
Production Systems and Information Engineering
Volume 5 (2009), pp. 1–10

IMPLEMENTING THE STATE-SPACE REPRESENTATION BY THE SUPER OPERATOR DESIGN PATTERN

Gábor Kusper
Eszterházy Károly University, Hungary
Mathematics and Informatics Institute
gkusper@aries.ektf.hu

Csaba Sass
Eszterházy Károly University, Hungary
Mathematics and Informatics Institute
sass.csaba@uni-eszterhazy.hu

Szabolcs Márien
InnovITech Kft., Hungary
szabolcs.marien@innovitech.hu

[Received ... and accepted ...]

Abstract. While implementing state-space representation technique of artificial intelligence we found that the mathematical design suggests having a separate state and operator classes. But this breaks the encapsulation principle of OOP. To solve this we introduce a new structural design pattern, the super operator design pattern. A super operator is a proxy to operators, it has a parameter which decides which operator to call, it returns what the operator returns. In this way one class can represent the state-space representation which contains the operators and the states, so we can preserve encapsulation. On the other hand we break separation of concerns and single responsibility principles, since we do not separate states and operators. The set of methods behind the super operator should have a similar contract, because the super operator can ensure only the common part of the contracts of the methods. If this common part is empty, then this design pattern cannot be used. The new design pattern is applicable if iteration over a set of methods is important; a set of methods have similar contract, for the client is all them same which one is called; the understandability is more important than reusability, e.g., educational purposes; forcing encapsulation over separation of concerns is essential.

Keywords: design patterns, state-space representation, super operator design pattern

1. Introduction

Artificial Intelligence (AI) as a topic is neither new and nor does Object Oriented Programing (OOP). This article is about a new way of implementation of the state-space representation technique of AI by using OOP.

There are some earlier articles where they are entangled to use the benefits of both sides, like the book [11]. Here the mentioned paradigm is mostly used for better modularity, clear hierarchical structure and reusability.

An article closer to our interest is [5] which is a PhD dissertation by Márk Kósa. The focus in his paper is how to teach graph searching algorithms using object oriented methods in Java.

As instructors of Computer Science BSc we had the possibility to teach design patterns and artificial intelligence at the same time. In our case artificial intelligence course covers well-known graph searching algorithms like backtrack and depth-first search, where the graph is generated from a state-space representation. We learned the design patterns and design principles can help us to implement the state-space representation in such a way, that our students can easily understand it and create their own state-space representation based on our new design pattern, called Super Operator.

Stata-space representation [6] is a technique to represent a problem, it is given by the sequence $\langle A, s, G, O\rangle$, where A is the set of possible states, s is the starting state, G is the set of goal states, and O is the set of operators. An operator can construct a state out of a state, i.e., it is a function from A to A. An operator has also a pre and a post condition. The pre-condition is a predicate which decides whether the input of the operator is valid or not, the post-condition decides whether the output of the operator is a valid or not.

In case of the well-known 3-missionaries and 3-cannibals problem [7] (There are 3 missionaries and 3 cannibals on the left side of a river, there is also a 2-sit boat on the left side. The problem is that cannibals are hungry, and are going to eat missionaries if there are more cannibals than missionaries on any sides. We have to move all of them to the right side in safe!) one possible state-space representation is the following: $A = \{a|IsState(a)\}$. $IsState(\langle ml, cl, boat, mr, cr\rangle) :\Leftrightarrow ml \in \{0,1,2,3\} \wedge cl \in \{0,1,2,3\} \wedge boat \in \{left, right\} \wedge mr \in \{0,1,2,3\} \wedge cr \in \{0,1,2,3\} \wedge ml + mr = 3 \wedge cl + cr = 3$, where ml means missionaries on left, cl means cannibals on left, mr means missionaries on right, and cr means cannibals on right. $s = \langle 3, 3, left, 0, 0\rangle$. $G = \{\langle 0, 0, right, 3, 3\rangle\}$. $O = \{move01, move02, move10, move20, move11\}$.

