Auto Pilot 1.0 Component Specification

1. Design

Auto Pilot makes use of the API defined by Phase Management to automate the project execution. Scheduled phases will be started if certain conditions are met, and open phases will be ended if certain conditions are met. Phase execution will be evaluated periodically or on events. Phase changes will be audited. A command-line interface is also provided.

The flow of the component starts by specifying the projects to auto pilot. Each project will be identified by its id. It must be possible to retrieve a list of project ids based on some criteria. The interface `AutoPilotSource` defines a pluggable mechanism to provide this functionality. `ActiveAutoPilotSource` is an implementation of it, which retrieves all projects which are active and have auto pilot switch (“`AutoPilot Option`”) on in their extended properties. It uses the Project Management component to accomplish this task.

Once we have the project id, we need to auto-pilot the project. This simply means advancing project phases. If any of the project’s open phases can be ended, it’s ended. Likewise, if any of the project’s scheduled phases can be started, it’s started. This is applied until no more phase changes could be made. Each phase change must be audited. The interface `ProjectPilot` is responsible for this task. `DefaultProjectPilot` implements this task using a post-order (postfix) tree traversal algorithm to analyze the phase dependencies and end/start phases accordingly. The actual end/start task is delegated to the Phase Management component. Auditing is done using the Logging Wrapper component.

`AutoPilot` is the main class of this component which combines both tasks above in an easy-to-use API. Additionally, it provides some convenient methods to explicitly advance project phases given its id.

`AutoPilotJob` adapts the `AutoPilot` class so that it can be run as a job using the Job Scheduling component. It provides the ability to execute `AutoPilot` at some configurable interval starting from midnight. The default value is 5 minutes. It also provides the command-line interface to either start the background poll job or run the auto pilot job once. The command-line provides comprehensive, yet easy to use, switches which enable access to all the above functionalities.

1.1 Design Patterns

Strategy pattern is used by `AutoPilotSource` and `ProjectPilot` to enable pluggable project source and piloting algorithm.

Adapter pattern is implemented by `AutoPilotJob` to adapt `AutoPilot` class so that it can be run as a job using TopCoder Job Scheduling component.

1.2 Industry Standards

None.

1.3 Required Algorithms

1.3.1 Advancing project phases

A project consists of several phases and each phase may depend on zero or more other phases. It's guaranteed by the Project Phases component that no cyclic dependency is allowed (this will raise an error).

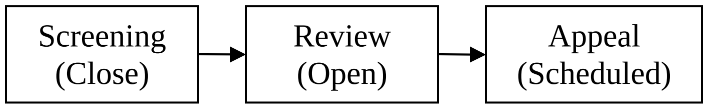
Advancing project phases is basically done by iterating the entire project phases, ending all open phases if possible, and starting all scheduled phases if possible. This process is repeated until no more phase change can be made. In pseudo code:

```
Project project = phaseManager.getPhases(projectId);
Phase[] phases = project.getAllPhases();
Do {
    foreach (Phase phase in phases) {
        if (phase status is open && can be ended) {
            end the phase
        }
        if (phase status is scheduled && can be started) {
            start the phase
        }
    }
} while (a phase change can still be made)
```

This algorithm might not be efficient however because there's no guarantee that the order returned by `getAllPhases` will be the most efficient to process.

In order to minimize the number of phase operation queries, we should take into account the phase dependencies. The rule-of-thumb is before processing a given phase, process all of its dependencies first. If we model the phase dependencies as a tree data structure, this rule translates directly to the post-order (postfix) traversal algorithm. This algorithm will make sure that all leaf nodes (phases without dependencies) are processed first and move forward to the next phase up in the tree.

To prove why processing based on phase dependency is more efficient, let's see the following simple sample:



Ignoring the dependency, let's assume the worst case where we're iterating backwards:

- 1st iteration:
  - Can appeal phase be started? No.
  - Can review phase be ended? Yes. End the review phase.
- 2nd iteration:
  - Can appeal phase be started? Yes. Start the appeal phase.
- 3rd iteration:
  - Can appeal phase be ended? No.

