Section 3: Payroll Architecture Handbook

Version 2004

Payroll Architecture Handbook

Description

This document supplements the course material for the Payroll Exercise used in the Object-Oriented Analysis and Design Using the UML course. It provides the architectural givens that support the development of the Payroll System design model during the course exercises.
This is because the OOAD course concentrates on demonstrating how architecture affects the design model. OOAD is NOT an architecture course. The OOAD course gives the students an appreciation of what an architecture is and why it is important.

In some sections of this document, the architecture is represented textually. The students, as part of the exercises throughout the course, will generate the associated UML diagrams. Thus, for the UML representation of the architecture, see the Payroll Exercise Solution.

Note: A SUBSET OF THE PAYROLL SYSTEM IS PROVIDED. Concentration is on the elements needed to support the Login, Maintain Timecard and Run Payroll use cases.

Architectural Mechanisms

Analysis Mechanisms

Persistency: A means to make an element persistent (i.e., exist after the application that created it ceases to exist).
Distribution: A means to distribute an element across existing nodes of the system.
Note: For this course, it has been decided that the business logic will be distributed.
Security: A means to control access to an element.
Legacy Interface: A means to access a legacy system with an existing interface.

Analysis-to-Design-to-Implementation Mechanisms Map

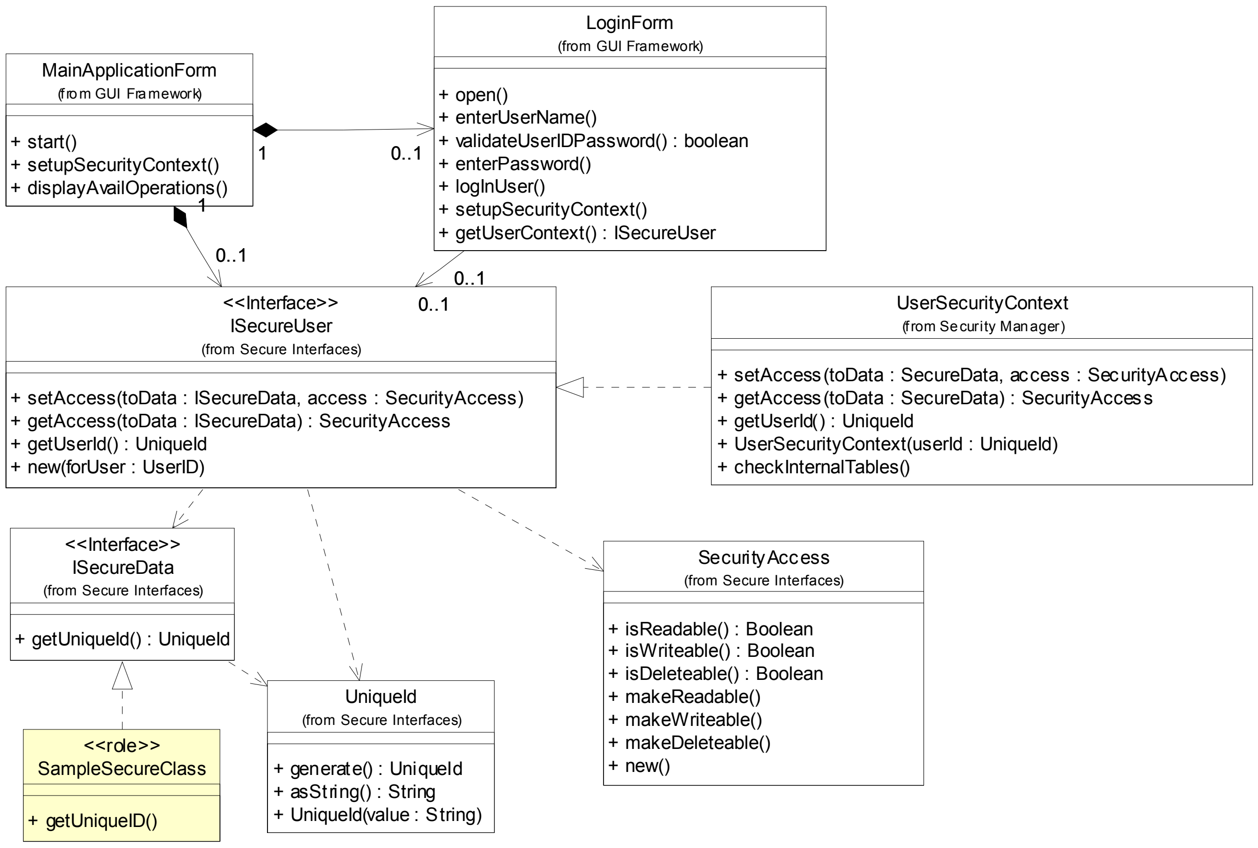
Analysis Mechanism	**Design Mechanisms**	**Implementation Mechanisms**
Persistency	OODBMS (new data)	ObjectStore
Persistency	RDBMS (data from legacy database)	JDBC to Ingres
Distribution	Remote Method Invocation (RMI)	Java 1.1 from Sun
Security		Reverse Engineered Secure.java and UserContextRemoteObject components
Legacy Interface		

Note: Remote Method Invocation (RMI) is a Java-specific mechanism that allows client objects to invoke operations on server objects as though they were local. Native Java RMI comes with Sun's Java 1.1.

Implementation Mechanisms

Security

Static View: Security



Class Descriptions

ISecureData : Analysis Mechanisms:
- Security

SecurityAccess : Analysis Mechanisms:
- Security

SampleSecureClass :

UserSecurityContext : Analysis Mechanisms:
- Security

UniqueId : Analysis Mechanisms:
- Security

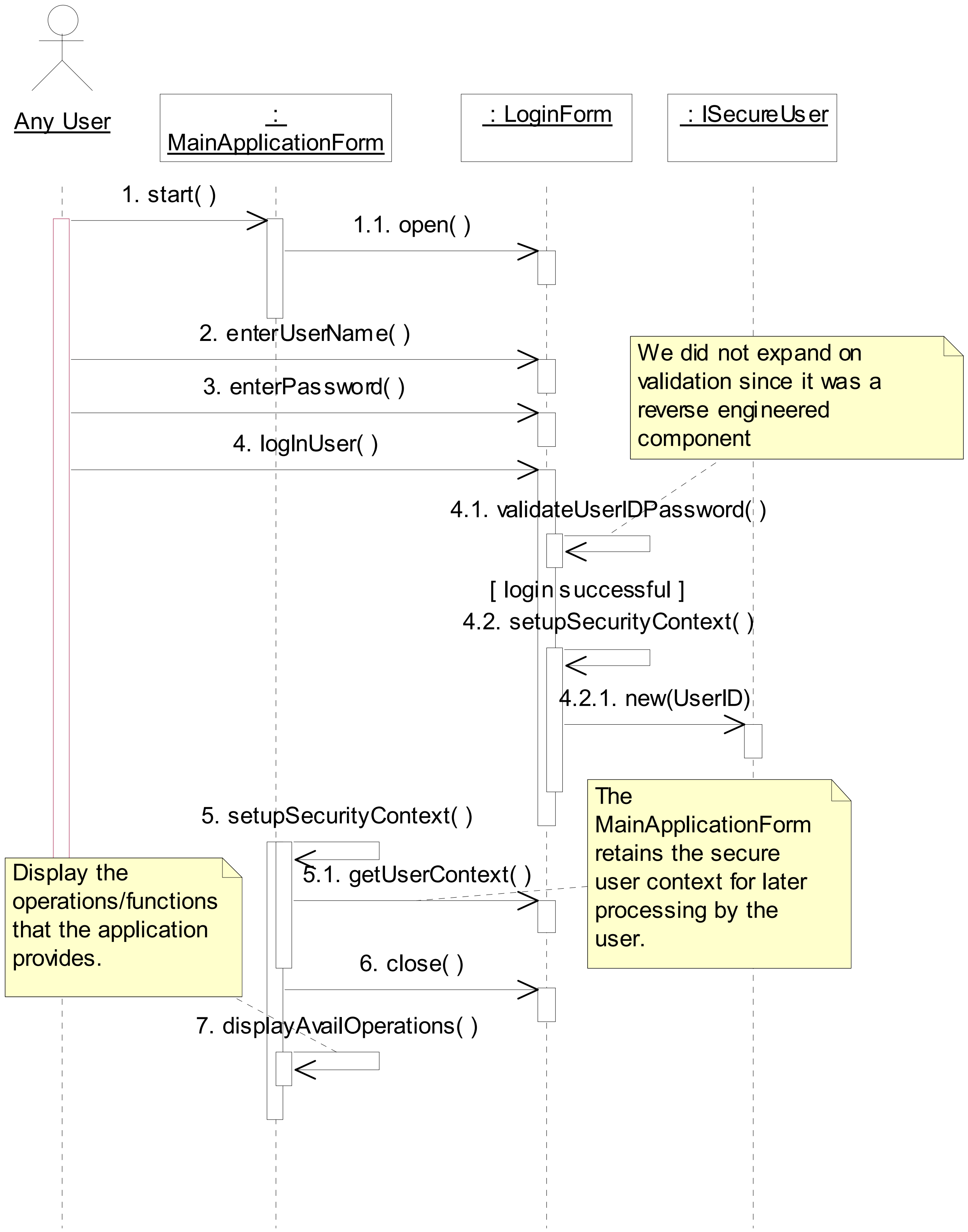
MainApplicationForm : Requirements Traceability:
- Usability: The desktop user-interface shall be Windows 95/98 compliant.

ISecureUser : Analysis Mechanisms:
- Security

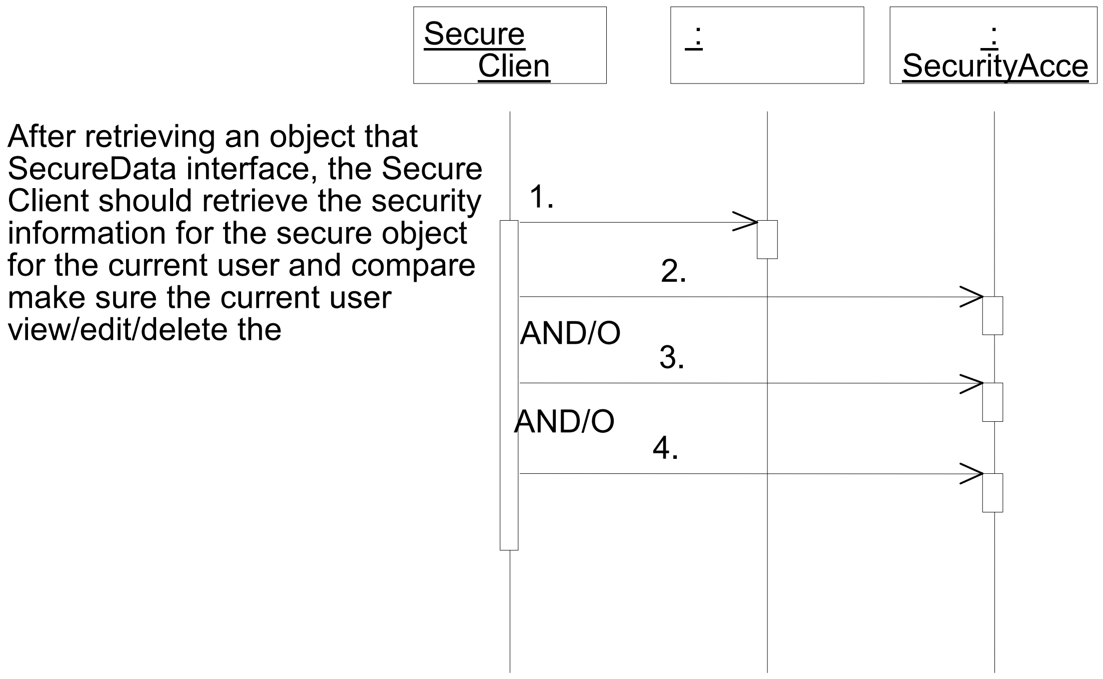
LoginForm : Analysis Mechanisms:
- Security

Requirements Traceability:
- Usability: The desktop user-interface shall be Windows 95/98 compliant.

