Last Modified: June 6, 2001

ECperf™ Specification

Version 1.0, Final Release

Sun Microsystems, Inc
Send comments to : ecperf_feedback@eng.sun.com

ECperf Specification ("Specification")
Version: 1.0
Status: FCS
Release: 06 June 2001

SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims based on your use of the Specification for any purposes other than those of internal evaluation, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND
U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DOD acquisitions).

REPORT
You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#90427/Form ID#011801)

Clause 1 - INTRODUCTION 7
1.1 Introduction 7
1.2 Business and Application Environment 8
Clause 2 - APPLICATION DESIGN 13
2.1 Model for Customer Domain 13
2.2 Model for Manufacturing Domain 14
2.3 Model for Supplier Domain 15
2.4 Model for Corporate Domain 16
2.5 Database Model 17
2.6 Corporate Database 17
2.7 Orders Database 20
2.8 Manufacturing Database 21
2.9 Supplier Database 23
Clause 3 - WORKLOAD DESCRIPTION 26
3.1 Definition of Terms 26
3.2 ECTransactions in the Customer Domain 27
3.3 ECtransactions in the Manufacturing Domain 30
3.4 ECtransactions in the Corporate Domain 33
3.5 ECtransactions in the Supplier Domain 34
3.6 Supplier Emulator 35
3.7 OrderEntry Application 36
3.8 Manufacturing Application 38
Clause 4 - SCALING AND RUN RULES 40
4.1 Definition Of Terms 40
4.2 Commercial Product Requirements 40
4.3 Scaling The Benchmark 41
4.4 Database Requirements 44
4.5 Bean Deployment Requirements 44
4.6 OrderEntry Driver Requirements 45
4.7 Manufacturing Driver Requirements 47
4.8 Computation of the ECperf Metric 47
4.9 Measurement Interval Requirements 48
4.10 Required Reporting 48
4.11 Transaction Property Requirements 50
4.12 Driver Rules 52
4.13 Supplier Emulator Rules 52
4.14 Benchmark Optimization Rules 52
Clause 5 - SUT CONFIGURATION 54
5.1 System Under Test (SUT) Requirements 54
5.2 Sample Configurations 54
Clause 6 - PRICING 57
6.1 Price/Performance Metric 57

6.2 Priced Components 57
6.3 Pricing Rules 58
Clause 7 - FULL DISCLOSURE 60
7.1 Definition of Terms 60
7.2 General Full Disclosure Requirements 60
7.3 Summary Statement 61
7.4 Clause 4 Scaling and Run Rules Related Items 61
7.5 Clause 5 SUT and Driver Related Items 62
7.6 Clause 6 Pricing Related Items 63
Clause 8 - HANDLING OF RESULTS 64
8.1 Results Submission 64
8.2 Violations in Published Results 65
8.3 Withdrawing a Result 65
8.4 Fair Use of ECperf Results 65
Appendix A - Bean Interface Definitions 67
Appendix B Sample Summary Statement 85
Appendix C UML Diagrams 89

Clause 1 - INTRODUCTION

1.1 Introduction

ECperf is a *Enterprise JavaBeans(EJB)™* benchmark meant to measure the scalability and performance of J2EE servers and containers.

Fundamentally, Enterprise JavaBeans is an infrastructure for building scalable, distributed applications which are implemented using component-oriented Object Transaction Middleware (OTM). As demand increases, Enterprise JavaBeans applications need to scale automatically. The ECperf workload, if it is to truly help improve the performance seen by real customers, should therefore possess the characteristics shown in Table 1.

TABLE 1. Golden Rules for ECperf Workload

Golden Rules	Characteristic	Description
Showcase Enterprise Components	Fully distributed	The business problem should necessitate use of worldwide services and data whose exact location is not known a priori.
	Redundant & Available Services	Services should be redundant and fully available with the runtime mapping unknown to the application developer.
	Middleware Focus	Strive to stress the middle-tier (whether this is logical or physical), rather than the client tier or the database server tier.
	Scalable business domain	As the size of the modeled business grows, the services, data, users, and number of geographically distributed sites should also grow.
	Universality	An enterprise bean should be capable of being deployed on any vendor's H/W and O/S, as long as a compliant container is present.
Command Credibility	Real-world	The performance workload should have the characteristics of real-world systems.
	Complexity	The workload should capture intra-, extra-, and inter-company business processes.
	Openness	The workload should be implementable atop all popular full- function EJB application servers.
	Neutrality	No workload features are selected with the goal of making any particular product look good (or bad). Fidelity to real-world business requirements is paramount.

TABLE 1. Golden Rules for ECperf Workload

Golden Rules	Characteristic	Description
Conform To Programmer Craft	RAD/IDE application development	The workload should be developed as if by a customer using reusable components intended for standardized deployment.
	Scoped development skills	The implementation should assume Fortune 500 developers who understand business logic, not systems programming.
Pattern Spec to Best Industry Practice	Simplicity	The workload should be easy to understand, straightforward to implement, and run in reasonable time with excellent repeatability.
	Industry-standard Benchmark	The workload should take into account the price of the system it is run on.

GUI and presentation are not the focus of this workload, nor are aspects of everyday DBMS scalability (e.g., database I/O, concurrency, memory management, etc.). These are adequately stressed by other standard workloads such as *TPC-C, TPC-D, TPC-W,* etc. ECperf stresses the ability of EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc.

Comment: While separated from the main text for readability, comments are a part of the standard and are enforced. However, the summary statements included as Appendix B, are provided only as examples and are specifically not part of this standard.

NOTE*: A notation such as this describes features in the specification that are not part of the current standard. By leaving the wording in the document, it is easier to expand the workload in future versions.*

1.2 Business and Application Environment

For reasons of interest, scope, and familiarity, we make use of *manufacturing, supply chain management, and order/inventory* as the "storyline" of the business problem. This is a meaty, industrial-strength distributed problem. It is heavyweight, mission-critical, worldwide, 7x24, and necessitates use of a powerful, scalable infrastructure. It's one that many Fortune 500 companies are interested in. Most importantly, it requires use of interesting *Enterprise JavaBeans* services, including:

- Transactional components
- Distributed Transactions
- Messaging and asynchronous task management
- Multiple company service providers with multi-site servers
- Interfaces to, and use of, legacy applications
- Secure transmissions
- Role-based authentication

- Object persistence

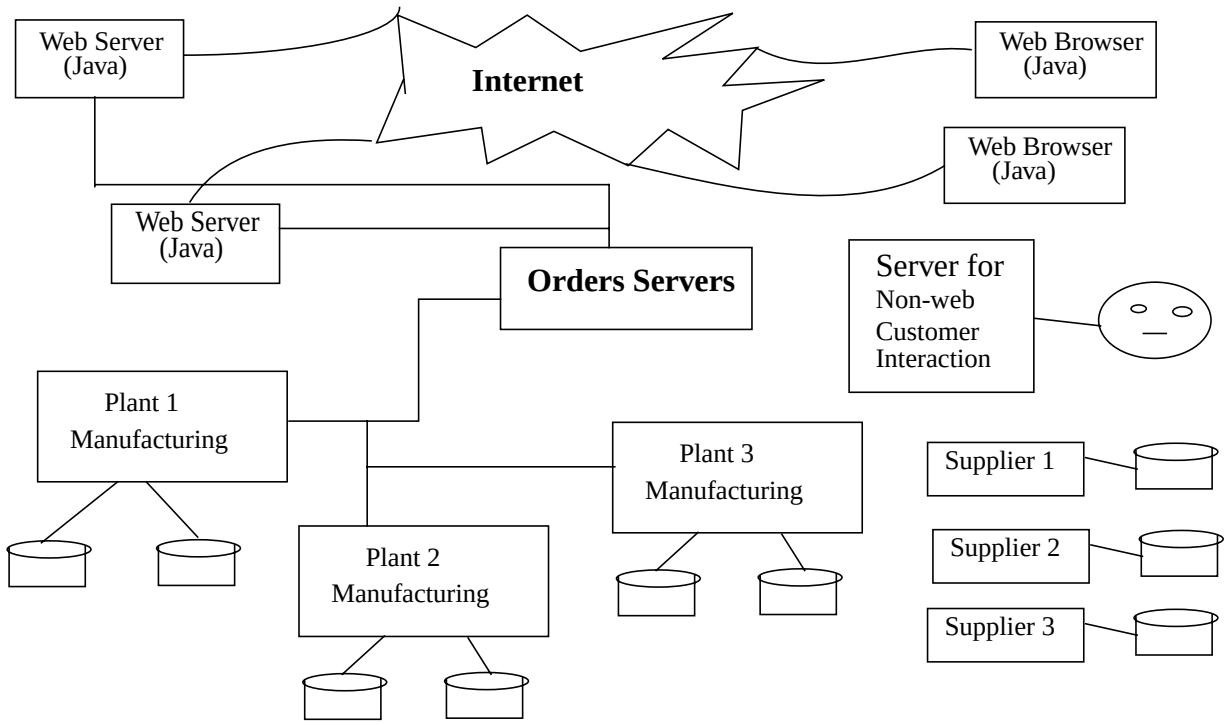


FIGURE 1: EJB World Wide Distributed Business

Businesses aggressively use the web to permit customers direct specification of product configuration, ordering, and status checking. The businesses strive to fully automate manufacturing, inventory, supplier chain management, and customer billing. Figure 1 depicts the essence of what such a business looks like.

Using “Just in Time” manufacturing concepts, they seek maximum efficiency and minimum inventories. Their customer and business processes run concurrently, are transactional, secure, worldwide, 24x7, and fully distributed. Companies like these want their application developers to focus on the business logic, not the complexities and subtleties of scalable transactions, messaging, directory/naming, security, authentication, or the mapping of services onto specific H/W. These businesses prefer to develop using a RAD/IDE infrastructure based on extensible, scalable, distributed components.

Figure 2 depicts the same business as Figure 1, but shows logical domains. The term **Domain** refers to a logical entity that describes a distinct business sphere of operations.

The four ECperf domains modelled are : Manufacturing, Supplier & Service Provider, Customer, and Corporate. In a company of the size being modelled, it is assumed that each domain has separate databases and applications. Most likely, they are implemented on separate computing hardware also. There are producer-consumer relationships between domains in the company and to outside suppliers and customers as well. For historical reasons, each domain may have distinct entity ID's (i.e., the customer ID used in the customer domain may be different from the ID used for that same customer in the Supplier domains). It is for this reason that global customer or supplier databases are likely to exist in the Corporate domain, with accompanying applications.

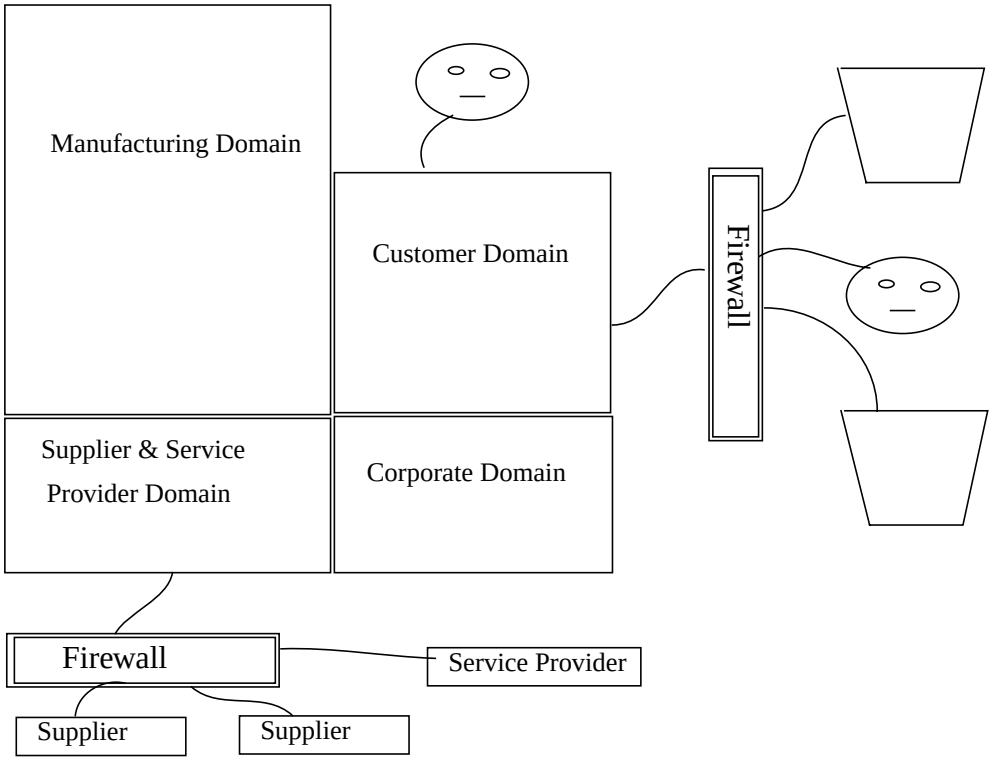


FIGURE 2: Multi-Domain Worldwide Business

Distributed databases and applications provide the foundation for the ECperf workload. Customers (including distributors, the government, and OEM's) contact the business through any number of methods, including directly through the web. Person-to-person contacts, such as over the telephone or through a salesperson, map into this web access scheme also, as the customer service representative or salesperson themselves use the web interface. All of the worldwide offices and plants make frequent access to data held in the other offices or plants, and must, at times, compare/collate/verify their data against that held worldwide. The company also interacts with completely separate supplier companies. Each supplier has its own independent set of computing resources.

The Enterprise business is a successful one. It grows via any number of different means. These include simple growth as well as diversification and corporate mergers. The first method, growth, means simply that the company gains more customers and sells more products. Its factories grow bigger, and it has more of them. The business needs a variety of suppliers; they grow also.

The company can also grow via diversification or through mergers. In both these cases, the company, now a conglomerate, may take on entirely new lines of business. It sets up new divisions, each with its possibly distinct set of customers and suppliers. The union of a new division's customers, manufacturing plants, and suppliers is called a **Production Group (PG)**. Figure 3 shows the state of the company after it has gone through a number of stages of growth. As the number of PG's increase, the sizes of the existing PG's grow larger and the newest one is the smallest. A single Corporate Domain acts as a coordination point between the various PG's. This represents a non-linear scaling model as found in the real world.

NOTE: *Release 1.0 is restricted to a single Production Group. See Clause 4 for detailed scaling information within this PG.*

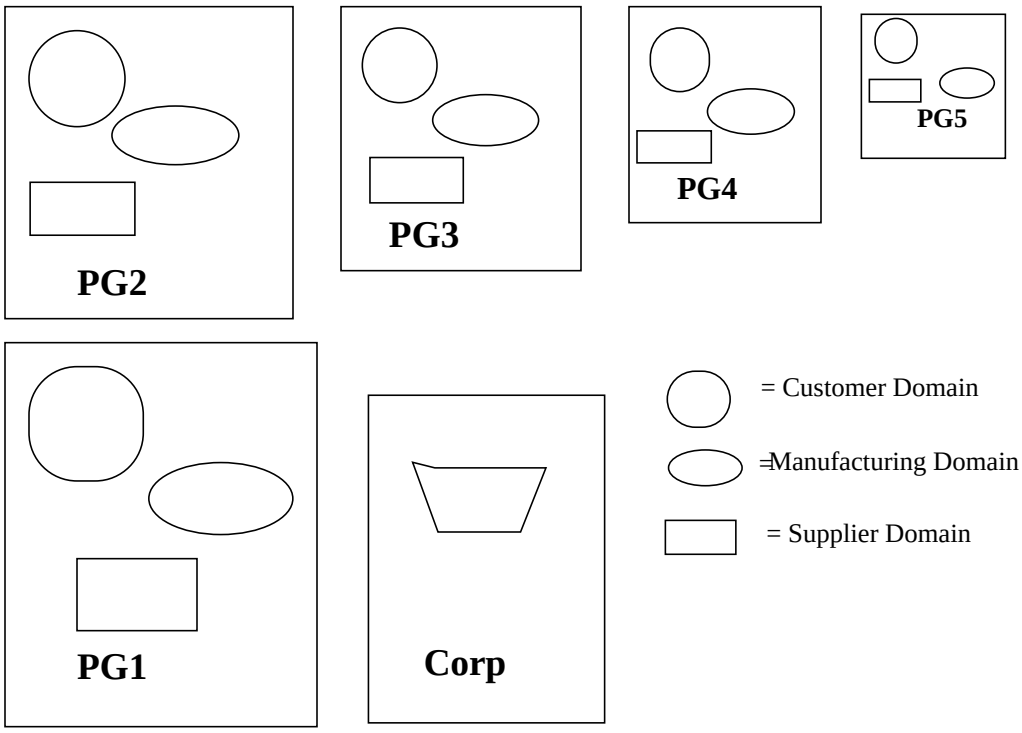


FIGURE 3: Scaling the Distributed Worldwide Business

The Enterprise business is a worldwide operation. Its manufacturing, assembly, or distribution plants are scattered among many cities. Each location has multiple computers, each of which might

be an application server, database server, multiplexer of client connections, or whatever. End users may have desktops, webtops, etc. It is not surprising that there are many different computers doing many different types of things for this business. Some computers provide just a single service (contain a single, vital database, for example), whereas others consolidate many services. Some computers may be configured in a Highly Available (H/A) configuration, so that services can continue if a node fails. Cost factors might dictate, on the other hand, that not all of the business' computers can have failover. Instead, the business may configure redundant services mapped to independent computers, leaving it to the runtime environment to find a provider for that business service.

The ECperf business consists of customers, plants (manufacturing, assembly, or distribution), and suppliers. The business is complex, distributed, and makes use of many independent computers arranged in a variety of topologies. It suggests a large number of interesting ways to stress application servers and middleware.

Clause 2 - APPLICATION DESIGN

2.1 Model for Customer Domain

Work in the Customer Domain is OLTP in nature. An OrderEntry application runs in this domain whose functionality includes adding new orders, changing an existing order and retrieving status of a particular order or all orders of a particular customer.

Orders are placed by individual customers as well as by distributors. The difference between the two is in the quantity of items ordered. Approximately 57% of work on the system is related to orders from distributors (i.e contain large number of items), 43% is from individual customers. Orders arrive from existing customers as well as new customers. In either case, a credit check is performed on the customer by sending a request to the Corporate domain. Various discounts are applied to the order depending on whether the customer is a distributor, repeat or first-time customer, etc.