From these 5 operators we give only one formally, the other ones can be constructed out of this one using the intuition that $moveXY$ means that we move X missionaries and Y cannibals by the boat to the other side.

$move01(\langle ml, cl, boat, mr, cr\rangle) = \langle ml, cl-1, right, mr, cr+1\rangle, if\ boat = left;$
$\langle ml, cl+1, left, mr, cr-1\rangle, otherwise.$

The operators of the state-space representation are called by a graph searching algorithm, because the graph is not ready when we start the algorithm, instead, some part of the graph is generated by the graph searching algorithm on thy fly. For example: backtrack generates a path, depth-first search generates a spanning tree of the graph. This means that we have to define an interface which is used by the graph-searching algorithms to call the operators. The two main possibilities are the following:

- We follow the mathematical description and we define separate classes like:
  - State, which represents a state,
  - Operator, which represents an operator,
  - StateSpaceRepr, which represents a state-space representation.
- We use the Super Operator Design Pattern and we define only one class:
  - State, which contains the operators and the states and represents state-space representation.

The first solution seems to be better since we know that separation of concerns [9, 10, 3] and single responsibility principle (for short: SRP) [8] are very strong OOP design principles [8, 2]. Separation of concerns states that if two or more concerns can be separated than it is a good idea to separate them, since the resulting source code will have better reusability. SRP state that each class should have only one responsibility, or in other words, each class should have only one kind of reason to change.

The problem is that in the case of the first solution we separate two concerns, state and operation, which belong very tightly together and hence we have to develop these two classes in parallel: the Operator has to know State and the State must be designed in such way that Operation should be easy to implement. Furthermore, if the State is changed then most probably we have to change the Operator and the other way around. This means that we have an implementation dependency, which is not easy to overcome.

These problems come from the fact that this solution breaks the encapsulation [1] OOP principles. It states that a class consists of a data-structure and methods (by other words: operators) on data structure. The data-structure is implemented by fields of the class. The values of the fields give the inner-state of instances of the class. The inner-state must not be modified from outside the class, only the methods of the class may modify the inner-state.

While we have only a few OOP principles [1], like abstraction, inheritance, encapsulation, and polymorphism, we have several OOP design principles [8, 2],

like separation of concerns, single responsibility principle (SRP), open-closed principle (OCP), Hollywood principle (HP), Liskov substitutional principle (LSP), interface segregation principle (ISP), law of Demeter, etc.

So, by the first solution we break a very basic principle, encapsulation, while we preserve some high level principles, separation of concerns, and SRP.

The second solution is introduced in this article, and has the name Super Operator Design Principle. In this solution we have only one class, the State class. The operators are methods of this class. The problem is that the number of operators and their functionality may vary wildly, although their contract is the same: they create a new state from an old one if their pre and post-condition is true. An idea would be to define a list of operators, but in case of this solution there is no Operator class. The other idea is to define a "super" operator which can call "basic operators". By "basic operators" we mean operators of the state-space representation.

In this case we need two methods:

- NumberOfOperators(), which returns the number of basic operators.
- SuperOperator(int i), which calls the i-th. basic operator and just returns what it returns.

In this way there is a well-defined interface for basic operators and we can implement the concerns of state and operation in the same class. So we preserve the encapsulation principle, but break the separation of concerns and the SRP. In the following sections we discuss more deeply these two solutions and their main variants.

2. Implementation of State-Space Representation Following the Definition

In this chapter we give the implementation of state-space representation which follows its definition. We use 4 classes:

- State, which represent a state,
- Operator, which represent an operator,
- StateSpaceRepr, which represent a state-space representation, and
- Node, which represent a vertex of the search graph generated out of the state-space representation.

