Designing components for the jungle

This chapter covers:
- Jungle Safari Shipping Company case study
- Stateful versus stateless review
- Building stateless components
- Overview of different component properties
In talking with many of you, we frequently field inquiries regarding stateful versus stateless component design, with many of you saying that you have only a few dozen users. Stateless component design is quintessential in developing robust, scalable, distributed applications. Sure, you may have only a few users, but how are you planning on taking advantage of Jaguar’s clustering capabilities with that stateful design? How are you going to scale when the user base grows from dozens to hundreds or thousands? Although Jaguar 3.5 introduced the ability to provide stateful failover, this feature is enabled only for components developed in Java (the failover algorithm uses Java serialization) and not other languages. The U.S. Navy has a saying that holds true in virtually every aspect of life. The saying is affectionately known as the seven Ps (for this book, the fifth P was modified):

Proper prior planning prevents pathetically poor performances.

In this chapter, we are going to look at writing good, clean component designs. We will discuss different design techniques and discuss the pros and cons of each technique. Both stateful and stateless approaches will be described, although the conversation will lean toward stateless techniques. We will end the chapter discussing the Memento design pattern.

2.1 What is the Jungle Safari Shipping Company?

Throughout Taming Jaguar, we will work with a fictitious company named the Jungle Safari Shipping Company. Safari Shipping runs a profitable business picking up large packages and delivering them deep into the heavily populated Jaguar jungle.

Lately, Safari has come under immense pressure from customers to provide a more real-time, accurate picture of the shipping status of a package. After evaluating vendors, Safari has decided to leverage its existing PowerBuilder experience. The company also feels that it has seen the future and that future is J2EE, so the company wants to use this project to begin to make the move over to PowerJ.

Safari runs a lean operation, and as such doesn’t require a large complex data model to implement its application (after all, we only have so many pages!). Figure 2.1 shows the tables used by the Safari Online Shipping (SOS) web site. To briefly cover the data model, customers are assigned a userID and password that grants them access to the web site to view the shipping status of their packages. Customers have the ability to indicate their notification preference (email, fax, and so on) in the notify_type column of the package table. When populated, the system will automatically send out a detailed delivery report, including the delivery date and signature, via the appropriate medium.

Safari’s software must be able to take its delivery details and disseminate that information via various broadcast mediums. These targeted broadcasts are to the company’s most important asset, its loyal customer base. Safari routinely picks up the package
from the customer and brings it to its shipping hub (sometimes even via riverboat on the Amazon). Once the package is at the hub, workers use a PowerBuilder application to enter the details of package.

Later, customers who desire delivery notification will have the ability to log on to the SOS web site and view a list of all the packages they have shipped. There, the customer can inspect the details of a package and optionally request delivery notification through one of the available notification types. Initially, Safari has identified that the broadcast notification service will include email, fax, and pager.

As stated earlier, Safari’s dedicated team of information technology (IT) staff and professional consultants has made the decision to use Sybase tools, including Jaguar CTS, and will be using both PowerJ and PowerBuilder for component development.

The IT team has decided to use PowerBuilder to develop the fax component, while PowerJ will be used for the email and paging. There is a rumor around the office that in the near future the company may wish to add support for another broadcast medium: synthesized voice (for voice mail messages).

Throughout _Taming Jaguar_, we will focus on component design principles and techniques to help Safari Shipping provide a powerful notification interface for its shipping business to make customers happier.

### 2.2 What are the different Jaguar component types?

Jaguar provides three distinctly different component types. Each component type is designed for a different purpose and each has its own set of strengths and weaknesses.
The actual implementation and instantiation of a component inside of Jaguar is directly related to the associated component properties for each type. We will look at these properties shortly, but first, let’s review the three types in table 2.1:

<table>
<thead>
<tr>
<th>Table 2.1 Jaguar component types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard</strong></td>
</tr>
<tr>
<td><strong>Shared</strong></td>
</tr>
<tr>
<td><strong>Service</strong></td>
</tr>
</tbody>
</table>

2.3 **What’s on a Component Properties Instances tab page?**

A component has a multitude of properties, but the discussion here focuses on the properties found on the Instances tab page on the Component Properties dialog. These properties play a crucial role in Jaguar’s approach to instantiation and access to a component. Each of these properties will be discussed below in detail in table 2.2, outlining what each is individually as well as how each property relates to the others and the net effect of the combination of properties on the component. The component properties dialog, shown in figure 2.2, is accessed inside Jaguar Manager through the right mouse button menu on a component.

