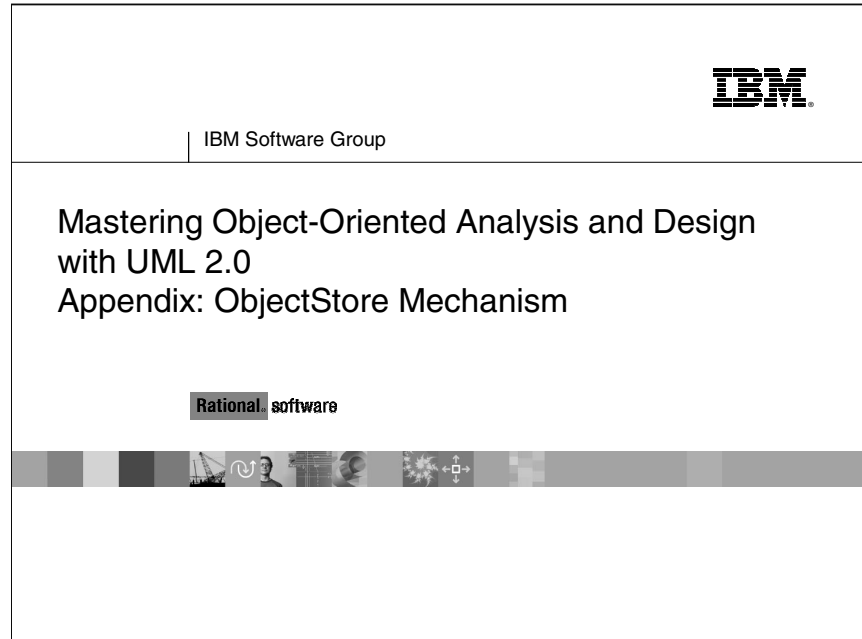


Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:



Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Note: If your students are interested in the OODBMS (ObjectStore) mechanism, these slides should be inserted while presenting the persistency mechanisms in the Identify Design Mechanisms module.

Identify Design Mechanisms Slides

The following slides can be inserted during the Identify Design Mechanisms module

2



Mastering OOAD w/ UML 2.0 – Instructor Notes

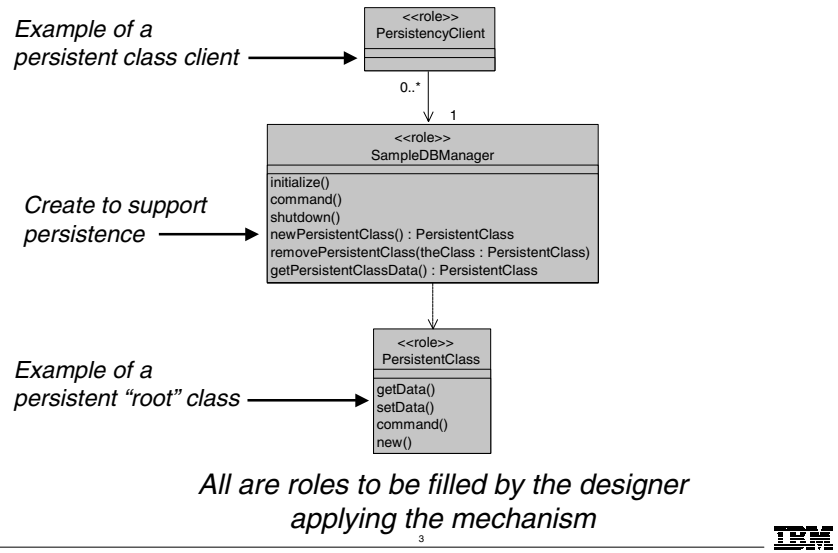
Instructor Notes:

The operations defined within SampleDBManager are specific to the classes that are persistent (specifically, the “root classes”). Every persistent class operation that updates the class data should be replicated in the SampleDBManager class. For example, if there is a class called MyClass that contains 2 operations: op1() and op2, the SampleDBManager class will need to include the following operations: NewMyClass(), UpdateMyClass(), DeleteMyClass(), doMyClassOp1(), etc.

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." (PSE stands for Persistent Storage Engine).

With OODBMS, you store and retrieve complete (association/aggregation) structure s, and not just individual objects. This is the crucial difference between OODBMS and RDBMS.

Example: Persistency: OODBMS: ObjectStore



The next few slides demonstrate the pattern of use of the persistent mechanism chosen for the OODBMS classes in our example: ObjectStore. The `<<role>>` stereotype is used to which classes are roles that should be filled by the designer as they apply the mechanism.

The SampleDBManager provides a single entry point into a specific ObjectStore database. There is one SampleDBManager class per ObjectStore database instance. Clients interface with the SampleDBManager class to get access to PersistentClass objects in the database. The client can create a new PersistentClass instance with the "newClass()" operation, or invoke a PersistentClass command with a "command()" operation (the "command()" operation is replaced with operations from the PersistentClass). The client initializes and shuts down the ObjectStore database through the SampleDBManager class, but does not need to be aware of any of the database details.

In the context of the ObjectStore database, the PersistentClass is considered the "root class".

You define the PersistentClass for persistent use the same way you define it for transient use. Other than the required "import com.odj.*" statement, there is almost no special code for persistent use of the PersistentClass. Once you import the ObjectStore classes, a special post-processor modifies the Java byte code to handle all the persistence. No other changes are required.

