Phase Management 1.1 Component Specification

1. Design

All changes performed when synchronizing documentation with the version 1.0.4 of the source code of this component and fixed errors in the CS are marked with **purple**.

All changes made in the version 1.1 are marked with **blue**.

All new items in the version 1.1 are marked with **red**.

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 or ended. 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.

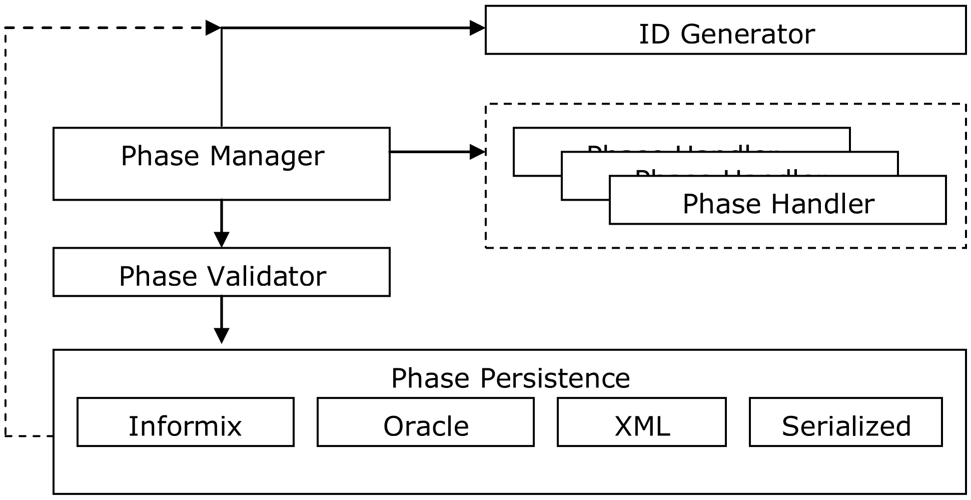
Anatomy of the proposed design

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

- We will provide a faзade 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 faзade.
  - The faзade 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 faзade will also expose a number of phase operations API such as starting and ending 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 or end operations can be performed.
  - We will not be providing an implementation of this handler with this design.

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.

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 addition we will have a programmatic interface where each of these pieces can be plugged in directly.

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. Validation will not be checking for foreign key constraints. This would be done by the persistence itself.

Phase Handler registry

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

Let's 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 look up 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.

Handler lifecycle and calling priority

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

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)

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.

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)

Changes in 1.1:

- Return type of PhaseManager#canStart(), PhaseManager#canEnd() and PhaseHandler#canPerform() methods was changed from boolean to a new OperationCheckResult entity. In case if some operation cannot be performed, an instance of OperationCheckResult entity is expected to additionally contain a message explaining why the operation cannot be performed.
- Development language was changed to Java 1.5.
- Removed support of cancel operation.

1.1 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.

Faзade: 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 faзade.

1.2 Industry Standards

JDBC
Informix

1.3 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.

