Applying Design Patterns to State-Space Search Applications

Philip W. L. Fong Edward Kim Qiang Yang

August 15, 1996

1 Introduction

This paper summarizes our experience in applying design patterns to develop intelligent systems based on heuristic search. Over the past few years, we have built a number of artificial intelligence (AI) planners, schedulers, and other kinds of problem solvers in languages including Common Lisp, C, and C++. As experience cumulates, we intended to design an object-oriented framework for building intelligent problem solvers. Our hope is that the framework allows various reusable AI techniques to be mixed and matched in a highly flexible manner. The effort resulted in a C++ framework called Plan++. Our recent implementation of AI planners, as well as our current foray into case-based planning systems, are all done on top of this framework.

In this paper, we report how the design pattern catalog [2] has shaped the way we design our framework. Our interest here is twofold. First, we want to demonstrate how design patterns can be employed to structure a search-based applications, so that the unique complexity of constructing AI software can be managed. Second, we want to discuss the role played by a design pattern catalog during our process of designing and evolving the framework. Prior to this project, our experience in object-oriented design was only limited to university courses in programming languages and courses in object-oriented specification and design (where the notion of design patterns were introduced). We want to report how the pattern catalog remedied our lack of experience.

We will first introduce what state-space search is (section 2), and discuss the resuse challenges that our framework is trying to meet (section 3). We then discuss how several design patterns were applied to shape the design of our framework. We report the forces involved in each decision, the solution we adopted, the alternatives we considered, and in what manner the design pattern catalog helped us in coming up with the solution. We summarize our experience in section 7.

2 State-Space Search

A significant number of AI and combinatorial optimization problems are instances of heuristic *state-space search* [7]. A state space is a set of *states* plus a set of *operators*. An operator is a transformation that generates one or more states given the input of a state. A state space, therefore, implicitly defines a graph in which nodes are states, and edges are operator applications. A state space search is basically a graph searching problem such that nodes in the underlying graphs are generated incrementally as the search proceeds. The *goal* of a search is a set of states with some interesting properties. A search is *successful* if it finds one of the goal nodes. A typical search routine looks like the following:

Search(start-state, operators, goal):
  mark start-state as *"generated"*;
  while (not all *generated* nodes are *expanded*) **do**
    select a node n that is *generated* but not *expanded*;
    generate all the neighboring nodes n' of n;
    mark all n' as *"generated"*;
    mark n as *"expanded"*;
    if (n satisfies the goal) **then return** n;
  end while;
  return *"failure"*;
end Search.

Intuitively, a node is "generated" the first time it is visited. A node is "expanded" if all its neighbouring nodes are "generated". The efficiency of a search is largely determined by the order in which the state space is traversed, which, in turn, is determined by the order in which nodes are selected for expansion, the definition of the operator, and the representation of the states.

Throughout the paper, we will use a sample AI planning domain to illustrate the effect of applying the design patterns. An instance of the *waterjug* planning problem is often described as follows:

> *"You have two jugs $\{A, B\}$, jug A holds m litres (L) of water, jug B holds n L of water. You are allowed to do three things with the jugs: fill up a jug, empty a jug, or pour the content of a jug to another (until one of them is either full or empty). Find a sequence of steps which will result in a jug holding kL of water."*

For example, in a sequel of the movie *Die Hard*, the protagonists are given two jugs of capacity 3L and 5L, and they are asked to measure out exactly 1L from the two jugs. Here a *state* can be *represented* by a vector (p, q), where p and q are the amount of water in jug A and B respectively. The *operators* are fillJug(), emptyJug(), and pourContentsOfJugIntoOther-Jug(); thus, the neighbourhood of a node in the state-space is *generated* by the execution of all possible operators from this state; then the *goal* states are all (p, q) so that one of p or q is 1.

3 Reusability in Search-Based Applications

Reuse is a unique challenge in search-based applications because of several reasons:

1. *Frequent shift of representation.* The efficiency and quality of search depends largely on the representation of state space. Designers building search application must experiment with multiple representation before an appropriate one is found. A reusable search engine should anticipate such frequent shift of representation.

2. *High demand for flexible composition.* Almost no single AI technique can be claimed as being omnipotent. Most of the techniques are heuristic in the sense that it works well in some domain, but not in other domains. Most of the time, more than one search technique has to be combined to yield satisfactory performance. Reuse, therefore, means not so much as the direct recycling of a predetermined set of search techniques, but instead the ability for future users to pick and choose various techniques that fits their domain, and to compose these techniques together in a flexible manner.