We give here the source code of the first 3 classes (the whole can be found here: `http://pastebin.com/QvYzDG4R`).

```
abstract class State {
    public abstract bool IsState();
```

```
    public abstract bool IsGoalState();
}
abstract class Operator {
    public State Operate(State a){
        if (!PreCondition(a)) return null;
        State b = operate(a);
        if (!PostCondition(b)) return null;
        return b;
    }
    public abstract bool PreCondition(State a);
    protected abstract State operate(State a);
    public abstract bool PostCondition(State a);
    public bool IsApplicableOn(State a) {
        if (!PreCondition(a)) return false;
        State b = operate(a);
        if (!PostCondition(b)) return false;
        return true;
    }
}
class StateSpaceRepr {
    State startState;
    List<Operator> operators;
    public StateSpaceRepr(State s, List<Operator> op) {
        this.startState = s; this.operators = op;
    }
    public bool IsState(State state) { return
       state.IsState(); }
    public State GetStartState() { return startState; }
    public bool IsGoalState(State state) { return
       state.IsGoalState(); }
    public List<Operator> GetOperators() { return operators; }
}
```

We can see that class State and Operator are abstract ones. If we have a problem, like the 3-missionaries and 3-cannibals problem, then we can create child classes by implementing the abstract methods. The State class contains two abstract methods:

- bool IsState(), which decides whether the instance is a state or not, and

- bool IsGoalState(), which decides whether the instance is a goal state or not.

One can argue that these two predicate could be implemented also in the StateSpaceRepr. In this way the State class will be empty and the IsState(State a) and IsGoalState(State a) will be abstract in the StateSpaceRepr. This solution is possible but we think that the presented solution fits better OOP principles, since IsState() and IsGoalState() are methods on a state, so they should be placed in the State class.

The State class is meant to be immutable. This means that if we run an operator on it, then it will be not changed, so we do not have to make a copy of before an operator call, so we do not implement the Cloneable interface. Our implementation of 3-missionaries and 3-cannibals problem (`http://pastebin.com/QvYzDG4R`) follows this convention.

The Operator class contains 3 abstract methods:

- bool PreCondiction(State a), which is the pre-condition of the operator,
- bool PostCondition(State a), which is the post-condition of the operator, and
- State operate(State a), which is the operator itself.

It is interesting that we have an Operate and an operate method at the same time. The Operate method is public but not polymorphic. It follows the template method design pattern and fix the order of PreCondition, operate, and PostCondition. It is also an example for inversion of control, since it calls and abstract method, operate, which will be implemented by the child class, so not the child calls the parent, but the parent calls the child. The operate method is abstract, i.e., polymorphic, and should be overridden by the child to implement the operator, as you can see here: `http://pastebin.com/QvYzDG4R`.

The StateSpaceRepr class contains the following methods:

- IsState(State state), it calls "state.IsState();",
- GetStartState(), gives back the initial (by other word: starting) state,
- IsGoalState(State state), it calls "state.IsGoalState();",
- GetOperators(), returns the list of operators.

This class just follows the mathematical description of state-space representation, which is $\langle A, s, G, O \rangle$, A is the set of states, s is the starting state, G is the set of goal states, and O is the set of operators. The only difference, that instead of set of states we have a predicate, the IsState() method, which can decide whether a State instance is really a state or not. The same applies to set of goal states.

The only problem with this implementation is that it reveals a part of its inner state, since it gives back the reference to operators by GetOperators() method. The rest of the code can change the list of operators through this reference, which can result in a hard-to-detect bug. So this implementation assumes that the rest of the code is trustworthy.

