Object-oriented pseudo-random data generator realizations

Tony Karavasilev

1Computer informatics, Plovdiv University “Paisii Hilendarski”, Bulgaria

E-mail: tony.karavasilev@gmail.com

Abstract. With the development of computational science and computer simulations, the randomness source has become a necessary part of every complex system and testing environment. Most programming languages, software frameworks and application libraries do not provide all different types of randomness generators and do not even implement high-quality algorithms for secure data generation. This paper presents an object-oriented solution for providing all the main classifications of randomness via random data generator classes that offer frequently used formats in information technology and in statistical hypothesis testing. It also presents usages for secure data generation and confirms the importance of true randomness in cryptography. The purpose of the developed practical experiments is to evaluate and show the differences in the sample generation quality of the sub-classifications of software randomness generators using the developed object-oriented components.

1. Introduction

The general language definition of randomness is the absence of a pattern or any predictability in an event. The random sequence of events, symbols or steps must also be unpredictable or without any visible combination order. Frequently, when testing with a huge amount of unpredictable events, also known as trials, patterns may still appear. An example of this may be the throwing of a dice or a coin multiple times until revealing some probability model or repeated behavior. [1]

It is important to mention that there is a link between randomness and many philosophical discussions about existence, natural laws and free will. For example, the indeterminism theory completely relies on randomness via the assumption that all events in nature are a cause of pure chance or even luck. While on the other hand, the determinism theory implies that everything happens for a necessary reason and has an underlying cause, which completely opposes the existence of true randomness itself. [2]

In mathematics and statistics, a random variable is an assignment of a numerical value to each possible outcome of an event scope or domain. This makes it possible to calculate a probability value for each valid event. When applying this scientific view, randomness becomes a valid measure of uncertainty or even applies a concept of chance in an outcome. There are many applications, such as the evaluation of the information entropy of a message by assigning a higher information rate to lower-probability events and finding the low-level building blocks in huge data samples. This way we can use it also as a measure of disorder in a system or a type of uncertainty associated with a random variable in a certain domain. [3]

The realization of practical randomness sources in computational science and software technologies relies completely on number theory algorithms and the building of random enough looking number sequences. Most of them rely on the current system time with a combination of binary operations between numbers in a certain valid domain. These algorithms have the ability to produce the same output by setting a custom value on initialization, also known as an initial seed or just seed. As a term in
computer science, randomness is considered as noise, which represents some irrelevant or meaningless data. Having said this, some high-quality algorithms are initialized with values generated by radio waves caused from magnetic reconstructions of the computer’s hardware devices, such as the central processing unit or the hard disk drive, etc. Operating systems make such low-level data accessible via the kernel’s software drivers as a source of noise that has higher entropy and can make the used mathematical algorithms far more unpredictable. Even though the generated noise is still limited to the byte event space, it is a lot harder to guess it and is more immune to logical attacks. [4]

Having said this, even if the used noise sources seem to be truly unpredictable, with time some of them will be scientifically explained and will no longer be looked upon as random. This can make our systems vulnerable to new zero-day attacks in the near future but also provides an easy mitigation method by just switching to another source or generation algorithm. Philosophically said, everything is random until we find an explanation and can reproduce the exact logic of the given process.

In computer systems, three main branches of the software random data generator realizations exist:

- Quasi-random number generator (QRNG);
- Pseudo-random number generator (PRNG);
- Cryptography secure pseudo-random number generator (CSPRNG).

The quasi-random type, also known as sub-random, is a low-discrepancy sequence. This type of sequence is uniformly distributed and the number of points in it is usually close to the measure of the whole count of the used set. Although they may fail most statistical tests for randomness, sub-random numbers have an advantage over pseudo-random numbers because they cover or fill the domain more evenly and quite faster. They tend to be very useful in financial mathematics and data mining algorithms. They also have the ability to repeat the same result via an initialization value. The most famous algorithms for this type are the Halton sequence and the Sobol sequence. [5]

On the other hand, the pseudo-random type appears to be truly random but in fact, it is not. The whole point of the generated sequence is to exhibit statistical randomness, which means to be close enough to true randomness on the first look by not having any recognizable patterns or visible order. This type is also controllable via an initialization value and is very useful in the realization of computer simulations, statistical analysis and software testing. Most software developers face this type of computer randomness realization first and may even use it for years before understanding the concept of true randomness. The most famous algorithms for this type are the linear congruential generator (LCG) and the Mersenne Twister generator (MTG). [3]

