Project Phases 2.0 Component Specification

1. Design

The Project Phases component will allow an application to define a project, its phases and all its dependencies. A start date, length of time and any other custom attributes necessary will define each phase. This will enable the application to automatically adjust the timeline of dependent phases.

Project class is designed to model the project, which is composed of multiple phases. And Phase class is designed to model each phase in the project. Each phase can also depend on another phase in the project.

A more flexible dependency specification is provided in version 2.0. A new Dependency class is introduced to represent the relationship between two phases. This class specifies whether the dependent phase will start/end after the dependency phase starts/ends with a lag time. Thus there are 4 types of relationship between dependency and dependent phase.

With the dependencies between phases, we could calculate the start/end date of all phases and the end date of whole project.

The date is calculated by Workdays component. If a phase starts on Friday, and its length is one day, this phase will end on Monday rather than Saturday.

A phase comparator is also provided to compare two phases base on the start and end date.

Change Note in V2.0

Project
In addition to the attributes from the previous version, each project is associated with a numeric identifier.

Phase Type
This is a new class. A phase type consists of a numeric identifier and a name.

Phase Status
This is a new class. A phase status consists of a numeric identifier and a name. The component has three built in phases – Scheduled, Open and Closed.

Phase
In addition to the attributes from the previous version, the following additions/modifications are made:

- Add a numeric identifier
- Add a phase type

- Add a phase status
- Phase length should be accurate to milliseconds
- Fixed start time would be interpreted as “start no early than”. It's the earliest point when a phase can start. Fixed start time is optional.
- Add scheduled start/end timestamps. They are the original plan for the project timeline. There is no extra logic attached to them.
- Add actual start/end timestamps. They are available for the phase that's already started. They can override the start/end time calculated.

Phase Dependency

Phase dependency specifies whether the dependent phase will start/end after the dependency phase starts/ends with a lag time. The lag time is accurate to milliseconds.

1.1 Design Patterns

Strategy Pattern

Both Comparator and Workdays interfaces are used in this pattern.

1.2 Industry Standards

None

1.3 Required Algorithms

1.3.1 Calculate phase start date

Phase start date is calculated in this way:

if actual start time exists, return it, otherwise take the latest date among:

a) fixed start time if it exists

b) if no start dependencies exist, the project start time

c) if start dependencies exist, the dependency calculations plus lag time

A recursion algorithm is provided. The time complexity is O(E), where E is the number of dependencies.

```
Date calcStartDate(Set visited, Map startDateCache, Map endDateCache) {
    // detecte cyclic dependency
    if (visited.contains(this)) {
        throw new CyclicDependencyException("cycle detected.");
    }
    // if actual start time exists, return it
    if (actualStartDate != null) {
        return actualStartDate;
    }
    // if the start date is cached, return it
    if (startDateCache.containsKey(this)) {
        return (Date) startDateCache.get(this);
    }

    // get the project start time, the phase can't start before this time.
    Date latest = project.getStartDate();
```

```
    // if the fixed start date exists, update the latest date if necessary
    if (fixStartDate != null && fixedStartDate.getTime() > latest.getTime()) {
        latest = fixStartDate;
    }

    // add this phase to the visited phases set
    visited.add(this);

    // for each start dependency,
    // calculate the dependency's start/end time plus lag time
    for (Iterator itr = dependencies.iterator(); itr.hasNext();) {
        Dependency dependency = (Dependency) itr.next();
        if (dependency.isDependentStart()) {
            // calculate the dependency's start/end time
            Date dependencyDate = null;
            if (dependency.isDependencyStart()) {
               dependencyDate = addDate(
                  dependency.getDependency().calcStartDate(visited,
                  startDateCache, endDateCache), dependency.getLagTime())
            } else {
               dependencyDate = addDate(
                  dependency.getDependency().calcEndDate(visited,
                  startDateCache, endDateCache), dependency.getLagTime())
            }
            // update the latest time
            if (dependencyDate.getTime() > latest.getTime()) {
                latest = dependencyDate;
            }
        }
    }

    // remove this phase from the visited phases set
    visited.remove(this);

    // cache the start date for this phase
    startDateCache.put(this, latest);

    return latest;
}
```