We did three iterations and four queries. Let's now try to process in the order of the dependencies first.

- 1st iteration:
  - Can review phase be ended? Yes. End the review phase.
  - Can appeal phase be started? Yes. Start the appeal phase.
- 2nd iteration:
  - Can appeal phase be ended? No.

We reach our final state faster by doing two iterations and three queries.

Note that it is not guaranteed that the dependency relationship is “end previous phase and start next phase”. In fact, it could be all four possible combinations of “start/end previous phase and start/end next phase”. Since this information is not readily available via the phase model, there’s no optimization we could do about it and we could stick to our rule-of-thumb of doing post-order traversal.

```
processPhase(phase) {
    // base condition: return if phase is null
    if (phase == null) return;

    // get all dependencies
    Dependency[] dd = phase.getAllDependencies();

    // recursively process all dependencies first
    foreach (Dependency d in dd) {
        processPhase(d.getDependency());
    }

    // process this phase
    if (phase status is open && can be ended) {
        end the phase
    }
    if (phase status is scheduled && can be started) {
        start the phase
    }
}
```

Since there’s no API to retrieve the first or last node in the tree and the fact that the tree might be disconnected, we’ll have to iterate the array returned by `getAllPhases()` and invoke the above algorithm for each phase. To optimize further this DFS algorithm, we could do some pruning by not processing an already processed node. To do this, we simply keep track of all processed phase id in a `Set`. Finally, we must keep reiterating the array until we make sure that no more phase change can be made. See `DefaultProjectPilot#advancePhase` for more detailed code.

1.3.2 *Command-line interface*

There are not any complicated algorithms here. Command Line Utility component is used for this purpose. The following switches are supported:

```
[-config configFile] [-namespace ns [-autopilot apKey]]
(-poll [interval] [-jobname jobname] | -project [Id[, ...]])
```

The command-line supports two kinds of operation:

- Run once (by specifying project)
- Poll mode (by specifying poll)

At least one of the options (poll/project) must be specified. Specifying both is also illegal. Here's an explanation of each switches:

- configFile specifies the configuration file to load upon startup. If not specified, it's assumed the configuration file is preloaded.
- ns/apKey refers to the namespace/key used to instantiate `AutoPilotJob`.
  In poll mode (using scheduler), apKey is optional; if specified, it's ignored. ns is used to configure scheduler. Default to AutoPilotJob class full-name.
  In project mode (one-shot), it's illegal to specify only one of ns or apKey. ns & apKey is used to configure AutoPilotJob. Default value for ns is `AutoPilotJob`'s full name. Default value for apKey is `AutoPilot`'s full name.
- poll is used to define polling interval in minutes. The default interval value is 5 minutes.
- jobname is an optional name for the job. The default value is "AutoPilotJob". It's an error to specify this option without specifying poll.
- project can be specified to process a list of project ids. If no id is specified, the `AutoPilotSource` is used instead.

1.4 Component Class Overview

AutoPilot

This is the main class which performs auto-pilot for projects. Auto-piloting a project is ending a project phase (if it's open and certain conditions are met) and starting a project phase (if it's scheduled and certain conditions are met). This class delegates the project phase execution to `ProjectPilot` interface. The projects ids to auto-pilot can be supplied programmatically or automatically searched from all projects who met certain criteria. The task of searching projects is delegated to `AutoPilotSource` implementation. Note, this class doesn't poll/execute phase change at certain intervals.

AutoPilotJob

Represents an auto pilot job that is to be executed using the Job Scheduling component. This class implements the `Runnable` and `Schedulable` interface. A new instance of this class will be created and executed (in a separate thread) by the Scheduler at a certain interval. This class simply encapsulates `AutoPilot` instance. The command-line interface is also provided via this class.

AutoPilotResult

Returned by `AutoPilot` to represent the result of auto-piloting a project. It contains the project id, the number of phases that are successfully ended, and the number of phases that are successfully started.

