

## Incremental Development Using Object Oriented Frameworks: A Case Study

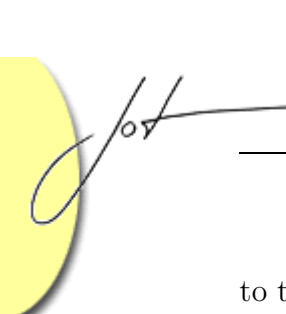
**Jason Hallstrom and Neelam Soundarajan**

Computer and Information Science, Ohio State University, Columbus, OH 43210

The *object oriented framework-based* approach is one of the most powerful approaches to incremental development. In this paper we report on our experiences in applying the framework-based approach to the domain of *genetic algorithms*. Although different genetic algorithm (GA) systems have much in common with each other, each such system is typically designed and implemented from scratch. Our goal was to design an OO framework that included those aspects that are common to different GA systems, with individual GA systems being implemented as applications on top of this framework. We show how such an incremental approach to implementing GA systems not only leads to code reuse but also to reuse of the effort involved in reasoning about the behavior of such systems. We also discuss how the XML-documentation mechanism of C# can be used to generate precise documentation for frameworks.

### 1 INTRODUCTION

An object-oriented (OO) *framework* [4] is essentially a set of abstract and concrete classes that collaborate in a precise manner to provide a common framework on which a range of applications can be built. The collaboration is in the form of key methods, referred to as *template methods* in the design patterns literature [6], that direct the flow-of-control and call appropriate methods, these being the *hook methods*, of various classes. In order to implement a complete *application* on such a framework, the application developer provides definitions for the hook methods that implement behavior appropriate to the particular application; a different developer could produce a different application by providing an alternate set of definitions for the hook methods. Thus the framework implements behavior that is common to the various applications that may be built on it, and the individual application enriches the common behavior as appropriate to the particular application by defining the hook methods appropriately. If the framework provides the right hooks, an entire new application can be developed with just the incremental effort involved in designing a new set of definitions for the hook methods.



In this paper we report on our experiences in using the framework-based approach to the domain of *genetic algorithms* [9, 14, 15]. Genetic algorithms (GA) have been developed for a variety of domains [3, 7, 10, 16]. Although different GA systems have much in common with each other, indeed it is this commonality that characterizes them as GA systems, typically each such system is designed and implemented from scratch. Our goal was to design an OO framework that included the common aspects of GA systems, with individual GA systems being implemented as applications on this framework. But mere code reuse is not the main point of frameworks. More important is the reuse of the common behavior embedded in the framework's design. Or to put it somewhat differently, the approach makes it possible to reuse, in understanding the behavior of each application built on the framework, the effort that has gone into understanding the behavior built into the framework. Therefore, the second important aspect of our work was to reason about our GA framework, and see how the resulting specifications can be reused when understanding the individual applications built on the framework.

The main contributions of this paper may be summarized as follows:

- It presents a GA framework as a case-study in the use of the framework-based approach; and presents a simple example GA system built on this framework.
- It shows how the framework based approach makes it possible not only to build related applications in an incremental manner, but also to specify and understand the behavior of such applications incrementally.
- It reports on our experiences with using the *XML-documentation* mechanisms of the *C#* compiler to provide precise documentation for frameworks.

We should stress that with respect to genetic algorithms, our paper does not break any new ground; the GA we build as an 'application' on our framework is fairly simple. However, a GA-expert who uses our framework should be able to experiment with new genetic algorithms easily since the framework will allow him or her to focus on whatever is new or unique to the algorithm in question and ignore those aspects that are common to all GAs. Indeed, that is precisely the point of using a framework.

The paper is organized as follows: The next section provides background on GAs and summarizes the main issues in reasoning about frameworks and applications. Section 3 presents the design of our framework; the key classes are described along with specifications of the most interesting methods; and we discuss the type of precise documentation we use for the framework. Section 4 presents a simple GA application built on our framework, and shows how we can reason about its behavior incrementally from the behavior of the framework. The final section summarizes our work and provides some pointers to future work.



## 2 BACKGROUND AND MOTIVATION

### Genetic Algorithms

In essence, a genetic algorithm works as follows: The program maintains a *population* of *individuals* where each individual represents a potential solution to the problem at hand. The program goes through a series of iterations. In each iteration, the *fitness* of each individual is evaluated using some appropriate measure and a new population formed by selecting the fitter individuals. Some members of the new population may undergo transformations by means of “*genetic*” operations such as