3. *Obscurity of module boundary.* AI techniques are very difficult to modularized. There are implicit interaction and hidden dependency among various components of the system. It takes a very in depth understanding of these techniques in order for the system architect to cleanly isolate the components.

The goal of developing the Plan++ framework is to provide an architecture in which individual search techniques can be mixed and matched in a flexible manner, so that search technogies can be readily applied to a wide range of application domains.

4 Encapsulating the Variation of State Representation

4.1 Variations in State Representation

The core search facilities is provided by a *search engine* object. Users construct a search engine object by supplying their problem description, and subsequent invocation of a member function will return solution search path to the users.

```
class Search {
public:
    ....
    Search(const State& start_state, ....);
    SearchPath<State> findSolution();       // Get solution. Return search path.
    ....
};
```

Now, within the `findSolution()` member function we implement the generic search algorithm in section 2. It is clear that the implementation of `findSolution()` needs a way to test if a state object satisfy the goal (*goal verification mechanism*), and a way to generate the neighbouring states (*successor generation mechanism*). We want to give the users of Plan++ the freedom to represent their states in a way that best fits their application domain. As such, the search engine should assume minimal knowledge about the implementation of the state objects. However, both the goal verification and the successor generation mechanisms are dependent on the search domain and its representation:

- Different domain have different ways of specifying and verifying a goal. As we move from one domain to another, or as the representation of states changes, a goal will be represented differently, and the algorithm used for checking a goal will be different.
- Different domains have different state space topologies. In particular, the representation of operators determines how the successors are generated.

In fact, even if we stay in one domain, and fix the representation of states, both the goal verification and successor generation mechanisms may still vary:

- As the application mature, one might want to search for goals that previous goal specification method fails to represent.
- One might discover that the reformulation of the search problem by altering the topology of the search space might speed-up the search procedure. This can be achieved by providing multiple successor generation strategies and allowing users of the application to configure the system with one of the alternative strategies. A real-life example is the SNLP planner [5] and its variants with various threat removal strategies [8]. All SNLP-based planners search in the same plan space. The implementation of states and goal verification mechanism are therefore fixed. However, the successor generation mechanism is different for different threat-removal strategies.

To summarize, we are dealing with the following forces:

- The generic search procedure is more or less standard. Its behaviour varies when we have a different mechanism for verifing goals and generating successors. We want to provide a way for the users of our framework to configure the search engine with the appropriate behaviour.
- We anticipate that both the goal verification and successor generation mechanisms will evolve as the users attempt to build increasingly sophisticated problem solvers. We want to provide an architecture that supports the enhancement and adaptation of the search engine.
- The goal verification and successor generation mechanisms access the state objects, which the search engine should assume zero knowledge if flexibility is to be achieved. We want to completely insulate the search engine from any access of the state objects.

To resolve the above forces, we adopted a design that we later recognized to be an incarnation of the Strategy pattern.

4.2 Goal Verifier

The Strategy pattern is applied to encapsulate the goal verification mechanism. We define a *goal verifier* class `Goal<State>`, which is a parameterized abstract base class.

```
template <class State>
class Goal {
public:
    ...
    virtual bool satisfied(const State& state) = 0;
    ...
}; /* Goal */
```

We then introduce into the search engine a pointer that refers to the above abstract base class.

```
template <class State>
class Search {
public:
    Search(const State& start_state,
           Goal<State>* goal,
           ....);
    SearchPath<State> findSolution();      // Get next solution
    ....
private:
    Goal<State> *goal_;                    // Pointer to goal object
    ....
}; /* Search */
```

When the `findSolution()` function needs to check if a state statisfies the goal of the search, it delegates the responsibility to the object referenced by `goal_`.

```
template <class State>
SearchPath<State> Search<State>::findSolution() {
    ....
    if (goal_->satisfied(state))
        ....
    ....
} /* findSolution */
```

With the above design, even if we change the goal verification mechanism, the search engine does not need to be changed. All that is required is that the users configure the search engine with a different concrete goal verifier object. The design is summarized in figure 1.