AutoPilotSource

Defines the contract for the source of an auto pilot instance. This interface defines a method to retrieve all project ids to auto-pilot. Interface of this interface will be created by `AutoPilot` using object factory component.

ProjectPilot

Defines the contract to pilot a project. This interface defines a method to advance a project's phase given its project id. Advancing a phase means ending all opened phases that can be ended, and starting all scheduled phases that can be started. Interface of this interface will be created by `AutoPilot` using object factory component.

ActiveAutoPilotSource

An implementation of `AutoPilotSource` that retrieves all currently active projects that have auto pilot switch on in its extended property. It uses Project Management component to search all projects which are active and have auto pilot switch ("Autopilot Option") on in its extended property.

DefaultProjectPilot

A default implementation of `ProjectPilot` which will advance project phases. It delegates the start/end operations to the Phase Management component. It attempts to minimize phase operation queries by implementing a post order traversal of the phases, which means processing project phases in order from the earliest phase (having no dependencies) up to the latest phase. Logging Wrapper will be used to audit phase changes.

1.5 Component Exception Definitions

AutoPilotException

This is the base exception of all other exceptions thrown by this component. It's not thrown directly by any classes.

ConfigurationException

This exception is thrown by constructors of `AutoPilot`/`AutoPilotJob` and the various interface implementations if any error occurs while configuring itself.

AutoPilotSourceException

This exception is thrown by `AutoPilotSource` implementations when an error occurs while retrieving project ids for the source of an auto pilot.

PhaseOperationException

This exception is thrown by `ProjectPilot` implementations when an error occurs while ending/starting a project phase. It contains the project id and the phase that cause the exception.

1.6 Thread Safety

Most classes of this component are not guaranteed to be thread-safe. Since most of the work is actually delegated to the Project Management and Phase Management components, the thread safety of the underlying components determine this component's thread safety. This component doesn't attempt to provide synchronized access to those components. Multi-threaded application is advised to synchronize on the AutoPilot instance to ensure that only one thread is advancing project phases.

However in a typical usage (from the command-line), this component will only have a single-thread so thread-safety should not be an issue in this context.

The `AutoPilotJob` class contains a mutable field `done` that is initially set to `false`. It'll be set to `true` once `run` is finished. This may cause problems when multiple threads are invoking the `run` method and we're trying to check whether the job is done. When one of the threads is completed, the variable will become `true` despite other threads that may still be running. However this class is supposed to be used by the Job Scheduling component. The Job Scheduling component will run an instance of `AutoPilotJob` in a single thread, so in the context of the job scheduler, it's thread-safe.

2. Environment Requirements

2.1 Environment

- At minimum, Java1.4 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- Project Management 1.0 – used to retrieve projects to auto-pilot that met some certain criteria.
- Phase Management 1.0 – used to retrieve and update project phases.
- Project Phases 2.0 – defines the project phase model.
- Job Scheduling 1.0 – scheduler to auto-pilot projects.
- Command Line Utility 1.0 – command line argument utility.
- Search Builder 1.2 – build search filter for projects.
- Object Factory 2.0 – create interface implementations.
- Base Exception 1.0 – base of component exceptions.
- Logging Wrapper 1.2 – auditing of phase changes.
- Configuration Manager 2.1.4 – provide configuration functionality.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.phase.autopilot
com.topcoder.management.phase.autopilot.impl

3.2 Configuration Parameters

AutoPilotJob

Parameter	Description	Values
Operator	The operator name that is used to do auditing.	String, optional, default to “AutoPilotJob”

Additionally most classes have constructors that are configurable using object factory. Refer to the Object Factory component on how to set the parameters accordingly. Default values are used if default constructor are used to instantiate the object. *`xxx.class`* refers to the class’ full name.

