Comparing the template method and strategy design patterns in a genetic algorithm application

Article *in* ACM SIGCSE Bulletin · December 2002

DOI: 10.1145/820127.820171 · Source: DBLP

2 authors:

Michael R. Wick
University of Wisconsin - Eau Claire
50 PUBLICATIONS **587** CITATIONS
SEE PROFILE

Andrew T. Phillips
United States Naval Academy
51 PUBLICATIONS **575** CITATIONS
SEE PROFILE

Comparing the Template Method and Strategy Design Patterns in a Genetic Algorithm Application

Michael R. Wick and Andrew T. Phillips
Computer Science Department
University of Wisconsin-Eau Claire
Eau Claire, WI 54701
{wickmr@uwec.edu, phillipa@uwec.edu}

Abstract

We present a genetic algorithm software project that serves to give students direct experience with choosing among multiple potentially applicable design patterns. This project has been carefully constructed to illustrate the power of design patterns in supporting encapsulation while at the same time providing a single context in which to compare and contrast similar design pattern alternatives.

1 Introduction

Design patterns [1] are becoming ubiquitous within computer science education. And while not all educators agree on the proper location of design patterns in the computer science curriculum, nearly all educators agree that these generalized solutions to commonly recurring design problems represent a wealth of knowledge to which the undergraduate student should be exposed. We agree, but we also believe that no matter where in the curriculum design patterns are introduced, we must not simply present the patterns as a collection of tools, but we must also give the students hands-on experience with determining when and why each pattern is an appropriate design choice. Because students typically struggle with selecting the correct design pattern for a given problem, they typically either fail to see the applicability of a particular pattern, or they blindly apply their favorite pattern to problems for which it is not appropriate.

As educators, we must focus our design pattern materials on applications that clearly illustrate to students the appropriate context in which to apply each pattern. This task is made even more difficult (but even more important) by the sometimes vague and overlapping applicability criteria presented in classic design pattern texts. In such cases, an ideal solution is to find a single application that clearly and unambiguously highlights the applicability criteria for each selected pattern, and that also effectively demonstrates the power that each distinct pattern provides.

In this paper, we present such an application. We present a single software design project that helps clarify the distinction between two of the most fundamental design patterns – Template Method and Strategy. In so doing, the application not only correctly illustrates when and why one design pattern is more appropriate than the other, but also effectively illustrates the power of both patterns in achieving a dynamic, robust, and flexible design.

2 Motivation

Encapsulation is one of the most fundamental design principles that we teach students. This philosophy of isolating design decisions from one another is so instrumental in design that it occurs (or should occur) in nearly every undergraduate course in a modern computer science curriculum. It is both this ubiquitous presence of encapsulation and the support of encapsulation provided by the Template Method and Strategy design patterns that warrant our claim that the two patterns are two of the most important design patterns in current use.

Of particular importance to us is the application of encapsulation to support multiple alternative implementations (we call this the *variant behavior problem*). When a design problem has multiple possible solutions, each with an associated cost and benefit, encapsulating separate design choices into separate class implementations isolates the impact of subsequent changes to the implementation and allows alternative implementations to be provided for use within varying application contexts.

Both the Template Method and the Strategy design patterns provide structural models for solving the variant behavior problem. There is, of course, a fundamental distinction between them. To allow for alternative implementations, the Template Method pattern uses inheritance, while the Strategy pattern uses delegation. And the subtle implications of these two differing approaches can dramatically affect the quality of the resulting design.

Given our premise that these patterns are of particular importance in that they support the fundamental design principle of encapsulation, we believe it is imperative that students be given an in-depth understanding of how to select the appropriate pattern. We have found that the implementation of a genetic algorithm simulator is an effective project for demonstrating this distinction to students. The genetic algorithm application is interesting and enjoyable, and it demands the full encapsulating power of both the Template Method and Strategy design patterns.

3 Background

For completeness, this section includes a brief introduction to genetic algorithms and the Template Method and Strategy design patterns.

3.1 Genetic Algorithms

A genetic algorithm is a search procedure based on the principles governing natural selection in a "survival of the fittest" biological world. John Holland did the original work on genetic algorithms in 1975 at the University of Michigan [3]. His work, and the work of his successors [2], focused attention on modeling the natural world in an attempt to provide an algorithmic abstraction of the adaptive process of natural systems.

