UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Designing a flexible, pseudo random generator for sensor data output in IoT systems

Oskar Carl
Angewandte Informatik (Bachelor)

UNIVERSITÄT DUISBURG ESSEN

Eingebettete Systeme der Informatik, Abteilung Informatik
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

Erstgutachter: Univ-Prof. Gregor Schiele
Zweitgutachter: Prof. Dr-Ing. Torben Weis
Zeitraum: 30. Dezember 2016 - 31. März 2017

Abstract

The expansion of the Internet of Things in the years to come will require new methods to handle and analyze the large amounts of data generated by it. Extensive and diverse data sets are needed to aid in the creation of these algorithms and applications. This work describes the design of a generator to synthesize sensor data sets to the configuration of user.
The design is based on existing research on generators used in different environments, focusing particularly on pseudo random numbers. It is flexible and easy to incorporate into various environments and is built to be expanded in different ways.
A provided reference implementation is shown to perform well and produce consistent and configurable outputs. It is able to generate appropriate amounts of data with different user-controllable properties such as distribution, location, and width.
The presented design and implementation should prove useful in future works dealing with various aspects of analyzing and handling large sets of sensor data.

Contents

1. Introduction **1**
1.1. The Task at Hand . . . 1
1.2. Outline . . . 2

2. Foundation **3**
2.1. Complex Seeded Generators . . . 4
2.2. Cryptographic Random Number Generators . . . 4
2.3. Scientific Random Number Generators . . . 5
2.4. Conclusions . . . 6

3. Design and Implementation **9**
3.1. Design . . . 9
3.1.1. Supplementary Instructions . . . 11
3.2. Object-Oriented Application in Java . . . 11
3.2.1. Java 8 API: *Random* . . . 12
3.2.2. Implementing the Design in Java . . . 12
3.2.3. General Process . . . 14
3.2.4. Rationale . . . 15
3.3. Reference Implementation . . . 17

4. Evaluation **21**
4.1. Adaptability/Expandability . . . 21
4.2. Configurability and Distribution of Outputs . . . 22
4.3. Repeatability . . . 23
4.4. Maximum Amount of Outputs . . . 24
4.5. Performance . . . 25
4.6. Summary . . . 26

5. Conclusions and Future Work **27**

A. Additional Implementation Diagrams **29**

B. Evaluation Results **33**

Bibliography **39**

Chapter 1.

Introduction

In recent times, a trend has emerged to embed network-connected devices wherever possible, creating the *Internet of Things* (IoT). Sensors are being deployed everywhere to improve predictions of natural events. Monitoring that has been done manually before – possibly even with analog devices – will now be managed remotely via digital networks. This introduces billions of new nodes into the internet, each producing small amounts of data to be sent over the network in regular intervals. The sheer amount of devices makes the volume of data extremely large compared to present numbers. Current projections predict that by 2020 embedded devices will account for the same amount of data as the entire internet generated in 2014 [13].
Because of this rapid growth it is important to create systems to analyze and utilize this information. For this it is vital to create suitable algorithms and applications to automate this process. An issue with this is the absence of standardized data models right now, though universal specifications are being developed [3]. This makes it complicated to use real-world data since the application would have to be adapted for each set independently. One solution to this is to use a generator to create synthetic data sets to work with. With a generator it is possible to create as many data sets as needed with a consistent model, making it easier to create and test applications and algorithms. These can be adapted to different models after correctness has been verified.
In the process of creating these applications and algorithms it can be desirable to focus only on the data itself, omitting the aspect of transportation. However, this also renders existing, widely used solutions such as the *ns-3* network simulator [10] impractical, as it heavily focuses on simulating the transactions between network nodes.
Because of this it is necessary to design a generator that is specifically tailored to support the creation of such applications and algorithms.

1.1. The Task at Hand

This work describes the design of one such generator for sensor data output.
The generator is intended to be used for testing of algorithms, but could also be useful in other applications such as simulation of sensor networks. For this, outputs should

be sufficiently randomly distributed to be usable for broad tests. This means that the generator will have a close relation to *random number generators* (RNGs), making it suitable to look into the process of random number generation. Additionally it has to be configurable to appropriately cover different special cases, which also makes the generator more versatile and useful to more types of applications.
Given the amount of data to be handled by those algorithms, the generator must also be able to produce sufficiently high numbers of output data sets. While the orders of magnitude are not in the realms of some large scientific projects in fields such as physics or complex biological systems, it is essential to take this into account.
Since a universal standard for sensors is being developed by the IEEE [3], it should be a feasible assumption that more sensor data will be in consistent formats in the future. Based on this it would be appropriate to use the in-progress IEEE 1451 standard as guideline for the data model of this generator. Furthermore the generator should be easily adaptable to generate different data models, possibly being adjusted to new guidelines set by IEEE 1451 in the future. This also means that the design should be as simple as possible to make it easier to alter.
It is assumed that only the output of the generator will be used, which allows complete exclusion of transportation aspects from the application. The elimination of transportation from generation will provide some benefits to the design of the generator, delivering gains in performance and reducing the overall complexity of the design. This makes it easier to adapt and more flexible, enabling quicker development and better optimization.

1.2. Outline

Chapter 2 will focus on existing research related to this task. To create a detailed list of requirements for the design, works related to scientific data generation will be reviewed. This also includes different types of generators, such as random number generators as well as pre-seeded generators.
Chapter 3 presents the proposed design of the generator and explains the decisions behind it. After this, a general implementation for object-oriented languages is given and utilized to explain the general procedure. This is then used to introduce the reference implementation, which is provided with this work.
In chapter 4 the design, implementation, and generated results will be evaluated based on conditions set before its creation.
Finally, in chapter 5 the results will be summarized and possible improvements and extensions will be suggested. This will also present potential starting points for future works.

Chapter 2.

Foundation

This chapter focuses on research related to the given task. The related works described here create a base and provide guidelines on how to handle different aspects of designing a generator. Multiple different types of generators will be discussed to collect ideas for the design.

Types of generators

A possible, non-exclusive categorization of generators is to differentiate between *complex seeded generators*, *cryptographic random number generators*, and *scientific random number generators*. The first type of generators are *complex seeded generators*. Those are generators that utilize pre-defined seeds to generate data from. This approach can be used to generate complex models, for example interactions in living organisms in biological or medical contexts. This way complex models can be generated without completely synthesizing them, which would require larger amounts of computations.
Then there are random number generators, which can provide a base for other types of generators, or be used directly in computations. As the generator will be closely related to or employ random number generators, *cryptographic random number generators* will also be included in this chapter. These are used in security applications and consequently have additional requirements to be able to withstand attempts to compromise the applications relying on them.
Lastly there are *scientific random number generators*, which focus specifically on large-scale scientific application, requiring better performance and scalability. They may also include special algorithmic features to be better suited for specialized applications.

2.1. Complex Seeded Generators

SynTReN

An example for this class of generators is *Synthetic Transcriptional Regulatory Networks* (SynTReN), which is a generator for complex simulated gene expression data. It was introduced by Van den Bulcke *et al.* in *SynTReN: a generator of synthetic gene expression data for design and analysis of structure learning algorithms* [14]. Its goal is to create data sets which are as close to real-world data as possible.
This generator simulates complex interactions in living systems to produce usable data sets for the development of algorithms in the medical field, using networks as models for the interactions. Real networks are used as a source, from which the synthetic data sets are built in multiple steps. For this the source networks are first split into smaller structures and then linked in different ways through one of multiple different algorithms. From this network the generator collects samples which are then slightly modified by adding noise. Most of these parameters are user-definable or user-selectable such as the source, the size of the network, and the noise levels [14].
Some ideas in this design can be transferred onto the design of a generator for sensor data. Configurability should be included so different types of data sets can be generated faster and more comfortably. This makes it easier to use different sets of data to analyze the behavior of algorithms in various situations, especially in edge cases. Compared to biological networks such as genetic data, sensor data is not especially complex, and data generated by this generator does not need to be particularly close to real world data. This makes it easier to generate sensor data without the need to simulate a sensor network and use real seeds.
From these findings it can be concluded that the generator should not need to be complex and that it has to provide the user with means to configure its behavior.

2.2. Cryptographic Random Number Generators

Yarrow

Yarrow, a family of cryptographic random number generators designed by Kelsey *et al.* [6] is used here as a reference on high-quality pseudo random number generators. Implementations of this RNG were in use for a long time (e.g. up until Apple OS X 10.10 [5, 4], released in 2014), even though alternatives were available and its original authors had designed an improved successor [2]. In *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*, the authors discuss a specific implementation which uses 160 bits of entropy [6]. Entropy is the random data used for seeding and has to be collected through external means such

as mouse inputs or hardware entropy generators.
The goal when designing Yarrow was to create a new, secure cryptographic (pseudo) random number generator which is specifically built to withstand different kinds of attacks. The first important step was to differentiate which requirements a random number generator to be used for cryptography has to meet compared to one that is built for general, non-critical use.
Pseudo random number generators used in cryptography should generate unpredictable numbers. This means the algorithm must produce number sequences in which the singular numbers are as unrelated to each other as possible, so they are hard to predict even with large amounts of computational power [6].
Yarrow tries to achieve this by combining two different methods of transforming numbers: a hash function and a block cipher. The hash function has to be one-way and it should not be possible to force collisions. The block cipher should also withstand known- and chosen-plain-text attacks and its output should have good statistical properties. In addition to this, the estimation of entropy is another essential part of the generator, as overestimating it can substantially weaken the generated output [6]. The authors claim that the pseudo random generator is secure if those requirements are fulfilled and the generator has enough entropy to start with. Yarrow-160 employs SHA1 as hash function and 3DES as block cipher [6].
Since the generator to be designed is not security-related or critical, it is unnecessary to guarantee the security requirements for cryptographic random number generators. However, the idea of transforming a number using cascading methods is something that can be useful to create outputs that deviate from the norm far enough to be useful for testing purposes.

