Phase Management Component 1.1 Specification

1. Design

Project Phases defines the logic structure of the phase dependencies in a project. This component builds a persistence and execution layer. Phases can be started, ended or cancelled. The logic to check the feasibility of the status change as well as to move the status will be pluggable. Applications can provide the plug-ins on a per phase type/operation basis if extra logic needs to be integrated.
An example of a phase type would be 'Screening' or 'Aggregation' when it pertains to a design scorecard.
Please note that version 1.1 changes will be colored coded with a red font used to denote new additions and a blue font used to show changes from 1.0 to 1.1
Transaction control if very important in persistence and 1.0 version had basically hard-coded that into a set of commit and rollback commands executed at a single level of a specific persistence action such as for example a delete operation.
Version 1.1 of this component provides a DAO implementation capable of running inside a transaction that is managed externally, possibly by a J2EE container. This new version is backward compatible and both new and old implementations are fully interchangeable in terms of the database schema. In other words for the user using this component they are not really aware of which version of the DAO is actually being used.

1.1.1 Anatomy of the proposed design

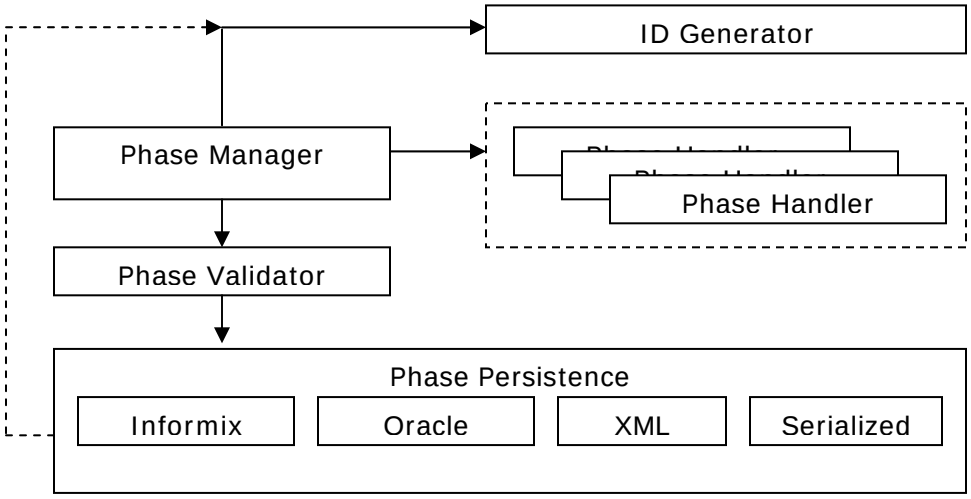
The design revolves around the following aspects of the component requirements:

- We will provide a façade for the user to interact with which acts as a manager of phases, phase states, and phase statuses. The user will mainly interact with this façade.
    - The façade will be responsible for managing persistence (pluggable), id generation for new phases, validation of phase objects (pluggable), as well as customized phase handling (pluggable)
    - The façade will also expose a number of phase operations API such as starting, ending, and canceling phases among others.
- We will also define and provide the ability to create persistence implementation, which can then be plugged into the manager. Persistence will be responsible for the following:
    - Retrieving project phases by project id (phases that belong to a project)
    - Retrieving a list of currently available phase types. An example of a phase type would be 'Screening' phase or 'Final Fixes' phase.
    - Retrieving a list of currently available phase statuses. These could be 'Closed' , 'Scheduled', and/or 'Started'
    - Phase CRUD operations. This will allow for creation, modification, and deletion of specific phases. We will also be able to read specific phases.
- We will define a Phase Validator contract which will be used by the manager to validate phase data as it is being persisted (so that no invalid phase data is persisted)
    - This design will provide a simple validator which will check that the phase object is actually filled up with data (i.e. there are no empty field values)
- We will also define a Phase Handler contract, which will provide an optional pluggable phase handling mechanism, can be configured per phase type/operation. The handler will provide the decision of whether the start, end or cancel operations can be performed.