Existing orders may be changed. The quantities of ordered items may be changed or the order may be cancelled in its entirety. Orders that have already shipped or have entered the shipping process cannot be cancelled.

A customer or salesperson can view the status of a particular order. A customer (typically a distributor) can also check the status of all his outstanding orders.

2.1.1 Bean Definitions

An UML Diagram for the Customer Domain is shown in Appendix C.2.

2.1.1.1 OrderSes

See Appendix A for a program listing of the OrderSes interface. It uses the various entity beans (OrderEnt, OrderLineEnt, ItemEnt) to implement its methods. The OrderEntry application uses the methods of the OrderSes interface and never accesses the entity beans directly.

2.1.1.2 OrderEnt

This entity bean manages access to the Orders table.

2.1.1.3 OrderLineEnt

This entity bean manages access to the OrderLine table and is called from within the OrderEnt bean. The application does not directly access this bean as it is a dependant object.

Comment: According to the EJB 1.1 specification, dependant objects are best implemented as separate Java classes (not enterprise beans). However, since the purpose of ECperf is to stress EJB performance, the decision was made to implement OrderLine as an entity bean.

2.1.1.4 ItemEnt

This entity bean manages access to the Item table and is called from within the OrderEnt bean. The application does not directly access this bean.

2.1.1.5 OrderCustomerSes

This stateless session bean is responsible for adding and validating customers within the Customer Domain. It uses the entity bean OrderCustomerEnt to accomplish its tasks.

2.1.1.6 OrderCustomerEnt

This entity bean manages access to the Customer table in the Orders database. It provides methods to add and validate customers and to compute a customer's discount.

2.1.1.7 CartSes

This stateful session bean simulates a shopping cart that a customer would use before placing an order. After a client adds items to the cart, the contents of the cart are bought using the OrderEnt bean.

2.2 Model for Manufacturing Domain

This domain models the activity of production lines in a manufacturing plant. There are two types of production lines: *Planned* lines and *Largeorder* lines. The planned lines run on schedule and produce a pre-defined number of widgets. On the other hand, the largeorder lines run only when a large order is received from a customer such as a distributor.

Manufacturing begins when a *Workorder* enters the system. Each workorder is for a specific quantity of a particular type of widget. The planned line workorders are typically created as a result of a forecasting application. The largeorder line workorders are generated as a result of customer orders. When a workorder is created, the Bill of Materials (BOM) for the corresponding type of widget is retrieved and the required parts are taken out of inventory. As the widgets move through the assembly line, the workorder status is updated to reflect progress. Once a workorder is complete, it is marked as complete and inventory updated.

As inventory of parts gets depleted, suppliers need to be located and purchase orders need to be sent out. This is done by contacting the Supplier Domain.

2.2.1 Bean Definitions

An UML Diagram for the Manufacturing Domain is shown in Appendix C.3 to C.5.

2.2.1.1 WorkOrderSes

Each instance of this stateless session bean manages one work order and uses the entity bean WorkOrderEnt, and the session bean BuyerSes in the Supplier Domain (see Clause 2.5) to accomplish its tasks.

2.2.1.2 LargeOrderSes

This stateless session bean handles the creation and searching of Large Orders. This bean uses the entity bean LargeOrderEnt to accomplish its tasks.

2.2.1.3 ReceiveSes

This stateless session bean is created by the ReceiverSes bean in the Supplier Domain (see clause 2.3.1.2) and is responsible for adding components to the inventory. This bean uses the entity bean ComponentEnt to accomplish its tasks.

2.2.1.4 PartEnt

This entity bean is the super class for all entity beans that require access to the component table in the manufacturing database. It is responsible for performing all operations on the Parts table.

2.2.1.5 AssemblyEnt

This entity bean is responsible for managing assemblies. An assembly is represented by a row in the Parts table and is made up of several components.

2.2.1.6 WorkOrderEnt

This entity bean manages access to the WorkOrder table in the manufacturing database.

2.2.1.7 LargeOrderEnt

This entity bean manages the custom orders (called large orders) that arise from the Customer Domain.

2.2.1.8 ComponentEnt

This entity bean manages access to the components and assemblies in the Parts table.

2.2.1.9 InventoryEnt

This entity bean manages access to the Inventory table in the manufacturing database. It provides methods to add and remove inventory.

2.2.1.10 BOMEnt

This entity bean manages access to the BOM table in the manufacturing database. It provides methods to retrieve information about the parts that are required to create a manufactured component.

2.3 Model for Supplier Domain

This domain is responsible for interactions with suppliers. The Supplier Domain decides which supplier to choose based on the parts that need to be ordered, the time in which they are required and the price quoted by suppliers. The company sends a purchase order to the selected supplier. The purchase order will include the quantity of various parts being purchased, the site it must be delivered to and the date by which delivery must happen. When parts are received from the supplier, the Supplier Domain sends a message to the Manufacturing Domain to update inventory.

2.3.1 Beans in Supplier Domain

An UML Diagram for the Supplier Domain is shown in Appendix C.6.

2.3.1.1 BuyerSes

This stateful session bean issues Purchase Orders based on requests it receives from the Manufacturing domain and uses the entity beans ComponentEnt, SupplierEnt, and POEnt to complete its tasks.

2.3.1.2 ReceiverSes

Each instance of this stateless session bean manages the delivery of components related to one Purchase Order. This bean uses the following entity beans to complete its tasks: POEnt, and ComponentEnt.

2.3.1.3 SupplierEnt

This entity bean manages access to the Suppliers table in the supplier database.

2.3.1.4 SupplierComponentEnt

This entity bean manages access to the SupplierComponent table in the supplier database. This bean is called from within SupplierEnt.

2.3.1.5 POEnt

This entity bean manages access to the PurchaseOrder table in the supplier database. It is also responsible for creation of the PO Lines that are in the purchase order.

2.3.1.6 POLineEnt

This entity bean manages access to the PurchaseOrderLine table in the supplier database.This bean is called from within POEnt.

2.3.1.7 SComponentEnt

This entity bean manages access to the Component table in the supplier database.

2.4 Model for Corporate Domain

This domain manages the global list of customers, parts and suppliers. Credit information, including credit limits, about all customers is kept solely in a database in the Corporate Domain. This is to provide maximal security and privacy.

For each new order, the Customer Domain requests a credit worthiness check to the Corporate Domain. Customer discounts are also computed in the Corporate Domain for each new order or whenever an order is changed. The discount computation is based on rules to determine which category the customer belongs to.

2.4.1 Beans in Corporate Domain

The UML Diagram for the Corporate Domain beans is shown in Appendix C.1.

2.4.1.1 CustomerEnt

This entity enterprise bean is responsible for keeping track of customer information in the Corporate database. See Appendix A for a program listing of the Customer interface. Client applications never make calls to this bean directly.

2.4.1.2 DiscountEnt

This entity bean manages the information on customer discounts.

2.4.1.3 RuleEnt

This entity bean manages the information about rules for customer discounts. The rules for the various categories of discounts are expressed as Java expressions and saved in the database.

2.5 Database Model

The components of the ECperf database are defined to consist of 4 separate and individual databases (See Clause 4.3.2 for how these databases can be combined). A database exists in each of the domains and consists of multiple tables. The benchmark implementation assumes a SQL database accessible via JDBC. This is because it is more likely that enterprises will move their applications to the component model first, leaving their data in legacy databases.

NOTE*: Only RDBMS that support JDBC are allowed in this version of the specification.*

2.6 Corporate Database

The Corporate database exists in the Corporate domain and has the master copy of customers, suppliers and parts. Some of this information will be duplicated in other databases. Assume that a nightly (or weekly) batch program will extract data from the Corporate database, massage it as necessary, and refresh the corresponding tables in the other databases.

2.6.1 Customer Table

The customer table contains information on all customers of the company. A part of this table is replicated in other domains. Confidential information such as credit rating/limit, account balance etc. is kept only in the Corporate database. Other domains must query this table to obtain the information.

Field name	Field Definition	Comment
C_ID	integer	customer id (Primary Key)
C_FIRST	varchar(16)	first name (unused if c_last is company)
C_LAST	varchar(16)	last name
C_STREET1	varchar(20)	street address, line 1
C_STREET2	varchar(20)	street address, line 2
C_CITY	varchar(20)	city
C_STATE	char(2)	state
C_COUNTRY	varchar(10)	country
C_ZIP	varchar(9)	zipcode
C_PHONE	varchar(16)	phone
C_CONTACT	varchar(25)	Contact person

Field name	Field Definition	Comment
C_SINCE	date	customer since
C_BALANCE	numeric(9,2)	account balance
C_CREDIT	char(2)	Credit rating ‘BC’ or ‘GC’
C_CREDIT_LIMIT	numeric(9,2)	credit limit
C_YTD_PAYMENT	numeric(9,2)	year to date payment made

2.6.2 Parts Table

There is a single global parts table that identifies all widgets of the company. These include widgets that are manufactured (i.e assemblies) and components that are purchased.

Field Name	Field Description	Comments
P_ID	char(15)	Encoded part number (Primary key)
P_NAME	varchar(10)	Part name
P_DESC	varchar(100)	Description
P_REV	varchar(6)	Revision number
P_UNIT	varchar(10)	Unit of measure for this part
P_COST	numeric(9,2)	Item cost
P_PRICE	numeric(9,2)	Item price
P_PLANNER	integer	Planner code - person who plans the part
P_TYPE	integer	Product code
P_IND	integer	Manufactured or Purchased part
P_LOMARK	integer	low water mark for inventory
P_HIMARK	integer	high water mark for inventory

The attribute, P_TYPE indicates what type of product this is (for ex: desktop/server/component). If P_IND specifies Manufactured, it should have a corresponding entry in the BOM table. Only Manufactured parts are part of the ITEM table in the Orders database.

2.6.3 Supplier Table

The supplier table contains a list of all suppliers for the company.

Field Name	Field Description	Comments
SUPP_ID	integer	supplier id (Primary key)
SUPP_NAME	varchar(16)	Supplier name
SUPP_STREET1	varchar(20)	street address, line 1
SUPP_STREET2	varchar(20)	street address, line 2
SUPP_CITY	varchar(20)	city
SUPP_STATE	char(2)	state

Field Name	Field Description	Comments
SUPP_COUNTRY	varchar(10)	country
SUPP_ZIP	varchar(9)	zipcode
SUPP_PHONE	varchar(16)	phone
SUPP_CONTACT	varchar(25)	Contact person

2.6.4 Site Table

The site table contains addresses of various sites of the company. These include manufacturing sites (locations where suppliers ship widgets to) and warehouses (locations from where the company ships widgets from).

Field name	Field Description	Comments
SITE_ID	integer	Primary key
SITE_NAME	varchar(16)	Supplier name
SITE_STREET1	varchar(20)	street address, line 1
SITE_STREET2	varchar(20)	street address, line 2
SITE_CITY	varchar(20)	city
SITE_STATE	char(2)	state
SITE_COUNTRY	varchar(10)	country
SITE_ZIP	varchar(9)	zipcode

2.6.5 Rule Table

The Rule table contains all the customer discount rules, one row per rule. Multiple types of rules are possible in the schema (identified by R_ID), but only one is used - namely, discount.

Field name	Field Description	Comments
R_ID	varchar(20)	Type of Rule (Primary Key)
R_TEXT	long varchar	Text of rule, can be multiple lines

2.6.6 Discount Table

This table holds the discounts for the various categories of customers. When a customer falls into a particular category (based on the rules in the Rule table), his discount level is looked up in the Discount table.

Field name	Field Description	Comments
D_ID	varchar(64)	Category of Customer (Primary Key)
D_PERCENT	integer	Percentage discount

2.7 Orders Database

The Orders database exists in the Customer Domain. It handles sales orders and runs the **OrderEntry** application. New orders can be created, existing orders changed, and status of orders retrieved.

2.7.1 Customer Table

Field Name	Field Definition	Comments
C_ID	integer	customer id (Primary Key)
C_FIRST	varchar(16)	first name (unused if C_LAST is company)
C_LAST	varchar(16)	last name
C_STREET1	varchar(20)	ship-to street address, line 1
C_STREET2	varchar(20)	ship-to street address, line 2
C_CITY	varchar(20)	ship-to city
C_STATE	char(2)	ship-to state
C_COUNTRY	varchar(10)	ship-to country
C_ZIP	varchar(9)	ship-to zipcode
C_PHONE	varchar(16)	phone
C_CONTACT	varchar(25)	Contact person
C_SINCE	date	customer since

2.7.2 Orders Table

Field Name	Field Description	Comment
O_ID	integer	Order ID (Primary Key)
O_C_ID	integer	Customer who placed this order
O_OL_CNT	integer	Number of order lines
O_TOTAL	numeric(9,2)	Order total
O_DISCOUNT	numeric(4,2)	Customer’s discount for this order
O_ENTRY_DATE	timestamp	Order entry date and time
O_SHIP_DATE	date	Order ship date
O_STATUS	integer	Status of complete order

2.7.3 OrderLine Table

Field Name	Field Description	Comment
OL_ID	integer	Order line id (Primary Key)
OL_O_ID	integer	Order id (Primary Key)
OL_I_ID	char(15)	Item id
OL_QTY	integer	Item Quantity
OL_STATUS	integer	Status: OUTSTANDING, SHIPPED
OL_SHIP_DATE	date	Order line ship date

2.7.4 Item Table

Field Name	Field Description	Comment
I_ID	char(15)	Item id (Primary key)
I_PRICE	numeric(9,2)	Item price
I_NAME	varchar(20)	Name
I_DESC	varchar(100)	Description
I_DISCOUNT	float	Discount for this item

2.8 Manufacturing Database

The Manufacturing database exists in the Manufacturing domain and handles the manufacturing of assemblies, BOMs, inventory and management of the shop floor. A **customer** is any person or organization that orders items. Customers order **widgets** - these are the products manufactured by the company (for simplicity, assume that the company is not a reseller). All manufactured widgets are **assemblies** and must have an entry in the BOM table. Assemblies are comprised of **components**. The list of all assemblies and components make up the global **parts** table. A single P_ID uniquely identifies every component that the company handles.

2.8.1 Parts Table

This table is derived from the Parts table in the Corporate Domain.

Field Name	Field Description	Comments
P_ID	char(15)	Encoded part number (Primary key)
P_NAME	varchar(10)	Part name
P_DESC	varchar(100)	Description
P_REV	varchar(6)	Revision number
P_PLANNER	integer	Planner code - person who plans the part
P_TYPE	integer	Product code

Field Name	Field Description	Comments
P_IND	integer	Manufactured or Purchased part
P_LOMARK	integer	low water mark for inventory
P_HIMARK	integer	high water mark for inventory

2.8.2 BOM Table

The BOM table holds the bill of materials for the various widgets produced by the company. Each widget (assembly) is comprised of multiple components. For simplicity, sub-assemblies are not dealt with (i.e the components in an assembly cannot in turn be assemblies).

Field name	Field Description	Comment
B_COMP_ID	char(15)	Component (Primary Key)
B_ASSEMBLY_ID	char(15)	Assembly to which this belongs (Primary Key)
B_LINE_NO	integer	Line number in BOM (Primary Key)
B_QTY	integer	Quantity/assembly
B_ENG_CHANGE	varchar(10)	Engineering change reference
B_OPS	integer	Op # - which step in the process this is used
B_OPS_DESC	varchar(100)	Operation description

There will be multiple rows for each assembly in this table that lists all its components.

2.8.3 WorkOrder Table

The manufacturing operations are managed by the use of WorkOrders. Workorders indicate what needs to be manufactured and are used to track progress through the assembly line.

Field name	Field Description	Comment
WO_NUMBER	integer	Work Order number (Primary key)
WO_O_ID	integer	Sales Order id if this is for a custom order
WO_OL_ID	integer	Orderline id in sales order
WO_STATUS	integer	Current status
WO_ASSEMBLY_ID	char(15)	Assembly being manufactured
WO_ORIG_QTY	integer	Original qty
WO_COMP_QTY	integer	Completed qty
WO_DUE_DATE	date	Date when the order is due
WO_START_DATE	timestamp	Date & time at which work began

The system creates a work order and assigns it a number. The work order may be for a batch assembly or for a single sales order item. If the latter, the fields WO_O_ID and WO_OL_ID will identify the particular order line this work order is for. WO_STATUS monitors progress of this work order and will contain values such as OPEN, STARTED, CANCELLED, COMPLETED, ARCHIVED.

2.8.4 LargeOrder Table

This table is a temporary repository for large custom orders that are received in the Customer Domain. The Driver will create work orders for each entry in this table and then delete it.

Field Name	Field Description	Comments
LO_ID	integer	Large order id
LO_O_ID	integer	Sales order id
LO_OL_ID	integer	Order line number of Sales order
LO_ASSEMBLY_ID	char(15)	Part being ordered
LO_QTY	integer	Quantity being ordered
LO_DUE_DATE	date	Date on which order is due

2.8.5 Inventory Table

The inventory table contains data about all parts - finished assemblies and components.

Field Name	Field Description	Comment
IN_P_ID	char(15)	Part number (Primary Key)
IN_QTY	integer	Amount in inventory
IN_ORDERED	integer	Qty ordered
IN_LOCATION	char(20)	Warehouse/bin
IN_ACC_CODE	integer	Finance code - is part usable?
IN_ACT_DATE	date	Date of last activity

2.9 Supplier Database

The Supplier database exists in the Supplier Domain and handles interaction with suppliers. Purchase orders for parts are issued to suppliers from this domain and received inventory transferred to manufacturing.

2.9.1 Supplier Table

This table is identical to the Supplier table in the Corporate database. See Clause 2.6.3

2.9.2 Site Table

This table is identical to the Site table in the Corporate database. See Clause 2.6.4.

2.9.3 Component Table

This table contains information on all the components that are purchased from outside suppliers. It's schema is derived from the Parts table in the Corporate database.

