools

describes the **User List**, the **User Dialog**, and the **Privileges Dialog**, a set of tools that allow you to create new user accounts, change account passwords, and assign group memberships.

14.1 Security Policy Browser

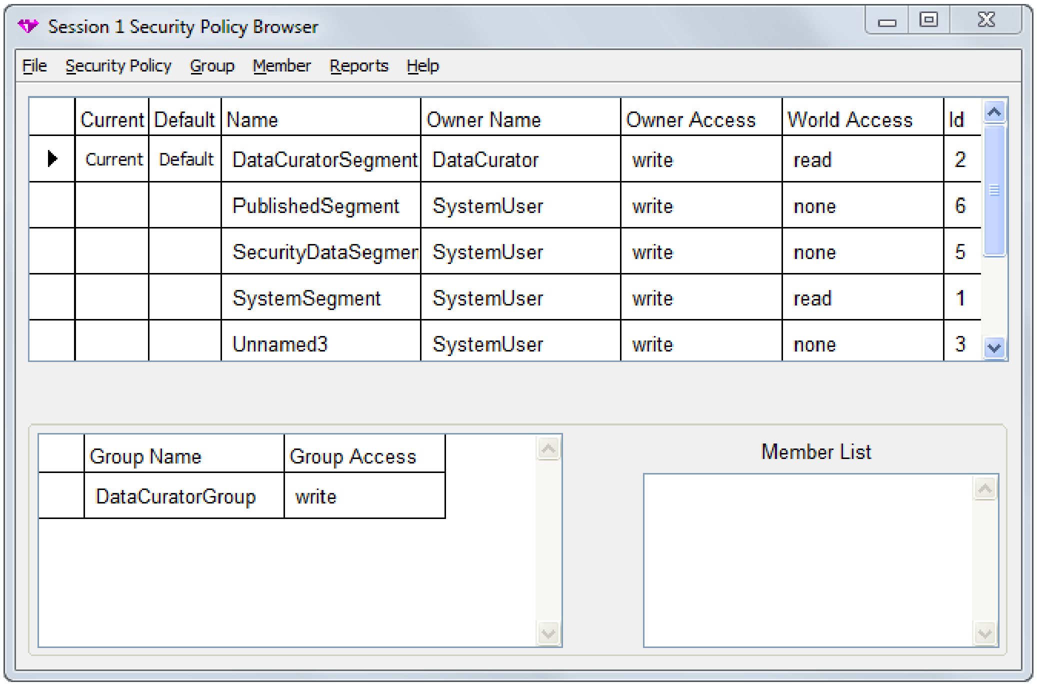
The Security Policy Browser allows you to inspect and change the authorization that GemStone users have at the object level. As explained in the section entitled "Object-level Security" beginning on page 92, each object in GemStone may be associated with an object security policy. The only users authorized to read or modify an object are those who are granted read or write authorization for the security policy with which the object is associated. The Security Policy Browser also allows you to add and configured Security Policies, and examine and change group memberships.

NOTE

In the 32-bit GemStone/S server product, and in GemStone/S 64 Bit 2.x, object security policies are known as Segments.

To open a Security Policy Browser, select **Browse > Security Policies** from the GemStone Launcher or through the User Dialog's **Object Security Policies** button.

Figure 14.1 Security Policy Browser



	Current	Default	Name	Owner Name	Owner Access	World Access	Id
▶	Current	Default	DataCuratorSegment	DataCurator	write	read	2
			PublishedSegment	SystemUser	write	none	6
			SecurityDataSegmer	SystemUser	write	none	5
			SystemSegment	SystemUser	write	read	1
			Unnamed3	SystemUser	write	none	3

	Group Name	Group Access
	DataCuratorGroup	write

The Security Policy Browser is divided into three sections. The upper section displays security policies; the lower left displays groups in a selected Security Policy, and the lower right displays members in the selected group.

Security Policy Pane

The security policy pane at the top of the window displays the security policies in SystemRepository.

You will notice that some security policies are named and some are unnamed. Named security policies are security policies that are referenced in a Symbol Dictionary that you have access to, such as Globals (for named kernel Security Policies). A Security Policy's name can be used to reference it in code, but does not affect it functionally.

