GemStone®

GemBuilder for Smalltalk User's Guide

Version 5.4

For use with Instantiations VA Smalltalk

December 2011

vmware®

GEMSTONE S™

INTELLECTUAL PROPERTY OWNERSHIP

COPYRIGHTS

PATENTS

GemStone software is covered by U.S. Patent Number 6,256,637 "Transactional virtual machine architecture", Patent Number 6,360,219 "Object queues with concurrent updating", Patent Number 6,567,905 "Generational garbage collector with persistent object cache", and Patent Number 6,681,226 "Selective pessimistic locking for a concurrently updateable database". GemStone software may also be covered by one or more pending United States patent applications.

TRADEMARKS

VMware, Inc.
15220 NW Greenbrier Parkway
Suite 150
Beaverton, OR 97006

Preface

About This Manual

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

Many GemBuilder features depend on the GemStone/S 64 Bit server, and details of server features vary between server products and versions. Please consult the documentation for the server product and version you are using for specific details for your server product and version.

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 64 Bit Programming Guide*. That book explains the basic concepts behind the language and describes the most important GemStone kernel classes.

In addition, you should be familiar with the VisualAge Smalltalk language and programming environment as described in the VisualAge Smalltalk product manuals.

Finally, 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*.

How This Manual is Organized

This manual is organized in three parts: basic concepts, the GemStone programming tools, and appendixes.

Part 1: Concepts and Programmatic Use

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

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

Chapter 3, Sharing Objects, describes the various mechanisms GemBuilder can use to coordinate your application's local objects with objects in the GemStone object server, thus making them persistent and sharable.

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

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

Chapter 6, Security and Object Access, describes the security mechanisms that are available in GemBuilder and explains how to control access to objects in a multiuser environment.

Chapter 7, Error-handling, discusses errors: how to handle them and how to recover from them

Chapter 8, Schema Modification and Coordination, explains how GemStone supports schema modification by maintaining versions of classes in class histories. It also explains how to synchronize schema modifications between the client and GemStone.

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

Chapter 10, GemBuilder Configuration Parameters, describes the GemBuilder for Smalltalk configuration options and how to set and use them.

Part 2: GemStone Tools

Chapter 11, The GemStone Tools: an Overview, describes several browser tools that allow you to manage sessions and transactions; log in and out of GemStone sessions; examine configuration parameters; and access commonly used GemStone Smalltalk expressions.

Chapter 12, Using the GemStone Programming Tools, explains how to use the GemStone browsers and tools to examine, modify, and create classes and methods in GemStone; execute and debug GemStone Smalltalk code; manage the connectors that establish relationships between client Smalltalk and GemStone server objects; and perform other tasks.

Chapter 13, Using the GemStone Administration Tools, describes the tools that let you manage access to objects, examine and modify GemStone SymbolLists and associated dictionaries, and administer user accounts.

Part 3: Appendixes

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

Appendix B, Client Smalltalk and GemStone Smalltalk, outlines a few general and syntactical differences between the client Smalltalk and GemStone Smalltalk dialects.

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.

Other GemStone Documentation

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

- *Programming Guide* – a programmer's guide to GemStone Smalltalk, GemStone's object-oriented programming language.
- *Topaz Programming Environment* – describes Topaz, a scriptable command-line interface to GemStone Smalltalk. Topaz is most commonly used for performing repository maintenance operations.
- *GemBuilder for C* – describes GemBuilder for C, a set of C functions that provide a bridge between your application's C code and the application's database controlled by GemStone.
- *System Administration Guide* – describes maintenance and administration of your GemStone/S system.

In addition, each release of GemBuilder for Smalltalk includes *Release Notes*, describing changes in that release, and platform-specific *Installation Guides*, providing system requirements and installation and upgrade instructions.

A description of the behavior of each GemStone kernel class is available in the class comments in the GemStone Smalltalk repository. Method comments include a description of the behavior of methods.

Technical Support

GemStone Website

http://support.gemstone.com

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

- Documentation for released versions of all GemStone products, in PDF form.
- Downloads and Patches, including past and current versions of GemBuilder for Smalltalk.
- Bugnotes, identifying performance issues or error conditions you should be aware of.
- TechTips, providing information and instructions that are not otherwise included in the documentation.
- Compatibility matrices, listing supported platforms for GemStone product versions.

This material is updated regularly; we recommend checking this site on a regular basis.

Help Requests

You may need to contact Technical Support directly, if your questions are not answered in the documentation or by other material on the Technical Support site. Technical Support is available to customers with current support contracts.

Requests for technical support may be submitted online, or by email or by telephone. We recommend you use telephone contact only for serious requests that require immediate attention, such as a production system down. The support website is the preferred way to contact Technical Support.

Website: http://techsupport.gemstone.com

Email: techsupport@gemstone.com

Telephone: (800) 243-4772 or (503) 533-3503

If you are reporting an emergency by telephone, select the option to transfer your call to the Technical Support administrator, who will take down your customer information and immediately contact an engineer. Please also open a ticket on the website, and include error and log information. Non-emergency requests

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

When submitting a request, please include the following information:

- Your name, company name, and GemStone server license number.
- The versions of all related GemStone products, and of any other related products, such as client Smalltalk products.
- The operating system and version you are using.
- A description of the problem or request.
- Exact error message(s) received, if any, including log files if appropriate.

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

24x7 Emergency Technical Support

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

Training and Consulting

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

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 9. Performance Tuning

Chapter 10. GemBuilder Configuration Parameters

Chapter 11. The GemStone Tools: an Overview

Chapter 12. Using the GemStone Programming Tools

Chapter 13. Using the GemStone Administration Tools

Appendix A. Packaging Runtime Applications

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.

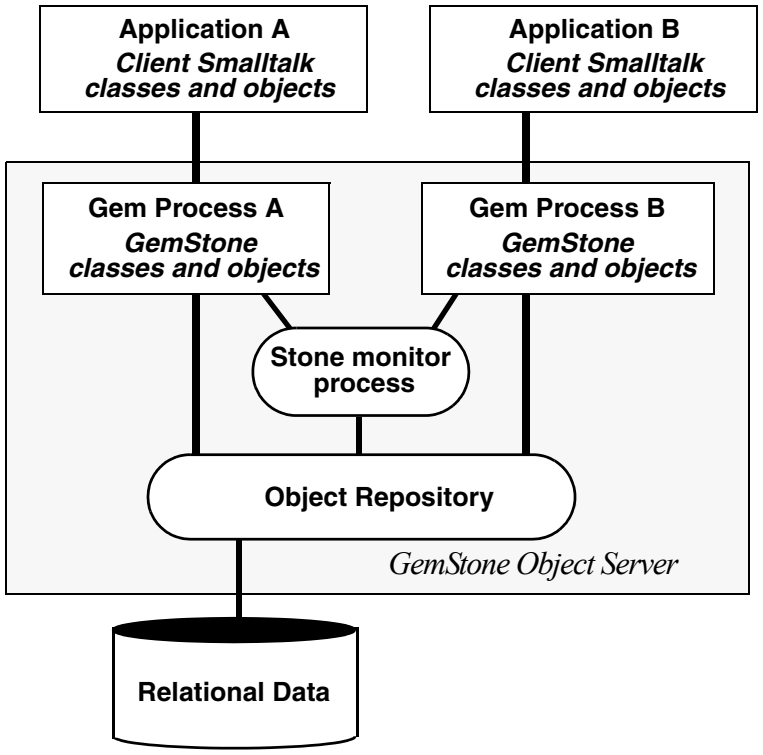
1.1 The GemStone Object Server

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 a Gem process to serve your application. Many Gem processes can actively communicate with a single repository at the same time.

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

Transparent access to GemStone

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

Many of the classes in the base 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.

Chapter 4 describes GemBuilder's mechanisms for making your client Smalltalk objects persistent, and Chapter 9 explains how to tune the system to minimize maintenance overhead and optimize performance.

GemStone's Smalltalk Language

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

GemStone Smalltalk is like single-user client Smalltalk in both its organization and syntax. Objects are defined by classes based on common structure and protocol and classes are organized into an *is-a* hierarchy, rooted at class Object. The class hierarchy is extensible; new classes can be added as required to model an application. The behavior of common classes conforms to the ANSI standard for Smalltalk. GemStone's class hierarchy is discussed in the introductory chapter to the *GemStone/S 64 Bit 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 subsequent chapters of this manual:

- A *GemStone System Browser* lets you examine, modify, and create GemStone classes and methods.
- A *GemStone System Workspace* provides easy access to commonly used GemStone Smalltalk expressions.
- *GemStone Inspectors* let you examine and modify the state of GemStone objects.
- A *GemStone Breakpoint Browser* and a *Debugger* let you examine and dynamically modify the state of a running GemStone application.
- A *Session Browser* allows you to manage sessions and transactions.
- A *Connector Browser* allows you to manage the connectors that establish relationships between client Smalltalk and GemStone server objects.
- A *Class Version Browser* can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.
- A *Symbol List Browser* allows you to examine the GemStone Symbol Lists associated with UserProfiles, add and delete dictionaries from these lists, and manipulate the entries in those dictionaries.
- A *Settings Browser* allows you to examine and set the configuration parameters for GemBuilder.
- A *User Account Management Tool* allows you to create new user accounts, change account passwords, and assign group membership.
- A *Security Policy Tool* 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 GemStone to store those objects that need to exist between sessions and must be shared with other users. For example, objects representing information in your application such as financial statements, employee health records, or library book cards would certainly require storage in GemStone. Some methods for manipulating the persistent data can also be usefully coded in GemStone Smalltalk and stored in GemStone for improved efficiency.

You don't need to store 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.

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

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 132.

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 *GemStone/S 64 Bit Programming Guide* gives a variety of tips in the chapter entitled "Tuning Performance."

1.4 Delivery and Deployment

GemBuilder is provided in the form of ENVY applications named GbsRuntime, GbsTools, and CstMessengerSupport.

- GbsRuntime and CstMessengerSupport are required for all uses of GemBuilder.
- GbsTools contains development and administration tools that are normally used only during development. It is almost always desirable to have GbsTools

present during development, but GbsTools can be omitted from most deployed applications.

Public and Private Classes and Methods

GemBuilder adds many classes and methods to your client Smalltalk image. Some of these we consider 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. Most GemBuilder classes and methods we consider 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 because they may have undocumented side effects, and because they are subject to change from release to release.

A GemBuilder class is private if its name begins with the prefix `Gbx`.

A GemBuilder method can be marked private in any of several ways:

- Any method defined in a private class is private unless the class comment indicates otherwise.
- 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. This User's Guide documents 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 the "gb". Methods with this prefix are reserved for GBS.

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.

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 with the GemStone object server product release. 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, but continue to use the same version of the GemStone server, the same clientLibraries will be used. The client libraries must be on the platform-dependent search path.

To set the client library, use the GbsConfiguration setting "libraryName". You may also execute:

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

For more information on this setting, see Chapter 10, "GemBuilder Configuration Parameters." To determine the correct client library name to use for your GemStone/S server product and version, see the *GemBuilder for Smalltalk Installation Guide*.

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 Session Browser.

RPC and Linked Sessions

With VA Smalltalk, gems run as a separate operating system process and respond to Remote Procedure Calls (RPCs) from its client. The session that this gem supports is called an *RPC* session.

On platforms that host the GemStone object server and its runtime libraries, and in which 64-bit client Smalltalk environments are available, one Gem can be integrated with the application into a single operating system process. That Gem is called a *linked* session. VA Smalltalk does not support linked sessions.

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.

These operations can be performed using GemBuilder's Session Browser and Session Parameters Editor. For more on using GemBuilder's graphical tools, see "The GemStone Session Browser" on page 158.

The remainder of this chapter discusses managing sessions programmatically, using three GemBuilder classes: 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 identify the object server and user account to which you wish to connect. This information is stored in an instance of GbsSessionParameters and added to a list maintained by GBSM. You can provide the information through window-based tools or programmatically. Both methods are described in later sections. In either case, you must supply these items:

- **The name of the GemStone repository**
  For a Stone running on a host other than the Gem host (described 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 for GemStone/S 64 Bit*) For instance, for a Stone named "gs64stone" on a host named "pelican", specify an NRS string of the form:

  ```
  !@pelican!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 213.) 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 if netldi is run 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 name and password if the netldi is running in guest mode.

- **Gem service**
  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
  ```

Once defined, an instance of GbsSessionParameters can be used for more than one session.

Defining Session Parameters Programmatically

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

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

Storing Session Parameters for Later Use

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

```
GBSM addParameters: aGbsSessionParameters
```

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

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

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

```
GBSM removeParameters: aGbsSessionParameters
```

Password Security

You can control the degree of security that GemBuilder applies to the passwords in a session parameters object. For example, when you create the parameters

object, you can specify the passwords as empty strings. When the parameters object is subsequently used in a login message, GemBuilder will prompt the user for the passwords.

For example:

```
mySessionParameters := GbsSessionParameters
   newWithGemStoneName: '!@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.

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 specific session a message 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-execute** 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 Session Browser. But before the commit takes place, all other session-dependent components are notified that a commit is about to occur. So a related application component, such as an open browser containing modified text, prompts you for permission to discard its changes before allowing the commit to proceed.

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

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

mySession `addDependent:` *myObj*

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

mySession `removeDependent:` *myObj*

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

Example 2.2

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

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

receive the `updateRequest:` message with any one of the following aspect symbols as its argument:

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

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

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

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

Example 2.3

```
updateRequest: aspect

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

^(#(queryAbort queryCommit queryEndSession)
  includes: aspect)
    ifTrue: ["My session wants to commit or abort.
              OK unless user doesn't want to."
              self askUserForPermission ]
   ifFalse: ["Let any other action occur."
              true]
```

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

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

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

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

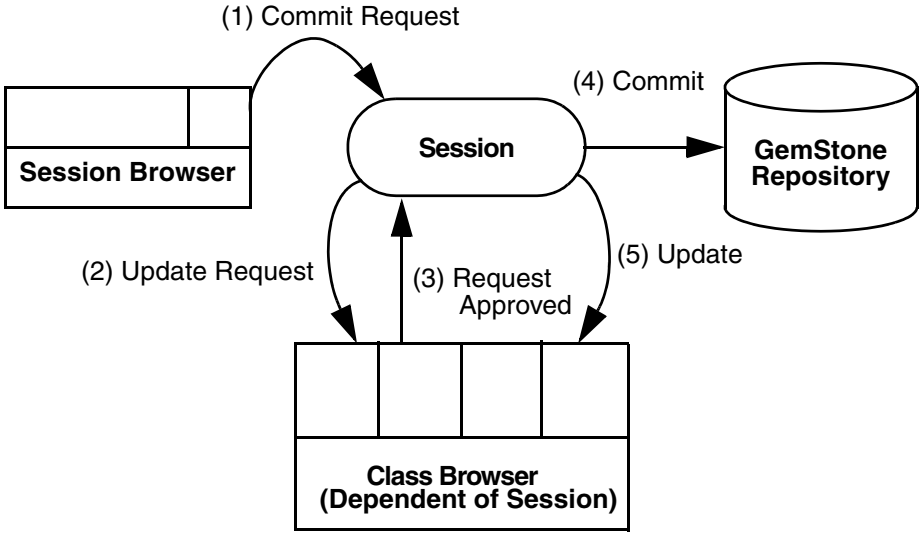
Each registered dependent should be prepared to receive an `update:` message with one of the above aspect symbols as its argument. Example 2.4 shows how a session dependent might implement an `update:` method.

Example 2.4

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

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

Figure 2.1 summarizes the sequence of events that occurs when a session queries a dependent before committing. In the figure, the Session Browser sends a commit request (`commitTransaction`) to a session (1). The session sends `updateRequest: #queryCommit` to each of its dependents (2). If every dependent approves (returns **true**), the commit proceeds (4). Following a successful commit, the session notifies its dependents that the action has occurred by sending `update: #committed` to each (5).

Figure 2.1 Committing with Approval From a Session Dependent



Chapter 3

Sharing Objects

This chapter describes how GemBuilder shares objects with the GemStone 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 Mechanisms
discusses the various ways replication mechanisms interact, and describes

how to determine whether an application object becomes a forwarder, stub, or replicate.

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

3.1 Which Objects to Share?

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

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

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

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

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

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

State 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 a 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, it executes locally.

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.

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

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.

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.

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 to dot 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 91.

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 152). 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 an application named UserClasses.

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 with 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/S 64 Bit 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.

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 that knows only which GemStone server object it represents. It responds to all messages by passing them to the appropriate server object, where data resides and behavior is implemented. (For historical reasons, this is the kind of forwarder usually meant when a discussion merely says "forwarder." This kind of forwarder is also called a server forwarder.)

- A *forwarder to the client* is a GemStone server object that knows only which client Smalltalk object it represents. It responds to all messages by passing them to the appropriate client Smalltalk object, where data resides and behavior is implemented.

You can create forwarders in several ways:

- Declare a connector as a forwarder upon login. For example, connect the GemStone global variable BigDictionary as a forwarder to the server so that it isn't replicated in the client.
- Specify that a given instance variable must always appear in the other object space as a forwarder to the server (using a replication specification, discussed starting on page 59). For example, a reference application might implement a specification that declares the class variable Atlas as a forwarder to the server.
- Prefix `fw` to a method name to return a forwarder from any message-send to GemStone. For example, to return a forwarder from a GemStone name lookup, send the GbsSession method `fwat:` or `fwat:ifAbsent:` instead of `at:` or `at:ifAbsent:`.
- Create a forwarder to the server explicitly using the message `#asForwarder` to any instance of GbsObject. For example:

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

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

Sending Messages

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

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

Arguments

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

When an argument is a block of executable code, special care is required: for details, see "Replicating Client Smalltalk Blocks" on page 70.

Results

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

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

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

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

Defunct Forwarders

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

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

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

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

Consider the following forwarder for the global BigDictionary:

Example 3.1

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

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

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

3.4 Replicates

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

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

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

To do so, GemBuilder must know about the structure of the two objects and the mapping between those structures. GemBuilder manages this mapping on a class basis: replicates must be instances of classes whose definitions are connected, by means of a class connector, to definitions of the corresponding class in the other

object space. GemBuilder handles many obvious cases automatically, but nonstandard mappings require you to override certain instance and class methods from class Object's GemStone support protocol. Nonstandard mappings are discussed starting on page 52.

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 55)

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 57) 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 visible to the session's private view of the GemStone repository, but it doesn't become part of the shared repository until your session commits—only then are your changes accessible to other users.

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

To disable automatic dirty-marking, execute:

```
GBSM autoMarkDirty: false
```

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

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

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

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

Instance Variable Mapping

As discussed in "Class Mapping" on page 43, before GemBuilder can replicate objects, it must know their respective structures and the mapping between them. By default GemBuilder maps instance variables by name. You can override this default either by suppressing certain instance variables, thereby rendering them invisible to an application, or by explicitly specifying a mapping between nonmatching names.

Suppressing Instance Variables

Some client Smalltalk objects, however, must define instance variables that are relevant only in the client environment—for example, a reference to a window object. Such data is transient and doesn't need to be stored in GemStone. Situations can also arise in which the GemStone class defines instance variables that a given application will never need; many applications can share repository objects without necessarily sharing the same concerns. Mapping allows your application to prune parts of an object.

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

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

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

Example 3.2

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

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

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

Nonmatching Names

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

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

In this way your application can accommodate differing schemas.

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

Example 3.3

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

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

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

Example 3.4

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

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

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

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

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

Stubbing

Often 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 is server counterpart into the client image. The client Smalltalk replicate then responds to the message.

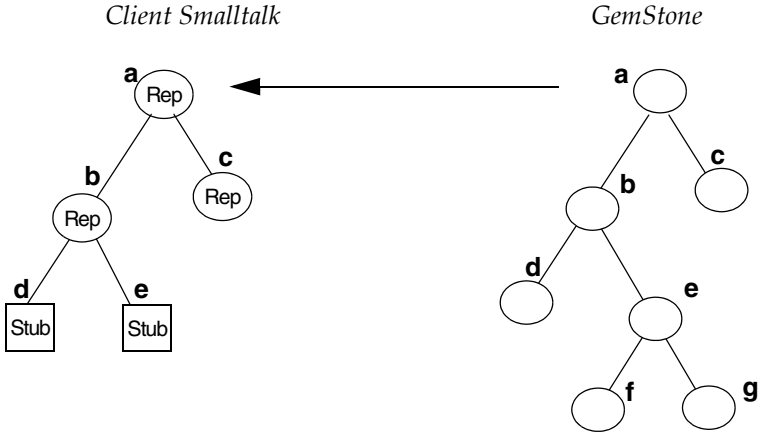
When GemBuilder faults automatically, it replicates the object hierarchy to a certain level, then creates stubs for objects on the next level deeper than that. The number of levels that are replicated each time is the *fault level.*

A fault level of 1 follows an object's immediate references and faults those in. A fault level of 2 follows one more layer of references and replicates those objects, too. Figure 3.1 illustrates an application with a fault level of 2.

Faulting at Login

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

Figure 3.1 Two-level Fault of an Object



Faulting in Response to a Message

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

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

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

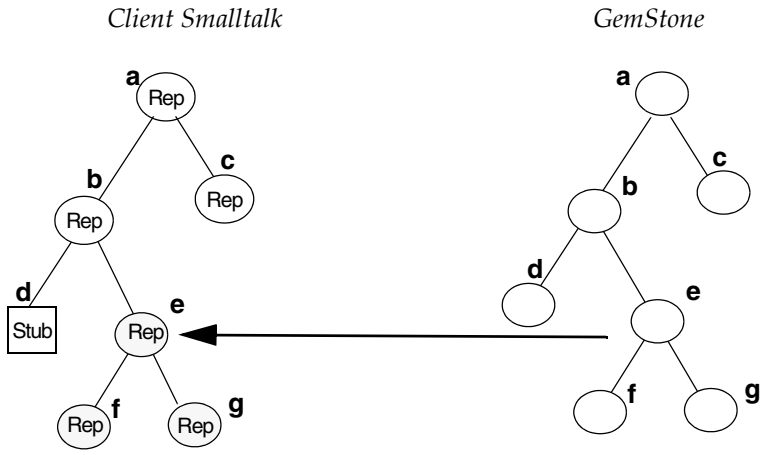
```
demonstrateProblem

    | 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:

```
    secondTemp := 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 is a global parameter.

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

Faulting in Changes From Other Sessions

Each session maintains its own view of the GemStone object server's shared object repository. The session's private view can be changed by the Smalltalk application when it adds, removes, or modifies objects—that is, you can see your own changes to the repository—or the Gem can change your view at transaction boundaries or after a session has executed GemStone Smalltalk.

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

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

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

- An *immediate* fault policy means that, when GemBuilder detects a change in a repository object, it updates the replicate immediately.

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

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

Overriding Defaults

GemBuilder has a default fault level of 4, specified by the configuration parameter `faultLevelRpc`.

- You can override the default 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 59.

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 136.

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

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

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

Defunct Stubs

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

modified on the 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 Smalltalk operation that encountered the defunct stub.

Replication Specifications

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

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

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

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

Example 3.5

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

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

```
super replicationSpec
```

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

(*instVar* *whenFaulted*)

instVar can be either:

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

whenFaulted is one of:

`stub` — faults in the instance variable as a stub.

`forwarder` — faults in the instance variable as a forwarder to the server.

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

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

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

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

Example 3.6

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

NOTE

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

Replication Specifications and Class Versions

As explained in "Mapping and Class Versions" on page 46, client Smalltalk classes connect not simply to GemStone Smalltalk classes, but to specific GemStone class versions. A class connector connects to at most one GemStone version.

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

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

Multiple Replication Specifications

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

By default, the message `replicationSpec` returns the default replication spec. Change this by sending the message `replicationSpecSet:`

#*someRepSpecSelector* to an instance of GbsSession. With this message, you can specify multiple replication specs, selecting one dynamically according to circumstances. The following procedure shows how:

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

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

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

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

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 GemStone traverses an object, it also traverses 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 replicate the association, and the key in the association. The default implementation for Dictionary class >> replicationSpec therefore contains #(indexable_part min 1), and Association class >> replicationSpec contains #(key min 1).

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

You could ensure this by implementing Employee class >> replicationSpec to include #(firstName min 1) and #(lastName min 1). Or, if you had a special Dictionary class for Employees, you could include #(indexable_part min 3) in that dictionary class's replication spec. However, this could cause the entire Employee to be 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 traversed. If a superclass specifies one way of handling an instance variable, and a subclass reimplements `replicationSpec` to handle the same variable in a different way, the last occurrence takes precedence.

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

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

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

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

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

not:

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

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

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

Customized Flushing and Faulting

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

Modifying Instance Variables During Faulting

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

To provide these buffers, reimplement the class methods:

```
namedValuesBuffer
indexableValuesBuffer
```

To unpack these buffers correctly, reimplement the class methods:

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

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

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

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

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

NOTE

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

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

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

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

NOTE

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

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

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

For example:

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

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

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

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

Modifying Instance Variables During Flushing

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

`namedValues`

Implement this to return a copy of the object being stored or an instance of

GbsBuffer sized to match the number of named instance variables in the client object. The store operations then access this buffer for storing in GemStone.

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

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

Example 3.7

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

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

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

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

Example 3.8

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

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

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

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

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

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

```
indexableSize
      ^self size
```

Mapping Classes With Different Formats

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

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

Return	Format
0	pointers
1	bytes
2	NSC or nonsequenceable collection

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

Limits on Replication

Replicating blocks, and scaled decimals, and collections with instance variables can present special problems, discussed below.

Replicating Client Smalltalk Blocks

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

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

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

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

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

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

to GemStone to be compiled and executed.

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

To avoid these limitations, consider using block callbacks instead. Block callbacks are discussed starting on page 73.

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

Image-stripping Limitations

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

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

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

—is instead coded:

```
aForwarder selectNameEquals: #Fred
```

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

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

When the block is encoded entirely on the GemStone server 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/S 64 Bit 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 the GemStone server. Replication is not supported for blocks that reference instance variables, class variables, method arguments, or temporary variables declared external to the block's scope.

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

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

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

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

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

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

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

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

^ result.
```

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

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

Restriction on References to self or super

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

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

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

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

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

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

Explicit Return Restriction

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

- A replicated block cannot contain an explicit return.

For example:

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

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

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

Replicating GemStone Blocks in Client Smalltalk

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

If restrictions hamper you, consider using block callbacks instead.

Block Callbacks

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

Block callbacks have the following advantages over block replication:

- Block callbacks don't require a compiler or decompiler.
- Block callbacks don't suffer the context limitations of block replication. The block can reference self, super, instance variables, and non-local temporaries;

it can also perform explicit returns. For example, the following expression works correctly as a block callback, but fails if you try to replicate the block:

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

Block callbacks have the following disadvantages:

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

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

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

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

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

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

Replicating Fixed/Scaled Decimals

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

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

In VisualAge, a ScaledDecimal is represented as a 31-digit number, with a field width and a scale indicating how many of the digits are to the right of the decimal

point. When creating a VisualAge ScaledDecimal from a GemStone instance, any further digits in the fractional component are truncated.

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

Example 3.9

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

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

Example 3.10

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

Client Copies

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

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

NOTE

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

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

```
aGbsObject asLocalObjectCopy
```

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

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

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

```
aClientObject asGSObjectCopy
```

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

To do this reliably:

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

NOTE

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

3.5 Precedence of Replication Mechanisms

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

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

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

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

- For classes that use no other mechanism, the configuration parameter `faultLevelRpc` is 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-Execute**, **GS-Display** or **GS-Inspect** 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/S 64 Bit 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 3.11 shows how to use `evaluate:` to execute code.

Example 3.11

```
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.

Example 3.12

```
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. Table 3.1–Table 3.5 list the results of sending any of several conversion messages to these objects.

NOTE

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

Table 3.1 Delegate Conversion Protocol

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

Table 3.2 Forwarder (to the Server) Conversion Protocol

Message	**Return Value**
`copy`	copies associated server object and returns replicate of copy
`asLocalObject`	undefined

Table 3.2 Forwarder (to the Server) Conversion Protocol

Message	Return Value
`asGSObject` (not recommended for customer applications)	associated delegate
`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
`asGSObject` (not recommended for customer applications)	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self`
`fault`	`self`, whose instance variables are now also replicates to the configured fault level
`stubYourself`	`self`, which has become a stub

Table 3.4 Stub Conversion Protocol

Message	Return Value
`copy`	shallow copy; receiver becomes a replicate
`asLocalObject`	undefined
`asGSObject` (not recommended for customer applications)	associated delegate
`asForwarder`	`self`, which has become a forwarder
`beReplicate`	`self` (use `fault` to become a replicate)
`fault`	`self`
`stubYourself`	`self`

Table 3.5 Conversion Protocol for Unshared Client Objects

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

Chapter

4 Connectors

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.

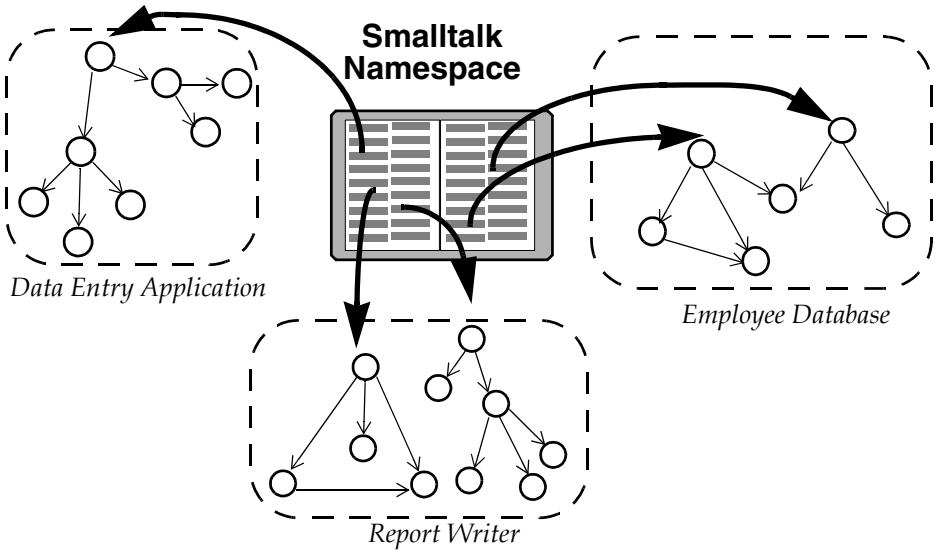
You can take advantage of this hierarchical structure to minimize application overhead. Identify the object at the root of each subsystem of shared objects, and then connect only these root objects. Depending on how you've defined configuration parameters and related matters, you can synchronize entire 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.

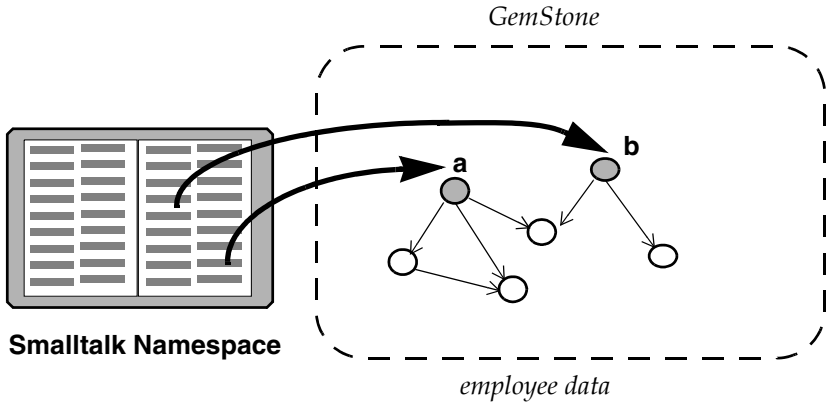
Figure 4.1 Connecting Application Roots



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 46.

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 50, or they do not propagate at all, as is normally the case with class connectors, as described in "Class Mapping" on page 43.

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 185, 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 143 for details on setting session-specific configuration parameters. The detailed description for this configuration parameter is on page 148.

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 43, 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

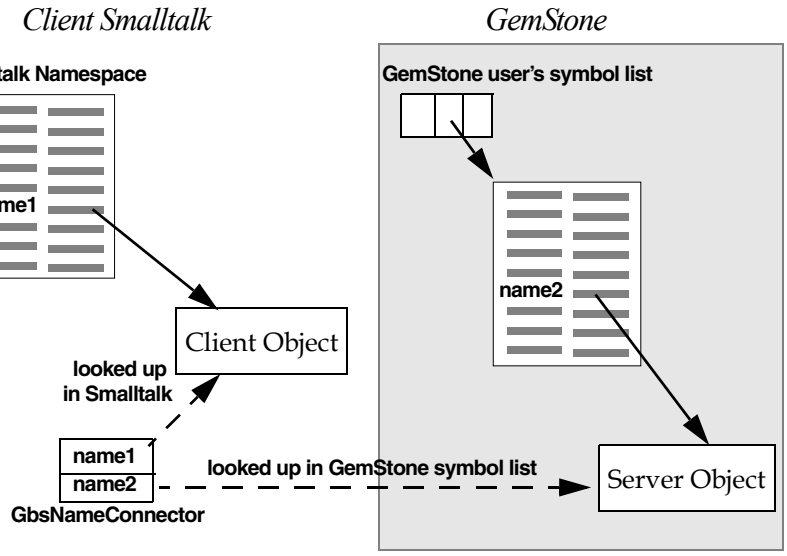
Connecting by Name

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.