Field Name	Field Description	Comments
COMP_ID	char(15)	Encoded part number (Primary key)
COMP_NAME	varchar(10)	Part name
COMP_DESC	varchar(100)	Description
COMP_UNIT	varchar(10)	Unit of measure for this part
COMP_COST	numeric(9,2)	Current best cost for this part
QTY_ON_ORDER	integer	Keeps track of outstanding POs
QTY_DEMANDED	integer	Qty requested by Mfg
LEAD_TIME	integer	Lead time for delivery
CONTAINER_SIZE	integer	Amount to order

2.9.4 PurchaseOrder Table

The company issues Purchase orders to its suppliers. Each PO will contain multiple lines (one for each part).

Field Name	Field Description	Comment
PO_NUMBER	integer	Primary key
PO_SUPP_ID	integer	Supplier for this PO
PO_SITE_ID	integer	Site to ship this PO to

2.9.5 PurchaseOrderLine Table

Field Name	Field Description	Comment
POL_NUMBER	integer	Purchase orderline number (Primary Key)
POL_PO_ID	integer	Purchase order to which this belongs (Primary Key)
POL_P_ID	char(15)	Part number
POL_QTY	integer	Quantity ordered
POL_BALANCE	numeric(9,2)	Balance outstanding
POL_DELDATE	date	Delivery date requested
POL_MESSAGE	varchar(100)	

Comment: POL_P_ID is the part number. In reality, it should be the suppliers part number. To do this, a mapping between the part numbers and the suppliers is required (a complicated scenario, since it is desirable to have multiple suppliers for each part). The supplier part number is usually part of the BOM but to keep things simple, a single part number is used throughout.

2.9.6 SupplierComponents Table

This table maintains a list of suppliers for each component that is purchased. Whenever a supplier responds to a bid request, information in this table is updated with the latest pricing and availability information.

Field Name	Field Description	Comment
SC_P_ID	char(15)	Component
SC_SUPP_ID	integer	Supplier for this component
SC_PRICE	numeric(9, 2)	Supplier's price for this component
SC_DISCOUNT	float	Percentage discount
SC_QTY	integer	Quantity for discount
SC_DEL_DATE	integer	Delivery in days (from date of ordering)

Clause 3 - WORKLOAD DESCRIPTION

3.1 Definition of Terms

3.1.1 The term **Test Sponsor** refers to the organization(s) that are publishing a benchmark result. This is usually the primary hardware or software platform vendor.

3.1.2 The term **Driver** or **application** refers to code that drives the benchmark. The Driver implements the run rules described in Clause 4, keeps track of various statistics and reports the final metric. See Clause 4.12 for more details on the Driver.

3.1.3 The term **EJB Container** (or **Container** for short**)** refers to the entity that controls the lifecycle of the enterprise beans of the ECperf workload. Refer to the EJB 1.1 Specifications for more details.

3.1.4 The term **ECtransaction** refers to a remote method call on an Enterprise Java Bean.

3.1.5 The term **business transaction** refers to a unit of work initiated by the Driver and may involve one or more ECtransactions.

3.1.6 The term **database transaction** as used is this specification refers to a unit of work on the database with full ACID properties as described in Clause 4.11. A database transaction is initiated by the Container or an enterprise bean as part of a transaction.

3.1.7 The term **[x .. y]** represents a closed range of values starting with x and ending with y.

3.1.8 The term **randomly selected within [x .. y**] means independently selected at random and uniformly distributed between x and y, inclusively, with a mean of (x+y)/2, and with the same number of digits of precision as shown. For example, [0.01 .. 100.00] has 10,000 unique values, whereas [1 ..100] has only 100 unique values.

3.1.9 The term **non-uniform random function (NURand)**, is used in this specification to refer to the method used for generating customer ids, and part numbers. This method generates an independently selected and non-uniformly distributed random number over the specified range of values [x .. y], and is specified as follows :

NURand(A, x, y) = ((random(0, A) | random(x, y)) % (y - x + 1)) + x

where:

expr1 | expr2 stands for the bitwise logical OR operation between expr1 and expr2

expr1 % expr2 stands for expr1 modulo expr2

random(x, y) stands for randomly selected within [x .. y]

A is a constant chosen according to the size of the range [x .. y]

NURand is defined in the TPC-W benchmark specification which is copyrighted by the TPC.

3.1.10 The term **Measurement Interval** is used in this specification to refer to a steady state period during the execution of the benchmark for which the test sponsor is reporting a performance metric (See Clause 4.9 for detailed requirements).

3.1.11 The term **Response Time** is used in this specification to refer to the time elapsed from the first byte sent by the Driver to request a business transaction until the last byte received by the Driver to complete that business transaction (See Clauses 4.6.2 and 4.7.1 for detailed requirements).

3.1.12 The term **Injection Rate** is used in this specification to refer to the rate at which business transaction requests from the OrderEntry Application in the Customer Domain are injected into the SUT.

3.1.13 The term **Delay Time** is used in this specification to refer to the time elapsed from the last byte received by the Driver to complete a business transaction until the first byte sent by the Driver to request the next business transaction. The Delay Time is a function of the response time and the injection rate. For a required injection rate, the delay time will be smaller for larger response times.

3.1.14 The term **Cycle Time** is used in this specification to refer to the time elapsed from the first byte sent by the Driver to request a business transaction until the first byte sent by the Driver to request the next business transaction. The Cycle Time is the sum of the Response Time and Delay Time.

3.2 ECTransactions in the Customer Domain

The primary application that runs in the customer domain is OrderEntry. This workload does not concern itself with user interactions that are typical in an OrderEntry application. For example, searching for item ids, getting item descriptions, obtaining and verifying customer information etc. are not dealt with. The OrderEntry Driver implements the business transactions in this domain by making method calls to the CartSes, OrdersSes and OrderCustomerSes beans. See Clause 3.7 for a description of the OrderEntry application. The ECtransactions are listed below:

3.2.1 NewOrder

This ECtransaction will enter a new order for a customer having a certain number of order-lines.This ECtransaction is implemented by the newOrder method of the OrderSes bean. It gets the price of all items ordered, taking into account the item discount and calculate the order total. It then computes the customer's discount for this order and calls the Corporate domain to check whether the customer has sufficient credit to cover his purchase. If the customer does not have enough credit, it returns an error. Otherwise, it enters a row in the orders table, and as many rows in the OrderLine table, as the number of items. If the order is a custom order, it will trigger a LargeOrder to be created in the Manufacturing domain.

```
/**
     * newOrder: Enter an order for a customer
     * @param customerId   - Id of customer
     * @param quantities
     * @return int - Id of order
     * @exception DataIntegrityException - if customer/items don't exist
     * @exception InsufficientCreditException - if customer credit check fails
```

```
     * @exception CreateException - if creation of order fails
     * @exception RemoteException  - if there is a system failure
     */
    public int newOrder(int customerId, ItemQuantity[] quantities)
        throws InsufficientCreditException, DataIntegrityException,
               RemoteException, CreateException;
```

3.2.2 ChangeOrder

This ECtransaction will make changes to an existing undelivered order. This ECtransaction is implemented by the changeOrder method of the OrderSes bean. The OrderLine rows are modified appropriately. The quantity fields are updated and the new order total computed. Customer discount and credit are re-evaluated for this new order total. If the quantity is 0, the OrderLine is deleted. If a new itemId is passed, a new order line is added.

```
/**
     * changeOrder: Changes an existing customer order
     * @param orderId   - Id of order being changed
     * @param quantities - itemId, qty pairs of orderlines being changed
     * @exception InsufficientCreditException - if customer credit check fails
     * @exception RemoteException  - if there is a system failure
     */
    public void changeOrder(int orderId, ItemQuantity[] quantities)
        throws InsufficientCreditException, RemoteException;
```

3.2.3 OrderStatus

This ECtransaction gets the status of a particular order and all its orderlines. This ECtransaction is implemented by the getOrderStatus method of the OrderSes bean. It finds the customer id and order ship date for the order that corresponds to orderId. For all order lines that belong to this order, it retrieves the item id, quantity of items ordered and orderline ship date.

```
/**
     * getOrderStatus: Retrieves status of an order
     * @param orderId   - Id of order
     * @return OrderStatus object
     *
     * @throws DataIntegrityException
     * @exception RemoteException  - if there is a system failure
     */
    public OrderStatus getOrderStatus(int orderId)
        throws DataIntegrityException, RemoteException;
```

3.2.4 CustomerStatus

This ECtransaction gets the status of all outstanding orders for a particular customer. This ECtransaction is implemented by the getCustStatus method of the OrderSes bean. For each order that has not yet been delivered for this customer, it retrieves the order id, count of orderlines and order ship date. For all order lines in this order, it retrieves the item id and quantity.

```
/**
     * Get status of all orders of a Customer
     *
     * @param customerId      int customer id
     * @return          Array of CustomerStatus objects (one for each order)
     *
     * @throws DataIntegrityException - if customer doesn't exist
     * @exception RemoteException  - if there is a system failure
     */
    public CustomerStatus[] getCustomerStatus(int customerId)
        throws DataIntegrityException, RemoteException;
}
```

3.2.5 cancelOrder

This ECtransaction cancels an order by deleting the order row as well as its associated orderline rows.

```
/**
     * cancelOrder: Cancel an existing customer order
     * @param orderId   - Id of order being changed
     * @exception RemoteException  - if there is a system failure
     */
    public void cancelOrder(int orderId) throws RemoteException;
```

3.2.6 AddCustomer

This ECtransaction adds a new customer to the Customer domain. It calls the AddCustomer transaction in the Corp domain to add the customer to the Corporate database as well.

```
/**
     * This method adds a new customer with the specified info
     *
     * @param info - all fields of OrderCustomerEnt
     * @return id - Customer id of newly created customer
     */
    public int addCustomer(CustomerInfo info)
        throws InvalidInfoException, DataIntegrityException, RemoteException;
```

3.2.7 ValidateCustomer

This ECtransaction validates an existing customer by checking if the customer's information has properly formed fields. The primary purpose of this ECtransaction is to simulate some business logic.

```
/**
```

```
     * This method checks for the existence of a customer with
     * the specified id.
     */
    public void validateCustomer(int id)
        throws DataIntegrityException, RemoteException;
```

3.2.8 AddToCart

This ECtransaction adds an item to the Cart using the add method of the CartSes bean.

```
/**
     * Method add - Add items to the cart
     * @param item_qty It has item id and quantity to be added.
     * @exception RemoteException If there is a system failure.
     */
    public void add(ItemQuantity item_qty) throws RemoteException;
```

3.2.9 BuyCart

This ECtransaction buys the contents of the Cart by calling the buy method of the CartSes bean. This method in turn calls newOrder on the OrderSes bean.

```
/**
     * Method buy - Buy the contents of the cart. This will call newOrder method
 *      of OrderSes bean.
     *
     * @return int - Order Id of the new order created.
     * @exception RemoteException If there is a system failure.
     * @exception CreateException If creation of newOrder fails.
     * @exception InsufficientCreditException If newOrder fails due to bad credit
     */
   public int buy() throws RemoteException, CreateException, InsufficientCreditEx-
ception;
```

3.3 ECtransactions in the Manufacturing Domain

The Manufacturing domain is responsible for creating and executing workorders and managing the assembly lines.

3.3.1 ScheduleWorkOrder

This ECtransaction is called from the Manufacturing Application (see Clause 3.8) that simulates the normal manufacturing activity of planned lines or the transaction is called for a single sales order. It creates a row in the workorder table, identifies the components that make up this assembly in the BOM and assigns the required parts from inventory. When this ECtransaction is called for a single sales order (large order), the large order is deleted after the workorder is created.This transaction can cause calls into the Supplier Domain if it needs any parts with low inventory.

```
/**
     * Method to schedule a work order if called from the Mfg application
     * @param assemblyId    Assembly Id
```

```
     * @param qty           Original Qty
     * @param dueDate       Date when order is due
     *
     * @return id of workorder created
     * @exception RemoteException if there is a system failure
     */
    public Integer scheduleWorkOrder(String assemblyId, int qty, java.sql
        .Date dueDate) throws RemoteException;

/**
     * Method to schedule a work order if called from the Customer Domain
     * @param salesId       Sales order id
     * @param oLineId       Order Line ID
     * @param assemblyId    Assembly Id
     * @param qty           Original Qty
     * @param dueDate       Date when order is due
     *
     * @return id of workorder created
     * @exception RemoteException if there is a system failure
     */
    public Integer scheduleWorkOrder(
        int salesId, int oLineId, String assemblyId, int qty,
            java.sql.Date dueDate) throws RemoteException;
```

3.3.2 UpdateWorkOrder

As each station completes its operation, it will update the work order status. The workorder status is maintained using the State Pattern described by the UML Diagram in Appendix C.4 and C.5.

```
/**
     * Update status of a workorder
     *
     * @param wid                       WorkOrder ID
     * @exception               RemoteException if there is
     *                          a communications or systems failure
     */
    public void updateWorkOrder(Integer wid) throws RemoteException;
```

3.3.3 CompleteWorkOrder

This ECtransaction marks the work order as complete and transfers widgets to inventory. It gets the assembly id of this work order and updates the inventory for this assembly. It updates completed quantity and workorder status to COMPLETED. Note that workorders persist indefinitely.

```
/**
       * completeWorkOrder: workorder cancel.
       *
       * Transfer completed portion to inventory.
```

```
     * @param wid              WorkOrder ID
     * @return                 boolean false if failed to complete
     * @exception              RemoteException if there is
     *                         a communications or systems failure
     */
    public boolean completeWorkOrder(Integer wid) throws RemoteException;
```

3.3.4 CreateLargeOrder

This ECtransaction is initiated from within the Customer Domain whenever a large custom order has been entered. It will cause the creation of a work order based on the sales order. The work order will then go through the normal manufacturing flow until it is complete.

```
/**
     * Method createLargeOrder
     *
     *
     * @param orderId of sales order
     * @param oLineId within sales order
     * @param assemblyId - item in sales order
     * @param qty
     * @param dueDate
     * @return largeorder id
     * @throws RemoteException if system failure
     *
     */
    public Integer createLargeOrder(
        int orderId, int oLineId, String assemblyId, short qty,
            java.sql.Date dueDate) throws RemoteException;
```

3.3.5 FindLargeOrders

This ECtransaction is called by the Driver and it retrieves all large orders that currently exist in the LargeOrder table. Note that the large orders are deleted by scheduleWorkOrder after they are scheduled.

```
/**
     * Method findLargeOrders
     * Retrieve all currently existing large orders
     * @return Vector of LargeOrderInfo elements, each containing :
     * largeOrderId, orderId, orderLineId, assemblyId, quantity and dueDate
     *
     * @throws RemoteException if system failure
     *
     */
    public java.util.Vector findLargeOrders() throws RemoteException;
```

3.4 ECtransactions in the Corporate Domain

3.4.1 AddCustomer

Add a new customer. This ECtransaction is invoked from within the Customer domain to dynamically add new customers.

```
/**
    * Create a new customer.
    * @param CustomerInfo info containing :
    * firstName, lastName, address, contact, customerSince, balance and
    * yearToDatePayment.
    */
    CustomerEnt create(CustomerInfo info)
        throws java.rmi.RemoteException, javax.ejb.CreateException;
```

3.4.2 HasSufficientCredit

This ECtransaction checks whether a customer has a credit balance to cover his purchase. This is called from NewOrder and ChangeOrder ECtransactions in the Customer Domain when a customer enters/updates an order.

```
/**
     * This method is called from the OrderBean in the Customer domain
     * to check whether a customer has sufficient credt.
     * @param amount - amount of credit required
     * @return true if sufficient credit exists, else false
     */
    public boolean hasSufficientCredit(double amount) throws RemoteException;
```

3.4.3 GetPercentDiscount

This ECtransaction computes the customer's discount for a particular order, based on discount rules and customer history. This is called from the NewOrder and ChangeOrder ECtransactions in the Customer Domain.The bulk of the work is performed by a rule engine parser which parses the discount rules stored in the C_RULE table and determines which category the customer falls into. It then looks up the C_DISCOUNT table to get the percentage discount for this customer category.

```
/**
     * Method getPercentDiscount
     * @param amount
     * @return percentage discount
     *
     * @throws DataIntegrityException
     * @throws RemoteException
     *
     */
    public double getPercentDiscount(double amount)
```

```
        throws DataIntegrityException, RemoteException;
```

3.5 ECtransactions in the Supplier Domain

The Supplier Domain is responsible for the supply of components required to complete work orders. Orders for components are received from the Manufacturing Domain and sent to external suppliers. When orders are delivered inventory in Manufacturing is updated.

3.5.1 AddComponent

This ECtransaction will check if there are any outstanding Purchase Orders and add the component to a list of components to be ordered.

```
/**
     * add: Add components that are low in inventory to PO.
     * @param componentID  - Id of component that is being added to PO.
     * @param qtyRequired - number to be purchased.
     * @exception RemoteException - if there is a system failure.
     */
    public void add(String componentID, int qtyRequired)
        throws RemoteException;
```

3.5.2 Purchase

Once all components that are low in inventory have been added to a purchase order (AddComponent), ScheduleWorkOrder calls Purchase to buy the components. This ECtransaction will find all suppliers that can deliver all components within the required lead time and select the supplier that provides the best price and create the Purchase Order. It sends the Purchase Order to the Supplier emulator in XML format over a http connection.

```
/**
     * purchase: Issue the Purchase Order.
     * @exception ECperfException
     * @exception RemoteException - if there is a system failure.
     * @exception CreateException - if creation of Purchase Order fails.
     * @exception FinderException
     */
    public void purchase()
        throws RemoteException, FinderException, CreateException,
               ECperfException;
```

3.5.3 DeliverPO

This ECtransaction is called from the Supplier Domain servlet when it receives a message from the Supplier Emulator indicating that a purchase order has been delivered. It finds all the Purchase Order Lines and updates the POL_DELDATE field. This transaction will update inventory in Manufacturing, and, update QTY_ON_ORDER and QTY_DEMANDED in the COMPONENT table

```
    /**
     * deliverPO - indicate the part of a PO has been delivered.
     * @param del - contains information about the POLine that has been delivered.
```

```
 * @exception RemoteException - if there is a system failure.
 */
public void deliverPO(DeliveryInfo del) throws RemoteException;
```

3.6 Supplier Emulator

The Supplier Emulator is a Java Servlet that can run inside any java enabled web server. It is not part of the SUT, hence must run on a machine that is external to it. It communicates with the SUT by sending and receiving XML documents via a HTTP connection that conforms to the HTTP 1.1 protocol. The emulator provides the Supplier Domain with a way to emulate the process of sending and receiving orders to/from suppliers.