AutoPilotJob

Parameter	Description	Values
namespace	The configuration namespace for the object factory.	String, default to *AutoPilotJob.class*
`autoPilotKey`	Object factory key for AutoPilot instance to use.	String, default to *AutoPilot.class*

AutoPilot

Parameter	Description	Values
namespace	The configuration namespace for the object factory.	String, default to *AutoPilot.class*
`autoPilotSourceKey`	Object factory key for AutoPilotSource instance to use.	String, default to *AutoPilotSource.class*
`projectPilotKey`	Object factory key for ProjectPilot instance to use.	String, default to *ProjectPilot.class*

ActiveAutoPilotSource

Parameter	Description	Values
namespace	The configuration namespace for the object factory.	String, default to *ActiveAutoPilotSource.class*
`projectManagerKey`	Object factory key for ProjectManager instance to use.	String, default to *ProjectManager.class*

DefaultProjectPilot

Parameter	Description	Values
`namespace`	The configuration namespace for the object factory.	String, default to *DefaultProjectPilot.class*
`phaseManagerKey`	Object factory key for PhaseManager instance to use.	String, default to *PhaseManager.class*

See an example config here.

3.3 Dependencies Configuration

Configure all dependent components accordingly.

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

None.

4.3 Demo

4.3.1 Auto-pilot projects (advance project phases) programatically

```
AutoPilot autoPilot = new AutoPilot();

// advance phases for all projects that are active and
// have Auto pilot switch on in its extended property
AutoPilotResult[] result = autoPilot.advanceProjects(“TCSUser”);

// print result
System.out.println(“Projects processed: “ + result.length);
for (int i = 0; i < result.length; i++) {
    System.out.println(“  ID: “ + result[i].getProjectId() +
        “ – “ + “ ended: “ + result[i].getPhaseEndedCount() +
        “started: “ + result[i].getPhaseStartedCount());
}

// advance phases for the given project ids
result = autoPilot.advanceProjects(new long[] { 123, 456 },
    “TCSDESIGNER”);

// advance phases for one project id
AutoPilotResult r = autoPilot.advanceProjects(111,
    “TCSDEVELOPER”);
```

4.3.2 Auto-pilot projects using scheduler at some configurable interval

```
// poll every 5 minutes (this uses the internal scheduler)
AutoPilotJob.schedule(“AutoPilot”, “AutoPilotJob”, 30);

// do some other tasks
// stop internal scheduler
// app cannot exit if scheduler is not stopped
AutoPilotJob.getScheduler().stop();



// create a job to do AutoPilot every 15 minutes
Job pilotJob = AutoPilotJob.createJob(“AutoPilotJob”, 15);

// use our own Job Scheduler
Scheduler scheduler = new Scheduler(“AutoPilot”);
scheduler.addJob(pilotJob);
scheduler.start();

// do some other processing


// stop jobs
scheduler.stop();
```

4.3.3 Command line interface

```
// poll every 5 minutes (in Unix use & to put in background)
java AutoPilotJob –poll &

// poll every 60 minutes (in Unix use & to put in background)
java AutoPilotJob –poll 60 &

// advance phase for all active projects
// with auto pilot switch on in its extended property
java AutoPilotJob –project

// advance phase for the given project ids
java AutoPilotJob –project 111,222,333

// configure with autopilot.xml in namespace
// with the autopilot key, poll every 15 mins, job name is TCSJob
java AutoPilotJob –config autopilot.xml –namespace autopilot
  -autopilot apKey –poll 15 –jobname TCSJob

// advance a particular project phase
java AutoPilotJob –config autopilot.xml –namespace autopilot
  -autopilot apKey –project 123
```

4.3.4 Advanced usage (override auditing & pluggability demo)

```
// override auditing
public class NewPilot extends DefaultProjectPilot {
    public NewPilot() {
    }

    protected doAudit(Phase phase, boolean isEnd,
        String operator) {
        // do some auditing here
    }
}

// create autopilot
AutoPilot pilot = new AutoPilot(
    new ActiveAutoPilotSource(),
    new NewPilot());

// advance phase for all active projects
// with auto pilot switch on in its extended property
pilot.advanceProjects(“AUDIT”);
```

5. Future Enhancements

- More `AutoPilotSource` implementations.
- More sophisticated `ProjectPilot` implementations.