Title: Pattern Integration - Variations of State

Authors: Jim Odrowski and Paul Sogaard, Sprint Corporation
Contact Name: Jim Odrowski
Email Address: odrowsk@ibm.net
Postal Address: 22754 West 220th St., Spring Hill, KS 66083
Phone Number: (913) 534-3653

Abstract

In working with a large application development effort a number of challenges involving the management of object states have arisen. The application's developers have embraced the State pattern as the fundamental way to manage an individual object's states. But through the course of development a number of complications have been identified that deal with object states and the dependency between the states of related objects. Some variations on the basic State pattern are needed to solve these design problems. The following patterns build upon the State pattern, integrating it with other patterns, to help form solutions to the problems.

Patterns

I. Composite State

II. Managing State Dependencies

III. Decorated State

IV. Orthogonal States

V. Current and Pending State

VI. Collections of State

VII. Restoring Previous State

Note: Supporting diagrams are not attached, but are available.

I. Composite State

Problem

A composite object has leaf components that can be in a number of different states. We want to be able to trigger a state transition at the composite level, cascading state changes down to each leaf component or sub-composite.

Context

A Financial Portfolio consists of Investments which could be of different types (such as Stock, Bond, Mutual Fund, Real Estate, etc.). Investments can be organized into sub-portfolios (such as a Stock Portfolio). Each Investment has a state (such as bought, sold, ready to sell), but Portfolios do not have a state, they just hold onto Investments.

We may transition the state of each Investment separately, or we may want to transition state from the "root" level (Financial Portfolio), or from a sub-portfolio level (Stock Portfolio). For example, a customer wants to sell all of their Stocks, so transition their entire Stock Portfolio to "ready to sell". Or, if their entire Financial Portfolio was sold, transition all Investments to "sold". Refer to Context Diagram.

Solution

The Composite and State patterns work together for this solution. A Portfolio is structured as a composite, with each Investment as a leaf component. Each Investment leaf component holds a reference to a State object, representing its current state. The State pattern is implemented to manage the state transitions of an Investment. Refer to Solution Diagram - Object Structure.

The key to the two patterns working together is the abstract Component class of the Composite structure, which defines the public interface for changing state. A composite (a Portfolio) will implement thechangeState request by getting its components and sending each one the changeState request. The composite itself does not have a state, therefore, it has no transition to make. A leaf component will implementchangeState by accessing its current State object and delegating the request to it. Thus, state change requests can be cascaded down from any level of composite. Refer to Solution Diagram - Object Interaction.

Consequences

You must determine what to do in the event that a leaf component rejects a state change request driven down by a composite. In some cases, a notification can be returned addressing leaf components that could not transition as requested, but allowing processing to continue. In other cases, it may be necessary to stop processing and "back out" all previous state changes (see the pattern: Restoring Previous State).

II. Managing State Dependencies

Problem

Sometimes dependencies exist between the states of related, but different objects. These objects may be part of a container-content structure. State transitions can be driven by either level, the container may drive state changes down to its contents, or the contents may reach a "collective" state that drives a state change up to the container. The problem is how to manage the state dependencies between these different objects.

Context

Project management tools deal with projects and the tasks which make up the project. Work is done at the task level. The completion of the individual tasks can drive the completion of the project as a whole.

This pattern focuses on the design problem of coordinating state dependencies between projects and tasks. In the domain addressed, a basic project/task management tool was needed by a group of engineers. The engineers organized their work in a project. The project consisted of installation tasks which were completed in the field. Projects have three possible states. Tasks also have three possible states. The collective state of all tasks can drive a change in state of the project. Also, a project can drive the state transitions of all its tasks.

Entry conditions must be tested to determine if a requested state change is allowable. Some of the entry conditions may be for the container object to first check the state of its dependents, and likewise for a dependent to first check the state of its container to determine if a state change is valid. Refer to Context Diagram.

Solution

The following design patterns were integrated to form a solution to this problem:

> State - used to manage the individual states of Project and Task.
>
> Mediator/Observer - a "StateChangeManager" was used to observe and coordinate state changes between Projects and Tasks.

The key to this solution is the StateChangeManager class that mediates state changes between a Project and its Tasks. A Project and each of its Tasks keep a reference to an instance of the same StateChangeManager. The StateChangeManager has a reference to the Project. The Project holds on to a collection of its dependent Tasks. Refer to Solution Diagram.