In addition to the security policies displayed in the Security Policy Browser, all users also have read and write authorization to GsIndexingObjectSecurityPolicy. Because authorization changes should not be made to that security policy, 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. Entering an invalid value in a cell results in a warning, and the cell reverts to the original value.

In the security policy definition area, the table displays and allows you to edit the following:

Current—You can set the security policy to be your current security policy. When you create any objects in GemStone, the objects is assigned it to your current security policy.

Default—You can set the security policy to be your default security policy. When you log into GemStone, your current security policy is set to this default. If a user does not have write access to their default security policy, they will be unable to log in.

Name—This displays the name of the Object Security Policy, but does not update your dictionaries with any changes.

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. This may remove access for the previous owner, depending on the groups and world access.

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

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

NOTE

Be careful when changing the authorizations on any security policy that a user may be using as a current or default security policy. If the account does not have write authorization in its default security policy, the account cannot log in.

Id—The Security Policy Id number of each security policy is displayed. This information cannot be modified.

Security Policy Menu

Use the **Security Policy** menu bar item, or the security policy pane pop-up menu, to create new security policies and to set current and default security policies for yourself. The following options are available:.

Create...	Creates a new security policy. In the Create Security Policy dialog, enter a name for the security policy and a symbol dictionary in which to create the named reference. Private security policies would usually have the name in that user’s UserGlobals, while those for large groups of users may be kept in Globals or in a the SymbolDictionary that is used by all these users.
Make Current	Makes the selected security policy your current security policy. When you create an object, GemStone assigns it to your current security policy.
Make Default	Makes the selected security policy your default security policy. This is the security policy used for your current security policy when you log into GemStone. Users must have write access to their default security policy in order to log in.

Group Pane

The bottom left of the window is the Group pane. In this area you can assign authorizations to groups of users instead of individuals. Groups are typically organized for users who have common data access requirements.

When you select a security policy in the Security Policy Pane, the group pane displays the groups that have access to the security policy. When you select one of the groups, its members appear in the Member Pane to the right.

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 183.

> *NOTE*
>
> *Be careful when changing the authorizations on a group or removing member. If a user is using the group for access to their default security policy, and the group or user membership is removed or authorization changes, the account cannot log in.*

Group menu

Use the **Group** menu bar item, or the group pane pop-up menu, to add and remove groups from a security policy. The following options are available:

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 security policy 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 security policy unless it has been provided by the security policy's owner or world access.

Member Pane

The bottom right of the window is the Member List. When a group is selected, this displays the members of the group.

Member Menu

Use the **Member** menu bar item, or the Member pane pop-up menu, to add and remove members from a group. The following options are available:

Add...	Adds a user to the group. Enter any valid user name in the Add Member dialog and choose OK. The user must already exist in the system.
Remove...	Removes the selected user from the group. (This does not delete the user from GemStone, only from the group)

Security Policy Tool Menus

The following additional menus are available in the Security Policy Browser.

The File Menu

Use the **File** menu to commit work done in the Security Policy Browser, to abort the transaction, to update the tool's view of security policies, groups, and users in the current session, and to close the Security Policy Browser.

Commit	Commits all the changes made anywhere in GemStone during the current transaction. After you commit, you are given a new, updated view of the repository.
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.
Update	Updates the information in the current window: gives you a new, updated view that reflects the most recent version of the repository.
Close	Close the window. Any changes made are kept, but not committed to the repository.

Reports Menu

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

Group Report	Produces a list of all groups in GemStone and the users in each group.
Security Policy Report	Produces a list of security policies the user has read authorization for and displays information about each one: its owner, its groups, and the access privileges for owner, groups, and world.
User Report	Produces a list of all GemStone users and shows each user's group memberships.

Help Menu

The **Help** menu contains one item.

Session Info	Provides information about the session for the window and about the current session.

Using the Security Policy Browser

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

Checking Security Policy Authorization

To find out who is authorized to read or write that security policy, do the following:

1. Bring up the Security Policy Browser by selecting **Browse > Security Policies** or by choosing **Object Security Policies** in a GemStone User Dialog.
2. Choose **Reports > Security Policy Report**. The resulting list contains all security policies.
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 Security Policy Authorization

Assuming that you have appropriate privileges and access, you can use the Security Policy Browser to change the authorization of a security policy.