The supplier emulator accepts a purchase order from the BuyerSes bean in the Supplier Domain, processes the purchase order, and then delivers the order lines contained in the purchase order to the ReceiverSes Bean after sleeping for an amount of time based on the lead time of the component. This interaction between the Supplier Domain and the Emulator is shown in Figure 4.

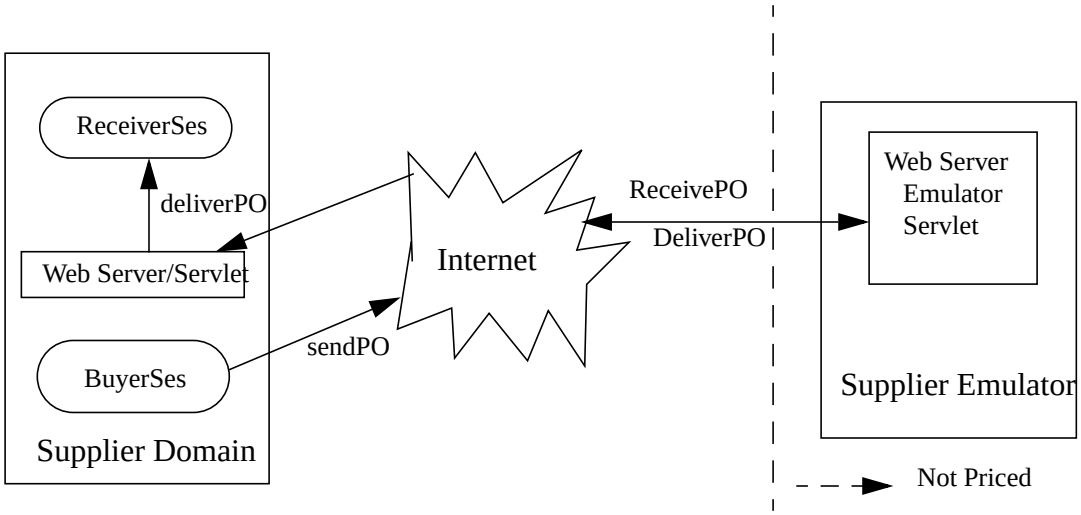


FIGURE 4: Supplier Emulator Interactions

3.6.1 Purchase Orders

Purchase orders are received from the Supplier Domain (see Clause 3.5.2) in XML format over a HTTP connection. Contained in the XML document is the details of the purchase order:

poNumber	id of the purchase order
siteId	id of the site that the purchase order should be delivered to
numLines	the number of purchase order lines that are in the purchase order

For each line in the purchase order, the following must be provided:

lineNumber	Line number in purchase order
Id	id of part
qty	qty to order
price	price of part from this supplier
leadTime	maximum lead time that the part must be delivered within

3.6.2 Parts Delivery

On receipt of a purchase order, the Emulator will parse the XML and each purchase order line is extracted. If the XML is in the correct format a positive response is returned over the HTTP connection, otherwise a negative response will be returned. The HTTP connection is then closed.

The purchase order lines are then processed one at a time in ascending order based on lead-time. For each line, an XML document is generated that contains:

poNumber	id of purchase order that the line is part of
poLine	line number in Purchase Order
pId	id of part being delivered
qty	quantity being delivered

The XML for the purchase order line is then transmitted via HTTP to a servlet in the Supplier Domain, which in turn calls the ReceiveSes Bean. Note that a servlet in the Supplier Domain is required as the communication between the Supplier Domain and the Emulator is over http.

3.7 OrderEntry Application

The OrderEntry Application is responsible for entering new orders, changing existing orders and retrieving order status.

Each new order is placed on behalf of a customer, whose Id is chosen using NURand (A, 1, *nCust*) where A is 255 and *nCust* is the number in the Customer Pool (See Clause 4.3.4). If the selected customer does not exist, a new customer is added using randomly generated data for the various fields in the O_Customer table. If the selected customerId is within the initial database population, the customerId is verified.

The next step is to determine the orderlines. The items for the orderlines are chosen using NURand (A, 1, NUM_ITEMS) where A is 63 and NUM_ITEMS is the number of items being manufactured by the company. The count of orderlines in the order is randomly selected within [1..5]. 90% of the time, the order is placed by a regular customer for whom the total number of widgets is randomly selected within [10..20]. This number is divided equally amongst all the orderlines. The average order size is thus 15 widgets, with each orderline having an average quantity of 5. 10% of the time, the order is assumed to be placed by a distributor (called Large Order). For these orders, the total number of widgets is randomly selected within [100..200], again dividing them equally amongst all the orderlines, the count of which is randomly selected within [1..5]. Thus, the overall average order size is 150 widgets for a large order, which are equally distributed amongst all the orderlines.

The average number of widgets for Large Orders is 10 times that of a regular order. Let's compute what this means for an Injection Rate of 10. On an average there will be 5 new Orders received per second (based on the mix requirement in 4.6.1). 4.5 of these orders have an average size of 15 and 0.5 are from a distributor with an average size of 150. The average widgets ordered per second is thus 67.5 (4.5 * 15) for a regular order and 75 (0.5 * 150) for a Large Order. Thus, 53% of the widgets ordered are from Large Orders. Generalizing this, for an Injection Rate of Ir, there will be 6.75*Ir widgets for regular orders and 7.5*Ir widgets for large orders, giving a total widget ordering rate of 14.25*Ir.

Finally, the Driver places the order. 50% of the time the order is placed by calling the newOrder method of the OrderSes bean. The other 50% of the time, the order is placed by first adding items to a Shopping Cart. The Driver chooses the Cart to use by using an array of 1000 Carts and selecting from this array thus :

```
cartId = -ln(x) * 100
where x = random number from a uniform distribution such that (0 < x <= 1)
if (cartId > 1000) cartId = 1000;
```

If cartId is not null (i.e the Cart already exists), the Driver deletes all the items in the Cart. It then adds all the order lines one by one. After all the items are added, the Cart's contents are bought. The Shopping Cart is then deleted 90% of the time. 10% of the time, the Cart is left intact and could be later re-used.

The Driver saves the orderId returned by all the successful newOrder transactions in a *newOrderIds* array for use by the changeOrder and getOrderStatus transactions.

For every changeOrder business transaction, the orderId is randomly selected within [1..NUM_ORDERS] plus the newOrderIds array, where NUM_ORDERS is the number of Orders in the initial database. The Driver first performs a getOrderStatus ECtransaction to find the status and orderline count *n*, in the order it would like to change. If it finds that the order has been shipped or no longer exists, or if the order has only 1 orderline, it will re-try the transaction on a different order. 90% of the time, the orderlines will be changed. The remaining 10% of the time, the order is deleted. For orderlines to be changed, the count of lines to be changed is randomly selected within [1..*n*]. For each of these order lines, the quantity is alternatively incremented or decremented. For an order that is to be deleted, the Driver takes care to do this only on one of newOrderIds i.e, an order that exists in the initial database population is not deleted. This preserves the cardinality of the tables for successive runs.

For the getOrderStatus business transaction, an orderId is randomly selected within [1..NUM_ORDERS] plus the newOrderIds array where NUM_ORDERS is the number of Orders in the initial database. The getOrderStatus ECtransaction is executed using the selected order id.

For the getCustStatus business transactions, the customerId of the customer is randomly selected within [1..NUMCUSTOMERS] where NUMCUSTOMERS is the number of customers in the initial database and the corresponding ECtransaction executed.

For all of the business transactions, the Driver keeps track of the average response times, targeted and actual cycle times, count of transactions, etc. and generates a report with the final metric and other statistics at the end of the benchmark run.

3.8 Manufacturing Application

The Manufacturing Application runs two types of assembly lines for each PG: Planned and LargeOrder. For simplicity, it is assumed that the work necessary to manufacturing distinct widgets is identical. The only real difference between the lines are how they are scheduled.

The manufacturing application keeps track of the number of workorders initiated, the number of widgets produced and the average time for the completion of each workorder.

3.8.1 Planned Lines

The Planned Lines run continuously and concurrently, producing, on average, 47% of all widgets. The rate at which widgets are manufactured must roughly equal the rate at which widgets are being ordered. The idea is Just In Time Manufacturing. The factory floors are composed of a grand total of *p* Planned Lines (see Clause 4.3.5 for the values of *p*), each with 3 *Stations*. A Station marks a distinct operation in the manufacturing flow, such as multi-step manufacture, test, packaging, etc.

The Manufacturing application spawns threads for each Planned Line. Each Planned Line generates workorders at an average rate of 0.2 per second (or 1 workorder in 5 seconds). Each workorder will produce on average 11.25 widgets of a particular type. The type is selected by using NURand (A, 1, NUM_ITEMS), where A is 63 and NUM_ITEMS is the number of items being manufactured by the company. 75% of the time, the quantity for the workorder is chosen as 11 and 25% of the time, the quantity is chosen as 12. This yields a rate of 11.25 widgets per workorder.

The first station will get the workorder id once it is generated. It will then wait for its designated sleep time and update the workorder status. The sleep time is a constant amount of .333 seconds. The workorder moves along to the next station and so on until it reaches the final station (each station is aware of where in the chain it stands). Thus the workorder will spend 1 second in all the 3 stations combined. At the final station, the workorder is marked complete, the items are transferred into inventory. The Response Time Tr from workorder start to finish is measured and the Delay Time Td is computed as:

```
Td = 5 - Tr
```

Since the Planned Line needs to produce 1 widget in 5seconds, Tr can be as high as 5seconds. If Tr is lower than 5(i.e Td > 0), the Planned Line will sleep for Td seconds before looping back to create the next workorder.

3.8.2 LargeOrder Lines

The LargeOrder Lines handle production above and beyond the rate at which the Planned Lines can produce a particular widget. Aggregated, activity on these lines originate from 10% of all orders received and represent 53% of all widgets manufactured.

The Customer Domain notifies the Manufacturing Domain, every time a Large Order is received. The LargeOrder line started from the manufacturing application periodically checks for the existence of Large orders and immediately schedules a new workorder for each.

Like a Planned Line, a Large Order Line has 3 Stations and the same average time spent per station. There are as many concurrent Large Order lines as incoming orders require.

Clause 4 - SCALING AND RUN RULES

4.1 Definition Of Terms

4.1.1 The term **Deployment Unit** refers to a Container or set of Containers in which the Beans from a particular domain are deployed.

4.1.2 The term **ECclient** refers to a thread or process that holds references to Enterprise Beans in a Deployment Unit. An ECclient does not necessarily map into a tcp connection to the Container.

4.1.3 The term **ECperf Reference Beans** refers to the implementation of the Enterprise Beans provided for the ECperf workload.

4.1.4 The term **ECperf Kit** refers to the complete kit provided for ECperf. This includes the ECperf Reference Beans, the Driver and load programs.

4.1.5 The term **BBops/min** is the ECperf metric and denotes the average number of successful **B**enchmark **B**usiness **OP**eration**S** per minute completed during the Measurement Interval. BBop/min is composed of the total number of business transactions completed in the Customer Domain, added to the total number of workorders completed in the Manufacturing Domain, normalized per minute.

4.1.6 The term **$/BBops** is used in this specification to refer to the total cost of the SUT (see Clause 7) divided by the number of BBops.

4.1.7 The term **Resource Manager** is used in this specification to the software product that manages a database and is the same as a Database Manager.

4.2 Commercial Product Requirements

4.2.1 The following functions must be implemented using commercially available and supported products :

- Operating System
- Web Server & Container (used in the Supplier Domain and Supplier Emulator)
- EJB Server & Container
- Java Virtual Machine
- Database Server

4.2.2 The SUT must provide all application components with a runtime environment that meets the requirements of the Java 2 Platform, Enterprise Edition, Version 1.2 (J2EE) specification during the benchmark run. The SUT must meet the J2EE compatibility requirements and must be branded

Java Compatible, Enterprise Edition.

Comment 1: The current version of the ECperf benchmark does not make use of some of the J2EE capabilities, for example :

- JMS
- JavaMail
- JavaBeans Activation Framework

Comment 2 : A new version of a J2EE Compatible Product must have passed the J2EE Compatibility Test Suite (CTS) by the Availability Date. See Clause 6.3.2 for availability requirements.

4.2.3 The version of the ECperf Kit used to run the benchmark must match the version of the ECperf specification (this document).

4.3 Scaling The Benchmark

The throughput of the ECperf benchmark is driven by the activity of the OrderEntry and Manufacturing applications. The throughput of both applications is directly related to the chosen Injection Rate. To increase the throughput, the Injection Rate needs to be increased. The benchmark also requires a number of rows to be populated in the various tables.

Comment: The intent of the scaling requirements is to maintain the ratio between the ECtransaction load presented to the SUT, the cardinality of the tables accessed by the ECtransactions, the Injection Rate and the number of ECclients generating the load.

4.3.1 Scaling Requirements

4.3.1.1 Database scaling is defined by the Orders Injection Rate Ir. The scaling is done as a step function in such a manner that it will not be required to re-load the database every time the value of Ir changes.

4.3.1.2 The cardinality of the Site and Supplier tables is fixed.

4.3.1.3 The cardinality of the customer related tables, namely, customer, orders, orderline, and workorder will increase as a step function C, which is defined as:

```
C = the smallest multiple of 10 >= Ir
```

For example, if Ir = 72, then C = 80.

4.3.1.4 The cardinality of the item related tables, namely, Parts, BOM, Inventory and Item will increase more slowly as a step function P, which is defined as:

```
P = smallest multiple of 100 >= Ir
```

For example, if Ir = 72, P = 100.

4.3.2 Database Scaling Rules

The following scaling requirements represent the initial configuration of the tables in the various domains, where C and P are defined in 4.3.1.3 and 4.3.1.4.

TABLE 2 :Database Scaling Rules

Domain	Table Name	Cardinality (in rows)	Comments
Corporate			
	C_Site	1	
	C_Supplier	10	
	C_Customer	75 * C	
	C_Rule	1	
	C_Discount	6	
	C_Parts	$(11 * P)_1$	P Assemblies + 10 * P Components
Orders			
	O_Customer	75 * C	NUM_CUSTOMERS
	O_Item	P	NUM_ITEMS
	O_Orders	75 * C	
	O_Orderline	$(225 * C)_1$	Avg. of 3 per order
Manufacturing			
	M_Parts	$(11 * P)_1$	
	M_BOM	$(10 * P)_1$	
	M_Workorder	P	
	M_Inventory	$(11 * P)_1$	
Supplier			
	S_Site	1	
	S_Supplier	10	
	S_Component	$(10 * P)_1$	Avg. of 10 components per assembly
	S_Supp_Component	$(100 * P)_1$	
	S_PurchaseOrder	$(.2 * P)_1$	2% of components
	S_PurchaseOrderLine	P_1	Avg. of 5 per purchase order

1. These sizes may vary depending on actual random numbers generated.

4.3.3 Scaling the Databases and Containers

4.3.3.1 To satisfy the requirements of a wide variety of customers, the ECperf benchmark can be run in either of two categories, namely, **Centralized** and **Distributed**. Results published on the

Centralized workload cannot be compared with results published on the Distributed workload.

4.3.3.2 In the **Centralized** version of the workload, all the 4 domains are allowed to be combined. This means that the benchmark implementor can choose to run a single Deployment Unit that accesses a single database that contains the tables of all the databases. However, a benchmark implementor is free to separate the domains into their Deployment Units and still run a single database. There are no requirements for XA 2-phase commits in the Centralized workload.

4.3.3.3 The **Distributed** version of ECPerf is intended to model application performance where the world-wide enterprise that ECperf models performs transactions across business domains employing heterogeneous resource managers. In this model, the workload requires separate Deployment Units and separate databases, as well as seperate DBMS instances in all the Domains. XA-compliant recoverable 2-phase commits are required in ECtransactions that span multiple domains. The configuration for this 2-phase commit is required to be done in a way that would support hetergenous systems.

Comment : Even though implementations are likely to use the same Resource Manager for all the Domains, the EJB Servers/Containers and Resource Managers cannot take advantage of the knowlege of homogeneous Resource Managers to optimize the 2-phase commits.

4.3.3.4 The SUT consists of one or more physical computing nodes, which number is freely chosen by the implementor (including SUT node count == 1), subject to the Pricing requirements of Clause 7. The entire number of required databases and EJB Containers can be mapped to SUT computing nodes as required, subject to the minimum count requirements listed above. The implementation must not, however, take special advantage of the co-location of databases and Containers, other than the inherent elimination of WAN/LAN traffic.

4.3.4 Scaling the OrderEntry Application

To stress the ability of the Container to handle concurrent sessions, the benchmark requires a minimum number of ECclients equal to 5 * Ir where Ir is the chosen Injection Rate. The number doesn't change over the course of a benchmark run, although to do so would be more representative of real-world Ecommerce.

For each new order, each ECclient takes on a distinct identity drawn from the cardinality of the *Customer Pool*, which is defined as nCust = 100 * C. For example, if Ir = 100, the database is initially populated with 7500 customers (NUM_CUSTOMERS), new orders will be placed on behalf of 10,000 customers.

4.3.5 Scaling the Manufacturing Application

The Manufacturing Application scales in a similar manner to the OrderEntry Application. Since, the goal is Just-in-time manufacturing, as the number of orders increase, a corresponding increase in the rate at which widgets are manufactured is required. This is achieved by increasing the number of Planned Lines *p* proportionally to I_r as

$p = 3 * I_r$

Since the arrival of large orders automatically determines the LargeOrder Lines, nothing special needs to be done about these.

4.4 Database Requirements

4.4.1 All tables must have the properly scaled number of rows as defined by the database population requirements (see Clause 4.3).

4.4.2 No database objects, e.g indexes, views etc. can be defined beyond those included in the reference schema scripts in schema/sql in the ECperf Kit (See 4.1.4). The datatypes can be modified provided they are semantically equivalent to the standard types specified in the scripts.

4.4.3 In any committed state, the primary key values must be unique within each table. For example, in the case of a horizontally partitioned table, primary key values of rows across all partitions must be unique.

4.4.4 The databases must be populated using the ECperf load program, provided as part of the ECperf kit. The load program uses standard SQL INSERT statements and loads all the tables via JDBC and should work unchanged across all DBMS. However, modifications are permitted for porting purposes. All such modifications made must be disclosed in the Full Disclosure Report.