Dynamic View: Secure User Set-Up

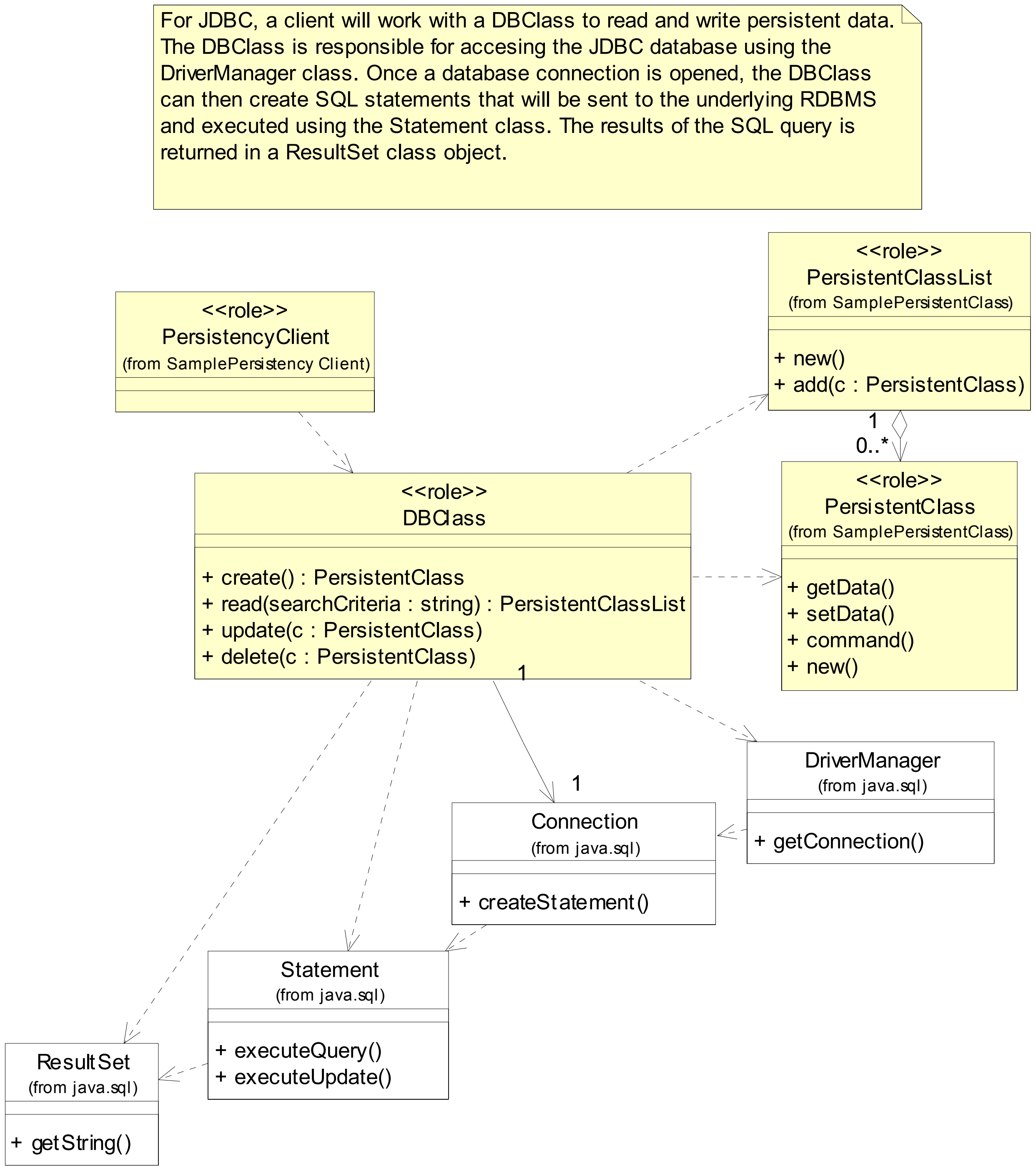

Any User
: MainApplicationForm
: LoginForm
: ISecureUser
1. start( )
1.1. open( )
2. enterUserName( )
3. enterPassword( )
4. logInUser( )
4.1. validateUserIDPassword( )
[login successful]
4.2. setupSecurityContext( )
4.2.1. new(UserID)
5. setupSecurityContext( )
5.1. getUserContext( )
6. close( )
7. displayAvailOperations( )
We did not expand on validation since it was a reverse engineered component
The MainApplicationForm retains the secure user context for later processing by the user.
Display the operations/functions that the application provides.

Dynamic View: Secure Data Access


Secure Clien
:
: SecurityAcce
After retrieving an object that SecureData interface, the Secure Client should retrieve the security information for the secure object for the current user and compare make sure the current user view/edit/delete the
1.
2.
AND/O
3.
AND/O
4.

Persistency - RDBMS - JDBC

Static View: Persistency JDBC


For JDBC, a client will work with a DBClass to read and write persistent data. The DBClass is responsible for accesing the JDBC database using the DriverManager class. Once a database connection is opened, the DBClass can then create SQL statements that will be sent to the underlying RDBMS and executed using the Statement class. The results of the SQL query is returned in a ResultSet class object.
<<role>> PersistencyClient (from SamplePersistency Client)
<<role>> PersistentClassList (from SamplePersistentClass)
+ new()
+ add(c : PersistentClass)
1
0..*
<<role>> PersistentClass (from SamplePersistentClass)
+ getData()
+ setData()
+ command()
+ new()
<<role>> DBClass
+ create() : PersistentClass
+ read(searchCriteria : string) : PersistentClassList
+ update(c : PersistentClass)
+ delete(c : PersistentClass)
1
1
DriverManager (from java.sql)
+ getConnection()
Connection (from java.sql)
+ createStatement()
Statement (from java.sql)
+ executeQuery()
+ executeUpdate()
ResultSet (from java.sql)
+ getString()

Class Descriptions

PersistencyClient : An example of a client of a persistent class.

PersistentClass : An example of a class that's persistent.

PersistentClassList :

Statement : The class used for executing a static SQL statement and obtaining the results produced by it. SQL statements without parameters are normally executed using Statement objects.

DBClass : A sample of a class that would be responsible for making another class persistent.
Every Class that's persistent will have a corresponding DBClass (e.g., Student will have a DBStudent class).

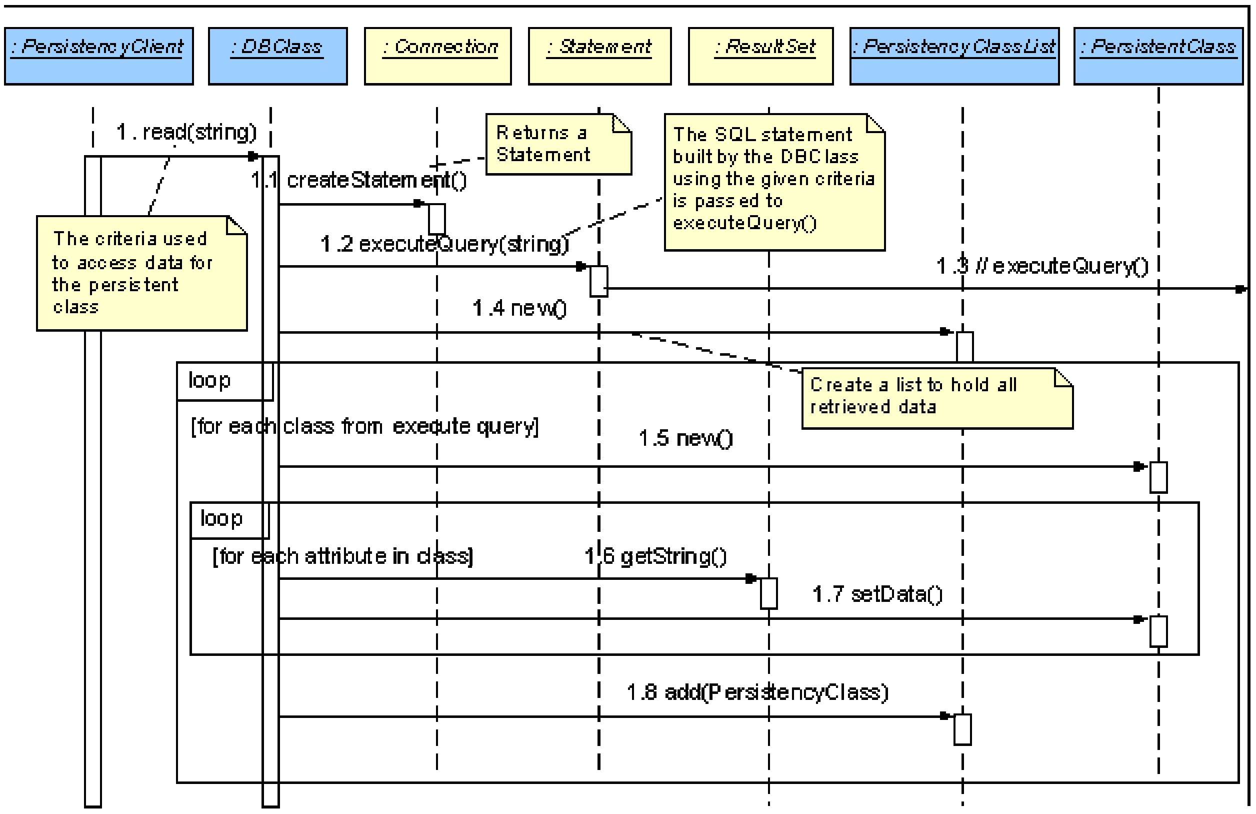
With an RDBMS, you need a mapping of objects/classes to tables, and you must recreate the (association/aggregation) structures. DBClass is a database interface class which understands the OO-to-RDBMS mapping and has the behavior to interface with the RDBMS. This database interface class is used whenever a persistent class needs to be created, accessed, or deleted. The database interface class flattens the object and writes it to the RDBMS and reads the object data from the RDBMS and builds the object.

Connection : A connection (session) with a specific database. Within the context of a Connection, SQL statements are executed, and results are returned.

ResultSet : A ResultSet provides access to a table of data. A ResultSet object is usually generated by executing a Statement.