The most important type of randomness realization is the cryptographically secure pseudo-randomness. Its output sequence is harder to guess and is not controllable via seed values. The crypto-randomness algorithms used may vary but are far more complex than what the previous two types use internally. They pass statistical tests and are immune to attacks, even if the attacker knows the initial state passed to the algorithm or the consumed entropy pool location. They harvest the full force of high-quality sources of noise via the operating system’s kernel. This type of randomness realization is suitable for cryptography purposes and network protocol algorithms. The best cyber security practices completely rely on the quality of the unique identifiers, tokens, keys, salts and nonces. The most famous algorithms for this type are the Yarrow generator and the Fortuna generator. [6]

Most programming languages, software frameworks and application libraries do not provide a lot of different types of randomness sources, which creates many problems with the overall application security status. In most cases, there is only a common pseudo-randomness algorithm and some basic cryptographically secure pseudo-randomness realization, both providing only basic data formats like positive numbers and raw bytes. This creates many disadvantages, for example:

- Not suitable for all scientific use cases;
- Not easily injected into other class services;
- Not enough data generation output formats available;
- No object-oriented class hierarchy exists or only global functions exist;
- Security, performance and thread safety may vary across different language versions.
The main purpose of this article is to provide an object-oriented solution for the implementation of all the main classifications of randomness via random data generator classes that support frequently used formats in information technology and computational science. This paper includes a practical implementation of an object-oriented class hierarchy for randomness realizations and extended sample generation experiments to illustrate the main differences between the three sub-classifications of software randomness. This research is also a part of ongoing work in developing an advanced PHP object-oriented software framework for cryptographic services.

2. The theoretical object-oriented solution

In most programming languages, the provided functions or classes only support basic pseudo-randomness implementations and in most cases at least one cryptography secure pseudo-randomness algorithm. No matter the case, the first step of creating an object-oriented solution is to build sophisticated wrapper classes and create low-level native realizations for all missing algorithms.

2.1. Building a suitable class hierarchy

Have in mind that even if we create classes as simple wrappers of other language provided classes or functions this would not be enough for an object-oriented solution to work. We must stay away from the anti-pattern known as God object, which is a class object that knows too much or does too much. The realization must carefully split reusable code into an abstract base class and force the desired implementations via interfaces or abstract methods. The abstract representation of randomness must force all derived objects to implement at least integer and byte generation methods. It must provide a constructor method specification for forcing the same type of instance creation for all derived classes.

Note that not every sub-classification of randomness can support initialization seeding, like the cryptography secure pseudo-randomness. It is a good idea to keep this functionality away from the base class and move it to an optional interface specification. Even if you leave it as an abstract method at the base class, you must never implement any logic for it when creating a cryptography secure generator and can force it to throw an exception on invocation. Figure 1 shows an example of a class hierarchy as a Unified Modelling Language (UML) class diagram.

![Diagram](image)

**Figure 1.** UML Diagram for an Example Randomness Class Hierarchy

This hierarchy can become more sophisticated as a realization, for example moving the generation or validation methods into interfaces and implementing them directly into the abstract base class. In some cases, you may even want to split methods into multiple abstract classes that inherit each other.

2.2. Creating native language realizations

Most programming languages have a limited amount of features in them for several reasons like compatibility, test coverage, security, performance and paradigm specific features. When a researcher or a developer needs some specific computational algorithm, in most cases, the implementation is a complete language native one. In the case of algorithmic information theory studies, the native code simulation of randomness sources is done via number theory specific realizations based on binary
operations and mathematical formulas. For cryptography purposes, the realization must also always consume a high entropy physical source of noise provided via the operating system’s kernel.

Typically, programming languages do not provide a low-discrepancy sequence generator. When creating one, we must keep in mind that the implementation of this type of space-filling algorithm is resource hungry. This comes from the need to keep and access a huge set of numbers in memory for manipulations that can truly slow things down. The most common optimizations are using a faster but worse shuffling algorithm or a smaller supported domain size. Some approaches even use the same hardcoded sequence for speed reasons, which is not truly acceptable for research purposes.