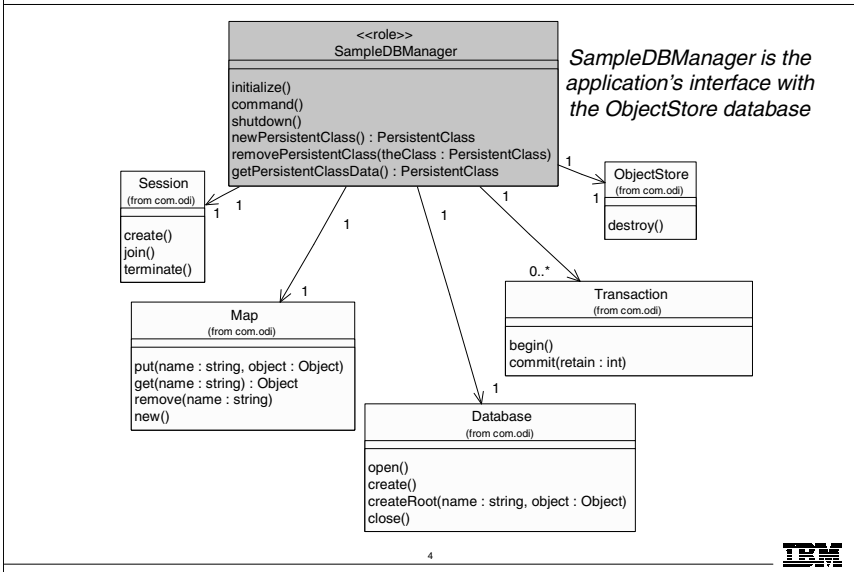
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The classes to be replaced by concrete classes by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals). There are no SampleDBManager objects stored in the database, which means that the SampleDBManager class is not required to be persistence-capable. ObjectStore has it's own back-end database that it accesses. This is all “behind the scenes” and doesn't need to modeled here.

In ObjectStore, you can have one or more databases in which you store stuff. Essentially, the database is a glorified file. When you connect to a given database, you are essentially opening up that file. Of course in a client/server context the “file” is actually over on the server. Object is the class in Java that every other class inherits from.

Example: Persistency: OODBMS: ObjectStore: SampleDBManager



The above diagram provides a more detailed look at the SampleDBManager class that must be defined by the designer (hence the <<role>> stereotype). The **SampleDBManager** class contains most of the database-specific code, such as starting and ending transactions. The SampleDBManager class has 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.

The **Database** class is an ObjectStore class that represents the actual ObjectStore database. Before you begin creating persistent objects, you must create a database to hold the objects. An application can access more than one Database instance at a time.

The **Map** is a persistent map container classes that stores key/value pairs with a unique key. 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.

The **ObjectStore** class defines system-level operations that are not specific to any database.

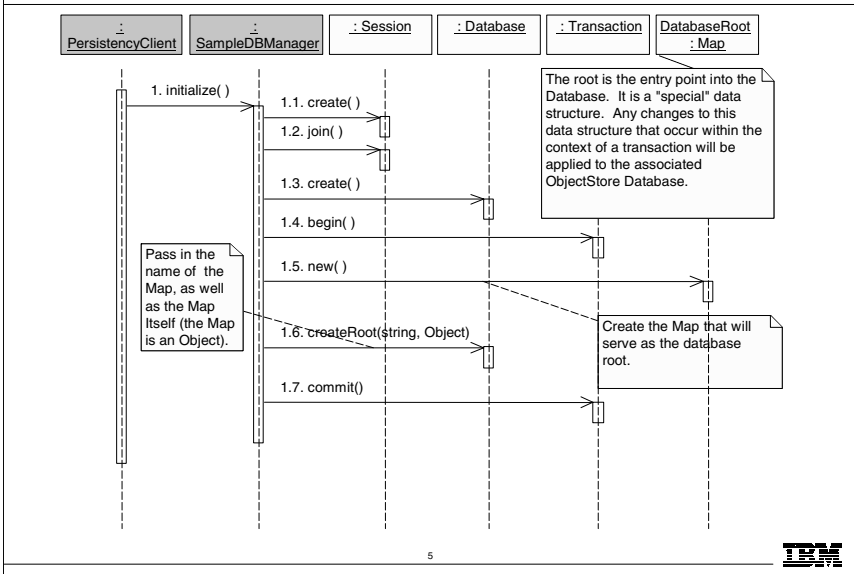
A database **Session** must be created in order to access the database and any persistent data. A session is the context in which ObjectStore databases are created or opened, and in which transactions are executed. All persistent objects must be accessed within an ObjectStore **Transaction**. Only one transaction at a time can exist in a session.

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Initialize



Initialize must occur before any persistent class can be accessed.

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. 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.

In the above example, a Map was chosen. It will contain instances of the root class and all "reachable" classes.

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

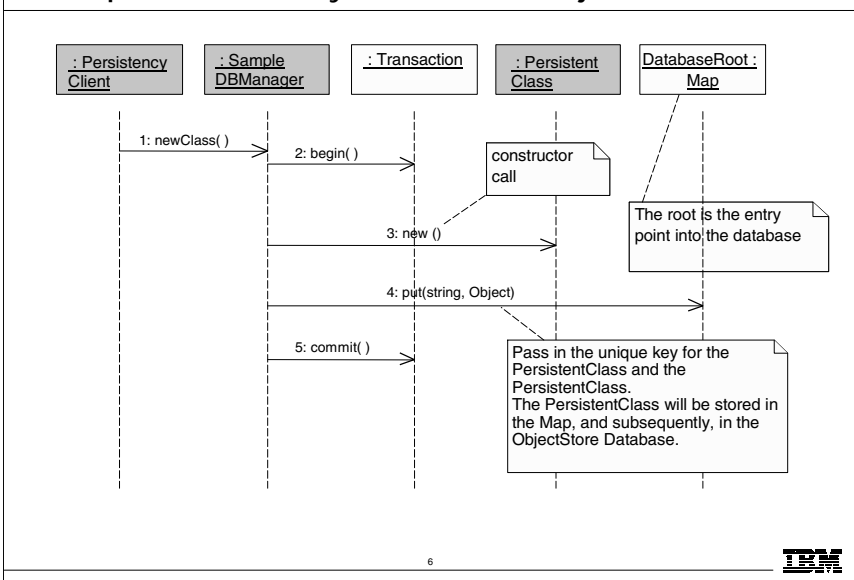
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Spend the most time on this interaction diagram, as this one that will be applied in Use-Case Design.