    - We will not be providing an implementation of this handler with this design.
- Transactional control as well as connection management will be abstracted out into a set of API such as `startTransaction()`, `commitTransaction()`, and `rollbackTransaction()` for transaction control management, and `getConnection()` and `disposeConnection()` methods for connection management. All these methods will be provided in an abstract class, which will form the base for all DAOs in this component. Each implementation of the DAO will then decide how to deal with the specifics of the methods.
- All Informix specific implementation behind the persistence API will also be abstracted out so that the variations of DAO, which differ only in transactional aspects, do not have to repeat the code. All the SQL based code will be the same for all Informix based DAOs; the only difference will be how they deal with transactions and connection management. To that effect we will provide two DAO implementations:
    - We will have a self managed transaction DAO which will basically start, perform actions, and then either commit or rollback on the specific connection being used. This is how the version 1.0 InformixPhasePersistence class behaved and this behavior has been preserved here. The only change is in the architecture and class hierarchy, but to the end user nothing has changed about this class which internally delegates all the transactional aspects to the specific API mentioned above which is implemented specifically to work with local connection based transactions.
    - We will also have a DAO, which will behave transactionally in a manner, which allows outside transactional management. In other words this DAO is unmanaged when it comes to transactions. This would allow such technology as EJB (j2ee) to control the transactions for the DAO's persistence externally.

Here is a simple diagram depicting the structure of the API for this component

Diagram 1. Design Overview



This can be described as follows:

1. Phase Manager will give the ability to register different Phase Handlers.
2. A phase validator can be configured for this manager. The manager will then use this validator to perform a sanity check on phase data that is about to be persisted.
3. We can plug into the manager a persistence implementation that will be used to persist phase data to such data stores as a database (such as Oracle, DB2, etc…) an xml file, or even a serialized storage in disk.
4. Phase Manager will be configured with an id generator. Persistence will then be passed this id generator from the manager. This will allow for separation of persistence from id configuration and allow for different persistence implementations to actually use the same id generation implementation without any extra configuration.

1.1.2 Configuration aspects

PhaseManager implementation will be integrated with configuration manager and will be capable of loading an instance of a validator, all handlers, persistence instance, and id generator instance.
In turn the implementation of persistence (for Informix) will be configured with a db connection factory to generate connections.
In addition we will have a programmatic interface where each of these pieces can be plugged in directly.

1.1.3 Validation aspects

The current validation implementation will be very basic and something of sanity check. It will simply ensure that the required fields (NOT NULL fields) in the database schema for the phase table (it can be found in the \docs\phase_management.sql file)
Validation will not be checking for foreign key constraints. This would be done by the persistence itself.

1.1.4 Phase Handler registry

The point of the registry is to pair up a Phase Type and an Operation (such as CANCEL, END, START) with a specific handler. Thus we could have a handler handler-a that is registered under <'Screening', PhaseOperation.CANCEL>, which would then be used when a particular operation is being performed.

Lets say that we have the following two handlers registered:

```
Handler-A: <'Screening', PhaseOperation.START>
Handler-B: <'Screening', PhaseOperation.END>
```

If the manager is now called with a phase (which is of the type 'Screening') as follows:

```
manager.start(phase, "ivern");
```

Then the manager would lookup a handler based on the fact that we have a start method call which maps internally to START operation, and on the fact that the input phase is of 'Screening' Type. The lookup would produce the Handler-A for the operation.

1.1.4.1 Handler lifecycle and calling priority

Since handler registration is optional the manager will follow this simple lifecycle when dealing with start/end/cancel operations:

canStart() - if a handler exists use handler.canPerform(), otherwise check phase.calcStart() against the current timestamp.
canEnd() - if a handler exists use handler.canPerform(), otherwise check phase.calcEnd() against the current timestamp.
canCancel() - if a handler exists use handler.canPerform(), otherwise return true.
start() - perform the logic defined in RS, then if a handler exists do handler.perform().
end() - if a handler exists do handler.perform(), then perform the logic defined in RS.
cancel() - if a handler exists do handler.perform(), then perform the logic defined in RS.

