Design Patterns for Planning Systems

Qiang Yang and **Philip W. L. Fong** and **Edward Kim**
School of Computing Science
Simon Fraser University
Burnaby, BC Canada V5A 1S6
qyang@cs.sfu.ca

Abstract

In this work, we are interested in building a software engineering discipline for planning system design. Our objective is to enable planning systems to become more configurable and modular, with the help of object libraries capturing well designed experiences and a common planner-design pattern catalog. It is hoped that the planning systems thus constructed will be more reusable and modifiable. It is also hoped that this effort will contribute to the movement towards industrially applicable planning systems, to supply this dynamic subfield of AI with a rigorous software engineering discipline than just smart algorithms for plan generation.

To this end, we focus on an object oriented design methodology to planning. We focus on a collection of design patterns for modularizing different search-related parts of a typical planning system. We illustrate our concept using the C++ language, although our experience apply equally well to all object oriented languages. This work is represents our continuing research in knowledge acquisition and maintenance effort in planning systems design.

Introduction

This paper summarizes our experience in applying design patterns to develop intelligent planning 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 focus in text-based planning systems, are all done on top of this framework.

In this paper, we report how the design pattern catalog (GHJV94) has shaped the way we design our planning system's framework. As a starting point, we will concentrate on search aspects of planning algorithms, leaving the rest of the planning framework as a part of our future work. Our interest here is twofold. First, we want to demonstrate how design patterns can be employed to structure a search-based planner design, so that the 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. We want to report how the design pattern catalog allowed us to flexibly configure planning systems for different applications. Our work follows closely the current trend in AI Planning in making the systems more modular (BVB96; BHB97).

We will first introduce the concept of design patterns. We then discuss planning as search (section), and the need to reuse past planning frameworks (section). We will show several design patterns useful for modular planner design applied to our planner framework. We report the issues 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 .

Design Patterns

Design pattern is a recent movement in software engineering. The basic concept is to reuse design knowledge. Design pattern is a genre by which standard, expert-tested solution to canonical design problems is rigorously is documented. The term design pattern could also refer to the solution itself. Such design patterns usually solve a design problem by imposing new organization in the software, and layout a particular protocol in which individual computational entities should interact. By conforming to such restriction, the target program would become more maintainable. For example, iterators "provide a way to access the element of an aggregate object sequentially without exposing its underlying representation." Iterators can be found in nearly all object-oriented container class library. It defines a protocol by which client subprograms can access the internal of an aggregate object.

Design patterns can be shown informally as boxes with connections between them. A popular notation is an class/object diagram known as the OMT diagram (Object Modeling Technique). Figure 1 (a) shows the

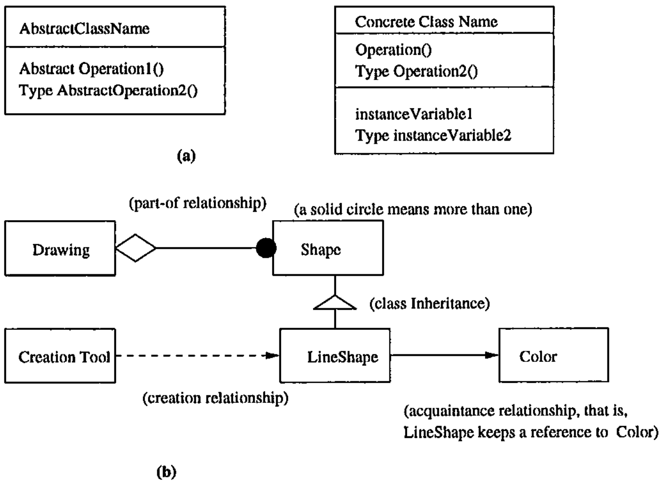

Figure 1: Introduction to OMT diagram

OMT notation for abstract and concrete classes. A class is denoted by a box in the figure. The key operations of the class appear below the class name. Any instance variables appear below the operations.

Figure 1 (b) shows various relationships between classes. The OMT notation for class inheritance is a triangle connecting a subclass (LineShape) to its parent class (Shape). The rest of the notations are explained in the figure. In short, OMT is a convenient way of communicating good design experiences in object oriented design, showing relationships between classes and objects.

A collection of well-used design patterns forms a design pattern catalog. Such catalogs are widely available on the Internet and in various book formats. Work in design pattern mining from large-scale software code is also starting to generate important results.

Planning as Search

A significant number of AI and combinatorial optimization problems are instances of heuristic *state-space search* (Pea84). A state space is a set of *states* plus a set of *operators*. Planning is no different. In planning one can differentiate between state-based search (BK95) and plan-space search (Wel94; Yan97). In both frameworks, one define a search engine and a processing module for generating a set of successor nodes from a given current node. Both styles of planning are instances of a state-space search.