4.5 Bean Deployment Requirements

4.5.1 The Test Sponsor must run the ECperf Reference Beans. The ECperf Reference Beans come in both CMP and BMP versions. The Sponsor can choose to deploy either CMP or BMP or a mix of both. See also Clause 4.2 for Container requirements.

4.5.2 The only changes allowed to the ECperf Reference Beans are in the classes that implement the bean-managed persistence (BMP). The only changes allowed to the BMP code are for porting similar to Clause 4.4.4. All code modifications must appear in the Full Disclosure Report, along with an explanation for the changes.

4.5.3 The deployment descriptors supplied with the ECperf Reference Beans must be used without any modifications.

Comment: Ideally, the benchmark should require the deployment of the same ejb-jar files in all benchmark implementations. However, some environment properties in the xmls are dynamically generated at build time. Further, due to product immaturities, the deployment descriptor contents are allowed to be combined into a single file, split into multiple ones etc. as long as their contents do not change.

4.5.4 Commit Option A, specified in the EJB 1.1 specification is not allowed. It is assumed that the database(s) could be modified by external applications.

4.5.5 Vendor-specific flags or parameters to avoid doing Store operations at the end of a transaction are

not allowed.

4.6 OrderEntry Driver Requirements

4.6.1 Transaction Mix Requirements

The OrderEntry Driver repeatedly performs business transactions in the Customer Domain. Business transactions are selected by the Driver based on the mix shown in Table 3. Since we want to test the transaction handling capabilities of EJB Containers, the mix above is update intensive. In the real-world, there may be more readers than writers.

The actual mix achieved in the benchmark must be within 5% of the targeted mix for each type of transaction. For example, the newOrder transactions can vary between 47.5% to 52.5% of the total mix. The Driver checks and reports on whether the mix requirement was met.

TABLE 3. Mix Requirements for the OrderEntry Application

Transaction Type	Percent Mix
newOrder	50%
getOrderStatus	20%
changeOrder	20%
getCustStatus	10%

4.6.2 Response Time Requirements

The OrderEntry Driver measures and records the Response Time of the different types of business transactions. Only successfully completed business transactions in the Measurement Interval are included. At least 90% of the business transactions of each type must have a Response Time of less than the constraint specified in Table 4 below. The average Response Time of each transaction type must not be greater than 0.1 seconds more than the 90% Response Time. This requirement ensures that all users will see reasonable response times. For example, if the 90% Response Time of newOrder transactions is 1 second, than the average cannot be greater than 1.1 seconds. The Driver checks and reports on whether the response time requirements were met.

TABLE 4. Response Time Requirements for the OrderEntry Application

Transaction Type	90% RT (in seconds)
newOrder	2
getOrderStatus	2
changeOrder	2
getCustStatus	2

4.6.3 Cycle Time Requirements

For each business transaction, the OrderEntry Driver selects cycle times from a negative exponential distribution, computed from the following equation, such that the maximum average injection rate chosen can be achieved as best possible.

$Tc = -\ln(x) / I_r$

where:

ln = natural log (base e)

x= random number with at least 31 bits of precision, from a uniform distribution such that ($0 < x <= 1$)

I_r= mean Injection Rate

The distribution is truncated at 5 times the mean. For each business transaction, the Driver measures the Response Time Tr and computes the Delay Time Td as Td = Tc - Tr. If Td > 0, the Driver will sleep for this time before beginning the next transaction. If the chosen cycle time Tc is smaller than Tr, then the actual cycle time (Ta) is larger than the chosen one. The average actual cycle time is allowed to deviate from the targetted one by 5%. The Driver checks and reports on whether the cycle time requirements were met.

4.6.4 Misc. Requirements

The table below shows the range of values allowed for various quantities in the OrderEntry application. The Driver will check and report on whether these requirements were met.

TABLE 5. Misc. OrderEntry Requirements

Quantity	Targetted Value	Min. Allowed	Max. Allowed
Widget Ordering Rate/min	855 * Ir	812.4 * Ir	897.6 * Ir
LargeOrder Widget Ordering Rate/min	450 * Ir	427.8 * Ir	472.8 * Ir
RegularOrder Widget Ordering Rate/min	405 * Ir	384.6 * Ir	425.4 * Ir
% Large Orders	10	9.5	10.5
% Orders thru Cart	50	47.5	52.5
% Orders failed credit check	10	9	10
$ ChgOrders that were delete	10	9.5	10.5

4.6.5 Performance Metric of the Customer Domain

The Metric for the Customer Domain is **Transactions/min**, composed of the total count of all business transaction types successfully completed during the measurement interval divided by the length of the measurement interval in minutes.

4.7 Manufacturing Driver Requirements

4.7.1 Response Time Requirements

The Manufacturing Driver measures and records the time taken for a Workorder to complete. Only successfully completed WorkOrders in the Measurement Interval are included. At least 90% of the WorkOrders must have a Response Time of less than 5 seconds. The average Response Time must not be greater than 0.1 seconds more than the 90% Response Time.

4.7.2 Misc. Requirements

The table below shows the range of values allowed for various quantities in the Manufacturing Application. The Manufacturing Driver will check and report on whether the run meets these requirements.

TABLE 6. Misc. Manufacturing Requirements

Quantity	Targetted Value	Min. Allowed	Max. Allowed
LargeOrderline Widget Rate/min	450 * Ir	405 * Ir	495 * Ir
Planned Line Widget Rate/min	405 * Ir	364.8 * Ir	445.8 * Ir

4.7.3 Performance Metric of the Manufacturing Domain

The metric for the Manufacturing Domain is **Workorders/min**, whether produced on the Planned lines or on the LargeOrder lines.

4.8 Computation of the ECperf Metric

4.8.1 The two primary metrics of the ECperf benchmark are Benchmark Business Operations Per Minute (BBops/min) and a price performance metric defined as $/BBops.

4.8.2 The overall metric for the ECperf benchmark is calculated by adding the metrics of the OrderEntry Application in the Customer Domain and the Manufacturing Application in the Manufacturing Domain as:

BBops/min = (Transactions + Workorders)/min

4.8.3 All reported BBops/min must be measured, rather than estimated, and expressed to exactly two decimal places, rounded to the hundredth place.

4.8.4 The performance metric must be reported as BBops/min@Std or BBops/min@Dist for the Centralized and Distributed Workloads respectively. See Clause 4.3.3.2 and Clause 4.3.3.3 for a description of these Workloads. For example, if a measurement yielded 123.45 BBops/min on the

Centralized workload, this must be reported as 123.45 BBops/min@Std.

4.9 Measurement Interval Requirements

The Orders and Manufacturing Application must be started simultaneously at the start of a benchmark run.

The Measurement Interval must be preceded by a ramp-up period of at least 10 minutes at the end of which a steady state throughput level must be reached. At the end of the Measurement Interval, the steady state throughput level must be maintained for at least 5 minutes, after which the run can terminate.

4.9.1 Steady State

The reported metric must be computed over a Measurement Interval during which the throughput level is in a steady state condition that represents the true sustainable performance of the SUT. Each Measurement Interval must be at least 30 minutes long and must be representative of an 8 hour run.

Comment: The intent is that any periodic fluctuations in the throughput or any cyclical activites, e.g JVM garbage collection, database checkpoints etc. be included as part of the Measurement Interval.

4.9.2 Reproducibility

To demonstrate the reproducibility of the steady state condition during the Measurement Interval, a minimum of one additional (and non-overlapping) Measurement Interval of the same duration as the reported Measurement Interval must be measured and its BBops/min must be greater than the reported BBops/min. This reproducibility run's metric is required to be within 5% of the reported BBops/min.

4.10 Required Reporting

4.10.1 The frequency distribution of the Response Times of all business transactions in the Customer and Manufacturing Domains, started and completed during the Measurement Interval must be reported in a graphical format. In each graph, the x-axis represents the Response Time and must range from 0 to five times the required 90th percentile Response Time (N). This 0 to 5N range must be divided into 100 equal length intervals. One additional interval will include the Response Time range from 5N to infinity. All 101 intervals must be reported. The y-axis represents the frequency of each type of business transaction at a given Response Time range with a granularity of at least 10 intervals. An example of such a graph is shown below.

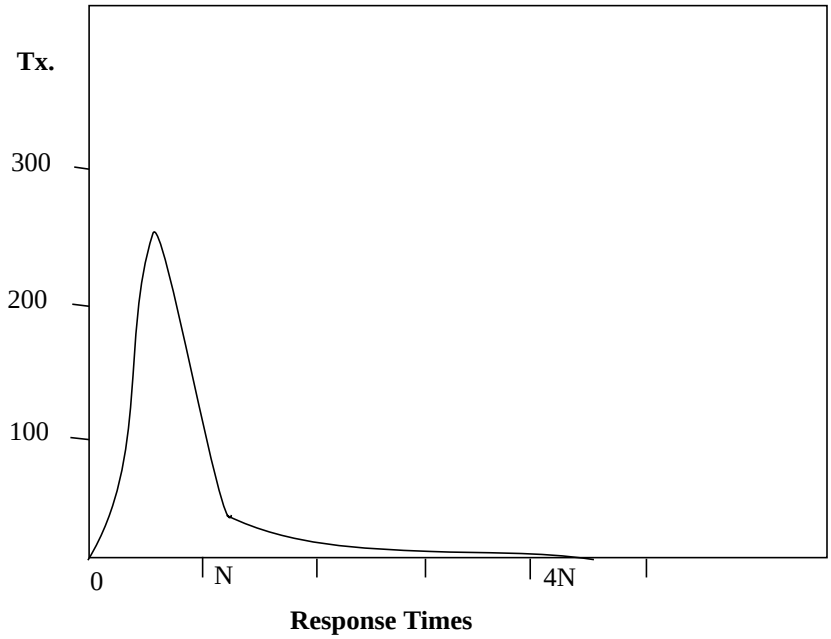


FIGURE 5: Sample Response Time Graph

4.10.2 A graph of the workorder throughput versus flashed time (i.e wall clock time) must be reported for the Manufacturing Application for the entire Test Run. The x-axis represents the elapsed time from the start of the run. The y-axis represents the throughput in business transactions. At least 60 different intervals must be used with a maximum interval size of 30 seconds. The opening and the closing of the Measurement Interval must also be reported. An example of such a graph is shown

below.

.

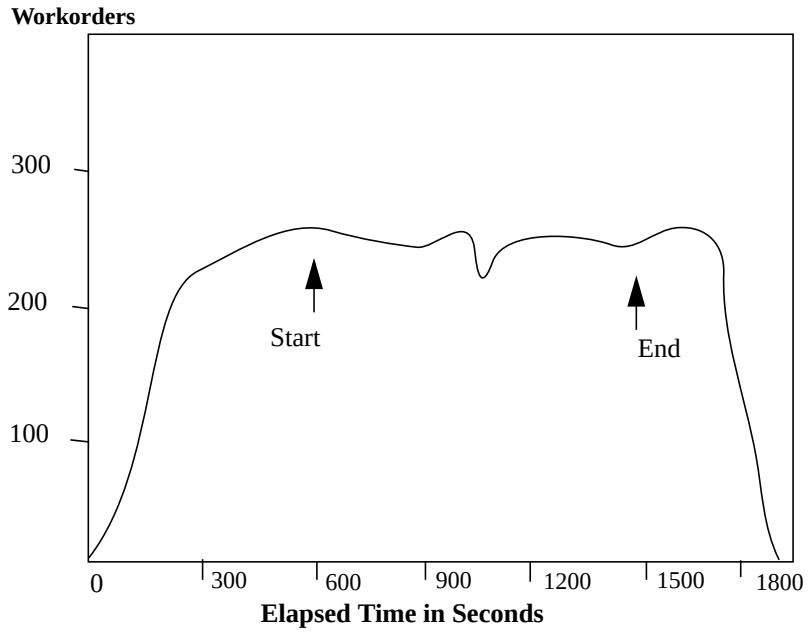


FIGURE 6: Sample Throughput Graph

4.11 Transaction Property Requirements

The Atomicity, Isolation and Durability properties of transaction processing systems must be supported by the system under test during the running of this benchmark.

4.11.1 Atomicity Requirements

The system under test must guarantee that database transactions are atomic; the system will either perform all individual operations on the data, or will assure that no partially-completed operations leave any effects on the data.

4.11.2 Atomicity Tests for the Centralized Workload

4.11.2.1 Atomicity Test 1

a. Choose a customer who has a bad credit by looking in the O_CUSTOMER table for a customer with the c_credit field equal to ‘BC’.

b. Modify the OrderEnt bean code to print the order id as soon as it generates it.

c. Enter a new order for this customer using the web client application, distributed as part of the ECperf kit. Note the order id printed by the bean code. The transaction should fail generating an InsufficientCreditException.

d. Retrieve the status of the noted order id in step c. The order should not exist.

e. Query the database table O_orderline for rows where ol_o_id match the order id printed in step c. There should be no rows returned.

4.11.2.2 Atomicity Test 2

a. Choose a customer with good credit by looking in the O_CUSTOMER table for a customer with the c_credit field equal to 'GC'.

b. Enter a new order for this customer using the web client application. The transaction should succeed. Note the order id returned.

c. Retrieve the status of the noted order id above. The order along with the orderlines entered in step b. should be displayed.

4.11.3 Atomicity Tests for the Distributed Workload

4.11.3.1 Atomicity Tests 1 and 2 above for the Centralized Workload must be performed.

4.11.3.2 Atomicity Test 3

a. Instrument the ejbCreate code in the OrderEnt bean in the Customer Domain to display the order id and orderLine ids of the newly entered order. Do the same for the LargeOrderEnt bean in the Manufacturing Domain. Change the LargeOrderEnt bean to add the following to the ejbStore method : `entityContext.setRollBackOnly();`

b. Enter a new order for any customer, ensuring that it is a largeorder. Note the values of the order id, orderLine ids and largeOrder id displayed.

c. The transaction should rollback. Verify that the rows in the O_orders, O_orderline and M_largeorder table with the ids retrieved in the above step do not exist.

4.11.4 Isolation Requirements

The various Isolation Levels are described in the J2SE documentation for java.sql.Connection. The EJB 1.1 specification leaves isolation to be specified at deployment time on a per resource basis. This means that all business methods of a bean share the same isolation level. Since multiple beans are typically involved in a transaction, and in order not to penalize the read-only transactions, it is required that all transactions have an isolation level of READ_COMMITTED or higher; i.e dirty reads are not allowed.

4.11.5 Durability Requirements

Transactions must be durable from any single point of failure on the SUT. In particular,distributed 2 Phase Commit transactions must be durable. There are no tests prescribed for durability, since this is typically a database property.

4.12 Driver Rules

4.12.1 The Driver is provided as part of the ECperf kit. Sponsors are required to use this Driver to run the ECperf benchmark.

4.12.2 The Driver communicates with the SUT using any standard protocol supported by the EJB Container, such as RMI/JRMP, RMI/IIOP etc.

4.12.3 The Driver must reside on system(s) that are not part of the SUT.

Comment: The intent of this clause is that the communication between the Driver and the SUT be accomplished over the network.

4.12.4 The Driver system(s) must use a single URL to establish communication with the Container in the case of the Centralized Workload and 4 URLS (one per Domain) in the case of the Distributed Workload.

4.12.5 EJB object stubs invoked by the Driver on the Driver system(s) are limited to data marshalling functions, load-balancing and failover capabilities. Pre-configured decisions, based on specific knowledge of ECperf and/or the benchmark configuration are dis-allowed.

4.12.6 The Driver system(s) may not perform any processing ordinarily performed by the SUT, as defined in Clause 5.1. This includes, but is not limited to :

- Executing part or all of the ECperf Beans
- Caching database or container specific data
- Communicating information to the SUT regarding upcoming transactions

4.13 Supplier Emulator Rules

4.13.1 The Supplier Emulator is provided as part of the ECperf Kit and can be deployed on any Web Server that supports Servlets 2.1.

4.13.2 The Supplier Emulator must reside on system(s) that are not part of the SUT. However, the Supplier Emulator may reside on one of the Driver systems.

Comment: The intent of this clause is that the communication between the Supplier Emulator and the SUT be accomplished over the network.

4.14 Benchmark Optimization Rules

Benchmark specific optimization is not allowed. Any optimization of either the configuration or products used on the SUT must improve performance for a larger class of workloads than that defined by this benchmark and must be supported and recommended by the provider.Optimizations must have the vendor’s endorsement and should be suitable for production use in an environment comparable to the one represented by the benchmark. Optimizations that take advantage of the

benchmark's specific features are forbidden. Examples of inappropriate optimization include, but are not limited to, taking advantage of the specific SQL code used, the sizes of various fields or tables, or the number of beans deployed in the benchmark.

Clause 5 - SUT CONFIGURATION

5.1 System Under Test (SUT) Requirements

5.1.1 The SUT comprises all components which are being tested. This includes network connections, Application Servers/Containers, Database Servers, etc.

The SUT consists of:

- The host system(s) (including hardware and software) required to support the Workload and databases.
- All network components (hardware and software) between host machines which are part of the SUT and all network interfaces to the SUT.
- Components which provide load balancing within the SUT.

Comment 1: The intention is to require any components which are required to form the physical TCP/IP connections (commonly known as the NIC, Network Interface Card) to be part of the SUT.

Comment 2: A basic configuration consisting of one or more switches between the Driver and the SUT is not considered part of the SUT.However, if any software/hardware is used to influence the flow of traffic beyond basic IP routing and switching, it is considered part of the SUT.

5.1.1.1 The SUT must have sufficient on-line disk storage to support any expanding system files and the durable database population resulting from executing the ECperf transaction mix for 8 hours at the reported BBops/min.

5.1.2 The SUT services remote method calls from the Driver and returns results generated by the ECperf Reference Beans which may involve information retrieval from a RDBMS. The database must be accessed only from the ECperf Reference Beans (or the Container acting on behalf of a bean), using JDBC.

5.1.3 The SUT must not perform any caching operations beyond those normally performed by the servers (EJB Containers, Database Servers etc.) which are being used.

Comment: The intention is to allow EJB Container and Database Server caching to work normally but not to allow the implementation to take advantage of the limited nature of the benchmark and to cache information which would normally be retrieved from the Servers.