1.3.2 Calculate phase end date

Phase end date is calculated in this way:

if the actual end time exists, return it, otherwise take the latest date among:

a) calcStartDate() plus phase length

b) if end dependencies exist, the dependency calculations plus lag time

A recursion algorithm is provided. The time complexity is O(E), where E is the number of dependencies.

```
Date calcEndDate(Set visited, Map startDateCache, Map endDateCache) {
    // detecte cyclic dependency
    if (visited.contains(this)) {
        throw new CyclicDependencyException("cycle detected.");
    }
    // if actual end time exists, return it
    if (actualEndDate != null) {
        return actualEndDate;
```

```
        }
        // if the end date is cached, return it
        if (endDateCache.containsKey(this)) {
            return (Date) endDateCache.get(this);
        }

        // calculate the phase start date plus phase length,
        // the phase can't end before this time.
        Date latest = addDate(
          calcStartDate(visited,startDateCache,endDateCache), length);

        // add this phase to the visited phases set
        visited.add(this);

        // for each end dependency
        // calculate the dependency's start/end time plus lag time
        for (Iterator itr = dependencies.iterator(); itr.hasNext();) {
            Dependency dependency = (Dependency) itr.next();
            if (!dependency.isDependentStart()) {
                // calculate the dependency's start/end time
                Date dependencyDate = null;
                if (dependency.isDependencyStart()) {
                   dependencyDate = addDate(
                      dependency.getDependency().calcStartDate(visited,
                      startDateCache, endDateCache), dependency.getLagTime());
                } else {
                   dependencyDate = addDate(
                      dependency.getDependency().calcEndDate(visited,
                      startDateCache, endDateCache), dependency.getLagTime());
                }
                // update the latest time
                if (dependencyDate.getTime() > latest.getTime()) {
                    latest = dependencyDate;
                }
            }
        }

        // remove this phase from the visited phases set
        visited.remove(this);

        // cache the end date for this phase
        endDateCache.put(this, latest);

        return latest;
    }
```

1.3.3 *Calculate and Cache Project Date*

```
    public void calculateProjectDate() {
        Map startDateCache = new HashMap();
        Map endDateCache = new HashMap();
        // for each phase, calculate and cache start/end Date
        for (Iterator itr = phases.iterator(); itr.hasNext();) {
            Phase phase = (Phase) itr.next();
            Phase.setCachedStartDate(phase.calcStartDate(
                  new HashSet(), startDateCache, endDateCache));
            Phase.setCachedEndDate(phase.calcEndDate(
                  new HashSet(), startDateCache, endDateCache));
        }
    }
```

1.3.4 Calculate date with Workdays Component

The minimum time unit in Workdays is minute. We need a conversion before calculate date.

```
Date addDate(Date date, long length) {
    return workdays.add(date, WorkdaysUnitOfTime.MINUTES, (int) length/60000);
}
```

1.3.5 Remove phase algorithm of Project

The removePhase(Phase) method of Project class requires:

- Remove the given phase
- Remove all its dependent phases
- Check the cyclic dependency of phases

This is a DFS algorithm without recursion. The complexity of algorithm is O (N + E), where N is the number of phases, E is the number of dependencies. The algorithm can be simply described as:

(1) A removedPhases set holding all the phases should be removed, visitedPhases set holding all the phases already visited. Initialize them with the given phase.
(2) For each phase of the project, use DFS to see whether this phase should be removed or kept. If any one of the dependency phases in the dependency chain of this phase will be removed, then this phase should be removed, otherwise this phase will be kept.

(2.1) A stack holding all the searched phases which are the dependency phases of current phase. An arrayStack holding the corresponding dependency array of the phase, an indexStack holding the corresponding index. Initialize the stack with the current phase, arrayStack with the dependency array of the current phase, indexStack with 0.
(2.2) While the stack is empty, return to (2) to check next phase, current phase will be kept.
(2.3) Get the index of indexStack.peek(), and increment the indexStack.peek() by 1.
(2.4) If the index has gone to the end of dependency array, then pop the stack, arrayStack and indexStack. Since no more dependency of need to see, we should go back one step.
(2.5) Get the dependency phase of arrayStack.peek()[indexStack.peek()].
(2.6) If the dependency phase has not visited before, we push this phase, corresponding dependency array and 0 to stack, arrayStack and indexStack respectively. Don't forget to make this phase visited. This is the meaning of Deep-First-Search!
(2.7) Otherwise, we need to do two things for this dependency phase:
(2.8) If stack contains this dependency phase, then cyclic dependency is detected! We should throw CyclicDependencyException and stop the whole method.
(2.9) If removedPhases contains this dependency phase, then we should remove all the phases holding in the stack, because the stack is actually a dependency chain! All the phases of the stack will be put into the removedPhases set. We then return to (2) to check next phase, current phase will be removed.

(3) Remove all the phases of removedPhases set.

1.4 Component Class Overview

AttributableObject

The class to provide an extensible way for including custom attributes for both Project and Phase class. The attributes are stored in key-value pairs, the type of key and value can vary in different situations, so they are both stored in Object type for best extensibility.

Project

It is the main class of this component, each application or component can be represented by a Project instance which is composed of a collection of phases depending on each other. For example, an application may be composed of design phase, design review phase, development phase, dev review phase, and deployment phase. The phases (except the first one) can only be started only if the former one is finished, and each phase has a start date, and length to finish, at the same time, the project has a start date so that we can know when will the project be finished.
All the phases added to the project and dependent should belong to the same project. When a phase in one project depends on the phase in another project, we can always combine the two projects into one so that all phases belong to the same project to make things easy.