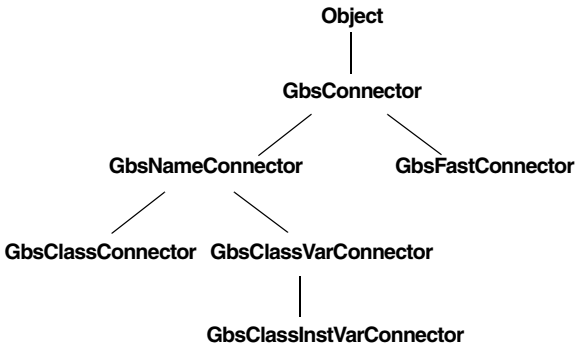
4.4 Making and Managing Connectors

To make and manage connectors interactively, see "The Connector Browser" on page 185. 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 32.)

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_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/S 64 Bit Programming Guide* entitled "Transactions and Concurrency Control."

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

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

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

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

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

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

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

5.1 Transaction Management: an Overview

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

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

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

During the course of a logged-in session an application can submit many transactions to the GemStone object server. In a multiuser environment, concurrency conflicts 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.

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

Another potential mode is *transactionless mode*. However, this mode is not usable from within GemBuilder.

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.

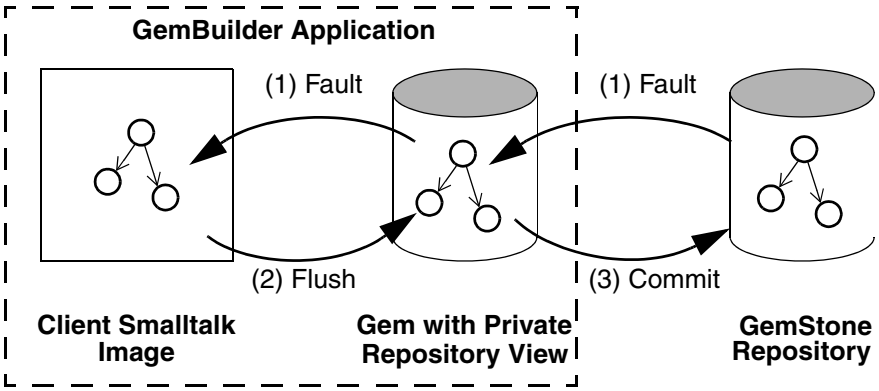
This chapter explains each of the topics mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced conflict classes. Be sure to refer to the related topics in the *GemStone/S 64 Bit Programming Guide* for a full understanding of these transaction management concepts.

5.2 Operating Inside a Transaction

While a session is logged in to the GemStone object server, 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.

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 Session Browser, select the session and click on the **Commit...** button; or in the Class Browser, use the pop up menu on the SymbolDictionary pane to select **commit**.

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 37.)

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 Session Browser, select a logged-in session and click on the **Abort...** button; or in the Class Browser, use the pop up menu on the SymbolDictionary pane to select **abort**.

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 108 for more discussion of the kinds of conflicts that can arise.)

5.3 Operating Outside a Transaction

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

For efficiency, an application may configure a session to operate *outside a transaction*. When operating outside a transaction, a session can view the repository, browse the objects it contains, and even make computations based upon their values, but it cannot commit any new or changed GemStone server objects. When a session is operating outside a transaction, the Stone may request that the session abort. A session operating outside a transaction can, at any time, begin a transaction.

No session is overhead-free: even a session operating outside a transaction uses GemStone resources to manage its objects and its view of the repository. For best

system performance, all sessions, even those running outside a transaction, must periodically refresh their views of the repository by committing or aborting.

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

Table 5.1 GbsSession Methods for Running Outside of a Transaction

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

To begin a transaction, send the message:

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

or:

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

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

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

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

Being Signaled to Abort

When you are in a transaction, GemStone waits until you commit or abort to reclaim storage for objects that have been made obsolete by your changes. When you are running outside of a transaction, however, you are implicitly giving GemStone permission to send your Gem session a signal requesting that you abort

your current view so that GemStone can reclaim storage when necessary. When this happens, you must respond within the time period specified in the STN_GEM_ABORT_TIMEOUT parameter in 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 configuration parameter STN_GEM_LOSTOT_TIMEOUT. The Stone forces an abort by sending your session an `abortErrLostOtRoot` signal, which means that your view of the repository was lost, and any objects that your application had been holding may no longer be valid. When your application receives `abortErrLostOtRoot`, the application must log out of GemStone and log back in, thus rereading all of its data in order to resynchronize its snapshot of the current state of the GemStone repository.

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

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

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

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

5.4 Transaction Modes

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

Automatic Transaction Mode

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

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

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

Manual Transaction Mode

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

To begin a transaction, send the message

aGbsSession `beginTransaction`

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

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

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

Choosing Which Mode to Use

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

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

Switching Between Modes

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

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

aGbsSession `transactionMode: #autoBegin`

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

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

aGbsSession `transactionMode: #manualBegin`

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

5.5 Managing Concurrent Transactions

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

1. It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have also modified during your transaction. If they have, then the resulting modified objects can be inconsistent with each other.
2. It 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 details about read and write operations, optimistic and pessimistic concurrency control, and other general information about GemStone transactions, refer to 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*. If the lock is granted, the method returns the receiver. (Lock types are described in the *GemStone/S 64 Bit Programming Guide*). If you don't have the proper authorization, or if another session already has a conflicting lock, an error will be generated.

When you request 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-execute**:

```
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.

5.6 Reduced-Conflict Classes

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

GemStone provides some reduced-conflict classes that can be used instead of their regular counterparts in applications that might otherwise experience too many unnecessary conflicts. For details, refer to the "Transactions and Concurrency Control" chapter of the *GemStone/S 64 Bit Programming Guide*.

5.7 Changed Object Notification

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

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

Notify sets 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 139.)

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
```

5.8 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*

For example:

Example 5.3

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

"Now, from any session logged into the same Stone, send a
signal.(This example uses the same session)"
GBSM currentSession
       sendSignal: 15
       to: (GBSM evaluate: '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/S 64 Bit Programming Guide* for details on using the error mechanism for change notification.

5.9 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.

```
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

Once objects have been successfully committed to GemStone, they can be damaged or destroyed only by mishaps that damage or erase the disk files containing your repository. GemStone provides several mechanisms for safeguarding the objects in your GemStone repository. These mechanisms are discussed in the chapter on creating and restoring backups in the *System Administration Guide for GemStone/S 64 Bit*.

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 GemStone/S 64 Bit* 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 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

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

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

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.

In GemStone/S 64 Bit, 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. In the 32-bit GemStone/S server product, every server object is associated with an object security policy that controls access to that object.

NOTE

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

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 *GemStone/S 64 Bit Programming Guide*.

Chapter

7 Error-handling

This chapter discusses errors: how to handle them and how to recover from them.

Error-handling and Recovery
explains how GbsError objects are created and used.

User-defined Errors
explains how to define and signal your own errors.

7.1 Error-handling and Recovery

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

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

Stack-based Error-handling

You can use the `on:do:` method to install error handlers to anticipate specific GemStone errors, as shown in Example 7.1.

Example 7.1

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

You can also create a handler to check for any GemStone error that falls in one of the following categories:

```
#compilerErrorSignal
#abortingErrorSignal
#interpreterErrorSignal
#fatalErrorSignal
#eventErrorSignal
```

For instance, this will handle any GemStone Smalltalk compiler error:

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

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

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

Session-based Error-handling

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

Example 7.2

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

User-defined Errors

You can define and signal your own errors in GemStone. For more information on how to do this, see the *GemStone/S 64 Bit Programming Guide.*

In a GemBuilder application, you define a generic GemStone error-handler by defining a standard client Smalltalk signal handler on the signal `GbsError errorSignal`. This handles any GemStone error, including user-defined errors.

If you want to define a client Smalltalk exception handler for a specific user-defined error, you will need to register an exception, GemStone error number, and a symbol representing that error with GbsError. To do this, send GbsError class>>`defineErrorNumber:name:signal:`.

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

Example 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 ).
```

In Smalltalk, the following code would signal your newly created error:

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

A generic signal-handler for all GemStone errors would trap this signal:

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

To explicitly handle your new error in client Smalltalk, you first need to define a name and signal for it. The new signal should inherit from GbsError errorSignal.

```
GbsError
   defineErrorNumber: 10
   name:  #myNewError
   signal: GbsError errorSignal newChild.
```

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

Example 7.4

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

For information on how to create GemStone error dictionaries and how to handle GemStone errors (predefined and user-defined) within the GemStone environment, see the chapter entitled "Handling Errors" in the *GemStone/S 64 Bit Programming Guide*.

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

7.2 Detecting GemStone Interrupts

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

Chapter

8 Schema Modification and Coordination

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

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

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

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

8.1 Schema Modification

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

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

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

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

The section entitled "Modifying an Existing Class" on page 181 explains how to create different versions of a class in GemBuilder.

Instance Migration Within GemStone

The migration operation in GemStone is flexible and configurable.

- Instances of any class can migrate to any other, as long as they share a class history. The two classes need not be similarly named or have anything else in common.
- Migration can occur whenever you want it to.
- You don't have to migrate all instances of a class at once; you can migrate only certain instances as needed.
- You can choose which values of the old instance variables are used to initialize values of the new instance variables. overriding the default mapping mechanism as necessary.

Setting the Migration Destination

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

OldClass `migrateTo:` *NewClass*

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

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

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

Migrating Objects

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

You can execute the following expression to identify instances that may need to be migrated:

```
SystemRepository listInstances: anArrayOfClasses.
```

The `listInstances:` message takes as its argument an array of classes and returns an array of sets. The contents of each set consists of all instances whose class is equal to the corresponding element in the argument *anArrayOfClasses*. Instances to which you lack read authorization are omitted without notification.

The simplest way to migrate an instance of an older class is to send the message `migrate` to the instance. If the object is an instance of a class for which a migration destination has already been defined, the object becomes an instance of the specified version of the class. If no destination has been defined, no change occurs.

You can bypass the migration destination or migrate instances of classes for which no migration destination has been specified by specifying the destination directly in the message that performs the migration.

The following messages (defined in class Class) specify a one-time-only operation that ignores any preset migration destination class.

`migrateInstances:`*aCollectionOfInstances* `to:`*DestinationClass*

`migrateInstancesTo:`*DestinationClass*

The `migrateInstances:to:` message migrates specified instances to a class; the `migrateInstancesTo:` migrates all instances of the receiver to a class.

Things to Watch Out For

There are a few things that you should be aware of when migrating objects.

- You cannot send a `migrate` message to `self`. Attempting to do so generates an error that reports "The object you are trying to migrate was already on the stack."
- You cannot migrate instances that you are not authorized to read or write.
- You need to be aware that the instance variable map used in migrating instances from one GemStone class to another is not the same as the instance variable map described in Chapter 3, whose purpose is to map instance variables from GemStone to Smalltalk.

Instance Variable Mapping in Migration

GemStone supports instance migration between two classes that belong to the same class history. For simple migrations, such as the addition or removal of an instance variable, GemStone provides a default migration mechanism that copies data from each instance variable of the old object to the instance variable of the same name in the new object (if one exists). You can write methods to customize this migration on a class-by-class basis.

When an object is migrated, it refers to the class and class instance variables that have been defined for the new version of the class. These variables have whatever values have been assigned to them in the class object.

The simplest way to retain the data held in instance variables is to use instance variables having the same names. If two versions of a class have instance variables with the same name, the values of those variables are automatically retained when the instances migrate from one class to the other.

However, the structure of the two classes may be different, and a one-to-one mapping may not be possible. For example, if the new class has an instance variable for which no corresponding variable exists in the old class, that instance variable is initialized to *nil* upon migration. Similarly, if the old class has an

instance variable for which no corresponding variable exists in the new class, the value of the old variable is dropped and the data it represents is no longer accessible from that object.

You may encounter situations in which you want to initialize a variable having one name with the value of a variable having a different name. This requires providing an explicit mapping from the instance variable names of the older class to the instance variable names of the migration destination. To do this you will need to override the default mapping strategy by reimplementing a class method named `instVarMappingTo:` in your destination class. This method is defined in Class to return an instance variable mapping from the receiver's named instance variables to those in the other class, but it can be customized in the new class to explicitly map the two different names.

There also may be times when you need to perform a specific operation on the value of a given variable before initializing the corresponding variable in the class to which the object is migrating.

For example, suppose that you have a class named Point, which defines two instance variables: *x* and *y*. These instance variables define the position of the point in Cartesian two-dimensional coordinate space. Now suppose that you define a class named NewPoint to use polar coordinates. The class has two instance variables named *radius* and *angle*. The default mapping strategy would cause Point objects to completely lose their position because the old and new classes have no instance variables in common.

This can be handled, however, by overriding a migration method in NewPoint by defining it to include an operation that transforms the values of *x* and *y* into values that can properly be assigned to *radius* and *angle*. In this case, the appropriate method to override is `migrateFrom:instVarMap:`. Then, when you migrate an instance of Point to an instance of NewPoint, the migration code that calls `migrateFrom:instVarMap:` executes the method in NewPoint instead of the one in Object that defines the default behavior. (This example is explained in detail in the *GemStone/S 64 Bit Programming Guide*.)

8.2 Schema Coordination

GemBuilder's goal in supporting schema migration is to provide an interaction between the client Smalltalk and GemStone that provides as much of GemStone's capabilities as possible, while minimizing the impact on the client Smalltalk system.

GemBuilder preserves the behavior of having only a single version of a given class in client Smalltalk at one time. That client Smalltalk class will be mapped to a specific version of a GemStone class, resolved at login time by its name. If, while faulting an object into the client Smalltalk, GemBuilder discovers that the object is an instance of a class that is a different version of the class that is in client Smalltalk, it will be faulted in in the format of the class in client Smalltalk and flagged so that if it is modified and written back to GemStone, it can be written out in the appropriate format.

For example, suppose you have a class named **C** in GemStone, and there are two versions of it: $\mathbf{C_1}$ and $\mathbf{C_2}$. Suppose that client Smalltalk has a representation of $\mathbf{C_2}$. Instances of $\mathbf{C_2}$ are replicated back and forth between client Smalltalk and GemStone, as usual.

If it attempts to replicate an instance of $\mathbf{C_1}$, however, GemBuilder will discover that there is no class mapping for $\mathbf{C_1}$. GemBuilder will then do the following:

1. It will fetch the name of the GemStone class and discover that there is a client Smalltalk class by the same name that is already mapped to a GemStone class.
2. It will verify that the two GemStone classes are in the same class history.
3. It will then ask GemStone to make a migrated copy of the object in $\mathbf{C_2}$ format and to replicate that migrated copy into client Smalltalk. The proxy associated with that client Smalltalk object will be flagged to indicate that the client Smalltalk object is a migrated representation of the GemStone object. If that object is later modified in client Smalltalk and subsequently needs to be written to GemStone, GemBuilder will first flush the object from client Smalltalk to GemStone as an instance of $\mathbf{C_2}$, then have GemStone migrate the object back to an instance of $\mathbf{C_1}$.

This process is fairly expensive. If you are running GemBuilder in verbose mode, the discovery of an client Smalltalk class that is mapped to an old version of a GemStone class (a version that is not the migration destination) will be logged to the transcript. If you see this happening frequently, you should consider migrating your instances to the GemStone class version corresponding to your client Smalltalk class.

Chapter

9 Performance Tuning

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

Selecting the Locus of Control
provides some rules of thumb for deciding when to have methods execute on the client and when to have them execute on the server.

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

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
shows how to change the initial cache size.

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

For further information, see the *GemStone/S 64 Bit Programming Guide* for a discussion on how to optimize GemStone Smalltalk code for faster performance. That manual explains how to cluster objects for fast retrieval, how to profile your code to determine where to optimize, and discusses optimal cache sizes to improve performance.

9.1 Selecting the Locus of Control

By default, GemBuilder executes code in the client Smalltalk. Objects are stored in GemStone for persistence and sharing but are replicated in the client Smalltalk for manipulation. In general, this policy works well. There are times, however, when it is preferable or required to execute in GemStone.