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Create



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 "put()" operation. The transaction is then committed.

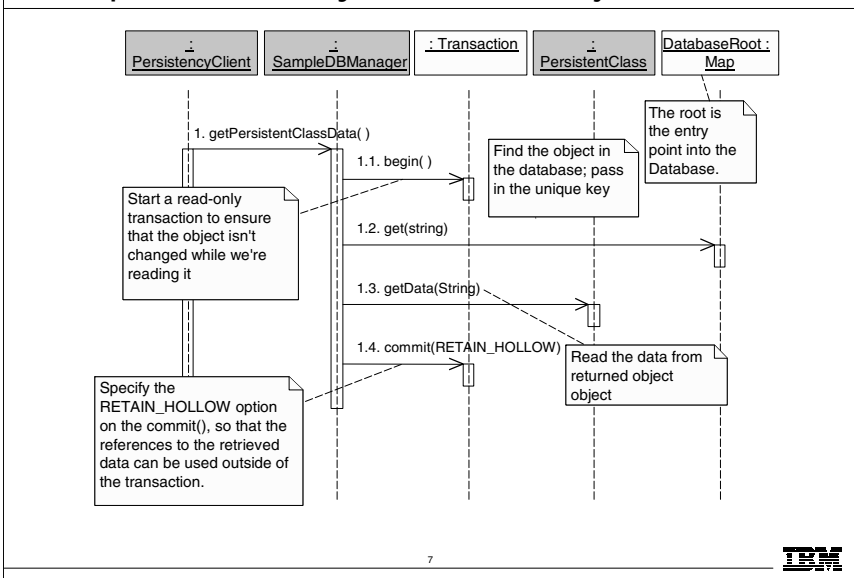
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Don't spend too much time on this diagram as it will not be applied in later modules. However, it is needed in order to better understand the persistency mechanism.

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Read



To read an object, the SampleDBManager first creates a new read-only transaction then looks up the object using the database root "get()" operation. Once the object has been found it can be read with the "getData()" method and the transaction committed. Once the transaction is committed the object can then be updated.

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.

Other possible values for the retain parameter include the following: ObjectStore.RETAIN_STALE, ObjectStore.RETAIN_HOLLOW, ObjectStore.RETAIN_READONLY, ObjectStore.RETAIN_UPDATE, and ObjectStore.RETAIN_TRANSIENT.

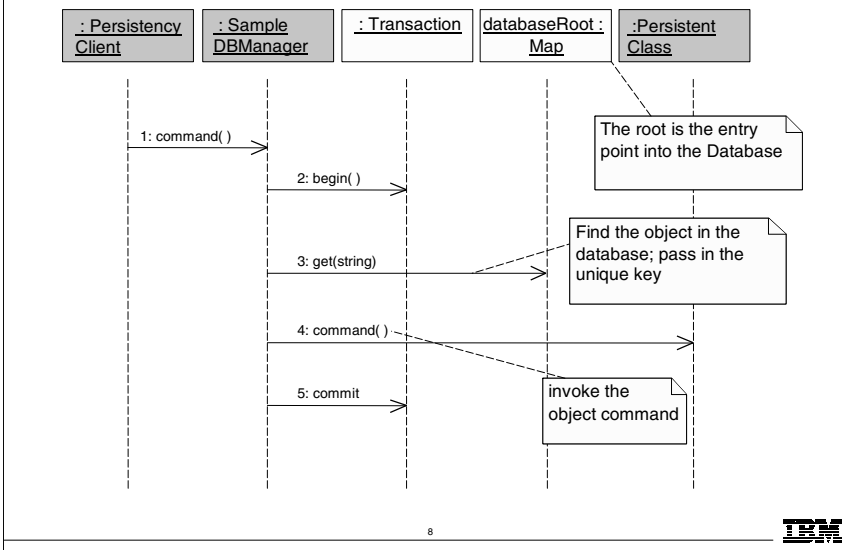
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Spend the most time on this interaction diagram, as this one that will be applied in Use-Case Design.

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Update



To update an object, the SampleDBManager first creates a new transaction then looks up the object using the database root "get()" operation (the string parameter is the unique ID for the class associated with the invoked SampleDBManager command(); knowing what this is the SampleDBManager's responsibility). Once the object has been found a command can be invoked on it (the command() operation shown above is just a placeholder for some command you might do on the PersistentClass). 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.

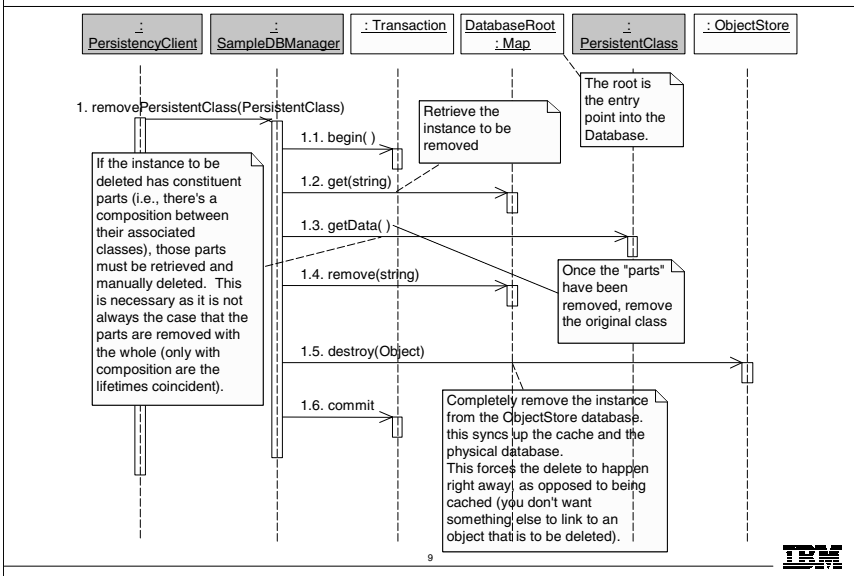
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Don't spend too much time on this diagram as it will not be applied in later modules. However, it is needed in order to better understand the persistency mechanism.