We illustrate the implication of this design using the waterjug example domain. Suppose the states in this domain are encapsulated in the class `Jug`. The goal of the waterjug problem is to reach a state in which one of the jugs holds a specific volume of liquid. To implement such goal verifier, we define a concrete subclass of `Goal<Jug>`.

```
class JugGoal : Goal<Jug> {
```

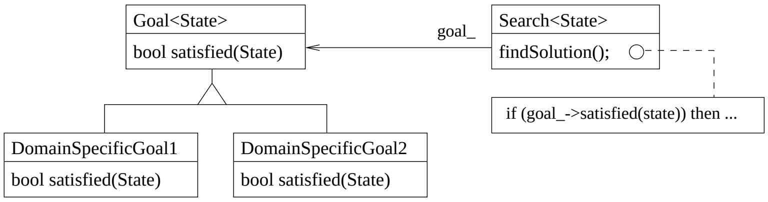

Figure 1: Goal

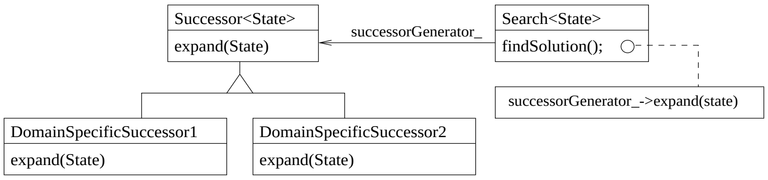

Figure 2: Successor

```
public:
    JugGoal(int target_volume) : t_(target_volume) { }
    bool satisfied(const Jug& j) {
        // Check if one of the jug in state j holds target_ litre of liquid.
    } /* satisfied */
private:
    int t_;
}; /* JugGoal */
```

Now, if we change our mind, and want to search for states so that the volume of both jugs are specified. For example, we want jug A to hold 1L and jug B to hold 2L. To allow such goal to be specified, we define a second goal verifier class for the jug domain.

```
class JugGoal2 : Goal<Jug> {
public:
    JugGoal2(int t1, int t2) : t1_(t1), t2_(t2) { }
    bool satisfied(const Jug& j) {
        // Determine if both jugs in j satisfied the goal condition.
    } /* satisfied */
private:
    int t1_, t2_;
}; /* JugGoal2 */
```

Notice that the change of the definition of a goal in the waterjug domain does not require any modification of the search engine.

4.3 Successor Generator

The Strategy pattern can be applied to isolate the variation of the successor generation mechansim in a similar way (see figure 2). We define a *successor generator* class `Successor<State>`, which, again, is a parameterized abstract base class.

```
template <class State>
class Successor {
public:
    ....
    virtual List<SearchStep<State> > expand(const State& state) = 0;
    ....
}; /* Successor */
```

Again, we introduce an abstract reference from the search engine to the abstract `Successor<State>` class, and delegates successor generation responsibility to the referenced object.

```
template <class State>
class Search {
public:
    Search(const State& start_state,
           Goal<State>* goal,
           Successor<State>* successor,
           ....);
    SearchPath<State> findSolution();          // Get next solution
    ....
private:
    Goal<State> *goal_;                    // Pointer to goal object
    Successor<State> *successor_;          // Pointer to successor generator
    ....
}; /* Search */

template <class State>
SearchPath<State> Search<State>::findSolution() {
    ....
    successor_->expand(state)
    ....
} /* findSolution */
```

Let us examine how the above design streamline the evolution of the waterjug planner. To implement the waterjug planner, a concrete successor generator class is derived from the abstract base class `Successor<Jug>`. The `expand()` method is overriden.

```
class JugSuccessor : public Successor<Jug> {
  ....
  List<SearchStep<Jug> > expand(const Jug& jugState) {
      // apply all possible operators to this jugState
```

```
      //   ie. fillJugA(), emptyJugB(), pourContentsOfJugBIntoJugA(), etc.
      // return list of all possible successor states;
  } /* expand */
  ....
};
```

After examining the solution of a number of instances of the waterjug problem, we notice that the subsequence "emptyJugB, pourContentsOfJugBIntoJugA" are often used in a solution. We then conjecture that introducing this subsequence as a new operator to the problem might speed-up the search significantly. We build a new successor class for the problem.

```
class JugSuccessor : public Successor<Jug> {
  List<SearchStep<Jug> > expand(const Jug& jugState) {
      // A variation of JugSuccessor that has an extra operator.
  } /* expand */
};
```

After testing the new successor generator, we notice that the introduction of the new operator increases the branching factor of the search, and actually degrades the performance of the search engine. We then decide to switch back to the original successor generator. Notice that none of the other classes has to be changed during the above evolution of the program[1].

4.4 Retrospection

By applying the Strategy pattern, we effectively insulate the search engine from the representational dependecies of the state class. The search engine always manipulate the states via the abstract services provided by the goal verifier class and the successor generator class. Because of the reification of the two algorithms, and because of the abstract coupling among the search engine and the two reified classes, users can vary the behaviour of the algorithms independently, and can configure the search engine with different variations of the reified classes.

