

# Dynamic Recursion Pattern

Gustavo Patow\*<sup>∇</sup>, Fernando Lyardet\*, Kent Beck<sup>°</sup>

\*LIFIA, Fac. De Ciencias Exactas, UNLP, Argentina

<sup>∇</sup>Grup de Grafics de Girona, Institut d'Informatica i Aplicacions, Universitat de Girona, Spain

<sup>°</sup>First Class Software

[dagush@ima.udg.es](mailto:dagush@ima.udg.es)

[fer@sol.info.unlp.edu.ar](mailto:fer@sol.info.unlp.edu.ar)

[kentbeck@csi.com](mailto:kentbeck@csi.com)

---

Dynamic Recursion

Object Behavioral

---

## Intent

Factorize a recursive problem in specific classes. Allow decomposing the recursive problem and isolating it from the underlying data structure.

## Motivation

Everybody knows what recursion is and how it should be implemented in a procedural language. The first analysis of recursion implementations is due Kent Beck [Beck92], but no formalization was given. The other work on an OOP implementation of the subject is due to Boby Wolf [Wolf97], but his work only deals with one aspect of the problem: Structural Recursion. Structural Recursion is a particular aspect of general recursion where the recursion is performed by a set of objects (a list or a tree, for example) that is already present in the computer memory. All the participating objects must exist *previously* the recursion starts.

As opposite to Structural Recursion, Dynamic Recursion does not put such a strong requirement on the way the recursion is performed. This is done simply by delaying the creation of the recursive elements until they are actually needed, and thus, delaying the decision of finishing the recursion until its very end. Examples of recursion that can *only* be handled with this pattern are the famous eight-rook problem [HVC91] or even a simple airplane-routing problem. Another important point to be taken into account is that implementing recursive behavior *into* an object is more or less the same as forcing it to use a certain built-in iterator instead of using the iterator pattern [GoF].

Let's start by a simple, widely known example: sorting an array by the well known MergeSort Strategy (the analysis would be identical for the QuickSort strategy)[Baa91]. The traditional way to do so in procedural programming can be translated as is to OOP by creating a new message `sort:` in a MergeSort class:

```
MergeSort >> sort:anArray from:lowerIndex to:upperIndex
| split |
(lowerIndex < upperIndex)
  ifTrue:[split = (lowerIndex + upperIndex) // 2.
    self sort:anArray from:lowerIndex to:split.
    self sort:anArray from:split+1 to:upperIndex.
    self merge:anArray from:lowerIndex to:split
      andFrom:split+1 to:upperIndex.].
```

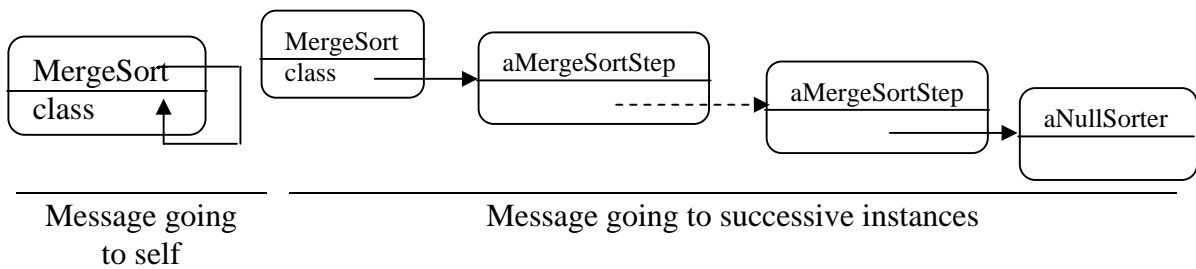
Here, we assume that a message `merge:from:to:andFrom:to:` exists in

MergeSort and performs the merging of the two lists, one in the range [lowerIndex,split] and the other one in [split+1,upperIndex].

Perhaps these eight lines of code would look fine. But if we are implementing a more complex application (like the airline routing problem described below, or a recursive Computer Graphics Ray Tracer [With81]) where recursion is not so obvious, it would be a good design choice to isolate the different behaviors in separate methods. Those methods would be one for the recursive case and one for the situation when recursion ends (in this case there is nothing to do when recursion finishes, but in general this is not the case). Although this would lead to have two different methods, computeRecursiveCase and computeFinishingCase, this wouldn't be a good decision, since each of them could have very complex behaviors and that would lead to several methods for each case.