One motivation for preferring execution in GemStone is to improve performance. Certain functions can be performed much more efficiently in GemStone. The following section discusses the trade-offs between client Smalltalk and server Smalltalk execution and how to choose one space over the other.

Beyond optimization, some functions can be performed *only* in GemStone. GemStone's System class, for example, cannot be replicated in the client Smalltalk; messages to System have to be sent in GemStone.

Locus of Execution

This section centers on controlling the locus of execution—in other words, determining whether certain parts of an application should execute in the client Smalltalk or in GemStone. Subsequent sections discuss other ways of tuning to increase execution speed.

Client Smalltalk and GemStone Smalltalk are very similar languages. Using GemBuilder, it is easy to define behavior in either client Smalltalk or GemStone to accomplish the same task. There are, however, performance implications in the placement of the execution. This section discusses several factors to weigh when choosing the space in which to execute methods.

Relative Platform Speeds

One consideration when choosing the execution platform is the relative speed of the client Smalltalk and the server Smalltalk execution environments. Your client Smalltalk will often run faster than GemStone on the same machine. GemStone's database management functions and its ability to handle very large data sets add some overhead that the client Smalltalk environment doesn't have.

Cost of Data Management

Execution cannot complete until all objects required have been brought into the object space. When executing in the client Smalltalk, this means that all GemStone objects required by the message must be faulted from GemStone. When executing in GemStone, this means that dirty replicates must be flushed from the client Smalltalk. In general, it is impossible to tell exactly which objects will be required by a message send, so GemBuilder flushes all dirty replicates *before* a GemStone message send and faults all dirty GemStone objects *after* the send.

Clearly, data movement can be expensive. Although the client Smalltalk environment might be more efficient for some messages, faulting the object into the client Smalltalk might overwhelm the savings. If the objects are all already there, however, or if the objects will be reused for other messages, then the movement may be justified.

For example, consider searching a set of employees for a specific employee, giving her a raise, and then moving on to another unrelated operation. Although a brute force search may be faster in your client Smalltalk, the cost of moving the data to the client may exceed the savings. The search should probably be done in GemStone.

However, if additional operations are going to be done on the employee set, the cost of moving data is amortized and, as the number of operations increases, becomes less than the potential savings.

GemStone Optimization

Some optimizations are possible only using GemStone execution. In particular, repository searching and sorting can be done much more quickly in GemStone than in your client Smalltalk as data sets become large.

If you will be doing frequent searches of data sets such as the employee set in the previous example, using an index on the server Smalltalk set will speed execution.

The *GemStone/S 64 Bit Programming Guide* provides a complete discussion of indexes and optimized queries.

9.2 Profiling

Profiling Client Smalltalk Execution

A good starting point for optimizing the performance of an application is to find out where most of the execution time is being spent. There are tools available for

profiling client Smalltalk code. GemStone also has a profiling tool in the class ProfMonitor. This class allows you to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method. See the chapter on performance in the *GemStone/S 64 Bit Programming Guide* for details.

Watching Stub Activity

A switch in the stub class, GbxObjectStub, allows you to see stubs in a debugger and logs faulting activity involving stubs.

```
GbxObjectStub stubDebugging: aBoolean
```

This method turns stub debugging support on (`stubDebugging: true`) or off (`stubDebugging: false`). When stub debugging is `true`, this class's superclass is GbsDebugStub, providing basic instance methods that allow the client Smalltalk debugger to operate among stubs without causing them to fault in the GemStone object.

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

Notice that applications that rely on these methods might get incorrect results when stub debugging is turned on. For example, sending `#class` to a GbxObjectStub normally causes a fault, returning the class of the replicated object, but when `stubDebugging` is on, the result of sending `#class` is GbxObjectStub.

Using Verbose Mode

GbsSession has a class variable, Verbose (a Boolean), which, if true, causes sessions to write messages to the system transcript when special events occur (such as logout, login, commit, and abort).

If your application sends `Block>>valueUninterruptably`, you may need to disable the GbsSession's logging of events to the transcript by sending `GBSM verbose: false`. Verbose mode uses `Transcript show:`, which eventually calls `Block>>valueUninterruptably`. If unstubbing occurs during execution of your application's `Block>>valueUninterruptably`, and the unstubbing activity triggers activity that is logged to the transcript, the client Smalltalk will fail.

9.3 Replication Tuning

The faulting of GemStone objects into the client Smalltalk is described in Chapter 3. As described there, a GemStone object has a replicate in the client Smalltalk created for itself, and, recursively, for objects it contains to a certain level, at which point stubs instead of replicates are created.

Faulting objects to the proper number of levels can noticeably improve performance. Clearly, there is a cost for faulting objects into the client Smalltalk. This is made up of communication cost with GemStone, object accessing in GemStone, object creation and initialization in the client Smalltalk, and increased virtual machine requirements in the client Smalltalk as the number of objects grows. For this reason, you should try to minimize faulting and fault in to the client only those objects that will actually be used in the client.

On the other hand, inadequate faulting also has its penalties. Communication overhead is important. When fetching an employee object, it is wasteful to stub the name and then immediately fetch the name from GemStone.

Controlling the Fault Level

By default, four levels of objects are faulted. 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 parameter `faultLevelRpc` 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. To examine or change this parameter, choose **GemStone > Browse > Settings** and select the **Replication** tab in the resulting Settings Browser.

NOTE

Take care when using a level of 0 to control replication. GemStone can store more objects than can be replicated in a client Smalltalk object space.

Preventing Transient Stubs

If only the `defaultGStoSTLevel` mechanism is used to control fault levels, it is possible to create large numbers of stubs that are immediately unstubbed.

To prevent stubbing on a class basis, reimplement the `replicationSpec` class method for that class. For details, see “Replication Specifications” on page 59.

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

```
GBSM traversalBufferSize: aSmallInteger.
```

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

9.4 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.

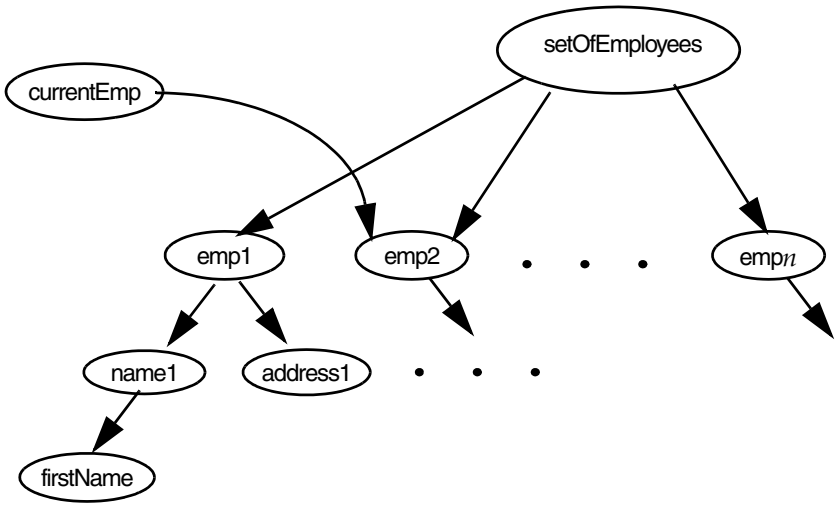
Measures you can take to control the size of the client Smalltalk object cache include explicit stubbing, using forwarders, and not caching certain objects.

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.

Also, be aware of the difference between two ways of modifying the value of an instance variable: by using an access method and by direct assignment. For example, consider an object with an instance variable named instVarX. You can assign the value 5 to instVarX in two ways:

```
insVarX := 5          (direct assignment)
self instVarX: 5      (access method)
```

When the object is replicated in your Smalltalk workspace, each of these assignments yields the same result. When the object is represented in the Smalltalk workspace by a stub, however, the stub must be faulted in as a replicate ("unstubbed") before the assignment can occur. The access method causes the stub

to be faulted in and yields the correct result. Direct assignment, however, does not cause the stub to be faulted in and can cause errors:

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

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

Using Forwarders

Another solution is to declare the `setOfEmployees` as a forwarder. See "Forwarders" on page 46.

Not Caching Selected Objects

Finally, it is possible to specify classes whose instances should not be added to the transparency caches. You can reimplement the instance method `shouldBeCached` to cause GemBuilder to not add instances of that class to the transparency caches.

This can help control the size of the caches, but it would do so at the expense of giving up two-space referential integrity for those objects, and this might not be an acceptable side effect in certain applications.

For example, classes whose instances are modifiable should probably not return `false` to this message, because any modifications to the object could not be propagated back to the original GemStone object, as GemBuilder would have no way of knowing which object in the repository it came from. Nor should classes whose instances rely on their identity in any way return `false` to this message.

An example of a class that could be considered a candidate for returning `false` is Float. If floats are omitted from the transparency caches, consider the following subtle implications: If a float **F** is referenced by two other objects, **A** and **B**, then after replicating **A** and **B** into the client Smalltalk, there will be two distinct (but equal) copies of the object **F** in the client Smalltalk. If one or both of **A** and **B** are modified in Smalltalk and flushed back to GemStone, there will now be two distinct (but equal) copies of **F** in GemStone. Typically, referential integrity for floats isn't crucial, because comparison between floats is usually by equality rather than identity.

9.5 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*. There are serious concerns when doing this. In general, such applications will be less portable and less maintainable. However, when used judiciously, there can be significant performance benefits.

In general, profile your code and find those methods that are heavily used to be candidates for primitives or user actions. The trick to proper use of primitives or user actions is to create as few as possible. Excess primitives or user actions make the system more difficult to understand and place a heavy burden on the maintainer.

For a description about adding primitives to your client Smalltalk, see the vendor's documentation. For adding user actions to GemStone, see the *GemBuilder for C* user manual.

To load the user action in client Smalltalk, execute:

```
GBSM loadUserActionLibrary: userAction
```

9.6 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.

Process-safe Transparency Caches

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

```
aGbsSession processSafeCaches: true
```

When **processSafeCaches** is `true`, subsequent logins use transparency cache dictionaries that are protected. However, some operations take a bit longer when using protected dictionaries.

Blocking and Nonblocking Protocol

GemStone operations may execute synchronously or asynchronously. If executing synchronously, the application must wait for a GemStone operation to complete before proceeding with the execution process that called it. Synchronous operation is known in GemBuilder as *blocking protocol*.

GemBuilder session also support asynchronous operation: *nonblocking protocol*. When the configuration parameter **blockingProtocolRpc** is `false` (the default), client Smalltalk processes can proceed with execution during GemStone operations. A session, however, is permitted only one outstanding GemStone operation at a time.

When **blockingProtocolRpc** is `true`, behavior is synchronous: the execution process must wait for a GemStone call to return before proceeding.

One Process per Session

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

Multiple Processes per Session

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

You may not have designed your application to run multiple processes under a single GemStone session. However, if your application uses signals and notifiers, chances are it is occasionally running two processes against a single GemStone session. Methods that create concurrent processes include:

```
GbsSession
   >>notificationAction:
   >>gemSignalAction:
   >>signaledAbortAction:
```

When the specified event occurs, the block you supply to these methods runs in a separate process. Unless your main execution process is idle when these events

occur, you need to take the same precautions as any other application running multiple processes against a single session.

Applications that have multiple processes running against a single GemStone session should take these additional precautions:

- coordinate transaction boundaries
- coordinate flushing
- coordinate faulting

GemBuilder provides a method, `GbsSession>>critical:` *aBlock*, that evaluates the supplied block under the protection of a semaphore that is unique to that session. The best approach to creating an application that must support more than one process interacting with a single GemStone session is to organize its logical transactions into short operations that can be performed entirely within the protection of `GbsSession>>critical:`. All of that session's commits, aborts, executes, forwarder sends, flushes and faults should be performed within `GbsSession>>critical:` blocks.

For example, a block that implements a writing transaction will typically start with an abort, make object modifications, and then finish with a commit. A block that implements a reading transaction might start with an abort, perhaps perform a GemStone query, and then maybe display the result in the user interface.

Coordinating Transaction Boundaries

Multiple processes need to be in agreement before a commit or abort occurs. For example, suppose two processes share a single GemStone session. If one process is in the process of modifying a set of persistent objects and a second process performs a commit, the committed state of the repository will contain a logically inconsistent state of that set of objects.

The application must coordinate transaction boundaries. One way to do this is to make one process the transaction controller for a session, and require that all other processes sharing that session request that process for a transaction state change. The controller process can then be blocked from performing that change until all other processes using that session have relinquished control by 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 chapter 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 page 145 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 "The Settings Browser" on page 164.)

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.

Table 10.1 Configuration Parameters for GemBuilder

Parameter	Legal values	Default	Scope
assertionChecks	true/false	false	Global
autoMarkDirty	true/false	true	Global
blockingProtocolRpc	true/false	false	Global
blockReplicationEnabled	true/false	true	Global
blockReplicationPolicy	#replicate/ #callback	#replicate	Global
bulkLoad	true/false	false	Global
confirm	true/false	true	Global
connectorNilling	true/false	true	Session-specific
connectVerification	true/false	false	Global
defaultFaultPolicy	#immediate/#lazy	#lazy	Global
eventPollingFrequency	any integer	5000	
eventPriority	any integer	3	Session-specific
faultLevelRpc	any integer	4	Session-specific
forwarderDebugging	true/false	false	Global
freeSlotsOnStubbing	true/false	true	Global
fullCompression	true/false	false	Global
generateClassConnectors	true/false	true	Session-specific
generateClientClasses	true/false	true	Session-specific
generateServerClasses	true/false	true	Session-specific
initialCacheSize	any integer	5003	

Table 10.1 Configuration Parameters for GemBuilder (Continued)

Parameter	Legal values	Default	Scope
`InitialDirtyPoolSize`	any integer	100	Session-specific
`libraryName`	any string	empty string	Global
`removeInvalidConnectors`	true/false	false	Global
`stubDebugging`	true/false	false	Global
`traversalBufferSize`	any integer	250000	Session-specific
`verbose`	true/false	true	Global

assertionChecks

This parameter is for the use of GemStone customer 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

blockingProtocolRpc

Determines whether to use blocking or nonblocking protocol. 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.

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. 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.

See "The Connector Browser" on page 185.

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

defaultFaultPolicy

Specifies GemBuilder's default approach to updating client Smalltalk objects whose GemStone counterparts have changed. When **`#lazy`**, GemBuilder responds to a change in a GemStone 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

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: 300
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

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. 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

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 43.

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 43.

Legal values: true/false
Default: true
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 43.

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

InitialCacheSize

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

Legal values: any positive Integer, but a prime number is recommended.
Default: 5003
Settings Tool tab: Cache Tuning
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: 100
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. 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 issue does not arise again at next login.

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

When `false`, invalid connectors are ignored.

See "The Connector Browser" on page 185.

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

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

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. We begin this chapter with an overview of the GemStone menu, then describe several tools that allow you to manage sessions and transactions; log in and out of GemStone sessions; examine configuration parameters; and access commonly used GemStone Smalltalk expressions.

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

The GemStone Session Browser
describes the GemStone Session Browser and Session Parameters Editor.

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

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

GemStone Workspaces
describes GemStone workspaces.

The System Workspace
describes the System Workspace.

Subsequent chapters describe the GemStone programming and administration tools:

- Chapter 12, "Using the GemStone Programming Tools," describes the menus and commands that allow you to execute GemStone Smalltalk code, access GemBuilder programming tools, and make use of GemStone Smalltalk debugging facilities.
- Chapter 13, "Using the GemStone Administration Tools," describes the Security Policy Tool, Symbol List Browser, and GemStone user account management tools. Taken together, these tools enable you to easily manage the object sharing and protection issues discussed throughout this manual.

11.1 GemStone Menu

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

As shown in Table 11.1, the **GemStone** menu provides commands for executing GemStone Smalltalk code and accessing the GemStone programming tools.

Table 11.1 The GemStone Menu

Sessions	Opens a GemStone Session Browser, allowing you to log into or out of the GemStone server and manage transactions. The Session Browser is described on page 158.	
Connectors	Opens a GemStone Connector Browser, allowing you to manage the connections between GemStone server and Smalltalk client objects. The Connector Browser is described on page 185.	
Browse	Produces a submenu with the following options:	
	All Classes	Opens a GemStone Browser, comparable to the client Smalltalk System or Classes Browser. The GemStone Browser is described in "The GemStone Session Browser" on page 158.
	Namespace...	Prompts for the name of a symbol dictionary, then opens a browser focused on that dictionary.