2.3. Scientific Random Number Generators

Java RNG for Grande

Interfaces and implementations of random number generators for Java Grande applications discusses how to generate random numbers for use in large-scale scientific projects [1]. It puts special emphasis on generating numbers usable in Monte-Carlo simulations, which produce results by repeating simulations a large number of times with randomized input. Four common algorithms are differentiated: *linear congruential generators* (LCGs), *lagged Fibonacci generators* (LFGs), *shift registers*, and generators which use a combination of the aforementioned. LCGs utilize an initial seed X_0 and generate numbers from this with

$$X_{n+1} = (a * X_n + c) \bmod m$$

where X_n is the n-th number in the sequence. The relations between the natural number constants a, c, and m determine the period length after which the sequence of generated

numbers will be repeated [7]. LFGs employ a generalized version of the recurrence relation describing the Fibonacci sequence. They also operate on numbers inside a ring, yielding

$$X_n = (X_{n-j} \odot X_{n-k}) \bmod m$$

where $\odot$ is any binary operation. X_n again denotes the n-th number in the sequence and the constants m, j and k are natural numbers. Shift register generators, for example *linear-feedback shift registers*, produce output similar to LCGs [1]. Using registers with feedback mechanisms, they combine parts of the previous output to generate a following number.
In the work it is stated that the programming language Java offers some advantages in regards to random number generation [1]. Since it is a popular programming language used in many applications, a large number of libraries exist which provide different algorithms that may be useful for scientific purposes. Furthermore Java provides transparent behavior and precision for numeric data types, which makes implementations portable and removes inconsistencies.
The authors also provide specific information on its provided RNGs. Java provides three classes for random number generation: *SecureRandom*, *Random*, and *Math*. Of these, *Math* is just a wrapper for *Random*, thus only two different classes are available. *SecureRandom* is an interface for generation of cryptographically secure numbers similar to those discussed in 2.2 [8]. The authors note that the specialized nature of cryptographic random number generators like this render them comparatively slow, which is undesirable in large-scale simulations. They also add that the possible use of hardware generators and non-linear algorithms in *SecureRandom* makes results non-reproducible. *Random* produces pseudo random numbers using an LCG and is versatile in its output. This is distinguished as a positive aspect but the 2^{48} period size of the output is said to be too small for large simulations such as Monte-Carlo simulations [1]. This indicates that this might be a fitting supplier for random numbers for smaller to mid-sized projects.
Another aspect the authors mention is repeatability, which they provide by using state-saving [1]. This grants different possibilities like verification of simulation results and testing the impact of algorithmic changes. As those features are desirable for the intended use cases of the new generator, those guidelines should be taken into account.

2.4. Conclusions

The first conclusion to gain from *Complex Seeded Generators* is that configurability of the generator is essential. Another finding is that because of the relative simplicity of the required models, simulation is not necessary to generate appropriate outputs.
Upon reviewing *Cryptographic Random Number Generators*, it has become clear that

security features are not required in the targeted use case and would constrain the design of the generator in unfavorable ways. Omitting these constraints means that the generator may gain performance improvements and avoid the need to introduce complex algorithms. Additionally, the practice of generating and modifying data using cascading methods may prove helpful in the design of the generator.
Furthermore the section *Scientific Random Number Generators* shows the need for repeatability, which in this case is implemented by using state-saving. In this section it is also stated that Java, due to its versatility in regards to random number generation, may be an appropriate language to use for an implementation of the generator. The Java *Random* class should be taken into consideration as a relatively fast and adaptable provider of pseudo random numbers.

Chapter 3.

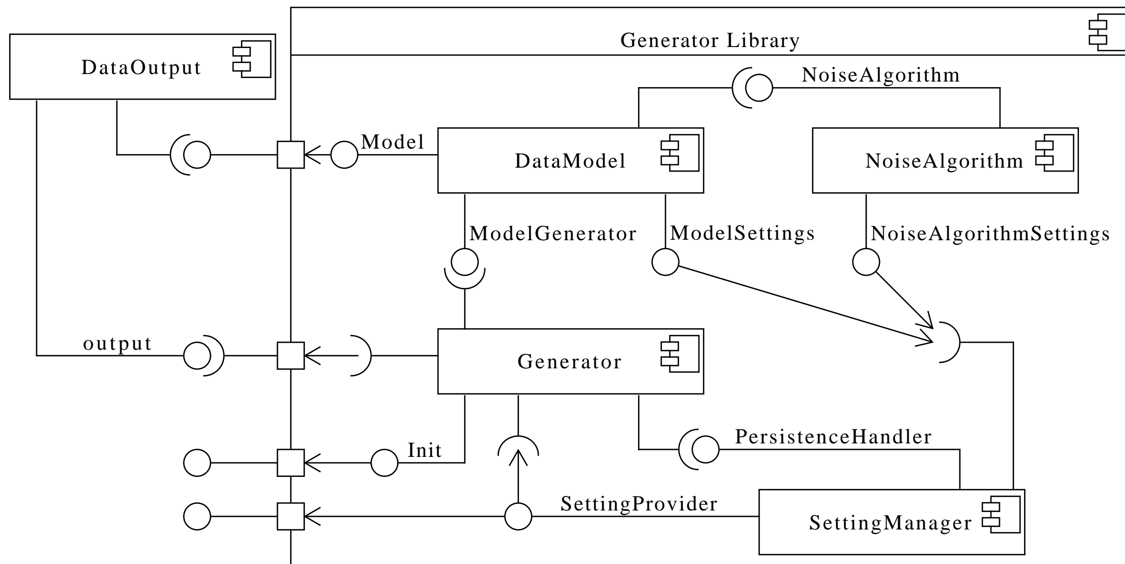
Design and Implementation

This chapter presents the specifics of the proposed design and explains decisions based on earlier research. The first section introduces the general design of the generator and validates decisions based on earlier research. It is accompanied by a component diagram of the design to provide a condensed synopsis of the layout, portrayed in figure 3.1. Then an overview of a possible implementation in an object-oriented language is given and the process of generation is portrayed, after which some additional design choices will be highlighted and explained. In addition to this, the reference implementation is introduced and specific obstacles and choices made in the process are explained.

3.1. Design

Based on the non-functional requirements defined in chapter 1 and findings from chapter 2 it is now possible to create the generic design of the generator. The design aims to avoid imposing any unnecessary regulations onto possible implementations while still providing an extensive framework with comprehensive support for different use cases. Therefore it intends to be as simple as possible with clear separation of responsibilities. Since the generator should generally be assumed to work in combination with other applications, it has to provide mechanisms to allow simple integration into existing appliances. To ease this interaction with applications the generator itself will be a modular library that can be directly incorporated into applications or built into a standalone solution by supplying it with only a few additional components.
The library consists of four components which have defined interfaces for communication. These components can be modified or exchanged with different ones as long as they provide the required functionality defined in the interfaces. This makes the whole application adaptable and easily extensible. The four components are *Generator*, *DataModel*, *NoiseAlgorithm*, and *SettingManager*. In addition to this the *DataOutput* has to be provided from the enclosing application.
Generator is the central component that is responsible for controlling the flow of the library. It provides ways to initialize all components and start the process of generation, and requires knowledge of the other components to do so. It needs to have access to the

Figure 3.1.: Generator Design Diagram



concrete generator of the *DataModel* to be able to produce and output the filled models to the *DataOutput*.
The *DataModel* provides the model to the *DataOutput*, the concrete generator to the *Generator*, and access to the settings as a way to configure the data that will be generated. Single data points are generated with the *ModelGenerator*, while *Generator* oversees the process of generating the user-configured amount and types of outputs.
For this the *DataModel* requires access to the *NoiseAlgorithm*, which is used to transform the data in the output models to achieve sufficient variance in the whole generated data set. This applies the concept of cascading methods found in section 2.1 onto the issue at hand. The *NoiseAlgorithm* also has to provide the *SettingManager* with an interface to configure it.
SettingManager is the component that provides configurability of the generator to the external application, and ultimately to the user. It provides methods to modify and retrieve settings and can accept and handle arbitrary settings for the *DataModel* and *NoiseAlgorithm* components. The library provides external access to the *SettingManager* so the enclosing application can configure it ahead of time. Additionally *SettingManager* provides a *PersistenceHandler* to store and load the configuration, including settings provided by other components. With this, all configuration is stored in the *SettingManager* component, and saving and retrieving it is possible on request. This makes it possible to reproduce results as long as all components exclusively rely on settings stored inside *SettingManager* and do not utilize any externally controlled sources for data.

3.1.1. Supplementary Instructions

In addition to this general design, there are guidelines the implementations need to follow to ensure the results satisfy the given requirements.
Following the same principle as the *IEEE TEDS* standard (IEEE 1451.4) [3], implementations of *DataModel* should be self-describing.
All information stored inside such an object has to be accompanied by some instruction on how the information is to be interpreted. As noted previously, implementations should not utilize inputs that require settings which are not defined in the *SettingManager*, as this would impede the reproducibility of results. For the same reason, implementations may not employ true random number generators. Settings for the generator should only be changeable through the *SettingManager* to ensure that all components receive consistent settings. Therefore, all components also have to query settings exclusively from the manager.
To simplify the process of building arbitrary *DataModel*s and *NoiseAlgorithm*s, implementations should employ factory patterns. As no particular procedures for data generation are specified, the quality and possible unique size of the output depends on the actual implementation.
This design omits the aspect of transportation and does not include any simulation to generate data, though implementations of *DataModel*s may choose to do so. It should be noted, however, that incorporating simulation into the generation of data will most likely have impacts on the performance of the library.

Performance of Implementations

Since the application should be able to generate large numbers of outputs, performance aspects should also not be disregarded. One way to improve performance is *concurrency*, assuming that splitting the work of generating the output can be done without negative side-effects. Since concurrency is based on making processes independent from each other to be able to execute them autonomously, it could be difficult to ensure that consecutive runs of the generator with constant settings produce identical results.
From this it can be concluded that including concurrency in data generation could possibly require a large amount of work and the extend of its profits is questionable. Implementations of the generator may instead focus on reducing the amount of computation required to generate a data object.

3.2. Object-Oriented Application in Java

Because the general design cannot fully ensure that the generator satisfies the given requirements, this section presents an application of the design and guidelines in a popular

object-oriented language. Many parts of this should be trivial to port to other object-oriented languages.
The choice of Java for this task is based on a few different factors which are detailed in section 2.3. Given that the referenced work recommends Java in part for providing suitable suppliers of pseudo-random numbers, some additional research on this matter seems appropriate.