When a Project state change drives Task state changes, the Project transitions to the new state, then notifies the StateChangeManager to trigger a state change on the Task. The StateChangeManager then triggers a transition on each Task held by the Project.

When the collective state of all Tasks drives a state transition at the Project level, each Task transitions to the new state, then notifies the StateChangeManager of the change. The StateChangeManager checks to see if all Tasks have reached the required state. Once the collective Task state is reached, the StateChangeManager triggers the appropriate state transition at the Project level.

A ChangeManager class was discussed in the GOF Design Patterns book as another application of the Mediator pattern used for coordinating complex updates. The ChangeManager mediates between subjects and observers.

> "When an object changes, it notifies theChangeManager, which in turn coordinates the update by notifying the object's dependents."

In the Project/Task example, the Tasks are the Project's dependents. The StateChangeManager is acting as the Mediator, and both Project and Task can be acting as an Observer of state changes on each other.

Consequences

Maintenance of the state dependency rules is a concern. TheStateChangeManager has the responsibility for encapsulating these rules, and at least they are isolated in one place.

You must determine what to do in the event that a state change request is rejected by a dependent object. It may be necessary to restore the requestor's previous state (see the pattern: Restoring Previous State) if all its dependent components cannot commit to the state change request.

III. Decorated State

Problem

One object may "wrap" functionality around another object. Each object (the "wrapped" object and the "wrapper") have their own states. The object's respective states have their own set of transition rules, but in certain contexts a state change on one object can trigger a state change on the other object. For instance, the "wrapper" object changes state and causes a state change on the "wrapped" object.

Context

In an equipment engineering function a piece of equipment inventory can be created by an engineering Task which is part of a Project. The Task has its own set of states and the Equipment Item it creates has its own set of states. As the Task changes state, for example from "released" to "accepted", a corresponding state change is triggered on the Equipment Item itself, from "planned" to "actual". Once the Equipment Item moves to the "actual" state, the Task information is logged to the Equipment Item activity log and the Task and its state (Task state) are no longer needed. Refer to Context Diagram.

Entry conditions must be tested to determine if a requested state change is allowable. Some of the entry conditions may be for a Task object to first check the state of the Equipment Item to determine if the requested state change is valid.

Solution

The following design patterns were blended to form a solution to this problem:

> State - used to manage the individual states of Task and Equipment Item.
>
> Decorator - the Equipment Item may be "decorated" by a Task. The Task "wraps" functionality around an Equipment Item which is optional, and may be withdrawn when no longer needed.

The key to this solution is the Task object which acts as a Decorator to the Equipment Item. The Decorator pattern is documented in the GOF Design Patterns book. It shows how to enclose an object within another object that adds behavior. The Decorator adds responsibilities to an object dynamically without subclassing. Decorator provides more flexibility than subclassing since "decorating" another object may be optional, and responsibilities added by the Decorator can be easily added or withdrawn. The Decorator acts as a "wrapper" around another object. Refer to Solution Diagram - Object Structure.

The Task has its own set of states. As a Task state transitions, one of the state transition functions it triggers is to change the state of the component it is "decorating", the

Equipment Item. Refer to Solution Diagram - Object Interaction. An Equipment Item will not even know that it is being decorated, likewise clients will not be able to tell the difference between a decorated object and an undecorated one. Once the Equipment Item is in the "actual" state, the Task's work is completed and therefore this Decorator can be withdrawn from the Equipment Item.

Modifications to an Original Instruction can be made (Exception Instruction). They supercede the Original Instruction, having their own Instruction State which also drives the state change on the Equipment Item (just as the Original Instruction's state did). Refer to Decorator Instance Relationships diagram.

Consequences

There may be complex state dependency rules that need to be handled. For example, what happens when the underlying Equipment Item transitions its state to "defective", would any associated Tasks be invalidated (until the item is repaired)?

"Delete" rules also need to be handled. Should deletes cascade from the Equipment Item to all of its associated Tasks?

IV. Orthogonal States

Problem

An object holds multiple states at a given time, with one state dependent upon another state. The behaviour of the object is dependent upon all of the states that the object can have at one time. How does a state manage its dependent states (or substates) and the dependent state transistions?