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Delete



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.destroy()`. 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.

Note: In case you are wondering why there is a retrieve from the database prior to deleting the object (for example, message 1.2 `get(string)`). This is necessary for the following reason: If the transaction that obtained the reference to the `PersistentClass` was committed with `RETAIN_HOLLOW`, then we could just use the reference that's passed in. If not, then we have to get a fresh reference from the database based on the parameter passed into the delete operation.

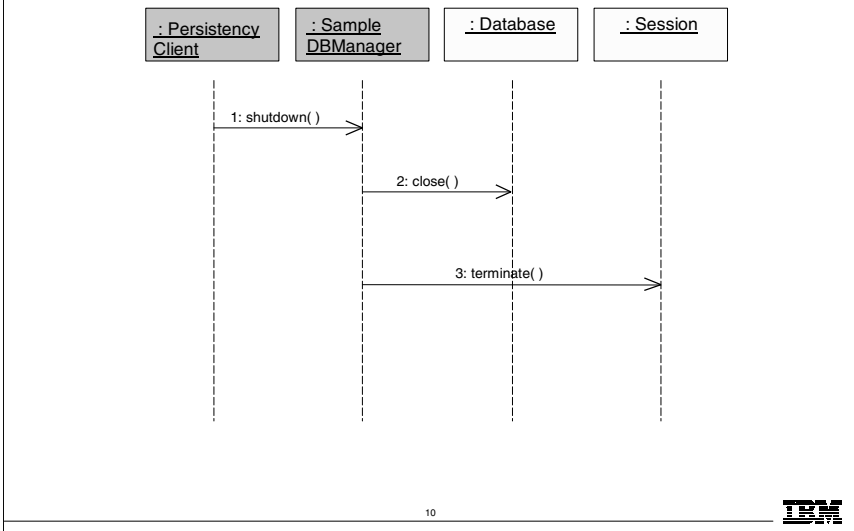
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Don't spend too much time on this diagram as it will not be applied in later modules. However, it is needed in order to better understand the persistency mechanism.

The classes to be replaced by concrete objects by the designer applying the mechanism are shown in blue (but this does not show up in the black and white manuals).

Example: Persistency: OODBMS: ObjectStore: Shutdown



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

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Note: On the presented slide, the italicized text is also blue, but this does not show up in the back-and-white manuals.

More information on ObjectStore roots:

Think of an object model, like the one for course registration. A roster may contain many Students, each Student has at least one Schedule, and each schedule has a bunch of CourseOfferings. When this is implemented there's several collection class needed for all the 1-to-many relations. A "root" is nothing but an access point for the database, a way to query the database for information. In the above example, we could choose many roots: Roster, Student, and Schedule. When making a root, all you're doing is saying "I have this particular collection class in my persistent object model and I want to make it an entry point for the database so I can do queries on it." So the root **is** the collection class with some special abilities in the sense that it supports ObjectStore queries.

Incorporating ObjectStore: Steps

- ♦ Provide access to the class libraries needed to implement ObjectStore access
 - *Dependency on com.odi*
- ♦ Select the database root class(es)
 - *Student class*
- ♦ Select the container class(es) that will serve as the database root(s) (contains the selected root class(es))
 - *Map (from com.odi)*
 - *Key will be Student ID*

11



The above is a summary of the steps that can be used to implement the OODBMS Persistency mechanism (ObjectStore). The italicized text describes the architectural decisions made with regards to Objectstore for our Course Registration example.

One of the most important decisions that must be made when incorporating ObjectStore into your design is what the root class(es) are going to be, and what container class will serve as the database root, where these classes will be stored. Remember the database root (the selected container class) is a special data structure. It is associated with a specific ObjectStore database (see the Initialize interaction diagram provided earlier in this module). Any changes to this "special" data structure that occur within the context of a transaction will be applied to the associated ObjectStore Database. There may be multiple database roots

For the Course Registration System, a single root class was chosen -- Student.

The selected container was the Map, where the unique key to access the Students will be StudentID.

The remaining steps are described on the following slide.

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

This slide is a continuation of the incorporating ObjectStore steps started on the previous slide. The italicized text describes the architectural decisions made with regards to JDBC for our Course Registration example.

Note: On the presented slide, the italicized text is also blue, but this does not show up in the back-and-white manuals.

Incorporating ObjectStore: Steps (continued)

- ♦ Create a DBManager (one per ObjectStore database instance)
 - *Single Course Registration Database => CourseRegDBManager*
 - *Will “live in” ObjectStore Support package*
- ♦ Add operations to DBManager to access entities in the OODBMS
 - *Create operations for Student and Schedule*
- ♦ Create/Update interaction diagrams that describe:
 - Database initialization and shutdown
 - Persistent class access: Create, Read, Update, Delete
- ♦ Implement persistent classes
 - Add “import com.odj.*” statement
 - *Implementer to include this statement*

Deferred

Out of scope

12



There is one DBManager class per ObjectStore database instance. For the Course Registration System example, there will be one ObjectStore database, the Course Registration Database, that contains student (and schedule) information for the university. Thus, there will be one CourseRegDBManager that will be in a new ObjectStore Support package.

Operations must be added to the DBManager class to access the OODBMS persistent entities in the database. For the CourseRegDBManager class, operations will be added to access Student and Schedule information since that is required for the core system functionality.

The PersistentClass is defined for persistent use the same way it is defined for transient use, with the addition of the required “import com.odj.*” statement.. Once you import the ObjectStore classes, a special post-processor modifies the Java byte code to handle all the persistence. Implementing the persistent classes is out of the scope of this course (it is part of Implementation, not Analysis and Design).