1.1.4.2 Storing the handlers in the manager (registry structure)

Since the handlers are registered based on a composition of phase type and operation we will create a composite key (used internally to the Manager – inner class), which will aggregate the two entities, implement its own `hashCode` and `equals` methods (so that it can be put into a `HashMap` properly)

1.1.5 Persistence

The persistence implementation will work on a number of related tables directly. The DAO pattern has not been applied to the individual tables since it would complicate the design. There is a single persistence interface and the implementation will be a single implementation class.

We have two DAO implementation variations here:

1. Managed Transaction DAO: Connection in this implementation will be obtained per transaction and transaction will be committed per each operation and the connection closed. This is not inefficient since the DB Connection Factory could easily provide pooled connections (in other words we do not worry about the efficiency of obtaining and discarding connections)
2. Unmanaged Transaction DAO: Connection in this implementation will be obtained per action but will not be in any way managed by this implementation. Calls to `rollbackTransaction()` will generate an exception which will then be re-thrown by persistence to the caller. The caller is then free to decide what to do. Note that because connections are managed in an abstract manner (through DB Connection Factory) and for example if an TX Connection is being serviced then a container such as a J2EE container could easily control the transactions outside of the DAO but at the signaling of the DAO.

1.2 Design Patterns

Strategy Pattern: This has been utilized to allow the ability to plug in different versions/implementations of the phase handlers, and validators.
This basically allows the users to swap in different implementations as needed without affecting the overall design.

Façade: This has been utilized as the front for the user to interact with, in effect hiding the complexity of all interaction with the different pieces of the framework. The PhaseManager class acts as a façade.

Type Safe Enumeration: TypeSafe enum is used to control constant instances of a set of values. This is used in classes such as PhaseOperation and PhaseStatus.

Template Method Pattern has been used with `AbstractPhasePersistence` to allow users to override the transactional as well as connection management API that would be used by other persistence API (as used in `AbstractInformixPhasePersistence`)

1.3 Industry Standards

JDBC
Informix

1.4 Required Algorithms

Here we will describe the main elements of the different operational pieces of this design that are somewhat complicated. In general this is a simple component and as such has no complicated algorithms.
In a number of cases the operations will actually span multiple database calls. To keep integrity of the data we will do each such multi-stage persistence operation as a single transaction.

1.4.1 Updating project phases

The update operation is actually a read/create/update and possibly delete all in one. We also have to be aware that dependencies (i.e. Dependency entities) might have to be created/updated/deleted for each phase in the project that we are updating.

One thing to note is that a phases (i.e. Phase class) aggregate dependencies (i.e. Dependency lass) and thus we must be careful how we deal with updating phases since we must take dependencies into consideration as well.

Since it is important that these stapes are as efficient as possible, please refer to **efficiency considerations** in section 1.4.5.

Here are the steps necessary to achieve this.

Input: Project instance (project) and an operator (operator)
Step 1. Using the project.getId() fetch from the database all phases where the project_id is equal to the id we just fetched. Put them is a list (database-list-a) We also copy from the project all the phases (reference copy only) into a project list (project-list)
- For each phase we also need to read in all the dependencies. For each phase we create a list (database-list-bx where x would be an index over all the phases) of dependencies that the phase currently holds in the database (some will be empty)

[start transaction] – call the startTransaction(…) method
Step 2. For each phase x in the input project:
- if phase x exists in database-list-a then we :UPDATE this phase in the database. We also remove the phase x from the list and the project-list.