<table>
<thead>
<tr>
<th>Table 2.2 Properties found on the Instances tab page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reentrant</strong></td>
</tr>
<tr>
<td><strong>Transient</strong></td>
</tr>
<tr>
<td><strong>Stateless</strong></td>
</tr>
</tbody>
</table>
Stateless (continued)

Checking the stateless property does not automatically enable Automatic Demarcation/Deactivation, nor does it necessarily deactivate the component instance after a method call.

Pooling

When pooling is turned on, Jaguar indefinitely postpones the physical destruction of the component instance. Pooling is another property that enhances overall component performance when turned on because Jaguar doesn’t waste time instantiating and destroying resources after each client has finished working with a component. Also, when turned on, Jaguar does not invoke the canReuse() function or the canbePooled! event. This event (or method) fires on a component only when pooling is set to false, providing the developer an opportunity to control if a component should or should not be pooled. Pooling of a stateful SessionBean is illegal.

Bind object

Bind object is a feature unique to EAServer at this time; this feature is not currently implemented by any of the component models presently supported in EAServer. In effect, it will permit multiple clients to call the same component instance via different containers (Jaguar Servers). This is different from a shared object that is tied to a single container. The idea is that such a component would serve as a storage component for persistent data across the cluster, and the container’s transactional services ensure the data is in effect replicated to each instance on each container.

Bind thread

By turning on bind thread, you are indicating that any method calls on the component must use the same thread Jaguar used to create the instance of the component because data is stored inside of that thread using thread local storage (TLS). This property is extremely critical for PowerBuilder and COM components deployed to a Windows NT machine. If the component is deployed to a Jaguar server running on any flavor of UNIX, then this property is ignored, even if the component was developed inside of PowerBuilder. In the Windows environment, PowerBuilder’s nonvisual DataStore actually uses internal resources that uses Windows’ TLS. However, the need for enabling bind thread is applicable only if the DataStore is stored in an instance variable and is not destroyed between method calls.
Bind thread
(continued)

In other words, if you use local variables, you can safely leave bind thread turned off. Non-Windows PowerBuilder virtual machines were written in such a fashion that there is no reliance on TLS. COM objects rely on TLS.

Sharing and Concurrency

These two properties are easiest to explain when they are discussed together. This is due to the inherent dependencies upon one another and the vast implementation difference between the combinations of the two settings. Sharing dictates the number of component instances Jaguar can create. Whenever sharing is turned on, Jaguar can create only a single instance of a component class. Concurrency determines if clients can simultaneously invoke method calls on the same instance. When concurrency is enabled, Jaguar allows the methods of a single instance to run in separate threads, and therefore to be accessed by more than one client at a time. It is worth noting that the EJB specification explicitly restricts the ability of a single instance to simultaneously service requests from multiple clients. For each scenario we are about to discuss, assume two clients attempting to interact with the same component class simultaneously.

Concurrency: off
Sharing: off
If you have both concurrency and sharing turned off, Jaguar will create multiple instances of a component class for multiple clients. However, only one instance of the component class can be active at any point in time, and Jaguar will block the creation of additional components if one of the instances is presently active because concurrency is turned off. For example, \texttt{methodA()} from Client 1 will be executed in Instance 1, and \texttt{methodA()} from Client 2 will be executed in Instance 2. But Client 2, however, will have to wait until the method call on Instance 1 finishes before its method call on Instance 2 is started.

Concurrency: on
Sharing: off
If you have concurrency turned on and sharing turned off, and the clients are both attempting to execute methods at the same time on the component class, Jaguar will create multiple instances of the component and will execute each method in parallel, or concurrently, on separate threads in separate instances.

Concurrency: on
Sharing: on
If you have both concurrency and sharing turned on, Jaguar will create only a single instance of a component class, which both clients will execute against, and there is the potential for multiple clients to be working with this single instance at any given point in time. PowerBuilder doesn’t support this combination because a PowerBuilder component is not thread-safe.

Concurrency: off
Sharing: on
If you have concurrency turned off and sharing turned on, Jaguar will create a single instance, against which both clients will execute. At any given moment in time, however, only one client may be actively working inside of the component, period. All other requests are queued and executed serially.

\textbf{WARNING: PowerBuilder Developers:} PowerBuilder components cannot support multiple methods inside of the same component executing in parallel because PowerBuilder components are not thread-safe.
2.4 What is the difference between stateful and stateless?

Stateful and stateless are terms that refer to the lifetime or duration of a component after a method invocation is made from a client connection and the execution of that method has finished. Stateful components have the ability to utilize instance variables to store persistent, client-specific data in between method invocations. These components are dedicated to a single client session.

