Un-muddling Implementation from Design via Algorithmic Design Patterns

Carolyn Pe Rosiene
Department of Computing Sciences
University of Hartford
W. Hartford. CT
rosiene@hartford.edu

Joel Rosiene
Department of Computer Science
Eastern Connecticut State University
Willimantic, CT 06226
rosienej@easternct.edu

***Abstract*—This work in progress paper presents an abstraction of problems to introduce more advanced algorithmic constructs early in the computers science sequence in a natural way. The approach is to use an abstract class that extracts the methods common to many classes of algorithms for a given problem. For example, an abstract class for divide-and-conquer algorithms may be defined first, before a specific type of sorting algorithm is defined. This refocuses the coverage onto approaches to construct algorithms for a given problem, rather than the implementation of simple existent algorithms - a top down approach starting from the class of problem.**

Keywords—design patterns, algorithms, programming design

I. *Introduction*

This work seeks to provide an abstraction of problems which is similar to how the Standard Template Library (STL) provides a common implementation of data structures. Provided that the programmer adheres to the library requirements, STL provides an implementation of the data structure, for example, a queue or heap.

It is proposed that problems (to be automated) have common characteristics. If these characteristics are identified and isolated beforehand, this facilitates its implementation using well-known problem solving design patterns. An abstraction may be developed for a selection of design patterns which identifies the methods required for the problem to be solved by a particular class of algorithm.

II. Motivation

An algorithm is a systematic solution to a problem able to be automated, and these problems are referred to as computable. Not all problems are computable. There are well-known algorithmic design patterns that may be used to "attempt" to discover an algorithm for a problem. For example, the greedy design pattern proceeds by selecting the best "action" at each step until the problem is solved. When the greedy design pattern leads to an algorithm, it may be the optimum solution since we are selecting the "best" action at each step. Some problems, such as making change with the minimum number of coins require additional conditions, such as the denomination of the coins, for the greedy design pattern to lead to an algorithm.

Often, students are exposed to specific problems from an implementation-first approach, whether it is an algorithm or a program. Only later are the more general forms introduced, after exposure to several known solutions to specific problems. This "muddles" the exposition, mixing the implementation with design. This goal is to present the major approaches to solve any problem, and extract the methods common to all problems.

Not all design patterns will lead to an efficient algorithm, or even an algorithm. However, it is hoped that by early exposure to the concepts, the students can generalize the design pattern to other problems.

The National Institutes of Standards and Technology (NIST) Software and Systems Division, maintains the Dictionary of Algorithms and Data Structures (DADS) [1]. This dictionary defines all the standard algorithmic types, and the known data structures and relative merits of each. Brassard, et al. [2], provides a partial treatment, and Russell and Norvig[3] extends the algorithmic design patterns to include rational agents.

The next section will provide a unified treatment by defining an abstract problem class that can be used in all algorithm design patterns when possible. The abstraction has been implemented in C++17.

III. Methodology

Using the standard definitions created by NIST [1] and Brassard, et al. [2], we create an abstract Problem class that exposes the methods that must exist for an algorithm design pattern to be applied. This promotes creative thinking about the problem, how to decompose it into subproblems, and what data structures could be used to improve the design. The alternative is to build up a suite of solutions to existent problems and hope, through repetition, that the student can generalize from the specific examples to the general case (e.g., from a bubble sort of a list, to a divide and conquer design pattern).

The following six design patterns are analyzed to abstract a base class from the NIST [1] descriptions:

1. Dynamic
2. Las Vegas
3. Monte Carlo
4. Greedy
5. Divide and Conquer
6. Backtrack

We will take each of these in turn and rework each algorithm to highlight the methods common to all problems, leading to an abstract base class, Problem.

1) Dynamic Algorithm Design Pattern

NIST DADS defines dynamic programming as "Solve an optimization problem by caching sub-problem solutions

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

(memoization) rather than recomputing them." [1] Our ever so slight change is to change the caching of the solution into two steps through introducing the function ***Seen*** which takes a problem identifier which uniquely identifies the problem, and the subsequent storage of the solution of the problem. ***Seen*** and ***StoreSolution*** are not part of the problem, and are dependent on the implementation as part of the Dynamic pattern (for example, a Bloom filter [1] to implement Seen).