5.2 Sample Configurations

5.2.1 Figure 7 shows an example layout of the Driver and SUT components for the Centralized Workload. In this example, the SUT consists of 3 Containers to handle the load. Multiplexing/Load balancing between the Containers is accomplished by a commercial product. All the Domains are combined into a single database.

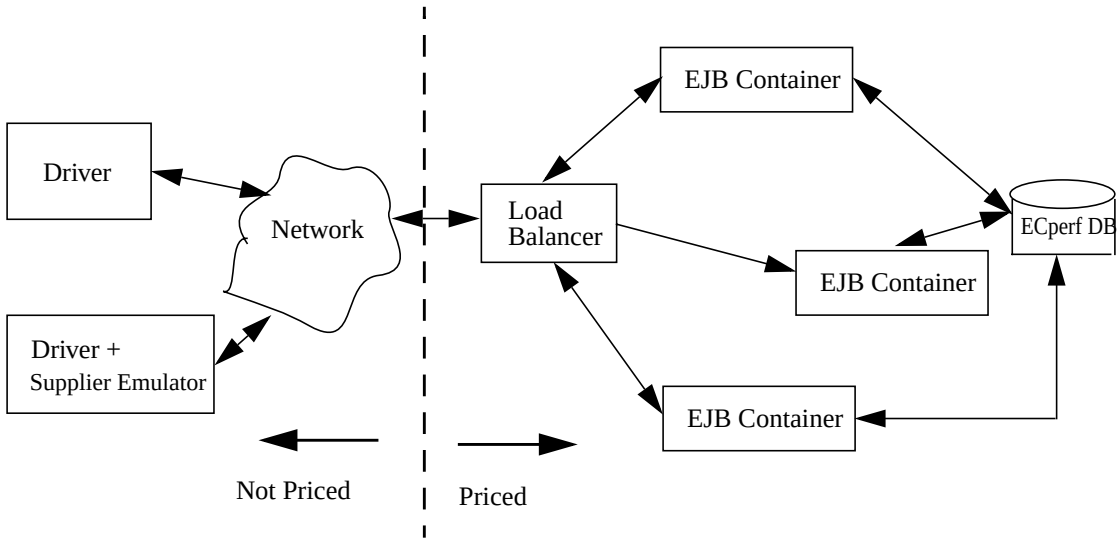


FIGURE 7: Example configuration for the Centralized Workload

5.2.2 Figure 8 shows an example layout of the Driver and SUT components for the Distributed Workload. In this layout, there is a single Container for each of the Domains that manages its own data-

base.

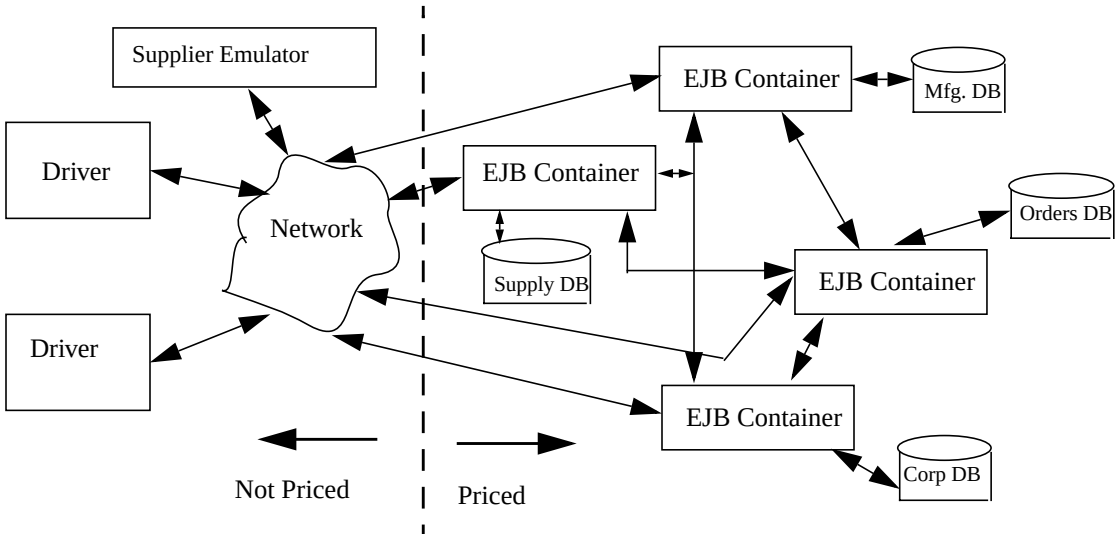


FIGURE 8: Example configuration for the Distributed Workload

Clause 6 - PRICING

6.1 Price/Performance Metric

6.1.1 In addition to the performance metric defined in Clause 4.8, ECperf includes a price/performance metric defined as price/BBops/min which is the total price of the SUT divided by the reported BBops/min.

6.1.2 The price/performance metric must be rounded up to the least significant digit of the currency unit. For example, if the total price is US$ 5,734,417 and the reported throughput is 105.12 BBops/min, then the price/performance is US$ 54,551/BBops.

6.2 Priced Components

6.2.1 SUT

6.2.1.1 The entire price of the SUT (see Clause 5.1) must be included, including all hardware, software and support for a 1 year period.

6.2.1.2 All hardware components priced must be new and not reconditioned or previously owned.

6.2.1.3 The number of users for ECperf is for each Domain and defined to be equal to the number of ECclients in the tested configuration. Any usage pricing for the above number of users should be based on the pricing policy of the company supplying the priced component.

6.2.2 Additional Components

6.2.2.1 Additional components such as operator consoles and backup devices must also be priced, if explicitly required for the operation, administration, or maintenance, of the SUT.

6.2.2.2 If software needs to be loaded from a particular device either during installation or for updates, the device must be priced.

6.2.3 Support

6.2.3.1 Hardware and software support must be priced for 7 days/week, 24 hours/day coverage, either on-site, or if available as standard offering, via a central support facility.

6.2.3.2 If a central support facility is priced, then all hardware and software required to connect to the central support must be installed and functional on the SUT during the measurement run and priced.

6.2.3.3 The response time for hardware support requests must not exceed 4 hours on any component whose

replacement is necessary for the SUT to return to the tested configuration.

6.2.3.4 Pricing of spares in lieu of the hardware support requirements is allowed if the part to be replaced can be identified as having failed by the customer within 4 hours. An additional 10% of the designated part, with a minimum of 2, must be priced. A support service for the spares which provides replacement on-site within 7 days must also be priced for the 1 year support period.

6.2.3.5 Software support requests must include problem acknowledgement within 4 hours. Software support must include at least 1 update distribution for both the software and its documentation over the 1 year period.

6.3 Pricing Rules

6.3.1 The intent of the pricing rules is to price the tested system at the full price a customer would pay. Assumptions of other purchases made by this customer are not allowed. This is a one time, stand-alone purchase.

6.3.2 All hardware and software used must be orderable by customers. For any product not already generally released, the Full Disclosure Report must include a committed general delivery date (see Clause 7.3.1). That date must not exceed 3 months beyond the Full Disclosure submittal date. All products used must be the proposed final versions and not prototypes. The generally released product performance must be within 2% of the published result.

Comment: The intent is to test products that customers will use, not prototypes. Beta versions of products can be used, provided General Availability (GA) is in 3 months.

6.3.3 All pricing used must be list pricing. No form of discounts are allowed.

6.3.4 It is permissible to use package pricing as long as this package is a standard offering.

6.3.5 All pricing should be in the currency unit which a customer in this country would use to pay for the system and reflect local retail pricing. Prices should be rounded up to the least significant digit of the currency unit being used. For example, all U.S pricing should be rounded up to the nearest dollar.

6.3.6 All pricing sources and effective date(s) of the prices must be disclosed.

6.3.7 All items supplied by a third party (i.e not a Test sponsor) must be explicity stated. Each third party supplier’s items and prices, must be listed separately.

6.3.8 Pricing shown in the Full Disclosure Report must reflect the level of detail a customer would see on an itemized billing. See Appendix B for a sample pricing disclosure.

6.3.9 Results can be re-published in another country with only pricing changes for which it is permitted to substitute local components from the original Full Disclosure providing the substituted products

are sold to the same product description or specifications.

Comment: The intent is to encourage local country pricing by allowing substitution of equipment for country specific reasons such as voltage, product numbering, industrial/safety, keyboard differences, etc., which do not affect performance.

Clause 7 - FULL DISCLOSURE

7.1 Definition of Terms

7.1.1 The term **Full Disclosure** refers to the information that must be provided when a benchmark result is reported.

7.1.2 The term **Full Disclosure Report** refers to the printable report in PDF or HTML forms that is part of a Full Disclosure.

7.1.3 The term **Full Disclosure Archive** refers to the soft-copy archive of files that is part of a Full Disclosure.

7.1.4 The term **Summary Statement** refers to the benchmark summary information that must be provided when a benchmark result is reported. The Summary Statement is also part of the Full Disclosure Report.

7.2 General Full Disclosure Requirements

7.2.1 A Full Disclosure is required in order for results to be considered compliant with the ECperf benchmark specification.

Comment 1: The intent of this disclosure is for anyone to be able to replicate the results of this benchmark given the appropriate documentation and software.

Comment 2: In the sections below, when there is no specific reference to where the disclosure must occur, it must occur in the Full Disclosure Report. Disclosures in the Archive or Summay Statement are explicitly called out.

7.2.2 The order and titles of sections in the Full Disclosure Report must correspond with the order and titles of sections from the ECperf standard specification (i.e., this document). The intent is to make it as easy as possible for readers to compare and contrast material in different Full Disclosure reports.

7.2.3 A Summary Statement must be included near the beginning of the Full Disclosure report (see Clause 7.3).

7.2.4 A statement identifying the benchmark sponsor(s) and other participating companies must be provided.

7.2.5 Diagrams of both measured and priced configurations must be provided, accompanied by a description of the differences. This includes, but is not limited to:

- Number and type of processors.
- Size of allocated memory, and any specific mapping/partitioning of memory unique to the test.

- Number and type of disk units (and controllers, if applicable).
- Number of LAN (e.g., Ethernet) connections, including routers, etc., that were physically used in the test.
- Type and the run-time execution location of software components (e.g., EJB Server/Containers, DBMS, client processes, software load balancers, etc.).

7.3 Summary Statement

7.3.1 The Summary Statement is a high-level view of the ECperf benchmark configuration and run results. An example of the Summary Statement is presented in Appendix B. The Summary Statement must include all of the information contained in this example in the same format for the benchmark being reported.

7.3.2 The Driver summary reports must appear as part of the Summary Statement. These include the ECperf.summary, Orders.summary and Mfg.summary files.

7.3.3 The Audit.report file generated by the Driver for run validation must appear as part of the Summary Statement.

7.4 Clause 4 Scaling and Run Rules Related Items

7.4.1 All commercially available software products used must be identified. Settings must be provided for all customer-tunable parameters and options which have been changed from the defaults found in actual products, including but not limited to:

- Operating system options.
- J2EE Server options.
- Web Container options used for the Supplier Domain and the Emulator.
- EJB Container options.
- Database options.

Comment 1: This requirement can be satisfied by providing a full list of all parameters and options.

7.4.2 For a new version of a J2EE Compatible Product, the date by which it is expected to have passed the J2EE Compatibility Test Suite (CTS) should be indicated.

7.4.3 The Orders Injection Rate used to load the database(s) must be disclosed.

7.4.4 The Full Disclosure Archive must include all table definition statements and all other statements used to set-up the database.

7.4.5 If the Load Programs in the ECperf kit were modified (see Clause 4.4.4), all such modifications

must be disclosed and the modified programs must be included in the Full Disclosure Archive.

7.4.6 All scripts/programs used to create any logical volumes for the database devices must be included as part of the Full Disclosure Archive. The distribution of tables and logs across all media must be explicitly depicted.

7.4.7 The type of persistence, whether CMP, BMP or mixed mode used by the EJB Containers must be disclosed. If mixed mode is used, the list of beans deployed using CMP and BMP must be enumerated.

7.4.8 If the ECperf Reference Beans were modified (see Clause 4.1.3), a statement describing the modifications must appear in the Full Disclosure Report and the modified code must be included in the Full Disclosure Archive.

7.4.9 All Deployment Descriptors used must be included in the Full Disclosure Archive.

7.4.10 The BBops/min from the reproducibility run must be disclosed (see Clause 4.9.2). The entire output directory from the reproducibility run must be included in the Full Disclosure Archive in a directory named RepeatRun.

7.4.11 The frequency distribution of response times for all the transactions must be graphed (see Clause 4.10.1).

7.4.12 A graph of the workorder throughput versus elapsed time must be reported (see Clause 4.10.2).

7.4.13 The scripts/programs used to run the ACID tests and their outputs must be included in the Full Disclosure Archive.

7.5 Clause 5 SUT and Driver Related Items

7.5.1 If any software/hardware is used to influence the flow of network traffic beyond basic IP routing and switching, the additional software/hardware and settings must be disclosed. See Clause 5.1.1.

7.5.2 The input parameters to the Driver must be disclosed by including the *config/run.properties* file and *bin/driver.sh* script used to run the benchmark in the Full Disclosure Archive. If the Launcher package was modified, its source must be included in the Full Disclosure Archive.

7.5.3 The bandwidth of the network(s) used in the tested/priced configuration must be disclosed.

7.5.4 The protocol used by the Driver to communicate with the SUT (e.g RMI/IIOP) must be disclosed.

7.5.5 If the Driver system(s) perform any load-balancing functions as defined in Clause 4.12.5, the details of these functions must be disclosed.

7.5.6 Thenumber and types of client systems used, along with the number and types of processors, mem-

ory and network configuration must be disclosed.

7.6 Clause 6 Pricing Related Items

7.6.1 A detailed list of hardware and software used in the priced system must be reported. Each separately orderable item must have vendor part number, description, and release/revision level, and either general availability status or committed delivery date. If package-pricing is used, vendor part number of the package and a description uniquely identifying each of the components of the package must be disclosed. Pricing source(s) and effective date(s) of price(s) must also be reported.

7.6.2 The total price of the entire configuration must be reported, including: hardware, software, and maintenance charges. Separate component pricing is recommended.

7.6.3 The committed delivery date for general availability (availability date) of products used in the price calculations must be reported. When the priced system includes products with different availability dates, the reported availability date for the priced system must be the date at which all components are committed to be available.

7.6.4 For any usage pricing, the sponsor must disclose:

- Usage level at which the component was priced.
- A statement of the company policy allowing such pricing.

Comment: Usage pricing may include, but is not limited to, the operating system, EJB server and database server software.

7.6.5 System pricing should include subtotals for the following components: Server Hardware, Server Software, and Network Components used. An example of the standard pricing sheet is shown in Appendix B.

7.6.6 System pricing must include line item indication where non-sponsoring companies' brands are used. System pricing must also include line item indication of third party pricing. See example in Appendix B.

Clause 8 - HANDLING OF RESULTS

This clause describe the pocess for results publication, challenge and withdrawal.

8.1 Results Submission

When a Test Sponsor decides to publish a ECperf benchmark result, he must generate the Full Disclosure as described in Clause 7 and submit it to the ECperf Review Committee by emailing it to ecperf_review@eng.sun.com. The ECperf Review Committee has a minimum period of 2 weeks to review the result and will vote on accepting it at their regularly scheduled meeting after the 2 week period.

During the review period, the members of the Review Committee will hold all review material in strict confidentiality. Within a member company this information should be shared with only those people necessary to provide a thorough review of the result.

At the end of the review, if there are no open issues related to a given submission, the result is approved and the Full Disclosure is posted on the ECperf results website. Once posted, the information contained in the disclosure is considered public.

If there are open issues for a given submission at the end of its review cycle that have not and can not be resolved to the satisfaction of the majority of the committee members, the committee votes that the submission is not compliant and should not be published. The Sponsor is informed of the problems with his result including all relevant clauses of the specification that are not met.

If minor corrections are made to a submission, e.g. documentation of tuning options or product-names, which do not affect performance, then the corrected result remains on the same publication-schedule as the original submission. If major corrections are made, e.g. a new run, different performance results, or different system configuration, then the corrected result starts review over as a new submission.

To ensure that submitters include all appropriate information without unfairly compromising their techmiques, the following guidelines must be followed by all members reviewing results:

- New and unique optimization techniques that appear in a full disclosure must not be utilized by another member until the original submission has completed its review cycle.

Submitters must accept that competitors attempting to surpass a result currently under review is considered common practice and is not prohibited. Competitors can also use published information found in press releases, product documentation, previously published ECperf results etc. at any time.

To assure fairness toward vendors whose products have been used to produce a benchmark result the following rule applies:

- A member may object to publication of results using its hardware or software product, during the results review period, if that product is deemed performance critical for the benchmark. If such an objection is made, the results will be held for six (6) weeks, unless the objection is withdrawn sooner. During the withholding period, the objector can submit a better result on the same configuration. In this case, the objector’s result will be reviewed

first and if accepted, the original submission will be rejected. If the objector does not choose to publish, then the original submission will be accepted if found compliant.

8.2 Violations in Published Results

The ECperf Expert Group may also undertake a re-review of a previously published result, if significant questions as to its adherence to the specification arise. The Expert Group reserves the right to mark a published result as non-compliant if it is found to be in violation of the Specification.

The following process has been established to handle instances where violations are discovered in results that have already been published :

- Anyone can file a Challenge against a published result by specifying the clauses in the specification that have been violated and how the violation occured. The challenge must be sent to the ECperf Expert Group.
- The Expert Group will evaluate the challenge and hear arguments from the challenger as well as the benchmark sponsor. It will then decide if the result is in violation and the nature of the violation.
- A **minor** violation typically involves documentation errors and can be rectified by the sponsor by submitting a new Full Disclosure. The revised Disclosure should only contain fixes for the errors found and will replace the currently published result.
- A **major** violation will result in the removal of the metrics from the summary published at the results website and replacing the column with “NC” (non-compliant). The Full Disclosure will be removed from the site and replaced with an explanation of why the result was found to be non-compliant.

8.3 Withdrawing a Result

A Sponsor may choose to withdraw a published result at any point by sending notice to the ECperf Expert Group. The result will be removed from the summary page of the ECperf results website and moved to the “Withdrawn Results” category. A withdrawn result cannot be referred to publicly or used for comparison by other vendors publishing results.