Now we give the implementation of the child classes of State and Operator in case of 3-missionaries and 3-cannibals problem:

```
class MCState : State { // Missionaries And Cannibals State
    int ml, cl, boat, mr, cr;
    public int MissionariesOnLeft { get { return ml; } }
    public int CannibalsOnLeft { get { return cl; } }
    public int Boat { get { return boat; } }
    public int MissionariesOnRight { get { return mr; } }
    public int CannibalsOnRight { get { return cr; } }
    public MCState(int ml, int cl, int boat, int mr, int cr) {
        this.ml = ml; this.cl = cl; this.boat = boat;
        this.mr = mr; this.cr = cr;
    }
    // this creates the start state
    public MCState() {
        ml = 3; cl = 3; boat = 1; // 1: left, -1: right
        mr = 0; cr = 0;
    }
    public override bool IsState() {
        return ml + mr == 3 && cl + cr == 3 &&
               ml >= 0 && ml <= 3 && cl >= 0 && cl <= 3 &&
               mr >= 0 && ml <= 3 && cr >= 0 && cr <= 3 &&
               (boat == 1 || boat == -1) &&
               (ml >= cl || ml == 0) && (mr >= cr || mr == 0);
    }
    public override bool IsGoalState() {
        return mr == 3 && cr == 3 && boat == -1;
    }
    public override bool Equals(object obj) {
        MCState other = (MCState) obj;
        return this.ml == other.ml && this.cl == other.cl &&
               this.boat == other.boat &&
               this.mr == other.mr && this.cr == other.cr;
```

```
    }
}
class MoveMC : Operator {
    int mToMove, cToMove;
    public MoveMC(int mToMove, int cToMove) {
        this.mToMove = mToMove;
        this.cToMove = cToMove;
    }
    public override bool PreCondition(State a) {
        MCState state = (MCState)a;
        if (state.Boat == 1) {
            return state.MissionariesOnLeft >= mToMove &&
                   state.CannibalsOnLeft >= cToMove;
        }
        else {
            return state.MissionariesOnRight >= mToMove &&
                   state.CannibalsOnRight >= cToMove;
        }
    }
    protected override State operate(State a) {
        MCState state = (MCState)a;
        int ml, cl, boat, mr, cr;
        ml = state.MissionariesOnLeft;
        cl = state.CannibalsOnLeft;
        boat = state.Boat;
        mr = state.MissionariesOnRight;
        cr = state.CannibalsOnRight;
        if (boat == 1) {
            ml -= mToMove; cl -= cToMove; boat = -1;
            mr += mToMove; cr += cToMove;
        }
        else {
            ml += mToMove; cl += cToMove; boat = +1;
            mr -= mToMove; cr -= cToMove;
        }
        return new MCState(ml, cl, boat, mr, cr);
    }
    public override bool PostCondition(State a) {
```

```
        return a.IsState();
    }
    public override string ToString() {
        return mToMove + ", " + cToMove;
    }
}
```

The problem with this implementation is that the MCState class has to reveal its inner-state by properties (we have a getter property for each attribute in MCState class), otherwise we could not write the MoveMC operator.

One can argue that in OOP it is quite natural that we have a getter for each attribute. This is true. The real problem is that MCState and MoveMC classes are tightly coupled, i.e., there is a strong implementation dependency between them. If we change the representation of a state, for example we use boolean attribute instead of integer to store where is the boat, then we have to change the operator, and if we change the operator, for example to have less basic operator, then most probably we have to change the state.

This strong implementation dependency comes from the fact that the MCState represents the data-structure, and MoveMC represents the methods on that data-structure.

This means that we break the encapsulation OOP principle, which states that in OOP the data-structure and its method is one entity which is called class. On the other hand we have high-level OOP design principles like separation of concerns and SRP. Separation of concerns states that if two or more concerns can be separated than it is a good idea to separate them, since the resulting source code will have better reusability. SRP state that each class should have only one responsibility, or in other words, a class should have only one reason to change.

We can see that state and operator can be separated, and hence, by separation of concerns, it is good to separate them. SRP is questionable here. MCState has only one responsibility, it represent a 3-missionaries and 3-cannibals state. MoveMC has also only one responsibility, it represent an operator of 3-missionaries and 3-cannibals problem. The question is that they change on the same reason or not? If yes, then we should not separate them, if not, then we should separate them because of SRP.