Our previous experience in building search-based applications allows us to identify the forces very quickly. Our familiarity of the design patterns in [2] turned out to help us in a very indirect way. It is the design principles shared by the design patterns that we tried to mimic. In particular, the repeated emphasis of composition as a desirable alternative to inheritence had greatly inspired our design. We were not able to recognize our design as an instance of Strategy until we carefully re-examine the patterns in [2].

At our first sight, the problem seemed to have a natural, but later found to be inadequate, solution. We thought we could define an abstract base class for the state objects, and provide

[1] The technique of building new operators by combining the old ones are called macro-operator learning. It is known that such learning does not guarantee speedup for the resulting problem solver. Such anomaly is called the *utility problem*. The scenario discussed here is only a mock-up example. Real macro-operator learning is more complex.

an abstract interface to access the goal verification and successor generation mechanisms. Users supply a concrete subclass of the abstract state class when moving to a new search domain.

```
class State {
public:
    ...
    virtual bool satisfied();            // Is goal satisfied?
    virtual List<State> successor();    // Generate a list of successor states.
    ...
};
```

This seemingly innocent arrangement creates more problems than what it solves. We rejected this inheritence-based solution based on the following reasons:

- Although the representation of states is fixed in one domain, both the goal verification and successor generation mechanisms might vary. Such arrangement forces the users of the framework to define a new concrete subclass of the state class if any one of the two mechanisms changes.
- Still worse, even if the goal verification mechanism is fixed, the exact goal that is being sought for is almost never fixed. We then have to provide a way for the users to specify the goal to be sought for. Although one can always do that in the constructor of the state object, it is clearly unnatural and inextensible (the state abstraction has to be re-opened everytime you have a new specification method for a goal). The same reasoning applies to the case of successor generation.
- Moreover, the goal specification has to be carried by every state. This is unreasonably expensive. The same applies to successor generation. Some successor generation mechanisms are programmable interpreter. They interpret some externally specified topology of the search space. For such successor generation mechanisms, the above arrangement will require all states to have access to the neighbourhood specification.

5 Encapsulating Variation of Search Control Strategy

5.1 Variations in Search Control Strategy

Search control strategy is the policy by which the search engine is employed to select a node for expansion. In almost all cases, a search control strategy is implemented with the help of a node store; generated nodes are stored in the node store. Everytime the search routine expands a node, it requests a node from the node store. A search control strategy therefore regulates which node in the node store should be returned to the search routine. For example, breadth-first search (BFS) is realized by making the node store a queue, thereby imposing a first-in-first-out (FIFO) ordering in node expansion. As such, the node store together with the ordering of nodes imposed by the search control strategy forms the instantaneous state of the search process.

Over the years, the AI community has developed many different search strategies. The simplest ones are strategies like breadth-first search or depth-first search (DFS). Others like depth-first iterative-deepening (DFID) [4] or A^* utilizes computational resources more efficiently and intelligently. No single search strategy is the best in all cases. Users must analyze the context of the application, and figure out, either empirically or heuristically, the search strategy that best fits their needs. In other words, the implementation of the search engine must be reconfigurable so that users can select the right search control strategy to use, either at compile-time or at run-time.

On the other hand, the search routine could be used in many different ways. Some users want to find one solution, some want to find all, while others might want to examine solutions one by one, picking the one that they are satisfied with. A search routine should therefore be resumable; that is, after finding a solution, the users should be allowed to restart it again. Multiple solution can then be obtained by successively resuming the search.

To summarize, we are dealing with the following forces:

- We want to avoid a permanent binding between the search engine and its implementation of its node store, so that reconfiguration can occur at both compile-time and run-time.
- We anticipate that, as the search application developed by our users get more and more sophisticated, they will want to experiment with newer and more special purpose search strategy. Standard search strategies provided by our library is going to be become limited. We need an arrangement which allows them to create new search control strategy without recompiling the rest of the application.
- To allow users to resume a search process, the instantaneous state of the search has to persist the life-time of the search process. We need a design in which the instantaneous state of the search can be retained so that the search process can be resumed and the next solution is sought.

5.2 Search Controller

The Bridge pattern is applied to resolve the above forces. In particular, we introduce a *search controller* class to encapsulate the implementation of the node store.

```
template <class State>
class SearchControl {
public:
    ....
    virtual SearchNode<State> remove() = 0;
    virtual void insert(const SearchNode<State>& node) = 0;
    ....
}; /* SearchControl */
```

The search engine maintains an abstract coupling with the search controller, thereby accessing its functionalities via a well-defined interface.