The top half of the Security Policy Browser shows the owner, the owner's access, and world access for each security policy in the repository. To change owner or world access for a security policy, click in the corresponding box, then use the pull-down menu to select the 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 security policy, and if it is the user's default security *policy, the user’s session will be terminated and the user will be unable to log in again.*

Controlling Group Access to a Security Policy

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

- Make sure the security policy is selected in the top half of the browser.
- To add a group to the authorization list for the selected security policy, 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 security policy, 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 security policy, 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.

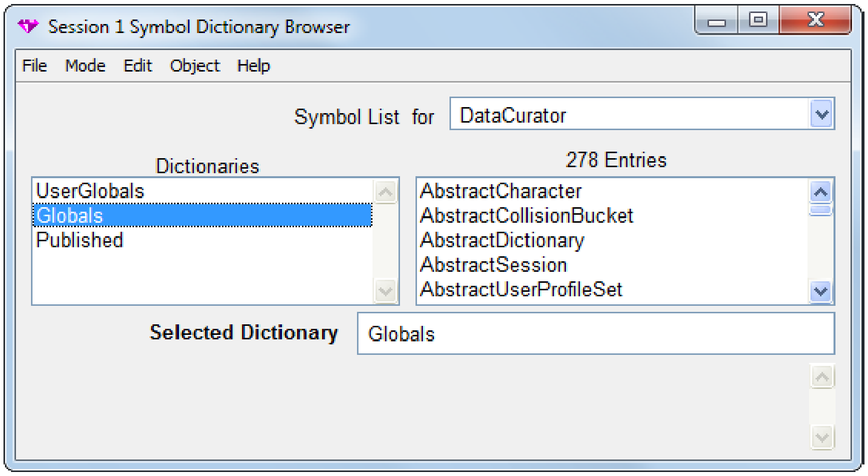
14.2 Symbol List Browser

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 security policy restrictions, to allow users to share application objects and name spaces developed by other users, and to publish them to other users.

To open a Symbol List Browser, select **Browse > Symbol Lists** from the GemStone Launcher, or click on the **Symbol List** button on a User Dialog.

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 14.2 Symbol List Browser



Symbol List for DataCurator

UserGlobals
Globals
Published

AbstractCharacter
AbstractCollisionBucket
AbstractDictionary
AbstractSession
AbstractUserProfileSet

Selected Dictionary Globals

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 server 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.

Commit	Commits all the changes made anywhere in GemStone during the current transaction. After you commit, you are given a new, updated view of the repository.
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.
Update	Updates the information in the current window: gives you a new, updated view that reflects the most recent version of the repository.
Close	Close the window. Any changes made are kept, but not committed to the repository.

Mode Menu

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

Edit Menu

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

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

Object Menu

The Object Menu allows you add a new dictionary, open an inspector on a dictionary entry, and open a browser on a class that is contained in a dictionary.

Add Dict **Add Entry**	Prompts for name of a new item to be added to the Dictionary or Entry list.
Inspect Dict **Inspect Entry**	Opens a GemStone inspector on the selected item.

Browse Class	If the selected entry is a class, opens a GemStone class browser on that entry. Performs the same operation in either Dictionary or Entry mode.

Help Menu

The **Help** menu contains one item.

Session Info	Provides information about the session for the window and about the current session.

14.3 User Account Management Tools

GemBuilder provides three User Account Management tools that allow the GemStone System Administrator to create and modify user accounts, change account passwords, and assign group membership. This section describes these three tools: the User List, the User Editor, and the Privileges Dialog.

NOTE
To perform most of the system administration functions described in this section, you must either be DataCurator or belong to the DataCuratorGroup.

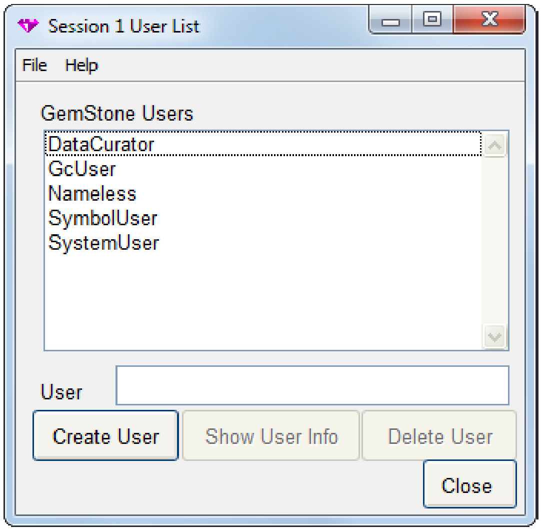
If you are responsible for GemStone system administration, refer the chapter on administering user accounts and security policies in the *GemStone System Administration Guide* for specific information on user account management. That chapter discusses the privileges you need to manage user accounts and 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.