Table 11.1 The GemStone Menu(Continued)

	Class...	Prompts for the name of a class, then opens a browser focused on that class.
	Senders of...	Prompts for the name of a message selector, then opens a method browser showing senders of that message.
	Implementors of...	Prompts for the name of a message selector, then opens a method browser showing implementors of that message.
	References to...	Prompts for the name of a variable, then opens a method browser showing all methods that refer to that variable.
	Methods with substring...	Prompts for a string, then opens a method browser showing all methods whose source contains that string.
Admin	Produces a submenu with the following options:	
	Users	Opens the GemStone 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 213.
	Namespaces	Opens a Symbol List Browser, allowing you to examine and modify symbol dictionaries and their entries. The Symbol List Browser is described on page 210.
	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 202.
Tools	Produces a submenu with the following options:	
	New GS Workspace	Opens a GemStone Workspace.

Table 11.1 The GemStone Menu(Continued)

	Open GS Workspace...	Prompts for a file name, then opens the selected saved workspace file in a GemStone workspace.
	GS File in...	Files the selected GemStone Smalltalk code into GemStone.
	Settings	Opens a Settings Browser in which you can examine, change, and store parameters for configuring GemBuilder. The Settings Browser is described on page 164.
	Breakpoints	Opens a Breakpoint Browser, allowing you to set and clear breakpoints in GemStone Smalltalk code. The Breakpoint Browser is described on page 198.
	System Workspace	Opens the GemStone System Workspace, a workspace containing a variety of useful GemStone Smalltalk and client Smalltalk expressions.
About GemBuilder	Opens a window providing the GemBuilder version and copyright information.	

11.2 The GemStone Session Browser

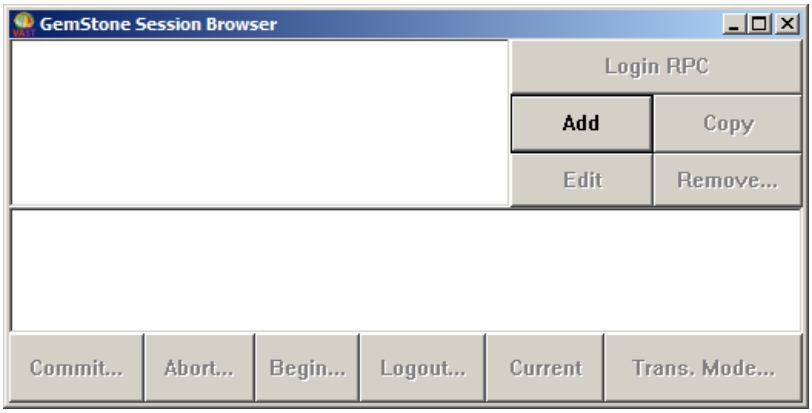
The GemStone Session Browser streamlines logging in and logging out of GemStone and managing sessions and transactions. This section explains how to invoke the Session Browser, and how to use it to define session parameters and to log in and out of GemStone.

Starting the Session Browser

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

   Figure 11.1 shows the Session Browser.

Figure 11.1 The GemStone Session Browser

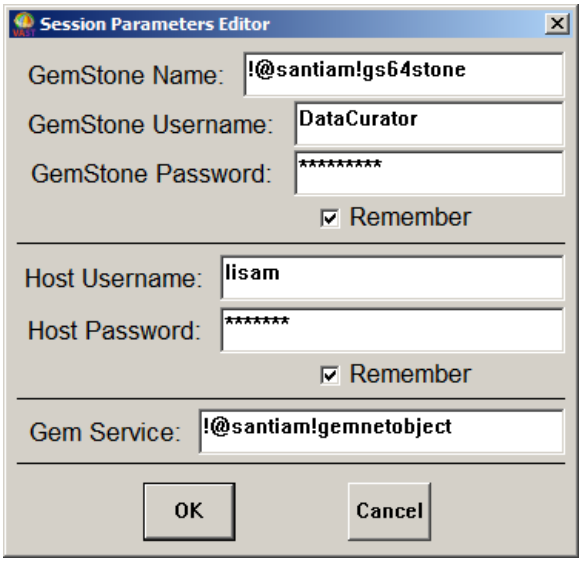


Opening the Session Parameters Editor

Select the **Add** button to define a set of session parameters. A Session Parameters Editor appears, as shown in Figure 11.2.

The first time this is done in a new image, the server-specific client libraries are loaded. Any problems in the client library configuration will show up now.

Figure 11.2 The Session Parameters Editor



Use the **Tab** key or the mouse to move through the fields in the login dialog, and the **Return** key to accept input or changes in the login dialog.

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 below), you must include the server's hostname in Network Resource String (NRS) format, as shown in Figure 11.2. (NRS format is described in an appendix to the *System Administration Guide for GemStone/S 64 Bit*.)
- **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 if netldi is run 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 netldi is run in guest mode.

- **Gem service**
  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
  ```

For maximum password security, leave the **Password** and **Host Password** fields empty, and the **Remember** boxes unselected.

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

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

Managing Session Parameters

Using the Session Browser buttons, you can manage your set of session parameters.

The Session Browser supports the following operations:

Table 11.2 Functions in the Session Browser

Add	Open an empty Session Parameters Editor.
Copy	Make a copy of the selected session parameters, and add it to the list.
Edit	Open a Session Parameters Editor on the selected session parameters.
Login RPC	Log in using the selected session parameters.
Remove...	Remove the selected session parameters

11.3 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.

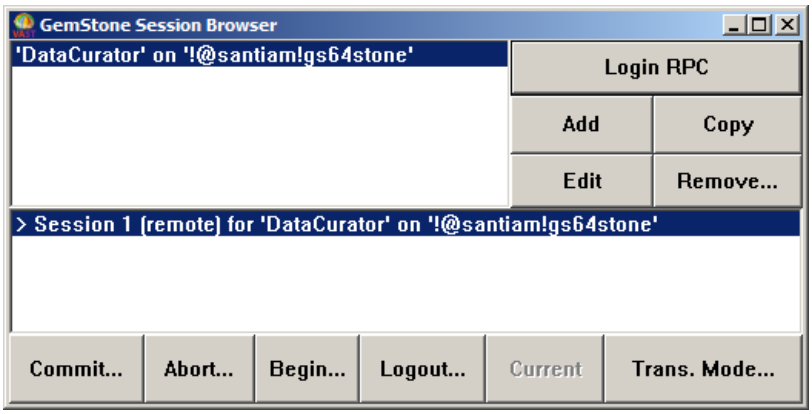
You can use the Session Browser to perform the same session management tasks that you can perform programmatically: log in to the GemStone server, view current sessions, set the current session, and log out of the GemStone server.

Logging In to GemStone

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

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

Figure 11.3 The GemStone Session Browser



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

Setting the Current Session

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

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

Logging Out of GemStone

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

Before logging out, GemBuilder prompts you to commit your changes, if 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.

The Settings Browser

The Settings Browser allows you to examine and set the configuration parameters for GemBuilder.

Opening the Settings Browser

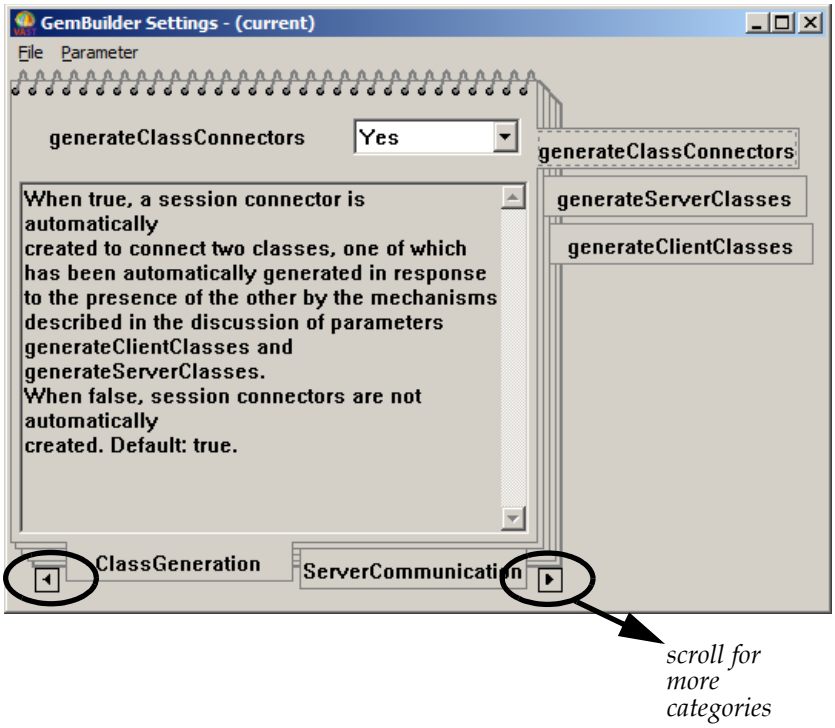
To open the Settings Browser, select **Tools > Settings** from the GemStone menu.

You can programmatically open a new Settings Browser by executing `GBSM ConfigurationTool new` in the client Smalltalk. The new tool will contain a copy of the values of the current configuration by default. To open the tool on its default configuration or on some other configuration use one of the following messages:

`open`	opens the tool on its current configuration
`openOn:` *aGbsConfiguration*	opens the tool on a copy of the given configuration
`openOnDefaults`	opens the tool on a copy of the default configuration
`openOnCurrent`	opens the tools on a copy of the currently-installed GemBuilder Configuration

The Settings Browser is shown in Figure 11.4.

Figure 11.4 The Settings Browser



Parameter Notebook

The Settings Browser uses a notebook metaphor to organize the various configuration parameters. Tabs on the bottom provide access to the categories of configurations, and tabs on the right side are for parameters within the selected category.

Menu items on the **File** menu allow you to load and save the settings contained in the notebook and to specify the parameter to be displayed.

Table 11.3 The File Menu

Load...	Provides a menu for selecting a source configuration. Menu options are:	
	Load From Current	Uses configuration values currently installed in GemBuilder.
	Load From Default	Uses the default configuration values.
	Load From Saved...	Brings up a dialog so you can enter the name of a saved configuration to use.
Save To Current	Installs the specified configuration's values in GemBuilder.	
Save...	Brings up a dialog in which you can select an existing named configuration or enter a new name.	
Configurations	Brings up a window that displays all named configurations and has buttons that allow you to delete or rename a setting and to open a Configuration Browser on a named setting.	
Find Parameter...	Provides a menu with the following choices:	
	By Name...	Shows a selection list of all parameters.
	Changed from Current...	Shows all parameters whose values differ from the configuration currently installed in GemBuilder.
	Changed from Default...	Shows all parameters whose values differ from the default configuration values.
Close	Close this dialog	

Each page of the notebook provides access to a single parameter, using the following fields:

- A label containing the name of the parameter.
- An editable field (a text entry field for Strings or Integers) or list of choices (when legal values have finite choices, e.g., true or false) that displays the current value of the parameter
- A text field containing a description of the parameter and its legal values and defaults

For each page the menu bar **Parameters** menu applies. This menu contains the following options.

Table 11.4 The Parameter Menu

Default Value	Copies the default value for that parameter into the entry field.
Revert Value	Copies the notebook's configuration value for that parameter into the entry field.

Parameter Categorization

The Settings Browser categories the parameters under tabs along the bottom. Selecting each heading provides access to individual related configuration parameters.

Table 11.5 Settings Browser Categorization

Cache Tuning	bulkLoad initialCacheSize initialDirtyPoolSize
Debugging	assertionChecks forwarderDebugging stubDebugging
Connectors	connectorNilling connectVerification removeInvalidConnectors
Class Generation	generateClassConnectors generateServerClasses generateClientClasses
Server Communication	libraryName blockingProtocolRpc fullCompression traversalBufferSize
Signals And Events	eventPollingFrequency eventPriority
User Interface	confirm verbose
Replication	autoMarkDirty blockReplicationEnabled blockReplicationPolicy defaultFaultPolicy faultLevelRpc freeSlotsOnStubbing

11.4 GemStone Workspaces

To open a GemStone workspace, choose **GemStone > Tools > New GS Workspace**. You can also open an existing file as a GemStone workspace using **GemStone > Tools > Open GS Workspace...**.

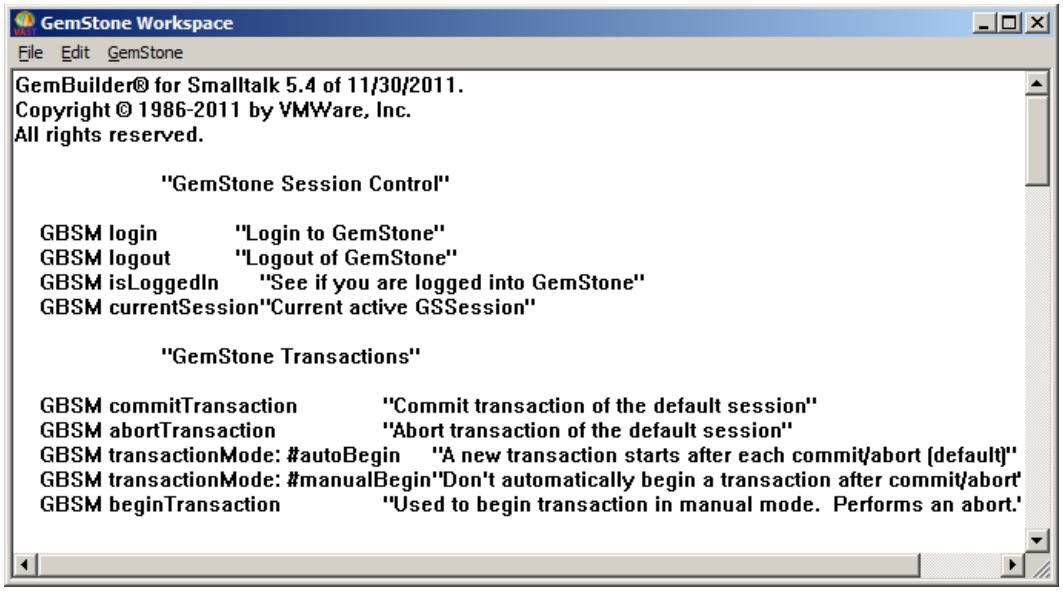
In a GemStone workspace, you can execute, display, inspect, or file in GemStone Smalltalk code using the **GemStone** menu, as well as perform these functions on client Smalltalk code using the **Edit** menu.

11.5 The System Workspace

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

To open a GemStone System Workspace (Figure 11.5), choose **GemStone > Tools > System Workspace** from the **GemStone** menu.

Figure 11.5 GemStone System Workspace



Chapter

12 Using the GemStone Programming Tools

After you install GemBuilder, many menus in your Smalltalk image contain additional commands for executing GemStone Smalltalk code and accessing GemBuilder programming tools. GemStone also provides GemStone Smalltalk debugging facilities similar to the debugging aids supplied by the client Smalltalk.

These tools are in many ways similar to those of the client Smalltalk, but with important differences. This chapter describes those differences.

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

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

The Connector Browser
explains how to use connectors in code or using the Connector Browser.

The Class Version Browser
describes a specialized Class Browser that can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

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

Breakpoints
describes breakpoints, setting breakpoints, and using the Breakpoint Browser

Debugger
describes GemBuilder's enhanced debugger

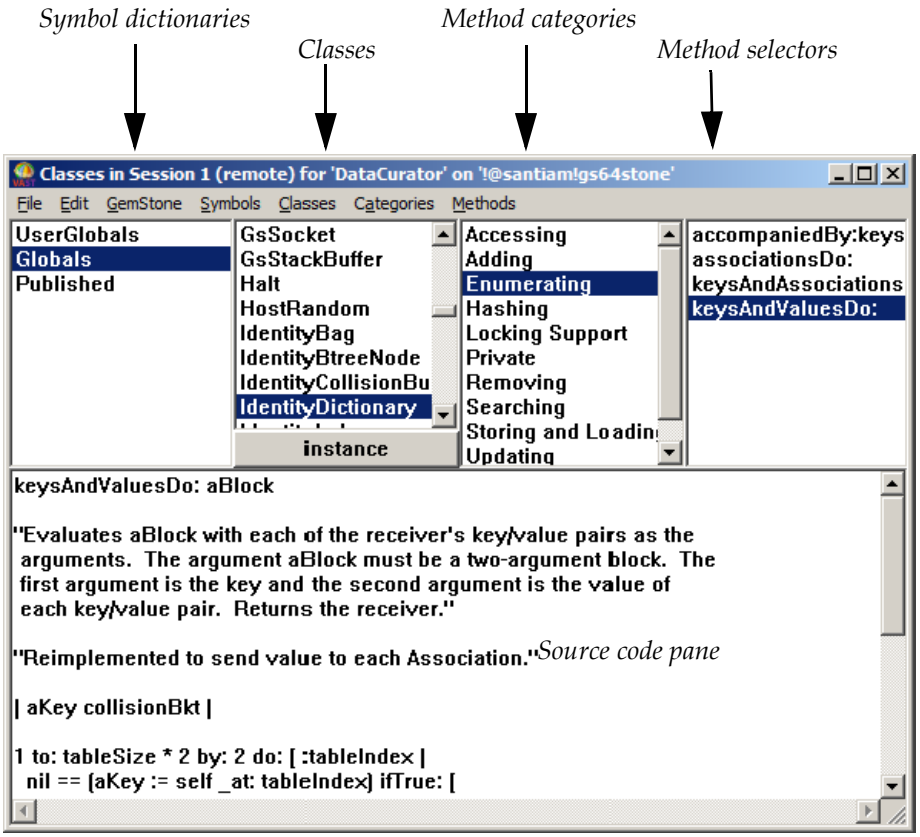
Stack Traces
describes GbsStackDumper, GemBuilder's enhanced stack dumping facility

12.1 Browsing Code

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

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

Figure 12.1 GemStone Classes Browser



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

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

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

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

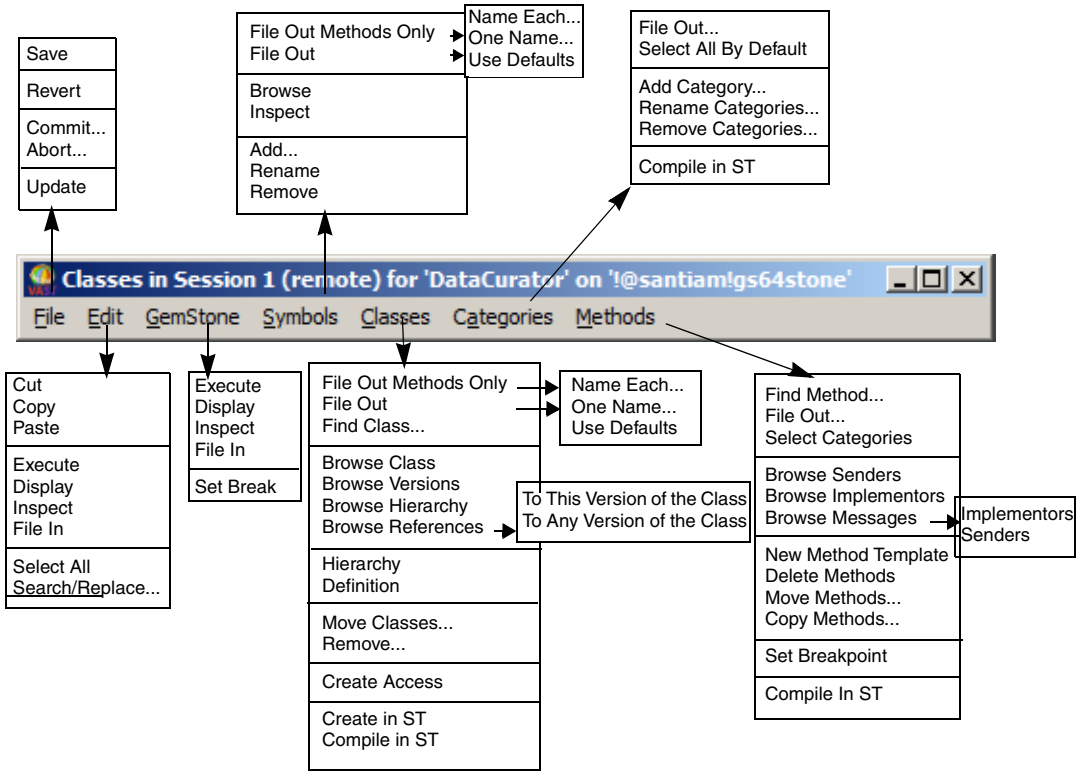
As in the comparable client Smalltalk browsers, you can switch focus between **instance** or **class** methods using the toggle button provided.

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

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

Figure 12.2 shows all the menus available from the GemStone Browser's menu bar.

Figure 12.2 Menus in the GemStone Browser



The following sections describe GemStone-specific commands in the drop-down menus.

The File Menu

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

Table 12.1 File Menu in the GemStone Browser

Commit	Attempts to save to the GemStone repository all modifications that occurred during the current GemStone transaction.
Abort...	Undoes all changes that you have made in the GemStone
Update	Update the view of the browser to the current state in the image.

The GemStone Menu

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

Table 12.2 GemStone menu in the GemStone Browser

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

Symbol List Menu

The following commands are available from the **Symbols** menu

Table 12.3 GemStone Browser's Symbol List Menu

File Out Methods Only	Prompts you for a file name under which to save all the methods in all the classes in the selected SymbolDictionary.
File Out	Prompts you for a file name under which to save all classes and methods definitions for all the classes in the selected SymbolDictionary.
Browse	Spawns a Dictionary Browser on the selected SymbolDictionary.
Inspect	Open an inspector on the selected SymbolDictionary.
Add...	Add a new Symbol Dictionary
Rename...	Rename the selected SymbolDictionary.
Remove...	Remove the selected SymbolDictionary. WARNING: Do not remove Globals.

Class Menu

The following commands are available from the **Classes** menu. A later section discusses the procedure required to add the definition of a new GemStone class to the currently selected symbol dictionary.

Table 12.4 Class Menu in GemStone Browser

File Out Methods Only...	Prompts you for a file name under which to save all methods of the selected class or classes.
File Out	Prompts you for a file name under which to save all classes and methods definitions for the selected class or classes.
Find Class...	Navigate to a specific class by name. The search string is case sensitive, and can include wild cards.
browse class	Open a Class Browser on the selected class.

Table 12.4 Class Menu in GemStone Browser(Continued)

Browse Class	Open a class browser on the selected class.
Browse Versions	Open a class version browser on the selected class.
Browse References	Open a method list on all references to the current version of the class, or all versions of the class.
Browse Hierarchy	Open a hierarchy browser on the selected class.
Hierarchy	Display the class hierarchy in the text pane.
Definition	Display the class definition in the text pane (the default)
Comment	Display the class comment in the text pane.
Move Classes...	Prompt for another SymbolDictionary to which to move the selected class.
Remove...	Remove the selected class.
Create Access	Creates methods for accessing and updating the instance variables of the selected class.
Create in ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, if one doesn't already exist. If it does exist, executing this menu item has no effect.
Compile in ST	Creates a client Smalltalk class having the same name and structure as the selected GemStone Smalltalk class, and compiles all currently defined methods for the class. If necessary, a notifier lists any methods that cannot be compiled in client Smalltalk.

Pop-up Text Pane Menu

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

The GemStone-specific commands available from a text area pane are shown in Table 12.5.

Table 12.5 Pop-up Menu in GemStone Browser's Text Pane

GS-execute	Executes the code in GemStone.
GS-display	Executes the code in GemStone and displays the result in the text area.
GS-inspect	Executes the code in GemStone and opens an inspector on the result.
GS-file In	Files the selected text into GemStone.
Set Break	Sets a breakpoint at the step point nearest the cursor location. If the cursor is not exactly at a step point, scans the method from the current cursor location on and sets a breakpoint at the next step point. See page 196 for a full discussion of using breakpoints.

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

Table 12.6 Additional GemStone Menu Items

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

12.2 Coding

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

About GemStone Smalltalk Classes

The process of creating classes in GemStone Smalltalk differs somewhat between GemStone server products and versions. For information specific to your server version, refer to the *GemStone/S 64 Bit Programming Guide* and to the subclass creation methods in the GemStone image.

Invariance

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

You can include the symbol `#instancesInvariant` in the Array passed to the `options:` keyword, or use subclass creation protocol with the `instancesInvariant:` keyword.

Classes themselves may also be invariant or not. Classes that are variant can be modified, e.g. you may add and remove instance variables; but you cannot create instances of a variant class. By default, class creation results in invariant classes.

You can include the symbol `#modifiable` in the Array passed to the `options:` keyword, or use subclass creation protocol with the `isModifiable:` keyword, to create modifiable classes. Sending `immediateInvariant` makes the class invariant and allows instance creation.

Non-persistent

In GemStone/S 64 Bit, you may also specify on a per-class bases that instances of the class are not persistent, meaning they cannot be committed to the repository; of that instances are dbTransient, in which the instances may be committed, but any data stored in instance variables of the instance is not persistent. Refer to the *GemStone/S 64 Bit Programming Guide* for details on `#instancesNonPersistent` and `#dbTransient`.

Versions

GemStone Smalltalk classes have a classHistory, which provides versioning for classes. Defining a class with the same name as an existing class and in the same symbol dictionary automatically creates a version of the existing class. When multiple versions of a class exist, only the latest is displayed in the browsers, and the display includes the sequence number of the class within the class history, in brackets; for example, `Employee[2]`.

Class creation can explicitly create or not create class versions using subclass creation protocol that uses the `newVersionOf:` keyword. For more details on class versions, see the chapter entitled "Class Creation, Versions, and Instance Migration" in the *GemStone/S 64 Bit Programming Guide*.

Defining a New Class

To define a new GemStone class:

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

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

The browser displays the class definition template:

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

This is the basic form of the subclass creation message in GemStone/S 64 Bit version 3.0 and later.

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

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

Step 5. In the parentheses following the `instVarNames:` keyword, supply the names of any instance variables. A class can define up to 255 named instance variables.

Step 6. In the parentheses following the `classVars:` keyword, supply the names of any class variables.

Step 7. In the parentheses following the `classInstVars:` keyword, supply the names of any class instance variables.

Step 8. Fill in the brackets after the `poolDictionaries:` keyword with any pool dictionaries that you want the class to access. Pool dictionaries are special-purpose storage structures that enable any arbitrary group of classes

and their instances to share information. When classes share a pool dictionary, methods defined in those classes can refer to the variables defined in the pool dictionary. Note that the curly braces syntax for Arrays is not understood in 32-Bit GemStone/S.

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

Step 10. In the parentheses following the `options:` keyword, you can specify a collection of symbols to define specific features of the new subclass. `options:` is available only in GemStone/S 64 Bit 3.0 and later. These options are special purpose and not commonly used; for details, see the "Class Creation" chapter of the *GemStone/S 64 Bit Programming Guide*.

Step 11. Accept or save your changes and commit your transaction to make the class part of the repository.

NOTE

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

For example, consider the following definition of a class named Employee. This creates a class with the given instance and class variables, and put the class in the UserGlobals symbol dictionary.

Example 12.1