<validate the phase object first>

- if phase x doesn't exist in database-list-a then :CREATE this phase in the database. We also remove phase x from the project-list.
- Execute updateDependencies(phase x) routine

Step 3. For each phase x that is still left in database-list-a:

- we :DELETE the phase from the database.
- Execute updateDependencies(phase x) routine

[end transaction] – call either the commitTransaction() or rollbackTransaction() method

Postcondition: Operator information would have been persisted as well for a successful update

updateDependencies(phase x):

Step 1. For each dependency y in phase x:

- If dependency y exists in database-list-bx then we update this dependency in the database.
  We also remove the dependency from both the database-list-bx and the phase object.
- if dependency y doesn't exist in database-list-bx then we create this dependency in the database.
  We also remove the dependency from the phase object.

Step 2. For each dependency y that is still left in the database-list-bx:

- We delete the dependency from the database.

if at any point we have an issue then we roll back (by calling the `rollbackTransaction()` method) and then transaction and throw an `PhasePeristenceException`.

1.4.2 Phases Persistence CRuD

Input: Phase instance and operator

Precondition: By this time the id for the phase would have been already generated. This is also assumed for dependencies.

CREATE:

[start transaction] – call the startTransaction(…) method

Step 1. Create a new phase record with all the elements from the phase object (as well as the operator input)

Step 2. For each dependency in the phase object we create a new phase_dependency record and populate it with the proper data.

[end transaction]

DELETE:

[start transaction] – call the startTransaction(…) method

For each dependency in this phase we remove all the dependencies linked to it. We delete each phase_dependency record that has this phase as either dependent_phase_id or dependency_phase_id. We also ensure that we delete the phase record itself. *Please refer to section 1.4.5 for efficiency considerations.*

[end transaction]

READ:
Using the phase.getId() we lookup the phase record with this id. We need to load all the dependency entities for this phase. We do this by loading all the phase_dependency records that have this phase.getId() as either dependent_phase_id or dependency_Phase_id. *Please refer to section 1.4.5 for efficiency considerations.*

1.4.3 Operator audit

Operator audit is based on simply filling in the create_user, create_date, modify_user, and modify_date field for each create and update operation on any of the provided tables.

When creating we do the following steps:

- Fill in the create_user and modify_user fields with operator, and fill in the corresponding dates for the creation and modification time.

When updating we do the following steps:

- Fill in modify_user fields with operator, and fill in the corresponding date for modification time. Use current time stamp.

1.4.4 Generating Ids for new phases and new dependencies.

When we have a Phase object instance how do we know if it is new or old? The simplest way would be to check it has an id set or not. If the id is not set it is brand new. If not, it is old. Currently there is no set protocol for this 'check'. As specified by the PM this will be done later (during development maybe)

1.4.5 Efficiency considerations

Since efficient processing is a requirement the developer will have to pay close attention in how they do the reading and the deleting operations. What we want to do here is to minimize the number of JDBC calls when we are working with lists of ids (like project id or phase id)
The idea is to build a list of IN elements to be executed against with one JDBC call:

The **Read** operation should work as follows when we are working with a list of projects denoted by '(…)':

```
// this will allows us to use only one connection to get all the phase records
// for the project ids in the list. This list has to be built dynamically from
// the list of provided project ids:
//       (in the +getPhases(projects:long[]) : Project[] Api call)
//
SELECT * FROM phase WHERE project_id IN (...)

// Same idea, one connection for a whole list
//
SELECT ... FROM phase_dependency JOIN phase WHERE project_id IN (...)

// Again, same idea, one connection for a whole list
//
SELECT ... FROM phase_criteria JOIN phase WHERE project_id IN (...)
```

The **Delete** should work as follows when we have a list of phase ids present:
These would actually be called up during the update operation or directly through deleteXXX(…) API.

```
// This will delete all the phase criteria for phase ids in the list.
//
DELETE FROM phase_criteria WHERE phase_id IN (...)

// Will delete all the dependedncied based on the p[rovided ids.
//
DELETE FROM phase_dependency WHERE dependency_phase_id IN (...)
                                        OR dependent_phase_id IN (...)

// Will delete all phases with the specified phase ids
//
DELETE FROM phase WHERE phase_id IN (...)
```

