# Schema Evolution

In this paper we need to consider the evolution of the domain schema and its impact on existing public API. Since we allow access to the rich domain model in the API, we should also provide a mechanism to allow the evolution of this model without impacting the API.

Let us consider an example to visualize this problem

Task has a tree structure with child tasks. A root task belongs to an entity called a project.

Let us say that the tree structure was removed in release 3.2 and a collection was tasks instead were added to the project entity along with a set of relationships to represent the tree hierarchy from release 3.2 onwards.

The framework needs to be enhanced to support the old API that works with the tree structure along with the new set of tasks and its set of relationships. In order to support this we introduce a few annotations to help the framework with this processing.

**public** **class** Task {

@Revision (droppedFrom=32)

**private** Set<Task> taskChildren;

@Upgrade(property="taskChildren", toRevision=32)

**void** upgradeTaskChildren(UpgradeEvent event) {

**if**(event.isCollectionAdd()) {

// create a collectionAddAction on the blueprint instance

} **else** **if**(event.isCollectionRemove()) {

// create a collectionRemoveAction on the blueprint instance

} **else** **if**(event.isSetter()) {

// perform whatever custom business logic is necessary

}

}

@Invariant(applicableFrom=32, applicableBefore=4)

**void** revisionSpecificInvariant() {

// This invariant is only valid for releases from 3.2 to any release before 4

}

}

A new annotation called “Revision” allows the developer to mark of attributes that are no longer considered part of the “active” model. The framework can guess the revision the API is working with by looking at the data that is sent along with the request. For e.g., if the request has the “taskChildren” data then we know the API needs to work with data that is pre-3.2 release.

Continuing with the example from above, the data represented by taskChildren is now captured in the Project entity as a set of tasks and relationships between them. This is depicted as follows:

**public** **class** Project {

**private** Set<Task> allTasks;

**private** Set<Object> relationships;

@Downgrade(property = "allTasks", fromRevision=32)

**void** downgradeToTaskChildren(DowngradeEvent event) {

**if**(event.isCollectionAdd()) {

//create a collectionAddAction on taskChildren on the respective Task object

} **else** **if**(event.isCollectionRemove()) {

//create a collectionRemoveAction on taskChildren on the respective Task object

} **else** **if**(event.isSetter()) {

// Do what is necessary depending on business logic

}

}

}

The Upgrade and Downgrade annotations are used to mark methods needed to do data migration between the old and new schema (domain model).

Upgrade and downgrade methods will be performed after all the regular actions are executed

This ensures all the data will be available for the upgrade and downgrade to do its migration