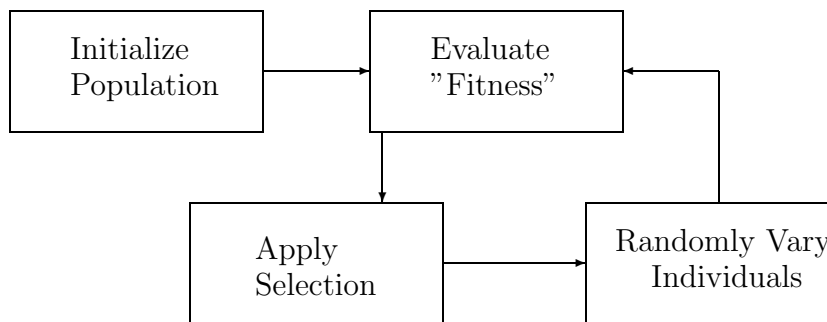


Figure 1: Structure of Genetic Algorithms

*mutation* (a unary transformation) and *crossover* (a binary transformation), giving the population for the next generation; this continues for a number of iterations, after which the program typically converges to a good, or even near-optimal, solution. Because of the similarity to natural selection and evolution in biological systems, individuals are also referred to as *chromosomes*. Pictorially, the structure of such algorithms may be represented [5] as in Figure 1. In [14], Michalewicz presents the code for a genetic algorithm with instructions on which sections of the code have to be replaced to implement other GA systems. As is widely recognized in the OO community, this type of code reuse by “cutting-and-pasting” can lead to serious problems due to unanticipated interactions between the new code and the old code.

One important point not explicit in Fig. 1 is that the selection operation involves the notion of ‘parents’ that are combined to produce ‘children’ who form the new population. This will be clear in our framework. Thus while pictures such as Fig. 1 convey an intuition for the underlying structure, the framework captures it fully and formally. And, of course, it ensures that we do not have to repeatedly code this common structure as part of each GA’s implementation.

### Framework and application behavior

The standard approach to reasoning about an OO system is to use precise specifications in the form of *pre-* and *post-* conditions for each operation of each class *C* of the system. The pre-condition of an operation specifies the condition that must

be satisfied at the time the operation is invoked; it may involve the values of any arguments that the operation receives, as well as the values of any of the attributes of  $C$ . The post-condition similarly tells us what conditions will be satisfied by the values of the arguments and the attributes of  $C$  when the operation finishes. And, in the case of operations that return a result, it also provides information about the result. The pre- and post-condition specifications embody the principle of *Design by Contract (DBC)* [12]: If the client satisfies her part of the contract, that of ensuring the pre-condition of the operation at the time of the call to the operation, then (and only then) the service provider guarantees his part of the contract, that of ensuring the post-condition at the time the operation finishes.

Let us now see how this applies when reasoning about frameworks and applications. Let  $F$  be a framework and  $A$  an application built on it. Suppose  $h()$  is a hook method, i.e., a method of a class  $C$  of the framework and that  $h()$  is redefined in  $A$  in the corresponding derived class  $D$ . As part of the specification of  $F$ , we would have established suitable pre- and post-conditions for<sup>1</sup>  $C.h()$ ; and when reasoning, in  $F$ , about the behavior of any template method  $t()$  that invokes  $h()$ , we would have appealed to this  $F$ -level specification of  $h()$ . In order to ensure that this reasoning about the behavior of  $t()$  remains valid in  $A$ , we must require that  $D.h()$  satisfies this  $F$ -level specification of  $h()$  since, in  $A$ , it is  $D.h()$  that  $t()$  will invoke. In other words, we must require  $D$  as well as other classes in the application  $A$  to be *behavioral subtypes* [11] of their respective base classes.

But this, by itself, is not sufficient. The reason we redefined  $h()$  in  $A$  is so that it will exhibit behavior that is tailored to the needs of  $A$ , rather than just the generic behavior implemented in its framework-level definition. More important, and the whole point of the framework-based approach, is that template methods such as  $t()$  that invoke  $h()$  will also, as a result of the redefinition of  $h()$ , exhibit appropriate application-specific behavior although they (the template methods) themselves are inherited unchanged from the framework. Correspondingly, we need to be able to reason about this application-specific behavior of  $t()$ . And in doing this, we should not have to go back to the code of the framework<sup>2</sup>. To be able to do this, we must include suitable additional information in the  $F$ -level specification of  $t()$  beyond what is normally contained in the pre- and post-conditions of  $t()$ , so that when  $A$  is built, we can arrive at the corresponding  $A$ -specific behavior of  $t()$  by ‘plugging-in’ the behavior of the hook methods as (re-)defined in  $A$ , into this  $F$ -level specification of  $t()$ .

What additional information do we need concerning the behavior of  $t()$  to be able to arrive at its  $A$ -specific behavior? In a typical application, we introduce new attributes in the derived classes defined in the application and redefine the hook methods so that they manipulate these new attributes appropriately. Therefore,

---

<sup>1</sup>We use the standard notation  $C.h()$  to refer to the method  $h()$  defined in  $C$ . Similarly  $D.h()$  will refer to the  $h()$  defined in the derived class  $D$ .

<sup>2</sup>In some cases the framework code may not even be available as for example in the case of a framework that was purchased from a software vendor.



when  $t()$  invokes these hook methods in  $A$ , the execution of  $t()$  will also affect these new attributes, the precise effects being dependent on which particular hook methods it invokes, in what order, how many times, and with what argument values. Hence it is this information about these hook-method invocations that we need to include in the specification of  $t()$ . One possible approach [2, 17] to doing this is in terms of a *trace* or *sequence* that records these invocations. While this works, the resulting specifications are rather unwieldy and complex. So in the next section we will use a more ‘programmer-friendly’ notation to express this information. As we will see, given a specification for  $t()$  that includes this additional information, we will be able, once  $A$  is designed and the richer behaviors of the hook methods specified, to plug these richer behaviors into the  $F$ -specification of  $t()$  to arrive at the  $A$ -specific behavior of  $t()$  without having to reanalyze its code.

