GemStone®

GemBuilder® for Java™ Programming Guide

Version 3.0

September 2010

GEMSTONE S™

INTELLECTUAL PROPERTY OWNERSHIP

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

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from GemStone Systems, Inc.

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

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

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

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

COPYRIGHTS

PATENTS

GemStone is covered by U.S. Patent Number 6,256,637 “Transactional virtual machine architecture”, Patent Number 6,360,219 “Object queues with concurrent updating”, and Patent Number 6,567,905 “Generational Garbage Collector”. GemStone may also be covered by one or more pending United States patent applications.

TRADEMARKS

GemStone, **GemBuilder**, **GemConnect**, and the GemStone logos are trademarks or registered trademarks of GemStone Systems, Inc. in the United States and other countries.

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

Microsoft, **MS**, **Windows**, **Windows 2000**, **Windows XP, Windows 2003, and Windows Vista** are registered trademarks of Microsoft Corporation in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemStone Systems, Inc.
1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006

Preface

About This Manual

This manual describes GemBuilder® for Java™ 3.0, an application programming interface (API) for developing distributed Gemstone® applications with a Java client and a GemStone server.

GemBuilder for Java pairs the portability and flexibility of Java with the scalability of the GemStone server. You can develop business applications around Java clients that take advantage of the server's multi-user object execution engine, transaction management facilities, and fault tolerant environment. The Java clients can be distributed as standalone applications or as applets that provide a universal client from a single, easily maintained source.

GemBuilder for Java consists of two parts: the programming interface between your Java client and the GemStone server, and Java-based tools for developing GemStone Smalltalk code and inspecting objects in the server.

The programming interface lets your client do the following:

- locate GemStone server objects and obtain stub references to them;
- send messages to GemStone server objects through stub references;

- send messages to Java objects from GemStone server objects that implement an adapter interface;
- execute ad-hoc Smalltalk code on the server; and
- handle exceptions signaled by server objects in a natural fashion.

The GemBuilder for Java Tools let you work with GemStone Smalltalk in the server. These tools, which can be incorporated into a Java development environment or used independently, are described in separate documentation, *GemBuilder for Java Tools Guide*.

GemBuilder for Java may be used against both 32-bit GemStone/S and GemStone/S 64 Bit. While the server products are similar, some features differ. GemBuilder for Java behavior that depends on server behavior may therefore vary depending on which server product and version you are running with. In this documentation, the term "GemStone/S" or "GemStone" is used when behavior is common; server specific differences are noted when necessary.

Assumptions

To make use of the information in this documentation, you need to be familiar with the GemStone server and with GemStone's Smalltalk programming language as described in the *Programming Guide* for GemStone/S or GemStone/S 64 Bit. These books explain the basic concepts behind the language and describes the most important GemStone kernel classes. The documentation assumes you have an existing GemStone application for which you want to create Java clients.

In addition, you should be familiar with the Java language, the Java API, and the development environment as described in your vendor's documentation.

Finally, this documentation assumes that the GemStone system has been correctly installed on your host computer as described in the *System Administration Guide* for your GemStone server and that GemBuilder for Java has been installed and configured according to the *GemBuilder for Java Installation Guide*.

How This Manual is Organized

Basic Concepts describes the overall design of a GemBuilder application and presents the fundamental concepts required to understand the interface between a Java client and the GemStone server.

Communicating With the Server explains how to communicate with the GemStone server by initiating and managing GemStone sessions.

Interacting with Server Objects describes how to locate objects in the server and obtain stubs referencing them, how to send messages to the objects or execute ad-hoc Smalltalk code, how to handle exceptions raised on the server, and how objects are marshaled between the Java client and the server.

Forwarding Server Messages to Client Objects explains how GemStone objects can send messages to Java objects by using an adapter to compile the message into Java code.

Managing Server 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.

Observing Session and Server Events explains how your application can monitor events in client sessions and certain events in the server.

Deploying Your Application explains the steps you need to take to deploy your Java clients for use with the GemStone server.

Other Useful Documents

- The *Javadocs* that are provided with the GemBuilder for Java product distribution are a key source of up-to-date information on GemBuilder for Java functionality.
- *GemBuilder for Java Tools Guide* describes the independent set of tools that let you explore and modify Smalltalk code in the server.
- The *Programming Guide* for GemStone/S and for GemStone/S 64 Bit describe the GemStone server and the GemStone Smalltalk language.
- If you will be acting as a system administrator, or developing software for someone else who must play this role, you should read the *System Administration Guide* for GemStone/S or for GemStone/S 64 Bit.

Technical Support

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

GemStone Web Site: http://support.gemstone.com

Use of this site requires an account, but registration is free of charge and provides immediate access.

All GemStone product documentation is provided in PDF form on this website. Documentation is also available at

http://www.gemstone.com/documentation

In addition to documentation, the support.gemstone.com website provides:

- Bugnotes, identifying performance issues or error conditions that you may encounter when using a GemStone product.
- TechTips, providing information and instructions that are not otherwise included in the documentation.
- Compatibility matrices, listing supported platforms for GemStone server 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.

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

Website: http://techsupport.gemstone.com

Email: support@gemstone.com

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

Your GemStone support agreement may identify specific designated contacts who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals.

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. 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.

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

24x7 Emergency Technical Support

GemStone offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, for issues impacting a production system. For more details, contact your GemStone account manager.

Training and Consulting

Consulting and training for all GemStone products are available through GemStone's Professional Services organization. GemStone periodically offers training courses at our offices in Beaverton, Oregon, or training can be arranged at your location. Contact your GemStone account representative for more details or to obtain consulting services.

Contents

Chapter

1 Basic Concepts

The GemStone Solution

This overview describes GemStone's solution for Internet and corporate intranet applications: the GemStone Server and GemBuilder for Java (GBJ).

About the GemStone Server

The GemStone server provides a wide range of services to help you build object-based information systems. GemStone:

- supports transaction-intensive, business-critical applications involving more than 1000 concurrent users and persistent object spaces in the tens of gigabytes. GemStone/S 64 Bit provides greater scalability for much larger applications requiring more object space.
- provides a distributed server architecture that allows the server and processes to be spread over multiple hardware platforms and operating systems in a heterogeneous computing environment
- provides object persistence on an instance basis (determined by reachability from other persistent objects) for greater flexibility compared to object server systems that use class-based persistence or object-relational mapping

- provides transactions having ACID properties (atomicity, consistency, isolation, durability)
- provides configurable concurrency modes and protocols for requesting locks on individual client Smalltalk objects and collections of objects, allowing developers to exercise fine-grained control over concurrent access to objects
- provides automatic referential integrity and an on-line garbage collection process that runs in the background
- supports embedded queries, path queries, collection queries, non-locking queries, multi-user indexes and extensible indexes
- can monitor events or changes in state of objects and send signals to other applications or to users, eliminating the need for inefficient polling

About GemBuilder for Java

The GemBuilder for Java API is a Java runtime package that provides a message forwarding interface between a Java client and a GemStone server:

- Java clients can locate GemStone server objects by name and obtain stub references to them.
- Java clients can send messages to GemStone server objects through stub references.
- GemStone server objects can send messages to Java client objects that implement an adapter interface.

The API does not include user interface classes or application frameworks. The focus is on transparent messaging and simple replication.

The GemBuilder for Java Tools are implemented in Java, so you can create Java clients using your preferred Java development environment, then create server-side applications in GemStone Smalltalk without leaving the Java environment. The set includes the following tools:

- A GemStone Browser lets you view the available GemStone Smalltalk classes and methods, create new classes, and add or modify methods.
- An Inspector lets you view objects residing in the GemStone server. You can examine the state of individual objects, modify them if desired, browse collections, or look at the contents of dictionaries.
- A Workspace lets you execute arbitrary strings of GemStone Smalltalk code. Use this tool to locate server objects to be examined in an Inspector, create

sample data for testing applications, assign access control to objects or collections, and perform administration tasks in the GemStone server.

- A Debugger is available whenever execution of a GemStone method results in a run-time error.

Integrating Information across the Enterprise

Through GemStone's distributed architecture, Java applications have the use of objects residing anywhere in the enterprise. And through GemStone's connectivity tools, Java applications can also have access to business data from relational and legacy databases:

- By using GemConnect, GemBuilder for Java clients have access to Oracle RDBMSs.
- GemStone Object Transaction Services provide complete control over transactions that include data from heterogeneous sources.

GemStone Sessions

All interaction with the GemStone server takes place in the context of a *session*, which is a login to the GemStone server under a particular GemStone User ID. Because this context remains in effect until the session is closed, it does not have to be reestablished for each individual service request.

Each session is associated with its own *Gem* process, which acts as the GemStone server for that session. It is the Gem process that accesses the shared objects and executes GemStone Smalltalk code.

Development Strategy

GemBuilder for Java supports distributed application development with the GemStone server. GemStone is used as an object management system in which Java clients can take advantage of GemStone Smalltalk method execution in the server. GBJ provides limited replication of simple data types to the client.

Application partitioning must be considered in the development of GBJ applications from the start because the client component is written in a different language than the server component.

