STATE SPACE EXPLORER PATTERN

BY:
Martin Fontaine and Dwight Deugo

CARLETON UNIVERSITY
OTTAWA
April 26, 1999

Pattern Name

State Space Explorer

Introduction

The present document addresses problem domains that can be solved using a state exploration system. The goal of a state exploration strategy is to identify a series of operations in order to reach a certain “goal state” from a defined “initial state”. State exploration strategies are tightly coupled with the problem domain, however, some common elements could be reused. This pattern provides a solution to common problems encountered in the elaboration of state exploration systems. This pattern does not provide a solution to the time complexity of such problems and is not intended to provide solutions to problems having large state spaces.

Context

You need to identify a series of operations to perform on a given system in order to successfully reach a desired state (your goal). In addition, you are facing a problem for which the only apparent way to reach the desired state seems to be a solution based on trial and error requiring many operations.

Fortunately, you can decompose your problem in terms of independent and subsequent states, and a set of well-defined operations is known. The execution of each of these operations on the system leads it into another state.

The initial state and the desired state(s) (an acceptable solution to your problem) are known in advance. Also, you know how to identify an invalid state that can be reached but will never lead you to the desired state. The number of possible states of the system can be significant (it could be in the order of magnitude of thousands depending on the resources of your system in terms of memory and CPU).

Many problems solved by computer scientists (especially in artificial intelligence) can be expressed in terms of an initial state, a series of operations, intermediate states and a final state [LS]. A good example is a chess game. The initial state is the initial chessboard. The operations are all the legal moves of the pieces. The intermediate states are all the possible board configurations. Final states include a stalemate or a checkmate[1].

Identifying the initial state and the final state of a given problem is, most of the time, the easiest part. Also, rules and legal operations that can change the state of the system are usually well defined. If we refer back to the chess game example, every chess player knows the initial board configuration, the legal chess moves, and, they know how to recognize a winning (or stalemate) board configuration.

[1] The chess game is a good example to explain the state exploration principles, but it is important to mention that this pattern does not provides a solution to the chess problem which is very complex and involves a very large state space.

On the other hand, finding the sequence of operations that lead you from the initial state to the desired state is complicated. Again, in the chess game example, it is not trivial to decide which move to do to win the game against a good chess player!

Problem

How can you identify of a series of operations, performed on a given system, to lead it state by state from its initial state to a desired state?

Forces

- **Reusing a similar object-oriented structure for similar problems avoids spending additional development time**. If some components of a system have provided successful results for existing systems, it is an advantage to reuse those components in other problem domains. Reuse avoids development time by sharing a proven and robust object in many applications.

- **Exploring a state space can be very demanding on memory if the system keeps track of all intermediate states**. In most real life situations, many states must be explored to reach the goal state. Many approaches tend to memorize the intermediate states to allow "backtracking" if a given sequence of operations leads to a non-successful state. A "backtracking" algorithm keeps the intermediate states into the system's stack in case it would have to generate another subsequent state from them after a non-successful exploration. Most of the time, it is implemented with recursive algorithms.

- **A state space exploration strategy developed for a specific problem may save computational time and memory space.** In fact, very efficient techniques may be used to accelerate the discovery of the desired state. These techniques are called "heuristics" and they are very specific to the problem domain. Heuristics give an indication of which states are probably "good" candidates for further exploration. Consequently, the usage of heuristics increases the probability of reaching the goal state rapidly.

- **A design that encapsulates state space exploration concepts into separate objects encourages fine-tuning of the exploration strategy.** If the design adopted by the developer encapsulate the state space exploration concepts, many heuristics may be tried on a specific problem just by changing the implementation locally in the appropriate objects. Experimentation and testing will help to find an appropriate heuristic.

- **Recursive code is complicated to understand and is demanding on system memory.** Recursive algorithms have been frequently used to solve state space exploration problems. Recursive algorithms may be intuitive to develop in situations where "backtracking" is needed when a series of operation leads to a non-valid state. Unfortunately, recursive algorithms keep all the intermediate states in the system stack. If many states need to be explored, the system may possibly run out of memory. Also, recursive algorithms are complicated to understand for a new developer that is not familiar to the problem domain.

- **A structured exploration technique is needed to avoid missing a possible solution**. If a random technique is used instead of a structured exploration technique, how can you be sure that your "exploration" technique will not cycle or explore continuously the same states? How can you be certain that a solution to the given problem exist or does not exist? If the desired state is reached, how do you retain the sequence of operations that leads to the goal? How can you find the operations that will reach your goal state optimally? If a solution is found, how can you guarantee its optimality?

Solution

The following paragraphs illustrate a step-by-step approach that could be used to design a system that resolves the presented problem. The solution explains how to define states and state transitions in order to reach a predefined goal-state efficiently.

1 - Elaborate a structure that can represent any possible state in which your system can exist.

Encapsulate this data structure in an object. The internal representation of this object depends on the problem domain. Use the State Pattern [GHJV] to represent each state. This object must provide public methods to express its current state. Also, this object must implements an execution method that will take an operation as a parameter. The returned value of this method is another state object resulting from the execution of the operation on the previous state.

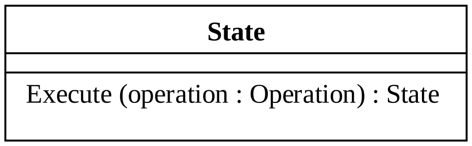


Figure 1

2 - Identify the initial state of your system.

The first instance of the “State” class will correspond to the initial state of your system. The nature of this initial state depends on the problem domain.

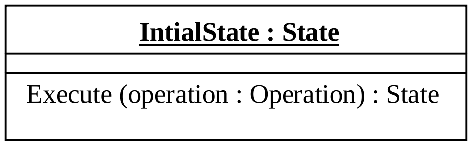


Figure 2

3 - Identify the desired state of your system.

At this point, you need to introduce an object that is more specific to the problem you are trying to solve. This object will have the responsibility of detecting that an instance of a "State" object corresponds to a desired state (we use the method IsFinal() : Boolean). By using this additional object, the state object has a limited set of responsibilities so it can be reused easily. This object will be referred as a "StateTransition". Note that the initial state previously defined will initially be part of a "StateTransition" object.

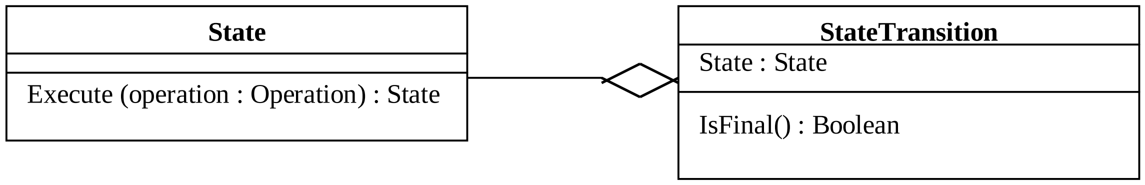


Figure 3

4 - Identify all the possible operations you can perform on your system.

An operation is an instance of the "Operation" class. An operation has to be accepted as a parameter by the "Execute(operation: Operation) : State" method of a "State" object. The responsibility of the "Operation" object is to send a message to the "State" that uses it as a parameter in order to generate a subsequent state from it.