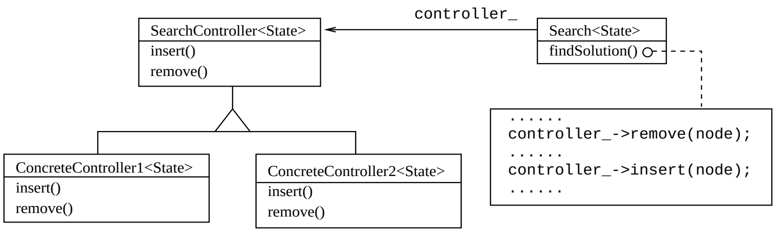

Figure 3: Controller

```
template <class State>
class Search {
public:
    Search(const State& start_state,
           Goal<State> * goal,
           Successor<State> * successor,
           SearchControl<State> * control);
    SearchPath<State> findSolution();      // Get next solution
    ....
private:
    Goal<State> *goal_;                    // Pointer to goal verifier
    Successor<State> *successor_;          // Pointer to successor generator
    SearchControl<State> *control_;        // Pointer to search controller
    ....
}; /* Search */

template <class State>
SearchPath<State> Search<State>::findSolution() {
    ....
    control_->remove()
    ....
    control_->insert(node)
    ....
} /* findSolution */
```

To build a new search controller, one simply develop a concrete subclass of the `SearchControl<State>` class. This design, as depicted in figure 3, resolves the forces in the following way:

- Users can configure the search engine with any search control strategy by passing the appropriate concrete search controller into the constructor of the search engine.
- There is a generic architecture for incorporate new search control strategies that are not found in our library.
- Since the search controller captures the instantaneous state of a search process, and since it has a life-time independent of the search process itself, it can be used to resume the search process when necessary.

Again, we illustrate the implication of the above design by looking at some examples. Our waterjug planner uses a breadth-first search strategy to conduct the search. The `BFS<State>` controller uses a queue as the node store to impose a FIFO ordering in node expansion.

```
template <class State>
class BFS : public SearchControl<State> {
public:
    ....
    void insert(const SearchNode<State>& node) {
        queue_.enqueue(node);           // Put node into queue.
    } /* insert */

    SearchNode<State> remove() {
        SearchNode<State> node;
        if (!queue_.empty()) {          // Proceed if queue isn't empty.
            node = queue_.front();      // Fetch the first node in the queue.
            queue_.dequeue();           // Remove the first node from queue.
        } /* if */
        return node;                    // Return node.
    } /* remove */
protected:
    Queue<SearchNode<State> > queue_; // Queue as the node store.
};
```

Examining the search performance, we notice that the same nodes in the search spaces are expanded more than once. This is due to the fact that the same state can be generated by two parent nodes, and both instances are introduced into the queue. It means that the above BFS controller should only be used with a problem in which the search space is a tree. We design a second BFS controller which checks if a node is already in the queue before inserting it. Let us call this controller BFS2.

```
template <class State>
class BFS2 : public SearchControl<State> {
public:
    ...
    void insert(const SearchNode<State>& node) {
        if (! set_.member(node)) {      // Proceed if node hasn't been generated.
            queue_.enqueue(node);       // Put node into queue.
            set_.insert(node);          // Record that node is generated.
        } /* if */
    } /* insert */
    // remove() remains the same.
    ...
protected:
    Queue<SearchNode<State> > queue_; // Queue as the node store.
    Set<SearchNode<State> > set_;     // Set to carry generated nodes.
```

```
};
```

Notice that no other part of the search framework has to be changed when we switch to the BFS2 controller.

5.3 Search Controller Decorator

Alert reader will notice that most of the code for `BFS<State>` and `BFS2<State>` are the same. The commonality of code is in fact not an accident. Consider the implementation of a DFS controller and its variation that searches a graph.

```
template <class State>
class DFS : public SearchControl<State> {
public:
    ....
    void insert(const SearchNode<State>& node) {
        stack_.push(node);          // Push node into stack.
    } /* insert */

    SearchNode<State> remove() {
        SearchNode<State> node;
        if (!stack_.empty()) {      // Proceed if stack isn't empty.
            node = stack_.top();    // Fetch the top node in the stack.
            stack_.pop();           // Delete the top node from stack.
        } /* if */
        return node;                // Return node.
    } /* remove */
protected:
    Stack<SearchNode<State> > stack_;
}; /* DFS */

template <class State>
class DFS2 : public SearchControl<State> {
public:
    ...
    void insert(const SearchNode<State>& node) {
        if (! set_.member(node)) {    // Proceed if node hasn't been generated.
            stack_.push(node);        // Push node into stack.
            set_.insert(node);        // Record that node is generated.
        } /* if */
    } /* insert */
    // remove() remains the same.
    ...
protected:
    Queue<SearchNode<State> > queue_; // Queue as the node store.
```