In a genetic algorithm, each potential solution to the problem is denoted by the term *chromosome*, which represents the encoded values of each of the variables of the problem. At the start of the genetic algorithm process, a collection, or *initial generation*, of randomly constructed chromosomes (potential solutions) is created. The genetic algorithm then follows a three-step process to transform the current generation of solutions to the next generation of solutions. This process provides a model of the natural world in which the only operators required for transforming solutions from one generation to the next are reproduction, crossover, and mutation.

The *reproduction* step selects from the current generation of chromosomes those that will survive to the next generation. This selection uses a probabilistic survival of the fittest mechanism based on a problem-specific evaluation of the chromosomes (modeling the concept of "fitness" from natural selection). This probabilistic survival mechanism is what directs the generations, over time, toward the most valuable (fit) chromosomes.

The *crossover* step then allows the introduction of new chromosomes into the population of potential solutions by randomly combining pairs of randomly selected existing chromosomes. This step simulates the crossover phenomena in biological reproduction where two chromosomes mate by splitting into randomly selected subsections and then rejoining to form two new chromosomes. The key point here is that crossover allows new chromosomes not present in the current generation to be added to the next generation.

Finally, the *mutation* step allows the random mutation of existing chromosomes so that new chromosomes may contain parts not found in any existing chromosomes. Again, this step simulates the natural phenomena of gene mutation during reproduction. In the genetic algorithm process, mutation allows the introduction of new solution elements by permitting the occasional random alteration of a single element of the chromosome encoding.

This three-step process is repeated, moving from generation to generation, until a generation is found that is "good enough". There are many ways to determine when a generation is good enough, for example by using the "value" of the best chromosome in the population or by simply iterating for a fixed number of generations. When the process terminates, the best chromosome selected from among the final generation is the solution to the problem. The following also presents a summary of this genetic algorithm process.

```
compute the initial generation
while (more generations) {
     reproduce the current generation
     crossover the current generation
     mutate the current generation
     increment to the next generation
}
report the "best" solution in the generation
```

This genetic algorithm process has been shown to produce near optimal solutions to a wide variety of difficult optimization problems [2].

3.2 The Template Method Pattern

The Template Method pattern deals with the proper division of responsibility between parent and child classes in an inheritance hierarchy. The Template Method pattern solves to the variant behavior problem by allowing each child class to specify variant aspects of the object's behavior while having the parent class define the invariant behavior that is common to all objects of this type. This is typically realized by providing an abstract parent class with a method that defines the invariant steps that will be taken by all objects in response to a particular request. However, each step within this method is defined as an abstract method to be specified by each child class in order to obtain the various desired behavioral options. The UML diagram in Figure 1 highlights this structure.

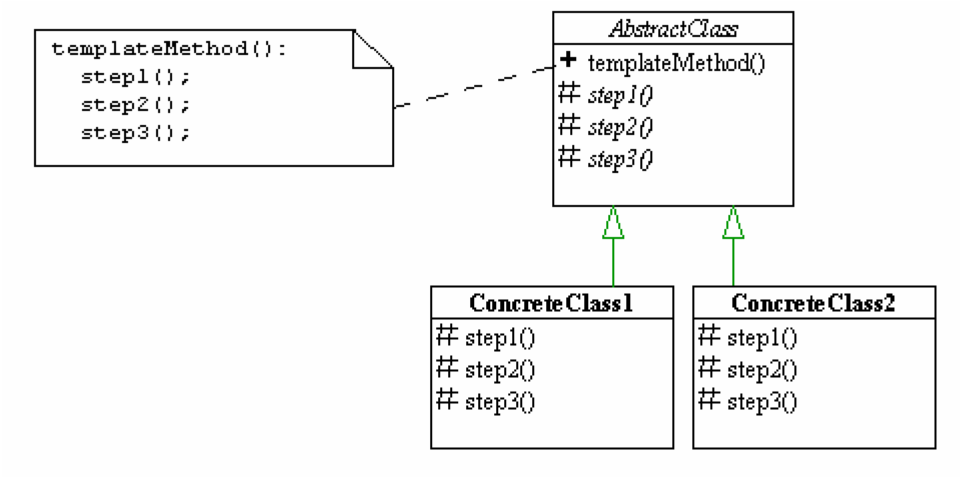


Figure 1: The Template Method Pattern

This structure supports behavioral variations of the resulting object by allowing each desired behavior to be defined in a separate child class.