An "Operation" object will implement the "performOn(aState : State) : State". The type of message sent depends on the operation. Also, all the possible instances of operations you can perform for a given problem must be accessible via the "StateTransition" object.

As an example, if a robot can move forward, backward, left or right, four instances of the "Operation" class will be accessible by a method in the "StateTransition"object. If a "move forward" operation is executed by a certain state of the robot, a resulting state will be returned expressing the robot in its new position.

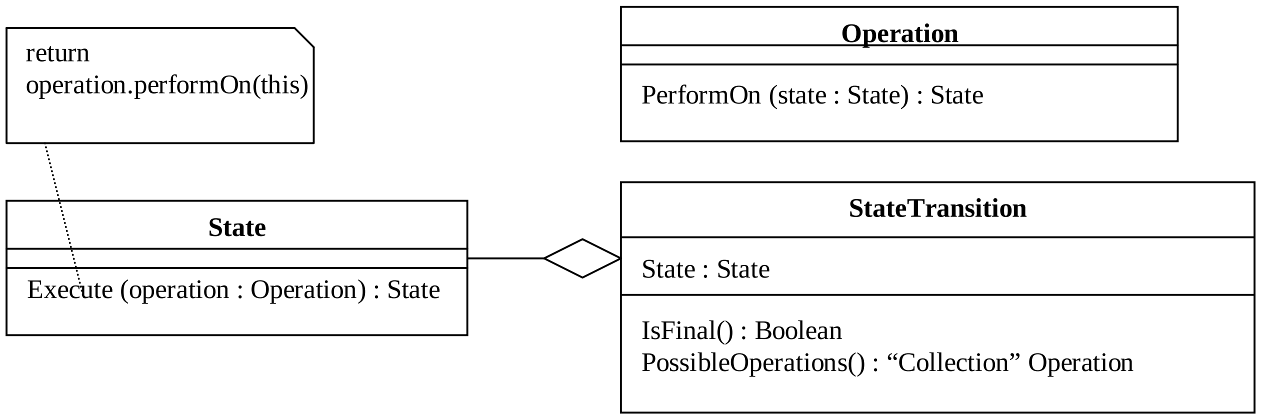
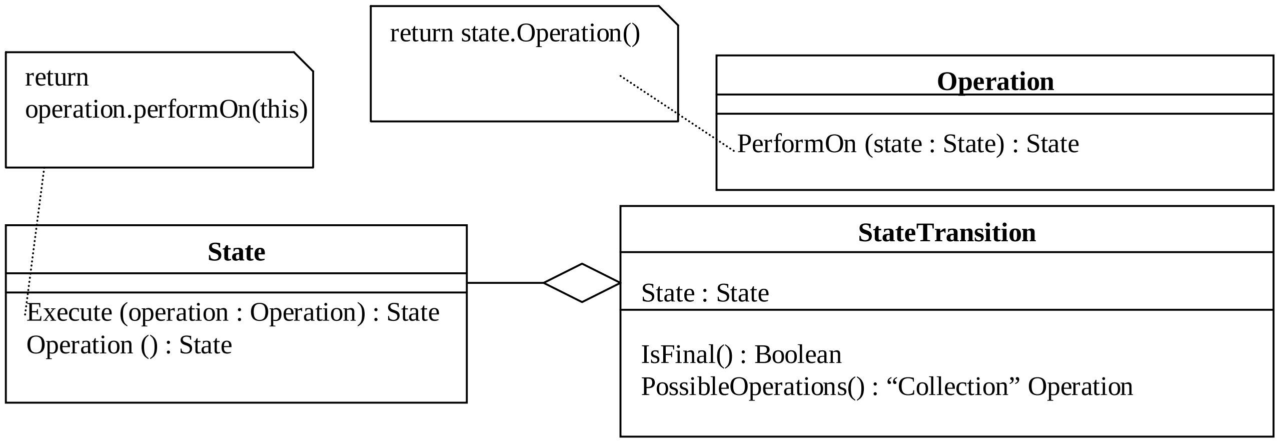

return
operation.performOn(this)
Operation
PerformOn (state : State) : State
State
Execute (operation : Operation) : State
StateTransition
State : State
IsFinal() : Boolean
PossibleOperations() : "Collection" Operation

Figure 4

5 - Define the behavior of your operations on the state of the system.

For each of your operations, it is necessary to define a method in your “State” object to concretely implement the effect of the operation on the current state. This situation is appropriate for using the Visitor Pattern [GHJV]. Again, the result of an operation is a subsequent state that is derived from the current state and the type of operation sent to it. Note that if the “State” object receives an “Operation” that it cannot execute, the NULL state [BW] object has to be returned to indicate that this operation cannot be performed in this state. As an example, if a certain robot is facing a wall and is asked to move forward, no possible state can be generated from this situation.

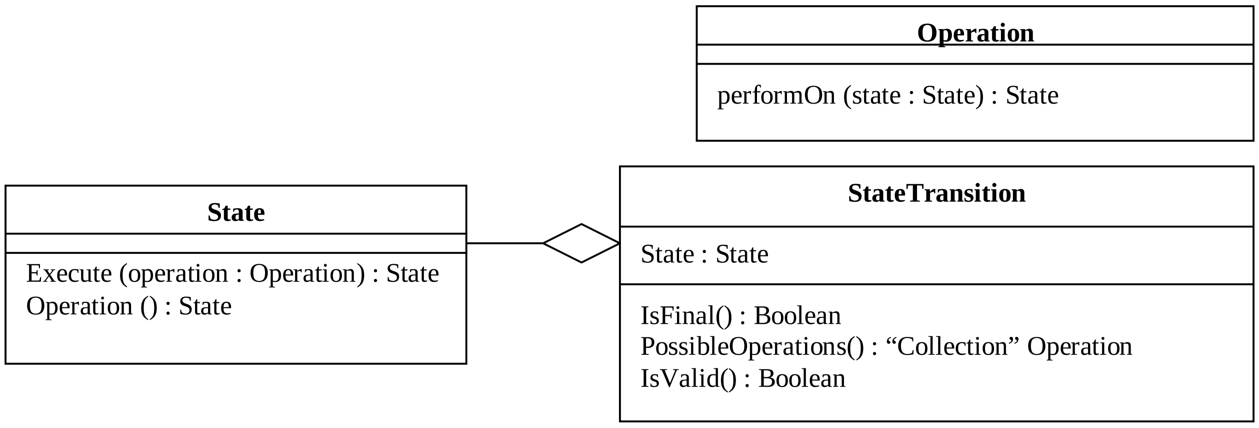


return operation.performOn(this)
return state.Operation()
Operation
PerformOn (state : State) : State
State
Execute (operation : Operation) : State
Operation () : State
StateTransition
State : State
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation

Figure 5

6 - Identify which states are not valid in the context of the problem.

Some states may be legally generated by your state exploration strategy without being valid to the problem domain. These states are considered invalid in the context of the problem. It is the responsibility of the “StateTransition” object to select its “State” and determine if it is a valid state. An example of a legally generated but not valid state could be the following: At a certain time during a chess game, you are allowed to move your queen in a certain position. However, by moving your queen, you expose your king to an opposite piece. This move is legally generated but not valid. Add the method “IsValid() : Boolean” to the “StateTransition” object to support this activity.