User List

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

- To bring up the User List from the GemStone Launcher, select **Browse > Users**.

Figure 14.3 User List



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

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

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.
Close	Close the browser. Changes are retained in the image, but not committed to the repository.

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

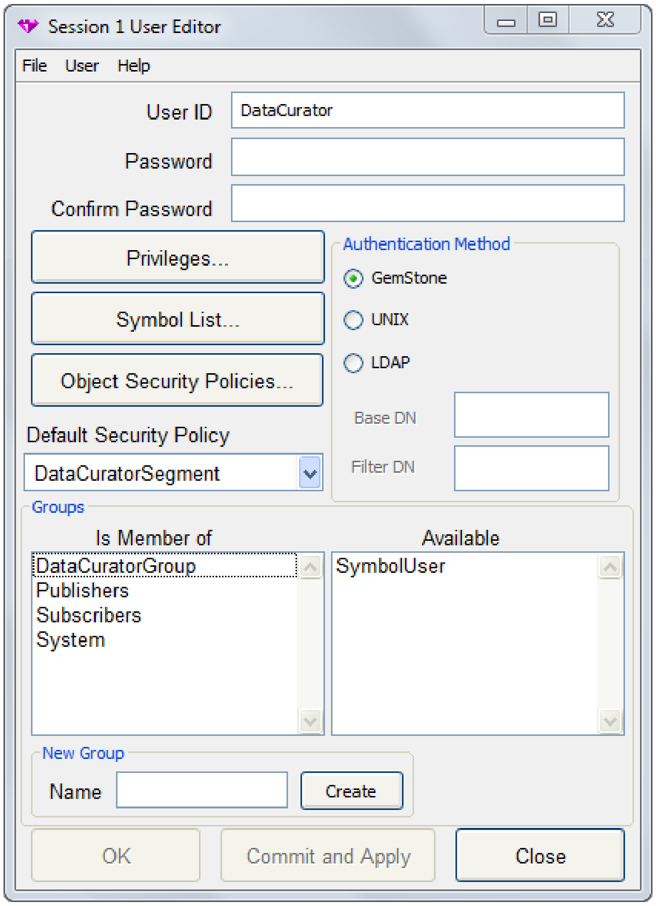
The following buttons are available on this window:

Create User	Brings up a User Editor in which you can define a new user.
Show User Info	Brings up a User Editor displaying privilege and group membership information for the selected user.
Delete User	Allows you to remove the selected user.

User Editor

The User Editor displays the attributes of a particular GemStone user. The GemStone administrator can examine and change the user's privileges or default security policy and can control the user's group membership.

Figure 14.4 User Editor



User ID: DataCurator

Password:

Confirm Password:

Authentication Method: ● GemStone ○ UNIX ○ LDAP

Default Security Policy: DataCuratorSegment

Groups

Is Member of: DataCuratorGroup, Publishers, Subscribers, System

Available: SymbolUser

NOTE
When using 32-bit GemStone/S or GemStone/S 64 Bit v2.x, the User Editor layout is substantially different. However, the basic functions are the same.

The following buttons and operations are available in this window:

User ID	The GemStone User ID that you are viewing details on. When creating a new user, fill in this with the new user's id.
Password	Enter the user's initial password when creating a new user. This is blank for existing users.
Confirm Password	Confirm (re-enter) the user's initial password when creating a new user. This is blank for existing users.

Privileges...	Brings up a Privileges Dialog (page 195), in which you can select privileges for this user.
Symbol List...	Brings up a Symbol List Browser (page 188) for this user.
Object Security Policies...	Brings up a Security Policy Browser (page 182).
Default Security Policy	Allows you to select an existing security policy to be the default security policy for this user.
Authentication Method	Click the button to indicate the method for performing authentication for this user: GemStone userId and GemStone password, UNIX user ID and password, or LDAP server. Authentication other than GemStone is only available in GemStone/S 64 Bit v3.0 and later. For details on configuring authentication, see the chapter on "User Accounts and Security" in the*System Administration Guide*.
Groups	The list on the left is the list of groups that this user belongs to; the list on the right is the list of available groups.