3.3 The Strategy Pattern

The Strategy pattern also provides a solution to the variant behavior problem, but it does so by delegating the definition of each variant step of a process to a class in a *separate inheritance hierarchy*. This is typically realized again by providing the original class (called the "context class" in this pattern) with a method that defines the invariant behavior. However, rather than having a derived class provide the variant behavior, a separate object outside of the inheritance hierarchy is created and is held responsible for providing that behavior. The UML diagram in Figure 2 highlights this structure.

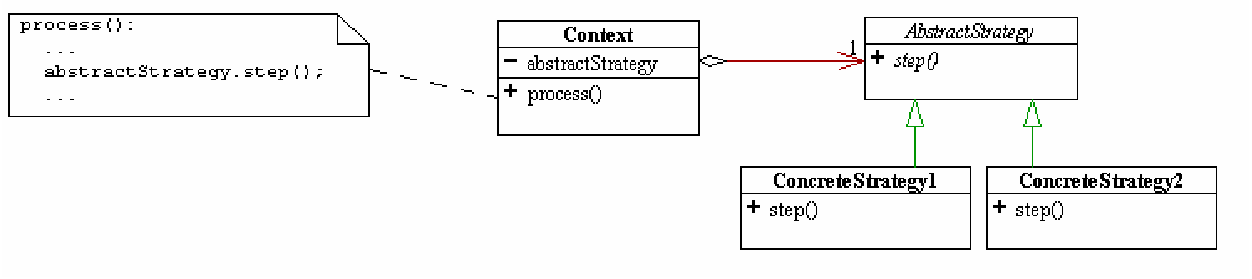


Figure 2: The Strategy Pattern

As previously stated, this design pattern also supports alternative behaviors, but this time by using delegation rather than inheritance.

4 Template Method versus Strategy

It is not difficult to see why students are confused over which of these patterns to select to achieve the desired variant behavior. The patterns are structurally similar and both support equally, at the highest level, the implementation of alternative behaviors. In fact, we have found that when left to their own accord, undergraduate students typically pick the pattern with which they are most comfortable and simply apply it any time they need to solve the variant behavior problem. Unfortunately, depending on the requirements of the application, the resulting design can be poor enough to offset the value of providing the variant behavior itself. The real problem is that it is not obvious to students that there are clear and specific reasons for choosing between the Template Method and Strategy patterns. In particular, the Strategy pattern should be selected when either of the following is true:

- ➢ The variant behaviors are applicable to a large class of differing client applications, and client data is typically accessed only through a general interface. In this situation, the variant behaviors can be reused in other unrelated applications that require the same variant behavior.
- ➢ The client application requires multiple variant methods (i.e. strategies), whose implementations do not require a maintained data representation that is shared between them.

On the other hand, the Template Method pattern should be selected when either of the following is true instead:

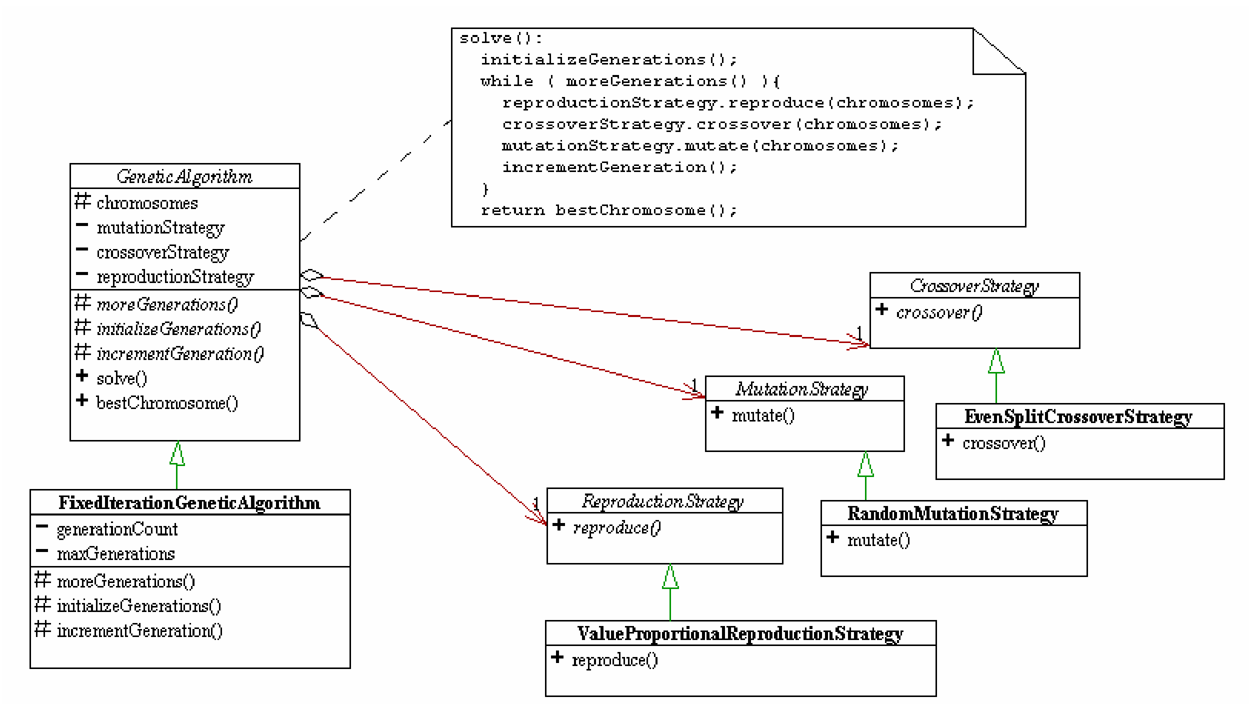
- ➢ The set of possible variant behaviors is specific to the client application, typically signaled by requiring unrestricted access to the client data.
- ➢ The client application requires multiple <u>interrelated</u> variant methods, frequently with a maintained shared data representation.

While these rules are short and simple, conveying to students a true understanding of their motivation and implication is difficult. To be effective, each student must experience software design projects that illustrate the appropriate application of the rules. In the following sections, we present a genetic algorithm application that uses these rules to identify the appropriate but separate application of <u>both</u> the Template Pattern and the Strategy Pattern in a single project.

5 The Genetic Algorithm Application

In this section we present the design of a genetic algorithm framework that effectively uses both the Template Method and Strategy design patterns. The genetic algorithm framework has the added benefit of providing a natural context in which to support a collection of variant behavioral definitions, each with specific implications on the overall effectiveness of the framework for solving particular problems.

Our genetic algorithm application is a framework (hereafter called the GAF – Genetic Algorithm Framework) that can be used to solve a wide variety of optimization problems. Like most frameworks, the GAF is domain-independent, providing hooks for the domain-dependent aspects of each application. The GAF provides the code for maintaining the collection of chromosomes and for performing the standard reproduce, crossover, and mutate operations on that collection. But it leaves the specific definition, or encoding, of the chromosome itself to the user of the framework, since the chromosome represents the domain-dependent problem representation manipulated by the GAF. A simplified UML diagram for the overall design is shown in Figure 3. In the next section we discuss this design in detail.



```
solve():
  initializeGenerations();
  while ( moreGenerations() ){
    reproductionStrategy.reproduce(chromosomes);
    crossoverStrategy.crossover(chromosomes);
    mutationStrategy.mutate(chromosomes);
    incrementGeneration();
  }
  return bestChromosome();
```

Figure 3: The Genetic Algorithm Framework

5.1 The Genetic Algorithm Class Model

While there are many supporting classes (like iterators, containers, etc), the core of the GAF consists of an integrated collection of four abstract classes (GeneticAlgorithm, Reproduce, Crossover, Mutate) and one interface (Chromosome) for connection to the application domain.

GeneticAlgorithm is an abstract class that represents the heart of the genetic algorithm process. This class uses the Template Method pattern to define the standard three-step sequence required of a genetic algorithm, while leaving the details of algorithm initialization, iteration, and termination to a choice of derived and concrete subclasses. The lower left side of Figure 3 illustrates how we elected to implement this variant behavior by using a concrete FixedIterationGeneticAlgorithm subclass of GeneticAlgorithm.