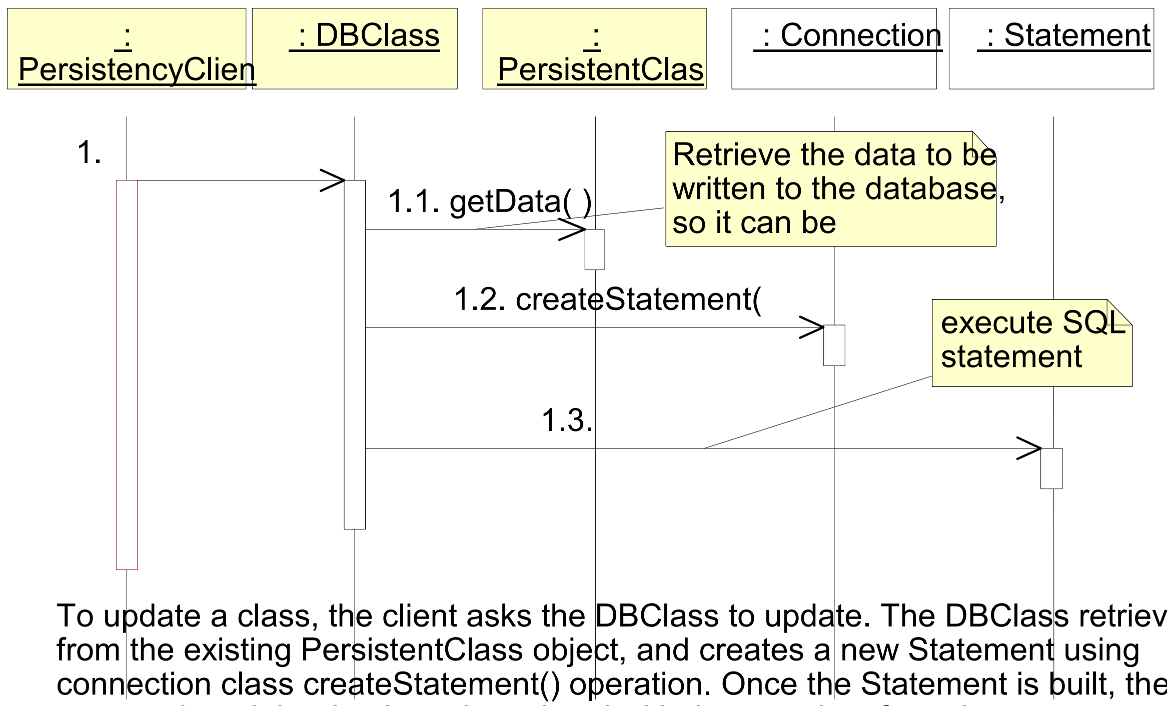
DriverManager : The basic service for managing a set of JDBC drivers.

Dynamic View: JDBC RDBMS Read


: PersistencyClient
: DBClass
: Connection
: Statement
: ResultSet
: PersistencyClassList
: PersistentClass
1 . read(string)
1.1 createStatement()
Returns a Statement
The SQL statement built by the DBClass using the given criteria is passed to executeQuery()
1.2 executeQuery(string)
1.3 // executeQuery()
The criteria used to access data for the persistent class
1.4 new()
Create a list to hold all retrieved data
loop
[for each class from execute query]
1.5 new()
loop
[for each attribute in class]
1.6 getString()
1.7 setData()
1.8 add(PersistencyClass)

Dynamic View: JDBC RDBMS Update

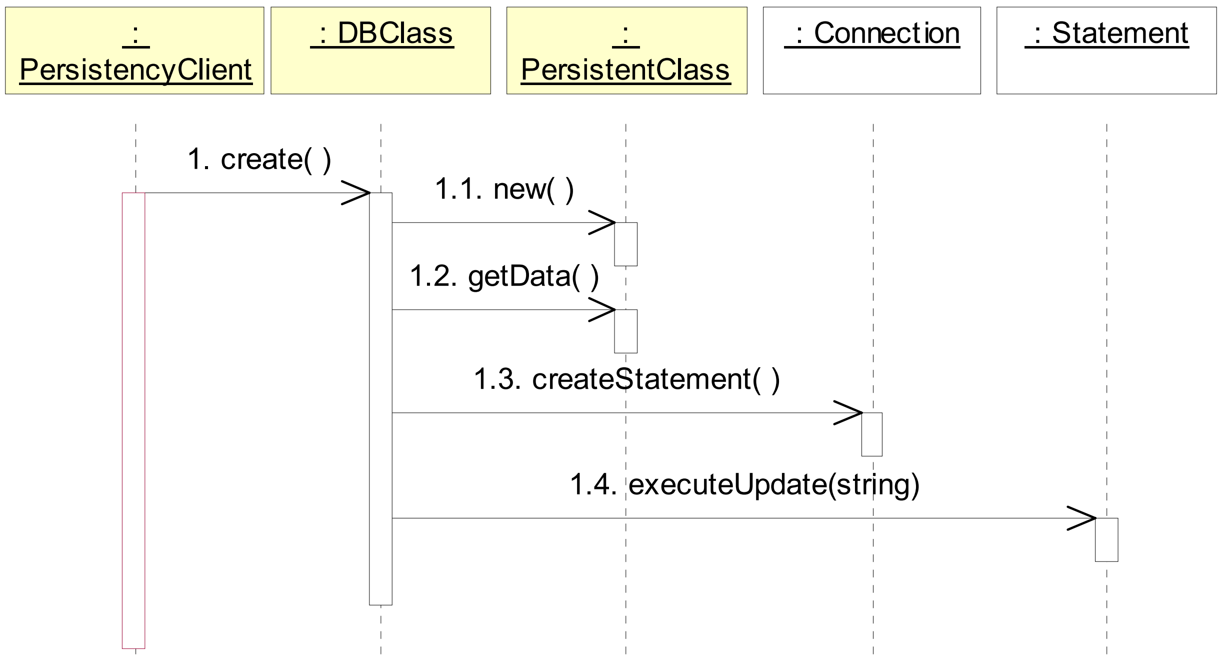
JDBC RDBMS



To update a class, the client asks the DBClass to update. The DBClass retriev
from the existing PersistentClass object, and creates a new Statement using
connection class createStatement() operation. Once the Statement is built, the
executed, and the database is updated with the new data from the

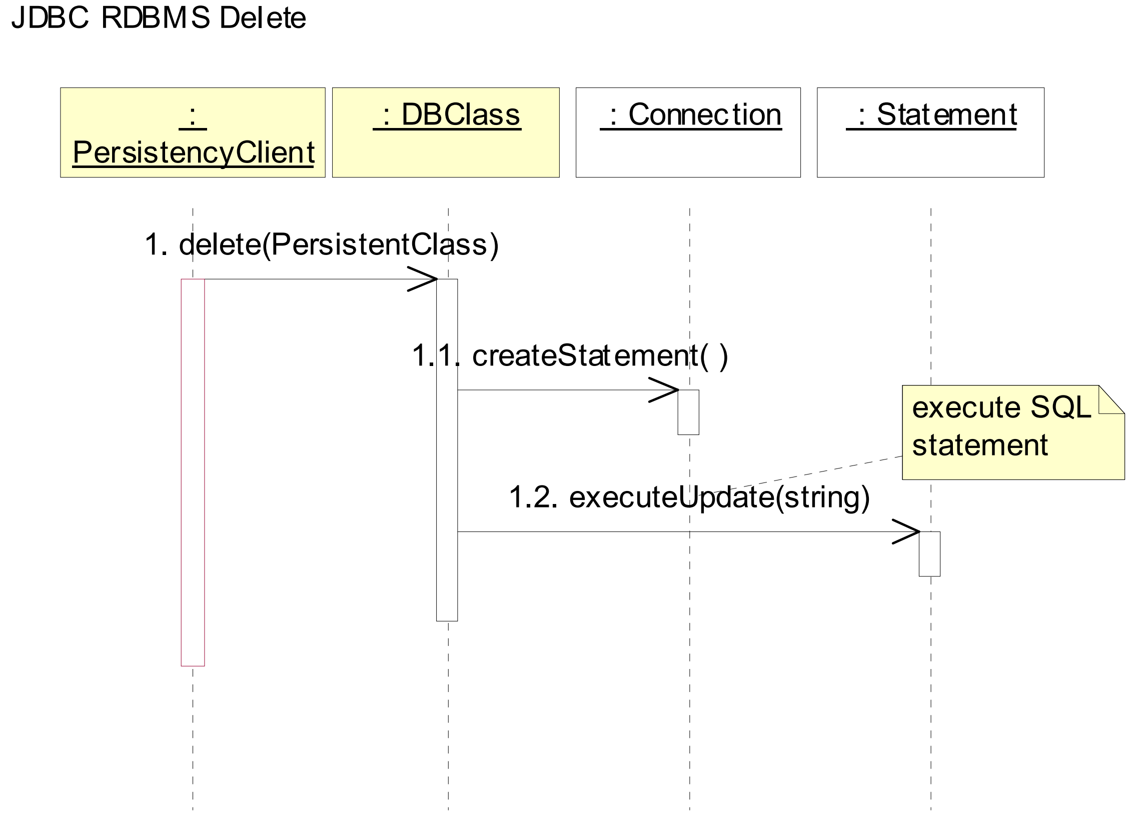
Dynamic View: JDBC RDBMS Create

JDBC RDBMS Create



To create a new class, the client asks the DBClass to create the new class. The DBClass creates a new instance of Persistent Class with default values. The DBClass then creates a new Statement using the Connection class createStatement() operation. The statement is executed and the data is inserted into the database.

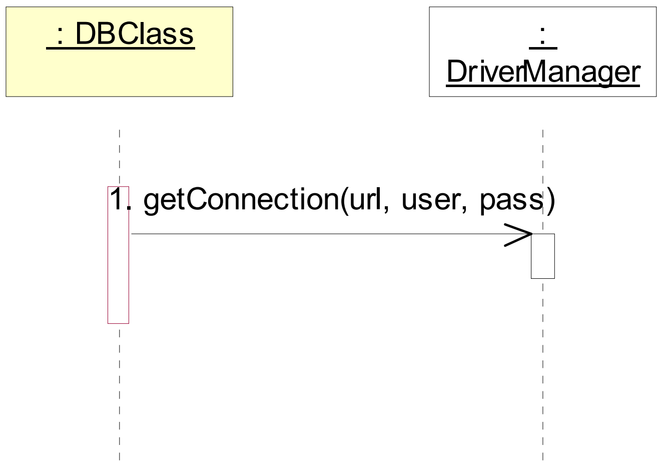
Dynamic View: JDBC RDBMS Delete



To delete a class, the client asks the DBClass to delete a specific class instance. The DBClass creates a new statement using the Connection class createStatement() operation and formulates the correct SQL statement for the object instance that's passed in. The statement is executed and the data is removed from the database.

Dynamic View: JDBC RDBMS Initialize

JDBC RDBMS Initialize



To initialize the connection, the DBClass must load the appropriate driver by calling the DriverManager getConnection() operation with a URL, user, and password.