1.5 Component Class Overview

1.5.1 com.topcoder.management.phase

PhaseManager << Interface >>
This is a contract for managing phase data for project(s). This describes functionality for manipulating phases with pluggable support for persistence CRUD. It manages connections, id generation, and phase handler registry,

PhaseHandler <<interface>
Optional pluggable phase handling mechanism can be configured per phase type/operation. The handler will provide the decision of whether the start, end or cancel operations can be performed as well as extra logic when the phase is starting, ending or canceling. Notice that the status and timestamp persistence is still handled by the component.

PhasePersistence <<interface>>
This is a persistence contract for phase persistence operations.

AbstractPhasePersistence <<abstract class>>
This is a simple abstract persistence class, which specifies the basic operations that all DAOs (i.e. persistence classes) would do. This class basically abstracts the 'plumbing' of what DAOs will deal with in terms of connections/transactions. Thus here we abstract out such actions like reading connection configuration data as well as creating overloadable API for getting a connection, closing a connection as well as managing transactions through connections or in a different manner (as in unmanaged transactions actually)

PhaseValidator <<interface>>
This is a validation contract for phase objects. Implementations will apply some validation criteria and will signal validation issues with a `PhaseValidationException` being thrown.

DefaultPhaseManager <<concrete class>>
Implementation of the PhaseManager interface.

HandlerRegistryInfo <<concrete class>>
For each handler that is registered with the manager we might want to know what the criteria was for registration.

PhaseOperation <concrete class>>
A convenient enumeration of phase operations.

PhaseStatus <concrete class>>
A convenient enumeration of phase statuses (Scheduled, Open, Closed).

1.5.2 com.topcoder.management.phase.persistence.db

InformixPhasePersistence<<concrete class>>
This is an actual implementation of database persistence that is geared for Informix database. This is a managed transaction implementation, which manages all transactional aspects such as commit or rollback internally to the class and on the local connection.

AbstractInformixPhasePersistence <<abstract class>>
This is an abstract class specifically implementing the Informix based SQL to achieve the persistence API implementation. The main reason why it is abstract is that the specifics of `commit()`, `rollback()`, and `disposeConnection()` will be dealt with by classes that will implement this particular class. This is important since these aspects might be implemented in different ways depending on different strategies. For example the `InformixPhasePersistence` descendant class implements these methods to do self-manager transactional control, but the `UnmanagedTrasnactionsInformixPhasePersistence` is designed to not manage any transactional aspects of the persistence actions. This is an actual implementation of database persistence that is geared for Informix database.

UnmanagedTransactionInformixPhasePersistence <<concrete class>>
This is a specific implementation of transactional control on top of database persistence actions geared towards an Informix Database platform. The main purpose of this class is to intercept any transactional actions and basically disregard them so that some other means of controlling the transactions could be used (i.e. Stateless Session EJB for example - container manager transaction) The most important aspect is that the `rollbackTransaction(...)` method simply throws the `PhasePersistenceException` which would then be caught by any of the C[R]UD (excluding any reads which are non-transactional) methods and which would force a re-throw of the exception to the caller. This allows the caller to properly rollback on their own.

1.5.3 com.topcoder.management.phase.validation

DefaultPhaseValidation <<concrete class>>
A simple validator for phases, which ensures that all the required fields, are actually present. This is a sanity check validation.

1.6 Component Exception Definitions

1.6.1 Custom Exceptions

PhasePersistenceException
This is a custom wrapping exception for any persistence issues coming from implementations.

PhaseManagementException
This is a custom exception to be used when there are some issues with doing operations/actions with/on phases.

PhaseValidationException
This is a custom exception to be used when phase validation fails. This will be thrown by the manager when updating a phase(s)

ConfigurationException
This is our own independent configuration exception which is thrown typically when we have a file based configuration issues.