In the pseudo code in Figure 1, we drop the parentheses from the zero-argument methods for readability, and introduce the **MAP** function that applies a function to a set. The *Dynamic* algorithm design pattern can solve problems that have the following methods:

- Problem.ID : A way to map the current problem (state) to a number.
- Problem.EasilySolvable : A way to determine if the current problem is easy to solve.
- Problem.Solve : A mechanism to solve the trivial version of the problem.
- Problem.EnumerateSubproblems : A way to break the problem into sub problems which can be solved. These problems might be solvable by another design pattern provided they can be identified as solvable.
- Problem.Combine : A mechanism to combine sub problem solutions back into a solution to the problem which lead to the sub-problem.

```
Dynamic(Problem) returns a Solution
  IF the Problem.ID has not been seen THEN
    Seen(Problem.ID) ← True
    IF Problem.EasilySolvable THEN
      StoreSolution(Problem.ID,Problem.Solve)
    ELSE
      Collect a set of Subproblems ←
        Problem.EnumerateSubproblems()
      StoreSolution(Problem.ID, Problem.Combine(
                          MAP(Dynamic,Subproblems)))
    ENDIF
  ENDIF
END Dynamic
  RETURN LookupSolution(Problem.ID)
END Dynamic
```

Figure 1 Dynamic Algorithm Design Pattern

2) Las Vegas Algorithm Design Pattern

Again, relying on DADS[1] for the definition of a *Las Vegas* algorithm: "A randomized algorithm that always produces correct results, with the only variation from one run to another being its running time."[1]

A *Las Vegas* random algorithm, see Fig. 2, is characterized by the selection of a random set of actions to be applied to the original problem, a method that refines the problem into a new problem given an action, and a method to check if the problem has been solved. The *Las Vegas* algorithm is the most general random algorithmic design pattern since it makes no requirements on assessment of the generated new problems since the problems may not be comparable.

```
LasVegas(Problem) returns a Solution
  NewProblem is a copy of the Problem
  WHILE NewProblem.Solved is false
    PossibleActions ←
        choose a random set from
        Problem.Actions
    NewProblem is a copy of the Problem
    FOREACH action in PossibleActions DO
      NewProblem ←
          NewProblem.Refine(action)
    ENDFOREACH
  ENDWHILE
  RETURN NewProblem.Solution
END LasVegas
```

Figure 2 Las Vegas Algorithm Design Pattern

3) Monte Carlo Algorithm Design Pattern

DADS refines a *Las Vegas* algorithm to a *Monte Carlo* algorithm through a bound on the error probability. See Figure 3. Similarly, but not equivalent, we fix a runtime bound and make the new problems comparable, and when the allotted time elapses based on the runtime bound, we return the best solution found so far.

Now, the difference between the *Las Vegas* and *Monte Carlo* design patterns is clear - the *Monte Carlo* pattern needs to be able to order instances of the problem, meaning it requires a method which, when given another unique problem, can judge if the current problem is "better". This ordering may not be possible, so we would need to resort to the *Las Vegas* design pattern. To use these new design patterns, we add the following abstract methods to our Problem class:

- Problem.Actions : The available actions that change the problem from one problem (id) to another, for example, exchanging two elements in a list, or placing a queen in a row on an N x N board.
- Problem.Refine(action) : Given the action, returns a new problem.
- Problem.Better(AnotherProblem.ID) : Subject to the goal, compares the problem to another problem.
- Problem.Solved : Returns true if the problem is solved.

```
MonteCarlo(Problem) returns an ApproximateSolution
  Start A Timer
  CurrentProblem is a copy of the Problem
  WHILE CurrentProblem.solved is false or Timer.Expired()
    choose at random an action from CurrentProblem.Actions
    NewProblem is CurrentProblem.Refine(action)
    IF (NewProblem.Better(CurrentProblem.ID) THEN
      CurrentProblem ← NewProblem
    ENDIF
  ENDWHILE
  RETURN CurrentProblem.Solution
END MonteCarlo
```

Figure 3 Monte Carlo Algorithm Design Pattern

4) Greedy Design Algorithm Pattern

The *Greedy* algorithm, Fig. 4, will select the best action given a particular problem. In some cases, this will lead to

an optimal solution as in Dijkstra's algorithm [1] provided the appropriate data structures are selected.

To use the *Greedy* design pattern, we need to be able to select the "best" action to take for a given problem.

We do not need any additional methods to apply the *Divide and Conquer* or the *Backtrack* design patterns, which follow below.

```
Greedy(Problem) returns Solution
   CurrentProblem is a copy of the Problem
   WHILE CurrentProblem.Solved is false
      Action ← CurrentProblem.SelectBestAction(
                              CurrentProblem.Actions)
   CurrentProblem ← CurrentProblem.Refine(Action)
   ENDWHILE
   RETURN CurrentProblem.Solution
END Greedy
```

Figure 4 Greedy Algorithm Design Pattern

5) Divide and Conquer Algorithm Design Pattern

Given a problem, we need to enumerate sub-problems. Subsequently, each of these sub-problems need to be solved as shown in Figure 5.