Operation
performOn (state : State) : State
State
Execute (operation : Operation) : State
Operation () : State
StateTransition
State : State
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
IsValid() : Boolean

Figure 6

7 - Determine the cost of a state transition.

Every time an operation is performed on the system, a cost is associated with it. A cost is determined by the operation and on which state it is performed. It is possible that in some systems, all operations performed on any state cost the same. Even in this situation, it is important to keep track of the accumulated cost to help the exploration strategy to choose an appropriate state for further exploration. An object associated with the “StateTransition” object will describe the cost. This object will be referred as a “Cost” object. The “Cost” object represents the accumulation of all the costs of each transition accomplished from the initial state to the current state represented by the “StateTransition”. Because the cost depends on the operation and on which state it is performed, the “Cost” object will be updated when performing an operation. To do this, the “Cost” object will be passed as a parameter, every time an operation is performed (See Pseudo Code, Table1 at the end of this section). A good example to express the cost aspect of a state exploration is to consider the shortest path problem. If 4 roads of 3km each have to be taken to reach a certain state, the accumulated cost will be 12km. If another state can be reached by taking 2 roads of 5km, the accumulated cost will be 10km. The accumulated cost helps the exploration strategy to wisely choose good candidates among different possibilities.

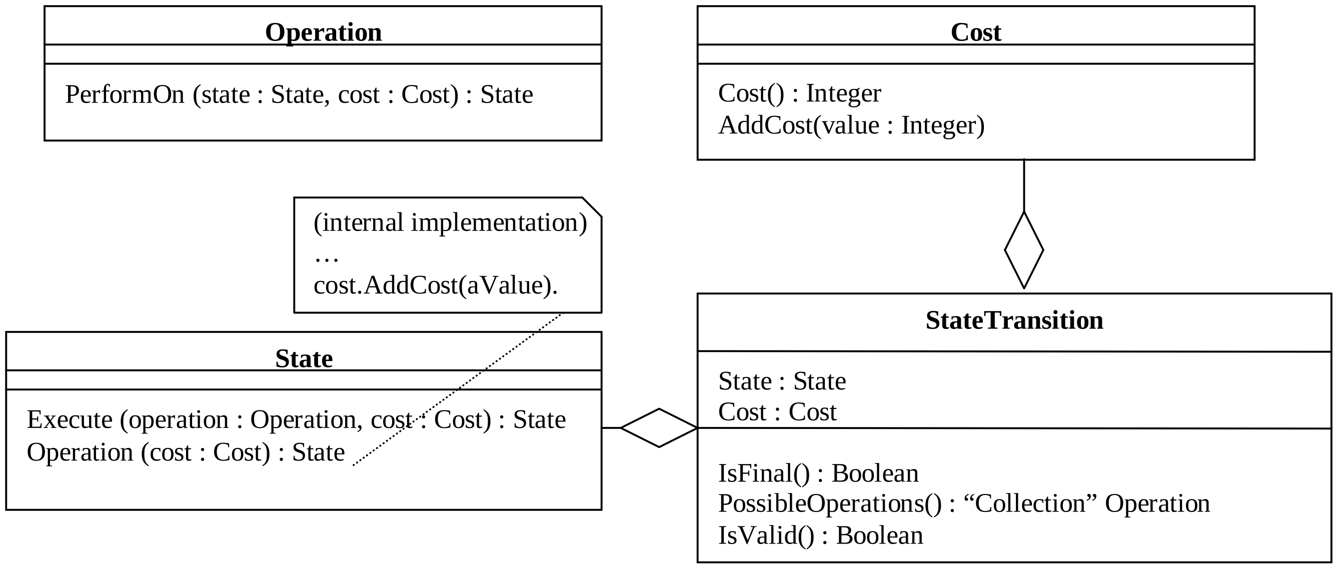

Operation
PerformOn (state : State, cost : Cost) : State
Cost
Cost() : Integer
AddCost(value : Integer)
(internal implementation)
…
cost.AddCost(aValue).
State
Execute (operation : Operation, cost : Cost) : State
Operation (cost : Cost) : State
StateTransition
State : State
Cost : Cost
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
IsValid() : Boolean

Figure 7

8 - Elaborate an exploration strategy.

Having defined the concept of state, state transition, cost and operation, it is now necessary to design the object that allows the exploration of the state space. This object is the “StateSpaceExplorer”. It contains an ordered collection of “StateTransition” objects. The “StateTransition” objects contained in this collection are “candidates” for further exploration. Consequently, this collection will initially contain an instance of a “StateTransition” referencing an instance of “State” which is the initial state. This is the first state that will be explored.

