Canon of Software Engineering (Patterns, Principles, and Practices) and the Super Operator Design Pattern

Canon is a set of books, movies, music and art, which holds the essence of our culture, it contains the common sense of people. The canon of software engineering is a set of artworks (books, papers, blogs, projects) which holds the essence of software engineering. The canon is not fixed, it changes time to time because scholars are arguing constantly what belongs to the "Top 100 of Best Software Engineering Books", what are the "Essential Books", the "Must-Read Books". The canon is important because it answers the question "What books should I read to become a better software engineer?". The canon relies on a school or on a tradition. For example Carnegie Mellon University issues the "Seminal Papers in Software Engineering: The Carnegie Mellon Canonical Collection" which is a list of papers which must be studied by their software engineering students. Another example is "SWEBOK, Guide to the Software Engineering Body of Knowledge" by IEEE Computer Society which is a very broad collection of software engineering terminologies. A more concise canon is the "SOLID design principles" which consists of the 5 most prominent object-oriented design principles. Other example is the GOF book which contains 23 design patterns. Most of us we use only best practices from XP, like pair programming, code review, continues integration. Uncle Bob (Robert C. Martin) has his book with the title: Agile Software Development, Principles, Patterns, and Practices. We think, this book belongs to the Canon of Software Engineering.

After we understand the importance of design patterns, we introduce the super operator design pattern, which helps us to implement the state-space representation technique of artificial intelligence. The state-space representation consists of the set of states, the initial state, the set of goal states, and the set of operators. An operator is a function, which creates a state out of a state. From this representation, one can build up a graph, where vertices are states and edges correspond to operation application. The graph searching algorithms, like backtrack, depth-first search, have to call systematically all operators, but we have to write these algorithms at the time when the state representation, and so the operators, are not known. So, we have to give a common interface for all possible state-space representation which will be used by the graph searching algorithms. To solve this problem, we have several possibilities, the first one is that we design a separate Operator and a separate State class. This solution fits best the mathematical description of state-space representation, but breaks the encapsulation OOP principle, which states that in OOP the data-structure and its method are one entity, which is called class. The second solution is described by the super operator design principle. Here the State class also contains a SuperOperator(int i) method, which calls i.-th operator of the state-space representation. This solution does not break encapsulation principle, but does break the principle called “separation of concerns”, which state that if we are able to separate two concerns, like state and operations on the states, then we should separate them. We can see that in case of implementing the state-space representation we cannot ensure both encapsulation and separation of concerns. The first solution gives better reusability; while the second one, the super operator design pattern, gives better understandability. We present both solutions in C#. We also highlight the most prominent variants.