1.3.1 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 phase IDs 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 class) 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 steps 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]

```
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]

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 the transaction and throw an PhasePersistenceException.

1.3.2 *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]
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]
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.*

1.3.3 *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.

1.3.4 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 Component Class Overview

DefaultPhaseManager

Default implementation of the PhaseManager interface.

The purpose of this class is to have a facade which will allow a user to manage phase information (backed by a data store). Phases can be started or ended. The logic to check the feasibility of the status change as well as to move the status is pluggable through the PhaseHandler registration API (registerHandler, unregisterHandler, etc.). Applications can provide the plug-ins on a per phase type/operation basis if extra logic needs to be integrated.

In addition, a phase validator can be provided that will ensure that all phases that are subject to persistent storage operations are validated before they are persisted. This is a pluggable option.

Changes in 1.1:

- Removed canCancel() and cancel() methods.

- Changed return type of canStart() and canEnd() methods from boolean to OperationCheckResult.

HandlerRegistryInfo

A simple data structure that encapsulates type/operation pairs for use as keys in the handler registry maintained by PhaseManager. As such its only methods are accessors and methods necessary to make it usable as a key in a HashMap.

OperationCheckResult

This class is a container for validation result returned by canStart() and canEnd() methods of PhaseManager and canPerform() method of PhaseHandler. It contains not only a boolean value indicating whether the requested operation can be performed or not, but additionally a message indicating why the operation cannot be performed. Implementations of PhaseManager and PhaseHandler should follow the following rules to make the code clearer:

1) OperationCheckResult.SUCCESS should be returned by canXXX() methods to indicate that operation can be performed for the given phase;

2) new OperationCheckResult(<<explanation>>) should be returned by canXXX() methods to indicate that operation cannot be performed for the phase and to provide a valid reasoning.

PhaseComparator

This is an inner class of DefaultPhaseManager. It is a comparator that compares Phase instances.

Changes in 1.1:

- Added generic parameter for implemented Comparator<T> interface.

- Added default constructor.

- Updated compare() method to use Phase instead of Object parameters.

PhaseHandler [interface]

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

When a user wants a phase to be changed, the manager will check if a handler for that phase (i.e. for that

PhaseType and for the operation being done such as START or END a phase) exists and will then use the handler to make decisions about what to do, as well as use the handler for additional work if phase can be changed

We have the following invocation scenarios from a PhaseManager implementation with reference to phase handlers:

- PhaseManager.canStart() - if a handler exists in the registry then the manager invokes the handler's canPerform() method to see if we can change phase (i.e. start a new phase) and if yes, then we use perform handler.perform() for any additional tasks to be performed.

- PhaseManager.canEnd() - if a handler exists in the registry then the manager invokes the handler's canPerform() method to see if we can change phase (i.e. end current phase) and if yes, then we use perform handler.perform() for any additional tasks to be performed.

- PhaseManager.start() - if a handler exists call handler.perform() before performing the associated persistence operations

- PhaseManager.end() - if a handler exists call handler.perform() before performing the associated persistence operations

Changes in 1.1:

- canPerform() method was updated to return not only true/false value, but additionally an explanation message in case if operation cannot be performed

PhaseManager [interface]

This is a contract for managing phase data for project(s). This interface provides functionality for manipulating phases with pluggable support for persistence CRUD (creation, update, deletion). It manages connections, ID generation, and phase handler registration.

The assumption is that each thread will have its own instance of a PhaseManager implementation.

Changes in 1.1:

- Removed canCancel() and cancel() methods.

- Changed return type of canStart() and canEnd() methods from boolean to OperationCheckResult.

PhaseOperationEnum [enum]

A convenient enumeration of phase operations as defined by the Phase Manager API which currently allows operations of "start" and "end". This is provided as a utility to the user when they need to identify the phase operation when registering a phase handler with a manager. This is used when creating HandlerRegistryInfo instances for handler registrations (used as keys to id handlers), which need to know what operation the handler will handle (as well as which phase status it will deal with - which is covered by PhaseStatusEnum).

Changes in 1.1:

- Made this class a Java enum instead of extending Enum from Type Safe Enum component.

- Removed CANCEL enum value.

PhasePersistence [interface]

This is a persistence contract for phase persistence operations. This is broken down into APIs that deal with basic CRUD of phase management (creation, update, deletion) for both individual phases and groups of related phases. The interface also includes methods that allow for retrieval of all phases linked to a particular project or projects.

This class works in conjunction with PhaseManager to insulate the end user from the details of phase persistence

Implementations should provide a constructor that accepts a single String argument representing the name of a com.topcoder.util.config.ConfigManager namespace. The implementation should initialize itself based on the specified namespace. This constructor will be used by the PhaseManager

PhaseManager to instantiate the persistence object when using the configuration manager

The configuration parameters are as follows.

- connectionName - the name of the connection (optional)

- ConnectionFactory.className - the name of the connection factory implementation

- ConnectionFactory.namespace - the namespace to use to instantiate the connection factory

PhaseStatusEnum [enum]

An enumeration of currently supported phase statuses.

Changes in 1.1:

- Made this class a Java enum instead of extending Enum from Type Safe Enum component.

PhaseValidator [interface]

This is a validation contract for phase objects. Implementations will apply some validation criteria and will signal validation issues by throwing PhaseValidationException.

DefaultPhaseValidator

A simple validator for phases that ensures that all the required fields are actually present. This is a sanity check validation, which ensures that all the required fields have been initialized with some values. The fields correspond to the non-Nullable fields for the phase table in the database.

1.5 Component Exception Definitions

ConfigurationException

Thrown by the DefaultPhaseManager(String namespace) constructor if a configuration parameter is missing or invalid.

Changes in 1.1:

- Extends BaseCriticalException instead of BaseException

- Added new constructors to meet TopCoder standards

PhaseHandlingException

This exception is thrown by DefaultPhaseManager and implementations of PhaseHandler when an error occurs while performing the phase transition or while determining whether the operation can be performed.

Changes in 1.1:

- Added new constructors to meet TopCoder standards

PhaseManagementException

Thrown by various PhaseManager methods when an error occurs. If the error was the result of an internal exception (such as a persistence problem), the PhaseManagementException will have an associated wrapped exception.

Changes in 1.1:

- Extends BaseCriticalException instead of BaseException

- Added new constructors to meet TopCoder standards

PhasePersistenceException

Thrown by PhasePersistence instances when an error occurs related to the persistent store.

Changes in 1.1:

- Extends BaseCriticalException instead of BaseException

- Added new constructors to meet TopCoder standards

PhaseValidationException

Thrown by PhaseValidator.validate() when an invalid phase is encountered.

Changes in 1.1:

- Extends BaseCriticalException instead of BaseException

- Added new constructors to meet TopCoder standards

1.6 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. PhaseOperationEnum and PhaseStatusEnum are enums and thus pose 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. DefaultPhaseValidation is a thread safe utility-like with no state.

Thread safety of this component was not changed in the version 1.1.

2. Environment Requirements

2.1 Environment

Development language: Java 1.5
Compile target: Java 1.5, Java 1.6
QA Environment: RedHat Linux 4, Windows 2000, Windows 2003

2.2 TopCoder Software Components

Project Phases 2.0

This is a required dependency that this component has to utilize. It defines project phase specific entities.

Configuration Manager 2.2
Used for configuring this component.

ID Generator 3.0
Used for generating IDs for new phases or dependencies.

Base Exception 2.0
Used as a base for all custom exceptions in this component.

Object Factory 2.0.1
Used for creating pluggable object instances when configuring the component.

2.3 *Third Party Components*

None

3. Installation and Configuration

3.1 *Package Name*

com.topcoder.management.phase
com.topcoder.management.phase.validation

3.2 *Configuration Parameters*

The following configuration can be used:

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.
PhaseValidator.className	Full class name of the validator to be used ***Optional.***	Any valid name will do.
PhasePersistence.className	Full class name of the persistence to be used ***Required.***	Any valid name will do.

Handler parameters:

Parameter	**Description**	**Values**
Handlers	The handlers which be registered when construct the manager. ***Optional.***	Handler1, testHandler

ObjectFactoryNamespace	Namespace for object factory. Required when Handlers exists.	Com.topcoder.util.objectfactory.
<handler>.phaseType	Phase type of the handler to handle ***Required.***	"Screening"
<handler>.operation	Operation to be performed on by the handler. ***Required.***	The value must be one of "start" and "end".
<handler>.handlerDef	The definition of the handler in Object Factory. ***Required.***	

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. ***Required.***	"com.topcoder.util.idgenerator. InformixSequenceGenerator"

3.3 Dependencies Configuration

Please see docs of dependency components to configure them properly.

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

Please see the demo.

4.3 Demo

4.3.1 General manager demo