In a state-space search framework, 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:

```
plan(init-plan, operators, goal-condition):
    mark init-plan as "generated";
    while (not all generated nodes are expanded) do
        select a node $n$ that is generated but not expanded;
        if ($n$ satisfies the goal-condition) then return $n$;
        generate all or some of the successor nodes $n'$ of $n$;
        mark all $n'$ as "generated";
        mark $n$ as "expanded";
    end while;
    return "failure";
end Search.
```

Intuitively, a plan node is "generated" the first time it is visited. A node is "expanded" if all its neighboring nodes are "generated". The efficiency of a search is partly 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. The efficiency is also defined by how and in which order a successor node is generated; in partial order planning, a node is generated by resolving threats and achieving open-preconditions. In HTN planning, a node can also be generated by reducing a non-primitive task by a task network. In case based planning a variety of repair operations can be used to generate successor nodes.

Throughout the paper, we will use a sample AI planning domain to illustrate the effect of applying the design patterns. An instance of the *water-jug* 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 $k L$ 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 pourContentsOfJugIntoOtherJug(); 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.

Reusability in Planner Design

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 technologies can be readily applied to a wide range of application domains.

Encapsulating the Variation of State Representation

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.

```
template <class State>
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 . 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 (MR91) and its variants with various threat removal strategies (PS93). 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 issues:

- The generic search procedure is more or less standard. Its behavior varies when we have a different mechanism for verifying goals and generating successors. We want to provide a way for the users of our framework to configure the search engine with the appropriate behavior.
- 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 issues, we adopted a design that we later recognized to be an incarnation of the Strategy pattern.

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>
```

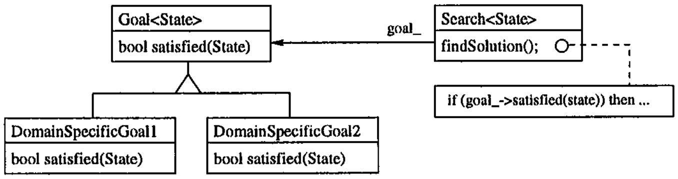

Figure 2: Goal

```
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 2.

We illustrate the implication of this design using the water-jug example domain. Suppose the states in this domain are encapsulated in the class JugState. The goal of the water-jug 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<JugState>.

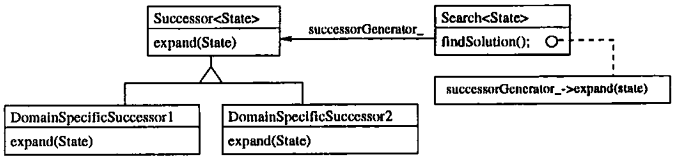

Figure 3: Successor

```
class JugGoal : Goal<JugState> {
public:
    JugGoal(int target_volume) : t_(target_volume) { }
    bool satisfied(const JugState js) {
        // Check if any of the jug in state js
// holds target_ litre of liquid.
    } /* satisfied */
private:
    int t_;
}; /* JugGoal */
```

Now, suppose we decide to use another goal condition for our planner, 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<JugState> {
public:
    JugGoal2(int t1, int t2) :
t1_(t1), t2_(t2) { }
    bool satisfied(const JugState js) {
// Determine if both jugs in js
//satisfy the goal condition.
    } /* satisfied */
private:
    int t1_, t2_;
}; /* JugGoal2 */
```

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

We can also easily modify the goal condition for it to serve a partial-order planner. In this case the class JugState encodes all necessary elements of a partial order plan, including steps, variables, initial and goal steps, causal links, threats and open preconditions.

```
class JugGoal3 : Goal<JugState> {
public:
    JugGoal3(): { }
    bool satisfied(JugState partial_order_plan) {
        // return True when
// (1) no open preconditions exist and
//   (2) no (negative) threats exist
// for all causal links
    } /* satisfied */
}; /* JugGoal3              */
```

Successor Generator

The Strategy pattern can be applied to isolate the variation of the successor generation mechanism in a similar way (see figure 3). 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 delegate 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 */
```

State-based Search Let us examine how the above design streamline the evolution of the water-jug planner. To implement the water-jug planner, a concrete successor generator class is derived from the abstract base class Successor<Jug>. The expand() method is overridden.

```
class JugSuccessor :
public Successor<JugState> {
  ....
  List<SearchStep<JugState> >
expand(const JugState js) {
// 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 water-jug problem, we notice that the macro operator "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<JugState> {
  List<SearchStep<JugState> >
expand(const JugState js) {
// 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. (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.)

Likewise, we can also easily choose to use a partial-order planning algorithm to implement successor generation. Here is an example:

```
class JugSuccessor :
public Successor<JugState> {
// A partial order planning example...
  ....
  List<SearchStep<JugState> >
expand(const JugState js) {
// Jug state js is now a partial order plan.
// Consider the plan js:
// If there are threats to a causal link
//  resolve the threats,
//    producing successor states;
// Else if there is an open precondition ?pre
//  Find all operators that can achieve ?pre
//  Find all existing steps that can achieve ?pre
//  Produce successor plans by inserting
//    new causal links
//  for achieving ?pre
// Endif
// return the list of all successor states;
  } /* expand */
  ....
};
```

Encapsulating Variation in Search Control Strategy

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. Every time 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) (Kor85) 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 issues:

- 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.

