ARCHITECTURAL DESIGN FOR PERFORMANCE: DETERMINING DISTRIBUTED SYSTEM SPEED FROM AN ARCHITECTURAL PERSPECTIVE

Patricia Carando*
Align360, 1430 Spring Hill Road, Suite 510, McLean, VA 22102, USA

Abstract

Few aspects of system development cause more concern to designers and frustration to users than performance. Most designers believe that they are designing with performance as a primary concern. Why then, are systems deployed that are significantly slower than anticipated, sometimes resulting in a complete design overhaul to meet performance needs? Experience suggests that the reasons are two-fold: (i) A failure to focus at an architectural level when designing for performance; (ii) An inability to gather requisite performance metrics early enough in the design cycle to affect the development outcome. This paper recommends two approaches that address these failings: 1) How to focus on total system throughput based on Use Case scenarios when designing for performance, and 2) How to create an architectural prototype that is used to gather performance metrics prior to making firm design decisions.

Keywords: Distributed system performance, architecture-centric design, architectural prototype, data-intensive Java application tuning

Izvleček

Le redki vidiki razvoja sistemov povzročajo toliko skrbi razvijalcem in toliko slabe volje uporabnikom kot zmogljivost. Razvijalci zvečine mislijo, da pri razvoju poskrbijo za največjo možno zmogljivost. Zakaj se torej uporabljajo sistemi, ki so znatno počasnejši kot je bilo pričakovano, tako da je včasih potrebno zasnovo popolnoma prenoviti, da bi dosegli ustrezno zmogljivost? Izkušnje kažejo, da sta za to dva razloga: 1) V želji, da bi imel sistem kar največjo zmogljivost, se razvijalci ne usmerjajo na določeno raven arhitekture; 2) Razvijalci nimajo možnosti, da bi že med razvojem sistema pravočasno preverili njegovo zmogljivost in tako prilagodili rezultat razvoja. Članek priporoča dva načina reševanja teh slabosti: 1) Upoštevali naj bi delovanje celotnega sistema na temelju scenarijev možne uporabe in 2) Še pred dokončno odločitvijo o njegovi zasnovi naj bi zmogljivost sistema preverjali na prototipih.

■ ■ ■

1. Introduction

Determining if a distributed architecture will meet its performance constraints is a daunting task. Often this determination is made after a considerable percentage of the system has been created; if the constraints are unmet, a performance release is planned. This performance release may involve re-architecting the system based on the newly gathered metrics.

This situation is brought about not because of a lack of concern about performance on the part of designers; rather, it is a result of the fact that one can't measure what doesn't exist. Measuring system performance just before first deployment is too late in the life cycle to impact design decisions, but this is often the first time that a sufficient amount of the architecture has been implemented to allow for metrics gathering.

How can this seemingly circular dilemma be addressed? How can metrics be gathered to validate the performance of an architecture prior to actually implementing it? One way is the creation of an archtectural performance prototype (APP). The purpose of the APP is to implement the most important design decisions relating to performance sufficiently to verify them. Failing that, the APP is an opportunity to experiment with alternate designs that can meet the system performance criteria.

* *Patricia Carando has been designing and building distributed systems for 16 years. Early research work included Distributed Artificial Intelligence systems applied to oil well exploration. For the last 10 years, Ms. Carando has been consulting on commercial, distributed system development in a variety of industries. These include telecommunications, materials provisioning, document management, and system design for fault-tolerant computing. She is currently a principal in the electronic commerce company The e4Speed Initiative in McLean, Virginia.*

This paper illustrates the following:

- How to focus on total system throughput based on Use Case scenarios when designing for performance, and
- How to create an APP that is used to gather performance metrics prior to making firm design decisions.
- Focus of the Recommendations

While applicable to n-tier distributed systems in general, the recommendations in this paper target a Java-based n-tier system, with a significant relational database aspect. A typical n-tier system architecture of this type is illustrated in Figure 1. Because many current Web applications and new electronic commerce applications are of this character, these recommendations are broadly applicable. Of particular concern is the performance of systems implemented in Java. The advent of Enterprise Java Beans (EJB) [7] has made Java server implementation very attractive because of the ease of implementing and deploying a multi-user server. Enhancing the performance of such Java-based servers is an increasingly important issue.