At this point, the architect provides guidance to the designers and makes sure that the architecture has the necessary infrastructure to support the mechanism. Thus, the DBManager and the supporting architectural packages are defined, but the actual incorporation of the mechanism and the development of the detailed interaction diagrams are deferred until detailed design.

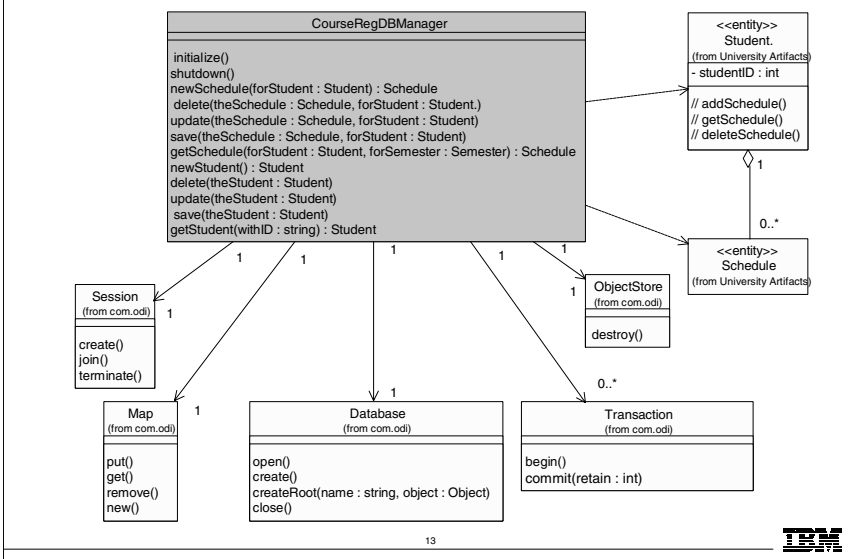
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The changes made to incorporate the mechanism are shown in blue (but this does not show up in the black and white manuals).

Showing multiple relationships on the same diagram is not a good practice. Recommend to the students that instead of this, they have a different diagram for each relationship (a separate diagram for associations, another diagram for dependencies, etc.). This provides a separation of concerns. It's difficult to comprehend more than 7 things +/-2.

Example: Incorporating ObjectStore



The above diagram shows the defined CourseRegDBManager which will control access to the persistent class, Student, and the Student's Schedules.

Remember, Student was selected as the "root class", and StudentID was selected as the unique id into the selected container class, the Map.

Operations to access Schedule were added to the CourseRegDBManager for easier design incorporation. The CourseRegDBManager manages the details of loading the appropriate Student (the selected root class) before accessing Schedule.

Note: In the above diagram, some attributes and operations have been suppressed for clarity.

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

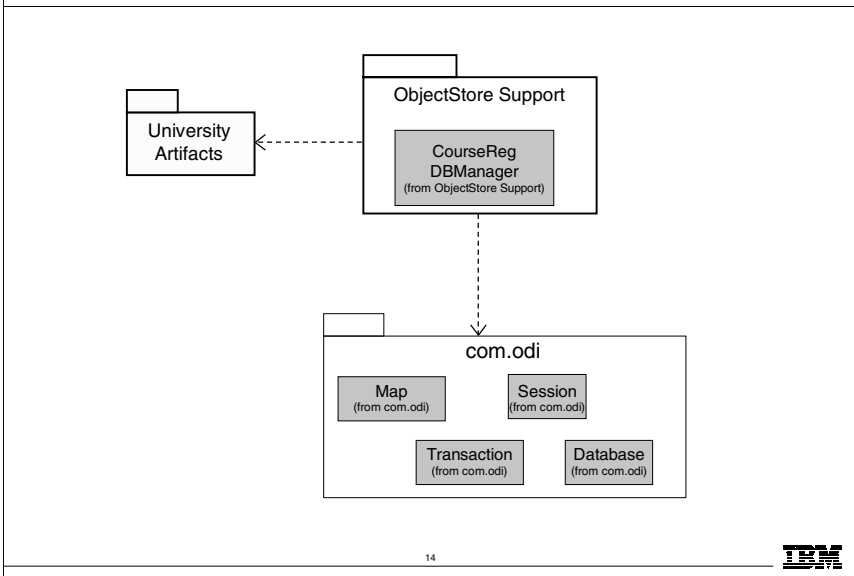
Regarding the naming convention for com.odi: Java originally suggested that when the domain name is reversed that COM is in all CAPS, but later versions suggested that it **not** be capitalized. So the convention now is just "com.odi" and not "COM.odi". We have used the lowercase standard in this course.

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Another example of the Java naming convention: If Rational had a Java package called "util" they'd call it "com.rational.util".

Example: Incorporating ObjectStore (continued)



The above diagram demonstrates the architectural implications (specifically, the package dependency implications) of incorporating ObjectStore into the course Registration System design:

- The ObjectStore Support package was created. It will contain the business-specific design elements that support the ObjectStore OODBMS persistency mechanism. This includes the previously created CourseRegDBManager.
- The CourseRegDBManager will need to the design elements that support the ObjectStore OODBMS persistency mechanism, so a dependency was added from the ObjectStore Support package to the com.odi package.

The name of the com.odi package in the model reflects the naming convention for third party Java software (i.e., use the reverse of the domain name). The Java language suggests this scheme so that different vendors don't step on each other with their Java classes. The idea is that domain names are unique so they're won't be any name clashes. Thus, com.odi has nothing to do with Microsoft COM/DCOM. "odi" stands for Object Design, Inc. which is the name of the company that makes ObjectStore.

- The CourseRegDBManager will need to have access to the persistent classes, so a dependency from the ObjectStore Support package to the the UniveristyArtifacts package was added.

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Note: If your students are interested in the OODBMS (ObjectStore) mechanism, these slides should be inserted while presenting incorporation of the the persistency mechanisms in the Use-Case Design module.