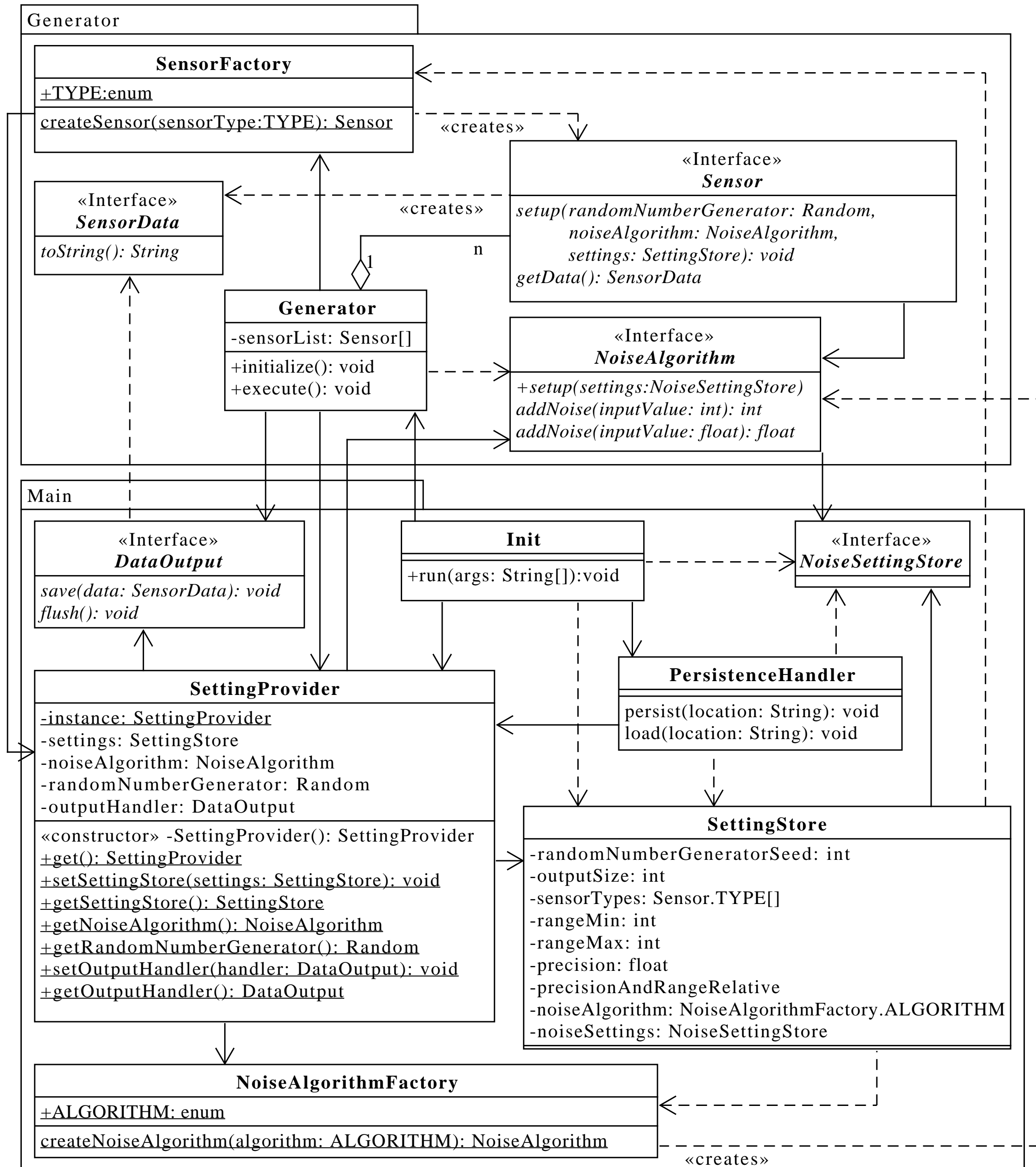
3.2.1. Java 8 API: Random

The *Java™ Platform, Standard Edition 8 API Specification* should provide more information on this. *Random* uses a single LCG with a 48-bit seed, producing output with a period length of 2^{48}. The output can have up to 32 pseudo random bits and can be supplied as one of several different data types [1, 8]. This means that *Random* will provide the same sequence of numbers when initialized with the same seed, making repeatability possible by simply storing the seeds. The period length of the generator should be large enough for the intended use-case, as indicated earlier [1].
Since the LCG used in *Random* is only based on multiplication, addition, and modulo operations on integers, it should also perform well compared to more complex random number generators. It should also be an appropriate choice if an implementation chooses to employ *concurreny*, as it is *thread-safe*. The class is, however, affected by contention, potentially mitigating some of the benefits of concurrency [8].

3.2.2. Implementing the Design in Java

To correctly interpret the class diagram in figure 3.2 it should be discussed how the different aspects of the design have been transferred onto Java.
The library is split into two packages: *Main* and *Generator*. The *Main* package is made up of classes used for handling settings and high-level initialization, while the *Generator* package includes everything participating in the actual generation of output data.
The *Generator* component is realized through the *Generator* and *Init* classes. They are responsible for initialization of the library and controlling the other components. If command-line arguments are to be provided, they are interpreted and handled by *Init*. *DataOutput* provides an interface for *DataOutput* component, where the generated data objects will be handed to for further processing by external applications. The *SettingManager* is implemented by *SettingProvider*, to manage configuration and retrieval of settings, *PersistenceHandler*, and *SettingStore*, which holds the application settings and is able to store additional settings objects. *SettingProvider* stores the *SettingStore* alongside derived settings and the *DataOutput*. *Sensor* and *SensorData* represent the *DataModel* component, where *Sensor* is used to generate single *SensorData* objects. Settings for the *DataModel*, in the design represented by *ModelSettings*, are directly incorporated into the *SettingStore*. The *NoiseAlgorithm* component is represented by the

Figure 3.2.: Generator Class Diagram in Java

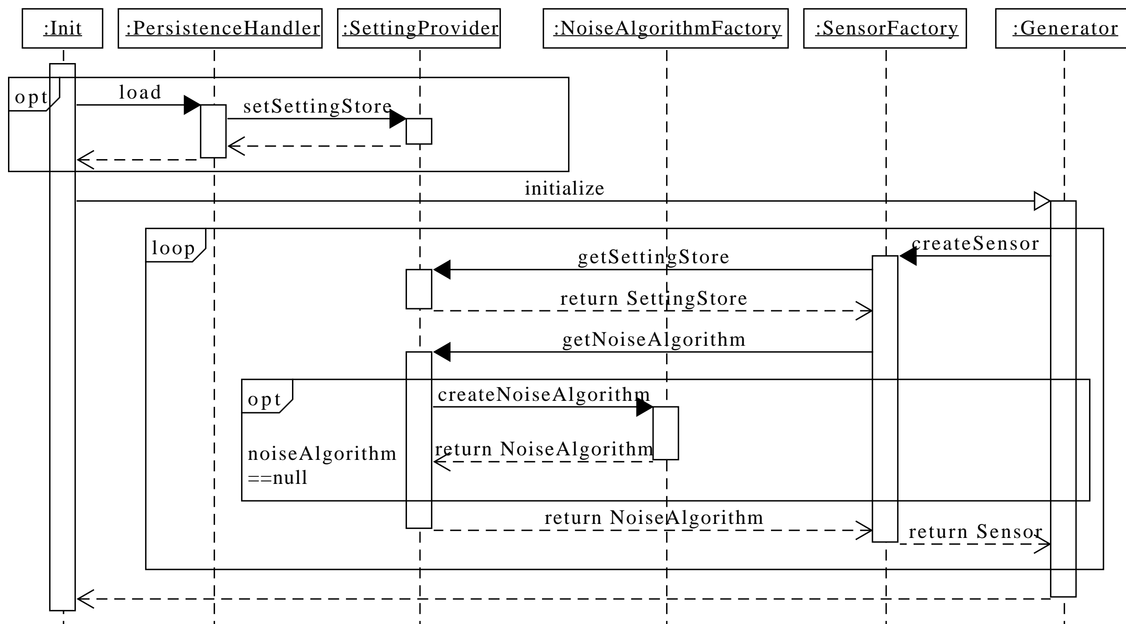


interface of the same name and *NoiseSettingStore*.
Both *SettingStore* and *NoiseSettingStore* hold primitive settings that can be arranged by the user to influence the output data. A factory pattern is used to create *Sensor* and *NoiseAlgorithm* objects.

3.2.3. General Process

The process is split into two stages: *initialization* and *data generation*.
The initialization is described by the sequence diagram in figure 3.3. Before starting the generator it is possible to configure it through the exposed *SettingProvider*, *SettingStore*, and *NoiseSettingStore*, the latter of which depends on the *NoiseAlgorithms* provided by the specific implementation. A *DataOutput* needs to be set in *SettingProvider* before starting, since the data cannot be transferred to the application without it.
The entry point to the library is the *Init.run* method, which processes command-line arguments before launching the generator. When *run* is called, *Init* may initialize the required settings objects depending on current availability and given parameters. Then it calls *Generator.initialize*, which sets up all needed objects according to the settings obtained from *SettingProvider*, after which the initialization of the library is complete.

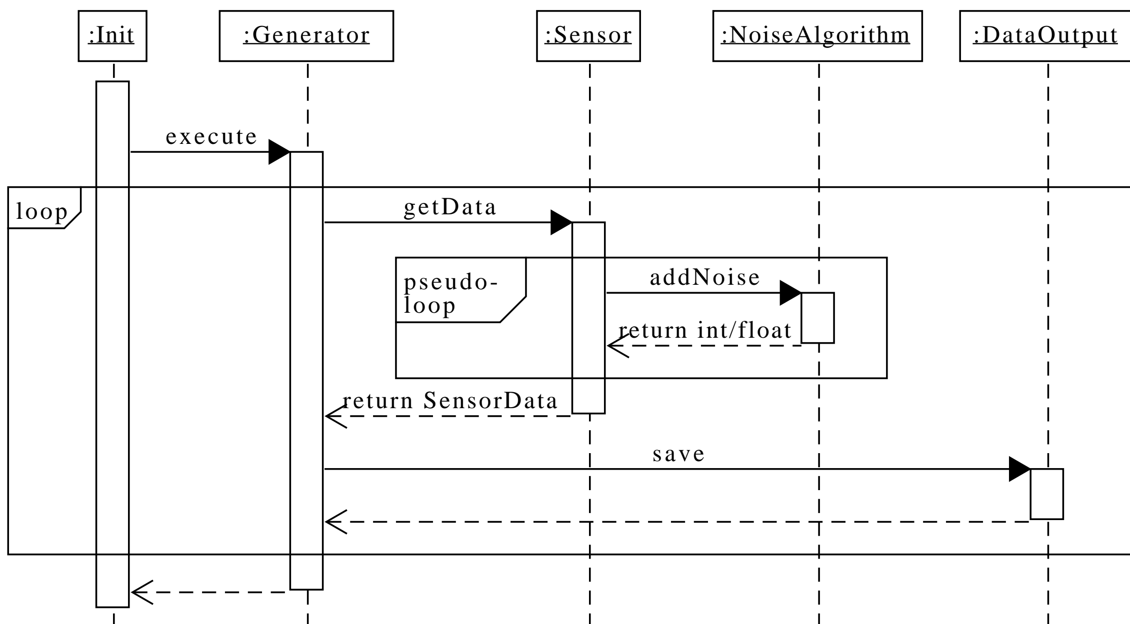
Figure 3.3.: Initialization Sequence Diagram



After the initialization *Init.run* calls *Generator.execute*, handing over control to the functional part of the library. The *Generator* will then use the created *Sensors* to generate

SensorData. Each *Sensor* generates data corresponding to the type given to *SensorFactory* when creating it. When *Sensor.getData* is called, a *SensorData* object is generated, applying the pre-defined *NoiseAlgorithm* where appropriate. This process is depicted as *pseudo-loop* in figure 3.4. The object is then returned to the *Generator* that transfers it to the *DataOutput*. This is repeated according to user specifications, after which *DataOutput.flush* is called to force processing of possible non-empty buffers to prevent data loss. Afterwards the library hands back control to the execution environment. The whole process is described in figure 3.4.