For this reason, PowerBuilder components should leave concurrency turned off when sharing is turned on! Our discussion of sharing and concurrency wouldn’t be complete if we didn’t discuss the role of bind thread in conjunction with these two properties. If sharing and bind thread are turned on, this forces Jaguar to invalidate the concurrency option regardless of the actual setting because the component is forced to remain associated with the thread that created it. With sharing turned on (resulting in at most a single instance), it is illogical and not possible to have multiple threads executing in parallel against the component.

Unfortunately, time elapses between method calls from the client to the server, and during this time the server is using resources (memory) to maintain the component’s state. This approach doesn’t scale well as the number of concurrent users increases. In figure 2.3, we see two clients connecting to an instance of Jaguar Component A. Each client then invokes methodA(). After the call to methodA, the client is waiting for the user to acknowledge and respond to the return value before invoking methodB(). Pay particular attention to the fact that for each client, both methods are guaranteed to execute against the same instance inside Jaguar, and that the component will remain dedicated to the client session that connected to it. This is good in a stateful component because we may have stored state in methodA that we will use in our subsequent call to methodB. However, these instances remain bound to the client session regardless of the amount of time a user
CHAPTER 2
Designing components for the jungle

takes to respond, not to exceed the timeout property of the component (timeout=0 means indefinitely).

Stateful designs require the server to maintain resources for a component even though the associated client may not be actively involved with that component. It is because of this requirement that a stateful design does not scale well when the server is faced with a large increase in the number of client connections. However, a stateful component’s unique ability to retain instance data between method calls does provide advantages in certain business rule implementations, and although we will emphasize a stateless design, our intent is not to stereotype all stateful components as a bad design. In fact, some business process implementations are cleaner using a stateful design.

A stateless component does not have the ability to use instance variables to maintain client-specific data between method invocations. This does not mean that stateless components are not allowed to use instance variables. In fact, there are a large number of production-level stateless components the authors have seen that successfully use instance variables. Storing a component reference is one example of a stateless component using an instance variable. The overhead required to look up the component would only occur once, and each method execution on the stateless component can reuse the instance. This technique results in a slight performance gain on the component. The instance variable is instantiated in the constructor! event of the component and reset in the destructor! event of the component by the developer. As another example, a component can partition redundant logic inside of private or protected methods that are invoked from the public method. This keeps the method signatures of the private and protected methods small because they have access to the instance variables. Remember, this approach assumes the instance variable stores either non-client-specific data or data that is reset in the deactivate! event. In the event a developer uses this type of logic with DataStores or COM objects, it is suggested that the component set its bind thread property to true. As discussed previously in this chapter, if it is not set, unexpected results may occur because future method invocations may be running on a different thread—a thread that is unable to access data stored through the use of TLS.

If the state of the instance variables needs to persist between component method calls, prior to resetting the instance variables, their values need to be stored somewhere else besides inside the component. In fact, it is very common for a stateless component to persist its state somewhere external to Jaguar, like a database or a flat file. In figure 2.4, we see the same two clients from earlier invoking the same methods. The client still presents the return value from methodA() to the user in between method invocations and waits for the user to respond before invoking the second method, but this time Jaguar does not hold onto the component instance between the calls, and of course does not maintain any state values either.
In fact, if the components are multithreaded and not shared (concurrency=true, sharing=false), Client 1 and Client 2 could theoretically be invoking `methodA()` and `methodB()` on different component instances at the same time, but in different threads. Given these settings, a stateless design allows Jaguar to better manage incoming client method requests. Jaguar is now in a position to determine that it does not even need to create another instance of the component to satisfy the requests from a new client. Instead, if it can, it will just grab an available thread from Jaguar’s thread pool and associate that thread with the requested method, conserving valuable system resources.

The concern may arise that if it is possible to invoke only a single method, why go through the trouble of using instance variables and persisting state? This is actually an easy question to answer. A stateless component is bound to a single client only during the execution of a single `public` method. At the conclusion of the public method, the component is released, and typically put back into the instance pool for the next client who requests the component. Notice the emphasis here on public method. A component’s class can support public, private, and protected methods, regardless of the development language. It is extremely common for developers to partition their script, pulling out recurring code and placing it into a function to conserve the object’s footprint in memory.

A stateless component will live only for the life of the single public method invocation; however, to stress a point made earlier, that public method may invoke other private or protected methods that reference and utilize component instance variables. This is one reason why developers may use instance variables inside of a stateless component. In this case, however, developers must pay particular attention to the component properties, discussed earlier in the chapter, if any of the instance variables are instantiated and populated in the `constructor!` event of the component.