### 3 GA FRAMEWORK

Our GA framework consists of four classes, `Chromosome`, `ChromosomePopulation`, `GAFController`, and `GAREporter`. The main hook methods are in the `Chromosome` class. This is to be expected since the key differences between one GA and another are the differences in the behaviors of their respective chromosome types. When building a new GA, most of the work will consist of (re)defining `Chromosome`’s hook methods. `GAFController` contains the key template method, `runGA()`, that implements the evolutionary algorithm common to all GAs. `ChromosomePopulation` is used to represent the evolving population. `GAREporter` allows the framework to report the results of the algorithm; this class’s methods may also be redefined in the application, allowing us to customize the reports for the particular application. We focus on `Chromosome` and `GAFController`<sup>3</sup>.

#### Chromosome class

The `Chromosome` class appears<sup>4</sup> in Fig. 2. `Fitness` is a `get`-only property corresponding to the *fitness* of the individual in question. It is declared `abstract` since the precise details of how the fitness value is computed will obviously depend on the particular type of chromosome and hence will be defined in the application. Similarly, the `Mutate()` operation which (very likely, probabilistically) will modify the chromosome object will also be defined in the application.

---

<sup>3</sup>Due to space limitations, here we will only discuss select portions of our framework and one of the applications we have built using it. Full details, including the complete `C#` code and documentation are available from <http://www.cis.ohio-state.edu/~hallstro/GAF>

<sup>4</sup>In our presentation in this paper, we do not always strictly abide by the style guidelines[1] of `C#`, mostly in the interest of reducing white-space; thus we typically do not put a single “{” on a line by itself, preferring instead to move it to the end of the previous line. In the code that is on the web site, we do obey the guidelines.

```

public abstract class Chromosome {
    public abstract double Fitness
        { get; }

    public abstract void Mutate();

    public abstract ChromosomePair CrossOver(Chromosome cOther);

    public abstract override String ToString();

    public virtual bool AreInOrder(Chromosome cOther)
        { return(true); }
}

```

Figure 2: Chromosome Class

`CrossOver()` is a key operation of GAs involving combining two “parent” chromosomes to obtain two “child” chromosomes; `cOther` is the other chromosome object which will be ‘crossed-over’ with the current chromosome object to obtain the children chromosomes. This too will be defined in the application since how children chromosomes are created from parents depends on the particular GA. `ToString()` returns a representation of the current chromosome to be used by `GARepoter` to print out information about the chromosomes; the string representation of a chromosome object will of course depend on the type of the chromosome, so this also is an abstract operation. The final operation `AreInOrder()` allows us to compare two chromosome objects to see if they are ‘in-order’ based, possibly, on their fitnesses. This operation will also, typically, be defined in the derived class of the application, but we have provided a default definition here which always returns `true`, i.e., declares any two chromosomes to be in-order.

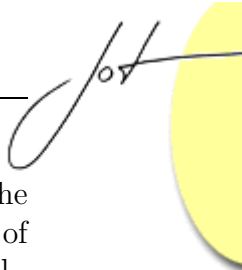
Consider the specification for this class. Consider `Fitness`. Since different applications will likely provide quite different definitions for the associated `get` operation which, for ease of reference in the discussion, we will refer to as `getFitness`, it would seem that the framework-level specification of this method should impose no conditions on it. But there is one condition that does make sense: that it not change the chromosome in any way. In other words, the state of the chromosome after a call to this method should be the same as before the call:

$$post.Chromosome.getFitness \equiv (self = self@pre) \quad (1)$$

In our post-conditions<sup>5</sup>, we use the `@pre` notation of *OCL* [18] to refer to the value of the particular variable at the time the method was invoked. So (1) tells us that `self`, the chromosome’s state<sup>6</sup>, is the same when the method finishes as when it started. Indeed, one could argue that the `get` associated with *any* property of *any* class should meet a requirement such as (1). But it may be unwise to impose such a blanket requirement since there might be situations where it *is* appropriate for

<sup>5</sup>We usually omit pre-conditions if, as in this case, they are trivial, i.e., `true`.

<sup>6</sup>`self` is the *Smalltalk* term for the ‘current’ object; it is the same as `this` in *C#* but `self` seems more descriptive so we use that.



a `get` to change the state of the object. This requires further experience with the notion of properties. One possible compromise might be to require it at the level of the *conceptual* specification of the class but not at the level of the concrete model.

When we define `getFitness` in the application, we will have to show, as required by behavioral subtyping, that it satisfies (1). Thus, although in the framework we have not defined `getFitness`, its framework-level specification gives guidance to the application developer about the conditions that its definition in the application must satisfy if the framework is to function as intended.