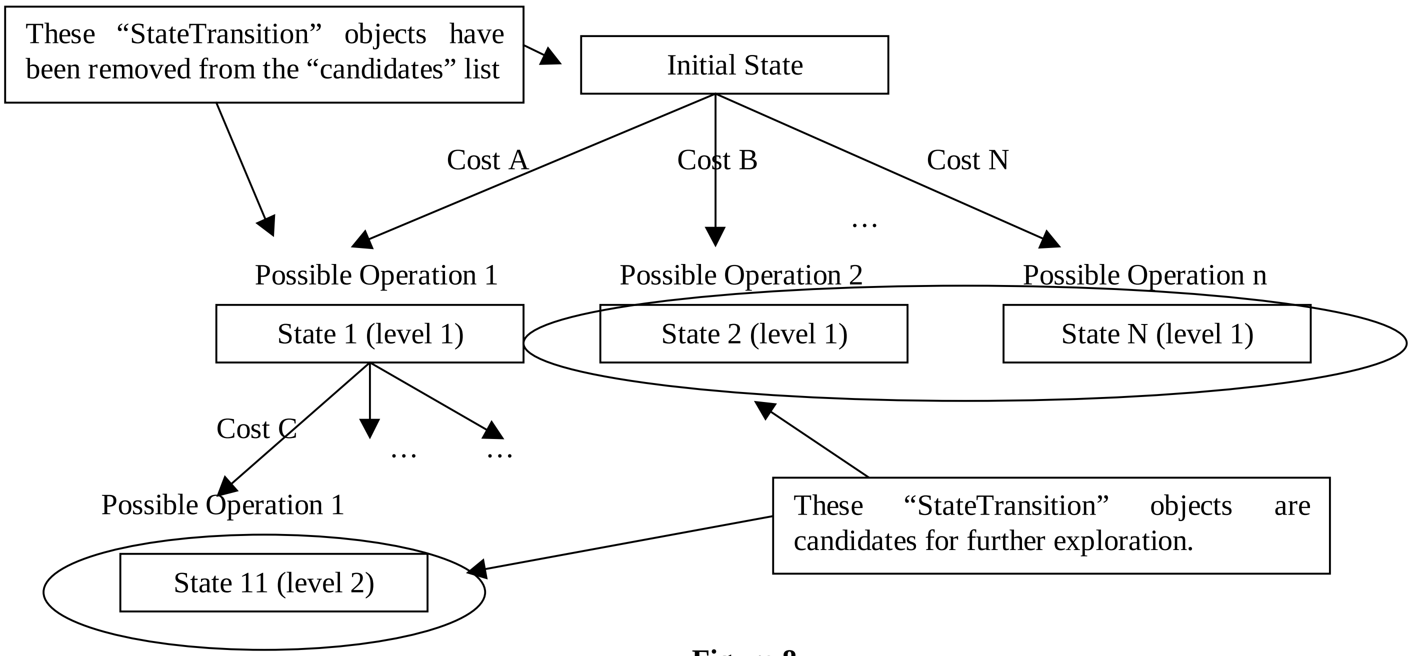


Figure 8

As shown in Figure 8, when a state is explored, it is removed from the "candidates" collection and some subsequent states are generated from it ("GenerateSubsequentStates() : "Collection" StateTransition" method in Figure 9). These subsequent states are generated by the execution of all the possible operations on the explored state. It is important to mention that all the possible operations will not necessarily lead to a subsequent state. As an example, assume that a robot can move left, right, back and forward. If this robot is facing a wall, it will not be able to move forward. Applying all the possible operations on the robot, which is in the state "facing the wall", will give you three subsequent states. To summarize, if *n* operations are possible, up to *n* subsequent states can be generated.

The generated states are added at the end of the "candidates" collection. Because the intermediate states are not kept, we need to keep track of all the operations executed from the initial state, up to each "candidate" state. A "StateTransition" object represents a candidate state, so this object will have the responsibility of gathering all the operations performed to reach its current state. A way to record executed operations has to be provided in order to display, or to provide the solution to the client of the state space explorer.

It is also important to mention that the "StateTransition" object keeps track of **the accumulated cost** resulting from the operations executed from the initial state to the current state. In Figure 8, the "StateTransition" object representing "State 11 (level 2)" will point on a "Cost" object that will represent the summation cost of "Cost A" and "Cost C".

The only remaining task is to choose a state among the given collection of candidate "StateTransition" objects for further exploration. The way this state is chosen depends on the problem domain. This pattern provides the information to wisely choose a state at a given stage of the process. The criteria to choose the next state to be explored is composed of the accumulated transition cost and the description of the state itself. Whatever is your exploration strategy, the logic to select a candidate state will have to be implemented in the method "ExploreCandidate()" of the object "StateSpaceExplorer". The "ExploreStateSpace()" method simply calls the

“ExploreCandidate()” method until the desired state is explored or no candidates exists for further exploration (See Pseudo Code Table1 for a detailled description).

It is important to mention that if a generated state is identical to a previously generated state, a cycle could possibly occurs during the exploration strategy. In other words, a sequence of states could be repeated infinitely. A good trick to avoid those kind of futile exploration is to consider the candidate with the lowest accumulated cost for further exploration. Exploring a cycle will increase the accumulated cost infinitely. Also, if your state space is very large or infinite, you might consider having a limit in term of time or number of transitions to avoid infinite processing.

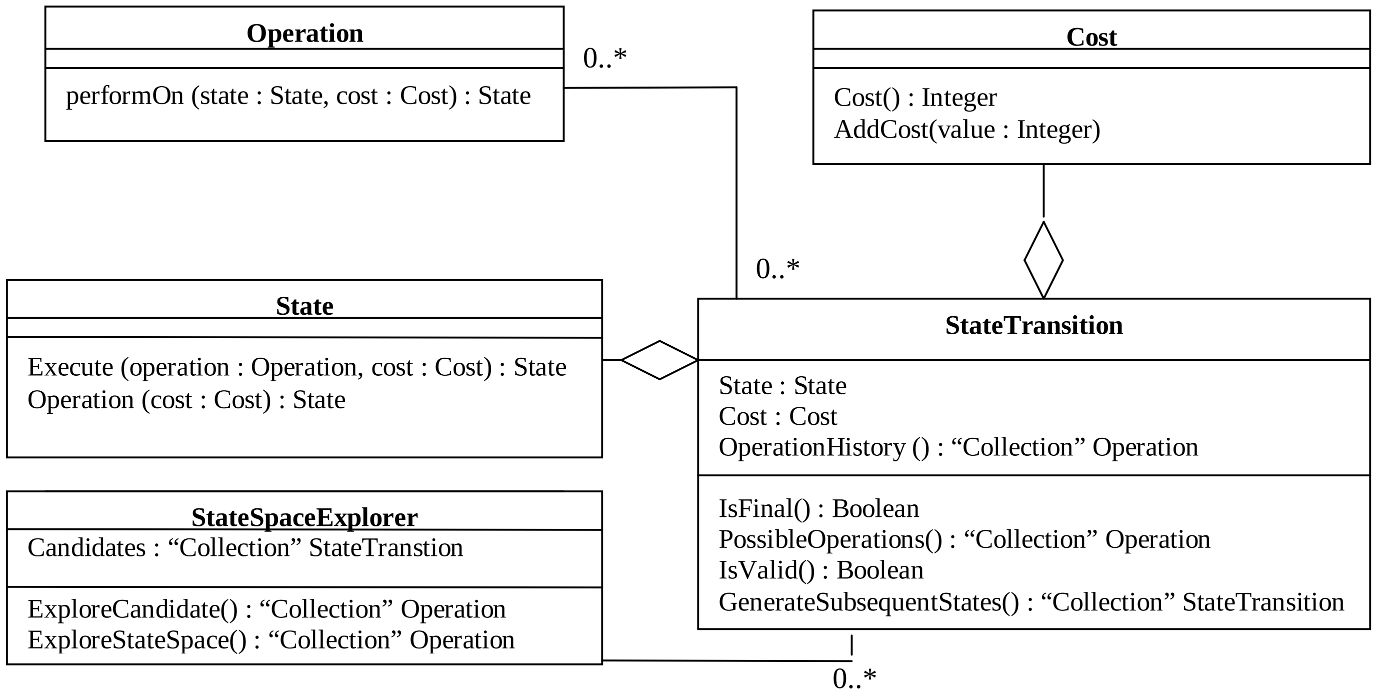

Operation
performOn (state : State, cost : Cost) : State
0..*
Cost
Cost() : Integer
AddCost(value : Integer)
0..*
State
Execute (operation : Operation, cost : Cost) : State
Operation (cost : Cost) : State
StateTransition
State : State
Cost : Cost
OperationHistory () : “Collection” Operation
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
IsValid() : Boolean
GenerateSubsequentStates() : “Collection” StateTransition
StateSpaceExplorer
Candidates : “Collection” StateTranstion
ExploreCandidate() : “Collection” Operation
ExploreStateSpace() : “Collection” Operation
0..*

Figure 9

Pseudo Code

GenerateSubsequentStates()	Declare → ReturnedCollection, CopyOfStateTransition
	FOR (Operation this.PossibleOperations) DO
	    CopyOfStateTransition:= this.copy();
	    CopyOfStateTransition.State :=
	    this.state.Execute(Operation, CopyOfStateTransition.cost);
	    CopyOfStateTransition.OperationHistory().add(operation);
	    IF(CopyOfStateTransition.state not NULL)
	        ReturnedCollection.add(CopyOfStateTransition);
	END DO
	Return ReturnedCollection

ExploreCandidate()	1)select a state S among Candidates (S must return true on IsValid());
	2)remove S from Candidates;
	3)IF (S IsFinal())
	    3.1)return S.OperationHistory();
	  ELSE
	    3.2)add S.GenerateSubsequentStates() to Candidates;
	    3.3) return NULL;

ExploreStateSpace()	DO (Solution := this.ExploreCandidate())
	    IF (Solution not NULL)
	        Return solution;
	WHILE (Candidates not empty)
	Return NULL;

Table 1

Example

Many real life problems may be solved by a state exploration technique. For the current example, the State Space Explorer Pattern will be demonstrated on the "Shortest Path" problem.