There is no reason that a single technique is used to solve the sub-problems, the only requirement is that the given sub-problem is easily solvable.

```
DivideAndConquer(Problem) returns SolvedProblem
   IF Problem.EasilySolvable THEN
      SolvedProblem ← Problem.Solve
   ELSE
      SubProblems ← Problem.EnumerateSubproblems
      SolvedProblem ← Problem.Combine(
                MAP(DivideAndConquer,SubProblems))
   ENDIF
   RETURN SolvedProblem
END DivideAndConquer

DivideAndConquer(Problem) returns Solution
   RETURN DivideAndConquer(Problem).Solution
END DivideAndConquer
```

Figure 5 Divide and Conquer Algorithm Design Pattern

6) Backtrack Design Algorithm Pattern

The *Backtrack* algorithm, Figure 6, simply tries all possible actions until a particular sequence solves the problem. If an action leads to an impossible problem, the action is rejected, and the routine continues until all alternatives are exhausted or the problem is solved.

It is helpful to create an *ImpossibleProblem* pattern, one that has no Actions, no Solution, and only sub-problems which are also *ImpossibleProblems*, but not required.

```
BackTrack(Problem) returns
                SolvedProblem or ImpossibleProblem
% An ImpossibleProblem has a Solution which is
                NoSolution
   IF Problem.Actions() is Empty
      RETURN ImpossibleProblem
   ENDIF
   IF Problem.Solved() is true
      RETURN Problem
   ENDIF
   FORALL action in Problem.Actions()
      NewProblem ← Problem.Refine(action)
      CurrentProblem ← BackTrack(NewProblem)
      IF CurrentProblem.Solved is True THEN
         RETURN CurrentProblem
      ENDIF
   ENDFORALL
   RETURN ImpossibleProblem
END BackTrack

BackTrack(Problem) returns Solution or NoSolution
   CurrentProblem ← BackTrack(Problem)
   RETURN CurrentProblem.Solution()
END BackTrack
```

Figure 6 Backtrack Algorithm Design Pattern

V. CLASS - PROBLEM

Examination of the algorithm design patterns leads to the abstract *Problem* class (**Error! Reference source not found.**) which can be added, if required, to a new design pattern.

The construction of the abstract *Problem* class now facilitates the construction of hybrid techniques in a natural way. For example, the *Divide and Conquer* pattern can be decomposed into sub-problems, some of which permit of an easy *Greedy* solution, sub-problems that may only have an easy random approach, and a set of sub-problems which divides and conquers in the conventional sense.

An interesting application of the abstraction is to map the problem of putting a list an unordered set of comparable objects into an ordering. Each of the methods is straight forward to define for the sorting problem. Then the sorting problem can be implemented in each of the patterns leading to a known sorting algorithm, including the randomized sorting algorithm Bogosort [1].

Problem

- ID – An identifier for the problem, might be a UUID based on the current state, or another representation of the problem.
- Solution – can be NoSolution
- Refine(Action) – Returns a New Problem which is the result of applying an action on the Problem
- EasilySolvable – returns true if the problem is easy to solve using any known algorithm.
- SelectBest(Actions) – given a set of actions select the known best choice for the problem. This may not be possible.
- EnumerateSubproblems – return a set of subproblems derived from the problem
- Solved – returns true if the problem is solved
- Actions – returns a set of actions for the problem. This can be very large (for example a list of pairs to exchange to sort a list)
- Combine(SubProblems) merges a set of problems into a single problem.

Figure 7 Abstract Problem Class

VI. Future Work

The plan is to examine other design patterns and refine the Problem abstract class as required. A library of generic algorithms that will solve any problem which implements the abstract problem class will be constructed and released into the public domain.

VII. Conclusion

An abstract Problem class has been constructed which facilitates known algorithm design patterns to the same problem. It also facilitates the abstraction of a given solution back to a set of methods, and then implemented in a new design pattern. By identifying and constructing algorithms for given sets of problems, we focus on the approaches to construct the algorithms, rather than focusing on the implementation of individual algorithms as is commonplace.

References

[1] National Institute of Standards and Technology, Dictionary of Algorithms and Data Structures, *https://xlinux.nist.gov/dads/*

[2] Brassard, G and Bratley, B, *Fundamentals of Algorithmics, 1st Edition*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA 1996

[3] Russell, S. and Norvig, P., *Artificial intelligence: a modern approach*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA 1995