Consider next the `CrossOver()` method. This too is abstract since the details of how we ‘crossover’ or combine two parent chromosomes to arrive at two child chromosomes depends on the particular GA and hence will be defined in the application. But at the framework level we can specify some conditions that this method must necessarily satisfy:

$$\begin{aligned}
 \text{pre.Chromosome.CrossOver}(c2) &\equiv (\text{typeof}(self) = \text{typeof}(c2)) \\
 \text{post.Chromosome.CrossOver}(c2) &\equiv \\
 &[(self = self@pre) \wedge (c2 = c2@pre) \wedge \\
 &(\text{typeof}(\text{result}.1) = \text{typeof}(self)) \wedge (\text{typeof}(\text{result}.2) = \text{typeof}(self))] \quad (2)
 \end{aligned}$$

The pre-condition requires that the argument `c2` be the same type as the *self* object. The post-condition asserts that neither the *self* object nor `c2` is modified, and that the (runtime) types of the two results returned are the same as that of *self*. The specifications of the other methods of `Chromosome` are similar and we omit them. In Section 3.3, we will see how the *XML*-documentation mechanism of *C#* can be used to generate documentation for frameworks, consisting of such specifications.

## GAFController class

The `GAFController` class appears in Figure 3. This class, and in particular the `RunGA()` method, is the one that “drives” the entire application. Many frameworks have a similar structure. `currPop` contains the current population. `gaRptr` is used for producing reports at various points during the evolution. `RunGA()` invokes various methods of `GAReporter` at the start and end of the entire run, and at the start and end of each “generation”. By redefining these methods in the application, these reports can be customized to the needs of the particular application. Our conceptual model of `GAFController` consists of a sequence of chromosomes to represent the current population and a conceptual reporter object which will be a sequence of *reports*.

Consider `SelectParents()`. This method, to be defined in the application, is used to choose “parents” from the current population. The parents may be chosen on the basis of fitness or other considerations, depending on the particular GA application. In the specification (3), *cP* denotes the sequence of chromosomes that is the current population and *gAR* denotes the reporter component of the conceptual model.

```

public abstract class GAFController {
    protected ChromosomePopulation currPop; // current population;
    protected GAReporter gaRptr; // object for generating reports;
    public GAFController(GAReporter gaRptr) { this. = gaRptr; }
    // Constructor.

    protected abstract void InitPopulation(); // Initializes currPop;
    protected abstract ChromosomePair SelectParents();
        // Returns a pair of chromosomes from currPop;
        // must not alter currPop;

    protected virtual void SetNewPopulation(
        ChromosomePopulation cChildren) { currPop = cChildren; }
    // Sets currPop to the next generation.

    final public void RunGA(int nn); // Should NOT be overridden.
    /* nn is number of generations the "evolution"
    should run for.
    RunGA() initializes currPop by calling
    InitPopulation(). Then goes through nn iterations
    of "evolution". In each iteration, it replaces
    the current population with a new population,
    using SelectParents(), Chromosome.CrossOver(),
    Chromosome.Mutate(), and SetNewPopulation() to create
    new population. Code omitted. */
    public ChromosomePopulation FinalPopulation
        { get { return(currPop); } } // Returns the final population.
}

```

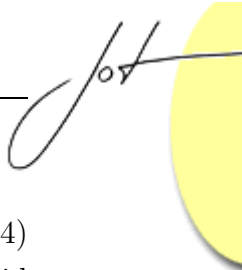
Figure 3: GAFController Class

$$\text{post.GAFController.SelectParents()} \equiv [(cP = cP@pre) \wedge (gAR = gAR@pre) \wedge (result_1 \in cP) \wedge (result_2 \in cP)] \quad (3)$$

This asserts that `SelectParents()` does not change the `GAFController` object, and both chromosomes in the result are from the current population (“ $\in$ ” denotes set membership). Note that this specification may be too strong. It would not allow the GA application developer to, for example, define this method to record, in `gAR`, information about the individuals selected as parents. We could allow this by weakening (3). The flip side of weakening the specs of the hook methods is that it would reduce the degree of reasoning reuse.

`InitPopulation()` is easily specified and we omit this spec. Next consider `SetNewPopulation()`. The default implementation simply sets `currPop` equal to `cChildren`. But more complex GAs are easy to imagine; for example, set the current population to be the union of some elements of `currPop` and some elements of `cChildren`, possibly based on the fitnesses of the individuals in question. Such behaviors are allowed by (4); indeed, (4) even allows us to introduce entirely new individuals that are not





$$post.GAFController.SetNewPopulation(childP) \equiv [(childP = childP@pre) \wedge (length(cP) > 0) \wedge (gAR = gAR@pre)] \quad (4)$$

in either  $cP@pre$  nor in  $childP$ , into the current population. If we want to forbid such behavior, we can do so by adding a clause such as

$$(cP \subseteq cP@pre \cup childP)$$

to this post-condition. This will ensure that each element that is in  $cP$  when this method finishes must have been in  $cP$  at the start of the method or in  $childP$  (or both).

Finally we turn to `RunGA()`. `RunGA()` works as follows: It first initializes the population. Then goes through `nn` iterations of “evolution”. In each, it replaces the current population with a new population; it also uses appropriate methods of `GAReporter` to produce reports at the start and end of each generation.