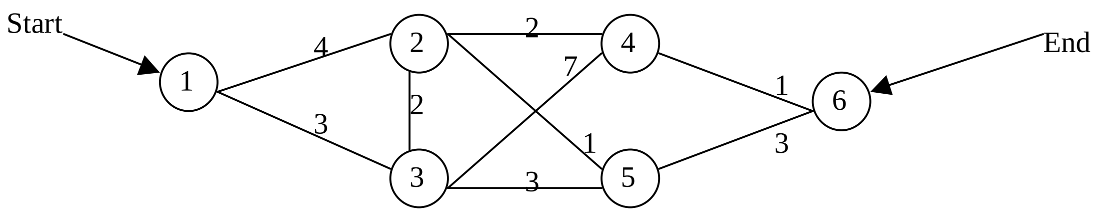


Figure 10

The objective is to reach node 6 starting from node 1 using the shortest path. The cost of a transition between two nodes is expressed on the line between the two nodes. In Figure 10, the shortest path from node 1 to node 6 is node 1, 2, 4 and 6.

Let see how the state explorer pattern can help to find the sequence of operations to reach node 6 from node 1.

1 - Elaborate a structure that may represent any possible state in which your system can exist.

We express the state in term of “Position”. A “Position” has a reference to an object that knows the graph configuration and the cost between each pair of nodes (The representation of this object is not explained in this article). Also, a “Position” contains the current node and a list of nodes that have never been explored.

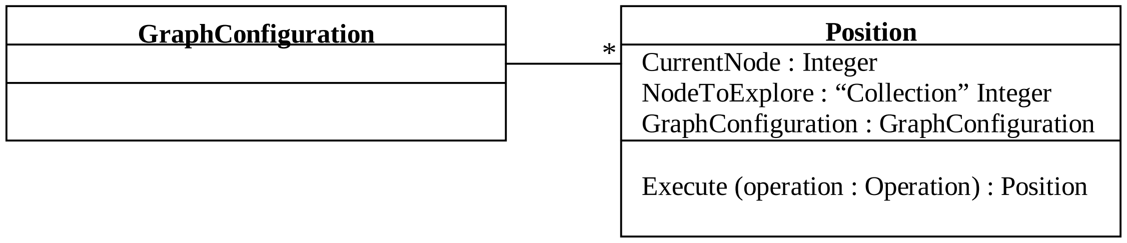

GraphConfiguration
*
Position
CurrentNode : Integer
NodeToExplore : “Collection” Integer
GraphConfiguration : GraphConfiguration
Execute (operation : Operation) : Position

Figure 11

2 - Identify the initial state of your system.

In this particular problem, the first instance of the “Position” object will have the following representation (assuming that the initial node is 1 and there are 6 nodes in the graph):

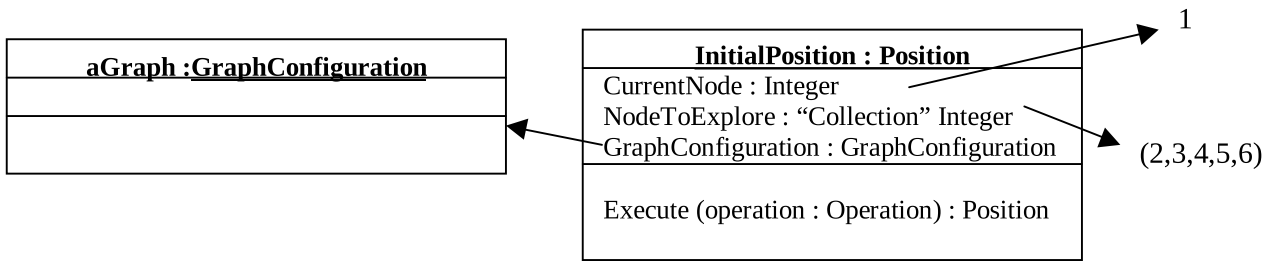

aGraph :GraphConfiguration
InitialPosition : Position
CurrentNode : Integer
NodeToExplore : “Collection” Integer
GraphConfiguration : GraphConfiguration
Execute (operation : Operation) : Position
1
(2,3,4,5,6)

Figure 12

3 - Identify the desired state of your system.

According to the given representation of a “Position” we can easily identify the desired state as being a “Position” with the current node value equal to 6. At this point we introduce the “StateTransition object that can be named “Path”.

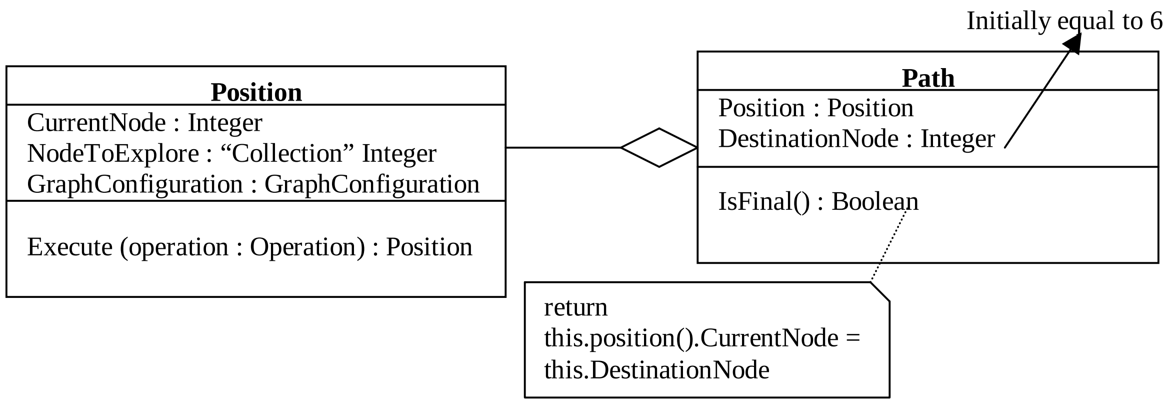

Initially equal to 6
Position
CurrentNode : Integer
NodeToExplore : "Collection" Integer
GraphConfiguration : GraphConfiguration
Execute (operation : Operation) : Position
Path
Position : Position
DestinationNode : Integer
IsFinal() : Boolean
return
this.position().CurrentNode =
this.DestinationNode

Figure 13

4 - Identify all the possible operations you can perform on your system.

In order to go through the graph, we need to choose the appropriate path from node to node. For this particular problem, the operations will be instances of an object called "NodeSelection". This object contains the node to be selected. The method " PossibleOperations() : "Collection" Operation" will return a collection of 6 instances of "NodeSelection". Each of these instances will contain a node number from 1 to 6.