- To **add a user to a group**, select the group in the Available list and drag it to the Is Member Of list.
- To r**emove a user from a group**, select the group in the Is Member Of list and drag it back to the Available list.

New Group Name and Create	In the Name entry box, enter the name of the new group that you wish to create, then click this button. The user is added to the new group.
OK	Makes all changes in the current transaction permanent, and close the dialog.
Commit and Apply	Makes all changes in the current transaction permanent.
Close	Close the dialog. Changes are retained in the image, but not committed to the repository.

File Menu

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

Commit	Commits all the changes made anywhere in GemStone during the current transaction. After you commit, you are given a new, updated view of the repository.
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.
Update	Updates the information in the current window: gives you a new, updated view that reflects the most recent version of the repository.

Close	Close the window. Any changes made are kept, but not committed to the repository.

User Menu

The **User** menu contains one item.

Rename	Prompt for a new name for the user. You may not rename the Server administrator accounts, such has DataCurator, GcUser, or SystemUser. Rename will unset the password for the account.

Help Menu

The **Help** menu contains one item.

Session Info	Provides information about the session for the window and about the current session.

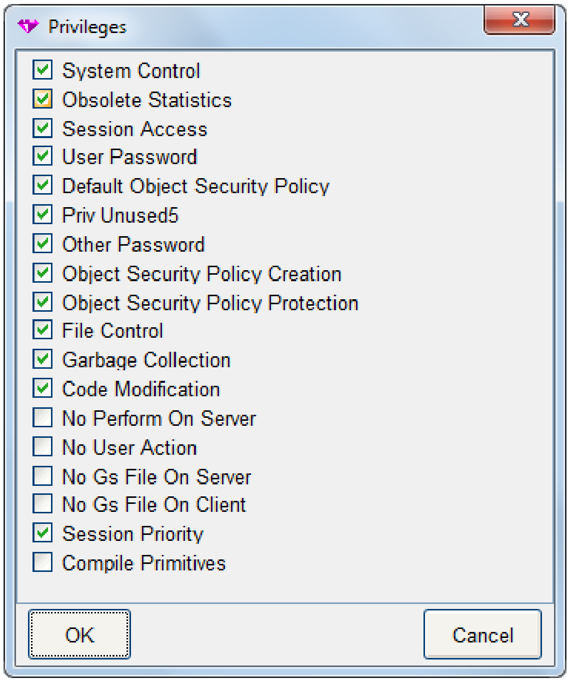
Privileges Dialog

Many functions in GemStone require privileges. For example, in GemStone/S 64 Bit, users require Code Modification before they can create classes and methods. Other privileges control global garbage collection, stopping and starting other sessions, access to external files, and performing operations on the host Operating System. Each user may be granted or not granted these privileges. The specific list of privileges varies slightly by product and version. Refer to the *System Administration Guide* for details on the privileges.

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 modify privileges for a user.

The Privileges Dialog is opened from the User Editor. It is shown in Figure 14.5.

Figure 14.5 Privileges Dialog



☑ System Control
☑ Obsolete Statistics
☑ Session Access
☑ User Password
☑ Default Object Security Policy
☑ Priv Unused5
☑ Other Password
☑ Object Security Policy Creation
☑ Object Security Policy Protection
☑ File Control
☑ Garbage Collection
☑ Code Modification
☐ No Perform On Server
☐ No User Action
☐ No Gs File On Server
☐ No Gs File On Client
☑ Session Priority
☐ Compile Primitives

.

Appendix

A

Application Deployment

GemBuilder is provided in the form of parcels named GbsRuntime, GbsTools, and CstMessengerSupport.

- GbsRuntime and CstMessengerSupport are required for all uses of GemBuilder. Part 1 of this manual documents the functionality provided by GbsRuntime.
- GbsTools contains development and administration tools that are normally used only during development, and are described in Part 2 of this manual. It is almost always desirable to have GbsTools present during development, but GbsTools can be omitted from most deployed applications.

Your client Smalltalk, VisualWorks Smalltalk, provides the ability to package runtime applications, and you should following the packaging instructions for your client Smalltalk version.