In each iteration, `RunGA()` produces the new population as follows: Use `SelectParents()` to select parents; use `Chromosome.CrossOver()` to “cross-over” parents to generate two children; use `Chromosome.Mutate()` to possibly “mutate” the children; add the children to a temporary population, created at the start of the iteration; repeat until the number of children is the same as the count of the current population; at the end of the iteration, use `SetNewPopulation()`, with the temporary population just created, to obtain the “next generation”.

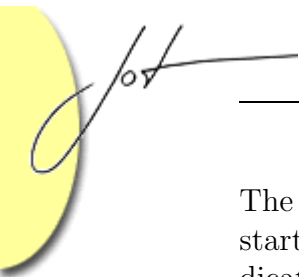
As we saw in Section 2, in order to allow us, at the application-level, to arrive at `RunGA()`’s richer behavior resulting from the redefinitions of these hook methods in the application, `RunGA()`’s specification must provide information about these calls and about how  $cP$  and  $gAR$  will be modified on the basis of the results of these calls. We do this by specifying the  $tPost$  (for “template-post-condition”) of this template method. Essentially,  $tPost.RunGA()$  gives us information not only about the final values of  $cP$  and  $gAR$  but also the calls to hook methods that `RunGA()` makes during its execution.

$$tPost.runGA(nn) \equiv (length(cP) = length(cP@pre)) \wedge \quad (5.1)$$

$$\langle \text{InitPopulation}(); gAR.InitRpt(nn, cP); \text{for } (i = 1, \dots, nn) \text{ do } NextGen(); gAR.FinalRpt(nn, cP); \rangle \quad (5.2)$$

$$NextGen() \equiv \langle gAR.BeginNewGenRpt(cP); \text{int } k := \text{currPop.Count}; \text{ChromosomePopulation } childPop = \langle \rangle; \text{for } (i = 1, \dots, k/2) \text{ do} \quad (5.3)$$

$$\{ \text{ChromosomePair } parnts = \text{SelectParents}(); \text{ChromosomePair } children = parnts_1.\text{CrossOver}(parnts_2); children_1.\text{Mutate}(); children_2.\text{Mutate}(); childPop.AddChromosomes(children_1, children_2); \text{SetNewPop}(childPop); gAR.EndNewGenRpt(cP); \rangle \quad (5)$$



The first clause, (5.1), simply states that the population size is the same as at the start of `RunGA`. The rest of the specification is enclosed in angle brackets (“`<>`”) to indicate that this part uses our ‘programmer-friendly’ notation to specify information about calls to hook methods. The first line of this part states that the first two hook methods that `RunGA()` will invoke are `initPopulation()` and `initRpt()` respectively (the latter being on the object `gAR`).

In the next line, we use a ‘macro’ notation. The macro `NextGen()`, specified in (5.3), captures the structure of hook method calls that `RunGA()` makes when going through one ‘generation’. Thus (5.2) asserts that the calls to `InitPopulation()` and `InitRpt()` are followed by *nn* instances of the structure specified in the `NextGen()` macro, followed finally by the invocation of `FinalRpt()` (on `gAR`). (5.3) gives us the structure of `NextGen()`: This consists of a call to `BeginNewGenRpt()`, followed by initialization of the ‘loop counter’ *k* to the `Count` value of the current population (`Count` being a property of the `ChromosomePopulation` class that gives us the size of the population in question), followed by a sequence of *k*/2 instances of the following: invoke `SelectParents()`, invoke `CrossOver()` on the parent chromosomes returned by `SelectParents()` to get the children, invoke `Mutate()` on each child, then use `AddChromosomes()` to add these to the new generation; and finally invoke `SetNewPop()` to update the current population with the new generation, `childPop`.

## Documentation

An important development [8] in the C# technology is the use of *XML-documentation* to provide useful information about the class; this information can consist of precise specifications of the class’s methods. Further, while we can use the ‘standard’ tags, we can also define new ones in a *style-sheet* and use these in the documentation. If the proper ‘tags’ are defined in the style sheets, one of the output of the C# compiler, an XML document containing all the documentation associated with the class, can be run (in conjunction with the style sheets) through an appropriate browser to produce very readable and informative output. We have experimented with this in our GA framework. An example of the C# source code that includes this type of documentation appears in Figure 4.

This piece of documentation corresponds to the method `Chromosome.CrossOver()` and contains the information we saw in the specification of this method. We have used a number of different tags here which are all defined in the associated style sheets (we will not look at these style sheets here; they are available from the web site mentioned previously). The resulting HTML document, viewed on a standard browser, appears in Figure 5. Documentation corresponding to two members, `CrossOver` and `Fitness`, of the `Chromosome` class are shown in this figure. The *contract* portions of the documentation for these two members are identical to what we saw earlier.