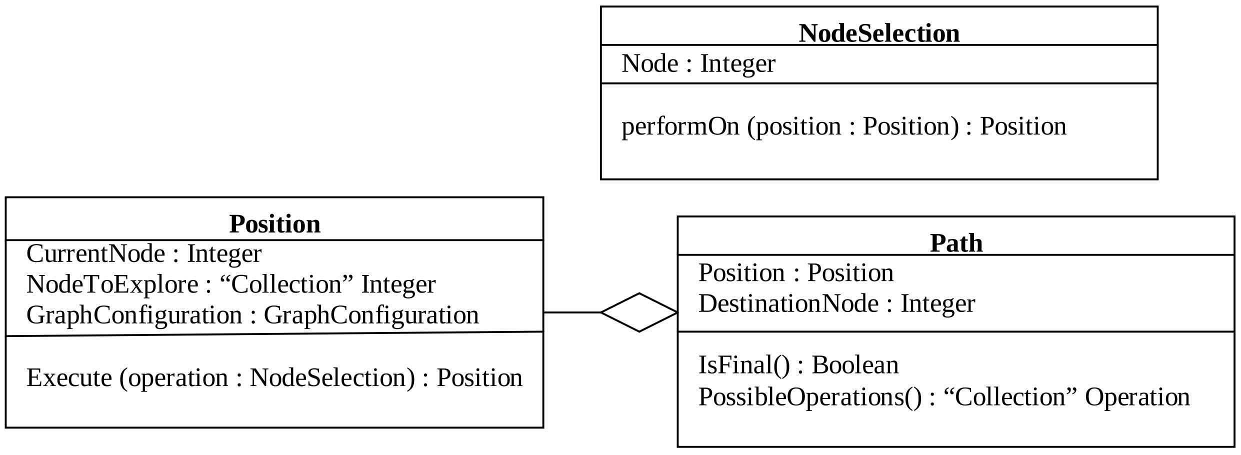

NodeSelection
Node : Integer
performOn (position : Position) : Position
Position
CurrentNode : Integer
NodeToExplore : "Collection" Integer
GraphConfiguration : GraphConfiguration
Execute (operation : NodeSelection) : Position
Path
Position : Position
DestinationNode : Integer
IsFinal() : Boolean
PossibleOperations() : "Collection" Operation

Figure 14

5 - Define the behavior of your operations on the state of the system.

In this case, "Position" will have to implement the method "SelectNode(aNode : Integer) : Position". This method returns a new instance of "Position" which contains the new selected node, as its current node and this new node is removed from its "NodeToExplore" list. Note that if the node has already been explored i.e. it is not in the "NodeToExplore" list, the execution of the operation returns NULL.

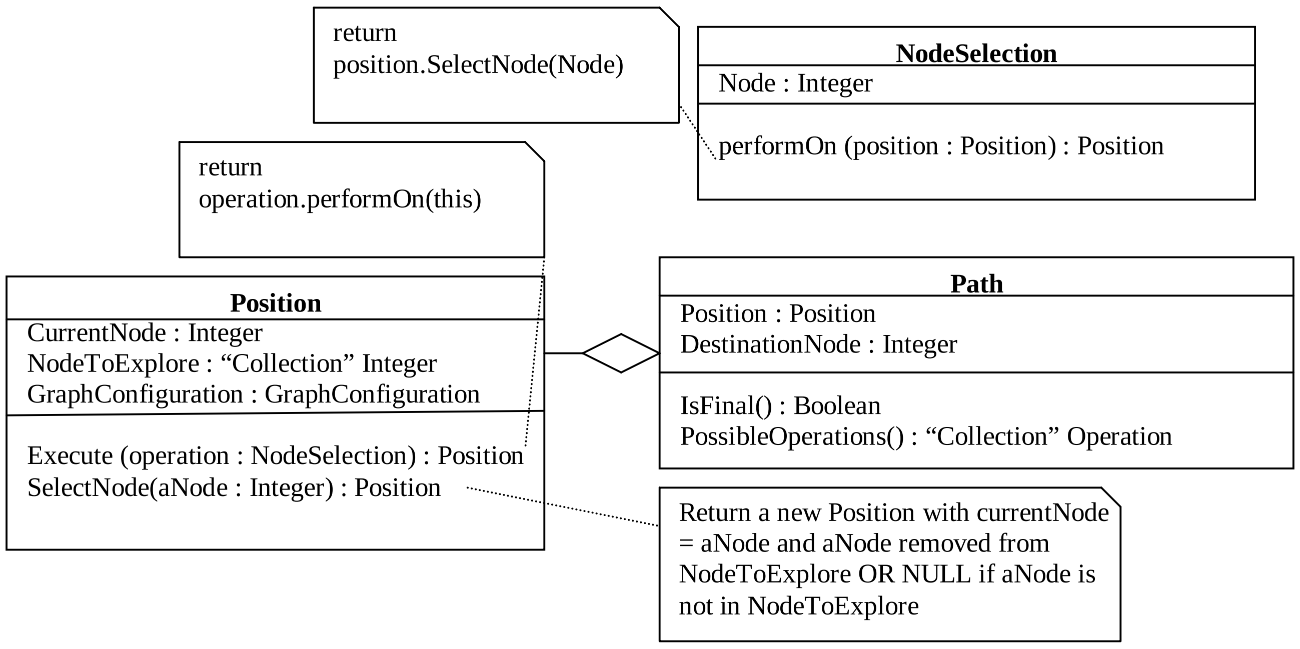

return
position.SelectNode(Node)
NodeSelection
Node : Integer
performOn (position : Position) : Position
return
operation.performOn(this)
Position
CurrentNode : Integer
NodeToExplore : “Collection” Integer
GraphConfiguration : GraphConfiguration
Execute (operation : NodeSelection) : Position
SelectNode(aNode : Integer) : Position
Path
Position : Position
DestinationNode : Integer
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
Return a new Position with currentNode = aNode and aNode removed from NodeToExplore OR NULL if aNode is not in NodeToExplore

Figure 15

6 - Identify which states are not valid in the context of the problem.

In this example, if a “Position” is returned after the execution of an operation, it is always valid. Consequently, “isValid() : Boolean” returns “true”.