**NOTE** Although there is no requirement to reset the instance variable data to default values during the execution of the `deactivate!` event, it is strongly recommended that developers form the habit of initializing their instance variables in the `activate!` event or clearing their variables in the `deactivate!` event. If a developer does not clear these values, the next client will inherit the value of the instance variables, and obviously
Designing components for the jungle

this could lead to undesirable results and a difficult situation to debug. The Jaguar component life cycle is not covered in this book. For more background information on the Jaguar component life cycle and how it impacts component state and its transactions, see chapter 8, “The Jaguar Component Life Cycle,” in Jaguar Development with PowerBuilder 7.

2.5 Is my component stateful or stateless?

Typically, a stateful component will set the Automatic Demarcation/Deactivation property to false, set the transaction property appropriately, set the instance timeout property greater than zero, and create a method (such as of_destroy()) to allow clients to deactivate the component. Figure 2.5 shows the Transactions tab page of the Component Properties dialog.

Stateful components may opt to leave the component’s timeout property set to zero. The importance of setting this timeout property cannot be stressed enough though. Zero indicates infinity, which means that the component will never deactivate itself unless it is explicitly asked to by the client. In the event the client inadvertently forgets to deactivate the component, the timeout property would kick in (if set above zero) and deactivate the component automatically, firing the deactivate! event in the process. If the timeout property is not set, clients forgetting to deactivate a component through setComplete() / completeWork() or set-
Abort() / rollbackWork() force a significant waste of server resources, potentially bringing the Jaguar server to its knees over a period of time.

A stateless component is much simpler to define compared to a stateful component. The only required setting needed to create a stateless component is Automatic Demarcation/Deactivation. When this property is turned on, the component is stateless. Period. Figure 2.5 shows the SurfSideVideoPB/n_customer component, included with Jaguar as a sample. Notice that Automatic Demarcation/Deactivation is selected, making the component stateless.

It is worth mentioning that with the advent of EAServer 3.5, Sybase introduced a new component property called com.sybase.jaguar.component.stateless. In the SurfSideVideoPB/n_customer pictured in figure 2.5, this property is set to false, yet the component is still stateless because of the demarcation setting. Turning this stateless property on causes the activate! and deactivate! events (and the corresponding Java method equivalents) in components to cease firing. However, using the stateless property by itself will not cause the component to become automatically deactivated after a method call.

2.6 How do I build a stateful component?

There are logically three different approaches developers can take when building a stateful component. In each of three approaches, Automatic Demarcation/Deactivation is turned off. The primary difference between the approaches is the way the component interacts with Jaguar-controlled database transactions. The different transactional abilities of a component are pictured in figure 2.5.

The first approach we will look at results in a stateful component with the Jaguar transaction attribute of Not Supported. In this scenario, the component is still capable of working against a database, including updating data, but the component does not participate in any of Jaguar’s transactions. In other words, the complete transaction logic is controlled by the developer, and Jaguar will never issue a commit or rollback command to the database on behalf of the developer. It is the sole responsibility of the developer to ensure that the component completes its database transaction properly. An important note is that a component that uses Not Supported cannot share its transaction with another Jaguar component.

Another common approach used to build a stateful component is known as early deactivation. A stateful component using early deactivation sets its Jaguar transaction attribute to either Required or Requires New. For example, let’s assume that our component has four setxxxxx() methods and a method called completeUpdate(Boolean). The client will invoke each of the set methods, in a logical order for the business rule being implemented. After all the necessary data has been passed to the component, the client invokes the completeUpdate() method to
save the data to the database. After running the appropriate SQL, the component
will call setComplete() / completeWork() or setAbort() / rollbackWork() to vote on the transaction based on error processing and business logic. Jaguar will use the vote to commit or rollback the transaction. It is important to note that the component will be deactivated from the client session, and all the state stored in memory is lost. This is why the term early deactivation is used, as Jaguar will deactivate the component but not the client.

The final stateful component design approach is a hybrid. In this example, we have a pure stateful component with the same set of methods described above, but the Jaguar transaction attribute is typically set to Not Supported. In the hybrid approach, the various set methods are invoked as they were before, and then when the completeUpdate() method call is made, it makes a call to another component, usually stateless, to handle the transaction processing, passing in all the instance data the component collected. In this case, the management of the transaction is deferred to the component receiving the call from our stateful component so that the stateful component remains activated to the client session.

2.7 How do I build a stateless component?

After reviewing stateful versus stateless components, and how to build a stateful component, the next step is to discuss how to build a stateless component. Building a stateless component is actually very easy. The real dilemma is how to maintain any stateful information between method calls. This is a very broad issue, and there are probably as many answers as there are consulting firms in a large city. For the rest of this chapter, we will be looking at different design techniques that can be applied to create a stateless component and manage its instance data. The following statement serves as a synopsis of our problem:

Instance variable information generated in a stateless component during a method invocation is lost at the conclusion of the method. Obviously, this information may be needed in future method invocations. Our goal in the remainder of this chapter is to provide insight into how a developer can manage this information between method invocations.