Figure 3.4.: Generation Sequence Diagram



3.2.4. Rationale

As this is only a general proposition of how the design could be implemented in Java, it aims to provide the largest possible amount of flexibility to the actual implementations. To achieve this, central elements such as the data model are not explicitly defined but rather provided as a lowest common denominator.

Generation

Consequently, *SensorData* only requires implementations to provide a universal way to retrieve data from it. The internal layout of classes implementing it is completely up to different implementations to decide, making it very flexible and easily possible to integrate new types of output data into the application. This means that *Sensor*s have to be implemented to generate the specific type of data while still providing the *getData* method for *Generator* to obtain the synthesized data. Therefore, a *Sensor* should store all variables and objects required to create a complete *SensorData* object, which may differ between types. This should include precision, minimum, and maximum values the output should typically have but also possibly other user-configurable variables. As the design is built around the concept of employing cascading methods for generation, the *Sensor* needs to know some object that provides pseudo random numbers to get basic numbers which can then be transformed as needed. This object is obtained through the user-provided settings.
A tool for said transformations is a *NoiseAlgorithm*, which it also receives upon creation. An implementation of *NoiseAlgorithm* has to provide methods to modify different numeric types which might be used and should also take advantage of method overloading to reduce complexity. It also has to accept a settings object to configure itself according to user specifications.
The *Generator* stores a list of *Sensor*s that are used to generate the data. This list is filled when *initialize* is called, where the *Sensor*s are created using the *SensorFactory* which uses the appropriate subclass of *Sensor* for each given type. When *Generator.execute* is called, it loops through the *sensorList* to request output data and then pushes it to the *DataOutput*, which means giving it over to the parent application. This is repeated a user-defined number of times, after which the library terminates.

Configuration

For all of this it is important that the user is able to define the behavior of the library as much as possible.
This is done by configuring settings inside a *SettingStore* object. It enables modification of the amount of output objects to be generated and the types of *Sensor*s to use for this. In addition to this the seed to be used for the generator providing base numbers to *Sensor*s, the typical range for outputs, and the precision can be set. The user can also decide if *Sensor*s should map the range and precision to *sane* numbers for each specific type they represent. This is done instead of utilizing a separate *SensorDataSettings* object since it reduces complexity when parsing settings. Most common types of sensors should be easily representable in a way that only requires those three primitive settings. Finally the *SettingStore* provides the option to select an implementation of

NoiseAlgorithm to use for mutation of output numbers.
The unique settings for this object are stored inside a *NoiseSettingStore*. This interface is only used to reference the object and has no required content as the settings completely depend on the particular implementation of *NoiseAlgorithm*. Settings are accessible through *setters* and *getters*, which are absent from figure 3.2 due to being of trivial nature and occupying too much space.
The *SettingStore*, and transitively the *NoiseSettingStore*, are stored inside a *SettingProvider* object. This class provides access to all settings to every object in the application. Additionally it provides access to complex objects derived from some settings inside the *SettingStore*. These objects are the pseudo random number generator *Random*, which provides base numbers in *Sensor*s, and the *NoiseAlgorithm*. The former of these is created by the *NoiseAlgorithmFactory*, which initiates the correct object according to the given parameter and settings stored in *NoiseSettingStore*. Finally, the provider also handles setting the *DataOutput* to push generated data to. The parent application has to hand the output object to the provider through *setOutputHandler* before the initialization takes place, else the *Generator* should fail. The instance of *SettingProvider* that holds the settings can only be created and accessed through the static methods of *SettingProvider*, making this a *singleton*. This implements the central management of settings as per instructions in 3.1.1.
Initialization is started by the *Init* class. Before calling the *Generator* it should process parameters, which could be command-line switches, and create and populate Settings objects as necessary. This class should also pass options regarding persisting of settings to the *PersistenceHandler*. It is responsible for saving and retrieving settings to and from persistent storage. For this it only needs to serialize the *SettingProvider* together with the settings objects contained in it.

3.3. Reference Implementation

The reference implementation provided with this work is based on the groundwork done in section 3.2. Full class diagrams of it can be found in appendix A *Additional Implementation Diagrams*.
The implementation is split into three subprojects: *core*, which is the generator library; the *DataOutput* component, *fileOutput*; and a single class that combines both into a *standalone* application.
The *core* library expands on diagram 3.2 by implementing a *DataModel* and a *NoiseAlgorithm*. The provided model component consists of *SimpleSensorData* and *TemperatureSensor*, which generates objects containing temperature data in °C or °F. The *currentValue* is generated by fetching a number from the *Random* object and transforming it with the stored *NoiseAlgorithm*. The unit of the output value is also randomly decided and if necessary the value is converted.
SimpleSensorData is a generic model, only containing a float value, a precision range in

float, and a unit string. Consequently it may hold many different types of data, even though it is currently only used for temperature data.
The implemented *NoiseAlgoritm* is *RandomNoiseAlgorithm*, which will add random noise to values based on user configuration. When given a number, it uses a separate *Random* object to decide if noise should be applied. If this is true, it adds or subtracts a randomly generated value inside a specified range.
In this implementation *Generator.initialize* is split into smaller parts which are executed on initialization of the object.

Figure 3.5.: Serialized Default Settings

```
!Main.SettingStore
noiseAlgorithm: RANDOM
noiseSettings: !Main.RandomNoiseSettingStore
   noiseFactor: 0.5
   range: 5
   seed: 98653489
outputSize: 1000
precision: 0.5
precisionAndRangeRelative: true
randomNumberGeneratorSeed: 2345876
rangeMax: 100.0
rangeMin: 0.0
sensorTypes:
- TEMPERATURE
```

This implementation uses *YamlBeans* [12] to handle serialization and deserialization of settings objects. This produces readable, easily modifiable settings files in *yaml* format, containing the full object graph of *SettingStore*. Figure 3.5 shows the serialized representation of the default settings of the application.
In this, lines 3 to 6 represent the *RandomNoiseSettingStore* contained inside the *SettingStore*.

- *noiseFactor* controls the probability of random noise being added to a value
- *range* is the one-sided range of the noise to add to or subtract from the raw value
- *seed* is the seed that is used for the *Random* object of *RandomNoiseAlgorithm*
- *noiseAlgorithm* sets which class should be used for noise
- *sensorTypes* is a list of *Sensor* types that should be used by the generator

- *rangeMax* and *rangeMin* set the range in which *Sensor*s are allowed to generate raw, noiseless data
- *precision* sets the standard precision *Sensor*s should write into generated data objects; this value is currently not modified inside the *Sensor*
- *precisionAndRangeRelative* allows the *Sensor*s to map the aforementioned user-set values to *sane* values on creation, depending on the type of the *Sensor*; *TemperatureSensor* centers the range delta around 25°C if this is set
- *randomNumberGeneratorSeed* is the seed that will be used for the *Random* object used by *Sensor*s
- *outputSize* defines how many data sets should be generated

The settings file can be saved and loaded using command-line parameters. The standalone application currently supports a few different parameters.

- *--help, -h* prints a list of available commands
- *--outputsize <long>, -n <long>* sets the amount of output that should be generated; default: 1000
- *--output <string>, -o <string>* sets the file the output will be written to; default: "output.txt"
- *--buffersize <long>* defines how many lines should be stored in the buffer at a time; default: 500
- *--save, -s* writes the configuration to a file
- *--load, -l* loads the configuration from a file
- *--settinglocation <string>* defines which file to use for configuration reads/writes; default: "settings.yml"
- *--createconfigonly* makes the application exit after writing the settings file

The *fileOutput* subproject implements the *DataOutput* interface and writes the output to the given location. It implements its own *FileOutputSettingStore* and uses a *FileOutputBuffer* to improve performance by aggregating disk writes. The buffer uses a *Timestamp* object to generate timestamps for data objects. One of these is provided by *SimpleCounterTimestamp* which simply returns incrementing ticks as timestamps.
The project employs *unit tests* where possible to ease development and prevent regressions.

Chapter 4.

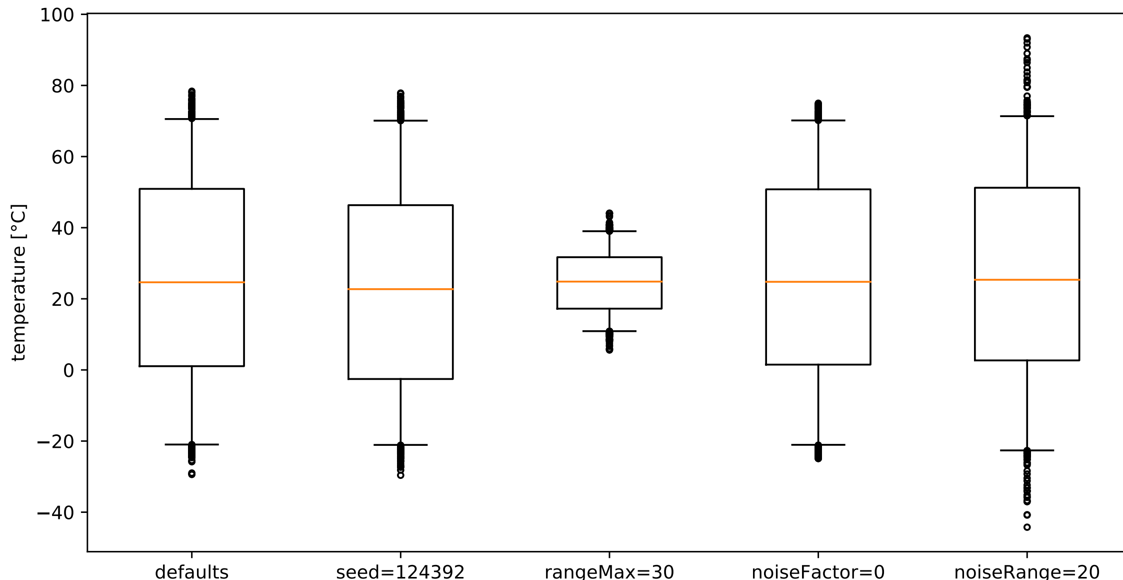
Evaluation

In the following sections the resulting design and the standalone implementation in particular will be evaluated based on a few of the important factors defined in the beginning.

