Object oriented design for random number generators

Alexandr Štefek

Military Academy in Brno

alexandr@stefek.cz

Abstract

This paper shows basic principles for object oriented design (OOD) in random number generators development. With OOD we can define basic class with virtual (and abstract) functions. Then its possible, in inherited classes, override base method implementation.

Random number generators

There are several methods for random number generating. The most known (in simulation used) is kongruent random number generator. This class of generator is described by next equation

$$x(k+1) = (ax(k)+c) \bmod m$$

In basic implementation we can simply define the function and state variable

```
//A=10309151; C=10309177; M=2147483647; X0=1
var
  X : Longint = 1;
const
  A = 10309151;
  C = 10309177;
  M = 2147483647;

function SomeRND: Longint;
begin
  X := (A * X + C) mod M;
end;
```

In some cases it is good implementation. But in general this design is not flexible. In the same code is not posible to implement something like

```
//A=10309151; C=10309177; M=2147483647; X0=1
var
  X : Longint = 1;
const
  A = 10309151;
  C = 10309177;
  M = 2147483647;

function SomeRND: Longint;
```

```
begin
  X := (A * X + C) mod M;
end;

//A=10309319; C=10309333; M=2147483543; X0=1
var
  X : Longint = 1;
const
  A = 10309319;
  C = 10309333;
  M = 2147483543;

function SomeRND2: Longint;
begin
  X := (A * X + C) mod M;
end;
```

Yes we can rename the used state variable and constant. But more flexible is implementation with OOD.

Object oriented design

As a base in OOD we can define class

```
  TAxRandom = class(TObject)
  public
    function Random: Double; virtual; abstract;
  end;
```

There is defined virtual abstract method for generating random numbers. Now we can inherit this base class for method overriding.

```
  TCongruentRandom = class(TAxRandom)
  private
    A, C, M: Double;
  public
    function Random: Double; override;
  end;
```

This class has own congruent random number generator constants. So we can create instances of this class and set for all instances diferent parameters for generation. With this implementation we reach some advantage, we have two (or more) independed random number generators.

Exteded object oriented design

We introduce in previous chapter basic class design. We can extend this design with some methods.

```
  TAxRandom = class(TObject)
  public
```

```
    constructor Create(const Params: string);

    procedure Initialize(Params: TStrings); virtual;
    procedure Randomize;
    function Random: Double; overload; virtual;
    function Random(const A, B: Double): Double; overload;
    function Random(Range: Longint): Longint; overload;
  end;
```

In this extended desing we introduce some new methods. There are some support for random number generator initialization (constructor `Create`, virtual method `Initialize`). This design supports randomization (`Randomize`). Method `Randomize` calls random-time method for readout random number. This process throws generator to valid random state. Overloaded methods `Random(const A, B: Double): Double;` and `function Random(Range: Longint): Longint;` is used for generating numbers from predefined interval or Longint type random number.

Class with registration

It is possible to extend the desing with random number generators registration.

```
  TAxRandom = class(TObject)
  public
    constructor Create(const Params: string);

    class function CreateRND(const Name, Params: string): TAxRandom;
    class procedure RegisterRND(const Name: string; AClass: TAxRandomClass);

    procedure Initialize(Params: TStrings); virtual;
    procedure Randomize;
    function Random: Double; overload; virtual;

    function Random(const A, B: Double): Double; overload;
    function Random(Range: Longint): Longint; overload;
  end;
```

With registration we can register the new class for implementing random number generator. Now the user can list all registered random number generators and choose one from list.

Some inherited classes

We can create some new classes for diferent types of randum number generators.

```
  TAxUniformRND = class(TAxRandom)
  end;

  TAxShuffleRND = class(TAxUniformRND)
  private
    FFirstRND: TAxUniformRND;
    FSecond: TAxUniformRND;
    FCells: array of Double;
  public
    procedure Initialize(Params: TStrings); override;

    function Random: Double; override;
  end;

  TAxLCG = class(TAxUniformRND)
  private
    FX0: Int64;
    FA: Int64;
    FM: Int64;
    FC: Int64;
  public
    procedure Initialize(Params: TStrings); override;

    function Random: Double; override;

    property A: Int64 read FA write FA;
    property C: Int64 read FC write FC;
    property M: Int64 read FM write FM;
    property X0: Int64 read FX0 write FX0;
  end;

  TAxBasedRND = class(TAxRandom)
  private
    FBaseRND: TAxUniformRND;
  public
    procedure Initialize(Params: TStrings); override;

    property BaseRND: TAxUniformRND read FBaseRND;
  end;

  TAxExponentialRND = class(TAxBasedRND)
  private
    FLambda: Double;
  public
    procedure Initialize(Params: TStrings); override;
    function Random: Double; override;

    property Lambda: Double read FLambda;
  end;

  TAxNormal = class(TAxBasedRND)
  private
    FStdDev: Double;
    FMean: Double;
  public
    procedure Initialize(Params: TStrings); override;
    function Random: Double; override;

    property Mean: Double read FMean write FMean;
    property StdDev: Double read FStdDev write FStdDev;
  end;

  TAxElimination = class(TAxBasedRND)
  public
    function Random: Double; override;
    function Density(const X: Double): Double; virtual;
  end;
```

Above is defined seven classes for generators. Some of classes is defined as besa class for other implementations (`TAxBasedRND`, `TAxUniformRND`).

Conclusion

In this article we show some usefull principles in object oriented design for random number generator implementation. The main advantage is

- creating independed generators
- future expansion possibility
- possibility generator registration