We will be reviewing a number of possibilities for designing and managing stateless components. These suggestions will attempt to point out both positive and negative characteristics of the approach.

- Client caching design
- Database caching design
- Flat-file caching design
- Shared component caching design
2.7.1 The client caching design

The theory behind the client caching design is extremely straightforward and very easy to implement. Clients already have to maintain a variable for the Jaguar component proxy, so it would not be too much trouble to add a few more variables on the client and maintain our state information on the client. At first glance, this seems like a very acceptable approach.

When applying a client caching design, the function signature (the method name, the data types of its parameters, and the return value) is extended to include a number of pass-by-reference parameters. These parameters provide state initialization at the beginning of the method invocation and then are written to (assigned) before the end of the method’s invocation to be returned to the client for storage until future use. A more extensive state requires a larger number of required method parameters.

Unfortunately, if we step back and think about this approach, we begin to realize that it violates a major rule of object-oriented and component development: encapsulation. To simplify, encapsulation is information hiding or the idea of limiting the access to all of the internal details of an object to the outside world. Violating this rule dispenses with object maintainability, not to mention data integrity, in a traditional two-tier Client/Server application, and even more so in an n-tier, distributed application.

When another variable needs to be added and managed, an application needs to be rebuilt on both the server as well as the client, and this needs to happen simultaneously. In other words, you don’t have the luxury of updating the server component on Monday and the client on Tuesday. This means application downtime and goes against our goal of achieving a highly available application. Because there may be thousands of clients spread across the globe, we are now looking at a major dilemma.

All in all, this technique is good for getting your feet wet in a simple research and development application, but not suggested for a large scale, mission critical production application where the goal is to be highly available.

NOTE The nature of a thin-web client does alleviate some of the problems associated with deploying a change to the server and client because the client application is still technically stored on a server and downloaded each time it is used. Web clients typically use client caching of state in a unique way. They use a cookie that stores state in name/value pairs in a file which is passed back and forth between the web client and the web server.
CHAPTER 2
Designing components for the jungle

REPORT CARD
Client caching design

<table>
<thead>
<tr>
<th>Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>This design is easy to understand and initially implement.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger function signatures are required in order to include extra state information.</td>
</tr>
<tr>
<td>Encapsulation is violated.</td>
</tr>
<tr>
<td>Network traffic is increased, which may require you to increase your network bandwidth.</td>
</tr>
</tbody>
</table>

2.7.2 The database caching design

The database caching design is an above-average design that typically represents the most realistic solution. The primary benefit of using the database to store a component’s state lies in the fact that the component is most likely already interacting with the database, so there is little to no effort required to get to the database. Unfortunately, many database administrators do not like the idea of transient data being stored inside the database and there are technical drawbacks to implementing the database caching design. Some databases, such as Oracle, do not support temporary tables to the extent that other vendors do. This results in a potentially large number of permanent tables holding temporary data. Also, because there is a high volume of interaction with these persistent-temporary tables, it is possible on a poorly tuned server to lead to a bottleneck in application performance.

![Connection cache design](image)

Figure 2.6
This illustration shows a sample connection cache design. It uses Adaptive Server Anywhere to store stateful data between method calls.

Another technical drawback is that relational databases traditionally cannot map persistent object data in a straightforward manner. Objects, by nature, can be quite complex. Mapping the instance data in the object to a single row in a table may not be straightforward or desirable. In fact, this is where object-oriented databases (OODBMS) tend to shine due to their unique ability to store an instance of an object as easily as if it were only a couple of text fields.

No where is it written that the Jaguar server is allowed to work only with a single connection cache. Utilize a different database engine or instance for storing the
primary business data—the stuff that needs to be backed up on a routine basis and represents the mission critical information than you do to manage your state. Figure 2.6 shows an architecture with an instance of Adaptive Server Anywhere used to manage the stateful session data and two additional databases located elsewhere for the application’s persistent data.

One of Jaguar’s strongest features is its ability to provide a Highly Available/Load Balanced implementation. These capabilities can be exploited only in a clustered environment, which involves two or more Jaguar servers working together to eliminate a single point of failure (we discussed setting up and administering a Jaguar cluster in chapter 1). Compared to all of the stateless component instance data caching techniques this chapter will look at, this is the most realistic option available because it ensures that state information will be available to the component regardless of the physical Jaguar server on which the component is executing. By using a database caching technique, new Jaguar servers can be added to the cluster, increasing the application’s scalability, without any extra work by the administrator or the developer.