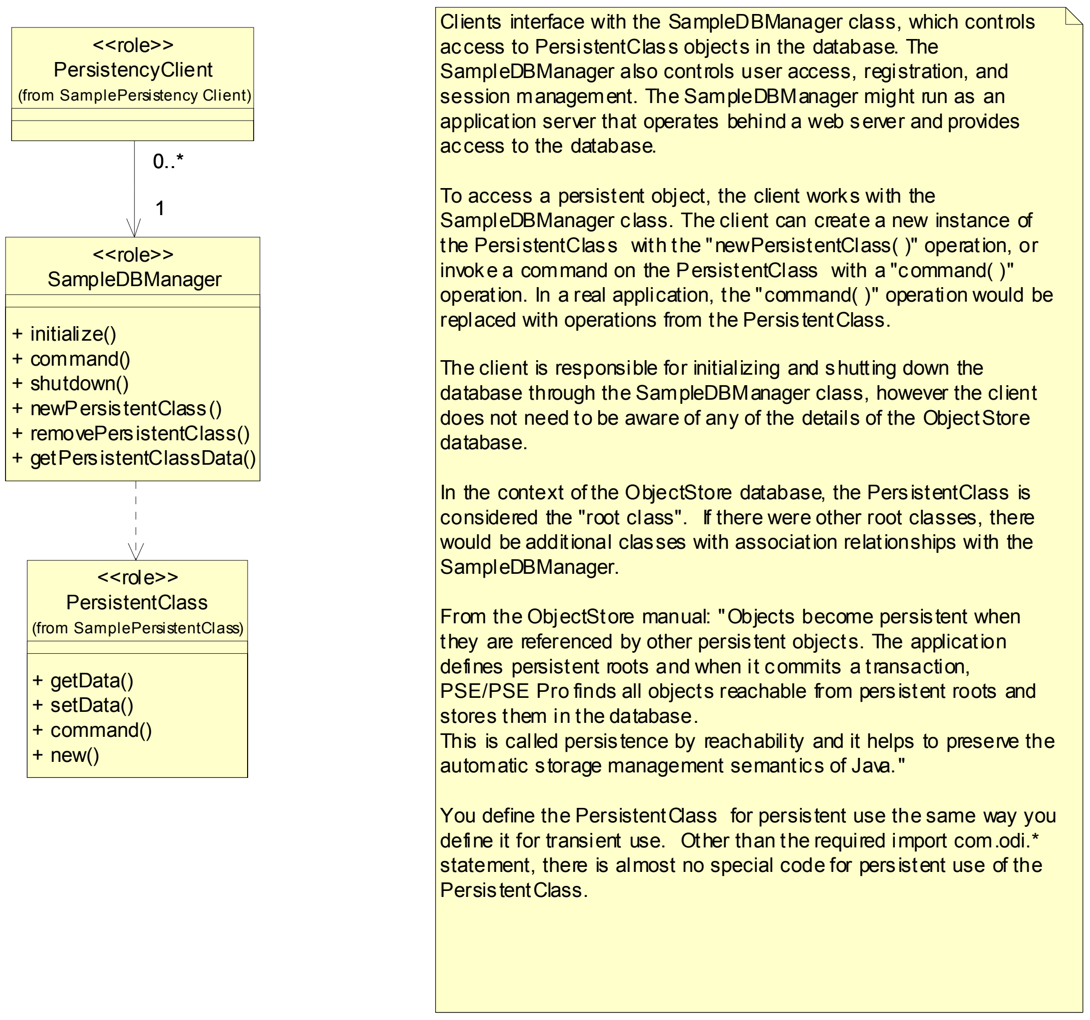
getConnection() attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.

Parameters:
url - A database url of the form jdbc:subprotocol:subname
user - The database user on whose behalf the Connection is being made
password - The user's password

Returns a Connection to the URL

Persistency - OODBMS - ObjectStore

Static View: Persistency – ObjectStore OODBMS



Clients interface with the SampleDBManager class, which controls access to PersistentClass objects in the database. The SampleDBManager also controls user access, registration, and session management. The SampleDBManager might run as an application server that operates behind a web server and provides access to the database.

To access a persistent object, the client works with the SampleDBManager class. The client can create a new instance of the PersistentClass with the "newPersistentClass( )" operation, or invoke a command on the PersistentClass with a "command( )" operation. In a real application, the "command( )" operation would be replaced with operations from the PersistentClass.

The client is responsible for initializing and shutting down the database through the SampleDBManager class, however the client does not need to be aware of any of the details of the ObjectStore database.

In the context of the ObjectStore database, the PersistentClass is considered the "root class". If there were other root classes, there would be additional classes with association relationships with the SampleDBManager.

From the ObjectStore manual: "Objects become persistent when they are referenced by other persistent objects. The application defines persistent roots and when it commits a transaction, PSE/PSE Pro finds all objects reachable from persistent roots and stores them in the database.
This is called persistence by reachability and it helps to preserve the automatic storage management semantics of Java."

You define the PersistentClass for persistent use the same way you define it for transient use. Other than the required import com.odi.* statement, there is almost no special code for persistent use of the PersistentClass.

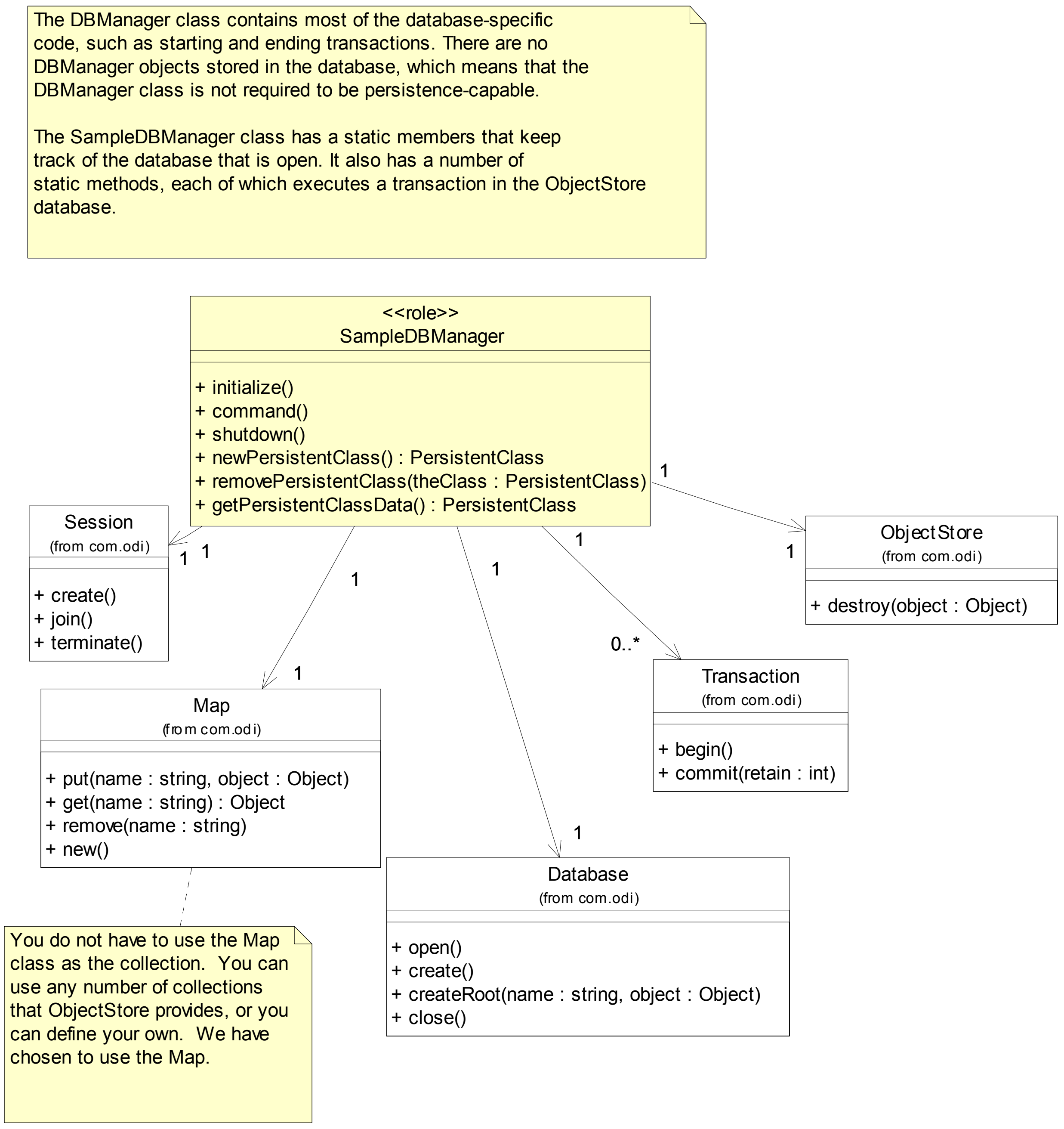
Class Descriptions

PersistencyClient : An example of a client of a persistent class.

SampleDBManager : Responsible for providing access to the persistent objects.
The SampleDBManager is an example of a class an ObjectStore user would write. It is a control class that provides a single entry point into a specific ObjectStore database. The user would add the appropriate operations to the class to access entities in the database. It is often implemented as a singleton, but doesn't have to be (if an application needs to have multiple instances of a database open at once, then each instance would have its own SampleDBManager). Both solutions would work, it just depends on how you want to do it.

PersistentClass : An example of a class that's persistent.

Static View: Persistency - DBManager Detail



The DBManager class contains most of the database-specific code, such as starting and ending transactions. There are no DBManager objects stored in the database, which means that the DBManager class is not required to be persistence-capable.
The SampleDBManager class has a static members that keep track of the database that is open. It also has a number of static methods, each of which executes a transaction in the ObjectStore database.
<<role>>
SampleDBManager
+ initialize()
+ command()
+ shutdown()
+ newPersistentClass() : PersistentClass
+ removePersistentClass(theClass : PersistentClass)
+ getPersistentClassData() : PersistentClass
Session
(from com.odi)
+ create()
+ join()
+ terminate()
ObjectStore
(from com.odi)
+ destroy(object : Object)
Transaction
(from com.odi)
+ begin()
+ commit(retain : int)
Map
(from com.odi)
+ put(name : string, object : Object)
+ get(name : string) : Object
+ remove(name : string)
+ new()
Database
(from com.odi)
+ open()
+ create()
+ createRoot(name : string, object : Object)
+ close()
1
0..*
You do not have to use the Map class as the collection. You can use any number of collections that ObjectStore provides, or you can define your own. We have chosen to use the Map.

Class Descriptions

Session : The class that represents a database session. A session must be created in order to access the database and any persistent data.

A session is the context in which PSE/PSE Pro databases are created or opened, and transactions can be executed. Only one transaction at a time can exist in a session.

Map : A persistent map container classes that stores key/value pairs.

Database : The Database class represents an ObjectStore database.

Before you begin creating persistent objects, you must create a database to hold the objects. In subsequent processes, you open the database to allow the process to read or modify the objects. To create a database, you call the static create() method on the Database class and specify the database name and an access mode.

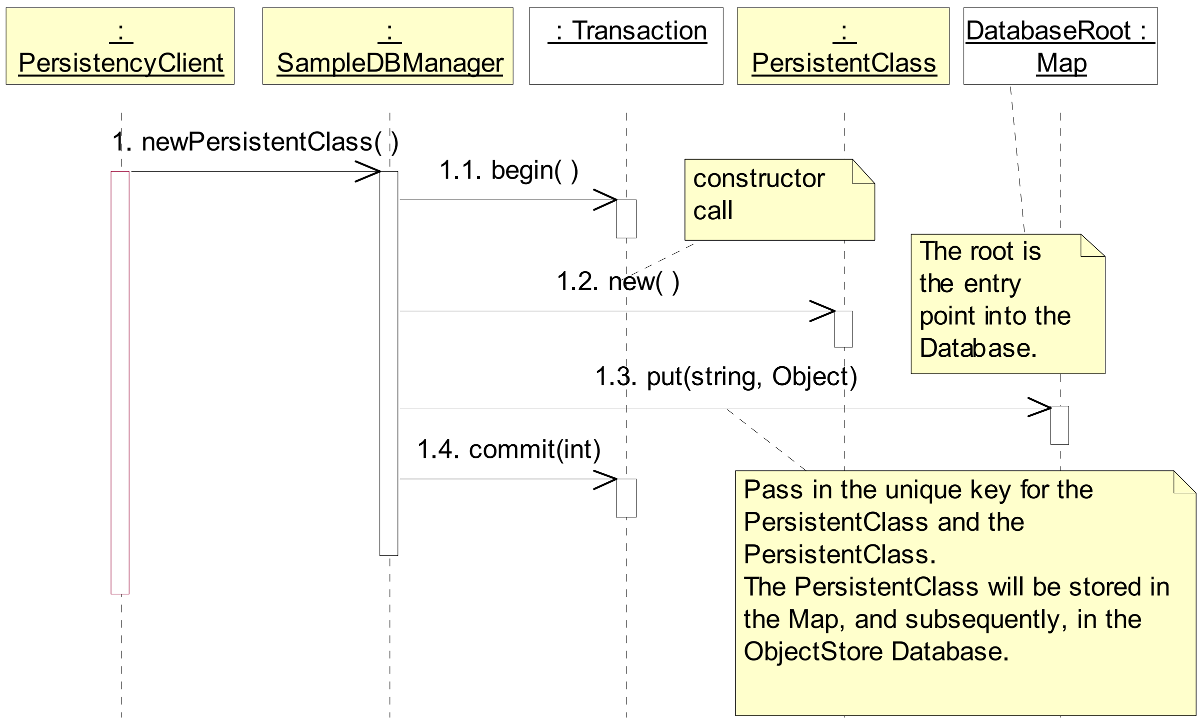
Transaction : An ObjectStore transaction. Manages a logical unit of work. All persistent objects must be accessed within a transaction.