Use-Case Design Slides

The following slides can be inserted during the Use-Case Design module

15



Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Note: On the presented slide, the italicized text is also blue, but this does not show up in the back-and-white manuals.

Incorporating ObjectStore: Steps

- ♦ Provide access to the class libraries needed to implement ObjectStore access
 - √ ▪ *Dependency on com.odi*
- ♦ Select the database root class(es)
 - √ ▪ *Student class*
- ♦ Select the container class(es) that will serve as the database root(s) (contains the selected root class(es))
 - √ ▪ *Map (from com.odi)*
 - √ ▪ *Key will be Student ID*

(continued)

√ = Done



16

The above is a summary of the steps that can be used to implement the OODBMS Persistency mechanism (ObjectStore). The italicized text describes the architectural decisions made with regards to Objectstore for our Course Registration example.

These steps were introduced in Identify Design Mechanisms. They are repeated here for convenience. The check marks indicate what steps have been completed. In Use-Case Design, where will continue incorporate this mechanism.

One of the most important decisions that must be made when incorporating ObjectStore into your design is what the root class(es) are going to be, and what container class will serve as the database root, where these classes will be stored. Remember the database root (the selected container class) is a special data structure. It is associated with a specific ObjectStore database (see the Initialize interaction diagram provided earlier in this module). Any changes to this “special” data structure that occur within the context of a transaction will be applied to the associated ObjectStore Database. There may be multiple database roots

For the Course Registration System, a single root class was chosen -- Student.

The selected container was the Map, where the unique key to access the Students will be StudentID.

The remaining steps are described on the following slide.

Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

Note: On the presented slide, the italicized text is also blue, but this does not show up in the back-and-white manuals.

In this section, we will concentrate on refining the use-case realizations initially defined in Use-Case Analysis to incorporate the OODBMS persistency mechanism.

Incorporating ObjectStore: Steps

- ♦ Create a DBManager (one per ObjectStore database instance)
 - √ ▪ *Single Course Registration Database => CourseRegDBManager*
 - √ ▪ *Will “live in” ObjectStore Support package*
- ♦ Add operations to DBManager to access entities in the OODBMS
 - √ ▪ *Create operations for Student and Schedule*
- ♦ Create/Update interaction diagrams that describe:
 - Database initialization and shutdown
 - Persistent class access: Create, Read, Update, Delete
- ♦ Implement persistent classes
 - Add “import com.odj.*” statement
 - *Student to include this statement*

Out of scope
√ = Done

17



There is one DBManager class per ObjectStore database instance. For the Course Registration System example, there will be one ObjectStore database, the Course Registration Database, that contains student (and schedule) information for the university. Thus, there will be one CourseRegDBManager that will be in a new ObjectStore Support package.

Operations must be added to the DBManager class to access the OODBMS persistent entities in the database. For the CourseRegDBManager class, operations will be added to access Student and Schedule information since that is required for the core system functionality.

The interaction diagrams provide a means to verify that all required database functionality is supported by the design elements.

The sample interaction diagrams provided for the persistency architectural mechanisms during Identify Design Mechanisms will serve as a starting point for the specific interaction diagrams that we will now define.

The PersistentClass should be defined for persistent use the same it is defined for transient use. Other than the required “import com.odj.*” statement, there is almost no special code for persistent use of the PersistentClass. Once you import the ObjectStore classes, a special post-processor modifies the Java byte code to handle all the persistence. No other changes are required. Implementing the persistent classes is out of the scope of this course. It is part of Implementation, not Analysis and Design, so it is out of scope of this course.

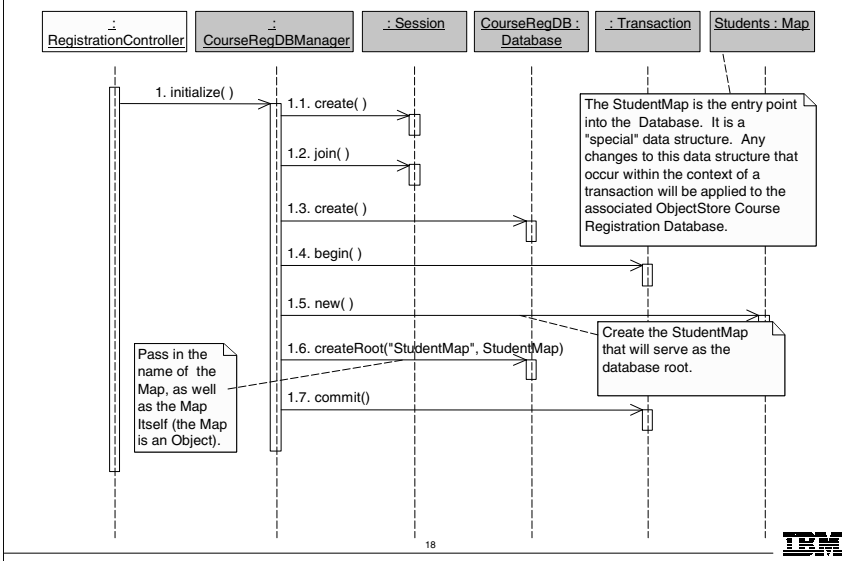
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/used as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: Initialize



The above diagram demonstrates how the ObjectStore database is initialized at system start-up. Initialization must occur before any persistent class can be accessed. Database initialization was not addressed during analysis, so this diagram does not have a counterpart in Use-Case Analysis.

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

To create the database, the `CourseRegDBManager` creates a new transaction and creates the "root" of the database with the "createRoot()" operation. In our example, the root will be the `StudentMap` data structure. It will contain instances of the `Student` class and all "reachable" classes (including `Schedules`). Remember, the root is the entry point into the `Course Registration 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 `Course Registration ObjectStore Database`.