One point is worth noting concerning this type of documentation. It assumes that we have access to the source code of the classes in question, such as our `Chromosome`



```
/// <hookMethod mustOverride="yes"/>
/// <summary>Used to generate a new pair of chromosomes by crossing
///     over self with cOther.</summary>
/// <param name="cOther">Chromosome with which to crossover.</param>
/// <returns>The pair of chromosomes resulting from the crossover :
///     <see cref="ChromosomePair"/></returns>
/// <contract>
///     <precondition>
///         (typeof(cOther) = typeof(self))
///     </precondition>
///     <postcondition>
///         [(self = self@pre) ^
///          (cOther = cOther@pre) ^
///          (typeof(result.1) = typeof(self)) ^
///          (typeof(result.2) = typeof(self))]
///     </postcondition>
/// </contract>
public abstract ChromosomePair CrossOver(Chromosome cOther);
```

Figure 4: Documentation of `Chromosome` Class: C# Source Code (partial listing)

class. Indeed, what we have in Fig. 4 is part of the source code of that class. Clearly, such an approach will not work if we are interested in documenting classes for which we do not have the source with precise contracts. We will return to this question in the final section.

## 4 GA APPLICATIONS

The GA literature [5, 14, 15] provides numerous examples of genetic algorithms of varying complexities over many different domains. In this section, we will briefly discuss one of these GAs that we have implemented as an application on top of the framework presented in the last section. We also briefly discuss the behavioral aspects of these applications.

### Even GA

Our application is adapted from a GA presented in [15]. Here each chromosome is a string of ten bits, i.e., a chromosome is an unsigned, ten-digit binary integer. The goal of the GA is to ‘evolve’ from an initial population of a random collection of twenty such integers to a population of ‘high fitness’ individuals, where fitness is measured by the number of 1’s in the integer, with even integers being fitter than odd integers.

Framework Base-Class [GAF]	
<b>GAFramework.Chromosome.Chromosome</b>	
[ Hook Method : Must Override = yes ]	
<b>GAFramework.Chromosome.Chromosome.Crossover(GAFramework.Chromosome.Chromosome cOther)</b>	
<u>Returns</u>	
GAFramework.Chromosome.ChromosomePair	
<u>Contract</u>	
<b>Pre-condition:</b>	(typeof(cOther) = typeof(self))
<b>Post-condition:</b>	[(self = self@pre) ^ (cOther = cOther@pre) ^ (typeof(result.1) = typeof(self)) ^ (typeof(result.2) = typeof(self))]
[ Hook Property : Must Override = yes ]	
<b>GAFramework.Chromosome.Chromosome.Fitness()</b>	
<u>Value</u>	
get-only : System.Double	
<u>Contract</u>	
<b>Pre-condition:</b>	true
<b>Post-condition:</b>	(self = self@pre)

Figure 5: Documentation of `Chromosome` Class: The browser view

To build a new application, we have to define the appropriate derived classes of `Chromosome` and `GAFController`. These derived classes, for `EvenGA`, appear in Figures 6 and 7. `EvenChromosome` is straightforward. `EvenGAFController` uses a variable `lastIndex` to remember the most recently chosen chromosome. `SelectParents()` chooses two chromosomes with the probability of each chromosome being chosen being proportional to its fitness, and uses `lastIndex` to go through the chromosomes in a round-robin manner. `Main()` simply creates a `GAREporter` object, an `EvenGAFController` object `eGAF`, and then applies `RunGA()` (of `GAFController`) to it, passing 1000 as the argument to `RunGA()`, as the required number of generations of evolution. `RunGA()`, as we saw, will initialize the population (using the `InitPopulation()` defined in `EvenChromosome`), and go through the specified number of generations, using the `SelectParents()`, `Crossover()`, and `Mutate()` operations defined in Figure 6 to evolve the population from one generation to the next.

Note the power of the framework-based approach. In implementing `EvenGA`, all we had to do was specify how fitness of a chromosome is evaluated, how it is mutated, how a pair of parents is ‘crossed’ to get a pair of children, and how parents are selected. Everything else is defined in our framework and reused in the application. Admittedly, this is a simple application but more complex GAs are built in the same fashion.

Now consider the behavior of `EvenGA`. First, we must check that the behavioral subtyping requirements are met; for example, that the methods defined in `EvenChromosome` satisfy the specifications of the corresponding methods of

```

public class EvenChromosome : Chromosome {
    private char[] chrDNA; // will have 10 char's, each is '0' or '1'
    public override double Fitness {
        get {
            /* returns (2.0 + no. of 1's - 2.0 if last char is 1).
            Code omitted. */ };
    public override void Mutate() {
        /* with probability of .01, change the value
        of one bit of the chromosome; */ };
    public override ChromosomePair CrossOver(Chromosome cOther) {
        /* returns two new chromosomes c1, c2; randomly chooses
        an i, 0 ≤ i ≤ 9;
        copies, into c1.chrDNA[], the values from
        self.chrDNA[0...i] and cOther.chrDNA[i+1...9];
        copies, into c2.chrDNA[], and from cOther.chrDNA[0...i]
        and self.chrDNA[i+1...9]. */
        };
    }
}

```

Figure 6: `EvenChromosome`

**Chromosome.** We first define the conceptual model of `EvenChromosome`. This is essentially the same as its concrete model: a sequence of 10 bits with the values as in `chrDNA[]`. The specification (1) of `Chromosome.getFitness` was:

$$post.getFitness \equiv (self = self@pre)$$

To show that `EvenChromosome.getFitness` satisfies this, we just need to check that its code does not modify `chrDNA[]`. Similarly, `Mutate()` and `CrossOver()` of `EvenChromosome` can be shown to satisfy the requirements of behavioral subtyping.