Using GemBuilder for Java with Your Development Environment

GemBuilder for Java works in conjunction with the Java development environment (JDE) of your choice. After installing GBJ on your development platform, do the following:

- Add `gbj30.jar` and `gbjopensrc30.jar` to your CLASSPATH.
- Add the GemBuilder for Java and GemStone shared library locations to the appropriate paths. Consult the *GemBuilder for Java Installation Guide* for details.
- Add an item to your JDE tools menu to launch the GBJ Tools. For instance:

  ```
  java com.gemstone.tools.GbjLauncher
  ```

- Add "`com.gemstone.gbj.*`" to the import list for each Java class where it is appropriate.
- Optionally, add an item to the JDE tools menu to open a browser on the online documentation home page, `index.html`. Or, add a bookmark in your Web browser.

Development Steps

The typical steps in developing an application using GemBuilder for Java are:

1. Model the system using accepted OO modeling techniques.
2. Identify objects that will provide server-based services.
3. Use the GBJ Tools to create server classes and methods.
4. Write the Java client classes using tools of your choice.
5. Test the application.

A Rapid Application Development approach could also be taken where classes and methods are built in both the client and the server during the building and testing of the application. In this approach, you would have the GemBuilder for Java Tools running along side the Java development environment. You would add or modify methods in the GemStone server, commit them to the server, and make the requisite iterative runs of the Java application within a Java development environment. Changes you make within the Tools session must be committed to the GemStone server before you test them in the Java application because the application being tested typically will run in a separate GemStone session.

Partitioning Your Application

The recommended approach to application partitioning is to keep user-interface work in the Java client, and to keep business objects — shared procedures, rules, and data — in the GemStone server. In general, the recommended approach is:

- Have the server handle object processing and queries. Avoid having the server side know about details of the user interface where that is not necessary. Instead, let the client request the objects the interface needs.
- Have the client make the minimum number of requests to the server by using the techniques described under “Accessing Complex Objects Efficiently” on page 37. Making requests efficiently is especially important for Internet applications where the network is quite slow.

The GemBuilder for Java Tools can also be used to test and debug GemStone server-side classes and methods independently of a Java development environment. The Workspace, Inspector, and Debugger tools are useful in performing unit tests on server classes and in examining the state of the GemStone server to find and correct bugs.

GbjObject, which extends java.lang.Object, is GemBuilder for Java's principal class. Instances are *stubs* that represent objects in the GemStone server.

Although the stub and the holder classes present one recommended approach to implementing the client, neither is inherently necessary; satisfactory results could have been achieved in other ways.

Chapter 2

Communicating With the Server

Overview

All interaction with the GemStone server takes place through a session that is dedicated to serve your Java client. Each session is an instance of GbjSession, which provides the environment and a number of important capabilities for interaction with the GemStone server. Through GbjSession, your Java client can:

- connect to and disconnect from the server
- locate server objects by name and obtain stub instances representing them
- install custom stub objects for specific kinds of server objects
- execute ad-hoc GemStone Smalltalk code in the server
- control transactions in the server
- install adapters that dispatch messages from server objects to client objects
- observe events in the server, such as a change to a specific object or a Gem-to-Gem (session to session) signal, and certain events that happen in another session object, such as the committing of a transaction.

Each session has a public variable, userData, in which you store session-specific information, such as context information that you want to be available elsewhere in the client. Because this variable is of type Object, it must be cast to the appropriate class when retrieving objects stored in it.

The session communicates with the GemStone server through a Gem process, which is created during login to the server.

The GbjSession not only provides a connection with the server, it also provides a context, such as a name space, object access authorizations, and privileges. Once the connection is established, your Java application can access GemStone server objects in any way permitted by the security privileges associated with that GemStone userId

Opening a Session

One of the first tasks your client needs to perform is to open a session with (log in to) the appropriate GemStone server. You do that in two steps:

1. Create and fill in an instance of GbjParameters.
2. Use the parameters to create an instance of GbjSession, then connect the session to the GemStone server.

The GbjSession not only provides a connection with the server, it also provides a context, such as a name space, object access authorizations, and privileges. Once the connection is established, your Java application can access GemStone server objects in any way permitted by the security privileges associated with that GemStone userId.

Creating the Session Parameters

An instance of GbjParameters defines the GemStone server, userId, and other information needed to log in to GemStone. The following code sets up parameters for a session when the user clicks the application's Connect button.

Example 2.1 Setting the Session Parameters

```
import com.gemstone.gbj.*;
GbjParameters params;
params = new GbjParameters();

// required parameters without defaults
params.userName = “DataCurator”;
params.password = “swordfish”;
params.serverName = “gs64stone”;

// parameters that have defaults as shown
params.gemnetName = “gemnetobject”;
params.transactionMode = GbjParameters.ManualTransactions;
```

The **userName** and **password** fields must match an existing GemStone userId in the server. There are no defaults for these fields. You must specify the stone to which you will connect to in **serverName.**

The **gemnetName** field determines which Gem service start up script is read when the session starts. For most GemStone installations, the name is `gemnetobject` (the default, for Bash, Bourne or Korn login shells) or `gemnetobjcsh` (for users of the C shell on GemStone/S only).

The **transactionMode** statement sets the way in which transactions are started in a GemStone session. The default mode under GemBuilder for Java is ManualTransactions, in which transactions must be started explicitly. AutomaticTransactions specifies a chained mode in which a transaction is started when the session connects to the server and new transactions begin after a commitTransaction or abortTransaction is processed. For further information, see “Transaction Modes” on page 62.

An RPC login may also require a host operating system user name and password as parameters (**serverOSUserName** and **serverOSPassword**), depending on the mode in which the NetLDI is running. For further information, see “Effect of NetLDI Mode” on page 22.

NOTE:
Ordinarily, a GemStone NetLDI network server must be running on the Stone's machine in order to spawn the gem process.

Effect of NetLDI Mode

The mode in which the NetLDI is running affects the ownership of the Gems that are spawned and the network authentication requirements. In general, the effect is the same as for other parts of a GemStone installation:

- The default mode requires authentication by way of a host user name and password. This information can be provided as session parameters or by a `.netrc` file. The Gem is owned by the specified host user.
- Guest mode makes it unnecessary to provide authentication. Under UNIX, this mode typically is combined with captive account mode, which causes the captive account to own the Gem processes. Under Windows NT, guest mode is the recommended mode; it reduces security but provides increased convenience.

For further information about the NetLDI, refer to the *System Administration Guide* for your GemStone server.

Creating the Session and Connecting to the GemStone Server

After you have created and filled in an instance of GbjParameters, the next step is to create the session instance and connect it to the server. This example is a continuation of the previous one.

Example 2.2 Connecting to GemStone

```
GbjSession mySession;
mySession = new GbjSession(params);
mySession.connect();
```

Launching Tools From Your Application

You can start the GemBuilder for Java Tools from your application to facilitate debugging from the GemStone session in which the application is running. (The class definition must include `com.gemstone.tools.*` in place of, or in addition to `com.gemstone.gbj.*`.)

To open a Tools launcher for a new session, embed this code in your application:

```
new GbjLauncher();
```

Alternatively, provide an active, logged in session as an argument to the constructor:

```
new GbjLauncher(aGbjSession);
```

Your application can use a session's launcher text pane as a transcript by sending the message transcript() to a launcher instance. Because the transcript is a Java TextArea, you can use appropriate protocol, such as appendText(). For instance,

```
this.launcher.transcript().appendText("Done");
```

Closing a Session

When the time comes to terminate the session, send the message close() to the session object, and perform any other housekeeping necessary.

WARNING:

Closing the session logs out the user's GemStone session. Any uncommitted changes are lost.

Chapter

3 Interacting with Server Objects

Overview

GemBuilder for Java (GBJ) provides a message-forwarding interface through which your Java client can interact with objects in the GemStone server. Your client code explicitly obtains *stubs,* which are client Java objects (instances of GbjObject or its subclasses) that represent objects in the GemStone server. A stub knows which GemStone server object it represents, and it responds to all messages in its protocol by passing them to the appropriate GemStone server object.

TERMINOLOGY NOTE:

Users of GemBuilder for Smalltalk will notice the term stub *as used here differs from that to which they are accustomed and corresponds more closely to what they would call a* forwarder. *This difference in terminology is unavoidable because it is anchored in the Java literature.*

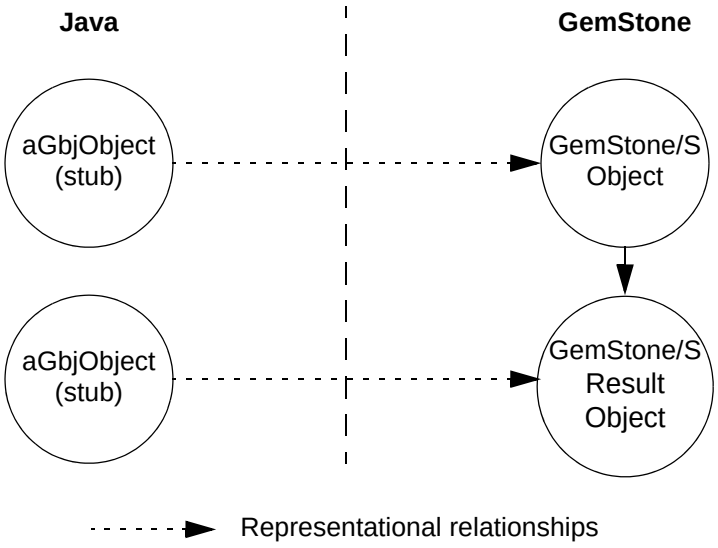
Because almost any message sent to a GemStone server object is capable of raising an exception in the server, a portion of your Java code will be devoted to catching and handling these exceptions when they are returned to the client. Ordinary Java techniques are appropriate, and there is a GBJ-specific subclass, GbjException.