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

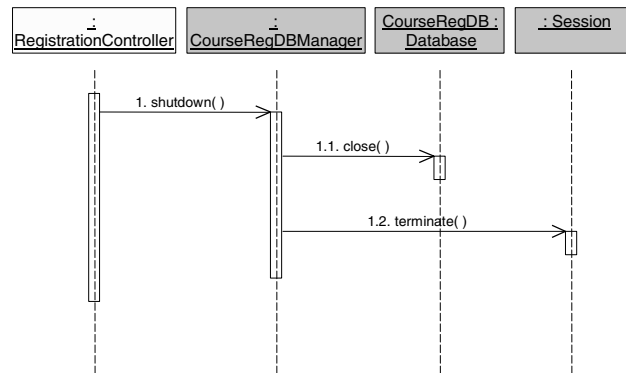
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: Shutdown



19



The above diagram demonstrates how the ObjectStore database is shutdown at system shutdown. Database shutdown was not addressed during analysis, so this diagram does not have a counterpart in Use-Case Analysis.

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

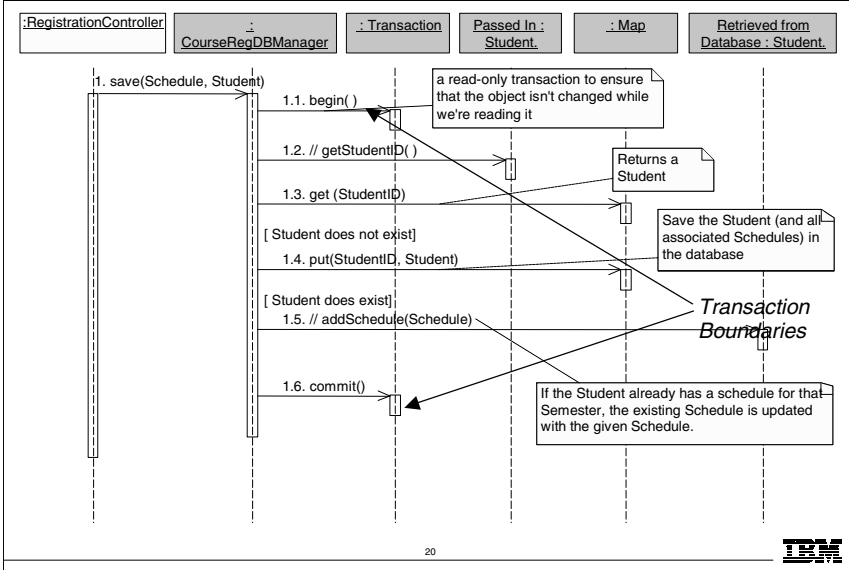
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: Create/Update



The above example is a fragment of the Register for Courses use-case realization. It describes what happens when a new Schedule is saved, or an existing Schedule is updated.

Remember, a separate **put()** to the Map is not necessary in the case of an update as the **get()** operation returns a reference to the Student and any changes to that Student (like adding a Schedule), if made in the context of a transaction, are automatically committed to the database..

Note: The example is unchanged with or without the application of the distribution mechanism, as the distribution is between controllers, and occurs before the above flow is invoked. The incorporation of the distribution mechanism is described in the RMI Mechanism module of the Additional Information appendix.

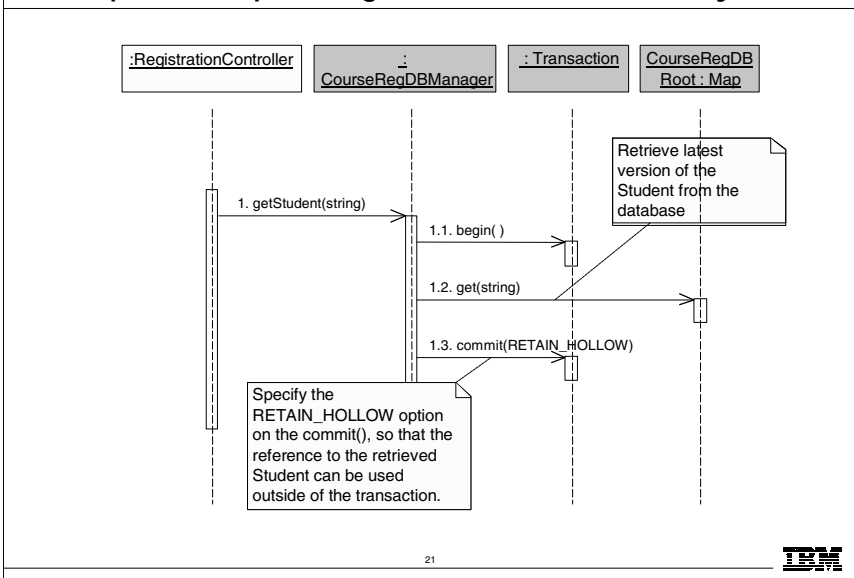
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: Read



The above example is a fragment of the Register for Courses use-case realization. It describes what happens when a particular Student is retrieved. Remember, because Student is the root class, when it is retrieved, all classes associated with it are retrieved, as well.

Note: The example is unchanged with or without the application of the distribution mechanism, as the distribution is between controllers, and occurs before the above flow is invoked. The incorporation of the distribution mechanism is described in the RMI Mechanism module of the Additional Information appendix.