A.1 Packaging

Required Parcels

In addition to code that defines your application, the packaged image must contain the parcel GbsRuntime, which contains the system code modified for GemBuilder. You will normally not include GbsTools parcel. In order to ensure that your image initializes correctly, your application must specify GbsRuntime as a prerequisite.

It is recommended to perform the packaging by starting with a new client image and loading the GemBuilder and application code.

If you will be only logging in RPC sessions, set the GemBuilder configuration parameter `libraryName` to the RPC version of the GemStone client libraries.

Packaging options

Names

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

Compiler required for replicating blocks

Replicating blocks requires that the compiler be included in the runtime environment. To ensure that your application is able to replicate Smalltalk blocks in the same manner as it did in the development environment, we recommend that you include the compiler.

A.2 Deployment

Shared Libraries

A deployed runtime application that uses GemBuilder needs to include all the server-specific shared libraries. Linked applications will also need to ensure that the appropriate server components are installed on the client node.

The *GemBuilder for Smalltalk Installation Guide* and the GemStone/S Server *Installation Guide* for the deployment platform provide the specific shared library requirements.

Appendix

B

Client Smalltalk and GemStone Smalltalk

This appendix outlines the few general and syntactical differences between the VisualWorks and GemStone Smalltalk languages.

B.1 Language Differences

GemStone's Smalltalk language is similar to client Smalltalk in both organization and syntax. However, the range of classes and features available, and the class names and internal implementation of basic features like disk file access, varies considerably.

The GemStone class hierarchy is described in the *GemStone Programming Guide*.

SelectBlocks and other Indexing-related notation

Client Smalltalk implements similar ANSI protocol for block select and related operations, using square brackets.

SelectBlocks in GemStone Smalltalk were designed for use with the GemStone indexing framework, and are not present in client Smalltalk. SelectBlocks use curly braces and do not allow message sends other than specific comparison operators.

```
myEmployees select: {:ea | ea.age >= 18}
```

The use of dots for path notation, where each dot indicates that the following is the name of an instance variable on the former, is intended to support SelectBlocks in allowing access to instance variables in the absence of message sends. This has no counterparts in client Smalltalk.

Array Constructors

Array constructors do not exist in client Smalltalk (Both GemStone and client Smalltalk provide the ANSI standard Array literals, using the syntax #(literal literal)).

In GemStone, array constructor syntax varies by server product and version. In GemStone/S 64 Bit version 3.0 and above, array constructors:

- use curly braces,

- use periods as separators,
- have no prefix, and
- can contain any valid GemStone Smalltalk expression as an element.

```
{'string one' . #symbolOne . $c . 4 . Object.new }
```

In 32-Bit GemStone/S and in GemStone/S 64 Bit version 2.x, array constructors:

- use square brackets,
- use commas as separators,
- are prefixed by #, and
- can contain any valid GemStone Smalltalk expression as an element.

```
#['string one', #symbolOne, $c, 4, Object new]
```

See the *GemStone Programming Guide*, Appendix A, for more details on Array constructors.

B.2 TimeZone Handling

The GemStone server, as a multi-user system, may have a number of TimeZones installed, although only one is the current TimeZone for a particular session. The instances of TimeZone include the rules governing such things as the start and end of Daylight Savings Time. GemStone server TimeZones are created based on the Zoneinfo or Olson TimeZone repository.

DateTimes internally store the time in UTC (GMT), but display themselves based on the local current TimeZone. DateTimes do reference an instance of TimeZone, but most server operations use the Gem's current TimeZone.

When server DateTimes are replicated to client TimeStamps, the GbxTimeZone is used to determine the TimeStamp's correct current local time. GbxTimeZone is set to the VisualWorks default TimeZone.

If the desired TimeZone is not the VisualWorks default, or if the GBS application changes the Gem's current TimeZone after a session has logged in, GBS cannot detect this. In this case, the client application needs to send the new message

```
GbsSession >> setClientTimeZoneFromServer
```

to re-replicate a copy of the timezone.

To explicitly set a specific time zone for the client, you can create the desired TimeZone on the server, and replicate it to the client, using the method

```
GbsSession >> clientTimeZone:
```

For example:

```
myGbsSession clientTimeZone:
  (mySession evaluate: 'TimeZone
      fromGemPath:''/foo/bar/America/New_York''').
```

This would be the case if the Gem and client are in different time zones, and you want the timezone to be same between the Gem and client.