We present a reason which applies only to the operator but not for the state, but we admit, that almost any other reason forces the changes both the state and the operator. The reason which applies only to the operator is that we would like to speed it up, for example by getting rid of the " if (boat == 1)" statement. The new operator looks like this:

```
protected override State operate(State a) {
    MCState state = (MCState)a;
    int ml, cl, boat, mr, cr;
    ml = state.MissionariesOnLeft;
    cl = state.CannibalsOnLeft;
    boat = state.Boat;
    mr = state.MissionariesOnRight;
    cr = state.CannibalsOnRight;
    ml -= mToMove * boat;
    cl -= cToMove * boat;
    mr += mToMove * boat;
    cr += cToMove * boat;
    boat = boat * -1;
    return new MCState(ml, cl, boat, mr, cr);
}
```

This means that the condition of SRP holds, so we should also do separation because of it. But there is still one more question. Do we really get better reusability? Can we reuse MoveMC without MCState? In fact not, we cannot reuse it, because of this line:

```
MCState state = (MCState)a;
```

The first step of the operator is to convert the input State into MCState, so to use MoveMC we have to use MCState.

On the other way we could reuse MCState without MoveMC, but it is not easy to find any scenario. A possible scenario is if we create another problem, which uses the same representation. For example, there is a 3-sit boat instead of the 2-sit one.

It seems that it does not pay off to break encapsulation in order to preserve separation of concepts and SRP.

3. Implementation of State-Space Representation using the Super Operator Design Pattern

If we use the super operator design pattern, then we can encapsulate the state and the operator in one class, although we do not know in advance how many basic operators there will be. The solution is the SuperOperator(int i) method, which can call the i.-th basic operator and returns what the basic operator returns.

In this way we need only one class, namely State, instead of State, Operator, and StateSpaceRepr from the previous section. Of course we still need the Node class which will represent the search graph generated out of the state-space representation, but now any information comes from the State class.

Here we give the source code of State (the whole implementation of state-space representation using super operator design pattern is here: `http://pastebin.com/6DNpQjVr`):

```
abstract class State : ICloneable {
    public abstract bool IsState();
    public abstract bool IsGoalState();
    public abstract int GetNumberOfOperators();
    public abstract bool SuperOperator(int i);
    public abstract object Clone();
}
```

One can see that this State class has 3 more methods compared to the State class from the previous section. The new methods are:

- GetNumberOfOperators(), which returns the number of basic operators,
- SuperOperator(int i), which calls the i.-th basic operator and returns what it returns,
- Clone(), which clones the state and which uses deep copy.

The first two methods are used to move the "interface for operators" from StateSpaceRepr to the State class. This means that these two methods are used by rest of the code to call the operators. The method call

- "state.SuperOperator(0);" calls the first basic operator,
- "state.SuperOperator(1);" calls the second basic operator,
- ...
- "state.SuperOperator(state.GetNumberOfOperators()-1);" calls the last one.

The idea is that at the time of the development of the graph searching algorithms, like backtrack, we do not know the concrete problem, we do not know the operators to call, we only know that we have to call (in the worst case) all of them. So we need an interface that is known at the time of developing backtrack and it can be used to iterate over the operators. This interface is the above two methods from class State:

- GetNumberOfOperators()
- SuperOperator(int i)

The third method, Clone(), is used because we prefer mutable state representation, where the operators can work on the attributes of the state. We know from backtrack that sometimes we have to go back to an earlier state, so before we apply an operator to a state, we have to clone t. This is done in the Node class, which is not presented in this article, but can be found here: `http://pastebin.com/6DNpQjVr`.

If one prefers to have immutable state representation then we need a kind of copy constructor instead of the Clone() method. We have implemented this variant, which can be found here: `http://pastebin.com/gZ9rF4Qz`.