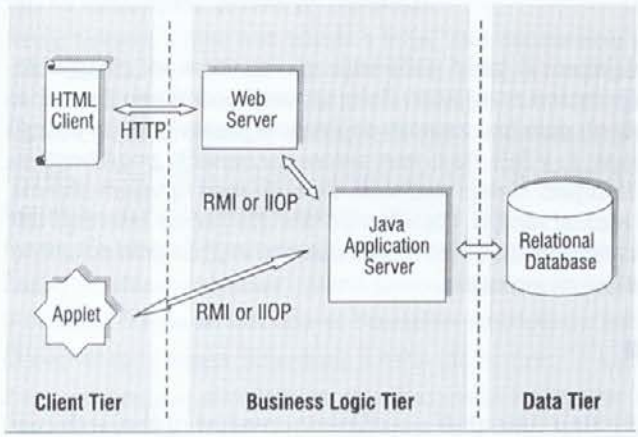


Figure 1: N-Tier Architecture

2. Creating the Architectural Performance Prototype

The body of this paper addresses how to create an APP for a system such as that shown in Figure 1. This figure illustrates a typical architecture that supports Web-based applications. In the *Data Tier*, one or more relational databases provide persistence for the application. In the *Business Logic Tier*, a Java-based Object Request Broker (ORB) or a Java application server applies business rules to data accessed and updated through the *Data Tier*. In the *Client Tier*, one or more implementations of a user interface that support the system's Use Cases direct the activities of the system. Depending on the server technology, the Java server and the Web server functionality may be combined into a single server.

The recommendations cover the following:

a) The selection of one (but not more than two) Use Cases from the system Use Case model that are most likely to represent the most data-intensive or computationally intensive activities.
b) Creation of stimulators to the systems services (databases, Java servers, Web servers) to determine their throughput under conditions of light and heavy use, given the chosen Use Cases.
c) Measurements of the servers' throughput under conditions of heavy and light load to determine lower bounds for performance, as well as typical or average throughput.
d) Suggestions for modifying design and managing expectations should preliminary performance indicators be less than optimal.

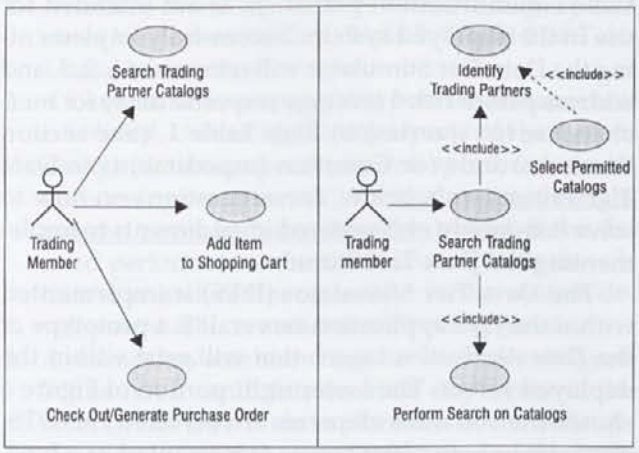
Recommendations in this paper are based on experiences in using this approach on several different projects. As is probably apparent from the context of the suggestions, the approach is best applied as part of the Rational Unified Process [2] (RUP) and can be considered one aspect of RUP's recommendation to develop an architectural prototype.

Selecting Use Cases for Prototyping

When the majority of use cases for a system have been defined, the analyst can begin to scope the use cases for risk. In this paper, we are interested in performance risk, but it is wise to include issues of criticality[1] in the choice of the use cases, as well. To illustrate this, we introduce an example.

Suppose that designers are building a business-to-business eCommerce system that allows corporate trading partners to electronically generate and transmit purchase orders for products. The system allows a trading member to browse their partners' catalogs, select merchandise for purchase, and to generate one or more purchase orders for electronic transmission to the partner corporation.[2] Such a set of capabilities is shown as use-cases in the left-hand side of Figure 2. On the right hand side of the figure is illustrated a more detailed break down of the elements in the *Search Trading Partners' Catalogs* use-case. These include *Identify Trading Partners*, possibly *Select Permitted Catalogs* (based on the Trading Member's alliances) and performing a *Search **on Catalogs*** that meets the Trading Member's *search* criteria.