The Message-forwarding Interface

Once a connection to the server has been established, your client can use the session to interact with persistent objects. The process typically involves two fundamental steps:

1. locating the object by name, which returns a *stub* representing the server object, and
2. sending a message through the stub and receiving a reply in the form of another stub representing the result returned in the server. In the case of simple data types, the stub acts as a Java wrapper for a cached value (see "Accessing a Stub's Cached Value" on page 36).

Figure 3.1 Stubs Representing GemStone Server Objects



Each stub is an instance of GbjObject or one of its subclasses. GbjObject provides protocol for working with the stub, including extracting the actual Java data from the GbjObject.

Method	Description
getOop()	return the object identifier (OID) of the server object it represents.
getSession()	return the GbjSession instance this stub uses to communicate with the server
getValType()	return an integer code representing the type of object as represented on the GS server
getObjType()	return an integer code representing the type of java object cached on the client as representative of the GS server object

t.

Integer code	Object type	Method
-1	Unknown	
0	Null	
1	Boolean	booleanValue()
2	Character	charValue(), byteValue()
3	Long	shortValue(), longValue()
4	Double	floatValue(), doubleValue()
5	Bytes	bytesValue()
6	OOPs	getObjs()
7	String	stringValue()
8	Double Byte String	stringValue()
9	Calendar	calendarValue()

Using Stub Protocol to Send Messages

GbjObject implements methods corresponding to the fundamental ones in the server's class Object. As a result, these are inherited by your stubs without additional effort. Here are a few examples of commonly used methods (for a complete list, see the class description):

at()	remoteEqualsIdentical()
atPut()	remoteSize()
equals()	segment()
in()	sendMsg()
isKindOf()	toString()
notNil()	

The class GbjCollection, a subclass of GbjObject, defines additional fundamental methods common to server Collection classes. For information, see “Working with Collections” on page 40.

NOTE:
A Java stub must explicitly implement all of the methods it wishes to forward. Java does not have a mechanism similar to Smalltalk’s doesNotUnderstand:.

Sending Dynamic Messages

The method `sendMsg()` in GbjObject lets you send GemStone Smalltalk messages to server objects without implementing corresponding protocol in Java.

The following example uses the server message firstName in a selection block to retrieve the first customer found with that name, then uses `sendMsg()` to send the message lastName to that instance. The method `detect()` is defined in GbjCollection. The object returned by `sendMsg()` is a GbjObject, and since the name is a simple data type, its value can be obtained from the stub's cachedValue member by using `stringValue()`.

Example 3.1 Sending a Message to a Server Object

```
GbjCollection myCollStub;
GbjObject aCust;

myCollStub = (GbjCollection) mySession.doit("HR_Customer allCustomers");
aCust = myCollStub.detect("[:cust | cust firstName = 'Fred']");
System.out.println("Last Name is " +
    aCust.sendMsg("lastName").stringValue());
```

Class GbjObject implements sendMsg() several ways with different method signatures. Here is a partial list:

Method Signature	Use
sendMsg(String)	unary message
sendMsg(String, Object)	one-keyword message or binary message
sendMsg(String, Object, String, Object) ...	two-keyword message

and so forth up to a message with five keywords. For instance:

```
GbjObject obj = mySession.doit("Array new: 10");
obj.sendMsg("size");
obj.sendMsg("at:", new Integer(1));
obj.sendMsg("at:", new Integer(1), "put:", Boolean.TRUE);
```

Because the last two examples take Java String / Object pairs, the int argument must be wrapped in an Integer object. GemBuilder for Java automatically converts this Java Integer to a GemStone Integer (SmallInteger or LargeInteger) as part of the marshaling process.

The GbjObject method `perform()` occasionally is a useful alternative to `sendMsg()` because it takes a user-specified number of arguments and the arguments can be treated as a unit.

This example builds an Array of arguments needed to create a new engineer instance, then uses the Array as an argument to `perform()`, which creates the new instance in the server. Finally, `sendMsg()` adds the instance to the existing Collection. The client class Engineer was previously registered as a stub for the corresponding server class.

Example 3.2 Using GbjObject.perform With an Argument List

```
GbjObject engClass = null;
Object args[] = {eml, firstNm, lastNm, ph};

// get stub for the class
engClass = sess.objectNamed("HR_Engineer");
HR_Engineer eng = null;
try {
     eng = (HR_Engineer) engClass.perform(
          "email:firstName:lastName:phone:", args, 4);
} catch (GbjEventException e) {
     new gemstone.tools.GbjDebugger(e);
}
engClass.sendMsg("addEngineer:", eng);
```

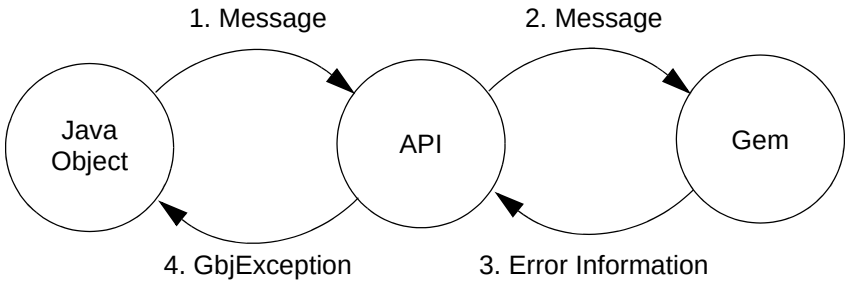
Handling Server Exceptions

When interaction with the GemStone server causes an error to be raised, notification is returned to the client in the form of an instance of GbjException, which extends java.lang.RuntimeException. The compiler ignores GbjException when enforcing catch-throw semantics. Therefore, a method that calls another method that throws a GbjException is not forced either to catch the exception, or to declare in its header that it throws the exception.

Here is a typical scenario: (1) the client object sends a message to a stub, which (2) the stub forwards to the server. If this message results in an error in the server, (3)

the GemStone error information is sent back to the GemBuilder for Java, which (4) throws it as a GbjException.

Figure 3.2 Throwing a GbjException



Typical code to handle the GbjException is similar to that for any other Java exception:

Example 3.3 Handling a GbjException

```
public String firstName() {
String nm = null;
try {
    nm = this.sendMsg("firstName").stringValue();
} catch (GbjEventException e) {
    System.err.println("GemStone event exception occurred"
        + e.getMessage());
} catch (GbjException e) {
    System.err.println(
"HR_Engineer>firstName() GemStone exception: "
+ e.getMessage());
}
return nm;
}
```

NOTE:

Most methods that access the server can cause an error to be raised in the server. Because try{}catch{} statements should be part of your GemBuilder for Java coding practice, they appear frequently in the examples in this documentation.

You can determine the nature of the exception by examining its category and number members. The class GbjGemStoneErrors provides variables for all of the GemStone kernel class error numbers (that is, for those associated with GbjExceptions that have category GemStoneError). The instance variable GbjException.kernel also holds this list. For instance, to find out whether a lock you obtained on an object in GemStone is dirty (that is, whether the object has changed since you started the current transaction), you could use this test:

Example 3.4 Catching a Specific GemStone Error

```
try {
    // obtain a lock in the server
}
catch (GbjException e) {
    if (e.number == GbjException.kernel.LOCK_ERR_OBJ_HAS_CHANGED) {
        // handle dirty lock
    }
    // else throw e
}
```

However, developers can catch more specific error conditions in GemStone using any of the following subclasses of GbjException. These subclasses correspond to the exception categories in GbjGemStoneErrors.java:

- GbjCompilerException
- GbjRuntimeException
- GbjAbortingException
- GbjFatalException
- GbjInternalException
- GbjEventException
- GbjNonKernelException

To Invoke a Debugger on an Exception

1. Have the class import `com.gemstone.tools.*`.
2. To open the debugger on a particular exception, use the GbjDebugger constructor with the exception as an argument. For instance:

```
catch (GbjException e) {
new com.gemstone.tools.GbjDebugger(e);
}
```

Representing Server Objects in the Client

Deciding Which Objects to Represent

GemBuilder for Java uses stubs (instances of GbjObject and its subclasses) to represent objects stored in GemStone to your Java client.

Because of the hierarchical structure of complex objects, you should begin by identifying the subsystems in your application that define persistent objects, and then identify a *root* object in each subsystem. The root objects of an application are the persistent objects from which other persistent objects can be reached by transitive closure; that is, either by direct reference or indirectly through any number of layers of references.

Each root object in the GemStone server in effect represents all other objects to which that object refers, such as its instance variables. And because those instance variables are represented, their instance variables are also represented, and so on, until you reach atomic objects that refer to no others, such as characters, integers, strings, booleans, or nil. The entire network of related objects forms a tree structure whose leaves are the final objects reached.

Obtaining a stub for the root object makes the entire subsystem in the server accessible to the client, since you can easily get stubs for other elements once you have a stub for the root. The most common kinds of root objects in the server are:

- global variables
- class variables
- class instance variables.

Obtaining GbjObject Stubs

There are three basic ways to obtain stubs:

- by looking up a named object in the server
- by sending a message to a server object through its stub and saving the stub that represents the object returned in the server
- by registering a Java class as a stub class for a corresponding server class

For related information, see “Accessing a Stub’s Cached Value” on page 36.