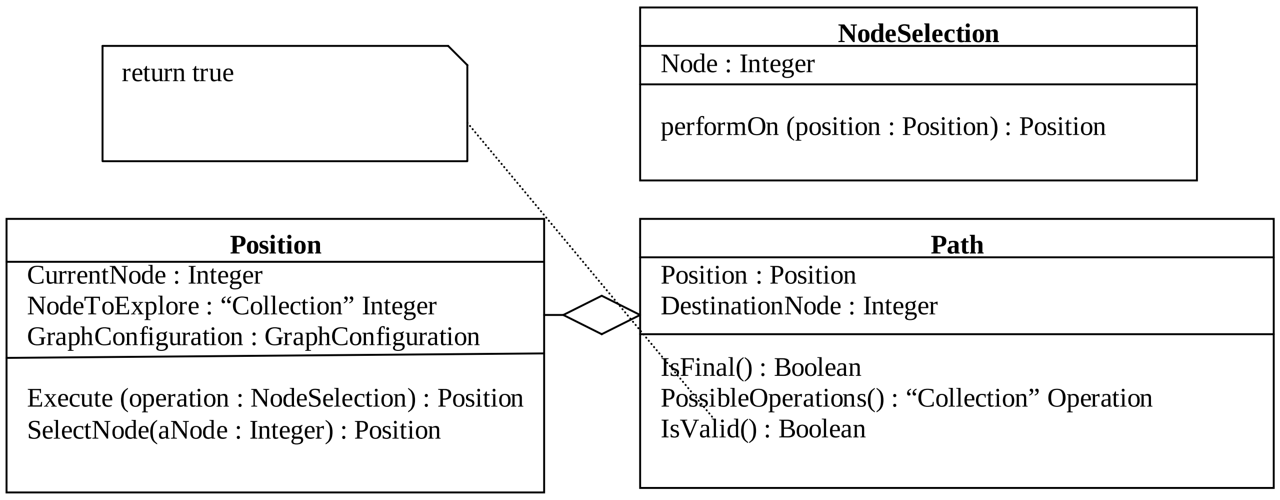

return true
NodeSelection
Node : Integer
performOn (position : Position) : Position
Position
CurrentNode : Integer
NodeToExplore : “Collection” Integer
GraphConfiguration : GraphConfiguration
Execute (operation : NodeSelection) : Position
SelectNode(aNode : Integer) : Position
Path
Position : Position
DestinationNode : Integer
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
IsValid() : Boolean

Figure 16

7 - Determine the cost of a state transition.

In the “Shortest path” problem, it is obvious that the cost of each operation is the cost associated with the transition between two nodes. In this example, the cost is expressed in the “GraphConfiguration” named earlier. This cost will be accumulated in an instance of the “Cost object, which is passed as a parameter when a node is selected.

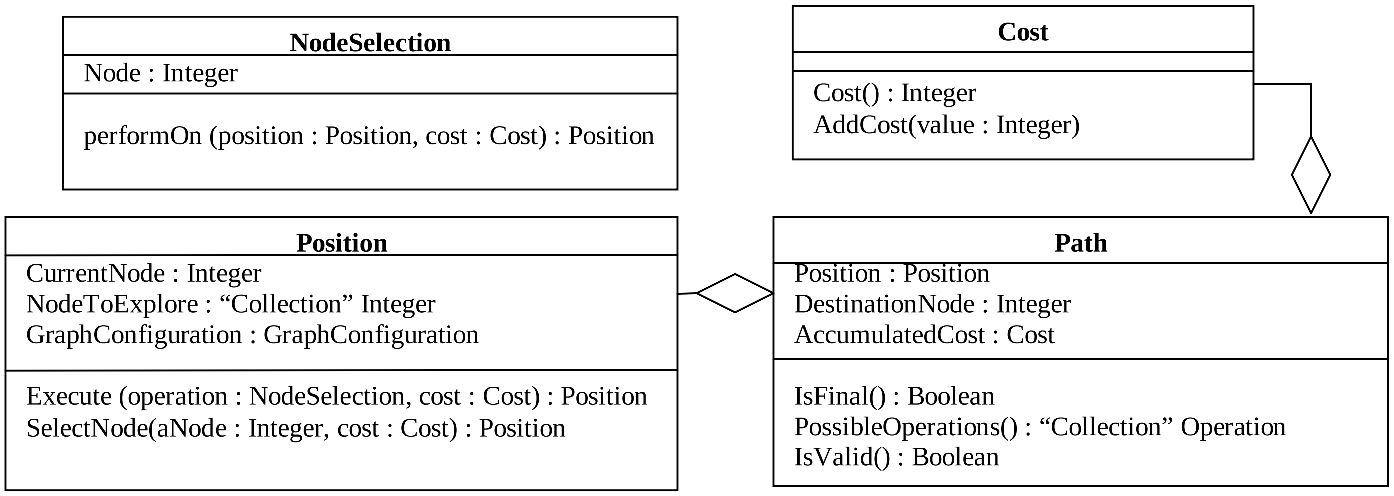

NodeSelection
Node : Integer
performOn (position : Position, cost : Cost) : Position
Cost
Cost() : Integer
AddCost(value : Integer)
Position
CurrentNode : Integer
NodeToExplore : “Collection” Integer
GraphConfiguration : GraphConfiguration
Execute (operation : NodeSelection, cost : Cost) : Position
SelectNode(aNode : Integer, cost : Cost) : Position
Path
Position : Position
DestinationNode : Integer
AccumulatedCost : Cost
IsFinal() : Boolean
PossibleOperations() : “Collection” Operation
IsValid() : Boolean

Figure 17

8 - Elaborate an exploration strategy.

At this point, the “Shortest path” problem is almost completed. The only thing we have to define is our state space exploration strategy. In other words, we have to decide how we will choose a candidate among the list of candidates for further exploration. For sure, the first candidate to be explored will be the initial “Path” object containing the initial “Position” object. For the next iterations, the candidate with the smallest accumulated cost will be chosen from among the candidates. Because our goal is to reach node number 6 using the shortest path, we will prefer the exploration of a path that appears to be the shortest one at the moment we have to continue the exploration of the state space.