Context

In a production management system, a Machinery Item Object is used to track individual equipment components. The management system can track items that are currently installed, items that were previously installed, and items to be installed in the future. Each item installed can be in a variety of states that are unique to installed Machinery Items. Uninstalled and Future Machinery Items each have their own set of unique states

Each Machinery Item has two states at a give time. The Installation State tells if an Machinery Item is planned, installed, oruninstalled. The Operation State tells the operational status of the Machinery Item, and has unique states dependent upon the Installation State. For example, a Machinery Item can be in the “Installed” state (Installation State), but can be either on-line, off-line, or defective (Operation State). The available operation states vary between Installation States, offering unique operational state transitions.

Solution

The following design pattern(s) were used to form a solution to this problem:

State - used to manage the individual states of Installation.

- within each installation state, used to manage the states of Operation.

The key to this solution is to use the state pattern to manage not only the (Installation) status of the Machinery Item, but also to manage the (Operation) states for each Machinery Item status. In the example above, when a Machinery Item is in an “Installed” state, its operational status could be “on-line”, “off-line” or “defective”. In effect, a hierarchy of states is created by assigning a Machinery Item Installation State to each Machinery Item Installation Item, and then assigning a Machinery Item Operation State to each Installation State.

The state pattern can be used recursively for problems where state can vary within state. To implement this, the state pattern is used on the domain object containing the highest level state (Machinery Item would have Machinery Item Installation State in the above

example). For each subclass of the state pattern, another state pattern would be applied (each concrete instance of Machinery Item Installation State would have a Machinery Item Operation State). If yet a finer granularity of states were needed, they could be added to the leaf state(s).

Consequences

Creates many classes due to the hierarchy of State managers, but encapsulates the behavior of each leaf level state.

V. Current and Pending State

Problem

An object can have a current state and 1-n pending states. How does the object manage these internal states and present a public interface for accessing state in a safe and understandable way?

Context

An Machinery Item can have one current state (ie. working, in-repair). The same item can have 1-n planned states (ie. planned install, planned change, planned removal) which represent future actions for the item. For example, the current state is "working", but there is a "planned change" pending which will take effect on a certain date. When the planned change takes effect, it willsupercede the current state.

A new item is created in the "planned install" state. After it is installed and ready for operation it is transitioned to the "working" state.

In some situations it is important to identify the fact that a planned state exists, in order to alert someone to a pending change, or to avoid possible conflicting actions. Also, past states should be logged for identifying an object's state change history.

Solution

This solution utilizes the State pattern to manage the states of an object, its allowable transitions, and transition actions. Each specific object state is represented as a state object. These state objects are subclassed off of a common abstract state class.

The key to this solution is giving the context object a reference to a StateManager, which has the responsibility to manage all possible states. The context object delegates all state-specific requests to its StateManager. The knowledge of managing different state categories (current state, pending states or past states) is defined in the StateManager. Refer to Solution Diagram.

For example, the initial state of a MachineryItem is "planned install". When the object is initialized, the initial state object is created and theMachineryItemStateManager places it in its pendingStates collection. When the item is installed, the item's state istransitioned from "planned install" to "working". TheMachineryItemStateManager places the working state in its currentState variable. The plannedState object is removed from the pendingStates collection and logged to the pastStates collection.

When a MachineryItem is asked to display itself, theMachineryItemStateManager can identify the currentState as well as any pendingStates. This will give visibility to pending actions on the item.

When the MachineryItem is in its initial state of “planned install”, this state object will be held in the pendingState collection and the currentState will be “nil”. If the item is asked for its current state, the MachineryItemStateManager could first check to see if the currentState variable is “nil”, if so, it can by default, return the most recentpendingState object.

Consequences

Since multiple pending states can exist, it is necessary to keep these objects in a planned date sequence. It may also be necessary to do validation of the pending states as they are added to the collection, ie. is this a valid pending state based upon its predecessor state?

VI. Collections of State

Problem

A context object may hold a reference to a collection of other objects. The objects held on to may be in a number of different states. Typically, access to these objects will be via their state. A request, for example, could be: "for Object A, give me all Object B's it references that are in State C". In high volume situations, traversing through Object A's entire collection of Object B's, interrogating the state of each Object B, may not meet performance objectives. A quicker way to retrieve objects by state is needed.