8.4 Fair Use of ECperf Results

The purpose of results publication is to allow vendors to publicize the performance of their products in Press Releases, marketing materials, etc. However, vendors need to exercise due diligence when using ECperf results and should adhere to the following rules :

- ECperf results should always be quoted using both the performance and the price-performance metrics and the category in which it was run, Centralized or Distributed. A pointer to the result on the official ECperf results website should be included.
- The term ECperf should only be used to refer to results published on the ECperf results website that followed the rules described in this Clause. Specifically, extrapolated results or results obtained in the lab cannot be referred to as ECperf results.
- Comparisons are not allowed against Withdrawn or Non-compliant results.

Appendix A Bean Interface Definitions

This Appendix describes the various enterprise beans and their interfaces. Note that the interface definition here may not be the latest. At all times, the ECperf kit contains the final version of the source code which must be used.

A.1 OrderSes Interface

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.

 *
 * $Id: OrderSes.java,v 1.5 2000/10/05 19:34:18 shanti Exp $

 *
 */
package com.sun.ecperf.orders.orderses.ejb;


import javax.ejb.*;

import java.rmi.*;

import com.sun.ecperf.common.*;
import com.sun.ecperf.orders.helper.*;


/**
 * This is the interface of the OrderSessionBean, which is a wrapper
 * for the Order and Orderline entity beans. The session bean also
 * implements the getCustStatus method to retrieve all orders
 * belonging to a particular customer
 */
public interface OrderSes extends EJBObject {

    /**
     * newOrder: Enter an order for a customer
     * @param customerId   - Id of customer
```

```
     * @param quantities
     * @return int - Id of order
     * @throws DataIntegrityException
     * @exception InsufficientCreditException - if customer credit check
fails
     * @exception CreateException - if creation of order fails
     * @exception RemoteException  - if there is a system failure
     */
    public int newOrder(int customerId, ItemQuantity[] quantities)
        throws InsufficientCreditException, DataIntegrityException,
               RemoteException, CreateException;

    /**
     * changeOrder: Changes an existing customer order
     * @param orderId   - Id of order being changed
     * @param quantities
     * @exception InsufficientCreditException - if customer credit check
fails
     * @exception RemoteException  - if there is a system failure
     */
    public void changeOrder(int orderId, ItemQuantity[] quantities)
        throws InsufficientCreditException, RemoteException;

    /**
     * cancelOrder: Cancel an existing customer order
     * @param orderId   - Id of order being changed
     * @exception RemoteException  - if there is a system failure
     */
    public void cancelOrder(int orderId) throws RemoteException;

    /**
     * getOrderStatus: Retrieves status of an order
     * @param orderId   - Id of order
     * @return OrderStatus object
     *
     * @throws DataIntegrityException
     * @exception RemoteException  - if there is a system failure
     */
    public OrderStatus getOrderStatus(int orderId)
        throws DataIntegrityException, RemoteException;
```

```
    /**
     * Get status of all orders of a Customer
     *
     * @param customerId      int customer id
     * @return          Array of CustomerStatus objects (one for each
order)
     *
     * @throws DataIntegrityException
     * @exception RemoteException  - if there is a system failure
     */
    public CustomerStatus[] getCustomerStatus(int customerId)
        throws DataIntegrityException, RemoteException;
}
```

A.2 ItemQuantity.java

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.

 *
 * $Id: ItemQuantity.java,v 1.2 2000/10/04 20:52:52 ramesh Exp $
 *
 */
package com.sun.ecperf.orders.helper;


/**
 * An object of this class is used to represent item, qty pairs
 * in line items of an order.
 */
public class ItemQuantity implements java.io.Serializable {

    /**
     * Constructor ItemQuantity
     *
     *
```

```
     * @param itemId
     * @param itemQuantity
     *
     */
    public ItemQuantity(String itemId, int itemQuantity) {
        this.itemId       = itemId;
        this.itemQuantity = itemQuantity;
    }

    public String itemId;
    public int    itemQuantity;
}
```

A.3 CustomerInfo.java

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 * $Id: CustomerInfo.java,v 1.3 2000/10/04 19:24:10 ramesh Exp $
 *
 */
package com.sun.ecperf.common;


import java.io.*;


/**
 * This class is the CustomerInfo object passed to the addCustomer
 * method of the OrderCustomer beans
 *
 * @author Shanti Subramanyam
 */
public class CustomerInfo implements Serializable {

    /**
     * Constructor CustomerInfo
```

```
     *
     *
     * @param first
     * @param last
     * @param address
     * @param contact
     * @param credit
     * @param creditLimit
     * @param balance
     * @param YtdPayment
     *
     */
    public CustomerInfo(String first, String last, Address address,
                        String contact, String credit, double creditLimit,
                        double balance, double YtdPayment) {

        this.firstName   = first;
        this.lastName    = last;
        this.address     = address;
        this.contact     = contact;
        this.since       = new java.sql.Date(new java.util.Date().get-
Time());
        this.credit      = credit;
        this.creditLimit = creditLimit;
        this.balance     = balance;
        this.YtdPayment  = YtdPayment;
    }

    public Integer       customerId;
    public String        firstName;
    public String        lastName;
    public Address       address;
    public String        contact;
    public String        credit;
    public double        creditLimit;
    public java.sql.Date since;
    public double        balance;
    public double        YtdPayment;
}
```

A.4 CustomerStatus.java

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.

 *
 * $Id: CustomerStatus.java,v 1.2 2000/10/04 20:52:51 ramesh Exp $
 *
 */
package com.sun.ecperf.orders.helper;


import java.io.Serializable;


/**
 * An object of this class is returned by the getCustomerStatus method
 * of the OrderSession bean.
 */
public class CustomerStatus implements Serializable {

    public int            orderId;
    public java.sql.Date  shipDate;
    public ItemQuantity[] quantities;
}
```

A.5 OrderCustomerSes Interface

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.

 *
 * $Id: OrderCustomerSes.java,v 1.3 2000/10/04 21:08:21 ramesh Exp $
```

```
 *
 */
package com.sun.ecperf.orders.ordercustomerses.ejb;


import javax.ejb.*;

import java.rmi.*;

import com.sun.ecperf.common.*;
import com.sun.ecperf.orders.helper.*;


/**
 * The OrderCustomerSession bean is a session bean whose interface
 * is exposed to the client.
 * It uses the underlying OrderCustomer entity bean to do its work.
 */
public interface OrderCustomerSes extends javax.ejb.EJBObject {

    /**
     * This method checks for the existence of a customer with
     * the specified id.
     */
    public void validateCustomer(int id)
        throws DataIntegrityException, RemoteException;

    /**
     * This method adds a new customer with the specified info
     *
     * @param info
     * @return id - Customer id of newly created customer
     */
    public int addCustomer(CustomerInfo info)
        throws InvalidInfoException, DataIntegrityException, RemoteExcep-
tion;

    /**
     * Method getPercentDiscount
     *
```

```
     *
     * @param customerId
     * @param total
     *
     * @return
     *
     * @throws DataIntegrityException
     * @throws RemoteException
     *
     */
    public double getPercentDiscount(int customerId, double total)
        throws DataIntegrityException, RemoteException;
}
```

A.6 WorkOrderSes Interface

```
/*
 *
 * Copyright (c) 1999-2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 * @author Ajay Mittal
 *
 * akmits@eng.sun.com 04/03/2000
 */
package com.sun.ecperf.mfg.workorderses.ejb;


import javax.ejb.EJBObject;

import java.rmi.RemoteException;


/**
 * This interface is the remote interface for the WorkOrderSes
 * session bean. This bean is stateless.
 *
 * @author Ajay Mittal
```

```
 *
 *
 */
public interface WorkOrderSes extends EJBObject {

    // if called from manufacturing app (driver)

    /**
     * Method to schedule a work order.
     * @param assemblyId    Assembly Id
     * @param qty           Original Qty
     * @param dueDate       Date when order is due
     *
     * @return
     * @exception RemoteException if there is a system failure
     */
    public Integer scheduleWorkOrder(String assemblyId, int qty, java.sql
        .Date dueDate) throws RemoteException;

    // if called from order domain

    /**
     * Method to schedule a work order.
     * @param salesId       Sales order id
     * @param oLineId       Order Line ID
     * @param assemblyId    Assembly Id
     * @param qty           Original Qty
     * @param dueDate       Date when order is due
     *
     * @return
     * @exception RemoteException if there is a system failure
     */
    public Integer scheduleWorkOrder(
        int salesId, int oLineId, String assemblyId, int qty,
            java.sql.Date dueDate) throws RemoteException;

    /**
       * completeWorkOrder: workorder cancel.
       *
       * Transfer completed portion to inventory.
```

```
         * @param wid               WorkOrder ID
         * @return                  boolean false if failed to complete
         * @exception               RemoteException if there is
         *                          a communications or systems failure
         */
        public boolean completeWorkOrder(Integer wid) throws RemoteException;

        /**
          * cancelWorkOrder: workorder cancel.
          *
          * Transfer completed portion to inventory. Abort remaining work
    order
          * @param wid              WorkOrder ID
          * @return                 boolean false if failed to complete
          * @exception              RemoteException if there is
          *                         a communications or systems failure
          */
        public boolean cancelWorkOrder(Integer wid) throws RemoteException;

        /**
         * Get completed Qty in the workorder
         *
         * @param wid                     WorkOrder ID
         * @return            int status
         * @exception         RemoteException if there is
         *                    a communications or systems failure
         */
        public int getWorkOrderCompletedQty(Integer wid) throws RemoteExcep-
    tion;

        /**
         * Get status of a workorder
         *
         * @param wid              WorkOrder ID
         * @return            int status
         * @exception         RemoteException if there is
         *                    a communications or systems failure
         */
        public int getWorkOrderStatus(Integer wid) throws RemoteException;
```

```
    /**
     * Update status of a workorder
     *
     * @param wid                       WorkOrder ID
     * @exception               RemoteException if there is
     *                          a communications or systems failure
     */
    public void updateWorkOrder(Integer wid) throws RemoteException;
}
```

A.7 LargeOrderSes Interface

```
/*
 *
 * Copyright (c) 1999-2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 */
package com.sun.ecperf.mfg.largeorderses.ejb;


import javax.ejb.EJBObject;

import java.rmi.RemoteException;


/**
 * This interface is the remote interface for the Large Order
 * session bean. This bean is stateless.
 * @author Agnes Jacob
 */
public interface LargeOrderSes extends EJBObject {

    /**
     * Method createLargeOrder
     *
     *
```

```
     * @param orderId
     * @param oLineId
     * @param assemblyId
     * @param qty
     * @param dueDate
     *
     * @return
     *
     * @throws RemoteException
     *
     */
    public Integer createLargeOrder(
        int orderId, int oLineId, String assemblyId, short qty,
            java.sql.Date dueDate) throws RemoteException;

    /**
     * Method findLargeOrders
     *
     *
     * @return
     *
     * @throws RemoteException
     *
     */
    public java.util.Vector findLargeOrders() throws RemoteException;
}
```

A.8 LargeOrderInfo.java

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 * $Id: LargeOrderInfo.java,v 1.2 2000/10/04 20:08:52 ramesh Exp $
 *
 */
package com.sun.ecperf.mfg.helper;
```

```
import java.io.*;


/**
 * Class LargeOrderInfo
 *
 *
 * @author
 * @version %I%, %G%
 */
public class LargeOrderInfo implements Serializable {

    public Integer       id;
    public int           salesOrderId;
    public int           orderLineNumber;
    public String        assemblyId;
    public short         qty;
    public java.sql.Date dueDate;

    /**
     * Method duplicate
     *
     *
     * @return
     *
     */
    public LargeOrderInfo duplicate() {

        LargeOrderInfo loi = new LargeOrderInfo();

        loi.id              = this.id;
        loi.salesOrderId    = this.salesOrderId;
        loi.orderLineNumber = this.orderLineNumber;
        loi.assemblyId      = this.assemblyId;
        loi.qty             = this.qty;
        loi.dueDate         = this.dueDate;

        return loi;
    }
```

```
}
```

A.9 BuyerSes Interface

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 * $Id:
 *
 */
package com.sun.ecperf.supplier.buyerses.ejb;


import javax.ejb.*;

import java.rmi.*;

import com.sun.ecperf.common.*;


/**
 * This is the interface of the BuyerSessionBean. It is the
 * interface from the Manufacturing Domain to the Supplier Domain.
 * The Mfg domain repeatedly calls the add method to add components
 * that are low in inventory.  When all components have been added
 * the Mfg domain calls purchase to issue the Purchase Order.
 *
 * Working on method getPOStatus to see how we can utilize it in Mfg.
 *
 * @author Damian Guy
 *
 */
public interface BuyerSes extends EJBObject {

    /**
     * add: Add components that are low in inventory to PO.
```

```
     * @param component_id  - Id of component that is being added to PO.
     * @param qty_required - number to be purchased.
     * @exception RemoteException - if there is a system failure.
     */
    public void add(String component_id, int qty_required)
        throws RemoteException;

    /**
     * purchase: Issue the Purchase Order.
     * @throws ECperfException
     * @exception RemoteException - if there is a system failure.
     * @exception CreateException - if creation of Purchase Order fails.
     * @exception FinderException
     */
    public void purchase()
        throws RemoteException, FinderException, CreateException,
               ECperfException;

    /**
     * getPOStatus: Retrieve status information of outstanding PO.
     * @param poID - Id of the Purchase Order.
     * @return POStatus - Object containg PO Status information.
     * @exception RemoteException - if there is a system failure.
     */

    //      public POStatus getPOStatus(int poId)
    //              throws RemoteException;
}
```

A.10 ReceiveSes Interface

```
/*
 *
 * Copyright (c) 1999-2000 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 * @author Ajay Mittal
 *
```

```
 * akmits@eng.sun.com 04/03/2000
 */
package com.sun.ecperf.mfg.receiveses.ejb;


import javax.ejb.EJBObject;

import java.rmi.RemoteException;


/**
 * This interface is the remote interface for the ReceiveSes
 * session bean. This bean is stateless.
 *
 * @author Ajay Mittal
 *
 *
 */
public interface ReceiveSes extends EJBObject {

    /**
      * Method to add components to the inventory.
      * @param compoID unique component ID
      * @param numComponents number of components to be added
      * @exception RemoteException if there is a system failure
      */
    public void addInventory(String compoID, int numComponents)
        throws RemoteException;
}
```

A.11 ReceiverSes Interface

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.

 * $Id: ReceiverSes.java,v 1.2 2000/10/04 22:27:48 ramesh Exp $
 *
```

```
 */
package com.sun.ecperf.supplier.receiverses.ejb;


//Import statements
import javax.ejb.*;

import java.rmi.*;

import com.sun.ecperf.supplier.helper.*;


/**
 * Remote interface for the the stateless session bean
 * ReceiverSes.
 *
 * @author Damian Guy
 */
public interface ReceiverSes extends EJBObject {

    /**
     * deliverPO - indicate the part of a PO has been delivered.
     * @param del - contains information about the POLine that has been
delivered.
     * @exception RemoteException - if there is a system failure.
     */
    public void deliverPO(DeliveryInfo del) throws RemoteException;
}
```

A.12 DeliveryInfo.java

```
/*
 * Copyright (c) 1998-2000 by Sun Microsystems, Inc. All Rights Reserved.
 * This software is the proprietary information of Sun Microsystems, Inc.
 * Use is subject to license terms.
 *
 */
package com.sun.ecperf.supplier.helper;
```

```
import java.io.Serializable;


/**
 * This class is the DeliveryInfo Object passed to the deliveredPO
 * method of the Receiver bean.
 *
 * @author Damian Guy
 */
public class DeliveryInfo implements Serializable {

    public int    poId;
    public int    line_number;
    public String part_id;
    public int    qty;

    /**
     * Constructor DeliveryInfo
     *
     *
     * @param poId
     * @param line_number
     * @param part_id
     * @param qty
     *
     */
    public DeliveryInfo(int poId, int line_number, String part_id, int
qty) {

        this.poId       = poId;
        this.line_number = line_number;
        this.part_id     = part_id;
        this.qty        = qty;
    }
}
```

Appendix B Sample Summary Statement

This Appendix shows an example of a Summary Statement. The Summary Statement must be included as part of the Full Disclosure and also appear at the beginning of the Full Disclosure Report (see Clause 8.3).

The first page of the Summary Statement must show the benchmark results and high-level configuration information. All of the commercial components used in the benchmark must be listed.

The second page of the Summary Statement is the Pricing spreadsheet that must adhere to the requirements in Clause 7. A sample Pricing spreadsheet is shown below.

The final pages of the Summary Statement must include the summary reports produced by the ECperf Driver for the benchmark run being reported. These are ECperf.summary, Orders.summary and Mfg.summary.