```
Object subclass: 'Employee'
      instVarNames: #( 'name' 'employeeNum' 'jobTitle' 'department'
                       'address')
      classVars: #('AllDepartments')
      classInstVars #()
      poolDictionaries: {}
      inDictionary: UserGlobals
      options: #()
```

Subclass Creation Methods

There are a variety of subclass creation messages, depending on the type of class you want to create. Subclass creation methods that begin with the keyword `byteSubclass:` or `indexableSubclass:` create classes that store data in indexed slots, rather than limited to instance variables.

Storage format is inherited, so if the superclass is already byte or indexable format, however, subclass creation methods that begin with the keyword `subclass:` create a subclass of the same storage format.

For a full list of available subclass creation methods for your server product and version, refer to the GemStone Smalltalk image.

For complete descriptions of the different kinds of classes, see the *GemStone/S 64 Bit Programming Guide* chapter on Class Creation.

Private Instance Variables

Some GemStone kernel classes have private instance variables. For example, the superclass of GemStone Bag class defines four, used by the object manager and primitives to implement features of nonsequenceable collections, such as adding indexing structures for efficient querying. Private instance variable names begin with an underscore (_).

Modifying an Existing Class

If you select an existing GemStone Smalltalk class, then modify and save the class definition, you create a new version of that class and all of its subclasses. The browser attempts to recompile all methods from the previous version into the new version. Methods that fail to recompile are presented in a method list browser, from which you can correct the errors. If the class has subclasses, they are also versioned and their methods recompiled.

When you modify an existing class, the tools will ask if you wish to commit the transaction and migrate all instance to the new version of the class. If you choose not to do this, you can migrate some or all instances of one version of a class to another version explicitly.

For more information on migrating instances, see the chapter entitled "Class Versions and Instance Migration" in the *GemStone/S 64 Bit Programming Guide*.

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

To create a new version of a class:

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

Step 2. Edit the definition as required.

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

Whenever you create a class with the same name as a class that already exists in the same symbol dictionary, the new class is automatically created as the latest version of the existing class and it automatically shares the same class history. Instances created after the redefinition have the new class's structure and access the new class's methods. Instances that were created earlier have the old class's structure and access the old class's methods, but they can be migrated to the new class.

Let's assume that you have a class named Employee with instance variables for `name`, `employeeNum`, `jobTitle`, `department`, and `address`, and that the class is defined as shown in Figure 12.1. Suppose that you decide that the class needs an additional instance variable named `salary` to represent the Employee's salary.

To do this, you can define a new version of the class Employee to include the new instance variable. Keeping the same name as the old class ensures that it shares the same class history as the previous version.

After you compile the class definition, the new class is named Employee, and all of the original instance and class methods are copied to the new class. Any existing instances will still belong to the original class and may have to be migrated to the new class.

Defining Methods

You can modify only methods for which you have write authorization— for example, methods that you have written for your own classes. You cannot modify any GemStone kernel class method—that is, any method that is defined for one of the predefined classes supplied with the GemStone system.

Public and Private Methods

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

Private GemStone methods are those whose selector is prefixed with an underscore (_), or that explicitly say they are private within the method comment.

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/S 64 Bit Programming Guide* for that version, Appendix A.

Saving Class and Method Definitions in Files

It's often useful to store the GemStone Smalltalk source code in text files. Such files make it easy to:

- transport your code to other GemStone systems,
- perform global edits and recompilations,
- produce paper copies of your work, and
- recover code that would otherwise be lost if you are unable to commit.

To save GemStone code in a file, use any of the GemStone browser's **file out** menu items. To read and compile a saved file, use any of the **Gs-File in** or **GS-File it in** menu items (in your client Smalltalk browser).

Saved GemStone files are written as sequences of Topaz commands. Example 12.2 shows a class definition in Topaz format:

Example 12.2

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

```
%

! Remove existing behavior from Address
doit
Address removeAllMethods.
Address class removeAllMethods.
%
! ------------------- 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
%
```

GemStone's filing out and filing in facilities are intended mainly for saving and restoring classes and methods without manual intervention. If this is all you want to do, then you don't need to understand the Topaz commands involved. However, it is also possible to create custom files that include commands to commit transactions and to create and manipulate objects other than classes and methods. If you want to perform such tasks, refer to the *Topaz Programming Environment*.

The file-in mechanism cannot execute the full set of Topaz commands. File-in is limited to the following subset:

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

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