One of the concerns that ultimately comes up after the decision is reached to use the database caching design is that of how the data should be indexed. This dilemma is quickly solved, courtesy of Jaguar. When a client establishes a connection with the Jaguar server, Jaguar is responsible for authenticating the users (making sure the users are who they say they are). After authentication occurs, Jaguar assigns a sessionID to the client session. The session ID is statistically guaranteed to be unique—even across different Jaguar servers. Without going into much more detail, this session ID can be conceptually thought of as akin to DCE’s universally unique identifier (UUID), or for Microsoft users, the globally unique identifier (GUID). Combine the session ID with the property name, and you have a primary key. This technique works well when the client uses a persistent connection. However, when a client does not rely on a persistent connection (such as Web / PowerDynamo), and uses Jaguar connection pooling, this technique is not viable, and another arbitrary session ID generation algorithm should be employed. For example, a simple implementation could generate a random number and concatenate the login ID of the user.

One final note about utilizing the database to store stateful data from stateless components is that it will be necessary to design a mechanism that will automatically purge out data after it is no longer needed. One approach is to have the stateless component have a method that the client can call letting the component know that the state can be flushed. A better approach is to use a Jaguar service object that deletes data based on the value of a last modified time stamp and determines if a predefined time duration has elapsed in which the data could safely be deleted.
At the end of this chapter, we will be discussing the Memento design pattern and use it to implement the database caching design we have discussed here.

**REPORT CARD**

**Database caching design**

**Pros**
- Many components already have connectivity to the database, so little to no extra programming required.
- This design can easily be implemented in a secondary database cache against an Adaptive Server Anywhere or similar database located directly on the Jaguar server.
- This design adapts to a Jaguar cluster without any component coding changes.
- Jaguar may provide a primary key in many situations—the session ID.

**Cons**
- A good database server with multiple disk controllers is required to keep bottlenecks to a minimum.
- There are possible political implications due to storage of temporary data in permanent tables (on some database systems).
- Network traffic increases, but usually traffic is on a higher bandwidth backbone than client connections.
- Persistent component data may be difficult to map into a relational schema.

### 2.7.3 The flat-file caching design

Utilizing a flat-file caching design is really a step backward and is ill-advised. Yes, file access is second nature for most programmers, making this approach very easy to understand and code. But as Jaguar component developers, we need to look at the big picture. First of all, you the developer would have to write all the management routines, including the file layout design (such as .ini, comma-delimited, or tab-delimited). This isn’t that large of an obstacle to overcome, but think about this next point. How are you going to coordinate file access between components requesting data simultaneously, assuming you used a single file to manage the component’s state? Some of you may have responded by saying we’ll just use multiple files, one for each user.

Using files to store stateful data between method invocations is probably acceptable in a design that will use only a single Jaguar server. However, most designs are going to want to take advantage of Jaguar’s advanced capabilities. For anyone who wants to design a system capable of exploiting all of Jaguar’s capabilities, we recommend using a database caching design over a flat-file design. To oversimplify, a database is specifically designed to handle concurrent access and remove platform-specific file access issues. In addition, Jaguar clusters can point to a single database to gain access to pertinent stateful data, which is much simpler than maintaining multiple files across multiple machines.
Okay, so you’ve decided against our recommendation and are going to use a flat-file caching design anyway. Let’s see what advice we can offer. First of all, to eliminate file access problems, it is probably very wise to implement one file per user. In this way, multiple instances of a component can be simultaneously reading and writing to their stateful files. Given this, we need to look for a way to generate a unique file name and probably pass it back to the client, where it must be cached for future method calls. So what are our options for generating a unique file name, and more importantly, where would we store these files?

If Windows NT servers are used exclusively, then you could utilize a couple of well-documented Win32 API calls from kernel32.dll, GetTempFileName() and GetTempDirectory().

```c
DWORD GetTempPath(
    DWORD nBufferLength,     // size of buffer
    LPTSTR lpBuffer          // path buffer
);

UINT GetTempFileName(
    LPCTSTR lpPathName       // directory name
    LPCTSTR lpPrefixString   // file name prefix
    UINT uUnique             // integer
    LPTSTR lpTempFileName    // file name buffer
);
```

If you are unsure about locking the component onto Windows NT, then there is another option available. As we did above in the database caching design, we could utilize the session ID assigned to the user as the file name. With this approach, there would be no need to pass anything back to the client for caching. In fact, this approach encapsulates your storage design from the client, making it superior to the two aforementioned Win32 methods.