Looking Up a Named Object in the Server

Performing an explicit name lookup in the server returns an instance of GbjObject representing the server object. This lookup is performed in the context of the current session logged in to GemStone; that is, it uses that session’s symbol list for name resolution. For instance, where AllEngineers has been defined previously as a global variable in the server:

Example 3.5 Resolving a Named Object

```
GbjObject myStub;
myStub = mySession.objectNamed("AllEngineers");
// use the stub as desired
System.out.println("Size: " +myStub.sendMsg("size").intValue());
```

Saving a Returned Stub

Sending a message to the server returns an instance of GbjObject (that is, a stub) to which you can send messages if appropriate. For instance, this code obtains a stub by using server class protocol to access AllEngineers; the stub is cast to type GbjCollection to make available the additional protocol defined for that class:

Example 3.6 Using the Result of GbjSession.doit

```
GbjCollection myCollStub;
myCollStub = (GbjCollection) mySession.doit( "HR_Engineer allEngineers");
// use the stub as desired
System.out.println("Size: " +myCollStub.sendMsg("size").intValue());
```

The result that is represented need not be a persistent object (one which is part of the committed repository).

Registering a Custom Stub

Registering a custom stub creates a correspondence between specific Java and GemStone *classes*, which can provide for more natural coding within your application. The stub class must be a subclass of GbjObject or GbjCollection. Message sends to the server that correspond to the designated GemStone class return a stub that is instantiated in Java as an instance of the stub class.

Registration can be performed during static initialization or on an session-specific basis at runtime. For instance, to register the Java class Engineer (a subclass of GbjObject) as a stub for the server class of the same name:

```
mySession.registerStaticStub(new Engineer(), "Engineer");
```

At runtime, you could use inherited GbjObject protocol to send messages to an instance of the class. For instance, this expression in class Engineer makes the engineer available to accept an assignment:

```
this.sendMsg("available: true");
```

The method `execute()` is inherited from GbjObject, and available: is an instance method defined by the server class.

By implementing the method `available()` in the Java class itself, the expression could become more natural:

```
this.available(true);
```

Implement `available()` as shown below:

Example 3.7 Adding Server Protocol to the Java Side

```
public void available(boolean avail) {
    if (avail)
        this.sendMsg("available: true");
    else
        this.sendMsg("available: false");
}
```

For more information about registering custom stubs, see “Obtaining Application-specific Stubs” on page 44.

Accessing a Stub's Cached Value

When a result of a request is retrieved from GemStone, and that GemStone server object is a kind of one of the classes listed under "Representing Server Objects in the Client" on page 33, the resulting Java object holds the pre-fetched value of the GemStone server object it represents. This action minimizes the number of round trips to the server needed to get a usable form of the result.

GbjObject provides methods for getting at this cached value. The value itself is stored in the GbjObject variable *cachedValue*, which is public and may be accessed directly. GbjObject also supplies the following methods to retrieve a cachedValue in an appropriate way:

`booleanValue()`	`floatValue()`
`charValue()`	`intValue()`
`dateValue()`	`longValue()`
`dbStringValue()`	`stringValue()`
`doubleValue()`	`stringBufferValue()` (cachedValue holds a String that is converted to a StringBuffer)

Note that if the cachedValue member is directly accessed and holds an integer, an Integer wrapper is what will actually be found. This is also true of the other non-object values that are held in cachedValue. The cachedValue of a stub for a GemStone DoubleByteString holds a `com.gemstone.gbj.DoubleByteString` wrapper. Clients must use `dbStringValue()` to access the String object held in a stub for a GemStone DoubleByteString.

Executing Ad-hoc Smalltalk Code

GemBuilder for Java provides two ways to execute *ad-hoc* code in the server, code that is not an existing compiled method.

The most general approach is to send `doit()` to a session object, which causes the code to be executed in that session's environment on the server. This example creates a key and value in the SymbolDictionary UserGlobals. The expression is sent to the session object for evaluation. The expression may contain temporaries and ^ (return) statements, but it may not contain refer to *self* or *super*.

Example 3.8 Executing Ad-hoc Code in the Server

```
mySession.doit(
    "UserGlobals at: #AllEngineers put: (HR_Engineer allEngineers)"
    );
```

Alternatively, the method `GbjObject.execute()` permits you to send an ad hoc message to an object. The receiver, the server object represented by the instance of GbjObject to which the message is addressed, is the execution context. The message may use *self* to refer to the receiver and may directly reference instance variables of the receiver.

Accessing Complex Objects Efficiently

While a stub provides access to all server objects reachable by a transitive closure on that object, it can be inefficient to access a number of instance variables individually. You should consider these approaches, either individually or in combination:

- Transfer all named instance variables in one request.
- Flatten the instance variables into an Array of fundamental Java data types, which are automatically stored in the stub's cachedValue elements.
- Create a *holder* object as a proxy for the server object. Have the holder store explicit replicates of a few frequently needed instance variables as simple data types, and include the stub itself so you can access instance variables when necessary.

The above techniques also provide the building blocks for handling collections efficiently. For further information, see “Working with Collections” on page 40.

Getting All Named Instance Variables

The method namedInstanceVariables() returns all of an object’s named instance variables in an Array of GbjObjects (stubs), including those instance variables inherited from superclasses. Instance variables that are simple data types in Java are stored in the stub's cachedValue element.

The instance variables are in the order determined by the GemStone message `Behavior >> allInstVarNames.`

This example first obtains a stub by sending the message myUserProfile to server class System, which is represented by a variable in GbjKernelObjects. Next, stubs for all of the UserProfile's named instance variables are obtained in a single request.

Example 3.9 Getting All Named Instance Variables

```
GbjObject uprofStub, uprofVars[];

uprofStub = mySession.kernel.System.sendMsg("myUserProfile");
uprofVars = uprofStub.namedInstanceVariables();
System.out.println("UserID: " +uprofVars[1].stringValue());
```

Flattening Objects in the Server

Many applications will want to pull information from the GemStone server to display in windows or otherwise use for user interfaces.

The recommended approach to accomplishing this task is to serialize simple data objects on the server side into sequenceable collections (such as Arrays or OrderedCollections) that can then be efficiently pulled into the client for processing.

This example gets information about one of the Customer objects, where *thisCustomer* is a previously obtained stub representing a particular customer. The method `allElements()` returns an array of GbjObjects holding the contents of the array constructed in the server.

Example 3.10 Flattening a Server Object

```
GbjObject fields[] = null;
fields = ((GbjCollection) thisCustomer.execute(
    "#[firstName, lastName, emailAddress]")).allElements();
String firstNameField = fields[0].stringValue();
String lastNameField =  fields[1].stringValue();
String emailField =  fields[2].stringValue();
System.out.println("Name: " +lastNameField +", " +firstNameField);
```

Replicating Objects Using Holders

An efficient way to handle complex server objects is to create a *holder* object in Java. Populate the holder with Java objects that replicate selected server instance variables in the form in which you need them. The holder can also store the stub itself for ease in sending additional messages to the server object. The concept explained here is particularly useful when it is extended to collections (see "Working with Collections" on page 40).

This example modifies the previous one on flattening objects by drawing on an additional class. MyHolder is a Java class with one instance variable that combines the separate first and last names in the server, and a *stub* instance variable. Where the previous example simply displayed the instance variables in a dialog, this example stores them in a holder. The stub can be used later for such tasks as retrieving the emailAddress instance variable by sending it the appropriate message.

Example 3.11 MyHolder.java

```
import gemstone.gbj.*;
public class MyHolder {
    public String name;
    public GbjObject stub;

// ...
}
```

Example 3.12 Replicating an Object in MyHolder

```
GbjObject fields[] = null;
MyHolder thisCustomerHolder = new MyHolder();
fields = ((GbjCollection) thisCustomerStub.execute(
    "#[firstName, lastName, emailAddress]")).allElements();
thisCustomerHolder.name = fields[0].stringValue() + ' ' +
    fields[1].stringValue();
thisCustomerHolder.stub = thisCustomerStub;
```

Working with Collections

Collections typically are the core of an application using the GemStone server, so it is important that Java clients deal with them efficiently. The class GbjCollection provides additional protocol for that purpose.

In general, most interaction with large collections should be by way of remote message sends, GemStone Smalltalk code that is executed in the server. Smaller collections, or selected portions of larger ones, may be explicitly replicated in the client using this two-phase approach:

1. Serialize the desired objects in the server so elements can be brought to the client efficiently. Typically, this step involves selecting and flattening the desired instances, then bringing them together in a new collection.
2. In the client, iterate over the resulting collection, placing a replicate of each server object into a Java holder.

The GbjCollection Protocol

The GbjCollection class implements most of the protocol that is common to all Collection classes in GemStone. Your client can invoke these methods directly from Java, letting GemBuilder for Java forward the appropriate Smalltalk message. Here are some representative methods; for a complete list, see the description of GbjCollection.

Protocol Category	Representative Methods
Adding	add() addAll()
Converting	asArray() asBag() asIdentitySet() asSortedCollection()
Enumerating	collect() detect() reject() select()
Removing	remove() removeAll() removeIdentical()

Searching	includes() occurrencesOf()
Sorting	sortAscending() sortDescending() sortWith()

Protocol Examples

The example below (shown in the example “Using GbjObject.perform With an Argument List” on page 30) uses GbjCollection protocol to add a new instance of Engineer.