4.1. Adaptability/Expandability

It was stated that the generator should be easily expandable and adaptable to be useful in different contexts. In the presented reference implementation three different components are expandable through relatively small changes.
Extending core functionality is simple and only requires the creation of one or two new classes and adding very few lines of code to the existing structure. First and possibly most importantly, introducing an additional *DataModel* requires few changes: If the intended output type can be represented as *SimpleSensorData* objects, only a new *Sensor* has to be created. This also has to be registered in the *SensorFactory* by adding a *TYPE* and expanding the *switch* statement in *createSensor*. Depending on the planned contents of the output it may be required to create another *SensorData* class and introduce additional settings into the *SettingStore*.
Creating a new *NoiseAlgorithm* can be done in a very similar way. For this, a *NoiseAlgorithm* class and a *NoiseSettingStore* class have to be implemented and the algorithm has to be registered in the *NoiseAlgorithmFactory*.
This also means that existing code can be easily adapted as components are largely independent from each other.
Adding a different *DataOutput* is also quite simple, as the interface is small and an object of the new implementation is simply given to the *SettingProvider* before calling *Init.run*. The required effort required for the implementation of the *DataOutput* itself cannot be evaluated as its complexity depends on the intended functionality.

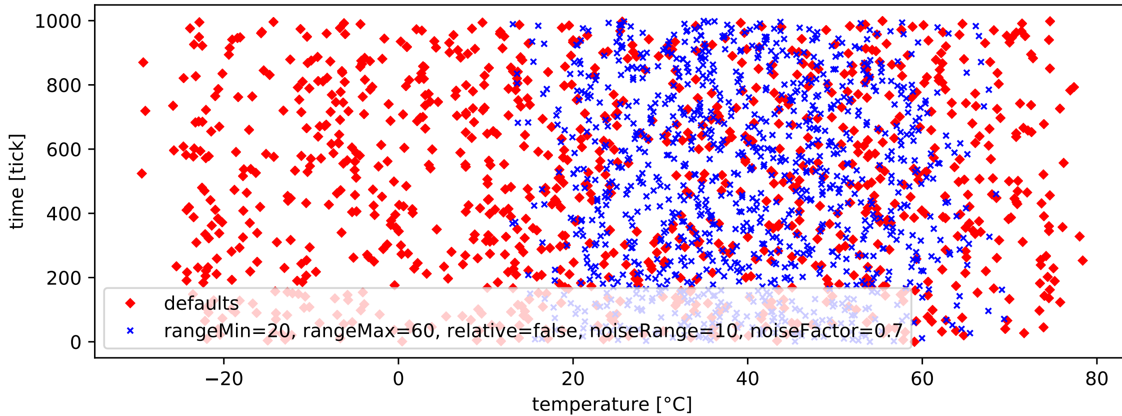
Figure 4.1.: Impact of Single Configuration Changes



4.2. Configurability and Distribution of Outputs

Furthermore it was emphasized that it is important for the user to be able to control many aspects of the output that will be generated. Figure 4.1 shows how specific settings impact the range and distribution of temperature values. The boxes show the second and third quartile with whiskers at the 5 and 95 percent marks. Each non-default plot has only one changed setting relative to defaults.
The diagram clearly shows how the *NoiseAlgorithm* influences outliers, while the range delta constrains the overall output range. This shows how the *NoiseAlgorithm* can be used to simulate faulty outputs and unreliable sensors. With this, the user may generate data sets with differing amounts of *erroneous* outputs, which may be useful to test robustness of algorithms regarding unexpected inputs.
Changing the seed does not meaningfully modify the range and distribution of outputs but scatterplots such as figure B.3 show that it produces completely different data sets. The ranges displayed in figure 4.1 are all roughly centered around 25°C because the *TemperatureSensor* adjusts the range delta around this value, as explained in section 3.3. Disallowing this lets the user freely define the region the outputs should be located in, which is clearly visible in figure 4.2.
Overall, the user has a lot of control over the outputs. On the one hand this is exhibited in changes of *location* and *width* of the set of numbers, as can be seen in figure 4.2. On the other hand it is also apparent in changes of *probability distribution*. An example of

Figure 4.2.: Impact of Different Configurations



this is given in figure B.1, showing changes from a nearly uniform distribution to a more continuous rise and fall on the edges, which can be attributed mostly to changes in noise settings.
These results demonstrate wide control over the generator, which should prove useful in diverse applications. Additional scatterplots showing changes in *randomNumberGeneratorSeed* and *noiseFactor* are presented in figures B.3 and B.4.

4.3. Repeatability

It was also demanded that the application generates repeatable results. With the presented implementation it can be guaranteed that multiple runs with identical settings will generate identical results.
As the official documentation states, the *Random* class itself fulfills this condition [8]. Since the only external inputs used in the process of generation itself are two *Random* objects, only their settings and the variables used to transform retrieved numbers need to be saved. The *SettingStore* object contains all seeds and non-derived variables used in the generation, so with the application being able store the object on disk, the sequence of instructions can be repeated as desired. This holds true as long as no extension of the library introduces variables that are retrieved from new external sources and concurrency is not implemented, which is discussed in 3.1.1.
These findings have also been confirmed in practical tests. An indication of this can be seen in figure B.4, where small changes in noise settings generate largely identical data sets in the same order.

4.4. Maximum Amount of Outputs

It is also imperative that the application is able to produce amounts of that are helpful in its intended use-cases.
How many non-periodic outputs can be generated largely depends on the period length of *Random*. Since it employs an LCG, which uses only the most recent output to generate the next one, the outputs will be periodic from the point at which a value is generated a second time. *Random* uses 48 bits in its LCG and generates a full period of 2^{48} numbers [1].
The next important factor is how many random numbers are requested per data set, as the periodic repeating of numbers might begin at a different point in the the overall algorithm. The generator uses three different methods of *Random*: *nextInt(bound)*, *nextFloat()*, and *nextBoolean()*. These methods internally call *next()*, which is the method implementing the LCG [8, 9]. As the official documentation and the source code of the *OpenJDK* platform show, the three methods may request different amounts of random numbers using *next()* [8, 9].
Because other cases are not trivial to calculate due to loops in the code, it is from here on assumed that only powers of 2 will be used as input for *nextInt(bound)*. This requires both *noiseRange* and the delta between *rangeMax* and *rangeMin* to be powers of 2.
Under this assumption, all three methods request one random number per call. The application uses two independent *Random* objects for raw numbers and noise, which both dispatch three calls to *next* per *SensorData* object. This means that both run synchronously in regards to the period of generated numbers. With

$$2^{48} = 281474976710656$$

and its sum of numbers

$$73 \bmod 3 = 1$$

not being divisible by 3 without remainder, the period will occur with an offset of one each time.
From this it can be said that the application is theoretically able to produce

$$3 * 2^{48} \approx 8,5 * 10^{14},$$

or around 850 trillion unique output objects from a single configuration before repeating. The sequence of outputs is also unique to every configuration of values that control the behavior of the *Random* objects. However, this assumes that outputs have large enough precision so these numbers can be represented without any rounding taking place.
Different data sets can be produced by changing any setting, amplifying the number of possible outputs, especially in relation to the order in which they will be generated. This should provide sufficient proof that the application can be used to provide large enough data sets to be useful in its intended field.

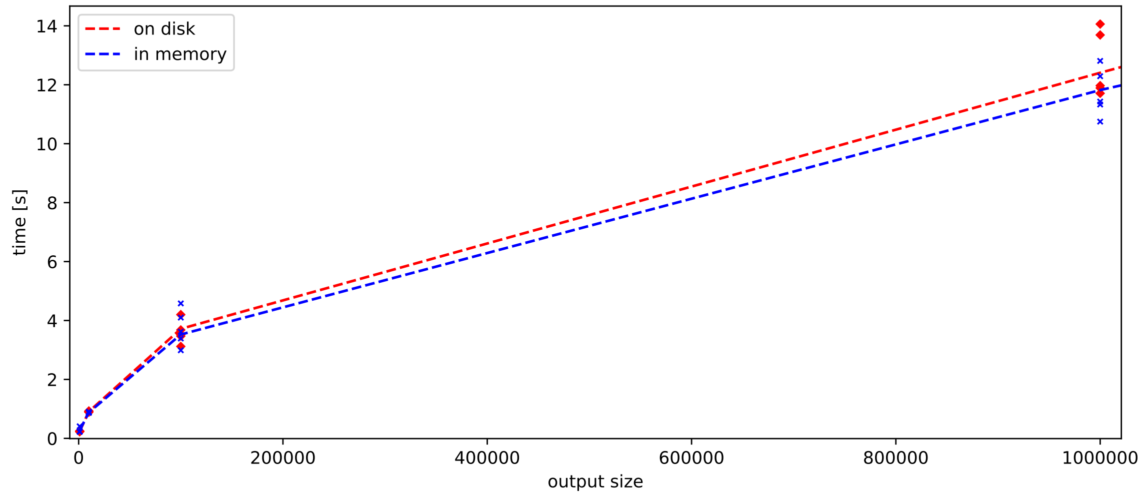
4.5. Performance

Since the generator will be used to generate large amounts of objects and may be used inside applications which process these objects directly, it is also important that generating objects is adequately quick. This was tested using simple methods with the standalone application considered as a black box.
For this, the *GNU Time* [11] utility was used in a script to measure the durations to generate different amounts of objects. The script can be seen in figure B.5. To attain a somewhat reliable average, each configuration was run five times. To measure the impact of disk writes, testing was done twice. Once with an output file located on disk and once inside a temporary file system inside the main memory. Aside from output size test runs used the default settings, which, among other things, set the file output buffer to 500 objects. No other programs on the system were accessing the relevant files or causing unusual system load while testing.
There are a number of other relevant environmental factors, that need to be known to appropriately evaluate the results:

- *CPU:* Intel Core i5 3570K @ 3.5GHz
- *RAM:* 8GB DDR3-1600
- *Disk:* OCZ Vertex3 Sata 3 SSD
- *File system on disk:* ext4
- *OS:* Arch Linux, Kernel 4.10.4
- *Java Platform:* OpenJDK (Runtime 1.8.0_121_b1; JVM 25.121-b13 64 bit)

Figure 4.3 shows test results up to 1 million objects and results up to 10 million objects are available in figure B.6.
Results use the sum of system and user cpu time, as this should be a better indicator for performance than real time measurements. The generation of 1 million and 10 million objects took around 12.7 and 97.4 seconds of cpu time when writing to disk, or 11.7 and 82.1 seconds respectively when writing to memory. This seems to indicate that increasing the buffer size to a more appropriate number should yield improvements.
However, the results show differences of up to nearly 20 percent when generating larger amounts of data, which is most likely owed to imperfect testing and additional external factors. Nevertheless these results should roughly demonstrate sufficient performance of the generator for the intended moderate use cases.