The immutable variant moves the task "copy the attributes" into the operator, which seems to be less natural to our students, so we prefer the mutable variant presented in this article, although it is more dangerous, because sometimes our student uses shallow copy instead of deep copy in the Clone() method. Usually they learn in the course of debugging what is the difference Of course we should also use deep copy if we choice the immutable variant but in this case deep copy take place inside the operator.

Here we give the source code of MCState (the whole implementation is here: `http://pastebin.com/6DNpQjVr`):

```
class MCState : State {
    int ml, cl, boat, mr, cr;
    public MCState() {
        ml = 3; cl = 3; boat = 1; // 1: left, -1: right
        mr = 0; cr = 0;
    }
    public override object Clone() { return
       MemberwiseClone(); }
    public override bool IsState() {
        return ((ml >= cl) || (ml == 0)) &&
               ((mr >= cr) || (mr == 0));
    }
    public override bool IsGoalState() {
        return mr == 3 && cr == 3 && boat == -1;
    }
    public override int GetNumberOfOperators() { return 5; }
    public override bool SuperOperator(int i) {
        switch (i) {
            case 0: return operate(0, 1);
            case 1: return operate(0, 2);
```

```
            case 2: return operate(1, 1);
            case 3: return operate(1, 0);
            case 4: return operate(2, 0);
            default: return false;
        }
    }
    private bool operate(int mToMove, int cToMove) {
        if (!preCondiction(mToMove, cToMove)) return false;
        if (boat == 1) {
            ml -= mToMove; cl -= cToMove; boat = -1;
            mr += mToMove; cr += cToMove;
        }
        else {
            ml += mToMove; cl += cToMove; boat = 1;
            mr -= mToMove; cr -= cToMove;
        }
        if (IsState()) return true;
        return false;
    }
    private bool preCondiction(int mToMove, int cToMove) {
        if (mToMove + cToMove > 2 || mToMove + cToMove < 0 ||
            mToMove < 0 || cToMove < 0) return false;
        if (boat == 1)
            return ml >= mToMove && cl >= cToMove;
        else
            return mr >= mToMove && cr >= cToMove;
    }
    public override string ToString() {
        return ml + "," + cl + "," + boat + "," + mr + "," +
           cr;
    }
    public override bool Equals(Object a) {
        MCState aa = (MCState)a;
        return aa.ml == ml && aa.cl == cl && aa.boat == boat;
    }
}
```

Please note, that the SuperOperator(int i) contains a "big-fat" switch statement which calls the basic operators depending on the value of parameter i.

Usually it is enough to write only a few operator with some parameter. In the above example there is only one operator with two parameters: operate(int mToMove, int cToMove). Although we have only one operator, there are 5 basic operators, because different parameters give different basic operators of the state-space representation. The 5 basic operators are:

- operate(0, 1), which moves no missionaries and 1 cannibal to the other side of the river,
- operate(0, 2), which moves no missionaries and 2 cannibals to the other side of the river,
- operate(1, 1), which moves 1 missionary and 1 cannibal to the other side of the river,
- operate(1, 0), which moves 1 missionary and no cannibals to the other side of the river,
- operate(2, 0), which moves 2 missionaries and no cannibals to the other side of the river.

The switch in the SuperOperator(int i) method calls them depending on the value of i. The order of the basic operator is all the same, but it might be that some order results in better running time.