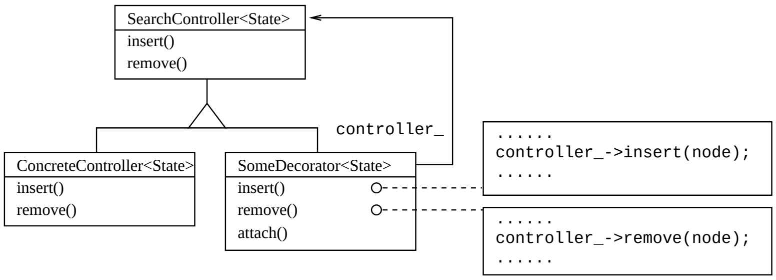

Figure 4: Controller Decorator

```
    Set<SearchNode<State> > set_;     // Set to carry generated nodes.
};
```

The repetition of code simply suggest that the notion of "graph version of a search control strategy" is an individual unit of reuse. In general, there are many variations to a given search control strategy. For example, a bounded search version of a search control strategy restricts search depth to a specific level, and a graph version of a search control strategy checks if a node is visited already before it is inserted to the node store. There is a depth-first iterative-deepening version of A^*. In fact, depth-first iterative-deepening itself could be considered a variation of DFS. Coding such variations for each search control strategy could be a very tedious, monotonic task. It will be very desirable if one can define the same variation once and then apply it to all search controllers. How do we capture behavioral variations of search controllers as reusable objects that can be flexibly combined with existing search controllers?

Examining the actual variations, we realize that they are nothing more than message transformations, that is, they perform additional regulation before the actual node insertion or removal occur. Decorator seems to be a natural choice to resolve the above forces. In particular, we define a `ControllerDecorator<State>` class.

```
template <class State>
class ControlDecorator : public SearchControl<State> {
public:
    ControlDecorator(SearchControl<State> * controller);    // Constructor
    virtual SearchNode<State> remove() = 0;                 // Remove a node
    virtual void insert(const SearchNode<State>& node) = 0; // Insert a node
    ....
private:
    SearchControl<State> *controller_;          // Controller being decorated
}; /* ControlDecorator */
```

Reusable variations of behaviour can now be encapsulated into the subclasses of the controller decorator class. They perform message transformation, and delegate the transformed message to the controller object referenced by the member variable `controller_` in `ControlDecorator<State>`. With this design, as depicted in figure 4, a decorator for graph searching can be coded as follows.

```
template <class State>
class GraphControl : public ControlDecorator<State> {
public:
    GraphControl(SearchControl<State> * control)
        : ControlDecorator<State>(control) { .... }

    void insert(const SearchNode<State>& node) {
            ControlDecorator<State>::insert(node
        if (! set_.member(node)) {
            ControlDecorator<State>::insert(node);
            set_.insert(node);
        } /* if */
    } /* insert */

    SearchNode<State> remove() {
        return ControlDecorator<State>::remove();
    } /* remove */
    ....
protected:
    Set<SearchNode<State> > set_;
}; /* GraphControl */
```

Now, composing a graph version of BFS for the waterjug domain becomes a simple task.

```
    SearchControl<Jug> *bfs  = new BFS<Jug>();
    SearchControl<Jug> *ctrl = new GraphControl<Jug>(bfs);
```

As we have seen, the application of Decorator pattern allows reusable variations of search control strategies to be isolated as independent components that can be flexibly composed with other search controllers.

5.4 Retrospection

The idea of a search controller has been implemented in other major AI architectures including Prodigy [6] and UCPOP [1]. Although their design was realized in Common LISP, the reason for using search controllers is more or less the same as ours, that is, to allow users to configure the search engine with arbitrary search control strategy and to let them define new search controller. Even in the very early stage of our design we already decided to follow this practice. We noticed that the resulting structure of our design looked very familiar, so we consulted [2], just to figure out that our design is simply an instance of Bridge.