One clue to solve this problem comes from carefully inspecting the code above: if we look carefully, we will note that the recursion needs not to be followed by the same instance, but by the *next* instance. That is, Two different instances should be in charge of continuing the recursion depending on which case we are in, the finishing one or the recursive one. Then, although silly for this simple example, we can factorize the different behaviors in two different classes: one for the recursive case, called MergeSortStep, and the other for the finalization of the recursion, the NullSorter, both descendants of the MergeSort superclass.

An instance diagram can be drawn comparing both solutions, the one with the message going to self and the other with the message going to successive instances.



Since the decision of which case (finishing or recursive) should follow implies now an instance creation, we should add it as a class message:

```

Array>>sort
  MergeSorter sort: self

MergeSorter class>>sort: anArray
  self
  sort: anArray
  from: 1
  to: anArray size

MergeSorter class>>sort: anArray from: fromInteger to: toInteger
  | sorter |
  sorter := fromInteger >= toInteger
  ifTrue: [NullSorter new]
  ifFalse:
    [MergeSortStep new
     setCollection: anArray
     from: fromInteger
     to: toInteger].
  sorter sort

```

In this implementation, all the information about the recursion (the collection and the two indexes) is stored as instance variables of the `MergeSortStep`, being the recursive message just `sort`. The `NullMergeSorter>>sort` message just performs nothing:

```
NullSorter >> sort
  "Performs the actions of the base case of the algorithm: nothing!"
```

and in the `MergeSortStep`:

```
MergeSortStep>>sort
| split |
split := splitPoint.
self
  sortFrom: from
  to: split.
self
  sortFrom: split
  to: to.
self merge: split
```

This time, the method `merge:` merges two parts of the same array at each side of the `split` point, and the method `sortFrom:to:` is used to call the `sort:From:to:` method of the `MergeSort` Class. The message `splitPoint` returns  $(from+to)//2$ .

We can see that this method results in the `MergeSort` class having two important class methods: the one that triggers the recursion and the Factory Method that decides which instance should follow the recursion. The `MergeSortStep` class stores the behaviour for the recursive case while the `NullSorter` class encompasses the base case behaviour. If the termination criterion is able to change or is too difficult to isolate in a simple expression, it could be factorised into a `TerminationCriteria` class. The `TerminationCriteria` has only as relevant method the one that determines if the recursion should stop or not, based on the recursion parameters passed to it.

## Applicability

Whenever a recursion problem appears, this pattern should be applied to get the desired degree of flexibility and modularization, factoring the different behaviors to different classes:

- 1) It must be possible to decompose the original problem into simpler instances of the same problem.
- 2) Once each of these simpler sub-problems has been solved, it must be possible to combine these solutions to produce a solution to the original problem.
- 3) As the large problem is broken down into successively less complex ones, those sub-problems must eventually become so simple that they can be solved without further subdivision.

## Structure

The user starts the Dynamic Recursion Pattern by sending a `performRecursion:` message to the `Step` class. The message is invoked with the desired parameters for the recursion and it will return the desired final value, after performing the whole process. The method `performRecursion:` of the `Step` Class, when called, creates the necessary recursive element to continue the recursion in the proper way.



If we need to handle more than one type of recursion for a given Step, like all the possible variants in Ray Tracing or Monte Carlo Path Tracing [Laf096], we should let each `recursiveStep` create new instances of its same class. Passing the previous Step as one of the recursion parameters can help us to do this, and using this instance as done in the Prototype Pattern. This method can be invoked through the old step in the `ifTrue:` part of the `performRecursion:` message in the Step Class as (this time using a `terminationCriteria` object to decide if recursion finishes)

```
Step Class >> performRecursion:recursiveParams with:oldStep
| nextStep |
nextStep := (terminationCriteria
             shouldTerminate: recursiveParams)
             ifFalse: [ FinishingStep new.]
             ifTrue: [ oldStep class new: recursiveParams.]
^nextStep performRecursion.
```

or, if is not possible to use the `class` message, like in C++, the following code could be used:

```
RecursionResult*
Step::performRecursion(RecursiveParams* data,
                      RecursiveStep* oldStep)
{
    RecursiveStep nextStep*;
    if (terminationCriteria->shouldTerminate(data))
        nextStep = new FinishingStep(data);
    else
        nextStep = oldStep->newRecursiveInstance(data);
    return nextStep->performRecursion();
}
```

The message `newRecursiveInstance(RecursiveParams*)` needs to be re-implemented for each subclass of `RecursiveStep`, in a way such that only two messages are implemented for each subclass of the Step abstract class: `newRecursiveInstance(RecursiveParams*)` and the message that performs the recursive contribution of this step, `performRecursion()` !