Example 3.13 Using GbjCollection.add()

```
/* Create the object in the server. */
GbjObject engClass = mySession.objectNamed("HR_Engineer");
Object args[] = {"johnd", "John", "Doe", "(555) 123-4567"};
GbjObject newEngStub =
engClass.perform("email:firstName:lastName:phone:", args, 4);

/* Get stub for server collection and add object. */
GbjCollection allEngineers = (GbjCollection)
engClass.sendMsg("allEngineers");
allEngineers.add(newEngStub);
```

Some methods in GbjCollection require arguments that implicitly involve messages to server objects. Selection blocks are one example; for instance, class Engineer on the server implements the messages firstName and lastName. In the following code, the block argument to detect: sends the message "firstName" to each element of the collection until it obtains a match, then removes that element.

Example 3.14 Sending Method Arguments to the Server

```
GbjCollection allEng = (GbjCollection)
mySession.doit("HR_Engineer allEngineers");
GbjObject anEngr = allEng.detect("[:eng | eng firstName = 'John']");
allEng.remove(anEngr);
System.out.println("Removed John");
```

Serializing the Collection in the Server

The recommended approach to copying small collections to the client is to traverse them in the server, serializing the objects into a stream that GemBuilder for Java can transmit to the client in the minimum number of request round trips. For instance, this part could be encapsulated in a Smalltalk method in the server:

Example 3.15 Serializing a Collection

```
serializeCustomers
^allCustomers inject: Array new into: [:array :each |
    array add: each;
          add: (each firstName); add: (each lastName);...;
    yourself ]
```

The resulting Array contains the series *customer1*, *firstName1*, *lastName1*, ..., *customer2*, *firstName2*, *lastName2*, and so forth. Notice that the first array element for each customer is the customer object itself; in the client, this element will become a stub representing that customer. Apart from the element stubs, each field should be one of the simple data types that can be stored in a stub's cachedValue field.

Another approach (used in Customer) is to formulate a String to be sent as an ad-hoc message to the current session. In this example, the message causes the server to sort the customers and return two elements for each, the customer's name (by concatenating firstName and lastName) and the instance itself. Because most of the work is performed in the server, the method that sends the String is a doit().

Example 3.16 Serializing the AllCustomers Collection

```
static String execString =
"  | custs custsAndNames | " +
"  custs := HR_Customer allCustomers asSortedCollection: [:a :b | " +
"    (a lastName < b lastName) or: " +
"    [(a lastName = b lastName) and: [a firstName < b firstName]]] ." +
"  custsAndNames := OrderedCollection new." +
"  custs do: [:cust | " +
"     custsAndNames add: " +
"         (cust firstName + Character space + cust lastName)." +
"     custsAndNames add: cust. ]." +
"  custsAndNames ";

/**
 * Return a stub representing the result of performing the
```

```
 * above String in the server.
 */
public static GbjCollection allCustomers(GbjSession sess) {
    return (GbjCollection) sess.doit(execString);
}
```

Unpacking the Collection in the Client

Once the fields have been serialized in the server, GbjCollection provides two ways to access them from the client:

- The method `elements()` returns a Java Enumeration of GbjObjects (java.util.Enumeration).
- The method `allElements()` returns an array of GbjObjects.

These approaches differ primarily in the interface they provide; the underlying transport mechanism is much the same except that `elements()` caches a buffer of elements at a time, while `allElements()` retrieves all remaining elements in one trip.

To Enumerate the Collection

The method `elements()` in GbjCollection creates an Enumeration that performs efficient caching of the server object's contents for minimal client-server traffic during the enumeration operation.

In our example, the method `allCustomers()` in class CustomerHolder creates a holder object for each customer, unpacking into it the customer's name and the stub for the server element. Each holder becomes an element of a Vector that is returned. The call to Customer.allCustomers() performs the serialization on the server side, as shown previously. Customer has been registered as a stub class.

Example 3.17 Enumerating the Serialized AllCustomers

```
public static Vector allCustomers(GbjSession sess) {
    GbjCollection allCustsStub;
    HR_Customer custStub;
    Vector allCustomers = new Vector();
    String name;

    allCustsStub = HR_Customer.allCustomers(sess);
    for (Enumeration e = allCustsStub.elements(); e.hasMoreElements();) {
```

```
            name = ((GbjObject) e.nextElement()).stringValue();
            custStub = (HR_Customer) e.nextElement();
            allCustomers.addElement(new CustomerHolder(name, custStub));
        }
        return allCustomers;
    }
```

To Unpack the Collection from an Array

The method `allElements()` in GbjCollection returns an array of GbjObject stubs. The following would iterate over such an array of customer fields, creating a CustomerHolder for each pair of elements and placing the holders in a Vector. The result would be much the same as using the Enumeration interface. Again, the call to `Customer.allCustomers()` performs the serialization on the server side, as shown previously.

Example 3.18 Unpacking AllCustomers from an Array

```
public static Vector allCustomers(GbjSession sess) {
    GbjCollection allCustsStub;
    HR_Customer custStub;
    GbjObject fields[];
    Vector allCustomers = new Vector();
    allCustsStub = HR_Customer.allCustomers(sess);
    fields = allCustsStub.allElements();
    for (int i=0; i<fields.length; i+=2) {
        allCustomers.addElement( new CustomerHolder(
            fields[i].stringValue(), (HR_Customer) fields[i+1]));
    }
    return allCustomers;
}
```

Obtaining Application-specific Stubs

You can supplement the GbjObject and GbjCollection classes provided by creating your own Java classes and registering them as stubs representing specific GemStone classes. Two mechanisms are provided so you can register the stubs at initialization time (called *static stubs*) or at runtime (called *session-specific stubs*).

Because these mechanisms build a correspondence between a Java class and a GemStone class, they permit you to code your Java client in a more natural way.

Registering Stubs at Static Initialization

During static initialization, you can register a class correspondence by providing an instance of the client (stub) class and the name of the server class. All sessions will have access to these stubs. The client class must be a subclass of GbjObject as shown below.

Example 3.19 Registering a Static Stub

```
public class Engineer extends GbjObject {
    // other useful code
    static {
        //during static initialization, register class as a stub
        GbjSession.registerStaticStub(new Engineer(),
            "Engineer");
    }
}
```

When a session connects to the server, GemBuilder for Java creates a registry specific to that session to map instances of Engineer and its subclasses in the server to the client stub class, using a hierarchy returned by the GemBuilder for Java server component. The server class must be in the session's symbol list.

GemBuilder for Java considers both the server class's classHistory and its class hierarchy in deciding which stub class to use in representing the server object. For instance, setting the stub class for Engineer also sets the stub class for any version of class Engineer as well as for any subclasses of Engineer and their various versions.

Because the stub registry is based on the inheritance hierarchy returned at connect time, it is reinitialized if the session is disconnected and then reconnected. Clients that modify the class hierarchy at runtime do not see the stub mapping change until they disconnect and reconnect the session.

Registering Stubs at Runtime

You can register a stub at runtime, but the stub will exist only in sessions where it is registered explicitly. To ensure that the mappings are handled correctly, they should be registered just after connection to the server. The registry cannot be created or modified while the session is in an unconnected state.

Two method signatures are available:

- registerStub(GbjObject, String) is similar to registerStaticStub() in that it takes the name of the server class as String.
- registerStub(GbjObject, GbjObject) takes a stub reference to the server class, which allows the mapping to a server class that is not in the session's symbol list. For instance, the following obtains the class stub by sending the message `remoteClass()` to an instance of that class.

```
GbjObject objStub, classStub;
objStub = mySession.objectNamed("someObject");
classStub = objStub.remoteClass();
mySession.registerStub((new SomeClass(), classStub);
```

Putting Client Data into the Server

There are several ways by which your client can place data in the GemStone server, all involving explicit action:

- Sending a message to a stub is the typical way to add an object to an existing Collection or update an anonymous instance variable. GbjObject and GbjCollection provide a number of methods for this purpose, such as atPut() and add(). If the existing object is persistent (that is, part of the committed repository), the addition also becomes persistent when the transaction is committed.
- The method putInServer() in class GbjSession places its argument in the server and returns a GbjObject representing it. This action by itself does not make the object persistent. To make it persistent, you must name it or make it reachable from a named object.
- Ad-hoc GemStone Smalltalk code, such as that executed by GbjSession.doit(), returns an object in the server, which GemBuilder for Java represents by a GbjObject. Again, this action by itself does not make the object persistent. To make it persistent, you must name it or make it reachable from a named object.

GemBuilder for Java maintains a Saved Objects set in which it tracks non-persistent objects that have been exported by the server to the client. The purpose of this set is to keep the objects from being garbage collected during the lifetime of the session. (It is analogous to the Export Set of certain other GemBuilder products.). GbjSession provides methods for examining and modifying this set, and GbjObject provides convenience methods for adding or removing a particular object.