Please note also, that GetNumberOfOperators() return 5, so there must be 5 cases in the switch of SuperOperator(int i) from 0 to 4, i.e., we can see the SuperOperator(int i) method as a mapping between the range $0 \ldots GetNumberOfOperators() - 1$ and the basic operators. We recall the methods GetNumberOfOperators() and SuperOperator(int i) in order to make it easier to the reader to check the above statements:

```
public override int GetNumberOfOperators() { return 5; }
public override bool SuperOperator(int i) {
    switch (i) {
        case 0: return operate(0, 1);
        case 1: return operate(0, 2);
        case 2: return operate(1, 1);
        case 3: return operate(1, 0);
        case 4: return operate(2, 0);
        default: return false;
    }
}
```

These two methods together are the heart of the Super Operator Design Pattern, which is defined more formally in the next section. It is important that the basic operators have the same goal: they create a new state out of the old

one. In other words they contract is very similar, although not the same. For example operate(0, 1) moves 1 cannibal, and operate(0, 2) moves 2 ones, but both of them result in a new state if their pre- and post-condition holds, in this case they return true. This is their common contract.

The super operator is suitable only if the client requires the common contract. If the client would like to move 2 cannibals, then the super operator design pattern is not good, because the client do not know which super operator parameter value maps to the operator which moves 2 cannibals.

The graph searching algorithms are not interested how the basic operators work, for them it is enough that they create a new state if they are applicable. So the super operator design pattern is suitable for them.

The graph searching algorithms must call all the basic operators in the worst case, but this is also possible by the super operator design pattern. It is important to note that the operate(int mToMove, int cToMove) method is private, so it is hidden from the rest of the code. It is hidden by the super operator. Since super operator hides other methods, we can also call it "proxy for methods", or for short "method proxy".

The super operator design pattern preserves the encapsulation OOP principle, since it keeps together the data-structure (the attributes of State) and its methods (the basic operators) in one class, which is the State class; but it breaks separation of concerns and SRP, since the State class fulfills the responsibilities of the states, operators, and the state-space representation.

Note that, that the implementation of Clone() is very easy in this example. We have to call only a C# specific method, called MemberwiseClone(), which creates a shallow copy of the instance. We know that Clone() has to create a deep copy, but since each attribute in MCState is a value-type one, there is no difference between shallow and deep copy. After this case study we give the Super Operator Design Pattern more formally in the next section.

4. Super Operator Design Pattern

In this section we define the Super Operator Design Pattern by the structure used in the GOF book [GOF1995], which is widely accepted by programmers.

First of all we have to discuss whether super operator is a creational, a structural or a behavioral design pattern.

Creational design patterns are used instead of the new keyword. The above case study, especial the immutable variant, creates new State instances, but this is not the intent of the pattern but this is the common contract of the methods behind the super operator. So this is not a creational pattern.

Behavioral patterns focus how objects communicate. This pattern gives conditions only for one class, so this is not a behavioral pattern.

Structural patterns focus on relationships of entities. This pattern describes how a set of methods and a special method, called super operator, are related: the super operator is a mapping between its parameter and between these set of methods, its parameter decides which method to call, it returns what this method returns. So this is a structural design pattern.

Intent. Give a common interface, the super operator, for a set of basic methods which have similar contract, the super operator hides them and ensures the least common contract of them, the parameter of super operator decides which one is called, so the client can iterate over the methods.

Also Known As. Proxy for methods, method proxy

Motivation. While implementing state-space representation technique of artificial intelligence we found that the mathematical design suggests having a separate state and operator classes. But this breaks the encapsulation principle of OOP. To solve this we might introduce a so called super operator which serves as a proxy to operators. It has a parameter which decides which operator to call, it returns what the operator returns.

In this way state class can contain the operators and the super operator and so we can preserve encapsulation. This means that to create a new state-space representation we need only implement a child class of state. In this way the new state-space representation will be concise and easy to understand. On the other hand we break separation of concerns and SRP, since we do not separate states and operators. Super operator design pattern favors understandability over reusability.

The super operator hides the methods (the operators), and with the help of "get number of operators" makes it possible for the client to iterate over them. In case of state-space representation, the backtrack algorithm can iterate over operators. The set of methods behind the super operator should have a similar contract, because the super operator can ensure only the common part of the contracts of the methods. If this common part is empty, for example, if one of the methods gives back integer and another one gives back string, then this design pattern cannot be used.