ObjectStore : Defines system-level operations that are not specific to any database.

SampleDBManager : Responsible for providing access to the persistent objects.
The SampleDBManager is an example of a class an ObjectStore user would write. It is a control class that provides a single entry point into a specific ObjectStore database. The user would add the appropriate operations to the class to access entities in the database. It is often implemented as a singleton, but doesn't have to be (if an application needs to have multiple instances of a database open at once, then each instance would have it's own SampleDBManager). Both solutions would work, it just depends on how you want to do it.

Dynamic View: ObjectStore – OODBMS Create

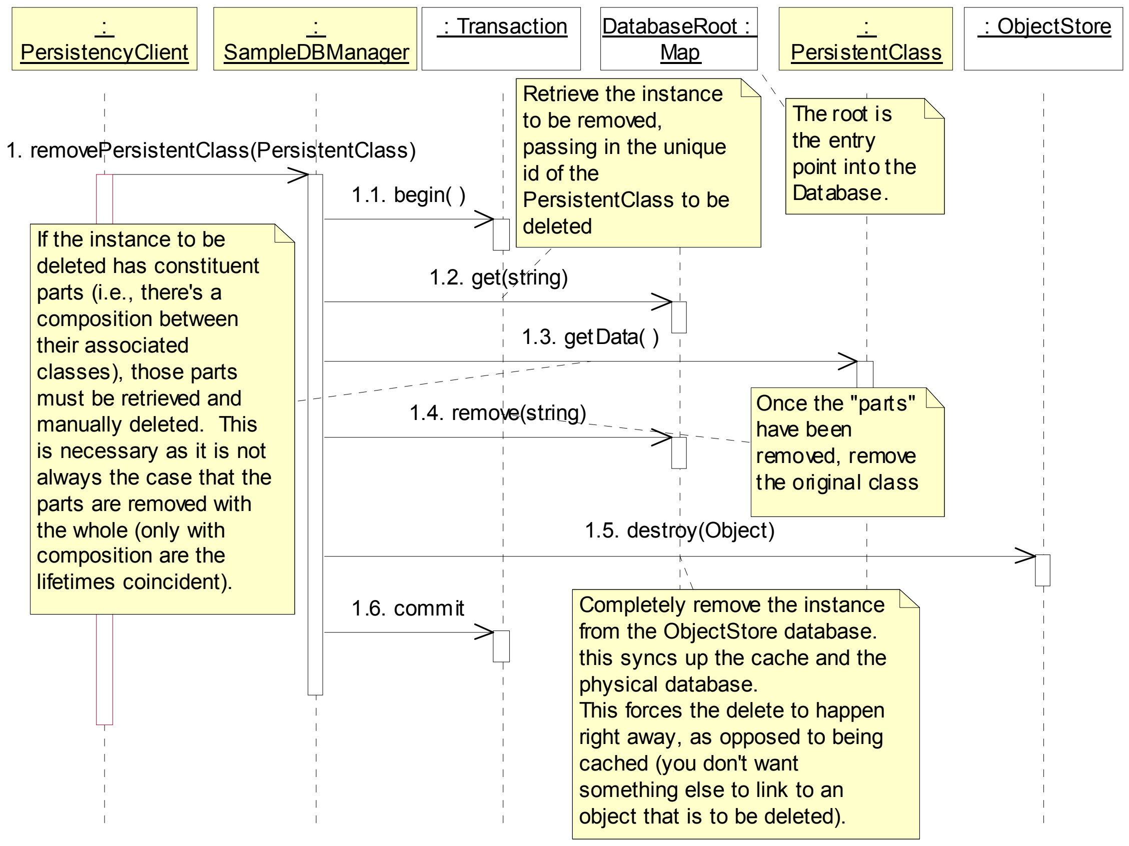
ObjectStore OODBMS Create


: PersistencyClient
: SampleDBManager
: Transaction
: PersistentClass
DatabaseRoot : Map
1. newPersistentClass( )
1.1. begin( )
constructor call
1.2. new( )
The root is the entry point into the Database.
1.3. put(string, Object)
1.4. commit(int)
Pass in the unique key for the PersistentClass and the PersistentClass.
The PersistentClass will be stored in the Map, and subsequently, in the ObjectStore Database.

To create a new instance of PersistentClass in the database, the SampleDBManager first creates a transaction and then calls the constructor for PersistentClass. Once the class has been constructed the class is added to the database via the root "put()" operation. The transaction is then committed.

Dynamic View: ObjectStore OODBMS Delete

ObjectStore OODBMS Delete

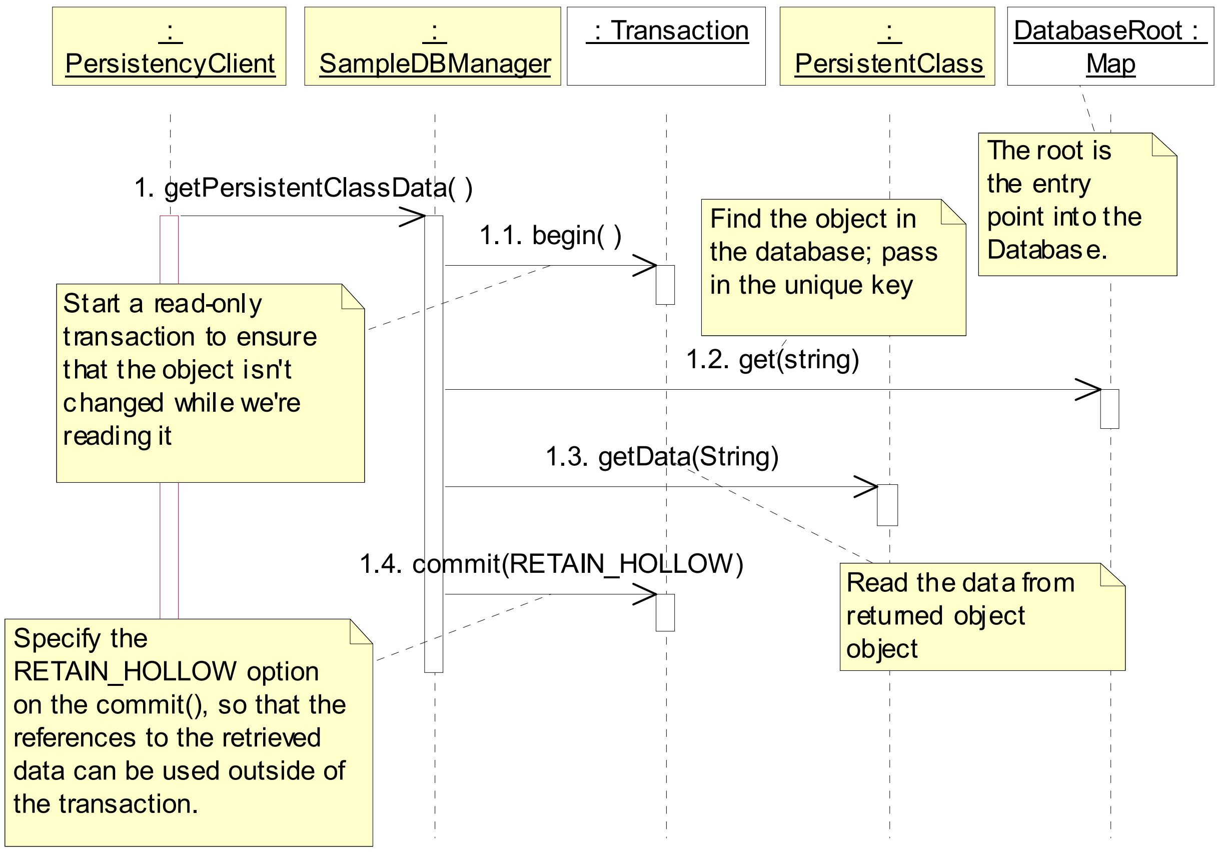

: PersistencyClient
: SampleDBManager
: Transaction
DatabaseRoot : Map
: PersistentClass
: ObjectStore
1. removePersistentClass(PersistentClass)
1.1. begin( )
1.2. get(string)
1.3. getData( )
1.4. remove(string)
1.5. destroy(Object)
1.6. commit
Retrieve the instance to be removed, passing in the unique id of the PersistentClass to be deleted
The root is the entry point into the Database.
If the instance to be deleted has constituent parts (i.e., there's a composition between their associated classes), those parts must be retrieved and manually deleted. This is necessary as it is not always the case that the parts are removed with the whole (only with composition are the lifetimes coincident).
Once the "parts" have been removed, remove the original class
Completely remove the instance from the ObjectStore database. this syncs up the cache and the physical database. This forces the delete to happen right away, as opposed to being cached (you don't want something else to link to an object that is to be deleted).

To delete an object from the database, the SampleDBManager first creates a new transaction, removes any constituent parts, and then removes the object using the database root "remove()" operation. The object is then completely removed from the ObjectStore database immediately via ObjectStore.destry (). Once the object has been removed, the transaction is committed.

Thus, in ObjectStore, delete really has two steps -- removal from the container class that is the database in memory, and removal from the physical database. that is because you want the deletion to occur right away, as opposed to being cached.