Phase

Represent the phase making up of the project. Any phase can only belong to one project and has a non-negative length in milliseconds. The phase also has the id, type and status attributes.
A phase could depend on a collection of other phases. The relationship between phases could be specified by the Dependency class. Dependent phase will start/end after the dependency phase starts/ends with a lag time.
Phase could be attached a fixed start timestamp. Fixed start time is interpreted as “start no early than”, It's the earliest point when a phase can start.
Phase could also be attached scheduled start/end timestamps and actual start/end timestamps. Scheduled start/end timestamps are the original plan for the project timeline. Actual start/end timestamps are available for the phase that's already started/ended. They can override the start/end time calculated otherwise.
The phase start and end date could be calculated based on the dependencies and the timestamps.

PhaseDateComparator

This class is an implementation of Comparator to compare the phases by start date and end date of the Phase. The two phases will first be compared by the start date, and then break the ties by end date of the phase. If two phases have the same start date and end date, the order in the sorted list is unpredictable.

PhaseType

This class represents the phase type. A phase type consists of a numeric identifier and a name. This class is serializable.

PhaseStatus

This class represents the phase status. A phase status consists of a numeric identifier and a name. This class is serializable.
This class defines three build in phase types- SCHEDULED, OPEN, CLOSED

Dependency
The Dependency class represents the relationship between two phases. This class specifies whether the dependent phase will start/end after the dependency phase starts/ends with a lag time. This class is serializable.

1.5 Component Exception Definitions

CyclicDependencyException
This exception will be thrown if cyclic dependency is detected in the project. It may be thrown when calculate the start/end time of the project or phase, and remove phase form project.

IllegalArgumentException

This exception is thrown when the argument is null, or the added/removed phases belong to different project, or the end date is earlier than start date. Refer to the documentation tab for more details.

ClassCastException

This exception is thrown from the PhaseDateComparator if the objects to compare are null or not of Phase type.

1.6 Thread Safety

This component is not thread-safe, the requirements asks for the mutable functionality of a lot of project/phase's attributes, making those methods synchronized will only complicate the situation, and the client is better to synchronize component externally.
To make the component thread-safe, except to make the access methods of the variables synchronized

2. Environment Requirements

2.1 Environment

Development language: Java1.4
Compile target: Java1.4

2.2 TopCoder Software Components

Workdays 1.0
This component is used for date calculation.
Base Exception 1.0
Base class for custom exception is taken from it

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.project.phases

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

None.

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

See the demo.

4.3 Demo

Note: This section has been updated according to the Demo class.

```
// 1) Create a simplified TC project with submission, review and fix phases.
Date projectStartDate = new Date();
Project project = new Project(projectStartDate, new MyWorkdays());
Phase submission = new Phase(project, 7 * 24 * 3600 * 1000);
Phase review = new Phase(project, 7 * 24 * 3600 * 1000);

// 2) Assign various attributes to them.
project.clearAttributes();
project.setAttribute("Project Name", "Project Phases Version 2.0");
project.setAttribute("Project Manager", "PM");
review.setAttribute("Initial Submissions", new Integer(8));

// 3) Build up the dependency graph.
// review phase will start immediately after submission phase ends
review.addDependency(new Dependency(submission, review, false, true, 0));

// 4) Set the scheduled start and end date of review phase
review.setScheduledStartDate(new Date(projectStartDate.getTime() + 7 * 24 * 3600
* 1000));
review.setScheduledEndDate(new Date(projectStartDate.getTime() + 14 * 24 * 3600 *
1000));

// 5) Submission end date is changed! PM decides to give review phase an early
start
submission.setActualEndDate(new Date(projectStartDate.getTime() + 6 * 24 * 3600 *
1000));
review.setLength(review.getLength() + 24 * 3600 * 1000);

// 6) Some submissions drop out after review.
review.setAttribute("Passing Submissions", new Integer(
    ((Integer) review.getAttribute("Initial Submissions")).intValue() - 1));

// 7) Sorry, fix phase is forgotten. Anyway we can fix.
Phase fix = new Phase(project, 7 * 24 * 3600 * 1000);
fix.addDependency(new Dependency(review, fix, false, true, 0));

// 8) Ivern wonders when it could enter final fix, and when it could be done.
Date finalFixStartDate = finalFix.calcStartDate();
Date finalFixEndDate = finalFix.calcEndDate();

// 9) PM wonders when the project could be done.
Date projectEndDate = project.calcEndDate();
```

```
// 10) It is amazing but looks like this component does not need final fix :-)
project.removePhase(fix);
```

5. Future Enhancements

A pluggable persistence layer will be added in the future to support the ability of loading and storing a project's phase data. A template mechanism can generate phases for similar project structure.