Context

A Sales Office tracks Orders, which can be in different states (e. pending, deferred, past due, completed, cancelled). Typically, access to a Sales Office's Orders will be by state. For example, give me all "pending" orders at the Kansas City Office. The context is the Sales Office, the target object is an Order, and the target object has a state, in this case "pending". Refer to Context Diagram.

Solution

Set up the context to hold separate collections of objects, classified by their state. Create a single Container object that the context holds a reference to. The Container keeps a collection of StateContainer objects which are identified by a particular state and in turn hold the collection of target objects in the same state. Refer to Solution Diagram - Object Structure.

When a new target object is created, it is added to a StateContainer collection based upon its current state. If a StateContainer for a particular state does not yet exist, it is dynamically created. Therefore, aStateContainer is only created if target objects exist in that state. Since target objects will be transitioning from one state to another, a StateContainer may be "drained" of all its objects, thus allowing it to be removed.

Retrieval of target objects can now be handled directly by the context asking its Container for all target objects in a particular state. The Container just has to locate the particular StateContainer in its collection of StateContainers, and return the instance, or notify the requester that target objects in that state do not exist. This check for non-existence is handled much more efficiently than traversing large collections of target objects looking for a particular state, only to find that none exist. Refer to Solution - Instance Diagram.

Consequences

Whenever a new target object is created it is given a reference to its context. In turn, the context is given a reference, or "back pointer" to the target object. In using this pattern,

the context's Container must now receive the target object, which it adds to the appropriate StateContainer. A question is whether or not the target object will ever be asked for its context, or will access to target objects always be through the context. If the latter is the case, the target object may not need a reference to its context.

If the Container receives a request to return all target objects in all states, it must concatenate the StateContainer collections. If asked for all target objects in state X and state Y, it must form a union of StateContainer collections.

Consideration must be made to ensure that deletion rules are enforced. For instance, if a target object is deleted, it must be removed from any StateContainer collections that reference it.

Consideration must also be given to specific database implementations which may offer indexing and retrieval by indexed values. Such a solution could eliminate the need for this pattern. Although, to limit the application's dependency on a particular database implementation, a more general solution may be best.

VII. Restoring Previous State

Problem

An object can move from one state to another, but frequently there is a need to go back to an object's previous state. This is most relavant when an object is part of a larger composite object (refer to composite state) and one of the objects cannot make a valid state transition. In that case, all of the objects in the cascaded state change must be returned to their previous state.

Context

Project management tools deal with projects and the tasks which make up the project. Work is done at the task level. The completion of the individual tasks can drive the completion of the project as a whole. If one of the tasks cannot transistion to the completed state, all tasks that have been transistioned to the next state must be returned to the original state

This pattern focuses on the design problem of coordinating state changes of multiple tasks. In the domain addressed, a basic project/task management tool was needed by a group of engineers. The engineers organized their work in a project. The project consisted of installation tasks which were completed in the field. Projects have three possible states. Tasks also have three possible states. The collective state of all tasks can drive a change in state of the project. Also, a project can drive the state transitions of all its tasks.

Entry conditions must be tested to determine if a requested state change is allowable. Some of the entry conditions may be for the container object to first check the state of its dependents, and likewise for a dependent to first check the state of its container to determine if a state change is valid. If the state change takes place for all tasks in a project, but only half of the tasks are moved to the next state before one of the tasks cannnot transition, then the previously transitioned tasks must be restored to their previous status.Refer to Context Diagram.

Solution

The following design patterns were integrated to form a solution to this problem:

> State - used to manage the individual states of project and task.
>
> Memento - used to capture the state of both the project and its tasks prior to making state changes.

As part of the state change in an object, the Project State or Task State would use the Memento pattern to copy the existing state of the object. This copy would be available

until the next state transition, in which the state would transition (forward) to the next state, or (backward) to the previous state. This solution would store a Memento of the Project State or Task State, but not a Memento of the Project Object or Task Object. This would allow for minimal resource requirements when implementing a restoration of previous state.

Consequences

All of the consequences of the Memento pattern that are discussed in the GOF book apply to this combination of patterns. Since the scope of the Memento should be limited to Project State or Task State, the expense of implementing Memento with State should be minimal.