Dynamic View: ObjectStore OODBMS Read

ObjectStore OODBMS Read

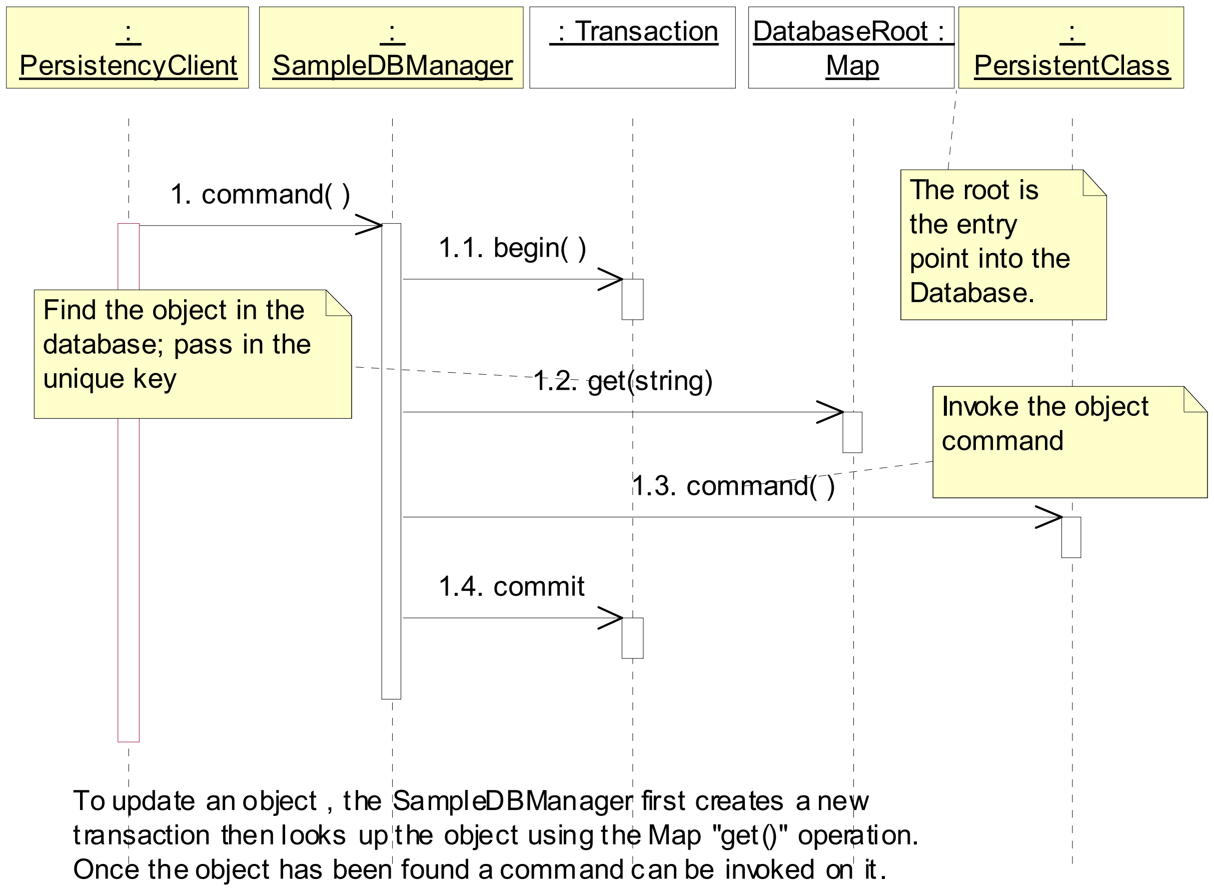

: PersistencyClient
: SampleDBManager
: Transaction
: PersistentClass
DatabaseRoot : Map
1. getPersistentClassData( )
1.1. begin( )
1.2. get(string)
1.3. getData(String)
1.4. commit(RETAIN_HOLLOW)
Start a read-only transaction to ensure that the object isn't changed while we're reading it
Find the object in the database; pass in the unique key
The root is the entry point into the Database.
Read the data from returned object object
Specify the RETAIN_HOLLOW option on the commit(), so that the references to the retrieved data can be used outside of the transaction.

To read an object , the SampleDBManager first creates a new read-only transaction then looks up the object using the Map "get()" operation. Once the object has been found it can be read with the "getData()" operation, and the transaction committed. RETAIN_HOLLOW is specified for the commit,. so the references to the object and the retrieved data can be used outside of the retrieval transaction. Once the transaction is committed the object can then be updated.

Note: Even though RETAIN_HOLLOW is specified, it does not guarantee the integrity of the reference outside of the transaction. There is still some risk that the reference could be outdated. RETAIN_HOLLOW basically says "I'm consciously taking such a risk". If that option was not used, then the references would not be available.

Dynamic View: ObjectStore OODBMS Update

ObjectStore OODBMS Update

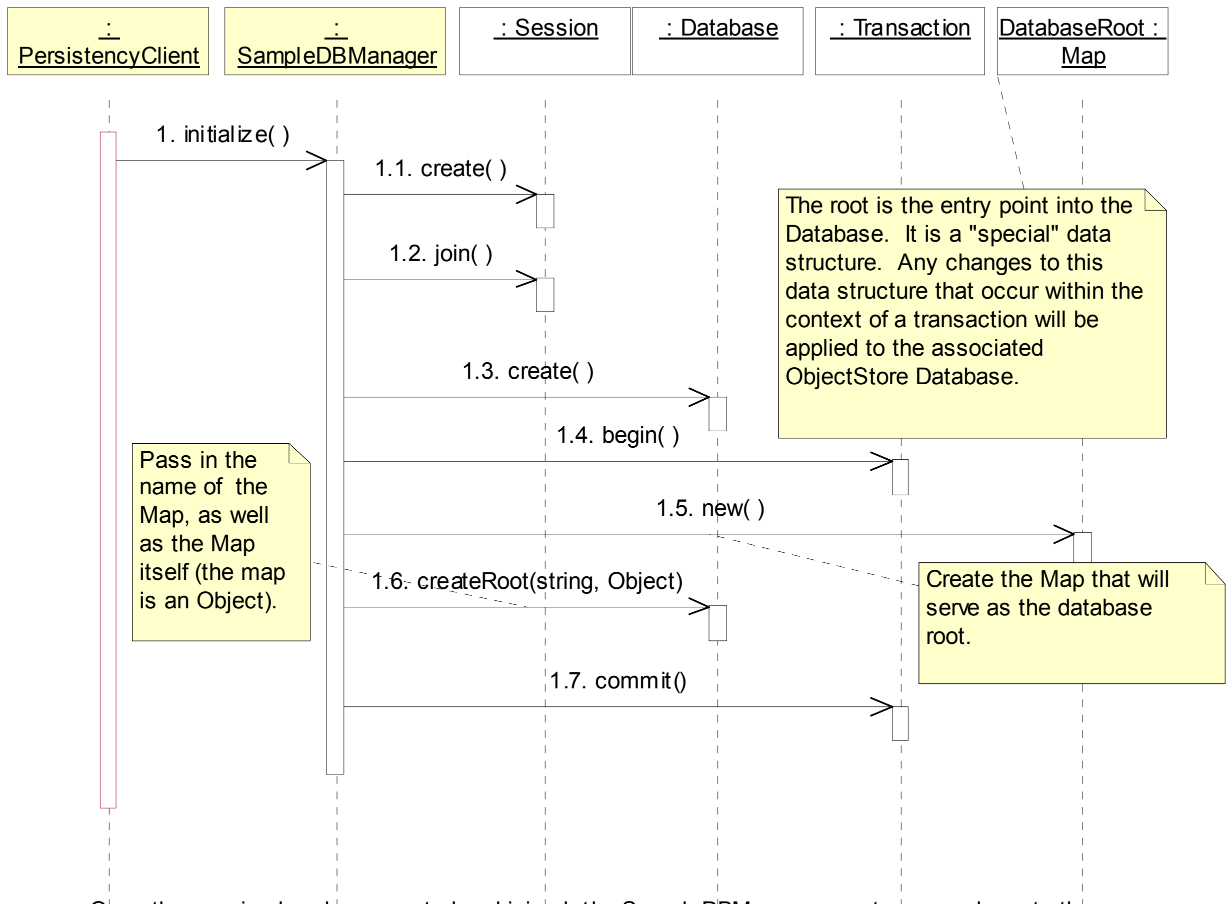


To update an object , the SampleDBManager first creates a new transaction then looks up the object using the Map "get()" operation. Once the object has been found a command can be invoked on it. When the command is complete the transaction is committed.

A separate put() to the Map is not necessary as the get() operation returns a reference to the persistent object and any changes to that object, if made in the context of a transaction, are automatically committed to the database.

Dynamic View: ObjectStore OODBMS Initialize

ObjectStore OODBMS Initialize


: PersistencyClient
: SampleDBManager
: Session
: Database
: Transaction
DatabaseRoot : Map
1. initialize( )
1.1. create( )
1.2. join( )
1.3. create( )
1.4. begin( )
1.5. new( )
1.6. createRoot(string, Object)
1.7. commit()
The root is the entry point into the Database. It is a "special" data structure. Any changes to this data structure that occur within the context of a transaction will be applied to the associated ObjectStore Database.
Pass in the name of the Map, as well as the Map itself (the map is an Object).
Create the Map that will serve as the database root.

Once the session has been created and joined, the SampleDBManager must open and create the new database.

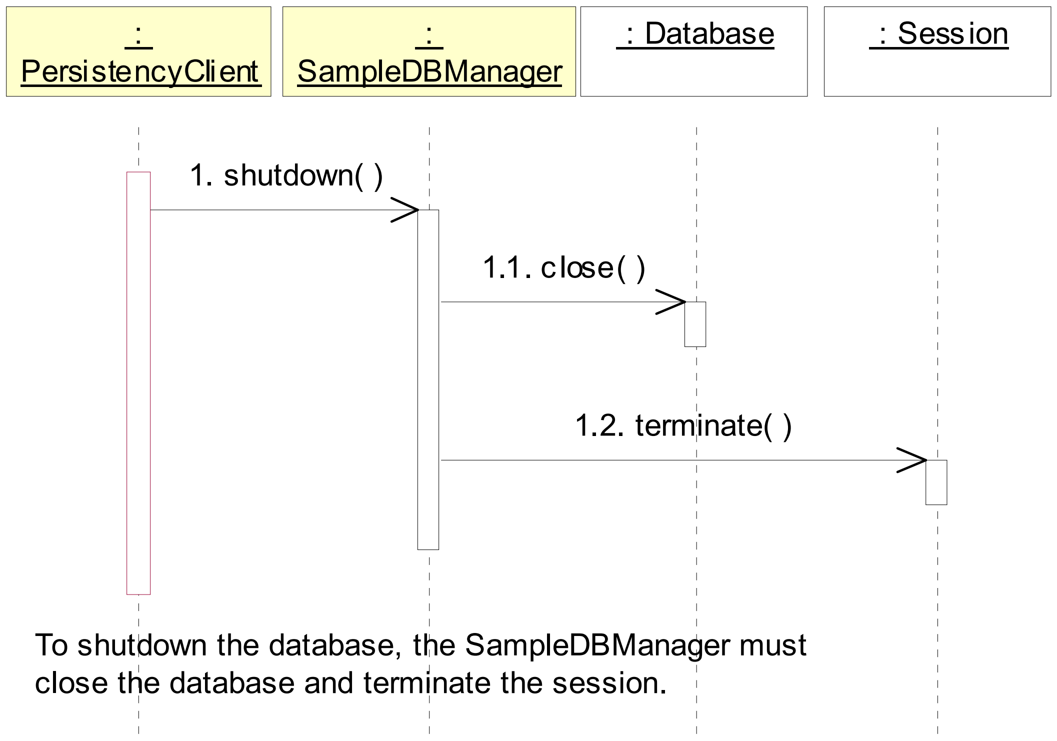
To create the database, the SampleDBManager creates a new transaction and creates the "root" of the database with the "createRoot()" operation.

The root is the entry point into the Database (the root class is the top-level class in the object database). It is a "special" data structure (in the above example, a Map that contains instances of the root class and all “reachable” classes). Any changes to this data structure that occur within the context of a transaction will be applied to the associated ObjectStore Database. There may be multiple database roots.

Once the root has been created, the transaction is committed.