```
display          omit
expectvalue      output
level            remark
limit            status
list             time
```

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

The file-in mechanism does not display execution results, either. Instead, it appends information to the System Transcript about the files it reads and the classes and categories for which it compiles methods.

Handling Errors While Filing In

If one of the modules (run commands or method definitions) that you're filing in contains a GemStone Smalltalk syntax error, GemStone displays a compilation error notifier that contains the erroneous module in a text editor. If you correct the error and then choose **Save**, GemStone recompiles the module and then processes the rest of the file.

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

12.3 The Connector Browser

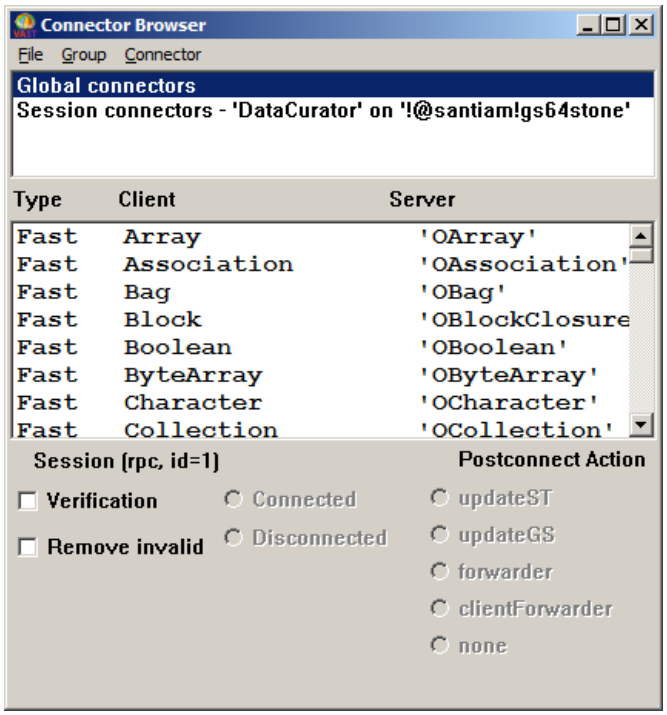
Chapter 4 describes *connectors*, which allow an application developer to explicitly declare an association between a root client object and a root server object. This section explains how to use GemBuilder's Connector Browser to make and manage connectors interactively.

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

- examine, create, and remove global or session-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; and
- examine or modify the postconnect action associated with a connector.

Figure 12.3 shows the Connector Browser.

Figure 12.3 The Connector Browser



The Group Pane

The top pane is the Group pane; it allows you to select either global connectors or those associated with an individual session. Global connectors are predefined to connect the GemStone 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.

The File and Group Menu Bar menus, and the Group pane popup menu, provide the following items:

Table 12.7 Group Menu in the Connector Browser

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

The Connector Pane

The middle pane is the Connector pane; it lists the connectors, their types, and descriptions in both the client and GemStone server Smalltalks.

The Menu Bar Connector Menu, and the Connector pane popup menu, offer the following items:

Table 12.8 Connector Menu in the Connector Browser

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, connect or disconnect objects, and set a connector's postconnect action.

Table 12.9 Options in the Control Panel

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

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

Postconnect Action

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

Table 12.10 Postconnect Action Options in the Connector Browser

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

To create a new connector:

1. Select the session in the Group pane.
2. Place the cursor in the Connector pane.
3. Select **add** from the menu.
4. When prompted, specify the type of connector.
5. When prompted, specify the names of the client and server objects.
6. When prompted, specify the name of the dictionary for the server object.
7. Specify the postconnect action.

To create a forwarder:

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

To change the postconnect action:

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

If your application initially stores its data in the client, and you intend to store the data 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.

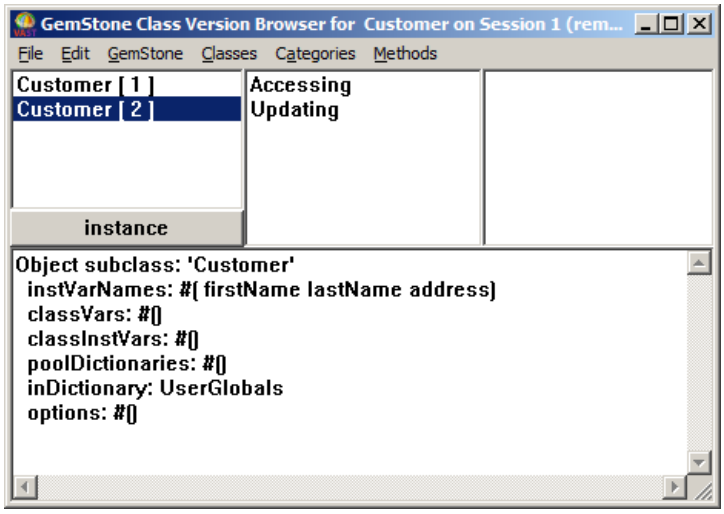
12.4 The Class Version Browser

The Class Version Browser is a specialized Class Browser that can be used for examining a class history, inspecting instances, migrating instances, deleting versions, and moving versions to another class history.

To open a Class Version Browser, select a class in a browser and choose **Browse Versions** from the Classes menu. If more than one version of a class has been created, the class list in the spawned browser displays the version number next to the class name.

A Class Version Browser is shown in Figure 12.4.

Figure 12.4 The Class Version Browser



Menus in the Class Version Browser

For the most part, the Class Version Browser's menus are the same as the menus in the Class Browser. However, the Class Version Browser's Classes menu contains the additional items **Inspect Instances** and **Migrate Instances...**. Also, note that the Classes menu items **Move Class...** and **Remove...** behave differently in this browser.

The layout of the browser is similar to the Class Browser. The Method Category and Message menus are the same as in a spawned Class Browser. The Classes menu, however, has additional functionality.

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

The commands available in the Class Version Browser are shown in Table 12.11:

Table 12.11 Class Menu in Class Version Browser

File Out Methods Only	Writes GemStone Smalltalk code defining the selected class version or versions' methods to be written to fa file or files in Topaz format. This menu item has three submenus controlling the name of the destination file: **Name Each....** prompts for a name for each file, **One Name...** prompts for a single file name, and **Use Defaults** uses a default name based on the class name.
File Out	Writes GemStone Smalltalk code defining the selected class version or versions' class definitions and all of its methods to be written to a file in Topaz format. This menu item has three submenus controlling the name of the destination file: **Name Each....** prompts for a name for each file, **One Name...** prompts for a single file name, and **Use Defaults** uses a default name based on the class name.
Browse Class	Opens a Class Browser that includes only the selected class version.
Browse Versions	Opens another Class Version Browser on this class history.
Browse Hierarchy	Opens a Class hierarchy Browser that includes superclasses and subclasses of the selected class version.

Table 12.11 Class Menu in Class Version Browser(Continued)

Browse References	This menu item has two submenus: **to this version of this class** and **to any version of this class**. Opens a method list browser containing all methods whose compiled code contains a reference to this version of the class, or to any version of the class in this class history.
Hierarchy	Lists the superclasses and subclasses of the current class. Any instance variable names declared in a class appear in the hierarchy list in parentheses.
Definition	Displays the definition (that is, the subclass creation message) of the currently selected GemStone class version. This is shown by default.
Comment	Displays the class comment.
Move Classes...	Moves the selected class version to another class history. Prompt for a target class, adds the selected version to the target class's class history, and updates the browser. The class name of the selected version is changed to that of the target class.
Remove...	Remove the selected class version from the class history. Upon confirmation to proceed, asks if the user wants to migrate instances. If yes, prompts for the migration target, migrates the instances and updates the browser.
Create Access	Creates methods for accessing and updating the instance variables of the currently selected class version.
Create in ST	Generate the selected class in client Smalltalk, if a mapping doesn't already exist. If it does exist, executing this menu item has no effect.
Compile in ST	Attempts to compile all methods (instance and class) of selected class version in corresponding client Smalltalk class.
Inspect Instances	Open an inspector on instances on the selected version.

Table 12.11 Class Menu in Class Version Browser(Continued)

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.

12.5 Debugging Overview

GemBuilder's debugging tools assist you in examining and modifying application objects during execution. These facilities enable you to perform the following operations:

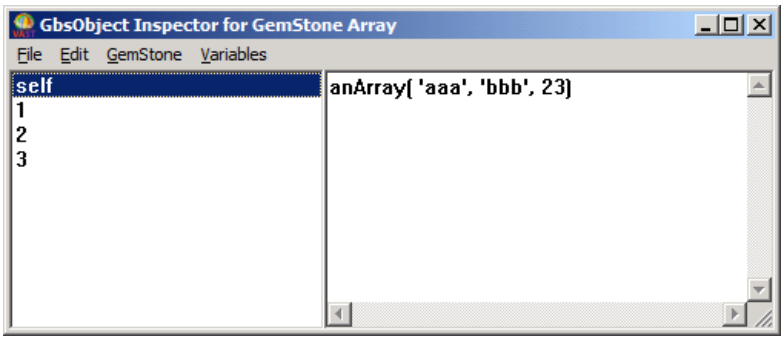
- You can view and alter the instance variables of server objects.
- You can step through execution of a method, examining the values of arguments, temporaries, and instance variables after each step.
- You can set, clear, and examine GemStone Smalltalk breakpoints. When a breakpoint is encountered during normal execution, a notifier appears and you can open a debugger with which you can interactively explore the contexts in the stack at the time execution halted.
- You can inspect or change the values of arguments, temporaries, and receivers in any context (stack frame) on the virtual machine call stack, then continue execution from the top of the stack. This means that you can find out what the system was doing at the time a breakpoint, or an error interrupted execution.
- You can execute a message expression within the scope of a given context.

12.6 Inspectors

To allow you to examine the values of GemStone server objects and modify them when appropriate, GemBuilder provides inspectors that are similar to the client Smalltalk inspectors. When you select a GemStone Smalltalk expression and execute **GS-Inspect**, a GemStone inspector opens. The GemStone inspector

(Figure 12.5) is similar to a client Smalltalk inspector; it has comparable panes and functionality.

Figure 12.5 GemStone inspector

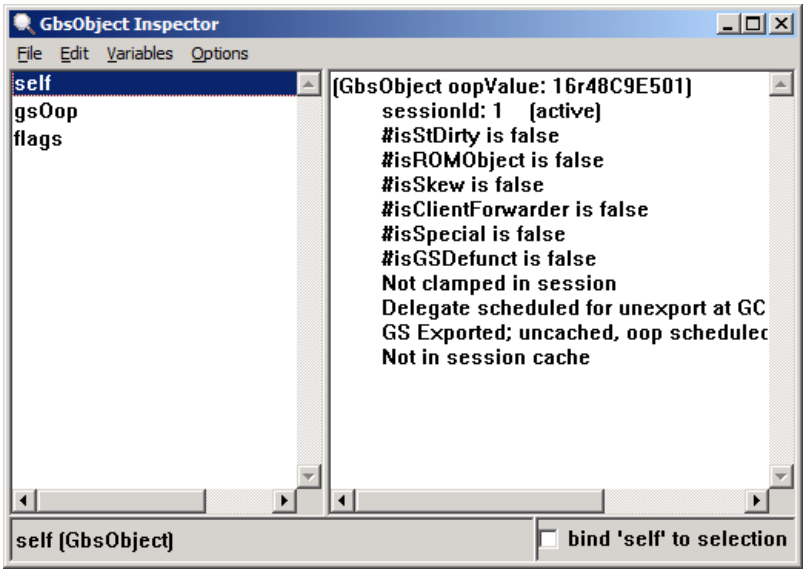


. The inspector contain the following GemStone-specific command:

Table 12.12 Commands in GemStone Inspector

Basic Inspect	Opens an inspector on the delegate object, an instance of GbsObject (see Figure 12.6).

Figure 12.6 GemStone Inspector Basic Inspect



Inspecting UnorderedCollections

When you're inspecting an instance of any nonsequenceable collection, which includes any subclasses of the GemStone server class UnorderedCollection, the following additional menu items are available.

Table 12.13 Commands for Inspecting NSCs

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

12.7 Breakpoints

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

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

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

Example 12.3 indicates step points with numbered carets.

Example 12.3

```
includesValue: value
^1

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

| found index size|

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

If you use the GemStone debugger (described starting on page 199) to step through this method, the first step takes you to the point where `includesValue:` is about to be sent. Stepping again sends that message and halts the virtual machine at the point where `found` is assigned. Another step sends that message and halts the virtual machine just before the result is assigned to `index`, and so on.

When the GemStone Smalltalk virtual machine encounters an enabled breakpoint during normal execution, GemStone displays a notifier, in which selecting **Debug** in the notifier opens a GemStone Debugger. In the Debugger, you can interactively explore the context in which execution halted.

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

Breakpoints for Primitive Methods

If you set a breakpoint in a primitive method, the break is encountered only if the primitive fails. Consider the method below:

```
= aString

<primitive: 160>
self _primitiveFailed: #=
```

When this method is invoked, GemStone first executes the machine code in primitive 160. If that code executes successfully, the primitive is said to succeed, and the method returns a value. Because no GemStone Smalltalk code has yet been encountered, the virtual machine has not yet reached the first step point. Only if the primitive fails will the virtual machine execute the message-send at the bottom of the method and thus encounter the breakpoint.

Breakpoints for Optimized Methods

Very simple methods are optimized by the GemStone Smalltalk compiler in such a way that they contain no step points. Naturally, you cannot set a method breakpoint if there are no step points. Methods that just returns `true`, `false`, **`nil`**, `self`, that set or assign to an instance variable, or that return a class or pool variable or a variable defined in a symbol dictionary, are optimized in this way, and have no step points.

Also, certain selectors, such as `==`, `ifTrue:`, and `to:do:`, are optimized by the compiler, and cannot take step points. Optimized selectors vary by server product and version, and are listed in the *GemStone/S 64 Bit Programming Guide*, Appendix A.

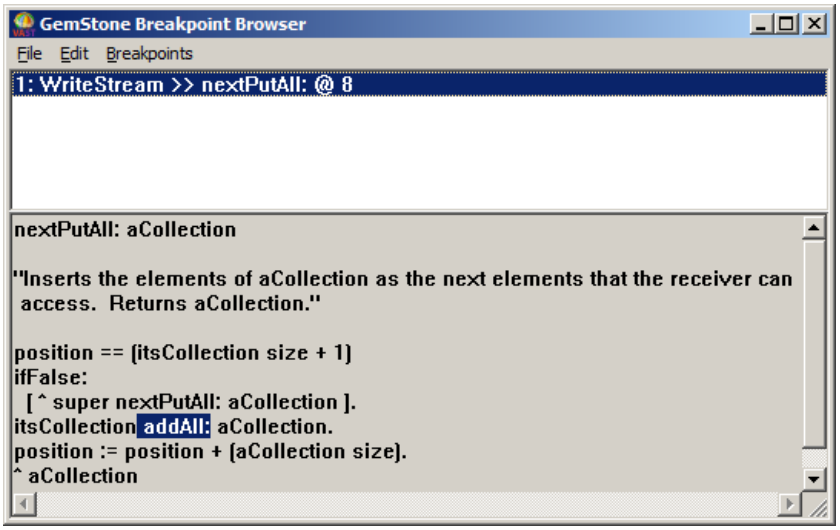
The Breakpoint Browser

You can set breakpoints in the source code pane of any browser, using the **Set Break** menu item described in Table 12.5 on page 177. You can also use the breakpoint browser, which lets you set, clear, and examine breakpoints for all classes and methods.

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

A breakpoint browser has two panes: the list of break points on top, and the source code associated with the selected breakpoint on the bottom. Figure 12.7 shows an example:

Figure 12.7 GemStone Breakpoint Browser with a Breakpoint



The Break Pane

The break pane displays a scrollable list of the active breakpoints. The items in the list look like this:

```
1: WriteStream >> nextPutAll: @ 8
```

In this example, a method break is set at step point 8 within the method `nextPutAll:` defined by class WriteStream.

The Source Pane

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

12.8 Debugger

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

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

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.

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

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

Like other GemBuilder text areas, the debugger source code pane provides commands to execute GemStone Smalltalk.

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 Smalltalk client debugger.