1.6.2 System and general java exceptions

IllegalArgumentException
Thrown by the manager as well as persistence implementations when illegal or a null argument is passed. This will also include empty strings (i.e. strings that when trimmed have length 0)

1.7 Thread Safety

Thread Safety is not a requirement for this component and thus it is not intrinsically made thread safe. On the side of individual classes we have the following:

1. HandlerRegistryInfo is thread safe since it is immutable
2. PhaseOperation is an Enum and thus poses no thread-safety issues.
3. DefaultPhaseManager is not thread-safe since it is mutable and its state (data such as handlers) can be compromised through race condition issues. To make this thread-safe we would have to ensure that all the methods that use the internal handlers map have their access synchronized.
4. InformixPhasePersistence acts like a stateless bean with utility-like functionality where function calls retain no state from one call to the next. Separate connections are created each time a call is made and thus (assuming the connections are different) there is no contention for a connection from competing threads. This class is thread-safe.
5. DefaultPhaseValidation is a thread safe utility-like with no state.
6. UnmanagedTransactionInformixPhasePersistence acts like a stateless bean with utility-like functionality where function calls retain no state from one call to the next. Separate connections are created each time a call is made and thus (assuming the connections are different) there is no contention for a connection from competing threads. This class is thread-safe.
7. Bother abstract classes (AbstractInformixPhasePersistence and AbstractPhasePersistence) are thread-safe as they have no modifiable/shared state.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.4, J2EE 1.3
- Compile target: Java 1.4, J2EE 1.3

2.2 TopCoder Software Components

Project Phases 2.0
This is a required dependency that this component has to utilize.
Configuration Manager 2.1.5
Used for configuration of the component's persistence, phase handlers as well as supporting DB Connection and ID Generator component.
DB Connection Factory 1.0
This is used for usage of configured connections.
ID Generator 3.0
Use to generate Ids for new phases or dependencies.
BaseException 1.0
Used as a base for all custom exceptions in this component.
TypeSafeEnum 1.0
Used to create custom enumerations.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.phase

com.topcoder.management.phase.persistence.db

com.topcoder.management.phase.validation

3.2 Configuration Parameters

The following configuration can be used:

DB persistence implementation parameters:

Parameter	Description	Values
connectionName	The name of the connection to be used ***Optional.***	Any valid name will do. Will use default connection if no name given.
ConnectionFactory.className	This is the class name of the connection factory ***Required***	Example: com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
ConnectionFactory.namespace	This is the namespace to pass to the connection factory. ***Required***	Any valid namespace.

Validator parameters:

Parameter	Description	Values
PhaseValidator.className	Full class name of the validator to be used ***Optional.***	Any valid name will do.

Persistence parameters:

Parameter	Description	Values
PhasePersistence.className	Full class name of the persistence to be used ***Required.***	Any valid name will do.

ID generator parameters:

Parameter	Description	Values
Idgenerator.sequenceName	Named sequence. Used to retrieve an IDGenerator that can service it. ***Required.***	"phaseManager"
Idgenerator.className	Name of the IDGenerator class that services the named sequence. Will attempt to find a generator already configured to handle the name sequence. ***Optional.***	"com.topcoder.util.idgenerator.InformixSequenceGenerator"

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

The user must do the following:

- Configure (or pass programmatically) the connection aspects
- Configure (or pass programmatically) the id generator information
- Configure (or pass programmatically) the pluggable persistence

4.3 Demo

4.3.1 General manager demo

```
// Create a manager using configuration
PhaseManager manager = new DefaultPhaseManager("com.acme.phase");

// create manager programmatically
manager = new DefaultPhaseManager(myPersistence, myIdGen);

// set the validator
manager.setPhaseValidator(myValidator);
```

```
// register a phase handler for dealing with canStart()
// assume that we have type (PhaseType)
manager.registerHandler(myHandler, type, PhaseOperation.START);

//
// do some operations

// start
if(manager.canStart(myPhase)){
        manager.start(myPhase, “ivern”);
}

// cancel
if(manager.canCancel(myPhase)){
        manager.cancel(myPhase, “ivern”);
}

// end
if(manager.canEnd(myPhase)){
        manager.end(myPhase, “ivern”);
}

// get all phase types
PhaseType[] alltypes = manager.getAllPhaseTypes();

// get all phase statuses
PhaseStatus[] allstatuses = manager.getAllPhaseStatuses();

// --- we create a project object myProject

// we update it.
manager.updatePhases(myProject, “ivern”);

// --- we create an array of projects objects: myProjects
manager.updatePhases(myProjects, “ivern”);
```