Pseudo-randomness is a part of every widely used language. Native implementations are important only when the built-in algorithm is old or not good enough. In many cases, the seeding algorithm implied is time-based and may need access to more precise sources of noise. Note that supplying certain seeding values based on the used algorithm may also create a poorer output sequence than expected. The realization of cryptography secure pseudo-random generators is mostly necessary when the built-in language functionalities do not provide a secure enough implementation. Most known approaches modify existing pseudo-random algorithms or use 3rd party implementations. As a strict rule, when writing in native code we must never insert a higher predictability rate into our algorithms.

2.3. Adding optional output formats
As discussed before, bundled language features mostly provide the generation of only positive numbers and raw bytes. The first problem every developer faces is the need for a wider range of output generation formats. When providing a software solution for random data generation, the first step is to implement negative number ranges. A simple algorithm for achieving this is extracting a number with at least half of the supported range on every generated number to provide both negative and positive numbers. Another approach may be pooling a number between zero and one, to simulate a Boolean expression, to choose if the number is positive or negative.

Another missing feature is the floating number output format that uses a certain rounding precision. This is very useful for statistical or financial simulations. A well-known algorithm is generating a number from a range and dividing it by the highest supported one in the same range. This way we achieve a probability floating number format between zero and one (as 0.0 – 1.0), widely used for statistical representation. Having this, we can easily multiply it to generated integer numbers to provide floating precision numbers in a bigger range or use it directly for some chance simulations.

A huge missing feature for every professional developer is the random string generation. This may include hexadecimal, Base64, ASCII, Unicode and other string formats. Every complex system needs the ability to provide secure passwords, strong authentication tokens, encryption initialization vectors and globally unique identifiers. A good realization would be creating a custom mapping of certain characters to certain positive numbers and just concatenating printable output characters based on generated random integer values in the range of the collection size.

2.4. Using the dependency injection principle
When creating a hierarchy of objects, we may come to the need for writing higher logic in container-based classes that reuse some of our objects as services. The approach of building container classes that can inject different objects of some type and use them as services that follow the same specification is the general definition of the dependency injection technique. [7]

Advanced examples for the usage of the dependency injection principle in random data generators is defining container objects for data shuffling and secure token generation. This way it is possible to supply different quality generators for the internal processing logic of the container. The usage of dependency injection increases the code reuse and decreases the coupling between a certain class and its dependencies. This specification allows all abstract randomness representations to be injectable into any class that inherits the container abstraction or implements some kind of service setter interface.
2.5. Backward compatibility and thread safety precautions
When creating an object-oriented solution, two main problems seem to always arise. The first one is backward compatibility and the second one is thread safety. When dealing with backward compatibility issues, a developer is attempting to keep the same behavior between language versions. To achieve this, the object creation requires native implementations, parameter type checks, various integer allocation optimizations and the execution of different logic for each used language version.

Thread safety issues exist even if the language does not have any asynchronous capabilities. The most frequent example is when the run-time execution fetches the same numerical sequence on each dynamic call in a certain scope or even causes process isolation failures by calling the same sequence for parallel requests. To mitigate these failure cases, a developer must always implement the algorithm logic in static memory and force the dynamic object creation to consume it via internal static method calls only. This will decrease the scope problems to a minimum and defines a single unified data pool for all object instances. When we want to deal with parallel request processing issues, which is a problem in both asynchronous and request-response architectures, we need to create a high-quality initialization logic that seeds unique values for every coexisting request. To cope with these problems, we must create shared memory locks for used resources at the object creation logic and consume only high-entropy sources of noise. A simplified method for non-cryptography secure generators would be using the current time to a microsecond as the initial value for the sequence generation algorithm.

3. The software solution implementation
With the wide case usage of pseudo-random data generators, the need for a practical implementation of the discussed theoretical solution is tremendous. The created software solution for all three main subtypes of abstract randomness is introduced as a part of a PHP object-oriented software framework for cryptographic services. The next sections provide a detailed practical analysis of the implementation and completely rely on the previously discussed realization approaches. The complete source is available and published online at the framework’s GitHub repository. [8]

3.1. Quasi-randomness implementation
As discussed before, the realization of quasi-randomness algorithms is always resource-demanding because of the uniform distribution and the huge amount of points used in a set. Any simple optimization may end up breaking the distribution and converting it to a pseudo-random sequence instead. This is why many operating systems reuse the same hardcoded quasi-random sequence into their kernel internal resources, which may be a great optimization but lowers the dynamic computational use cases. [5]