Figure 4.3.: Performance Testing Results



4.6. Summary

All of these evaluations show successful realization of a product that meets the predefined conditions.
The design itself is not overly complex and easily adaptable, while still providing extensive information for implementations. The application built from it is able to produce appropriate amounts of data that can be configured to exhibit different properties and it does so in a reliable and timely manner.
This showcases a result that is useful and well suited for the given task.

Chapter 5.

Conclusions and Future Work

In the beginning the focus of this work was laid on creating a generator for sensor data that is flexible, expandable, and suited to aid in the creation of new algorithms and applications for the IoT. The output of the generator was also demanded to be consistent so it can be used without requiring regular adaption of other applications.
Further research into related works showed that the generator would not need to be complex or provide security features and that no simulation would be necessary to synthesize data. It also became apparent that configurability and modularity would have to be basic principles the design should be based on.
In the presentation of the design, additional instructions were given to cover elements which could not sufficiently be described by the design alone. This design was then successively transferred into an implementation to further explain specific decisions and how different aspects of it help in achieving the intended results.
Finally, the reference implementation was evaluated based on the previous demands, where it showed positive results in performance, output scale, consistency, and generally fulfilling expectations set before creation.
With these results the design and implementation discussed here should prove useful in future works and aid in the development of new appliances.

Future Work

There are multiple ways for future works to expand on the results of this paper. The implementation can be expanded by implementing new data models and noise algorithms, possibly employing other algorithms such as gaussian noise to achieve various distributions. In addition to this, new ways to process generated data can be made available through local services or even web APIs. Another area for expansion are settings, which could be provided using a GUI or additional command-line options. In general, the modularity of the design allows for many possible additions and adaptations.

Appendix A.

Additional Implementation Diagrams

Figure A.1.: core Implementation Diagram

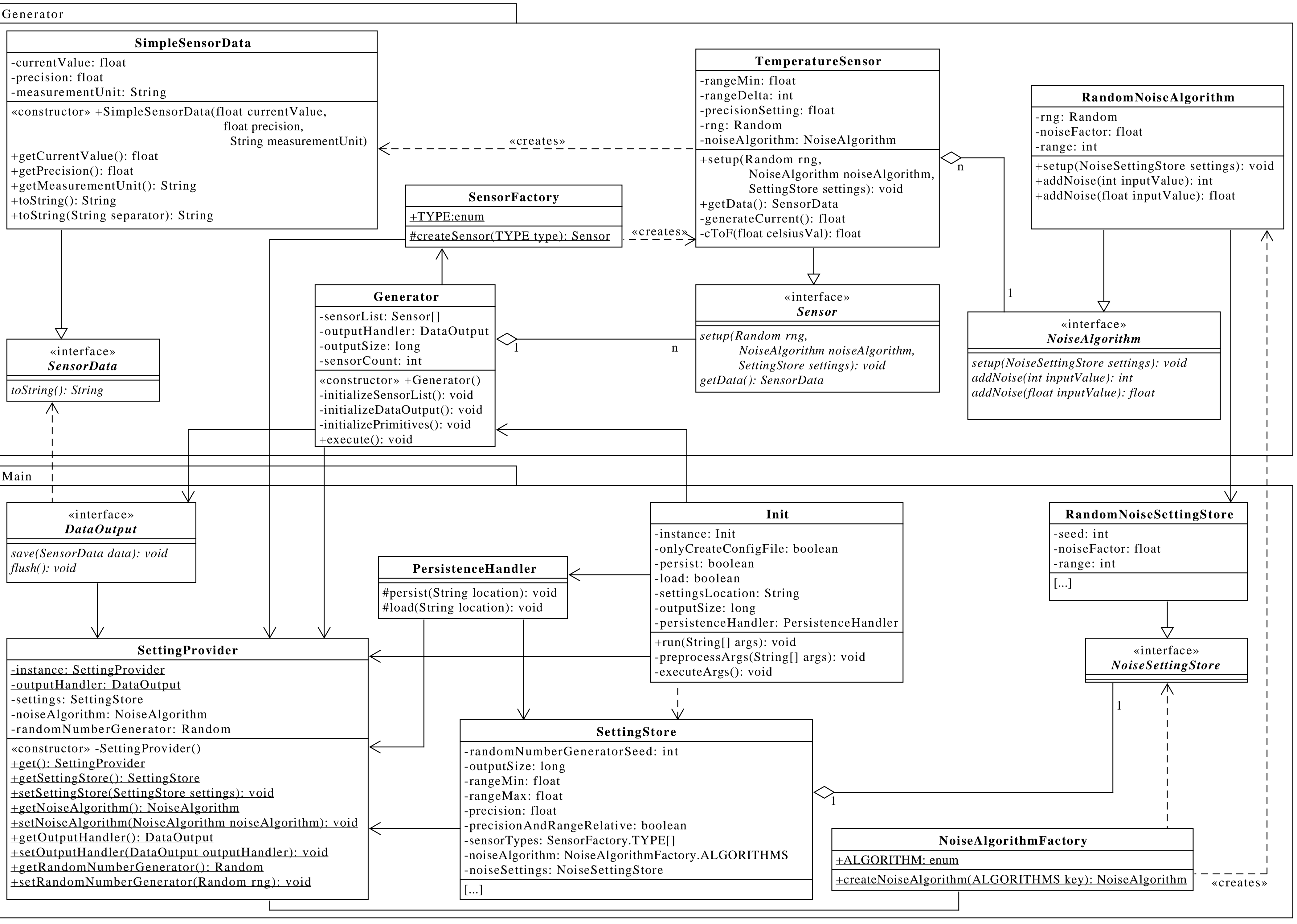
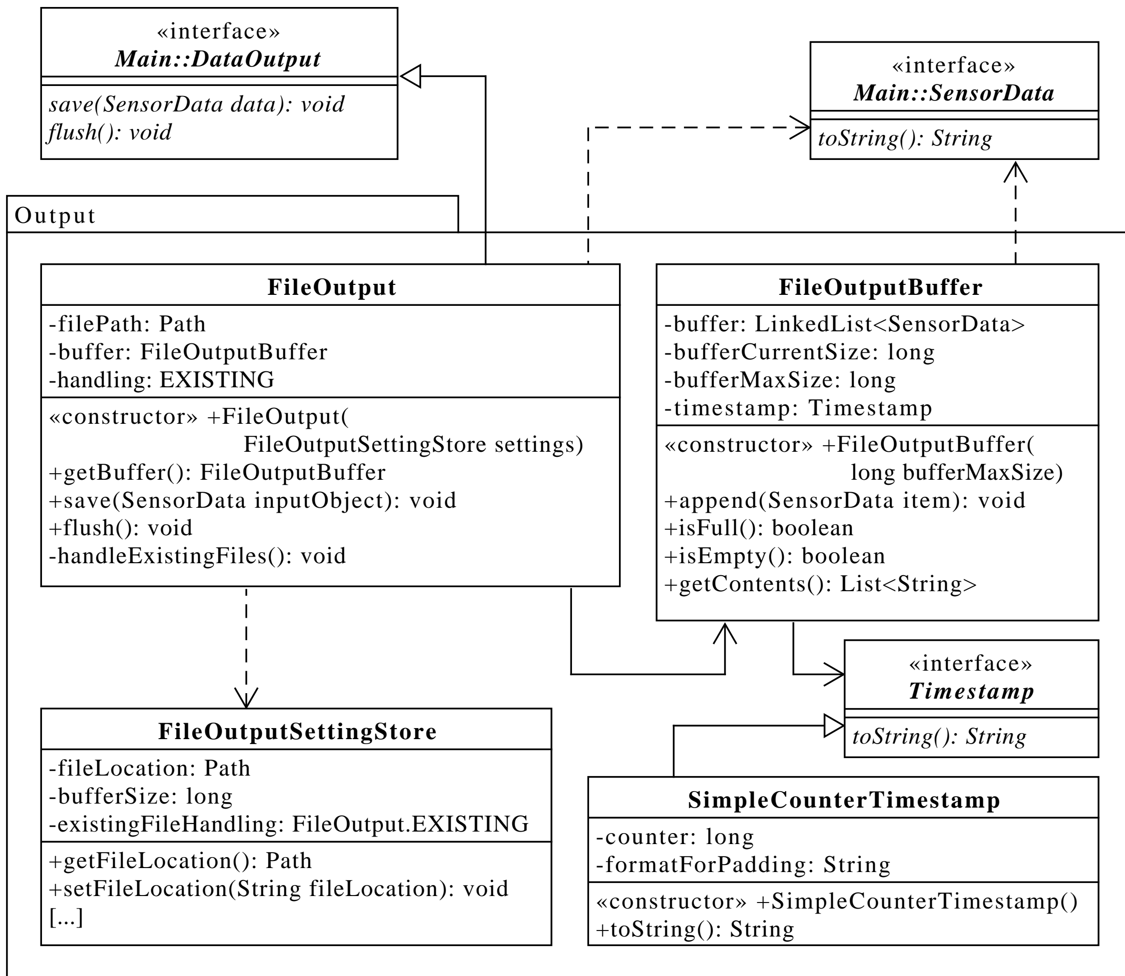


Figure A.2.: fileOutput Implementation Diagram



Appendix B.