## Participants

- **Step:** this is an abstract class that only holds as class variable the termination criteria for the current recursion. Creates the instances corresponding to the other two classes and calls the method that continues the recursion.
- **FinishingStep:** this is the one in charge of performing the actions related with the termination of the recursion (e.g.: for the factorial case, it returns 1)
- **RecursiveStep:** this is the one that performs the real recursion.

## Consequences

- Allows a good modularity and factorization into classes of the behaviors encountered in a recursive solution. The correct redistribution of responsibilities makes the code easier to maintain and extend.
- Makes classical recursive code a bit more difficult to read, since the recursive behavior is spread over three different but related classes.
- Dynamic Recursions increases the number of objects in an application. When the recursion is performed on a static data structure and there is no chance to change the recursion behavior (strategy) in the future, is better to use Structural Recursion.

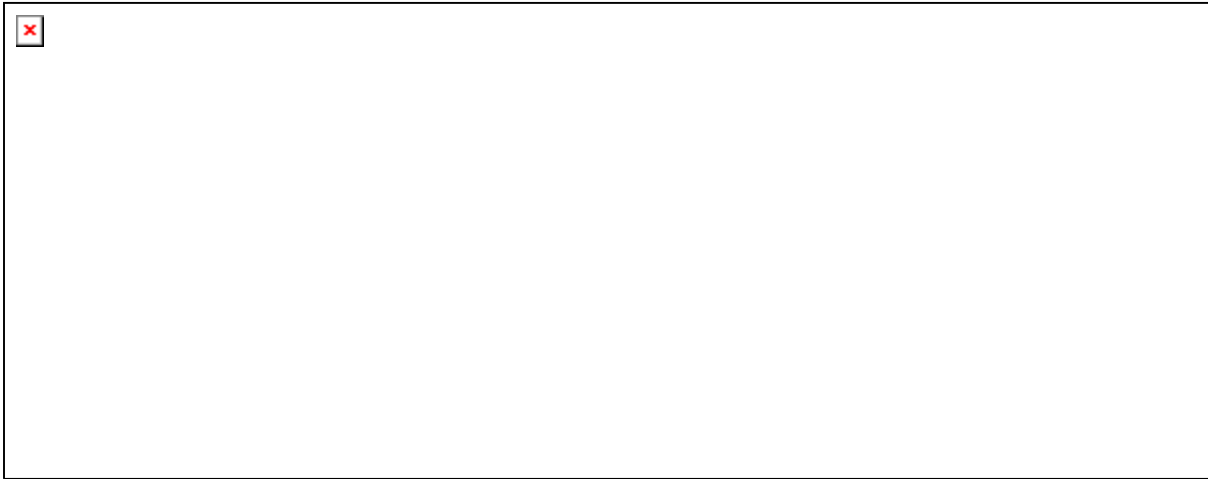
## Implementation

A general recipe for "translating" classical recursion to the Dynamic Recursion Pattern could be stated as

1. Determine and differentiate the recursive part and the base case. Those will be the two basic subclasses of an abstract class we will call "Step", and call the main method that implements the recursive (or base) behaviour `performRecursion`.
2. Determine the condition for terminating the recursion. Isolate this decision in the `performRecursion:` message in the `Step` class.
3. As a result of this decision, inside the `performRecursion:` method an instance of either `RecursiveStep` or `FinishingStep` is created. Call the `performRecursion` method of the created instance. Sometimes it is possible to use an instance method to give us the next iteration.
4. If we have common behaviour for both recursive classes, create a Template Pattern with the common code at the `Step` class and the differences coded at each recursive method.