If the component were developed in Java, developers have another option available to them known as object serialization. Object serialization is a topic in and of itself. To simplify, object serialization converts a Java object into a bit-blob. Once in this form, it can be sent anywhere, including into a file. When the object is needed again, the bit-blob is deserialized back into a Java object. Java handles most of the details of this object-to-blob conversion for you, and in fact, every EJB component supports the java.lang.Serializable interface as part of the EJB specification.

One final note about utilizing a number of flat files to store stateful data from stateless components: It will be necessary to design a mechanism that will automatically purge out old files after they are no longer needed. Regardless of which type of flat-file storage implementation is used, the best approach would be to design a simple Jaguar service that runs as frequently as necessary. Inside of this service, take a snapshot of all the files in the directory that contains the stateful files. Filter out
files that should be deleted by looking at their last modified time stamp and determining if enough time has passed in which the file could safely be deleted. The window that dictates if a file should be considered active or not is arbitrary based on your application requirements.

REPORT CARD
Flat-file caching design

Pros
- Network bandwidth does not increase.
- Operating systems supported by Jaguar CTS have easy-to-use file routines.
- Jaguar provides a unique filename for non-web clients—the session ID.

Cons
- Jaguar’s automatic failover abilities on stateless components are compromised.
- A single file design introduces a bottleneck into the system whereby a client’s component instance is waiting in a queue for file access to read/write data.
- The developer must design a system to purge out the files after so many minutes.

2.7.4 Shared component caching design

Depending upon the language used to implement a shared component, the capabilities of the shared component will differ (PowerBuilder components are not multi-thread capable). The purpose of this section is to discuss the design of your shared component to handle caching of stateful data, so we will not be getting into the technical differences here between shared objects written in PowerBuilder or Java. Review component properties earlier in this chapter, and see chapter 4 for more details on actually implementing a shared component.

Regardless of which language you are using to implement the shared component, if the purpose of the shared component is to cache data, the internal design of the component will probably be the same—name/value pairs associated with a key (necessary to differentiate between different users). We recommend using the session ID as the primary key, either the Jaguar session ID for persistently connected clients or your own arbitrary session ID for clients using pooled Jaguar connections. These keys could be stored inside of a Java hash table, a DataStore, or even in the database using a small table with three columns combining the database caching design with the shared component design, for example:

<table>
<thead>
<tr>
<th>PropsTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session_id</td>
</tr>
<tr>
<td>var_name</td>
</tr>
<tr>
<td>var_value</td>
</tr>
</tbody>
</table>
NOTE Although this is a shared component, the activate! and deactivate! events will fire each time a client establishes a connection to the shared object written in PowerBuilder. Therefore, the initialization of the DataStore should take place in the constructor! event and destruction (DESTROY lds_myDataStore) should take place in the destructor! event; bind thread and sharing properties should be set to true, and concurrency should be set to false. If you are looking for a good example of using a shared component from PowerBuilder, see the section entitled “Shared components” (8.7.1) of chapter 8 in *Jaguar Development with PowerBuilder 7*.

One clear advantage shared components have over everything else we have discussed up until this point is that the entire object is already instantiated and stored in RAM. Accessing memory is always going to outperform a database (even local) or file access. Also, DataStores and hash tables have native functionality, such as the datastore.Find() method, which makes locating a particular name/value pair very simple and very fast.

One problem with this technique, which is difficult to overcome, is the limited ability to propagate the shared component’s stateful data across multiple Jaguar servers in a clustered environment. A common argument is that “we’ll only deploy the shared component on a single server, so it won’t be necessary to propagate the values between the different servers.” This is a short-sighted design because the system cannot take advantage of load balancing or high availability. Combining this solution with the database technique helps solve this problem.

![Figure 2.7](image_url)

*Figure 2.7* This illustration highlights component distribution across a single Jaguar cluster.
Deploying a component, regardless of type, to only a single Jaguar server immediately creates a single point of failure. When designing and implementing an application that must be highly available, it is particularly important to avoid single points of failure. As a Jaguar cluster implements multiple name servers, so too should your design implement a component across multiple servers, as viewed in figure 2.7. In this way, if the server should encounter a catastrophic event, another server can take over the workload, albeit at a performance penalty.

REPORT CARD
Shared component caching design

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Network traffic does not increase.</td>
<td>- Jaguar’s automatic failover abilities are potentially compromised on components required to communicate with a shared object designed to run on a single server (single point of failure).</td>
</tr>
<tr>
<td>- This design can leverage a cache with the session ID as the primary key to store and look up name/value pairs.</td>
<td>- Atomic operations are potentially enforced against the shared object (if developed in PowerBuilder), resulting in a first in, first out (FIFO) queue and a bottleneck in the system.</td>
</tr>
</tbody>
</table>

2.8 What is the Memento design pattern?

Throughout the book we will refer to design patterns. The purpose of this book is not to regurgitate material from patterns books, such as Design Patterns by Gamma et. al. However, we would like to point out various design patterns that fit certain situations in an effort to help you develop better component designs.