1 *These are essential functions that the system must perform.*
2 *This example is simplistic, but embodies many of the issues that complex systems face: multiple data sources, inconsistent schema, and international locations that observe varied up times.*



Business-to-Bussiness eCommerce: a Simple Use Case | **Detail of the Use Case for Search Trading Partner Catalogs**

Figure 2: Use Cases for a Shopping System

Criticality and Risk in the Architectural Performance Prototype.

As the designers drill down into the descriptions of the use-cases, they note that the *Search Trading Partner Catalogs* use-case description embodies many unknowns. (These are based on the preconditions, main flow of events, exception flows, and additional notes embodied in the use case description in Figure 3.) Their risk analysis, based on this use-case, is shown in *Table 1*. The table lists a brief description of the risk and an estimate of the severity of the risk.

Risks 1 – 3 in *Table 1* address *criticality* risks—in order to access data from the data sources, logins must be created and schema must be known. These first three items are listed as risks in the table not because of their technical difficulty, but because of expected delays in setting up these capabilities due to bureaucracy and communications problems. Risk 4 is a contingency risk based on the possibility that data may be differently formatted or inconsistent amongst data sources and may need to be transformed to a common format. Risk 4 can be considered both a criticality and a performance risk. Risk 5 also is a criticality and performance risk: when data sources go offline, either intentionally or through some fault, data is not available and connection attempts may cause performance delays. Risks 6 and 7 are purely performance risks: one factor in total system throughput is the response of the data sources (Risk 6). Another factor is the ability of the server to process the data for presentation to the Trading Member.

Search Trading Partners' Catalogs Use Case Description

Search all catalogs of Trading Member's trading partner companies for the items matching the search criteria.

Preconditions:
- Login to trading partner site and access catalog.

Main flow of events
- Issue query
- Consolidate responses

Exception flows:
- Trading partner site may be down for PM
- Trading partner site may be unavailable because of network or system failure

Additional Notes:
- Trading partners are located globally;
- Access characteristics of the trading partner catalogs (schema, login authorization, network connectivity, etc.) are unknown.

Figure 3: Search Trading Partners' Catalogs Use Case Description

#	Task	Risk
1	Identify instance of each catalog	Low
2	Identify relevant query to access item list for each catalog	High
3	Identify login account and permissions for each trading partner site	Moderate
4	Transform differently formatted information into a common display format.	High
5	Determine scheduled outages of partner sites	High
6	Determine average response delay for item queries	High
7	Configure middleware for optimal performance given user load, data accessed.	High

Table 1: Risk Analysis for Search Trading Partners' Catalogs Use Case

The designers decide to prototype the *Search Trading Partners' Catalogs* use-case to validate the architecture and to ascertain that it will meet performance expectations. (Other use-cases in the system—*Select Item for Purchase, Purchase Merchandise*—are addressed similarly and deemed to be less of a performance risk: they are being implemented with known components of well-established characteristics.)

Building the Prototype

Having chosen the use-case to be prototyped, the designers must build the prototype that addresses the greatest performance risk. This prototyping effort consists of two activities. The creation of the Data Tier Stimulator and the creation of the Client Tier Stimulator. The intent of the exercise is to determine a rough estimate of system throughput for the most data intensive function.