The conceptual model of `GAFController` was a sequence of chromosomes, plus a reporter object. For `EvenGAFController`, the chromosomes must be `EvenChromosomes`. In addition, the model includes `UIndex`, an integer whose value is the index of the (second) even chromosome chosen by the most recent `SelectParents()` operation (and `-1` at the start). The mapping to the base class model simply omits the `UIndex` component. With this, it is easy to check that `SelectParents()` and `InitPopulation()` as defined in `EvenGAFController` do satisfy their base-class specs. Hence the template method `RunGA()` will continue to satisfy its framework-level specification.

Now consider the *enriched* behavior of `RunGA()`. First we need to consider the richer behavior of the hook methods. One important point is that by the very nature of genetic algorithms, many of the important functions are probabilistic, and this is true of many of the (hook) methods defined in `EvenGA`. But standard ways of reasoning about programs do not cater to probabilistic considerations. Standard DBC contracts can of course specify that a particular method can produce many different results (for a given initial state and argument values) but there is no

```

public class EvenGAFController : GAFController {
    private int lastIndex; // last chosen chromosome; used to
        // prevent repeated selection of same chromosome

    protected override ChromosomePair SelectParents() {
        /* From currPop (of GAFController), choose two chromosomes randomly,
        with probability proportional to their fitness, starting with
        chromosome at the (lastIndex+1) position in currPop, and going in a
        round-robin fashion; update lastIndex and return the pair of chosen
        chromosomes. */ }

    protected override void InitPopulation() {
        /* Initialize population to consist of 20 chromosomes with random
        bit patterns in each.*/ }

    static void Main(string[] args) {
        GARepoter gaRep = new GARepoter();
        GAFController eGAF = new EvenGAFController(gaRep);
        eGAF.RunGA(1000); }
}

```

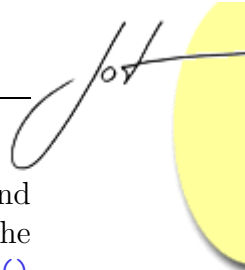
Figure 7: `EvenGAFController`

standard way for associating probabilities with the various possible results. Consider the behavior of the `EvenGA.SelectParents()`. From the description in Fig. 6 of this method, it is clear that the method may return *any* two individuals from the current population as the selected parents; and it would be easy to specify this (indeed, this is already expressed in the framework-level specification (3) of the method). Clearly this is inadequate. The key intuition behind methods such as `EvenGAFController.SelectParents()` is as follows: preferring parents with higher fitnesses but also allowing, with a lower probability, individuals with lower fitness to be chosen, in conjunction with random mutations, again with very low probability, allows the system to evolve, with high probability, a population of high fitness individuals. This is clearly not the same as random selection of parents and random mutations of individuals, which is what specifications such as (3) express.

One solution would be to extend the specifications of methods to allow us to associate specific probabilities with various results that the method may lead to. This would be important even in the case of `EvenGA.Mutate()`: as indicated in Fig. 6, this operation leaves the chromosome unchanged with a 99% probability. Consider the following specification:

$$\begin{aligned}
 \text{post.EvenChromosome.Mutate()} \equiv & \\
 & [(chrDNA[] = chrDNA[] @pre : 99\%) \wedge \\
 & (\text{diff}(chrDNA[], chrDNA[] @pre) = 1 \text{ bit}) : 1\%]
 \end{aligned} \tag{6}$$

This asserts that there is a 99% probability that the chromosome state will be the same as at the start of the operation; and that there is a 1% probability that the state at the end of the operation will differ from the state at the start by 1 bit. Developing ways to reason about probabilistic behavior will, we believe, become increasingly important as such algorithms are more widely used.



Once we have suitable specifications for `SelectParents()`, `Mutate()` and other hook methods defined in `EvenGA`, we can proceed to ‘plug’ them into the *tPost-condition* (5) of `RunGA()`. (5) tells us that in each generation, `RunGA()` calls `SelectParents()` and then applies `CrossOver()` to the results returned by `SelectParents()` to generate the *children* on which the `Mutate()` operation is applied and the resulting *children* added to the child generation; etc. This should let us conclude, given appropriate specifications such as (6), that when `EvenGA.RunGA()` finishes, the average fitness of the individuals in the final population will have a range of possible values, with the higher values having higher probabilities.

The above explanation of how we arrive at the richer behavior of the template method is admittedly informal. In [17], we have proposed a set of formal rules that can be applied to formalize this task. But the formalism of [17] requires the specification of the template method to be in a standard form, not in the programming language-like notation that we have used to specify `runGA()`. Moreover, that formalism does not cater to probabilistic specifications of the type we have proposed above. These are problems for future work.

## 5 DISCUSSION

The goals of our work were three-fold. First, as a case study of the framework-based approach, to apply the approach to capture the common aspects of genetic algorithms into a framework, so that when a new GA is implemented we can build it as an application on top of the framework rather than starting from scratch or trying to modify an existing GA by cutting-and-pasting portions of the code. The framework we designed and implemented seems to serve this purpose well. We have been able to implement different GAs on top of our framework in a natural fashion.