We have also considered the use of Template Method. We could have defined the `remove()` and `insert()` services as virtual member function in the search engine class `Search<State>`. Subclasses of the search engine would then need to implement the node store and override the two hook methods. Although this arrangement achieves more or less what we need, it is very inflexible. The Decorator pattern cannot be applied to factor out common variations had we applied the Template Method instead of Bridge, in which the node store becomes a first class object manipulatable outside of the search engine.

The need for Decorator was immediately identified as soon as we start to populate our library with concrete search controllers. We noticed the repetition in our code, and realized that Decorator is the most natural solution to our problem. The Decorator pattern is the only pattern that we consciously applied.

6 Building Heuristic Problem Solver

Many heuristic problem solvers requires the coupling with a first-principle problem solver. There are at least two cases such coupling occurs:

1. *Incomplete Heuristic.* It might be that the heuristic problem solver does not guarantee completeness (i.e. it might fail to find a solution), and therefore need to fall back to a first-principle (though less efficient) problem solver. A case-based reasoning system [3] is such an example. A case-based planner relies on previous problem solving experience to speed up planning. However, as the immediate problem cannot be solved with the use of previous cases, it needs to invoke a traditional problem solver (like a search routine in the previous sections) to perform systematic search.

2. *Heuristic Reduction.* It might be that the responsibility of the heuristic problem solver is simply to reduce the difficult problems into one or more easier subproblems. It then delegates the problem solving to a first-principle problem solver to complete the processing. Examples of this would be decomposition-based planners and abstraction-based planners. Decomposition-based planners decompose a hard problem into several easier subproblems, and invoke first-principle planners to attack the subproblems. Abstraction-based planners [9] solve the more critical part of the problem, and then use the result to constrain the search of the first-principle planners.

To build a heuristic problem solver, we then have to deal with the following forces.

1. *Flexibility of configuration.* The effectiveness of heuristic problem solving techniques like abstraction and decomposition is domain-dependent. Sometimes the introduction of a heuristic improves search efficiency and quality, sometimes it degrades both. You want to give the users choice and flexibility of picking and combining the right heuristic to be used in different situations. That is, you want user code to be independent of the heuristic choosen, so that reconfiguring the system heuristic becomes straightforward.

2. *Flexibility of composition.* The distinction between heuristic problem solver and first-principle problem solver is relative. A heuristic problem solver can be the first-principle problem solver of another more complex problem solver. For example, we could have a case-based reasoner which make use of an abstraction-based planner as its principle problem solver. The abstraction-based planner might make use of an exhaustive search engine as its first-principle problem solver. We want a structural mechanism that facilitates such recursive composition.

3. *Modularity.* Hard coding the first-principle problem solving mechanism into a heuristic problem solver makes evolution of either one unnecessarily difficult. When one has to be changed, the other will have to be considered together because of the lack of encapsulation boundary between the two.

How do we allow various problem solving techniques to be composed flexibly, so that we can build more complex problem solvers out of several simpler problem solvers?

We faced such a design problem when we tried to build a case-based reasoner for a bus routing problem. The case-based reasoner has a memory of previously solved cases. When new problem arrives, it checks if a plan in the case library can be adapted to solve the problem. In case it cannot, then an exhaustive search engine is employed to solve the problem, and the result will be saved in the case memory. We are not sure what kind of search engine we need in order to achieve best performance. Therefore, we want to delay the decision. We notice that we would probably need a similar mechanism for other kind of heuristic problem solvers that we might want to build in the future. We then look for a general solution form to this design problem. We look for a solution form that satisfies the following.

- We want a loose coupling between the heuristic problem solver and the first-principle problem solver. The heuristic problem solver will delegate problem solving responsibilities to the first-principle problem solver as soon as it realize that it cannot solve the problem on its own.

- There should be facility for configuring a heuristic problem solver with different kind of first-principle problem solver, so that the heuristic problem solver code can be reused without recompilation.

- The heuristic problem solver and the first principle problem solver should evolve separately.

- There should be a general approach to compose very complex problem solvers by using building blocks of elementary problem solvers.

- Also, retracting a heuristic problem solving technique from a composite problem solver should be made easy.