Dynamic View: ObjectStore OODBMS Shutdown

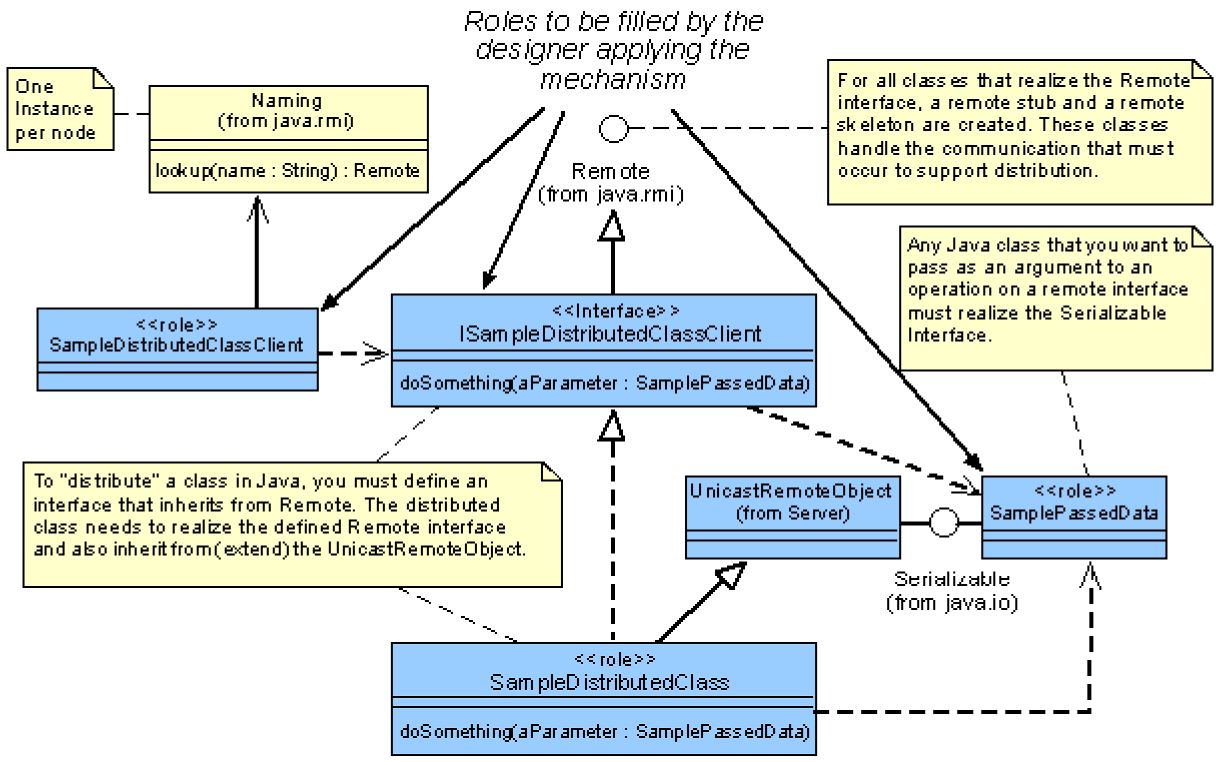
ObjectStore OODBMS Shutdown



To shutdown the database, the SampleDBManager must close the database and terminate the session.

Distribution - RMI

Static View: Distribution - RMI


Roles to be filled by the designer applying the mechanism
One Instance per node
Naming (from java.rmi)
lookup(name : String) : Remote
For all classes that realize the Remote interface, a remote stub and a remote skeleton are created. These classes handle the communication that must occur to support distribution.
Remote (from java.rmi)
Any Java class that you want to pass as an argument to an operation on a remote interface must realize the Serializable Interface.
<<role>> SampleDistributedClassClient
<<Interface>> ISampleDistributedClassClient
doSomething(aParameter : SamplePassedData)
To "distribute" a class in Java, you must define an interface that inherits from Remote. The distributed class needs to realize the defined Remote interface and also inherit from (extend) the UnicastRemoteObject.
UnicastRemoteObject (from Server)
<<role>> SamplePassedData
Serializable (from java.io)
<<role>> SampleDistributedClass
doSomething(aParameter : SamplePassedData)

Class Descriptions

Naming. :
 * This is the bootstrap mechanism for obtaining references to remote
 * objects based on Uniform Resource Locator (URL) syntax. The URL
 * for a remote object is specified using the usual host, port and
 * name:
 *<br>   rmi://host:port/name
 *<br>   host = host name of registry (defaults to current host)
 *<br>   port = port number of registry (defaults to the registry port number)
 *<br>   name = name for remote object

SampleDistributedClass : An example of a class that's distributed.

Remote :
 * The Remote interface serves to identify all remote objects.
 * Any object that is a remote object must directly or indirectly implement
 * this interface. Only those methods specified in a remote interface are
 * available remotely. <p>
 * Implementation classes can implement any number of remote interfaces
 * and can extend other remote implementation classes.

For all classes that realize the Remote interface, a remote stub and a remote skeleton are created. These classes handle the communication that must occur to support distribution.

SampleDistributedClassClient : An example of a client of a distributed class.

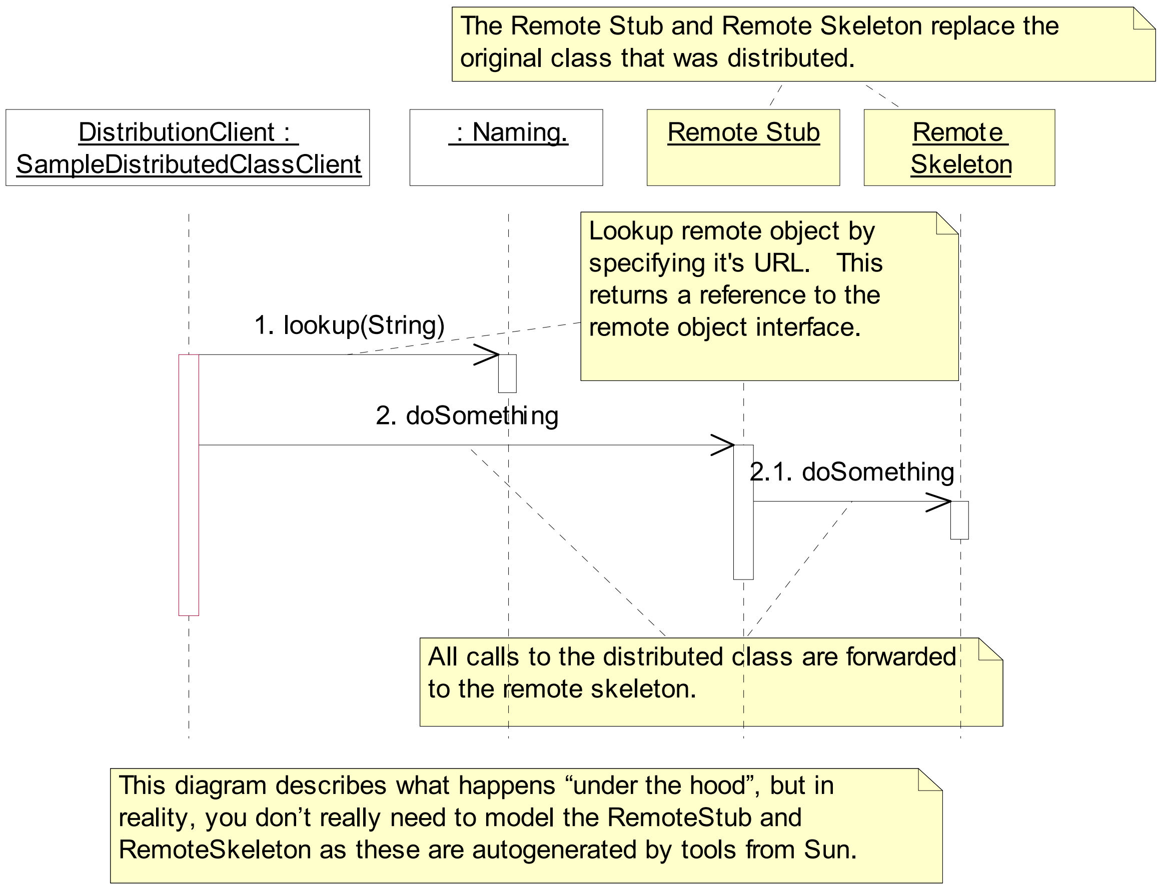
SamplePassedData : An example of data that is passed to/from a distributed class.

UnicastRemoteObject :

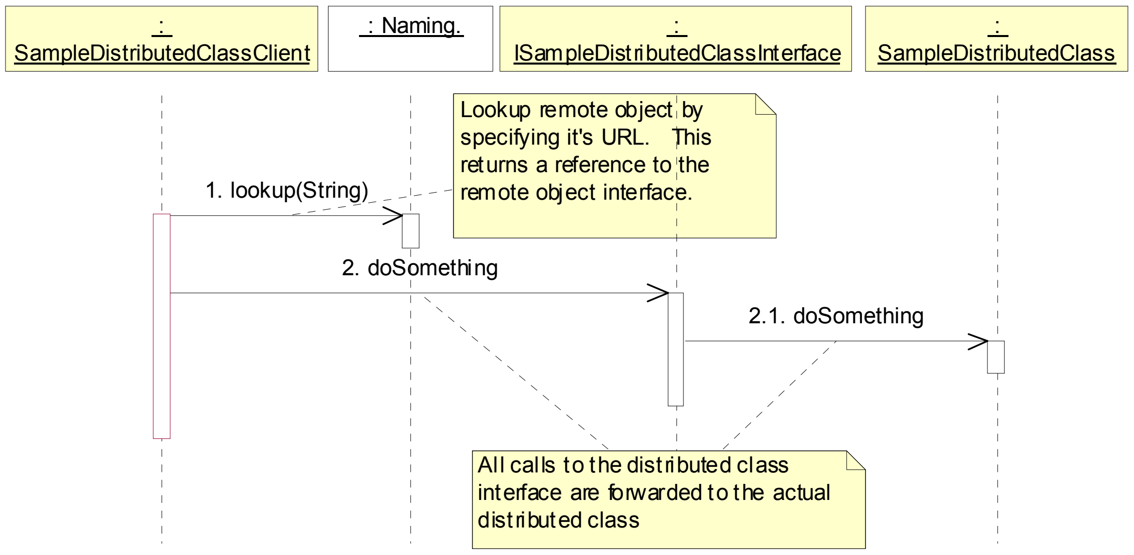
ISampleDistributedClassInterface : An example of an interface defined for a distributed class.

Serializable : Any Java class that you want to pass as an argument to an operation on a remote interface must realize the Serializable interface.

Dynamic View: Set Up Remote Connection (details)


The Remote Stub and Remote Skeleton replace the original class that was distributed.
DistributionClient : SampleDistributedClassClient
: Naming.
Remote Stub
Remote Skeleton
Lookup remote object by specifying it's URL. This returns a reference to the remote object interface.
1. lookup(String)
2. doSomething
2.1. doSomething
All calls to the distributed class are forwarded to the remote skeleton.
This diagram describes what happens "under the hood", but in reality, you don't really need to model the RemoteStub and RemoteSkeleton as these are autogenerated by tools from Sun.

Dynamic View: Set Up Remote Connection


: SampleDistributedClassClient
: Naming.
: ISampleDistributedClassInterface
: SampleDistributedClass
Lookup remote object by specifying it's URL. This returns a reference to the remote object interface.
1. lookup(String)
2. doSomething
2.1. doSomething
All calls to the distributed class interface are forwarded to the actual distributed class

Logical View

Architectural Analysis

Upper-Level Layers

- Application layer
- Business Services layer

Upper-Level Layer Dependencies

- The Application layer depends on the Business Services layer

Architectural Design

Incorporating ObjectStore

For the Payroll System, a single root class has been chosen -- Employee.