4.3.2 Persistence demo

```
// create a new persistence instance
PhasePersistence persistence = new InformixPhasePersistence(“com.acme.phase”);

// get all project phases for a project with id 1000
Phases[] projectPhases = persistence.getProjectPhases(1000).getAllPhases();



// create and persist a new Phase with ‘ivern’ as operator id
Phase newPhase = new Phase();
// initialize the phase with some new data
. . .

// persistence should know that this is a new phase
System.out.println(persistence.isNewPhase(newPhase) == true);

persistence.createPhase(newPhase, “ivern”);

// read that some phase back (assuming that phase with id 200 exists
Phase oldPhase = persistence.getPhase(200);

// delete a phase from persistence, we delete the phase we have just created
persistence.deletePhase(newPhase);
```

4.3.3 Transactional control demo

While version 1.1 update has not produced a tangible API change that the user would see, we will try to demonstrate the effects of using different transactional strategies that have been enabled on version 1.1

Here we will assume that the persistence being used is UnmanagedTransactionInformixPhasePersistence, which would have been done through configuration. We will assume here that we are using this class with a stateless session been (ejb) in a container that manages the transactions through TXDataSource connections:

1. Client calls persistence (via ejb) to perform an action like delete a phase
    a. UnmanagedTransactionInformixPhasePersistence instance obtains a new TX Connection through call to getConnection();
    b. We make a call to startTransaction() which in our case does nothing (no-op) as the transaction is controlled by the container
    c. We make a call to delete the phase on the connection
    d. If all went well we call commitTransaction() which is our case is a no-op so it doesn't do anything. The commit will actually be done by the container.
2. Client calls another persistence method this time to create a phase, note that the container would join a transaction already in progress (which is the one started in 1.)
    a. UnmanagedTransactionInformixPhasePersistence instance obtains a new TX Connection through call to getConnection();
    b. We make a call to startTransaction() which in our case does nothing (no-op) as the transaction is controlled by the container
    c. We make a call to create the phase on the connection
    d. Let us assume that we had an error and we need to rollback the transaction (both 1 and 2) We simply call the rollbackTransaction() method which will throw an exception which will in turn force us to re-throw the exception which will be caught by the EJB which will then signal to the container (like through a setRollbackOnly() method call for example) to rollback the transaction for both step 1 and step 2. Of course this would depend on the set up of the EJB, but here we are merely trying to show the possibilities.

We can contrast this with the way that InformixPhasePersistence (which manages its own) would do these two steps:

3. Client calls persistence (via ejb) to perform an action like delete a phase
    a. InformixPhasePersistence instance obtains a new TX Connection through call to getConnection();
    b. We make a call to startTransaction() which in our case does nothing (no-op) as the connection has already been set to non autocommit.
    c. We make a call to delete the phase on the connection
    d. If all went well we call commitTransaction() which is our case will actually commit the transaction.
4. Client calls another persistence method this time to create a phase, note that this is a brand new transaction.
    a. InformixPhasePersistence instance obtains a new Connection through call to getConnection();
    b. We make a call to startTransaction() which in our case does nothing (no-op)
    c. We make a call to create the phase on the connection
    d. Let us assume that we had an error and we need to rollback the transaction (we can only rollback 2) We simply call the rollbackTransaction() method which will rollback the transaction in the current connection.

5. Future Enhancements

- Custom Phase handlers could be implemented.