XYZ Corporation: BigServer 1234
ABC Inc. : UltraFast AppServer 1.0
ODS Software : Superior DBMS 3.5

Metrics: n.nn BBops/min@Std $xxx/BBops/min@Std
Availability Date: dd Mon yyyy
Bean Deployment Mode: BMP/CMP/Mixed
Configuration:

Diagram showing all systems in the SUT

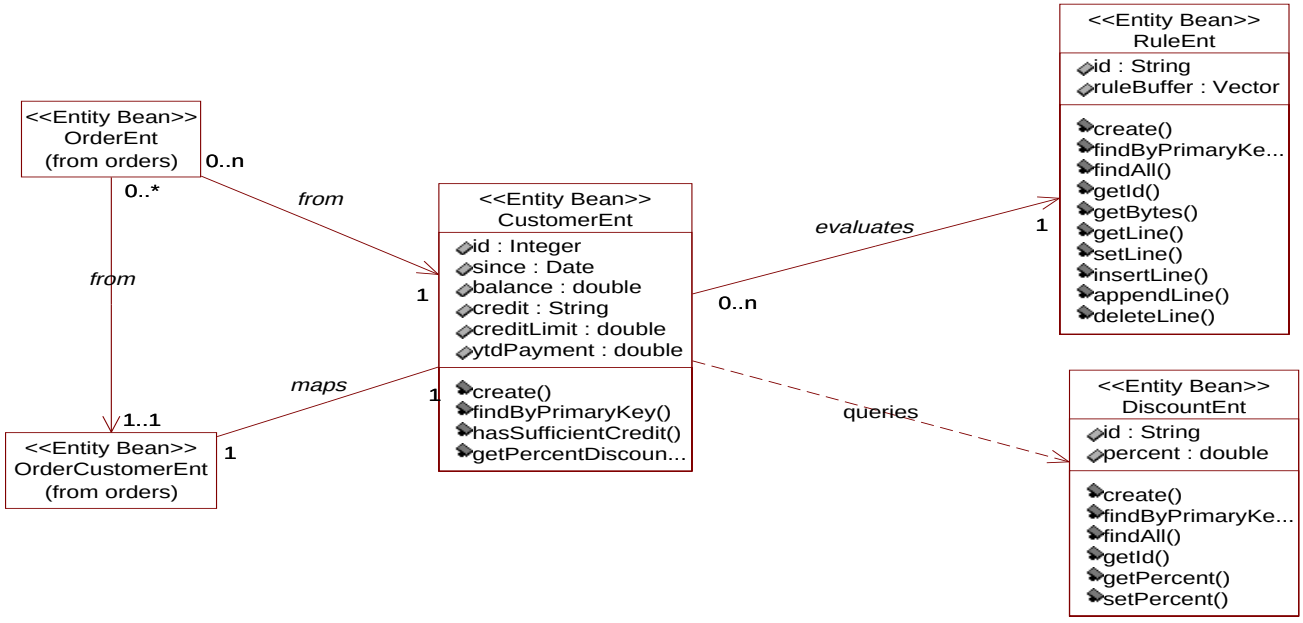
System	Software	Processors	Memory	Disk
BigServer 1234	UltraFast AppServer 1.0 JDK 1.3 NewOS 2.0	4xPentiumIII 700Mhz 2MB Ecache	4GB	9GB Internal
SmallServer 2	Superior DBMS 3.5 NewOS 2.0	2xPentiumIII 450Mhz 1MB Ecache	1GB	1 Fast Array 4x9GB

XYZ Corporation	Big Server 1234	ECperf Version 1.0 Standard Workload

Description	Part Number	Third Party Brand Pricing	Unit Price	Qty	Extended Price	Maintenace Price
Hardware						
1234 with 4xPentium III 700Mhz 2MB Ecache, 9GB HDD	201-A	XYZ	15,995	1	15,995	3,800
128 MB memory	8203	XYZ	11,495	1	11,495	
14” EPA SVGA Monitor	2600	XYZ	995	1	995	
SmallServer 2 with 2xPentiumII	SS12	XYZ	1,195	1	1,195	999
			Subtotal		**29,680**	**4,799**
Software						
XYZ Unix	U123	XYZ	995	1	995	
UltraFast AppServer	UFA1	ABC	21,000	1	21000	
Superior DBMS 1.0	RD123	ODS	10,000	1	10000	800
			SubTotal		**31,995**	**800**
			Total		**61,675**	**5,599**
		1-Year Cost of Ownership			**67,274**	
			OPS@Std		**25.34**	
			$/OPS@Std		**2654.85**	

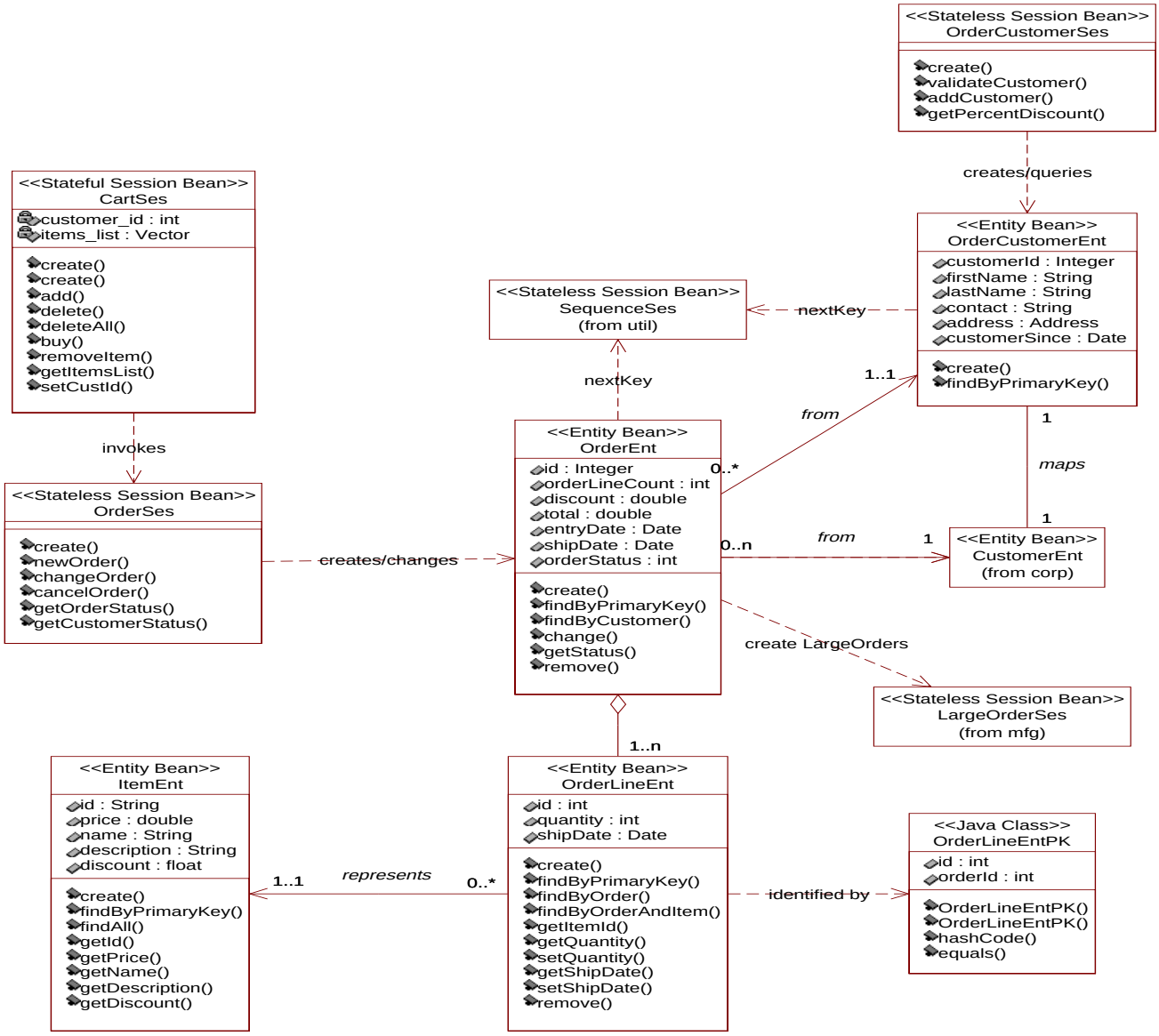
Appendix C UML Diagrams

C.1 Corporate Domain Class Diagram



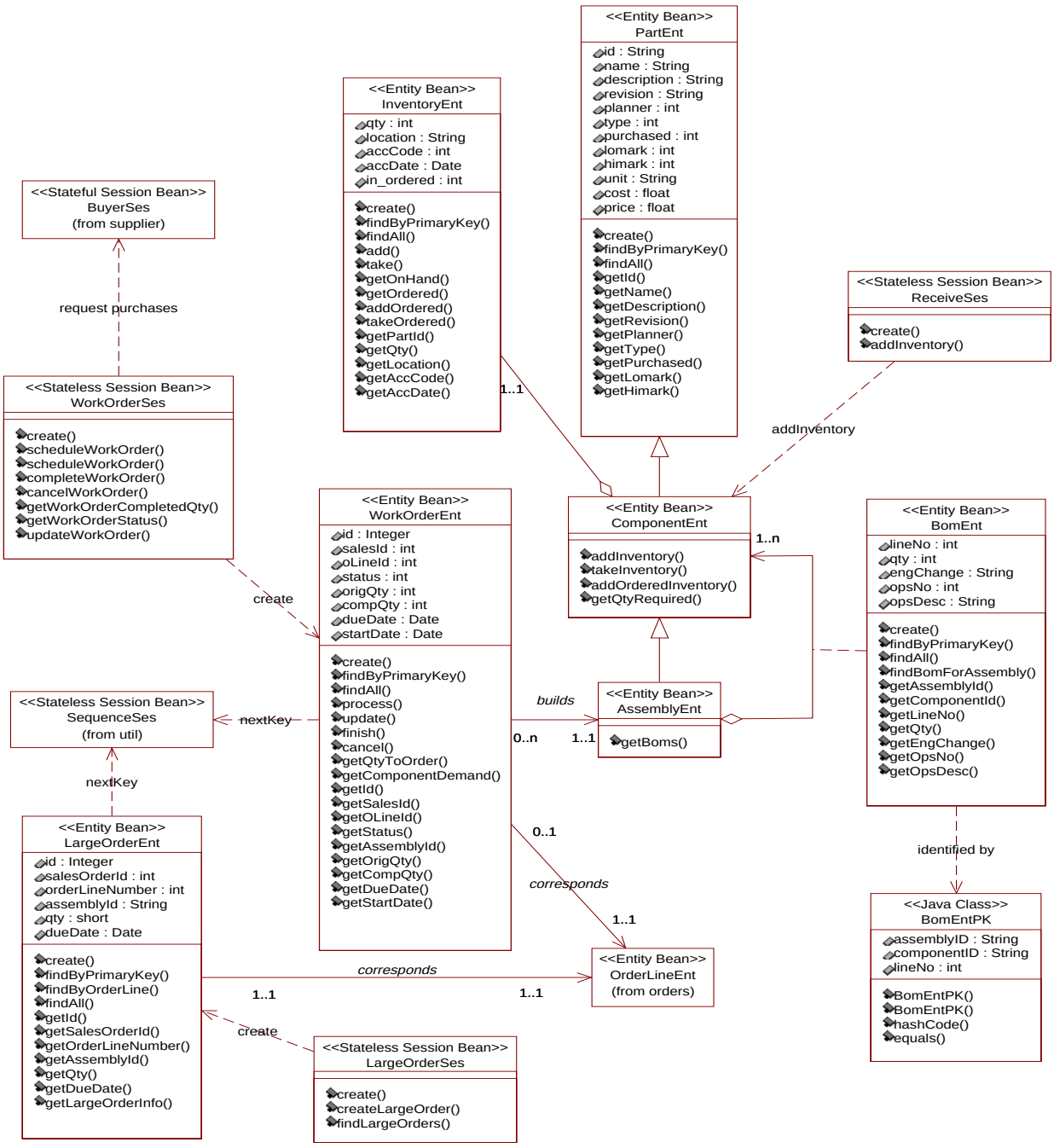
File: /home/akara/ecperf2/ecperf.mdl Fri Jan 26 12:08:39 2001 Class Diagram: corp / Main Page 1

C.2 Customer Domain Class Diagram

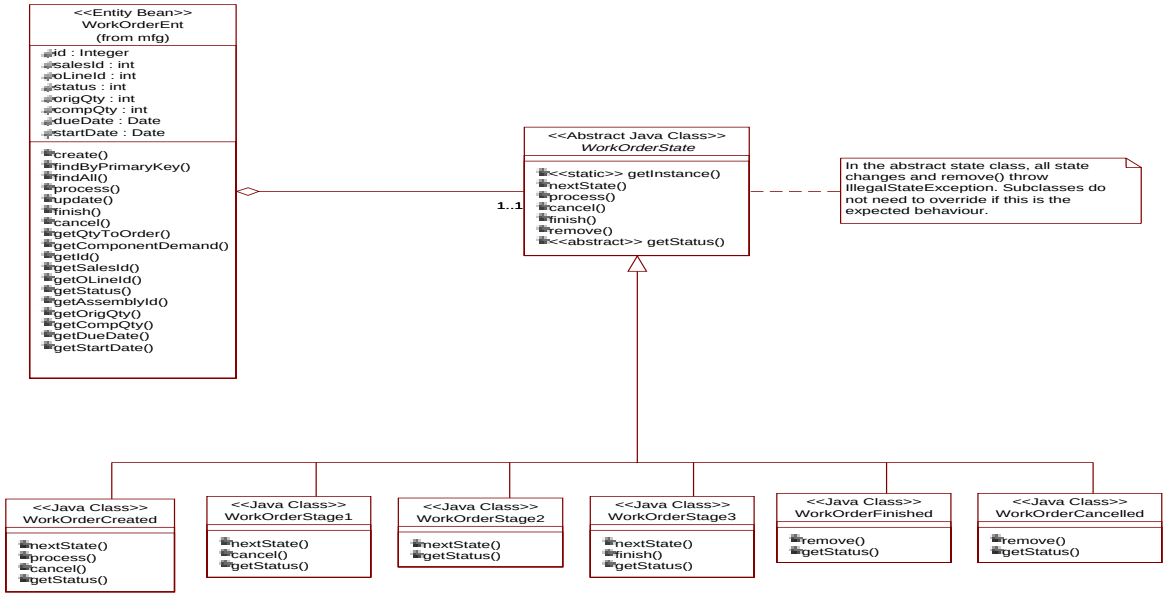


File: /home/akara/ecperf2/ecperf.mdl Fri Jan 26 12:12:21 2001 Class Diagram: orders / Main Page 1

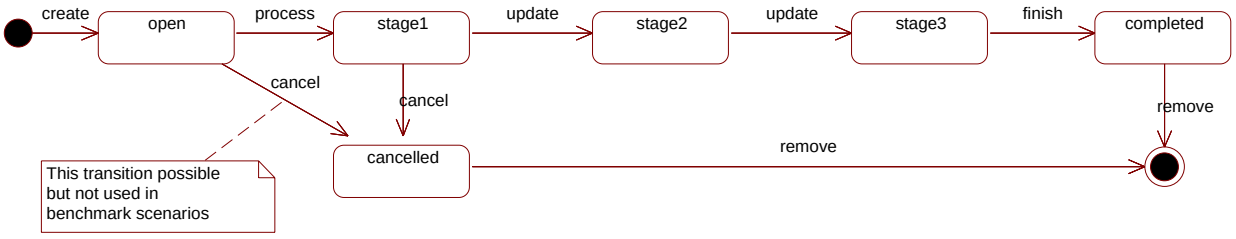
C.3 Manufacturing Domain Class Diagram



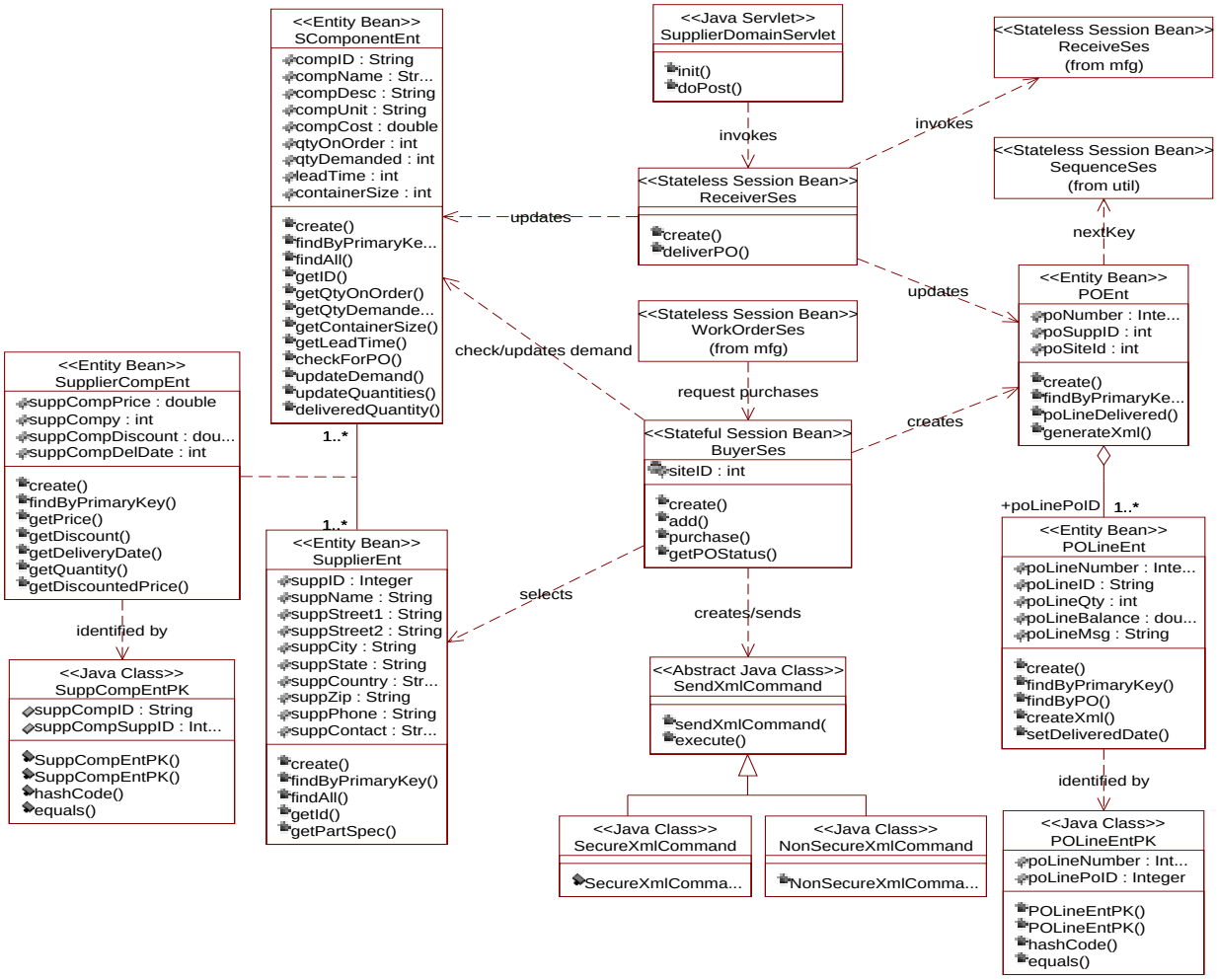
C.4 Manufacturing State Classes



C.5 WorkOrder State Transition



C.6 Supplier Domain Class Diagram



C.7 Key Generation Util Class Diagram