Consider Figure 4—a variation on Figure 1—showing the Client Tier, Business Logic Tier, and Data Tier

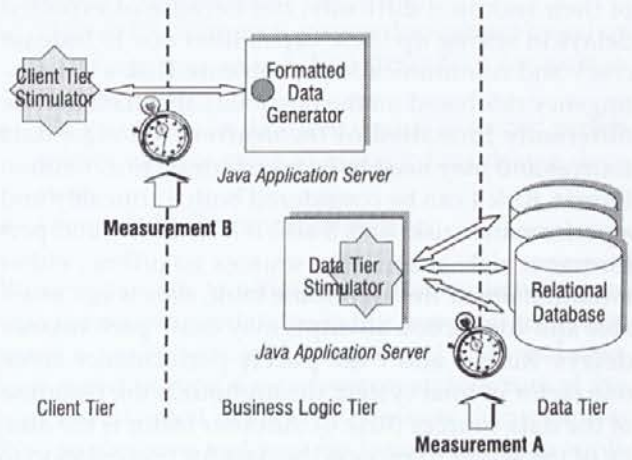


Figure 4: Client and Data Tier Stimulators with Measurement Points

for the distributed application. This figure illustrates the measurement points for the prototype. Measurement A is the Data Tier Stimulator measurement—measuring round-trip time from the server to the data sources. Measurement B is the Client Tier stimulator: a measurement of round-trip time from the client to the server. When the prototype and the measurements have been completed, adding together Measurements A and B should give a rough estimation of minimal throughput times for the most data intensive query.

Addressing the Performance Bottleneck

It is a truism that to optimize the performance of a (distributed) system one must first identify the bottlenecks to performance. Optimizing non-bottlenecks will not increase the throughput of a system. How, then, can one be assured that measuring data access will measure the real bottleneck?

Measuring the total throughput of the system is the goal of the exercise. If data access is a significant aspect of user interactions, it must be a major component of your throughput measurement. Further, data access is (probably) the major source of memory consumption in the server—a notorious source of performance degradation in Java [1], [3], [4], [5], [6]. Having eliminated this as a potential bottleneck in performance, one is free to address other areas of the server where performance issues could arise. Modeling sequence diagrams can be a good source of information on this—pointing out areas where major message activity or computation occurs.

Setting up the Data Tier Stimulator

To determine the response time of the data sources, they must be exercised from the server utilizing the same communication pathways as are intended for use in the deployed system. Successfully implementing the Data Tier Stimulator will retire risks 1, 2, 3, and address part of risk 6 (average response delay for merchandise list queries) in Risk Table 1. (See section Work-Arounds for Common Impediments to Data Tier Prototyping, below, for suggestions on how to solve commonly encountered impediments to implementing the Data Tier Stimulator.)

The Data Tier Stimulator (DTS) is implemented within the Java application server. It is a prototype of the *Data Abstraction Layer*[3] that will exist within the deployed server. The lower right portion of Figure 4 shows the software elements involved in the DTS. These include the Java server (represented as a box), the DTS classes (represented as a star), the communication pathways to the data stores (arrows), and the data stores themselves (cylinders labeled *Relational Database*).

The DTS should issue a query to each of the data stores; the query chosen should be the most data intensive query for the application. Hard-coding the query into a method of each DTS class is sufficient for the prototype. Using a single DTS class per data source is recommended. (See Appendix A for a Java example of a sample DTS class.)

The sequence diagram in *Figure 5* shows the series of events the DTS follows in exercising the data sources. The Merchandise Warehouse starts a timer (see Appendix B for a Java Timer class example) and fans out queries to the three data sources. When all query requests have returned, the timer is stopped. The elapsed time recorded by the timer is the (minimum) time needed to perform the query.

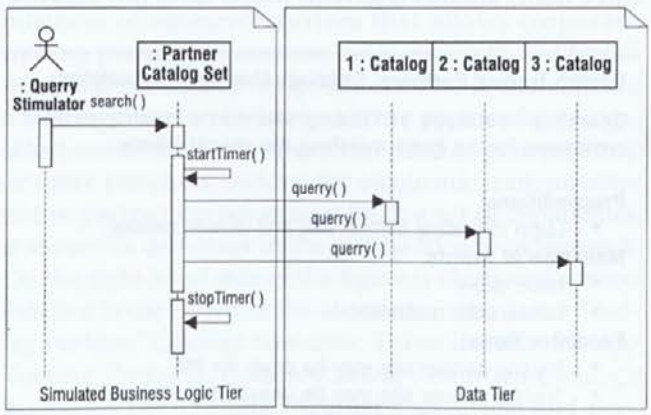


Figure 5: Sequence Diagram of Data Tier Stimulator

While this exercise is deceptively simple, a great deal is accomplished with this prototype:

3 *An architectural layer within a server that insulates the business logic layer from the details of data .*

1. *Necessary access information and knowledge of data stores has been established.* Risks 1, 2, and 3 of *Table 1* have been retired.
2. *A lower bound for accessing data from the data sources is established.* Unless significant aspects of the system change (database speed, network speed, faster multiprocessing on server, etc.) the total system throughput can never be faster than this bound. If the derived performance number is unacceptable there is time to re-architect or reconsider further system development. This is a start at addressing Risk 6.
3. *A rudimentary data abstraction tier has been prototyped.* While not a candidate for final deployment, the code in the DTS is an exploration of the data abstraction layer of the server. Pointing to the functioning prototype that accesses real data can alleviate the fears of stakeholders who may be anxious to begin implementation. This is not a technical issue, but can be very important politically.

Work-Arounds for Common Impediments to Data Tier Prototyping

Even a simple prototype like the Data Tier Stimulator can be difficult to implement. Common problems include:

a) *One or more of the data stores is unavailable.* This can occur for a number of reasons: delays in getting access because of permissions, firewalls, staff availability, etc.
Workaround. Define a test data set on a server that can be used for initial development of your performance prototype until the data store *becomes* available. Determine what the schema of the database is and what you'll have to access to satisfy your user interactions. Get a sample data set and begin your performance prototype using this same set and sample server.

b) *The schema for one or more data stores is undefined.* The general content of a data store may be known, but its full definition may not be in place when the performance prototype is being implemented
Workaround. This situation, while frustrating to performance determinations, can be an opportunity to design the data stores optimally. It is critical that the server designers and data modelers work closely together to design schema that will support queries suggested by the use-cases.

c) *The schema for one or more data stores is changing.* This situation is similar to 2 above, but has the added difficulty that the server will need to support the existing schema until the new schema is operational.
Workaround. In this situation, a very robust data abstraction layer must be implemented that has the same interface for both schema. This will help prevent data updates from rippling through to updates in the entire server.[4]

Setting up the Client Tier Stimulator.

Implementing the Client Tier Stimulator will help to determine the response time to a client request and the memory usage of the server under load. These are issues that are components of Risk 6 (average response delay for merchandise list queries) and Risk 7 (configure middleware for optimal performance given user load, data accessed) in Risk Table 1.

The CTS is implemented within a prototype client that is on a host remote from the server. This client need only issue a request to the server and receive a response. The request must elicit a response based on the same query sent to the data sources by the Data Tier Stimulator. That is, the response should be the consolidated, formatted version of the data that was returned to the Data Tier Stimulator. This is the form the data will have in the deployed system after being merged from the various sources, passed through the data logic, and readied for delivery to the client. While the raw data format and the processed data format may be similar, they are not usually identical. (Some data fields in the raw data format may be suppressed, computed fields added, data fields transformed, etc.)

The sequence diagram in Figure 6 shows the series of events the CTS follows in exercising the server. The Client starts a timer and sends a request to the server. A simulated response is generated and the data is returned to the Client. The timer is stopped at this point The elapsed time recorded by the timer is the (minimum) time needed to return data across the wire to the client. A second timer may be utilized to determine how long the client takes to format the display for the user. Both these timing figures may factor into redesign efforts should elapsed times prove too great.

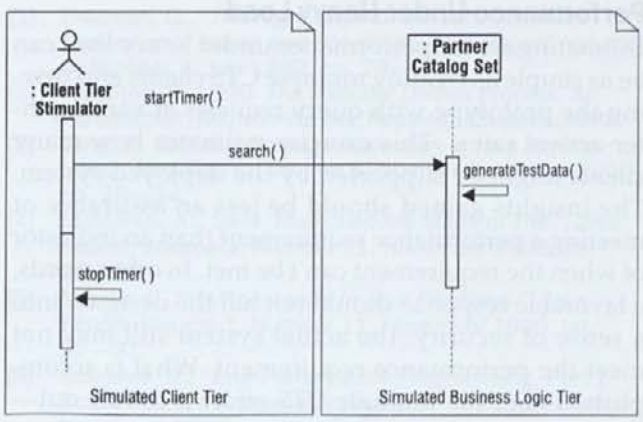


Figure 6: Sequence Diagram of Client Tier Stimulator

4 *Designing to prevent data-update-ripple-through is a discussion topic in its own right and can't be adequately addressed here.*