The selected container is the Map, where the unique key to access the Employees is EmployeeID.

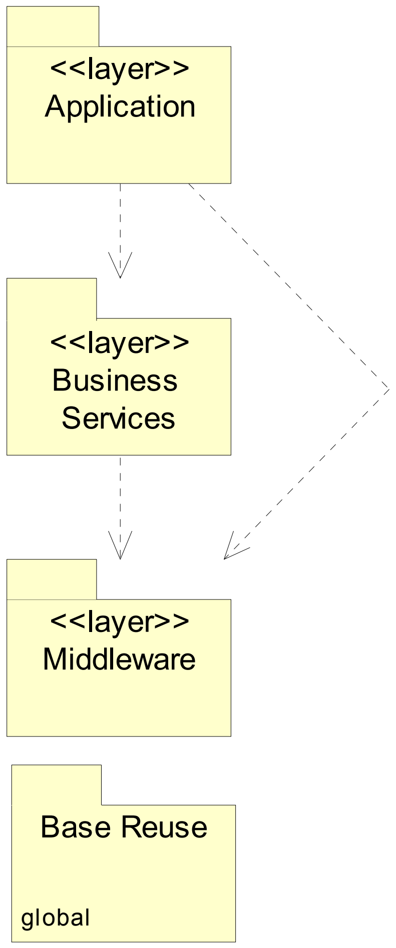
There is one DBManager class per ObjectStore database instance. For the Payroll System, there is one ObjectStore database, the Payroll Database, that contains employee information, including timecard, purchase order, and paycheck information. Thus, there is one PayrollDBManager that exists in the new OODBMS Support package.

For the ObjectStore persistency mechanism, the DBManager class includes operations to access the OODBMS persistent entities in the database. For the PayrollDBManager class, operations have been added to access Employee, Timecard, Purchase Order, and Paycheck information since that is required for the core system functionality.

During Identify Design Mechanisms, the architect provides guidance to the designers and makes sure that the architecture has the necessary infrastructure to support the mechanism. Thus, the PayrollDBManager and the supporting architectural packages and relationships (OODBMS Support) have been defined in Identify Design Mechanisms. However, the development of the interaction diagrams that describe these operations and where they fit into the existing use-case realizations has been deferred until detailed design (e.g., Use-Case and Subsystem Design).

The following diagram demonstrates the operations that have been defined for the PayrollDBManager during Identify Design Mechanisms:

Architectural Layers and Their Dependencies: Main Diagram



Layer Descriptions

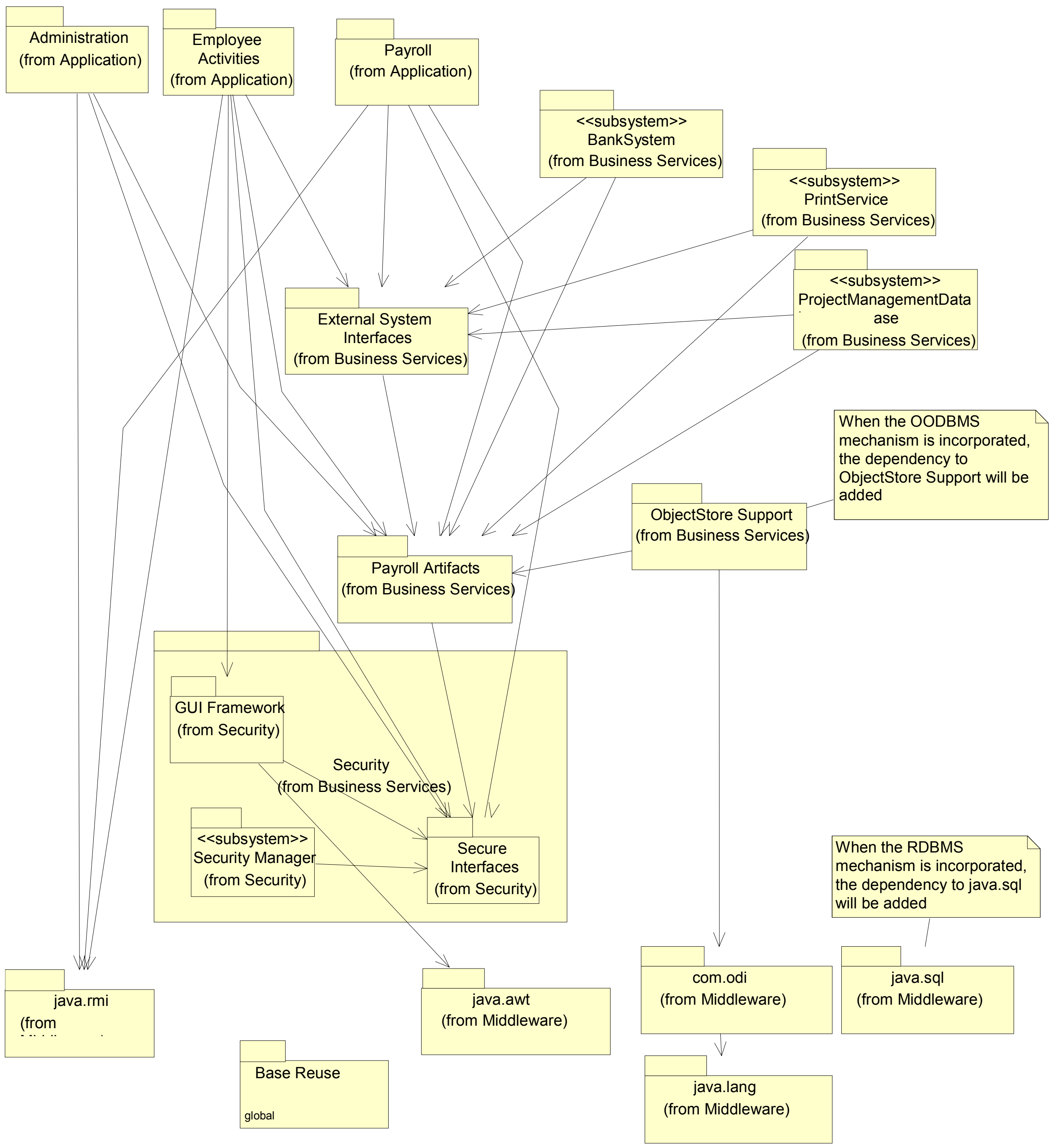
Application Layer: The Application layer contains application-specific design elements.

Business Services Layer: The Business Services layer contains business-specific elements that are used in several applications.

Base Reuse : Basic reusable design elements.

Middleware Layer: Provides utilities and platform-independent services.

Packages and Their Dependencies: Package Dependencies Diagram


Administration (from Application)
Employee Activities (from Application)
Payroll (from Application)
<<subsystem>> BankSystem (from Business Services)
<<subsystem>> PrintService (from Business Services)
<<subsystem>> ProjectManagementDatabase (from Business Services)
External System Interfaces (from Business Services)
When the OODBMS mechanism is incorporated, the dependency to ObjectStore Support will be added
ObjectStore Support (from Business Services)
Payroll Artifacts (from Business Services)
GUI Framework (from Security)
Security (from Business Services)
<<subsystem>> Security Manager (from Security)
Secure Interfaces (from Security)
When the RDBMS mechanism is incorporated, the dependency to java.sql will be added
java.rmi (from
java.awt (from Middleware)
com.odi (from Middleware)
java.sql (from Middleware)
Base Reuse
global
java.lang (from Middleware)

Package Descriptions

Employee Activities : Contains the design elements that support the Employee's applications.

Administration : Contains the design elements that support the Payroll Administrator's applications.

Payroll : Contains the design elements that support the execution of the payroll processing.

Payroll Artifacts : Contains the core payroll abstractions.

BankSystem Subsystem: Encapsulates communication with all external bank systems.

External System Interfaces : Contains the interfaces that support access to external systems. This is so that the external system interface classes can be version controlled independently from the subsystems that realize them.

PrintService Subsystem: Provides utilities to produce hard-copy.

ProjectManagementDatabase Subsystem: Encapsulates the interface to the legacy database containing information regarding projects and charge numbers.

java.awt : The java.awt package contains the basic GUI design elements for java.

com.odi : The com.odi package contains the design elements that support the OODBMS persistency mechanism. The name of the package in the model reflects the naming convention for 3rd party Java software. The convention is to use the reverse of the domain name, so if Rational had a Java package called "util" they'd call it "com.rational.util". This com.odi has nothing to do with Microsoft COM/DCOM; they are totally separate. There is nothing COM/DCOM related when using CORBA, RMI, or ObjectStore.

Base Reuse : Basic reusable design elements.

java.lang : The package contains some basic java design elements.

Security : Contains design elements that implement the security mechanism.

GUI Framework : This package comprises a whole framework for user interface management.

It has a ViewHandler that manages the opening and closing of windows, plus window-to-window communication so that windows do not need to depend directly upon each other.

This framework is security-aware, it has a login window that will create a server-resident user context object. The ViewHandler class manages a handle to the user context object.

The ViewHandler also starts up the controller classes for each use case manager.

Secure Interfaces : Contains the interfaces that provide clients access to security services.

Security Manager Subsystem: Provides the implementation for the core security services.

ObjectStore Support : Contains the business-specific design elements that support the OODBMS persistency mechanism. This includes the DBManager. The DBManager class must contain operations for every OODBMS persistent class.

java.rmi : The java.rmi package contains the classes that implement the RMI distribution mechanism. This package is commercially available with most standard JAVA IDEs.

java.sql : The package that contains the design elements that support RDBMS persistency.

Process View

Processes

The processes for the Payroll System will be the following:

One process per major interface or family of forms (e.g. EmployeeApplication):

- EmployeeApplication: Controls the interface of the Employee application. Controls the family of forms that the employee uses.

There is one process per major interface because these are now seen as separate, mutually exclusive applications that will run concurrently with each other.

One process per business service controller:

- PayrollControllerProcess
- TimecardControllerProcess

There is one process per controller because these activities will need to run concurrently with each other.

One process per external system:

- ProjectManagementDBAccess
- BankSystemAccess
- PrinterAccess

There is one process per external system. These processes manage access to those systems. Such access may be slow, so this allows other functionality to continue while the external system processes wait on the external system. These processes also synchronize access to the external systems from the other system processes.

To further improve throughput and turnaround, a Bank Transaction thread was defined to allow multiple accesses to the Bank System to occur concurrently. Each time a transaction needs to be sent to the Bank System, a different thread is used. The Bank Transaction thread will run in the context of the Bank System Access process.

In general, the above processes and threads were defined to support faster response times and take advantage of multiple processors.

Design Element to Process Mapping

- The classes associated with the individual user interfaces should be mapped to those processes.
- The classes associated with the individual business services should be mapped to those processes.
- The classes associated with access to the external systems should be mapped to those processes.

Deployment View

Nodes and Connections

The nodes of the physical architecture for the Payroll System are the following:

- Desktop PCs (processors)
- Payroll Server (processor)
- Bank System (processor)
- Project Management Database (processor)
- Printers (devices)

- The Desktop PCs are connected to the Payroll Server via the Company LAN
- The Printers are connected to the Payroll Server via the Company LAN
- The Payroll Server is connected to the external Bank System via the Internet.
- The Payroll Server is connected to the ProjectManagementDatabase via the Company LAN

Process-to-Node Map

The following processes run on the Desktop PCs:

- EmployeeApplication

The following processes run on the Payroll Server:
- PayrollControllerProcess
- TimecardControllerProcess
- ProjectManagementDBAccess
- BankSystemAccess
- PrinterAccess