12.9 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

```
GbsConfiguration 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 10-digit number derived from a time stamp.

Chapter

13 Using the GemStone Administration Tools

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

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

The 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.

13.1 The Security Policy Tool

The Security Policy Tool 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 119, 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 Tool also allows you to examine and change group membership.

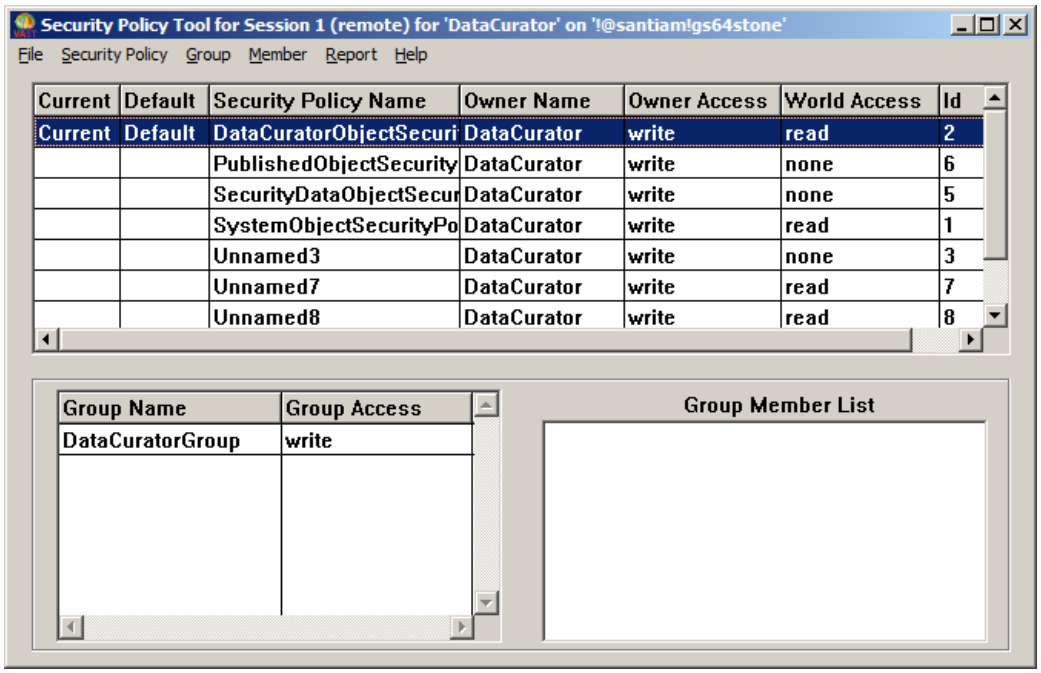
NOTE

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

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

To open a Security Policy Tool, select **Admin > Security Policies** from the GemStone menu or through the User Dialog's **Object Security Policies** button. Figure 13.1 shows a Security Policy Tool.

Figure 13.1 The Security Policy Tool



The Security Policy Tool is divided into two main sections. The upper section displays security policies. The lower section displays groups and their members.

Security Policy Definition Area

The security policy definition area at the top of the dialog displays the security policies in the SystemRepository for which the current user has read authorization.

You will notice that some security policies are named and some are unnamed. Named security policies are security policies that are referenced in a dictionary or symbol list. Unnamed security policies are those that are not referenced in any dictionary or symbol list.

In addition to the security policies displayed in the Security Policy Tool, all users also have read and write authorization to

GsIndexingObjectSecurityPolicy. Because authorization changes should not be made to that security policy, however, it is not included in the tool.

NOTE

Changes made to cells in the tables are accepted automatically as soon as you either press Return, make a selection in a combo box associated with the cell, or simply move the focus to another cell or field by moving the mouse. 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 upper portion), you can change the following:

Current — You can set the security policy to be your current security policy. When you create an object, GemStone assigns it to your current security policy.

Default — You can set the security policy to be your default security policy. This is the home security policy that is your current security policy when you log into GemStone.

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

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

- **read** means that a user can read any of the security policy's objects, but can't modify (write) them or add new ones
- **write** means that a user can read and modify 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 security policy or a default security policy. If the account does not have write authorization in its default security policy, the account cannot log in.

Security Policy Id — The Id number of each security policy is displayed. This information cannot be modified.

Group Definition Area

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

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

In the group definition area you can change the following:

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

Group Access — Group access can be changed in the same way as owner and world access. To change group access, type either **read** or **write** into the cell, as outlined for owner and world access on page 204.

NOTE

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

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

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

Security Policy Tool Menus

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

The File Menu

Use the **File** menu to commit work done in the Security Policy Tool, to abort the transaction, or to update the tool's view of security policies, groups, and users in the current session.

Table 13.1 File Menu in the Security Policy Tool

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

Security Policy Menu

Use the **Security Policy** menu to create new security policies and to manipulate existing security policies.

Table 13.2 Security Policy Menu in the Security Policy Tool

Create...	Creates a new security policy. You must have the Security Policy Creation privilege to use this option. In the Create Security Policy dialog, enter a name for the security policy and a symbol dictionary to store it in. Private security policies are typically kept in UserGlobals. Security policies for large groups of users are typically kept in Globals.

Table 13.2 Security Policy Menu in the Security Policy Tool(Continued)

Grab	Grabs a reference to the selected security policy and places it on the clipboard. This can be used to add a reference to a user's symbol list or for changing the default security policy of a user in the User Dialog.
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 home security policy that is your current security policy when you log into GemStone.

Group Menu

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

Table 13.3 Group Menu in the Security Policy Tool

Add...	Adds a new group. In the Add Group dialog, enter a name for the group and choose **OK** or **Apply**.
Remove...	Removes authorization for the selected group. This does not delete the group from GemStone. It only means that the current 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 Menu

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

Table 13.4 Member Menu in the Security Policy Tool

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

Reports Menu

Use the **Reports** menu to bring up a window displaying information about the security policies, users, and groups in your view of the repository.

Table 13.5 Report Menu in the Security Policy Tool

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 as to • its owner, • the groups for which it contains access information, and • the access it grants to the owner, groups, and world. This report includes the GsIndexingObjectSecurityPolicy, for which all users have read and write authorization.
User Report	Produces a list of all GemStone users and shows each user's group memberships.

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

Help Menu

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

Using the Security Policy Tool

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 for GemStone/S 64 Bit*.

Checking Security Policy Authorization

Anyone who has read authorization for a security policy can use the Security Policy Tool to find out who is authorized to read or write that security policy by doing the following:

1. Bring up the Security Policy Tool by selecting **GemStone > Admin > Security Policies** or by choosing **Object Security Policies** in a GemStone User Dialog.
2. In the Security Policy Tool, 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 either have Security Policy Protection privileges or are the security policy's owner, you can use the Security Policy Tool to change the authorization of a security policy.

The top half of the Security Policy Tool 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 tool.
- 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.

13.2 The Symbol List Browser

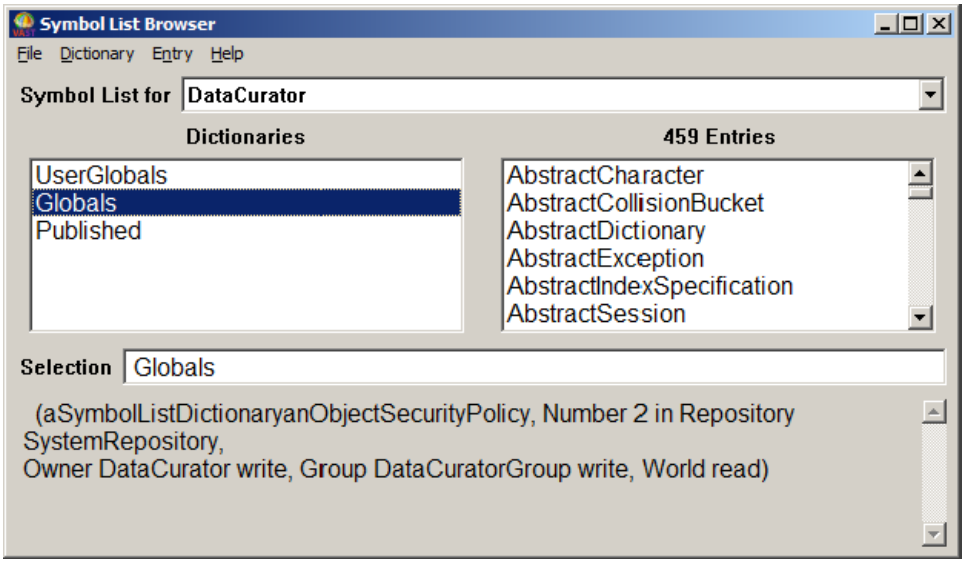
The Symbol List Browser is a tool for examining the GemStone SymbolLists associated with UserProfiles, adding and deleting dictionaries from these lists, examining the entries in those dictionaries and adding, deleting and inspecting the entries. References to dictionaries and dictionary entries can be copied between GemStone user accounts, subject to authorization and 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 **Admin > Namespaces** from the Gemstone menu, or click on the **Symbol List** button on a GemStone 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 13.2 shows the Symbol List Browser.

Figure 13.2 The Symbol List Browser



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

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

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

The Clipboard

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

Table 13.6 File Menu in the Symbol List Browser

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

Dictionary Menu

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

Table 13.7 Dictionary Menu in the Symbol List Browser

Cut	Removes the selected dictionary from the user's symbol list and places it in the session's clipboard.
Copy	Copies a reference to the selected dictionary into the session's clipboard.
Paste	Causes the reference to the dictionary object in the clipboard to be added to the SymbolList in the pane, with the name it had when it was put in the clipboard. If the name of the dictionary in the clipboard is already in use in the destination list, a Confirmer will pop up to allow replacing the old item, or to abort the paste operation.
Add...	Prompts for name of a new Dictionary to be added to the symbol list.
Inspect	Opens a GemStone inspector on the selected Dictionary.

Entry Menu

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

Table 13.8 Entry Menu in the Symbol List Browser

Cut	Removes the selected entry from its dictionary and places it in the session's clipboard.
Copy	Copies a reference to the selected entry into the session's clipboard.
Paste	Causes the reference to the entry in the clipboard to be added to the selected dictionary, with the name it had when it was put in the clipboard. If the clipboard entry's name is already in use in the destination list, a Confirmer will pop up to allow replacing the old item, or to abort the paste operation.
Add...	Prompts for name of a new entry to be added to the selected Dictionary.
Inspect	Opens a GemStone inspector on the selected entry.
Browse Class	If the selected entry is a class, opens a GemStone class browser on that entry.

Help Menu

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

13.3 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 GemStone User List, the GemStone User Dialog, and the Privileges Dialog.

NOTE

To perform most of the system administration functions described in this section, you must either be DataCurator or have certain privileges.

If you are responsible for GemStone system administration, refer the chapter on administering user accounts and security policies in the *System Administration*

Guide for GemStone/S 64 Bit for specific information on user account management. That chapter discusses the privileges you need to manage user accounts and explains how to add and remove users, set up user environments, change passwords and user privileges, and how to add and remove users from groups.

GemStone User List

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

- To bring up the GemStone User List from the **GemStone** menu, select **Admin > Users**.

Figure 13.3 GemStone User List

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

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

Table 13.9 GemStone User List: File Menu

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

Table 13.10 shows operations available in this dialog.

Table 13.10 GemStone User List: Users Menu

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

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

GemStone User Dialog

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

The GemStone User Dialog is shown in Figure 13.4.

Figure 13.4 GemStone User Dialog

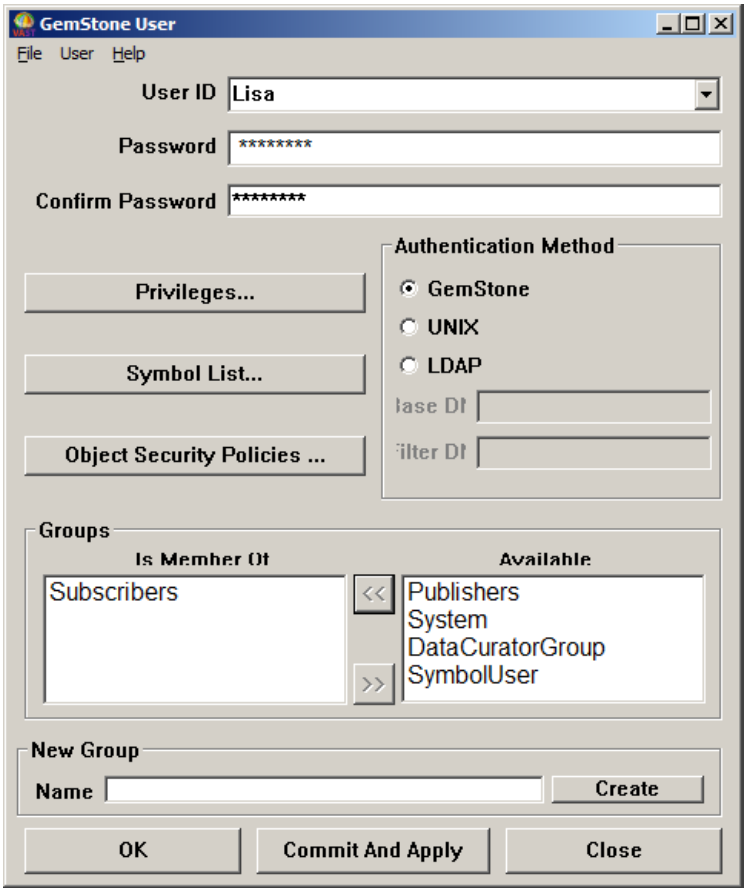


Table 13.11 shows the operations that are available in this dialog.

Table 13.11 Buttons in the GemStone User Dialog

Privileges...	Brings up a Privileges Dialog (page 219), in which you can select privileges for this user.
Symbol List...	Brings up a Symbol List Browser (page 210) for the designated user.
Object Security Policies...	Brings up a Security Policy Tool (page 202).
Authentication Method	Click the button to indicate the method for performing authentication for the selected 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 for GemStone/S 64 Bit*.
Create	In the New Group area 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.

- To **add a user to a group**, select the group in the Available list and use the **<<** button to move it to the Is Member Of list.
- To **remove a user from a group**, select the group in the Is Member Of list and use the **>>** button to move it back to the Available list.

The User Dialog has three menus: **File**, **User**, and **Help**. The **File** menu contains the following items.

Table 13.12 GemStone User Dialog: File Menu

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

The **User** menu contains one item, **Rename...**. This requests a new name for this user. You may not rename the DataCurator, GcUser, or SystemUser accounts.

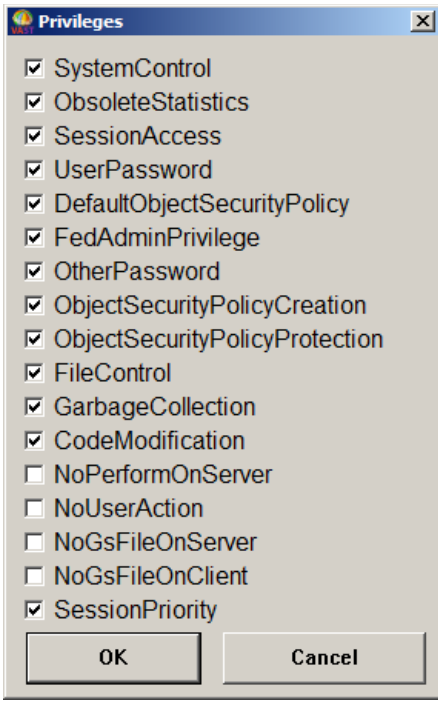
The **Help** menu contains one item, **Session Info**, which provides information about the session for the GemStone User List and about the current session.

Privileges Dialog

Certain system functions are customarily performed by the DataCurator; for example, many of the messages to System require explicit privilege to use. The privileges dialog displays the privileges an individual user possesses. You can use this dialog to examine a user's privileges, and—if you have the authority to do so—to select privileges for a user.

The Privileges Dialog is shown in Figure 13.5.

Figure 13.5 Privileges Dialog in GemStone User Window



The specific list of privileges and the methods that require these privileges vary between different server products and versions.

For more information on privileges, see the chapter on "User Accounts and Security" in the *System Administration Guide for GemStone/S 64 Bit*. The method comments in the image provide privilege requirements for executing specific methods.

Chapter

A

Packaging Runtime Applications

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

A.1 Prerequisites

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

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

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

Names

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

Replicating Blocks

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

Defunct Stubs and Forwarders

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

Shared Libraries

A deployed runtime application that uses GemBuilder needs to contain the shared libraries from the GemStone/S 64 Bit `/bin` directory.

A.2 Packaging

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

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

Step 3. Load your application code.

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

Chapter

B Client Smalltalk and GemStone Smalltalk

This appendix outlines the few general and syntactical differences between the IBM VisualAge Smalltalk and GemStone Smalltalk languages.

B.1 Language Differences

GemStone's Smalltalk language is very similar to client Smalltalk in both its organization and its syntax. GemStone Smalltalk extends the Smalltalk language with classes and primitives to add multiuser features such as transaction support and persistence. The GemStone class hierarchy is extensible, and new classes can be added as required to model an application. The GemStone class hierarchy is described in the *GemStone/S 64 Bit Programming Guide*.

A quick look at the GemStone class hierarchy shows that it differs from the client Smalltalk class hierarchy in that classes for file access, communication, screen manipulation, and the client Smalltalk programming environment don't exist, and in that the GemStone Smalltalk hierarchy contains classes for transaction control, accounting, ownership, authorization, replication, user profiles, and index control.

GemStone Smalltalk also introduces optimized selection blocks.

As a Smalltalk programmer, you will feel quite at home with GemStone Smalltalk, but you should take note of the differences outlined in this appendix.

Selection Blocks

Selection blocks in GemStone Smalltalk and the use of dots for path notation have no counterparts in client Smalltalk.

```
myEmployees select: {:i | i.is.permanent}
```

Array Constructors

Array constructors do not exist in client Smalltalk. In GemStone, array 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 }
```

Block Temporaries

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

One-way become:

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

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

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

Exception-handling

In client Smalltalk, exception-handling is implemented with two classes: Signal and Exception. In GemStone it is implemented with a single class: Exception. An Exception in GemStone is an object that represents state to be invoked in the event of an exception.

There are two types of exceptions in GemStone. In order of precedence, they are: 1) context exceptions, and 2) static exceptions. Static exceptions remain from run to run. Context exceptions are active as long as the context to which the exception belongs is on your call stack when an exception is signaled.

Client Smalltalk exception handling is analogous to GemStone context exceptions.

All nonfatal errors can be trapped by a GemStone application.

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

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.

GemStone server 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 DateTimes, the GbxTimeZone is used to determine the client object's correct current local time. The GbxTimeZone is defined at the time the gem logs into the GemStone server.

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 time zone to be different between the gem and client

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W