The basic algorithm for producing a low-discrepancy sequence has three main steps – generation (and scrambling), initial skipping and between calls leaping. The generation of a set of any amount of numbers is simple in any programming language. The shuffling or scrambling, on the other hand, may create different performance problems or if not used properly can even convert the sequence into a pseudo-random one instead. The solution for the implementation is to shuffle the set of points only once or one in a dozen of times for each program execution. This way we ensure that the sequence seems scrambled enough and will not hurt the uniform distribution of the randomness generator. The skipping step is a simple way of ignoring the first amount of numbers in a scrambled uniform set of numbers. This part can be skipped to use the whole set. It is a good idea to use a different skipping amount depending on the seed value for the generator or to hardcode a smaller value.

Finally, we must use a leap step to specify the number of points to ignore on every call to the generator. The leap step may be applied either on every call or be included in the initial sequence generation to reduce the overall memory usage. Defining a leap step of value one will make every call to the generator skip exactly one point and may lead to the decreasing of the usable sequence size to exactly half of the initial set. We may need to set this at zero if we want to use the generated set for domain filling purposes. Note that there may be a need for more than one internal sequence, because fetching in closer ranges (like bytes – $2^8$) will loop or scramble the set too often, thus cause extra load.
3.2. Pseudo-randomness implementation

The realization of pseudo-randomness algorithms in software platforms is simpler and more resource friendlier than the one for quasi-random generation. There is a huge amount of different methods for simulating statistical randomness in computer systems. As discussed before, most programming languages have at least one realization available internally.

If your language does not have any implemented, you can code one yourself or use one of the many open source implementations online. In most cases, when creating an implementation for your system it would be enough to create a wrapper object for the language provided functionality and manipulate values in a certain way to provide more features without breaking the pseudo-randomness characteristics. If the generated sequences are not good enough, then you can provide a high-entropy source for the algorithm’s initialization seeding value to improve its overall quality and security.

3.3. Cryptography secure pseudo-randomness implementation

As discussed before, the realization of a cryptographically secure pseudo-random generator is always tricky. When there is an algorithm available in your language, you must definitely use it directly. The best solution is to create a wrapper object for choosing and switching secure algorithms in the correct backward compatible manner for different language versions. Your code may also need to measure and evaluate all entropy sources for the current system and chose the best one available. Note that most programming languages update their cryptography secure pseudo-random algorithms and their internal cryptography libraries quite frequently, so make sure to keep your system up to date.

If your language does not support a secure enough algorithm, then always use only official high-grade open source implementations and consume the highest entropy source available. In some rare cases, you can even disable the seeding feature of a high-quality pseudo-random algorithm that has proven to be prone to security attacks and supply initialization values directly from the operating system’s high-entropy source or even via a 3rd party hardware chip [6]. It is important to note, that you must never create your own homemade algorithms and always stick to proven cryptography standards.

4. A practical comparison of different randomness realization types

After the complete implementation of all three main generator subtypes, there is a need for a practical comparison of their theoretical specification. The next experiments evaluate the correct source code realization and produce a visual correlation measurement between the outputs of the created data generator classes. The next sections describe the executed experiments and show their results.

4.1. The testing environment specification

All the practical experiments are run under a virtual machine environment created with Oracle VM VirtualBox version 6.0.6 hypervisor installed under Windows 10 Home Version 1809 x64 (Build 17763.437) as a host operating system. The setup of the guest virtual machine consists of running Apache 2.4.39 with PHP 7.3.5-FPM (the FastCGI Process Manager implementation). The hardware specification of the virtual machine and the used guest operating system is shown in Table 1.

| Component | Details |
|-----------|---------|
| CPU       | Intel i7-6700HQ, 2 cores, 2.59 GHz, 6 MB L3 |
| RAM       | DDR4, SODIMM, 4096 MB, 2.40 MHz |
| GPU       | Intel HD Graphics 530, 16 MB, 2.40 MHz |
| HDD       | 42 GB, 7200 RPM, 32 MB cache, 2GB swap |
| OS        | Ubuntu Server 18.04.2 LTS x64, Linux Kernel 5.0.10 |

The installation of the virtual machine has all existing updates, kernel drivers and virtualization-specific packages. The PHP and Apache configuration uses all default installation settings. The next
experiments generate images to create a visual representation of each randomness generator type. The generation of all images is done with the PHP extension GD2, also known as the GD library. The colour scheme used for the image generation is the common RGB representation, which is an additive model that uses a combination of red, green and blue as 8-bit integers to produce 16 777 216 (24-bits, 256^3) unique colours. The output images are in the famous Video Graphics Array (VGA) format with a resolution of 640x480 pixels, also known as the 480p standard, which uses a 4:3 aspect ratio.