```
// set up the config manager
ConfigManager.getInstance().add("config.xml");

// create a manager using configuration
PhaseManager manager = new DefaultPhaseManager("test.default");

// set up a simple project with a single phase
final Project project =
  new Project(new Date(), new DefaultWorkdaysFactory(false).createWorkdaysInstance());
```

```
final PhaseType phaseTypeOne = new PhaseType(1, "one");
final Phase phaseOne = new Phase(project, 1);
phaseOne.setPhaseType(phaseTypeOne);
phaseOne.setFixedStartDate(new Date());
phaseOne.setPhaseStatus(PhaseStatus.SCHEDULED);
project.addPhase(phaseOne);

// create some of the pluggable components
DemoIdGenerator idgen = new DemoIdGenerator();
DemoPhaseValidator validator = new DemoPhaseValidator();
DemoPhaseHandler handler = new DemoPhaseHandler();

PhasePersistence persistence = new DemoPhasePersistence() {
        public PhaseType[] getAllPhaseTypes() {
            return new PhaseType[] {phaseTypeOne};
        }

        public PhaseStatus[] getAllPhaseStatuses() {
            return new PhaseStatus[] {phaseOne.getPhaseStatus()};
        }

        public Project getProjectPhases(long projectId) {
            return project;
        }
    };

// create manager programmatically
manager = new DefaultPhaseManager(persistence, idgen);

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

// register a phase handler for dealing with canStart()
manager.registerHandler(handler, phaseTypeOne, PhaseOperationEnum.START);

//
// do some operations

// check if phaseOne can be started
OperationCheckResult checkResult = manager.canStart(phaseOne);

// start
if (checkResult.isSuccess()) {
    manager.start(phaseOne, "ivern");
} else {
    // print out a reason why phase cannot be started
    System.out.println(checkResult.getMessage());
}

// check if phaseOne can be ended
checkResult = manager.canEnd(phaseOne);

// end
if (checkResult.isSuccess()) {
    manager.end(phaseOne, "TCSDEVELOPER");
} else {
    // print out a reason why phase cannot be ended
    System.out.println(checkResult.getMessage());
}

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

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

// update the project
manager.updatePhases(project, "ivern");
```

5. Future Enhancements

None