Applicability. Super operator design pattern is used when:

- Iteration over a set of methods is important.
- A set of methods have similar contract, for the client is all them same which one is called.
- The understandability is more important than reusability, e.g., educational purposes.

- Forcing encapsulation over separation of concerns is essential.

Structure. There are at least two classes:

- The server class which contains the set of methods hidden by super operator.
- The client class which calls the super operator.

We assume that the client want to call (in the worst case) all of the hidden methods, it wants to call the best one first, or it is irrelevant in which order the methods are called. In this case the server has to have at least two methods:

- SuperOperator(int i, Object p1, Object p2, ..., Object pn)
- GetNumberOfMethods()

The SuperOperator calls the i.-th method with parameters p1, p2, ..., pn. It might call the i.-th method with fewer or ever more parameters. It returns what the i.-th methods gives back. The client calls the SuperOperator from a loop like this one:

```
for(int i = 0; i < server.GetNumberOfMethods(); i++)
{
    Object result = server.SuperOperator(i, p1, p2, ..., pn);
    ...
}
```

The Server class must be either immutable, or must be cloned before call of SuperOperator, or the last call of SuperOperator must be withdrew before the next call of SuperOperator. For the first two cases we have a sample implementation:

- `http://pastebin.com/gZ9rF4Qz`, where the server class is immutable,
- `http://pastebin.com/6DNpQjVr`, where the server class is cloneable.

The server has to maintain a heuristic if the client wants to call the best method at first, the second best afterwards. In this case the heuristic must decide which method is the best.

Consequences. If we use the super operator design pattern then we do not have to implement an Operator class, which results in less modular but more understandable solution.

Related Patterns. There are two structural patterns similar to super operator:

- Proxy, which hides a critical object in a transparent way, and
- Facade, which hides a complex module by a simple interface.

Proxy is similar to super operator, both of them hide some parts of the source code from the rest of the code, but proxy hides a critical object, most probably only one critical object, super operator hides a set of similar methods, most probably not only one method.

Facade is similar to super operator, both of them define a simple interface to hide some part of the source code from the rest of the code, but while faÃğade hides several objects which work together as a complex module, super operator hides only a set of similar methods.

5. Future work

We would like to study open source projects, like the ones hosted by Apache Software Foundation, whether the super operator design pattern is used by some of them or not. If yes, we should understand those variants and integrate them into the description of this pattern. If not, then we must consult software architectures and analyze their opinions about this new design pattern.

REFERENCES

[1] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston: *Object-Oriented Analysis and Design with Applications.* 3rd Edition, book, 2007.

[2] John Dooley: *Object-Oriented Design Principles, in Software Development and Professional Practice, book chapter.* Apress, pages 115-136, 2011.

[3] Erik Ernst: *Separation of concerns.* Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, USA, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, ISBN 0-201-63361-2, 1995.

[5] Márk Kósa: *Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába.* PhD dissertation, University of Debrecen, 2009.

[6] Gergely Kovásznai, Gábor Kusper: *Artificial Intelligence and its Teaching.* Lecture notes, Eszterházy Károly College, 2012.

[7] Vladimir Lifschitz: *Missionaries and cannibals in the causal calculator.* Proceedings the KR2000 conference, pages 85-96., 2000.

[8] Robert C. Martin: *The single responsibility principle*, in The Principles, Patterns, and Practices of Agile Software Development, pages 149-154, 2002.

[9] D.L. Parnas: *On the Criteria To Be Used in Decomposing Systems Into Modules.* Communications of the ACM, 15, 12, 1053-1058, 1972.

[10] D.L. Parnas: *Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs*. Lecture Notes in Computer Science, Programming Methodology, 1974.

[11] Bernard P. Zeigler: *Object-Oriented Simulation with Hierarchical*, Modular Models: Intelligent Agents and Endomorphic System, ISBN: 978-0-12-778452-6, 2014.