Determining Performance Bounds

Having completed the two parts of the prototype, the designers can now estimate minimal transit times from client-to-server-to-data–source and back by adding together the measurements from the CTS and the DTS. The minimum response time for the most data intensive query will be CTS + DTS + x, where x is the processing time for applying the business logic and data consolidation rules to the fetched data. If these numbers are "within the ballpark" of acceptable performance, the designers can expand the prototype to look at performance under load and optimal parameters for server sizing. If these numbers are off-scale, then redesign is necessary. Many possibilities for redesign exist; a few are addressed in Section 3.

Memory Configuration

In order to address Risk 7 (configure middleware for optimal performance given user load, data accessed), it is important to know how much memory the server will require to service the heaviest user queries. While there are many other factors that will need to be considered for optimal performance, determining memory usage is one of the most critical for server-side Java. Measuring memory usage for the query employed in the CTS will give an indication of how much memory is dedicated to holding client data. The memory test harness described in [4] can give an initial estimate.

Once a memory usage number is determined, double the figure to get an estimate of the real memory usage in the deployed system. Doubling the initial figure is necessary because the processed version of the data and the raw data will both be resident in memory until the response is sent. Unless the raw data can be read into the server, processed, and efficiently released as the query response is being prepared, roughly twice the memory of the formatted response may be consumed during processing.

Performance Under Heavy Load

Estimating server performance under heavy load can be as simple as creating multiple CTS clients and driving the prototype with query requests at varying inter-arrival rates. This exercise estimates how many clients might be supported by the deployed system. The insights gained should be less an assurance of meeting a performance requirement than an indicator of when the requirement can't be met. In other words, a favorable response should not lull the designer into a sense of security: the actual system still may not meet the performance requirement. What is accomplished with the multiple CTS effort is to rule out—very early—infeasible design approaches.

3. Modifying Design and Managing Expectations

If the performance prototyping of a system goes well and the early metrics indicate the design can easily meet or exceed performance requirements, the designer can happily continue with detailed design. However, should this not be the case, the designer needs to reconsider the approach.

Problems Revealed in the Client Tier Stimulator

Problems revealed by the CTS may include:

1. *The amount of data returned is causing significant delays.* The user may have an unbounded query that returns too much data. Two suggestions for avoiding this situation are:
   Suggestion 1a: Restrict the generality of the query prior to processing.
   Suggestion 1b: If large responses are mandatory, return data incrementally. This breaks the query-response into segments that give the impression of over-all better response.
2. *Response is impacted by memory management infrastructure.* Excessive memory usage has the memory allocator and garbage collector working overtime. If one can't add more memory, consider the following:
   Suggestion 2a: Prefer primitive types. Excessive object creation can cause significant memory allocation overhead [4]. Consider where primitives can replace objects in the design.
   Suggestion 2b: Use less memory by processing query responses–and the raw data that supports them–incrementally. (This is a variation of 1b above.)

Problems Revealed in the Data Tier Stimulator

Early detection of performance problems in the DTS can help in re-architecting (at best) or managing user expectation (at worst). A common problem found by the DTS is:

- *The database query response is too slow.* If the speed of the network and the database servers are not the culprits in slow database performance, enhancement may be accomplished by the following:
  Suggestion a. Change the schema. Optimizing the schema to better fit the needs of the user query can result in significantly better performance.
  Suggestion b. Modify the user interaction. If the query involves the execution of many subqueries, it is beneficial to try to simplify the use-case to minimize the complexity of queries. This approach should only be taken when the schema can't be modified.

When neither alternative is possible, the designer must inform the user community that response will

fall outside the performance boundaries. This knowledge is best transmitted before the developed system is erroneously found at fault.