Note that the abstract methods initializeGenerations, incrementGeneration, and moreGenerations in GeneticAlgorithm represent the template steps in the genetic algorithm process. These three methods together handle the transformation of chromosomes from one generation to the next. The Template Method pattern is appropriate in this context since the definitions of these three operations are highly interrelated, frequently sharing at least one common data structure. For example, one possible implementation choice is to select a FixedIterationGeneticAlgorithm subclass that uses a fixed generation count to determine when to stop the genetic evolution process. In this case, FixedIterationGeneticAlgorithm defines an integer count that is set to zero by initializeGenerations, is incremented by incrementGeneration, and is compared to some fixed maximum in moreGenerations. Of course, other approaches could be used instead. For instance, the implementation of a StabilityGeneticAlgorithm subclass could use a population measure based on "stability" to stop the genetic algorithm when *m* successive generations produce equal-valued results. Notice how this use of the Template Method pattern supports the required variant behavior natural to the application. The user of the GAF can tailor the generation maintenance implementation to fit the specific requirements of the application domain.

On the other hand, each of the reproduce, crossover, and mutate steps of the genetic algorithm process are implemented using the Strategy pattern (see the right side of Figure 3). For brevity, we describe only the reproduce step here, since the crossover and mutate steps are analogous. The Reproduce class serves as the abstract class from which all concrete reproduction strategies are derived.

The GeneticAlgorithm class invokes a concrete reproduction strategy by delegating this responsibility to the reproduce method of a ReproductionStrategy instance variable (local to the GeneticAlgorithm class). In this case, the abstract class ReproductionStrategy provides the hook in the genetic algorithm process for any specific reproduction strategy desired. One particular strategy that can be implemented is the "value proportional" strategy in which chromosomes survive to the next generation with a probability equal to their value divided by the value of the best chromosome in the current generation. Of course, many other concrete strategies would be possible here as well. The solve method of the GeneticAlgorithm class delegates the act of reproduction to the concrete class using dynamic typing to invoke the appropriate subclass reproduce method. Similarly, abstract classes for CrossoverStrategy and for MutationStrategy provide the hooks in the genetic algorithm process required to implement the other two steps of the genetic algorithm process.

The use of the Strategy pattern for the reproduce, crossover, and mutate steps allows the user to easily experiment with any combination of strategies for these steps. Because each strategy is isolated in its own class and doesn't share any maintained data structures, the strategies are independent and can be mixed and matched as appropriate. This is in direct contrast to the generation-manipulation methods that are inherently dependent on one another through a shared and maintained data structure. Using the pattern selection rules from section 4, the reproduce, crossover, and mutate steps call for the Strategy Pattern while the generation initialization, increment, and termination steps motivate the use the Template Method pattern.

Notice that all of the code of the GAF presented so far is domain-independent in that it doesn't depend on the particular problem being solved. The Chromosome interface provides this problem specific connection, and the entire GAF is written to work with any domain-specific class that adheres to the Chromosome interface. The Chromosome interface itself is very simple. It requires a crossoverWith method that defines how two chromosomes crossover, a mutate method that defines how a chromosome mutates, and a valueOf method that defines the "value" of a specific chromosome. Notice how the framework itself does not impose a particular problem representation or encoding on the application; the application can be represented in whatever design is appropriate as long as the design supports those three required manipulation methods.

6 Role in the Curriculum

The implementation of a genetic algorithm framework is a substantive project for undergraduate students at any level. However, we believe that this application could appropriately be used in at least two places. First, and most obviously, the genetic algorithm project could be given as a 2-3 week programming assignment within a senior-level design patterns course. Arguably, this is the point at which a student is most likely to comprehend the subtle and important distinctions between these two patterns. Second, we believe the genetic algorithm application is appropriate as a program-in-progress style project for a freshman-level introduction to programming course. If developed as part of a lecture or closed-laboratory sequence, even such inexperienced programmers could be asked to implement portions of the application giving them early hands-on experience with the competing options of inheritance versus delegation.

At UW-Eau Claire, we have used the application in both contexts. The application was originally developed in an advanced course on design patterns and later adapted for use in our CS1 course.

7 Summary

We have presented a programming application that serves to gives students hands-on experience with making the appropriate selection between two alternative design patterns, the Template Method and Strategy patterns. These design patterns are of particular importance as they directly support the fundamental design concept of encapsulation.

Complete Java code for the genetic algorithm application is available from the authors.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley Publishing (1995).

[2] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley Publishing (1989).

[3] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor (1975).