Evaluation Results

Figure B.1.: Probability Distribution of 100000 Output Values

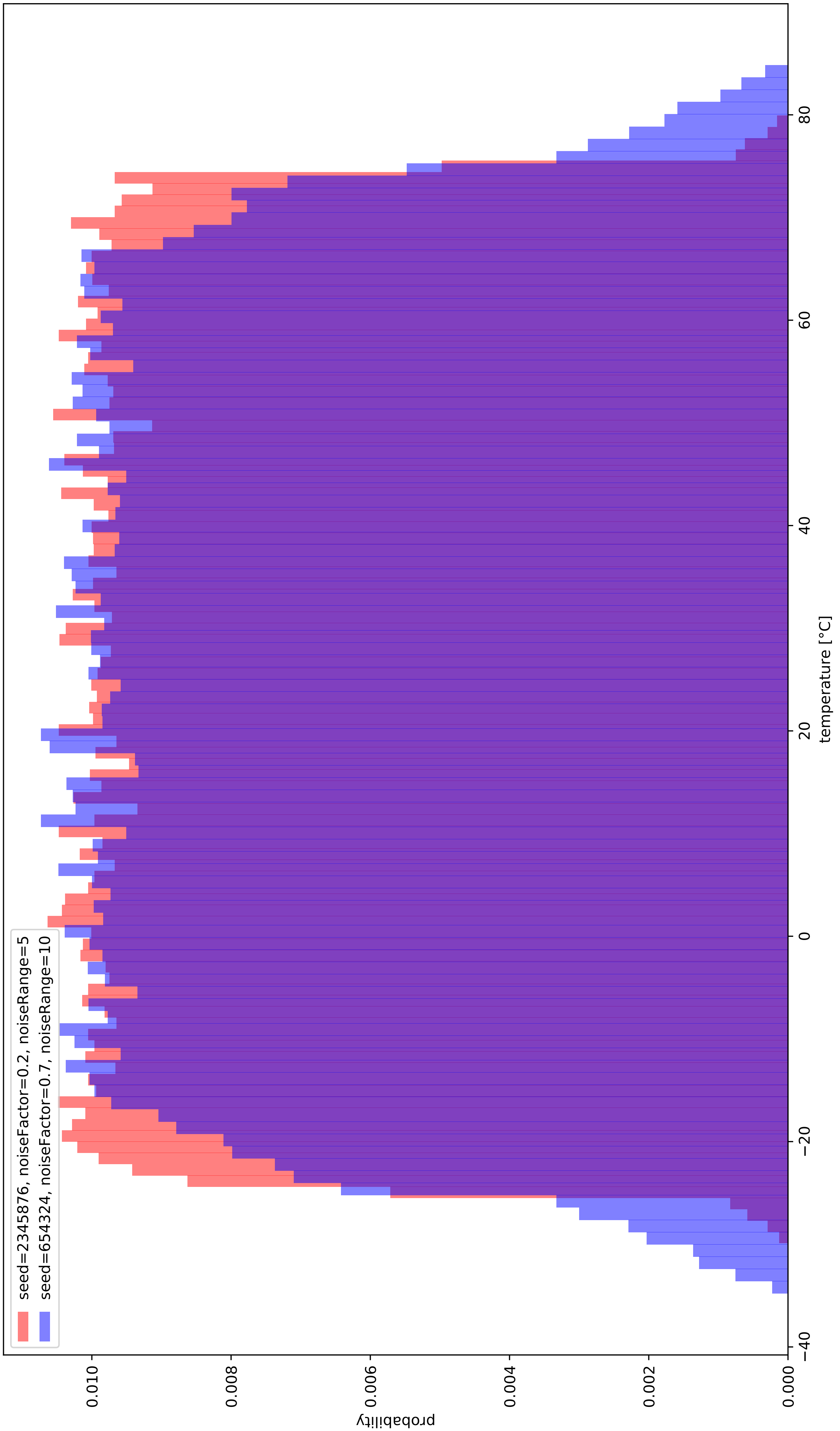


Figure B.2.: Scatterplot of 100000 Output Values

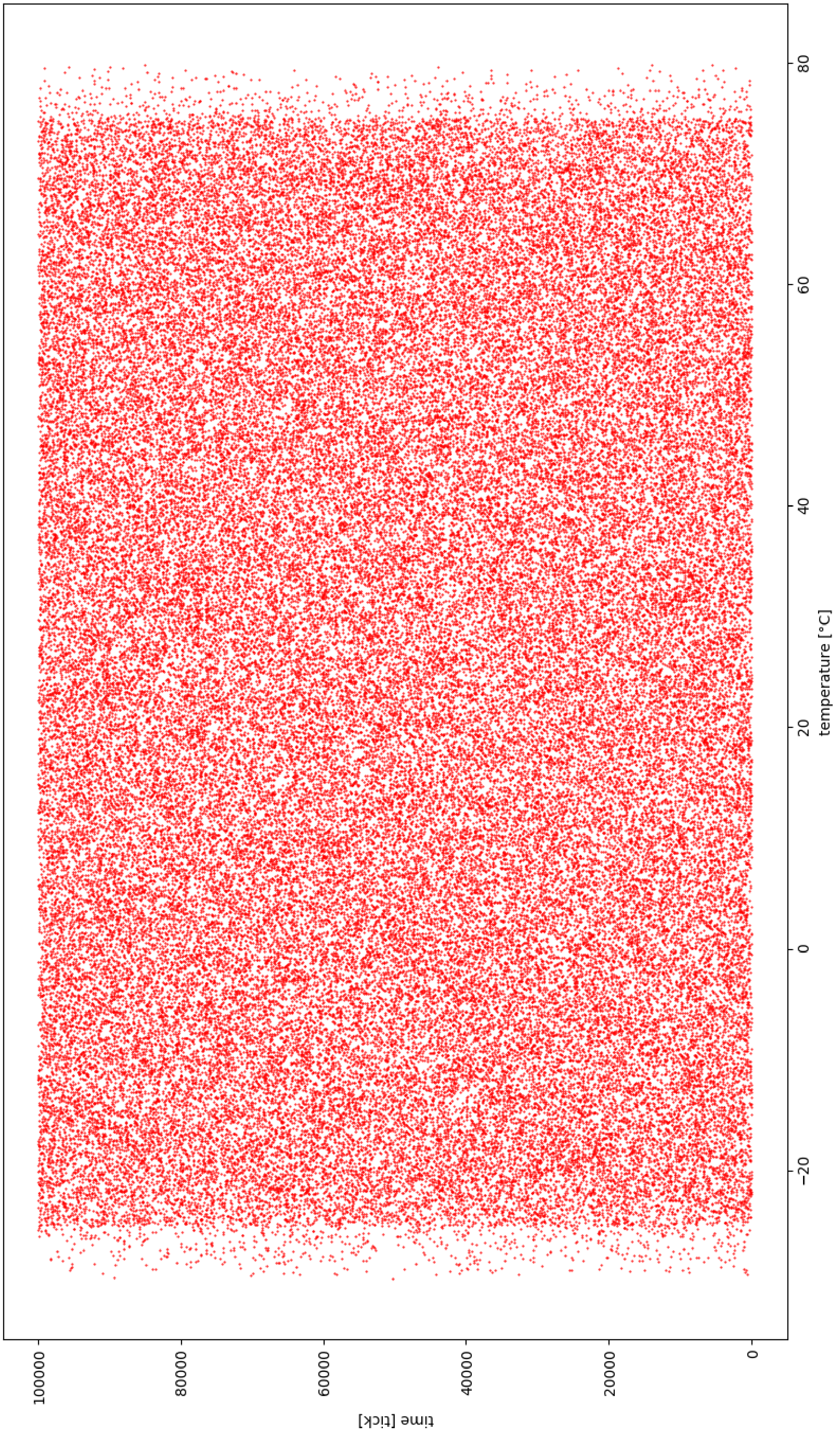


Figure B.3.: Impact of Different Seeds on 1000 Output Values

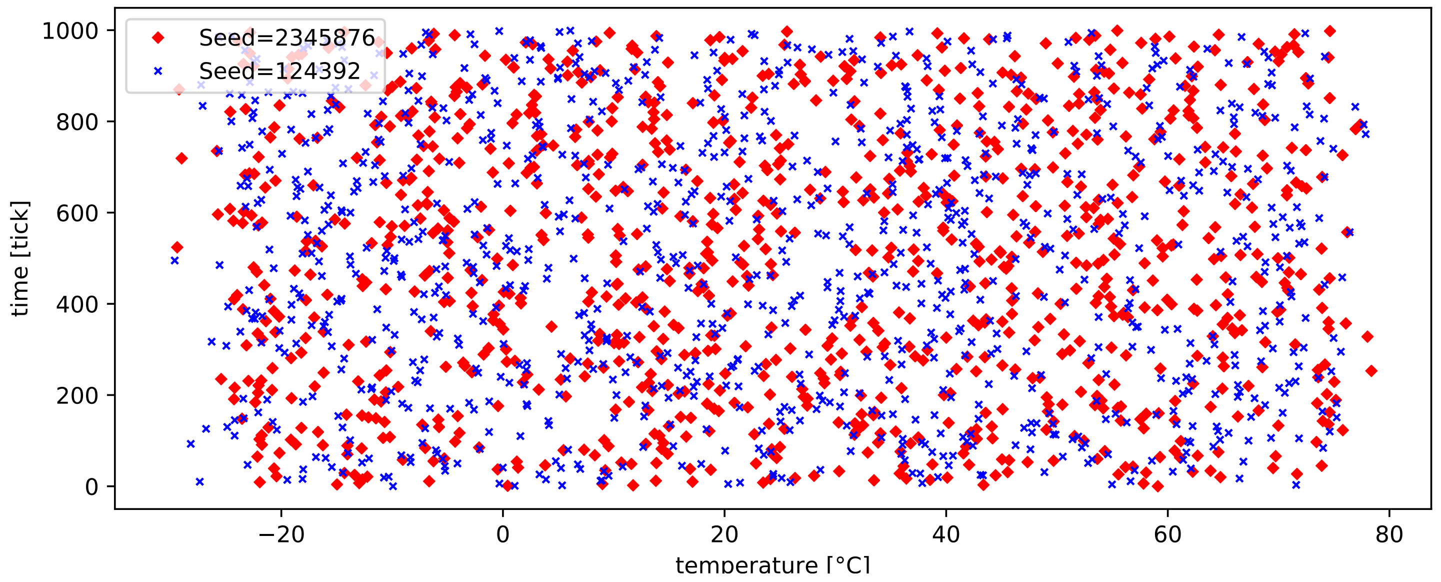
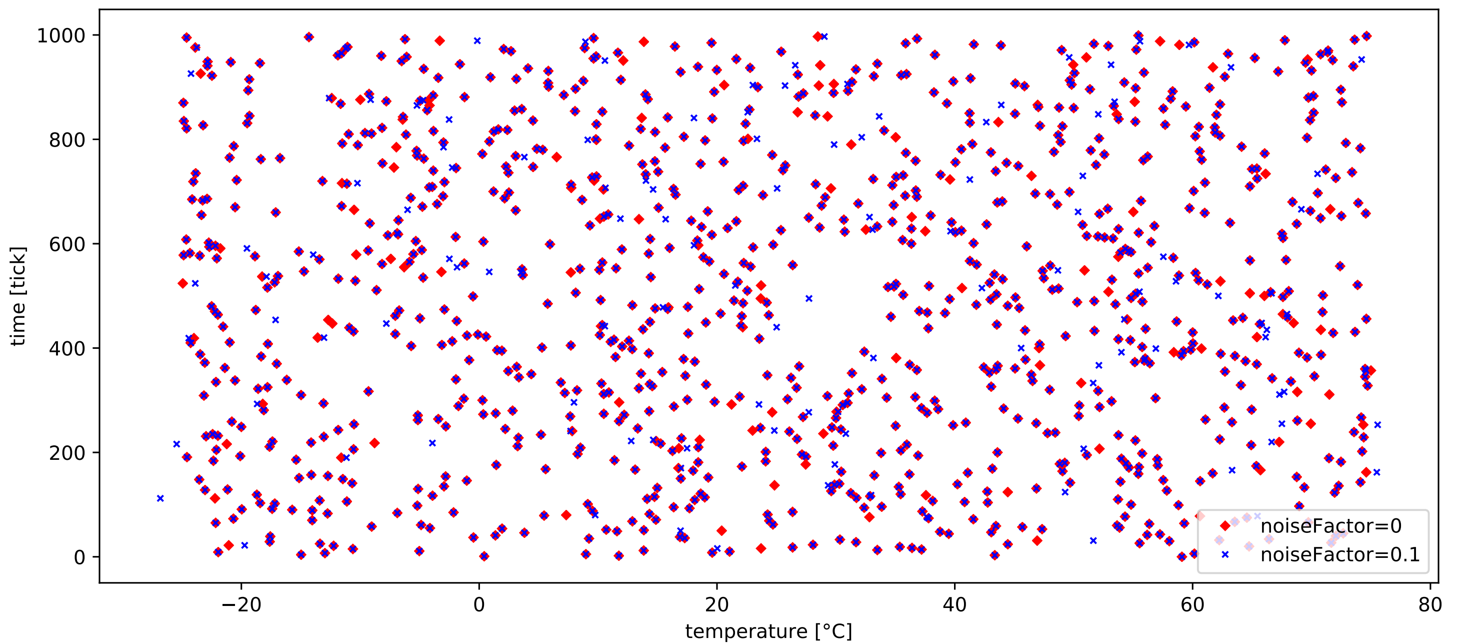