4. Conclusions

Determining the performance characteristics of a distributed architecture is an anxiety-provoking task, particularly if these characteristics can only be determined after the system has been created. This paper has illustrated how to ascertain these characteristics early in the design process so as to avoid the costly effort of re-architecting after an implementation. Steps illustrated include:

How to select the critical, data intensive use-cases from a risk list for performance prototyping.

a) How to employ the use-cases in creating Data-Tier and Client-Tier-Stimulators to derive performance metrics.
b) How to modify the design and manage expectations if preliminary performance indicators are poorer than what was anticipated.

While not applicable to all distributed system development, these approaches should serve as useful techniques in optimizing for server-side Java systems.

Appendix A.

A Sample Data Store Stimulator Class

The sample class Catalog1 of a class used in the Data Tier Stimulator. It would be employed as illustrated in *Figure 5: Sequence Diagram of Data Tier Stimulator.*

1. Create the instance. Initialize the timer
2. Call the Query method
   - Invoke the startTimer method: Starts the timer (See Appendix B, below, The Java Timer class)
   - Get a connection to the database
   - Execute the hard-coded SQL query
   - Stop the timer
   - Print out the elapsed time for performing the query

```
package Performance;
    // Copyright:   Copyright (c) 2000
    // Author:      P. Carando
    // Description:  A simple data tier stimulator
    public Catalog1 = new
                    Performance.Timer ();
     }
     public void query () {
      try {
       startTimer ();
       java.sql.Connection conn =
          java.sql.DriverManager.getConnection("<db url>");
       java.sql.Statement stmt = conn.createStatement ();
       java.sql.ResultSet r = stmt.executeQuery ("<query>");
       stopTimer ();
       System.out.println ("w1: " +  aTimer.elapsedTime ());
      } catch (java.sql.SQLException ex) {
       stopTimer ();
       System.out.println ("w1 exception: " + ex);
      }
     }
```

```
     private void startTimer () {
       aTimer.reset ();  aTimer.start ();
     }
     private void stopTimer () { aTimer.stop (); }
     }
```

Appendix B.

A Java Timer Class

This simple Java timer class uses the System clock to measure elapsed time in milliseconds.

```
package Performance;
    // Title:       Timer
    // Copyright:   Copyright (c) 2000
    // Author:      P. Carando
    // Description:  A simple timer
    public class Timer {
     private long startTime, stopTime, elapsedTime;
     private boolean started, stopped;

     public Timer() { reset (); }
     public void reset () {
       started = false;   stopped = false; startTime = 0;
       stopTime  = 0;   elapsedTime = 0;
     }
     public void start () {
       startTime = System.currentTimeMillis ();
       started = true;
     }
     public void stop () {
       stopTime = System.currentTimeMillis ();
       stopped  = true;
       elapsedTime = Math.max (stopTime - startTime, 0);
     }
     public long elapsedTime () {
       if (!started) return 0;
       if (!stopped) {
         elapsedTime =
           System.currentTimeMillis () - startTime;
       }
       return elapsedTime;
     }
    }
```

References

[1] Freeman, G., *"The Right tools for the Job, Common Performance Issues and Solutions"*, Java Report, Volume 4, Number 7, July 1999, pp. 29—34.

[2] Kruchten, P., 1999. *The Rational Unified Process, An Introduction*, Addison-Wesley, Reading, Massachusetts.

[3] Long, F., *"Avoiding Garbage Collection in Java Applications"*, Java Report, Volume 5, Number 1, January 2000, pp. 28—34.

[4] McManus, A., *"Java: Memories Are Made of This"*, Java Report, Volume 3, Number 11, November 1998, pp. 39—48.

[5] Nylund, J., *"Memory Leaks in Java Programs "*, Java Report, Volume 1, Number 11, November 1999, pp. 22—31.

[6] Sosnoski, D., *"Java Performance Programming, Part 1: Smart Object Management Saves the Day"*, JavaWorld, November, 1999, www.javaworld.com.

[7] Sun Microsystems, *Java 2™ Platform Enterprise Edition Specification, v1.2*, December 17, 1999.