Our second goal was to explore the issues involved in reasoning about OO frameworks and applications built on them. The key question was whether we could achieve the same kind of reuse in the reasoning effort as the framework-based approach allows us to achieve in the implementation effort. There were two important issues here: First, we had to make sure that hook methods defined or redefined in the application satisfy their framework-level specifications; this is handled by the requirement of behavioral subtyping [11]. Second, at the framework level we had to provide a richer specification for template methods than is possible using standard post-conditions; in particular, we had to include information about the hook methods that the template method in question calls, and how it uses the results of such calls, etc. We did this by using a programming-language like notation to specify this additional information. Given such a richer specification of the template method, we could, at the application level, plug-in the behavior of the hook methods as defined at the application level, to arrive at the application-level behavior of the template method without reanalyzing its code. Our third goal was to use the documentation facilities of *C#* to generate documentation precisely specifying our framework as well as the applications built using it. We believe that the results,

such as the fragment shown in Fig. 5, demonstrate the importance and usefulness of such facilities especially in the case of frameworks.

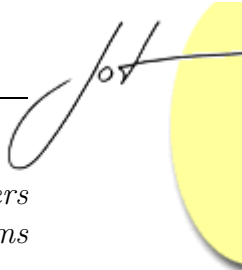
Let us now consider the possibility of adding such specifications to classes or frameworks that we might purchase, in compiled form (only), from a software vendor. Meyer [13] mentions a *contract wizard* that exploits the meta-data information that the C#/.NET framework provides to allow us to *add* class invariants and pre- and post-conditions to existing classes whose source code may not be available. This is very important because information about key methods such as `RunGA()` tends to be provided only informally; and frameworks are often supplied only in compiled form. If the *contract wizard* can be used to handle the ‘programmer-friendly’ specifications such as (5), we will be able to use it as follows: first convert, by hand, the informal documentation into precise specifications; next, use the contract wizard to add these specifications to the framework. We intend to investigate the possibility of using the contract wizard to deal with specifications such as (5).

We will conclude with a comment about our specification notation. Although the notation used in (5) for specifying the template methods is programming-language-like, these are specifications, not programs. In particular, they do not give us any information about how the method manipulates its local variables, etc. In future work, we plan to define this notation more precisely and to also make more precise the somewhat informal technique we used in the last section for going from the framework-level specification of a template method expressed in this notation to its application-specific behavior.

## REFERENCES

- [1] T Archer. *Inside C#*. Microsoft Press, 2001.
- [2] M. Buchi and W. Weck. The greybox approach: when blackbox specifications hide too much. Technical Report TUCS TR No. 297, Turku Centre for Computer Science, 1999. available at <http://www.tucs.abo.fi/>.
- [3] V. Dhar, D. Chou, and F. Provost. Discovering interesting patterns for investment decision making with glower – a genetic learner. *Data Mining and Knowledge Discovery*, 4:251–280, 2000.
- [4] M.E. Fayad and D.C. Schmidt. Special issue on object oriented application frameworks. *Comm. of the ACM*, 40, October 1997.
- [5] D.B. Fogel. An introduction to evolutionary computation and some applications. In K. Miettinen, M.M. Makela, P. Neittaanmaki, and J. Periaux, editors, *Evolutionary algorithms in engineering and computer science*, pages 23–43. John Wiley, 1999.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.





- [7] M. Gen, G.S. Wasserman, and A.E. Smith (Guest Editors). *Computers and Industrial Engineering Journal, Special issue on genetic algorithms and industrial engineering*. Vol. 30, No. 2, 1996.
- [8] E. Gunnerson. *A programmer's introduction to C#*. Apress, 2001.
- [9] J.H. Holland. *Adaptation in natural and artificial systems*. Univ of Michigan Press, 1975.
- [10] K. Juliff. A multichromosome genetic algorithm for pallet loading. In *Proceedings of the Fifth Int. Conf. on Genetic Algorithms*, pages 467–473. Morgan Kaufmann, 1993.
- [11] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, 16:1811–1841, 1994.
- [12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
- [13] B Meyer. .NET is coming. *IEEE Computer*, 34(8):92–97, 2001.
- [14] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer, 1996.
- [15] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1997.
- [16] R. Le Riche and R.T. Haftka. Genetic algorithms for minimum thickness composite laminate design. *Composites Engineering*, 3:121–139, 1995.
- [17] N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In W. Frakes, editor, *Proc. of Sixth Int. Conf. on Software Reuse: Advances in Software Reusability*, LNCS 1844, pages 100–116. Springer, 2000.
- [18] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.

## ABOUT THE AUTHORS

**Jason Hallstrom** is a graduate student in the Computer and Information Science Dept. at the Ohio State University. Jason is interested in various aspects of Software Engineering; his Ph.D. work deals with developing techniques and tools for building large systems in a flexible and reliable manner.

**Neelam Soundarajan** is an Associate Professor in the Computer and Information Science Dept. at the Ohio State University. His primary interests are in Software Engineering, specifically in reasoning about program behavior.