Search Controller

The Bridge pattern is applied to resolve the above issues. 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.

```
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 4, resolves the issues 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.

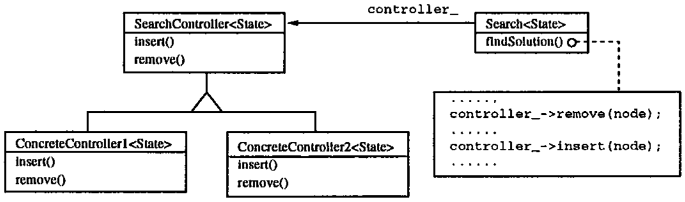

Figure 4: Controller

- 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 water-jug 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.

Search Controller Decorator

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

The repetition of code simply suggests 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 issues. In particular, we define a ControllDecorator<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 */
```

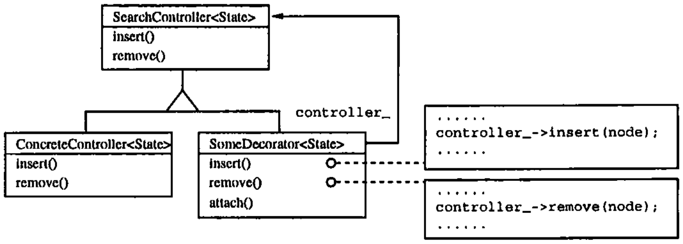

Figure 5: Controller Decorator

Here the insert and remove operations will be implemented by classes such as BFS and DFS. These classes only record the differences in data structures that are used for implementing BFS and DFS and so on. Reusable variations of behavior 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>. For example, a statement as follows can be used to instantiate the ControlDecorator by an implementation of breadth-first search control:

```
ControlDecorator<JugState> (bfs);
```

With this design, as depicted in figure 5, 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) {
    if (! set_.member(node)) {
// first perform data-structure specific operations:
        ControlDecorator<State>::insert(node);
// then perform a data-structure independent operatio
         set_.insert(node);
     } /* if */
    } /* insert */
    SearchNode<State> remove() {
    return ControlDecorator<State>::remove();
    } /* remove */
    ....
protected:
    Set<SearchNode<State> > set_;
}; /* GraphControl */
```

In the above, ControlDecorator¡State¿::insert(node) implements a data-structure dependent insertion function. The statement common to all graph control routines, set_insert(node), is now abstracted out. Now composing a graph version of BFS or DFS for the water-jug domain becomes a simple task:

```
    SearchControl<JugState>
*bfs  = new BFS<JugState>();
    SearchControl<JugState>
*ctrl1 = new GraphControl<JugState>(bfs);

    SearchControl<JugState>
*dfs  = new DFS<JugState>();
    SearchControl<JugState>
*ctrl2 = new GraphControl<JugState>(dfs);
```

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.

For example, we can define a *bounded search decorator*, which introduces a bound to the search depth of any search controller. To do that, it delegates insertion messages to the underlying controller only when the search depth is not exceeded. A removal message is always delegated to the underlying search controller as is. Such bounded search decorator can be coupled with any search controller. This saves the need to define a separate bounded version of every search controller.

Moreover, new search controllers can be built on top of existing search controller. For example, a depth-first iterative-deepening strategy repeatedly invoke depth-first search with increasing search bound until a solution is found. It has a linear space usage (as in depth-first search), but it guarantees optimality of solution (as in breadth-first search). Because its asymptotic time complexity is the same as both depth-first search and breadth-first search, it is a very appealing search strategy. Such a search controller can be defined as a decorator object coupled with a search bound decorator, which in turn is coupled with a depth-first search.

Summary

Frequent shift of representation, high demand for flexible composition, and obscurity in module boundary make reusing AI planning 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 planning 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 issues 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 issues are quickly remembered and applied.

Acknowledgment

The work is supported by grants from Natural Sciences and Engineering Research Council of Canada (NSERC), BC Advanced Systems Institute, ISM-BC and Canadian Cable Labs Fund.

References

L. Nunes de Barros, J. Hendler, and V.R. Benjamins. Par-kap: A knowledge acquisition tool for building practical planning systems. In *Proceedings of the 15th IJCAI*, pages 1246–1251. Morgan Kauffman Publishers, 1997.

Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward-chaining planner. Technical report, University of Waterloo, Waterloo, Ontario, Canada, 1995. Available via the URL ftp://logos.uwaterloo.ca:/pub/tlplan/tlplan.ps.Z.

L. Nunes de Barros, A. Valente, and V.R. Benjamins. Modeling planning tasks. In *Third International Conference on Artificial Intelligence Planning Systems, AIPS-96*, pages 11–18, 1996.

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

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

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

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

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

Daniel Weld. An introduction to least-commitment planning. *AI Magazine*, Winter, 1994:27–61, 1994.

Qiang Yang. *Intelligent Planning — A Decomposition and Abstraction Based Approach.* Springer-Verlag, 1997. ISBN 3-540-61901-1.