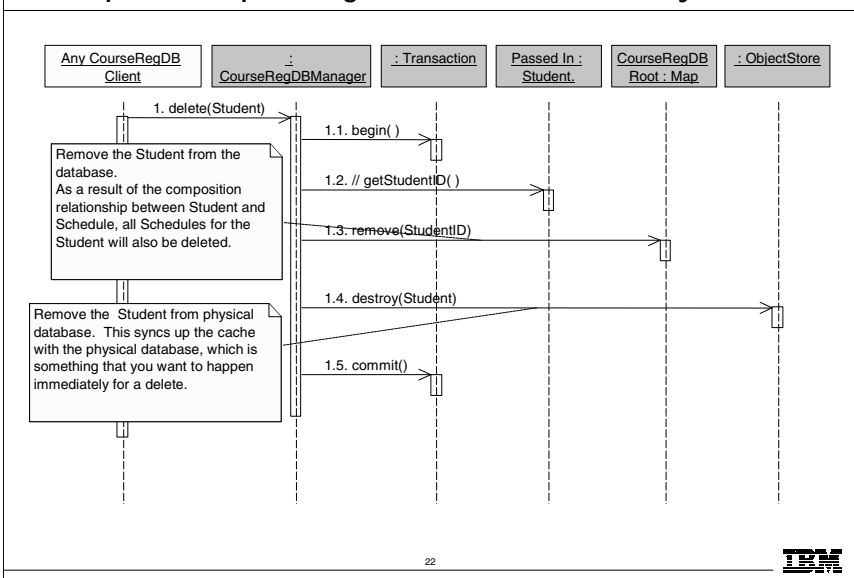
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: Delete



The above diagram describes what happens when a Student is to be deleted.

Note: This flow of events was not part of the original use cases we have been analyzing, but it is shown here for completeness (to show how an ObjectStore delete would work).

Remember: Deleting from an ObjectStore database really has two steps - removal from the container class that is the database in memory (the Map, in our example), and removal from the physical database. That is because you want the deletion to occur right away, as opposed to being cached.

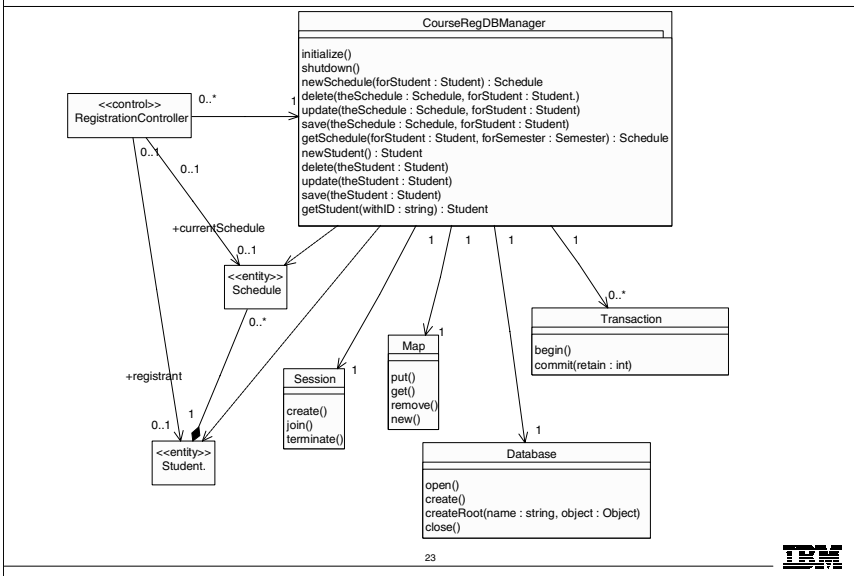
Mastering OOAD w/ UML 2.0 – Instructor Notes

Instructor Notes:

The items in blue were added/utilized as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Example: Incorporating OODBMS Persistency: VOPC



Mastering OOAD w/ UML 2.0 – Instructor Notes

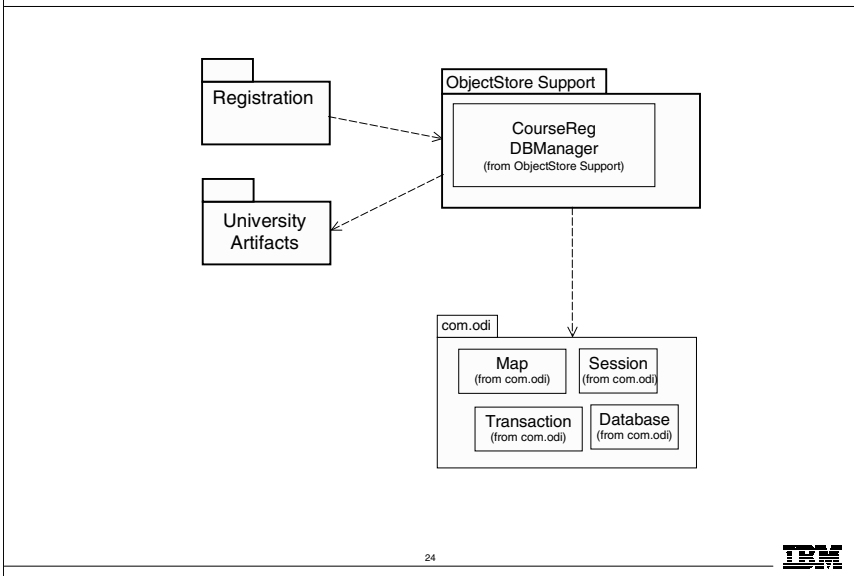
Instructor Notes:

The items in blue were added/used as part of incorporating ObjectStore.

The blue does not show up in the black and white student manuals.

Note: The relationships between the individual classes have been omitted for clarity of this diagram.

Example: Incorporating ObjectStore (continued)



The above diagram demonstrates the changes that must be made to the Course Registration Model to incorporate the ObjectStore persistency mechanism:

As discussed in Identify Design Mechanisms:

- The ObjectStore Support package contains the business-specific design elements that support the ObjectStore OODBMS persistency mechanism(i.e., the CourseRegDBManager).The CourseRegDBManager contains operations for persistent classes
- The CourseRegDBManager will need to have access to the persistent classes, so a dependency on the UniversityArtifacts package is needed.
- The com.odi package contains the design elements that support the ObjectStore OODBMS persistency mechanism.
- The CourseRegDBManager will need to access the design elements that support the ObjectStore OODBMS persistency mechanism, so a dependency was added from the ObjectStore Support package to the com.odi package.

For the Course Registration System, the persistency class clients are the control classes that “live” in the application packages. Thus, a dependency was added from the Registration package to the ObjectStore Support package to provide the persistency clients access to the CourseRegDBManager.