4.2. Generation of a black and white 480p image
The first test for our randomness generator objects is to represent each of the output numerical sequences as a black and white output images to measure the quality of the produces statistical noise. The implementation of the experiment includes generating numbers in a Boolean format to map the integer number zero to the black colour and the integer number one to the white colour. Each of the three randomness visual representations can be seen in Figure 2.

![Figure 2. Black & White Visual Product – QRNG vs. PRNG vs. CSPRNG](image)

As we can see from the results, the quasi-randomness implementation correctly differs from the other two types because of its uniformly distributed numerical set. The chaotic output of both the pseudo-randomness and the secure pseudo-randomness is evidently random enough, with more image artifacts for the secure one. It is important to note that running the same test with all RGB colours via the generation of a set of three 8-bit numbers (of 0-255 range) did produce the same output patterns.

4.3. Drawing of a polygon figure at a 480p image
The next experiment includes the drawing of a polygon using 10 000 generated points of white colour over a completely black background. The implementation generates points via pairs of two integer numbers to represent the X and the Y coordinates. Each of the generated points is only in the visible realm of the output image. The experiment’s result can be seen in Figure 3.

![Figure 3. Polygon Visual Product – QRNG vs. PRNG vs. CSPRNG](image)

The quasi-randomness visual product confirms the uniform distribution of the internally used set and the algorithm’s space-filling characteristics. The secure pseudo-randomness implementation produces a higher point density and has less visible patterns than the pseudo-randomness one does.
4.4. Drawing of multiple ellipses at a 480p image
The last test includes the drawing of 10,000 ellipses using random colours, shapes and coordinates. The initial background colour of the image is completely black. The width and the height of the ellipses will be between 3.00% and 80.00% of the picture’s resolution to produce different shapes. The implementation generates all central points via pairs of two integer numbers to represent the X and the Y location coordinates. The generated points are only in the visible realm of the output image, but the figure may not be completely. The visual product for each of the generators is shown in Figure 4.

![Figure 4. Ellipses Visual Product – QRNG vs. PRNG vs. CSPRNG](image)

The quasi-randomness visual product is evidently not statistically random. The secure pseudo-randomness generator produces more different ellipses than the pseudo-randomness one does.

5. Conclusion
This paper has created an object-oriented solution for providing all the main classifications of randomness via pseudo-random data generator classes that can output a vast range of information formats. It maps the best uses of each of the randomness subtypes and demonstrates their characteristics via visual representations.

This article successfully confirms the importance of software randomness in the fields of cryptography, mathematics, statistics and computer science.

Acknowledgments
University of Plovdiv Paisii Hilendarski, 24 Tzar Asen, 4000 Plovdiv, Bulgaria

References
[1] Menezes A, van Oorschot P and Vanstone S 1996 Handbook of Applied Cryptography ISBN: 9781439821916
[2] Svozil K 2018 Physical (A) Causality: Determinism, Randomness and Uncaused Events ISBN: 9783319708157
[3] Goldreich O 1998 Modern Cryptography, Probabilistic Proofs and Pseudorandomness (Algorithms and Combinatorics) ISBN: 9783540647669
[4] Matsumoto M, Saito M, Haramoto H and Nishimura T 2006 Pseudorandom Number Generation: Impossibility and Compromise DOI: 10.3217/jucs-012-06-0672
[5] Alon N, Coja-Oghlan A, Hán H, Kang M, Roedl V and Schacht M 2007 Quasi-Randomness and Algorithmic Regularity for Graphs with General Degree Distributions DOI: 10.1137/070709529
[6] Stipčević M and Kaya Koç Ç 2014 True Random Number Generators DOI: 10.1007/978-3-319-10683-0_12
[7] Hudli S and Hudli R 2013 A Verification Strategy for Dependency Injection DOI: 10.7763/LNSE.2013.V1.16
[8] Tony Karavasilev (Тони Каравасилев) 2019 TonyKaravasilev/CryptoManana DOI: 10.5281/zenodo.2604693