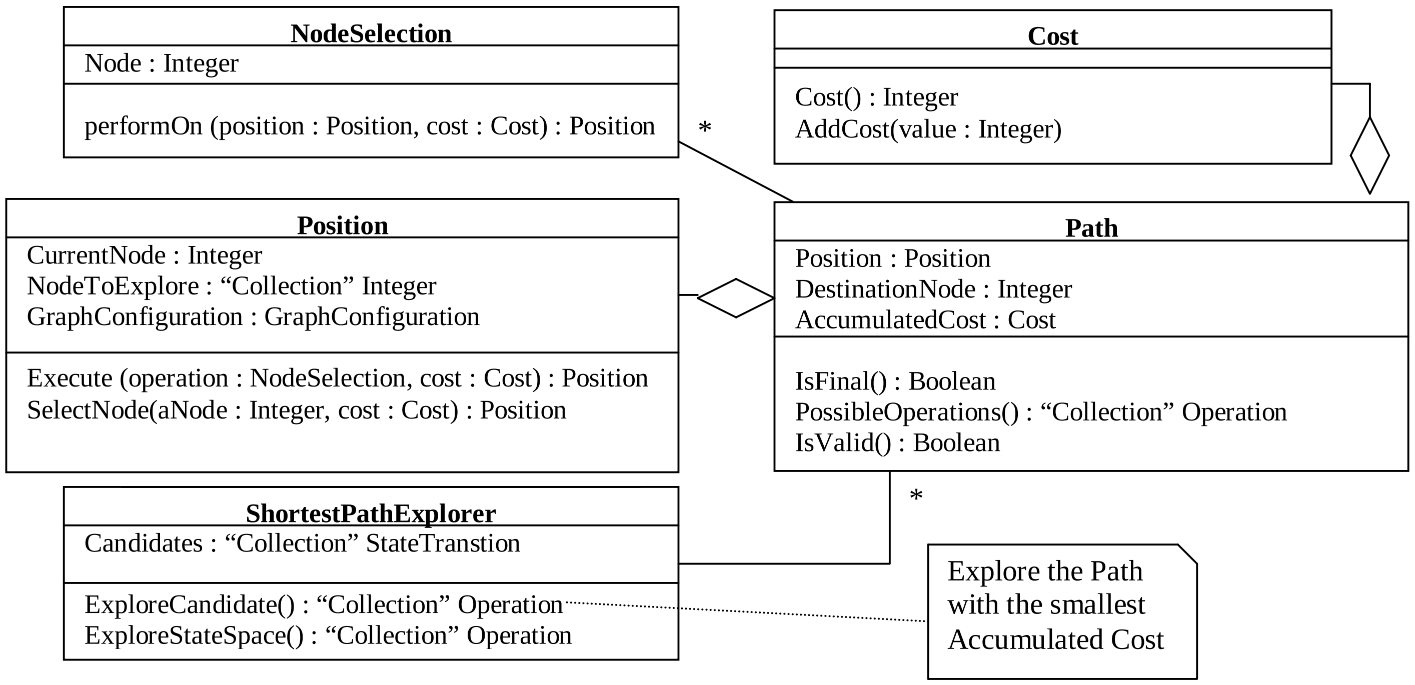

NodeSelection
Node : Integer
performOn (position : Position, cost : Cost) : Position
Cost
Cost() : Integer
AddCost(value : Integer)
Position
CurrentNode : Integer
NodeToExplore : "Collection" Integer
GraphConfiguration : GraphConfiguration
Execute (operation : NodeSelection, cost : Cost) : Position
SelectNode(aNode : Integer, cost : Cost) : Position
Path
Position : Position
DestinationNode : Integer
AccumulatedCost : Cost
IsFinal() : Boolean
PossibleOperations() : "Collection" Operation
IsValid() : Boolean
ShortestPathExplorer
Candidates : "Collection" StateTranstion
ExploreCandidate() : "Collection" Operation
ExploreStateSpace() : "Collection" Operation
*
*
Explore the Path with the smallest Accumulated Cost

Figure 18

Resulting Context

- **Recursive calls are not used.** The solution does not use recursion thus providing a limited usage of the memory and a clear and maintainable code.

- **Only one method has to be changed to modify the exploration strategy.** In fact, the exploration strategy resides in the "ExploreCandidate() : "Collection" Operation" method. The logic of an exploration strategy is to choose a candidate state among the "candidates" list based on the accumulated cost and the nature of the state itself. With the provided solution, "heuristics" can be tried and modified by affecting a single method.

- **No intermediate states are kept in memory. Only the operation history and the non-explored states are kept in memory.** This feature has the advantage to minimize the usage of memory. In fact, only the non-explored states are needed to operate a systematic exploration of the state space. Of course, an ordered history of the operations performed to reach the desired state need to be kept in order to provide a solution to the client object using the state explorer. Keeping the operations uses a lot less memory than keeping the intermediate states in order to provide a solution to the client object.

- **Reusable design concepts are provided to choose wisely a state exploration strategy.** The accumulated cost helps the system architect to elaborate an appropriate state exploration strategy. Algorithms such as depth-first and breath-first are easy to implement. Depth-first always choose the candidate with the highest accumulated cost and breath-first always choose the lowest accumulated cost. More sophisticated algorithms such as A-star [RK] can be implemented with this pattern. The accumulated cost and the nature of the states themselves constitute valuables elements to elaborate search heuristics.

- **Reuse is possible**

  - ✓ **State objects can be reused in different contexts.** The state object keeps its integrity by taking no responsibilities related to the state exploration. The state object is only responsible for applying some operations to itself in order to generate a subsequent state and to provide the cost of the applied operation. Reducing the responsibilities of the state object makes it more reusable and loosely coupled with the state exploration objects.

  - ✓ **Many specializations of the StateSpaceExplorer can be written to reuse successful heuristics.** Once a good heuristics has been successfully applied to many problems, it can be reused in many situations. Specializing and interchanging the StateSpaceExplorer objects can make the reuse of heuristics easy. Those objects encapsulate the state exploration strategy.

- **This pattern is not an optimization to solve problems with very large state space.** The time to explore and find a solution in a state space is directly proportional to the number of possible states that exists for a given problem. An appropriate choice of heuristic will always accelerate the time to find a solution, but in some circumstances, another technique may be considered when the state space is too large to have acceptable performances. Also, if the state space is considered too large for the desired performances, you might consider exploring only a part of the state space. As an example, it is currently impossible to explore in advance all the possible chessboard configuration during a chess game. However, it might be possible to explore the possible chessboard for the next two rounds.

- **If a solution exists in a finite state space, it is guaranteed to be found.** With the presented solution, it is possible to find the goal state in a finite state space if the exploration strategy considers the lowest accumulated cost. If the exploration strategy considers the highest accumulated cost, it might reach the goal state faster in some cases, but it might get stuck in cycles like it was mentioned in step 8 of the Solution section.

Related Patterns

- **Visitor.** Used to visit a state with a certain operation. This pattern allows the creation of any kind of operations without affecting the classes representing the state. The state object provides a public protocol that has to be complete but fairly stable in time. Any kind of operations can be built using one or a mix of methods provided by the state object.

- **Whole Part.** An instance of the Whole Part pattern is represented in this solution. The "StateTransition" object is, and must be, composed of a "State" object. Many behaviors provided by the "StateTransition" are delegated (completely or partially) to the "State" object (In this situation the "State" is a part of a "StateTransition").

- **State, Command and Null Object**. These patterns are not directly seen in the presented design, but they could be very useful to enhance the solution to many "real-life" problems.

Known Uses

The following algorithms [RK] can be implemented in the object-oriented world with the State Space Explorer pattern:

- **Depth-First**
- **Breath-First**
- **Hill Climbing**
- **A-Star (Best-First Search), without allowing to branch on an explored node.**

The previously stated algorithms implemented with the State Space Explorer pattern can help you to solve the following problems:

- **The shortest path problem as explained in this pattern**.

- **Building a schedule of different courses for a group of students without having a student with conflicting courses.** This problem can be seen as a state exploration problem. Placing a course at a given time can be an example of an operation. The course schedule can be seen as a state. The complete schedule without any conflicting courses for any students is the desired state.

- **Placing 8 queens on a chessboard without putting any queen in danger.** The goal of this problem is to place eight queens on an 8X8 chessboard without having two queens on the same horizontal, vertical or diagonal line. Placing a queen is an operation. A chessboard with two queens on the same line is invalid. The desired state is a valid chessboard with eight queens on it.

- **Travelling Salesman Problem**

- **The 8-Puzzle Problem.** As shown in figure 19, the goal of this puzzle is to place the eight squares in order (from 1 to 8) on a 3X3 board which have an empty square.

2	3	8
1	6	7
5	4	

Figure 19

References

[BW] B. Woolf, *Null Object*, In R. Martin, D. Riehle and F. Buschmann (eds.) Pattern Languages of Program Design 3, Addison-Wesley, Reading, MA. 5-18, 1998.

[GHJV] E.Gamma, R. Helm, R.Johnson, J. Vlissides. Design Patterns: *Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA. 1995.

[LS] G. Luger and W. Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Benjamin/Cummings Publishing Company, Inc., Chapter 3, 1993.

[RK] Elaine Rich and Kevin Knight. *Artificial Intelligence, Second Edition*. McGrawHill, 1991.