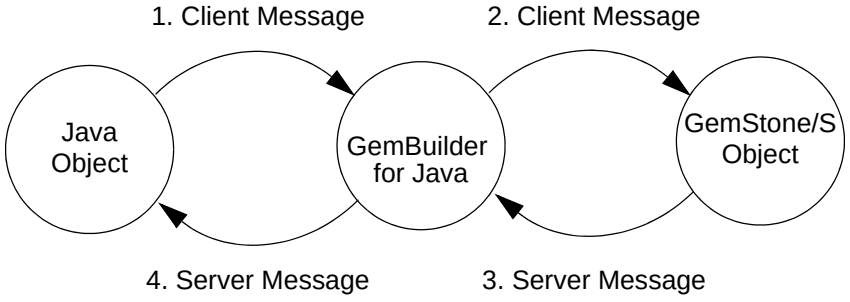
Chapter

4 Forwarding Server Messages to Client Objects

Overview

A typical server message scenario begins with a message from a client object to a server object, which GemBuilder for Java translates into a Smalltalk message. If the client message results in a server message back to the client, GemBuilder for Java (GBJ) must translate that Smalltalk message into an appropriate Java message and see that the message reaches the proper receiver.

Figure 4.1 Sending a Server Message to a Client Object



To do so, GemBuilder for Java uses the Java 1.3 Reflection API to find a matching Java method.

Alternatively, messages from server objects to client objects can be processed by a Java object that implements the GbjClientAdapter interface. The single method constituting this interface, dispatchGemStoneMessage(), translates the GemStone server message into a Java message and sends it.

Whichever mechanism you choose, GemBuilder for Java provides default mappings for certain basic methods:

GemStone Selector	Java Method
=	equals()
==	==
hashCode	hashCode()
getClass	getClass()
toString	toString()

If GBJ cannot find a method, it first checks to see if the selector matches one of these. It throws an exception only if the selector does not.

Using Reflection

Reflection is used to make Java method names from Smalltalk message selectors as follows:

- Smalltalk unary and binary message selectors are used without alteration as the Java method name.
- Smalltalk keyword selectors drop all but the first keyword; this, without the colon, is used as the Java method name.

With the resulting method name, GemBuilder for Java makes two tries at method lookup. The first uses object wrapper classes for unmarshaled arguments, such as Java class Double for an instance of GemStone Float. If this fails, the second try uses the Java primitive data type, such as double for an instance of GemStone Float. Arguments that cannot be converted to Java types are passed as instances of GbjObject.

To preserve the integrity of Java objects and to avoid illegal execution, GemBuilder for Java uses the public reflection interface. Therefore, method invocation is restricted to those methods published in the public Java reflection mechanism.

Implementing the Adapter Interface

If reflection is not appropriate for your application, two other mechanisms provide for messages coming from the server:

- The receiving object in the client can implement the GbjClientAdapter interface itself, thereby serving as its own adapter.
- The client can register another class as an adapter, in which case the adapter class must implement GbjClientAdapter and must know to how to forward an appropriate message to the receiving object.

The GbjClientAdapter interface requires one method to be implemented, dispatchGemStoneMessage().

The client adapter receives the following Java objects from GemBuilder for Java:

- the session (a GbjSession)
- the receiver (an Object)
- the method selector (a String)
- the arguments (a Vector)

Registering a Client Adapter

Your client can register an adapter during static initialization or at runtime:

- The static form establishes the adapter for all sessions.

```
Adapter engAdapter = new Adapter();
GbjmySession.registerStaticAdapter("Engineer",
  engAdapter);
```

- The runtime form is effective only for the particular session in which it is invoked.

```
Adapter engAdapter = new Adapter();
mySession.registerAdapter("Engineer", engAdapter);
```

Dealing with Multithreading

Each message received from the server for local processing causes GemBuilder for Java to create a separate thread in which to handle it. This action is necessary because the client thread that sent a message to the GemStone server is blocked waiting for a response.

CAUTION:
The creation of a separate thread to handle each server message makes applications that use client adapters inherently multithreaded. You must exercise appropriate caution to process the message in the correct context.

Because the GemStone server is not multithreaded itself, GemBuilder for Java limits access to the server from application threads so only one application thread is allowed into the communications layer at a time. Adapter threads are allowed to recursively send messages back to the server and are not blocked. Application threads are put to sleep while a message to the server from another application thread is being processed.

Message-sends in the Server

All server messages intended for a client object must be directed to a particular *client forwarder*, an instance of the private GemStone class GbjForwarder. In a typical scenario, the client forwarder is created from the client message that initiates the sequence.

For example, suppose much of the user interface knowledge resides in the server instead of being confined to the client. The client dialog class might implement something like the following to add an element to AllEngineers:

Example 4.1 Notifying Client from Server

```
String firstName, lastName, email, phone;
GbjCollection EngineersStub;
GbjObject result;
// code here to get data from text fields.

// now, add entry in server, close dialog if true returned:
try {
    result = EngineersStub.sendMsg("addEngrNotifying:", this,
        "firstName:", firstName, "lastName:", lastName, "email:",
        email);
    if (result.booleanValue()) {
        this.close();
    }
catch (GbjException e) {
    // handle exception
}
```

When the referent of “this” (the object to be notified) is unmarshaled in the server, the object will be represented by an instance of GbjForwarder because it is neither a stub object nor one of the simple data types for which value is cached. If the server code needs to return an error message to the client, it directs the message to this GbjForwarder. The server method might be implemented like this:

Example 4.2 Sending Notification from the Server

```
addEngrNotifying: dialog firstName: first lastName: last email: email
    | msg |
    msg := self validateFields: #[first, last, email].
    msg notNil ifTrue: [
        dialog displayError: msg.  "requires a client adapter"
        ^false ].
    [ System beginTransaction .
        "add engineer to AllEngineers class variable"
      System commitTransaction
     ] whileFalse .
    ^true
```

If the server detects invalid input, it forwards a message to the client.

If you're using reflection, you must write a method named displayError() that takes an argument of type String. GemBuilder for Java will find this method using reflection and invoke it. Here's how you can code displayError():

Example 4.3 Handling the Message Using Reflection

```
public void displayError(String msg) {
        System.err.println(msg);
}
```

If you have registered an adapter, the client object (or an instance of the registered adapter class) will receive dispatchGemStoneMessage() with the final arguments being the String "displayError:" and an array containing the contents of the server temporary variable "msg". Here's how you can code dispatchGemStoneMessage() to receive the message from the server:

Example 4.4 Handling the Message Using a Registered Adapter

```
public Object dispatchGemStoneMessage(GbjSession aSession, Object receiver,
String selector, Object args[]) {
    if (selector.equals("displayError:")) {
         this.displayError(args[0]);
    }
}
```

The GbjSession instance keeps an export set of objects represented by GbjForwarders to ensure they are not garbage-collected. GbjSession includes methods to examine and modify the export set; see, for instance, exportedReferences().

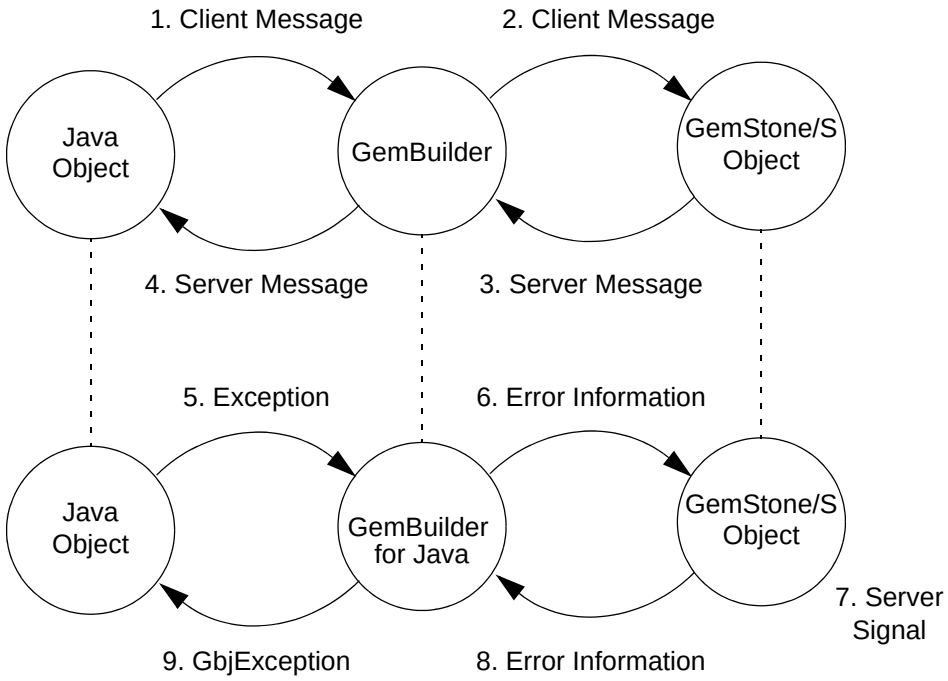
Exceptions Raised in the Client

Sending a message from the server to the client leads to the possibility of an exception being raised in the client. Such an exception will be returned to the server, where it may be transformed yet again into a Java GbjException that GemBuilder for Java throws to the client.

The following figure extends one used previously (in "Overview" on page 47). The sequence continues in the lower grouping, beginning with step 5 in which the Java object throws an exception in response to the server message. GemBuilder for Java catches this exception and (6) transmits it to the server. The server transforms the

exception information and (7) signals an error, #RT_ERR_CLIENT_FWD. The arguments are the detail text of the Java exception, and the Java stack. Unless the signal is handled in the server, the error information is sent back to the client (8), where GemBuilder for Java transforms it into a GbjException and (9) throws it to the application thread that started the sequence.