## Known Uses

- Let's look at a quite more complicated example: the classic Eight Queens problem. The classic algorithm can be found at any standard textbook on algorithms, for example, see (Wirth76). We assume knowledge by the reader of this classical recursive example, so we can see that the corresponding class diagram would be



In this case, since there is a common behaviour between the recursive and the finishing step, we should extract the common code, giving as result that the recursive methods are used as a Template Pattern:

```
Queen >> tryOn:aQueenBoard
| row found |
row := 1.
found := false.
[(found not) and: [row <= 8]] whileTrue:
  [(aQueenBoard isSafeAt: self column with:row)
   ifTrue:[
    self putAt: row on: aQueenBoard.
    found := Queen continueOn: aQueenBoard
              at: self column.
    (found not) ifTrue:[self removeOn: aQueenBoard]].
   row := row + 1].
^found.
```

The message `isSafeAt:with:` of the `QueenBoard` class answers if it safe to put the Queen at that position, that is, if it is not menaced by another, previously placed queen.

- Now, we can face another complex classical recursive problem: The airline routing problem. This problem is the one of finding for a requiring passenger the shortest (or the cheapest, or all existing, or anything else) route from a given departure city to another destination city. The cities are represented as nodes in a graph and an arc connects two cities if there is a flight that joints them. In our design of the solution, the nodes are stored in the `City` class objects, which contain the city name and the collection of all departing flights from this point (a typical graph representation). The `Flight` class (the arcs of our graph) contains information about the departure and destination city of this flight, and the collection of the airlines flights that provides this service.

This problem is a classical recursive one, where backtracking is fired by the encounter of the destination city, or the situation where no new city is reachable from the current city. The application of the Dynamic Recursion Pattern to this problem is also a straightforward one, the key to find the solution is the definition of the `Plane` class, with its two subclasses: `FlyingPlane` (Recursive case) and `LandedPlane` (Finishing one). The class diagram for this solution would be



Creating a delegation mechanism that moves the creation decision from its encapsulation in the message factory can further enhance this scheme. Creating a wrapper around the city nodes can do this by the usage of the Decorator Pattern. Those decorators can be of two types: `TransitCity` and `DestinationCity`. The first one receives the `newPlane` message and returns a `FlyingPlane` as answer, while the `DestinationCity` returns a `LandedPlane`.

- The Monte Carlo Path Tracer developed for the European project SIMULGEN, developed on top of SIR, the rendering architecture of the University of Girona, Spain, relies completely on the structure just described. See the web page at <http://w3imagis.imag.fr/SIMULGEN/SIMULGEN.html> for more information on this project.

## Related Patterns

**Rondabout:** this pattern can be used for the style and writing of the different components of the Dynamic Recursion Pattern, ameliorating its legibility.

**Factory Method:** Use Factory Method for the Step Class creation message.

**Visitor:** Dynamic Recursion can be used in combination with the Visitor Pattern to handle hierarchical, non-homogeneous data structures: An Iterator can visit the objects in the structure by calling their operations, but can not handle different type of objects. Instead, the traversing of the structure can be left to Dynamic Recursion and the elements of the data structure can be safely visited with the use of Visitor, as before.

**Strategy:** Also, as seen in the introductory example, Dynamic Recursion can be used in combination with the Strategy Pattern in order to allow different algorithm strategies to be implemented on a given static data structure, instead of polluting the data structure classes with different versions of the Structural Recursion Pattern.

Structural Recursion: When the recursion is performed on an existing data structure, and there is no need to create different recursion strategies, Structural Recursion might be a better implementation choice.

## Acknowledgements

We are indebted to Kent Beck, our Shepherd, since his valuable comments greatly improved the quality and content of this paper. We also wish to thank our respective labs for their support and encouragement during the maturation and writing of this paper.

## Bibliography

[Baa91]	Baase, Sara, "Computer Algorithms. Introduction to Design and Analysis", 2nd ed., Addison-Wesley, Re. 1991
[Beck92]	Beck, Kent; " ... ", Smalltalk Report.
[GOF]	E. Gamma, R. Helm, R. Johnson and J. Vlissides: "Design Patterns: elements of reusable object-oriented software". Addison Wesley, 1995.
[HVC91]	Paul Helman, Robert Veroff, Frank M. Carrano , "Intermediate problem solving and data structures : Walls and mirrors", 2nd ed, Redwood City, Calif. : The benjamin/Cummings, 1991.
[With80]	Whithed, Turner "An Improved Illumination Model For Shaded Display" CACM, Vol 23, No. 6, june 1980, pp 343-349.
[Wirth76]	Wirth, N "Algorithms + Data Structures = Programs", Prentice-Hall, 1976.
[Wolf97]	Bobby Woolf. The Object Recursion Pattern. 5 <sup>th</sup> annual conference on Pattern Languages of Programming. Monticello, Illinois. TR #WUCS-98-25