Figure B.4.: Impact of noiseFactor on 1000 Output Values



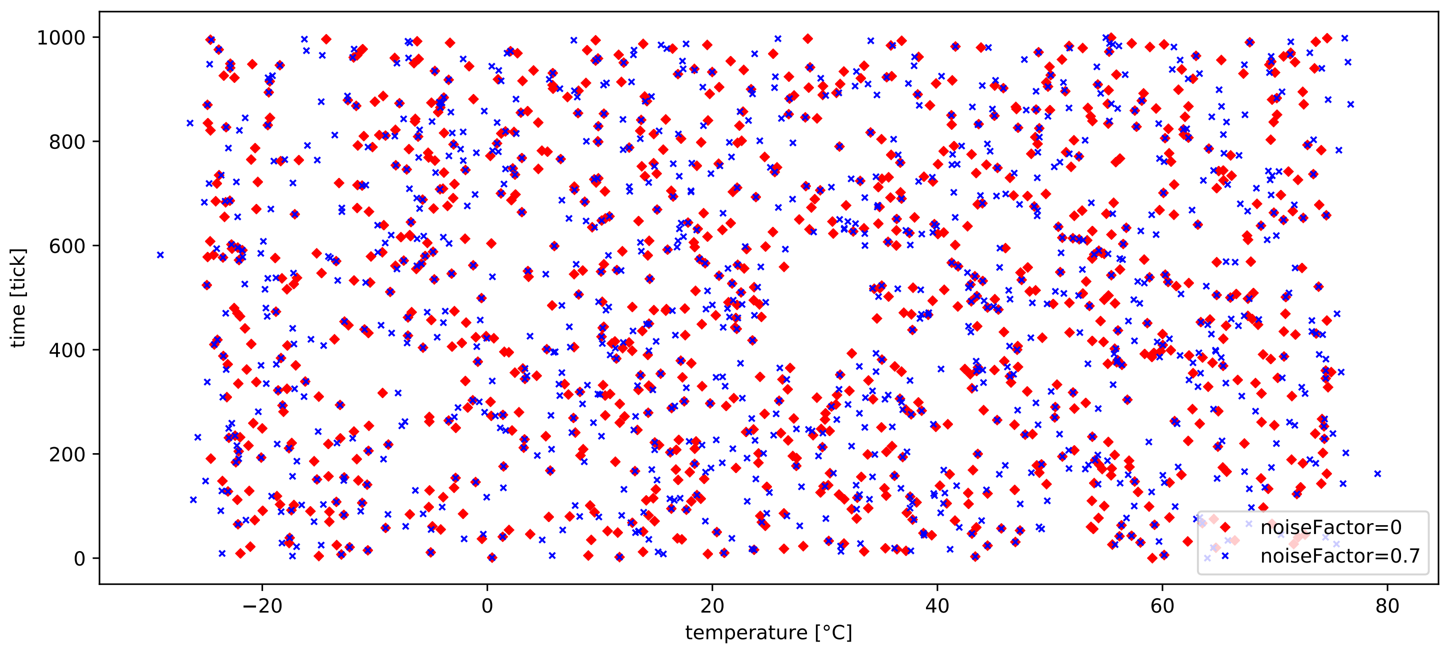
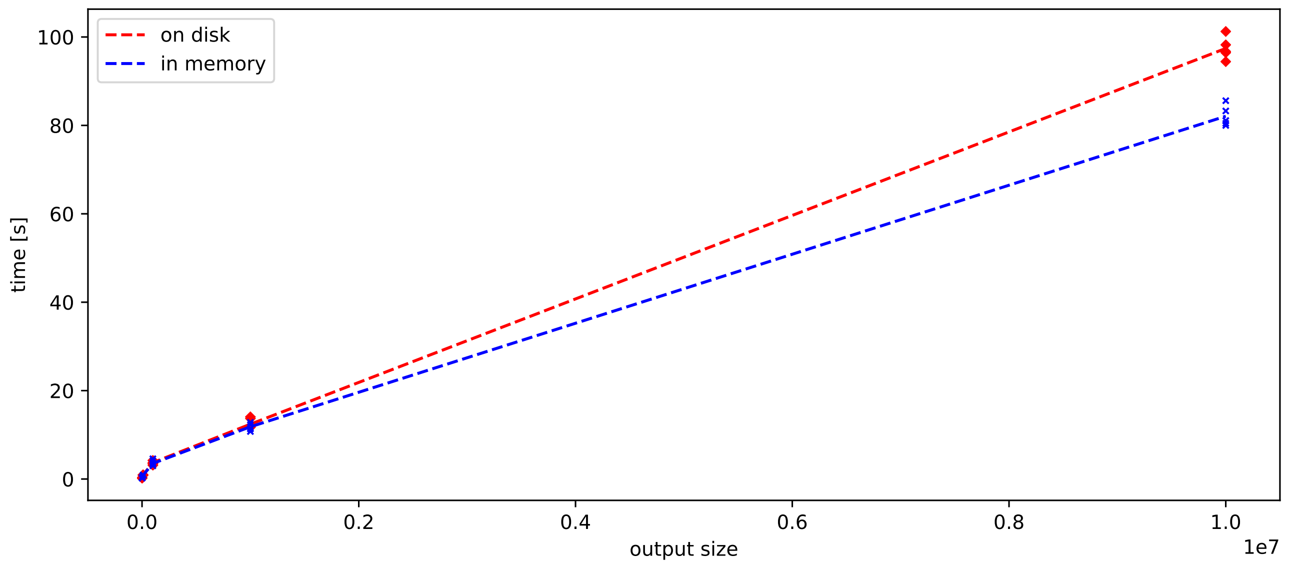


Figure B.5.: Performance Measuring Script

```
1  #!/bin/bash
2  i=1000
3  gencommand='java -jar iot_generator_standalone-0.5-Alpha1.jar'
4  while [ $i -le 10000000 ] ; do
5          echo "Output size: ${i}" | tee -a results.txt
6          for run in {1..5} ; do
7                  echo "Run ${run}" | tee -a results.txt
8                  /usr/bin/time --output=results.txt --append --portability $gencommand
                   ↪  -n $i
9          done
10         (( i=10*$i ))
11 done
```

Figure B.6.: Full Performance Testing Results



Bibliography

[1] P. D. Coddington, J. A. Mathew, and K. A. Hawick. "Interfaces and implementations of random number generators for Java Grande applications". In: *High-Performance Computing and Networking: 7th International Conference, HPCN Europe 1999 Amsterdam, The Netherlands, April 12–14, 1999 Proceedings*. Ed. by Peter Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 873–883. ISBN: 978-3-540-48933-7. DOI: 10.1007/BFb0100647. URL: http://dx.doi.org/10.1007/BFb0100647.

[2] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. "Cryptography Engineering". In: John Wiley & Sons, Feb. 7, 2011. Chap. Generating Randomness. ISBN: 9781118080917. URL: http://www.ebook.de/de/product/21159566/niels_ferguson_bruce_schneier_tadayoshi_kohno_cryptography_engineering.html.

[3] IEEE. *IEEE 1451 Family of Smart Transducer Standards*. URL: http://grouper.ieee.org/groups/1451/0/body%20frame_files/Family-of-1451_handout.htm (visited on 02/28/2017).

[4] Apple Inc. *OS X 10.9.5 Source*. 2010. URL: https://opensource.apple.com/release/os-x-1095.html (visited on 02/19/2017).

[5] Apple Inc. *OS X 10.9.5 Source, xnu-2422.115.4*. 2010. URL: https://opensource.apple.com/source/xnu/xnu-2422.115.4/bsd/dev/random/ (visited on 02/19/2017).

[6] John Kelsey, Bruce Schneier, and Niels Ferguson. "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator". In: *Selected Areas in Cryptography: 6th Annual International Workshop, SAC'99 Kingston, Ontario, Canada, August 9–10, 1999 Proceedings*. Ed. by Howard Heys and Carlisle Adams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 13–33. ISBN: 978-3-540-46513-3. DOI: 10.1007/3-540-46513-8_2. URL: http://dx.doi.org/10.1007/3-540-46513-8_2.

[7] D.E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Pearson Education, 2014. Chap. 3 - Random Numbers. ISBN: 9780321635761. URL: https://books.google.de/books?id=Zu-HAwAAQBAJ.

[8] Oracle. *Java™ Platform, Standard Edition 8 API Specification*. 2016. URL: https://docs.oracle.com/javase/8/docs/api/ (visited on 02/19/2017).

[9] Oracle. *OpenJDK 8 Source Code: java.util.Random*. 2017. URL: `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Random.java` (visited on 03/27/2017).

[10] ns-3 project. *ns-3*. discrete-event network simulator. URL: `https://www.nsnam.org/`.

[11] The GNU Project. *GNU Time*. Free Software Foundation. URL: `https://savannah.gnu.org/projects/time/` (visited on 03/28/2017).

[12] Nathan Sweet and Ola Bini. *YamlBeans*. 2008. URL: `https://github.com/EsotericSoftware/yamlbeans` (visited on 03/27/2017).

[13] Vernon Turner. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. EMC Digital Universe with Research & Analysis by IDC. 2014. URL: `https://www.emc.com/leadership/digital-universe/2014iview/index.htm` (visited on 03/05/2017).

[14] Tim Van den Bulcke et al. "SynTReN: a generator of synthetic gene expression data for design and analysis of structure learning algorithms". In: *BMC Bioinformatics* 7.1 (2006), p. 43. ISSN: 1471-2105. DOI: `10.1186/1471-2105-7-43`. URL: `http://dx.doi.org/10.1186/1471-2105-7-43`.