Figure 4.2 How a Client Exception Propagates to Calling Thread



You can install an exception handler in the server with code like this:

Example 4.5 Installing an Exception Handler

```
Exception
    category: GbjSignals
    number: clientForwarderError
    do: [ :ex :cat :num :args |
        ex remove.
        ^0 ].
```

Chapter

5 Managing Server Transactions

Overview

The GemStone server provides an environment in which many users can share the same persistent objects. The server maintains a central repository of shared objects. When a GemBuilder for Java (GBJ) application needs to view or modify shared objects, it logs in to the GemStone server, starting a session as described in "Opening a Session" on page 20.

A GemBuilder for Java 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 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 for Java 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 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 server. In a multiuser environment, concurrency conflicts will arise that can cause some commit attempts to fail. *Aborting* (rolling back) the transaction refreshes the session's view of the repository in preparation for further work.

In order to reduce operating overhead, a session by default runs outside a transaction, thereby temporarily relinquishing its ability to commit. A session running outside a transaction, called *manual transaction mode*, must explicitly begin a transaction before making changes that it will commit. Manual transaction mode is the default for GemBuilder for Java sessions, although the Development Tools override this default during login by placing the session in *automatic transaction mode*.

GemBuilder for Java 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 and higher concurrency, while *pessimistic concurrency control* uses locks of various kinds to improve a transaction's chances of successfully committing. GBJ 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 Programming Guide* for a full understanding of these transaction management concepts.

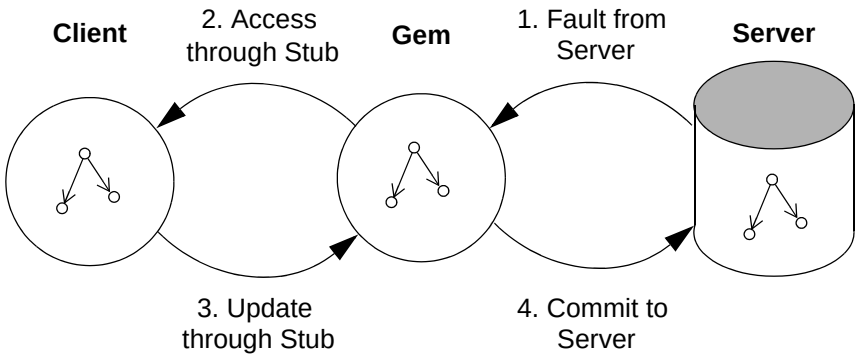
Separate topics explain each of the concepts mentioned here: transactions, committing and aborting, running outside a transaction, automatic and manual transaction modes, optimistic and pessimistic concurrency control, and reduced-conflict classes. Be sure to refer to the related topics in the *GemStone/S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

Operating Inside a Transaction

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

This figure shows a client and its repository, along with a common sequence of operations: (1) accessing an object from the shared repository, (2) creating an explicit replicate in the client, (3) modifying an object in the private view maintained in the Gem, and (4) committing the object's changes to the shared repository.

Figure 5.1 GemStone Application Workspace



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

For efficiency, GBJ does not duplicate the entire contents of the server into the Gem. GemBuilder for Java contains only those objects that have been accessed from the server or created by your client for sharing with the server. This action minimizes the amount of data that moves across the boundary from the server to the Gem.

CAUTION:

Because GemBuilder for Java does not update objects in the Java client at transaction boundaries, information copied into the client may no longer reflect the current state of the repository. Whenever you commit or abort a transaction, you should reinitialize copies of server objects to their current state in the Gem and shared repository. Alternatively, object change notification can be used for this purpose.

Committing a Transaction

When a client submits a transaction to the GemStone server for inclusion in the shared repository, it is said to commit the transaction. To commit a transaction, send the message:

```
aGbjSession.commitTransaction()
```

or, in the GemStone Browser, choose **Session > Commit**.

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

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

A commit request fails if the server detects a concurrency conflict with the work of other users. When the commit fails, the commitTransaction() method returns false.

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

Aborting a Transaction

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

```
aGbjSession.abortTransaction()
```

or, in the GemStone Browser, choose **Session > Abort**.

Aborting has these effects:

- The transaction (if any) ends. If the session's transaction mode is automatic, GemBuilder for Java starts a new transaction. If the session's transaction mode is manual (the default), 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. *You should reinitialize objects you have copied to the client.*

Handling Commit Failures

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

Following a commit failure, your session's private view of persistent objects may differ from its pre-commit state:

- The current transaction is still in effect. Nevertheless, you must end the transaction and start a new one before performing computations and 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. *You should reinitialize objects you have copied to the client.*

Following a commit failure, your session **must** refresh its private 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:

```
aGbjSession.doit("System transactionConflicts")
```

This expression returns a stub representing 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 65 for more discussion of the kinds of conflicts that can arise.)

Operating Outside a Transaction

Operating *inside* of a transaction involves a certain amount of overhead because GemBuilder for Java monitors the operations that occur, gathering all the necessary information required to prepare the transaction to be committed.

Operating *outside* of a transaction, however, saves some of the overhead of tracking changes, which may be significant in some applications. The session can view the repository, browse the objects it contains, and even make computations based upon their values, but it cannot commit any new or changed GemStone server objects. A session operating outside a transaction can, at any time, begin a transaction.

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.

These methods in GbjSession support running outside of a transaction:

Method	Description
beginTransaction()	Aborts and begins a transaction
isAutomaticTransactionModeSet()	Returns true if the server is in automatic transaction mode and false if not
setTransactionMode(“autoBegin”)	Sets the transaction mode to automatic (chained transactions, with an implicit begin after a commit or abort)
setTransactionMode(“manualBegin”)	Sets the transaction mode to manual (explicit *beginTransaction* required)
setTransactionMode(“transactionless”)	Sets the transaction mode to transactionless (aborting the current transaction, if any)

To begin a transaction, send the message:

```
aGbjSession.beginTransaction()
```

or, in the GemStone Browser, choose **Session > Begin**.

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

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

Being Signaled to Abort

When you are in a transaction, GemStone waits until you commit or abort to reclaim storage for objects that have been made obsolete by your changes. When you are running outside of a transaction, however, you are implicitly giving GemStone permission to send your Gem session a signal requesting that you abort your current view so that GemStone can reclaim storage when necessary. When this happens, you must respond within the time period specified in GemStone's STN_GEM_ABORT_TIMEOUT configuration parameter. If you do not, GemStone forces an abort and sends your session an abortErrLostOtRoot signal (GbjGemStoneErrors.ABORT_ERR_LOST_OT_ROOT), which means that your view of the repository was lost, and any objects your client had copied may no longer be valid. When your client receives abortErrLostOtRoot, the session has been aborted; your client must reinitialize all of its data from the Gem to reflect the current state of the GemStone repository.

You can detect an abortErrLostOtRoot signal and control what happens when you receive a signal to abort by implementing GbjObserver.update().

For example

Example 5.1

```
public void update(GbjObservable aSession, String aString, Object arg)
{
    if (aString.equals("event")) {
        GbjException e = (GbjException) arg;
        if (e.number == GbjException.kernel.RT_ERR_SIGNAL_ABORT) {
            aSession.abortTransaction();
            // refetch objects from Gem
            try {
                //re-enable generation of the error
                aSession.doit("System enableSignaledAbortError");
            }
            catch (GbjException ex) {
                System.out.println(ex.getMessage());
            }
        }
        if (e.number == GbjException.kernel.ABORT_ERR_LOST_OT_ROOT) {
            // refetch objects from Gem
        }
    } else {
        System.out.println(aString + " " + arg);
    }
}
```

This causes your GemBuilder for Java session to abort when it receives a signal to abort, and then to reinitialize copies of server objects in the client. If an abortErrLostOtRoot signal is received, the client detects it and reinitializes its copies.

Transaction Modes

A GemBuilder for Java session, by default, always is outside a transaction when it logs in. After logging in, the session can operate in either of three transaction modes: manual, automatic, or transactionless.

Manual Transaction Mode

In manual transaction mode, the session remains outside a transaction until you begin a transaction. This is the default mode in GemBuilder for Java. 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.

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.

Automatic Transaction Mode

In automatic transaction mode, committing or aborting a transaction automatically starts a new transaction. In this mode, the session operates within a transaction the entire time it is logged into GemStone. To run this way, a session must switch to automatic transaction mode or specify that mode in the login parameters.

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. For this reason, lengthy transactions can impede garbage collection of objects in the repository that are otherwise unneeded.

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, which is the default mode in GemBuilder for Java.

Transactionless Mode

In transactionless mode, the session remains outside a transaction. If all you need to do is browse the repository or make changes to objects in the client only, transactionless mode can be a more efficient use of system resources, because GemBuilder for Java does not need a commit page, nor will it elicit garbage collection.

Starting transactionless mode always causes an abort to refresh the session's private view of the repository.

Choosing Which Mode to Use

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.

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 transactionless mode if the work you are doing requires looking at objects in the repository only.

Switching Between Modes

To find out if you are currently in a transaction, execute

```
aGbjSession.inTransaction()
```

This returns true if you are in a transaction and false if you are not.