We’ve just finished discussing different approaches for storing the internal state of a stateless object. There is a behavioral pattern known as the Memento design pattern. This design pattern is intended to store an object’s internal state so that the object can restore this state at some later point in time. One of the primary benefits of the Memento design pattern is that it accomplishes its task without violating encapsulation. For instance, the client caching design discussed earlier in the chapter revealed that the client would be responsible for storing and managing a stateless component’s state, thereby violating encapsulation and making maintenance down the road a near impossibility.

The Memento design pattern is documented as a stand-alone object that is responsible for saving the state of a requestor object. Ideally, the memento object itself would be stateless and rely on one of the different designs we looked at previously in this chapter to cache state like a database. In the activate! event of our stateless component, we would restore the component’s state by connecting to this
memento class and invoking a method (let’s name it `getState()`), and pass it in our primary key (probably the session ID and the name of the component). In the `deactivate!` event, this same stateless component would again connect to the memento class and invoke a similar method which would store the state (let’s name it `setState()`), and pass it in our primary key. Obviously, we need to work with more than just the primary key—specifically we need to read and write the instance data.

The instance data of each component is most likely going to vary, both in number and in style. The decision to implement a single memento class that works with objects across the board versus a memento class for each component is a difficult one to make. It is recommended to utilize a single memento class that works across the board. This forces a neutral design and allows for future growth with no extra programming required. It also keeps the number of components to a minimum.

To eliminate the data-type issues, we recommend that the data type of each instance variable be encapsulated away from the memento class. This can be accomplished by casting everything to a string. Although this puts pressure on the stateless component in the `activate!`/`deactivate!` events to perform data conversion, it keeps the Memento design pattern very simple. For instance, consider this possible function signature for `getState()` and `setState()`:

```csharp
int getState( string sessionID, &
            string componentName, &
            ref string argNames[], &
            ref string argValues[] )

int setState( string sessionID, &
             string componentName, &
             string argNames[], &
             string argValues[] )
```

The session ID and the component name together represent the primary key for storing the name/value pairs. During the `deactivate!` event, the stateless component would connect to the memento class, cast all of its internal instance variables to a string array, and invoke the `setState()` method. Internally, the `setState()` method may be implemented similar to the following high-level algorithm:

```
DELETE from the db all outdated data tied to the session ID / component name

FOR EACH argument
    INSERT a row into the database
    with the name/value pair
NEXT

RETURN a SUCCESS or FAILURE result
```

When the stateless component runs the `activate!` event, it can restore its state by connecting to the memento class and invoking the `getState()` method, and cast
the values back to the appropriate data type for each of its instance variables. Internally, the `getState()` method may be implemented similar to the following high-level algorithm:

```plaintext
SELECT from the database all data associated with the Session ID and the componentName
Enumerate the name/value pairs into the reference arrays provided by the requestor
RETURN the number of name/value pairs
```

There are a couple of miscellaneous notes about implementing this memento class. First and foremost, use a DataStore instead of embedded SQL when performing the `INSERT` into the database. This should yield better performance and cleaner code inside of the `get` method. In the `set` method, use an embedded SQL `DELETE` statement so that we don’t waste network bandwidth retrieving the old values that are no longer of any use.

Because the memento class is itself stateless and easily pooled, go ahead and define an instance variable to hold your memento class. Create the instance of the memento class in the `constructor!` event and destroy it in the `destructor!` event. Using this technique, the `activate!` and `deactivate!` events can get down to business immediately retrieving and storing data, respectively. This type of design does require an additional step. Over time, data in this table may become stale for a number of reasons, including client session time out and unexpected system failure (such as sudden power loss). It is necessary to periodically clean out this data, and in chapter 4 we will demonstrate Jaguar services—perfect for doing this type of work.

Using a memento class is a great approach for managing stateful data in a stateless world. It is a fully documented design pattern, exposes only two straightforward `get` and `set` methods, and encapsulates the storage mechanism from every component. This encapsulation affords developers the ability to switch between most of the different caching mechanisms described earlier in this chapter without having to recompile and redeploy each and every component. In other words, developers could choose to start out using a flat-file caching mechanism and switch to a database caching technique later simply by rewriting the memento class `get` and `set` methods and redeploying the object.