As we look for solution form that satisfies the mentioned requirement and easily incorporable into our bus routing planner, we recall the frequent use of abstract coupling in [2] to resolve similar forces. We then come up with the following design for our bus routing planner. First, we define an abstract bus routing planner class.

```
class Abs_BR_Planner {
public:
    virtual solve(...) = 0;
}; /* Abs_BR_Planner */
```

Second, we capture our case-based reasoning mechanism into a concrete subclass of `Abs_BR_Planner`.

```
class CBR_BR_Planner {
public:
    CBR_BR_Planner(Abs_BR_Planner *first_principle_solver);
    virtual solve(....);
    ....
private:
    Abs_BR_Planner *first_principle_solver_;
}; /* CBR_BR_Planner */
```

The constructor of `CBR_BR_Planner` simply initializes the `first_principle_solver_` pointer to the abstract planner being passed into the constructor.

```
CBR_BR_Planner::CBR_BR_Planner(Abs_BR_Planner *first_principle_solver)
    : first_principle_solver_(first_principle_solver) { .... }
```

The problem solving function `solve()` delegates problem solving responsibility to the planner referenced by `first_principle_solver_` when it fails to find cases that solves the same problem.

```
CBR_BR_Planner::solve(....) {
//  if not able to find cases from library then
//      first_principle_solver_->solve(....);
} /* solve */
```

Now, we can provide various form of exhaustive problem solver as concrete subclasses of the `Abs_BR_Planner`.

```
class BFS_BR_Planner {
public:
    ....
    solve(....) {
        SearchControl<BRState> *control;
        Search<BRState> *search_engine;
        control = new GraphControl<BRState>(new BFS<BRState>);
        search_engine = new Search<BRState>(.... control);
        search_engine->findSolution();
        ....
    } /* solve */
}; /* BFS_BR_Planner */
```

To configure a case-based planner with a best-first search engine as first-principle solver, we do the following.

```
Abs_BR_Planner *planner = new CBR_BR_Planner(new BFS_CBR_Planner);
```

Right after we sketched the above solution on a whiteboard, we immediate noticed that the general form of this solution is actually the Chain of Responsibility pattern. In the general case, more than one heuristic problem solver will be available, and they can chain up into a composite problem solver. The problem solving message is passed along the chain so that every problem solver has a chance to tackle its own share of the problem. Because the structure of heuristic problem solvers for different domain are different, we are not yet able to incorporate this architecture into the current version of Plan++. We believe that, as our experience cummulates, a general realization of this architecture will be available in our framework. Currently, we take note that the same pattern can be applied to make heuristic problem solvers more composible.

7 Summary

Frequent shift of representation, high demand for flexible composition, and obscurity in module boundary make reusing AI techniques extremely nontrivial. In this paper, we summarized our experience of applying design patterns to the design of a reusable framework for search-based applications. We learned several lessons in this process:

- The complexity of building AI applications can be managed by good object-oriented design practices. Design pattern catalogs make such knowledge accessible to AI system builders.
- The terminologies offered by the design pattern catalog greatly improved our communication. At first, we had difficulty communicating the proper use of controller decorator to our users. But then we use the design pattern catalog to introduce the pattern behind the code, then thereafter, they grasp it quickly.
- The way design patterns catalogs assist a software designer can be very indirect. Although sometimes we directly discern the applicability of a pattern (as in the case of designing the controller decorator), most of the times, it is the understanding of the principles behind the design patterns that inspire our design practices. Usually, it is only after finishing design that we recognize that our design turn out to be an instance of a known pattern.
- The above phenomenon reinforces the notion that the explanation and elaboration of how forces are resolved in many pattern genre is the most valuable part of a pattern. Most of the times, the actual solution form is not recalled, but the technique being used to resolve forces are quickly remembered and applied.
- We also notice that we have not done a lot of catalog searching when we design our framework. The design pattern catalog served more as a training tool to us than a catalog that we have to consult when we face a design problem. To us, design pattern catalogs are not a design cookbook: it makes us better cook instead of giving us recipes to follow. However, we keep questioning ourselves, could this actually be a limitation of the existing pattern genre? Are there ways for design pattern catalogs to assume a more participatory role than a mere background training tool? Can the pattern genre be improved so that people who are not familiar with patterns would still be able to look up a solution as they need it? We believe a better indexing or classification scheme would greatly improve the usefulness of a design pattern catalog.

References

[1] A. Barrett, K. Golden, J. S. Penberthy, and D. Weld. Ucpop user's manual, version 2.0. Technical Report 93-09-06, Department of Computer Science and Engineering, University of Washington, 1993.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[3] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, 1994.

[4] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(2):97–109, 1985.

[5] D. McAllester and D. Rosenblitt. Systematic nonlinear planner. In *AAAI '91*, pages 634–639, 1991.

[6] S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Dept. of Computer Science, Carnegie-Mellon University, 1989.

[7] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[8] M. Peot and D. Smith. Threat-removal strategies for partial-order planning. In *AAAI '93*, pages 492–499, 1993.

[9] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.