To change to automatic transaction mode, execute the expression:

```
aGbjSession.setTransactionMode("autoBegin")
```

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

To change to manual transaction mode, execute the expression:

```
aGbjSession.setTransactionMode("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.

To change to transactionless mode, execute the expression:

```
aGbjSession.setTransactionMode("transactionless")
```

This message automatically aborts the current transaction, if any, changes the mode to transactionless, and provides a fresh view of the repository.

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.

- 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.
- It checks to see whether other concurrent sessions have committed transactions of their own, modifying an object that you have read during your transaction, while at the same time you have modified an object that the other session has read.
- It checks for locks set by other sessions that indicate the intention to modify objects that you have read or to read objects you have modified in your view.

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

Read and Write Operations

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

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

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

The following conditions signal a possible concurrency conflict:

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

Optimistic and Pessimistic Concurrency Control

The GemStone / S server provides two approaches to managing concurrent transactions: optimistic and pessimistic, controlled via the GemStone / S server configuration parameter CONCURRENCY_MODE. GemStone / S 64 Bit always operates with a CONCURRENCY_MODE of FULL_CHECKS, which is pessimistic concurrent transaction management. The following discussion applies only to GemStone / S.

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

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

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

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

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

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

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

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

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

Setting the Concurrency Mode

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

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

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

Setting Locks

GemBuilder for Java provides locking protocol that allows application developers to write client Java code to lock objects.

A GbjObject or one of its subclasses 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:

```
aGbjObject.readLock()
aGbjObject.writeLock()
aGbjObject.exclusiveLock()
```

Note

GemStone/S 64 Bit does not support exclusiveLocks; this type of lock may only be used with GemStone/S servers.

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

When you request an exclusive lock, an exception will be thrown if another session has committed a change to aGbjObject since the beginning of the current transaction. In this case, the lock is granted despite the exception, 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 aGbjObject, then refetch its value through the stub. The lock will remain, however, after the transaction is aborted.

NOTE:

GemStone Smalltalk provides a number of locking methods in the server for which there is no corresponding implementation in GemBuilder for Java. For information, refer to the image comments and the GemStone/S Programming Guide.

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 method removes a specific lock:

```
aGbjObject.removeLock()
```

To clear all locks for the session if the transaction is successfully committed, send this message:

```
aGbjSession.commitAndReleaseLocks()
```

Reduced-Conflict Classes

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

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

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

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

Here are brief descriptions of the reduced-conflict classes. For details about these classes and their usage, see the *GemStone/S Programming Guide* and comments in the GemStone image.

- RcCounter maintains an integral value that can be incremented or decremented. A single instance of RcCounter can be shared among multiple concurrent sessions without conflict.
- RcIdentityBag provides the same functionality as IdentityBag, except that no conflict occurs on instances of RcIdentityBag when a number of users read objects in the bag or add objects to the bag at the same time. Nor is there a conflict when one user removes an object from the bag while other users are adding objects, or when a number of users remove objects from the bag at the same time, so long as no more than one of them tries to remove the last occurrence of an object.
- RcKeyValueDictionary provides the same functionality as KeyValueDictionary except that no conflict occurs on instances of RcKeyValueDictionary when users read values in the dictionary or add keys and values to it (unless one tries to add a key that already exists) or when users remove keys from the dictionary at the same time (unless more than one user tries to remove the same key at the same time).

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

Chapter

6 Observing Session and Server Events

Overview

GemBuilder for Java (GBJ) provides two interfaces to support monitoring of events: GbjObserver and GbjObservable.

The GbjObserver interface consists of a single method, update(), through which objects are notified of events transpiring in other objects of interest. Each class that will be an observer must implement this interface. The action to be taken depends largely on whether the object being observed is a GemBuilder for Java session or an entity in the server.

The GbjObservable interface consists of methods that manipulate the list of observers to be notified and initiate the notification process. The class GbjSession implements this interface, and ordinarily it is the only class that needs to do so.

Observing Session Events

Your client can receive notification of significant session events by registering its interest with the object in which the action takes place. For instance, the Help Request Browser needs to be notified of a change committed through an editing dialog open on a particular help request so the browser can update its display.

Sessions report client events using the update() message, which has three parameters: *obj* (a GbjObservable), *notificationMessage* (a String), and *argument* (an Object). The second and third parameters map to events in the following way:

Event	notificationMessage	argument
Transaction commit	"commit"	null
Transaction abort	"abort"	null
Transaction begun, in manual transaction mode	"begin"	null
Informational message	"message"	message string
Close of session	"close"	null

Observers of session events are notified in the same thread that caused the event to take place. Notification is received only for events in the client session, not for similar actions taking place in Smalltalk code being executed in the server. That is, invoking `mySession.commitTransaction()` in the client results in notification to observers, but invoking `System Class >> commitTransaction` in the server does not.

Client objects can explicitly trigger informational message events by invoking notifyObservers(), thereby passing information about events that otherwise would not be subject to notification. For example, an informational message event could be used programmatically to provide notification of a commit initiated in the server by means of doit().

To Monitor Session Events

1. Signify interest by sending addObserver() to the current session. For example:

   ```
   mySession.addObserver(this);
   ```

2. Implement GbjObserver to handle the notification. For example, when the session commits or aborts the current transaction, or begins a transaction in manual transaction mode, the Help Request Browser calls its method to reinitialize the display. If the session is closed, the browser closes itself.

Example 6.1 Handling Session Events

```
public void update(GbjObservable aSession, String message, Object argument)
{
    if ((message.equals("abort")) ||
        (message.equals("commit")) ||
        (message.equals("begin"))) {
             this.reinitialize();
    }
    if ((message.equals("close")) {
        this.sessionClosed();
    }
}
```

Observing Server Events

Clients can observe events in the server that are of general interest, such as object change notification and Gem to Gem signals. The process is much the same as for monitoring session events, except that the notification always is in a different thread from the one the application is using. (For information about monitoring session events, see “Observing Session Events” on page 71.)

For server events, the update() parameter *notificationMessage* always has the value "event", and *argument* is an instance of GbjException.

NOTE:

Because notification of server events arrives in a different thread, programmers must ensure actions the observer takes are thread-safe with respect to the application. Also, since notification of the event does not suspend execution in GemStone, the observer may have to wait until the currently executing requests finish.

Although the server events of interest are represented in the GemStone server as exceptions (as are errors) GemBuilder for Java distinguishes them, notifying observers of the event rather than throwing them as GbjExceptions.

Clients can identify event circumstances by comparing the number member of the exception instance with those defined in class GbjGemStoneErrors. This example obtains the GbjException, then checks whether it represents a Gem-to-Gem signal:

Example 6.2 Handling a Server Events

```
public void update(GbjObservable session, String message, Object arg) {
    if (message.equals("event")) {
        GbjException event = (GbjException)arg;
        if (event.number ==
            GbjException.kernel.RT_ERR_SIGNAL_GEMSTONE_SESSION) {
            // handle signal, then re-enable signal reception
            try {
                event.session.doit("System " +
                  "enableSignaledGemStoneSessionError");
            } catch (GbjException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Chapter 7

Deploying Your Application

Overview

Clients using GemBuilder for Java (GBJ) can be deployed in two ways, as shown in the accompanying illustration:

- as applets that run in the context of a Java-enabled web browser
- as standalone Java applications

Figure 7.1 GemBuilder for Java Deployment

Deployment Steps

This topic explains the steps you should take in deploying your GemBuilder for Java client as a applet (to be run in a Web browser) and as a standalone application.

To Deploy an Applet

1. Install the HTML page containing the `<APPLET>` tag on the HTTP server, and install the applet class file in the location specified in the tag's `CODE` and `CODEBASE` attributes.

2. Install the GemBuilder for Java class library, `gbj30.jar`, either local to the browser or in the same location as the applet class file.
3. Unless they are coded in the applet, provide the server's name (such as gs64stone) and netldi name to the user. Also provide a GemStone userId (account name) and password.

To Deploy a Standalone Application

1. Unless they are coded in the application, provide the server's name (such as gs64stone) and gem service. Also provide a GemStone userId (account name) and password.
2. Make sure a GemStone NetLDI is running on the Stone's machine.
3. Make sure the Java runtime system (typically `...\bin\java`) is in the user's path.
4. Install the GemBuilder for Java class library, `gbj30.jar`, where it will be accessible to the client, and add the library's location to the `CLASSPATH` in the manner required by your runtime Java system.

Appendix

Glossary

Adapter

An object that adapts messages for another object, converting them to a form the receiving object can understand.

Applet

A Java program that is meant to be run in the context of a Java-compatible browser, rather than as a standalone program. Another kind of Java program, called an *application*, can run independently of a browser.

Enumeration

An object in Java that is used to iterate over the contents of a collection. It is similar to a Stream object in Smalltalk.

Firewall

A gatekeeper computer that protects a local network by filtering traffic to and from an external network, such as the Internet.

Gem

A GemStone server process that provides server access to clients. Currently, each session connects to the server through a dedicated Gem process.

Marshal

The process of serializing an object into a stream before sending it to another process.

NetLDI

A GemStone network server process, which provides information services and spawns other processes for GemStone clients.

Observer

A client object that receives notification of noteworthy events in a session or in the server.

Persistent Object

An object that is stored in the GemStone server and accessible through a root object.

Simple Data Type

One of the following types, which can be stored in the cachedValue field of a GbjObject: Integer, Long, Float, Double, Character, Boolean, null, Date, String, gemstone.gbj.DoubleByteString, or a serialized collection of proxies.

Stone

A GemStone process that oversees other server processes; it is also known as the repository monitor.

Stub

An object that forwards messages to an object in another program, possibly on a different computer. GemBuilder for Java uses stubs to forward messages from client objects to objects in the GemStone server.

Index

F

G

H

I

J

L

M

N

O

P

R

S

T