DOI 10.5162/ettc2020/6.1

Beholder
Automated Data Validation in Flight Test

Miguel Arevalo Nogales
Airbus Defence and Space
miguel.arevalo@airbus.com

Abstract:
Current aircraft development mandate for very complex data acquisition systems, the amount of parameters being stored has grown over the years from a few thousands to almost half a million. Data validation and sanitization has become a priority, especially given current (unattended) data analysis techniques. These two problems mandate for an **automated verification and validation** system, capable of detecting anomalies in real time [1].

Airbus Defence and Space has developed a custom solution to this problem, it is known as **Beholder**. This software leverages user knowledge in the form of rules, making them into a validation piece of code which is automatically loaded and executed over a data flow.

Airbus **Daemons** builds up on current Data Server technology to support an automated process execution system. The combination of both, Daemons and Beholder, provide Flight Test with an automated, unmanned data validation procedure, improving data quality and enhancing response times to erroneous sensors or recordings.

Key words: Automation, validation, rule, sensor, acquisition, testing.

Introduction

Aircraft's data analysis can take place in several phases, from system development to in service maintenance. We can, broadly speaking, group these phases into:

- Development
- Certification
- Delivery
- In service

These phases require different types of data acquisition systems, ranging from very intrusive at the very beginning to almost non-existing in the "in service" phase.

Development phase is, probably, the most intense phase regarding data acquisition. Design offices have to test their theoretical models in real world conditions, system integration has to be tested and design problems may arise. The consequence is data acquisition is really intense, not only internal aircraft data is recorded, but custom instrumentation is installed and prone to changes, having to deal with changing needs. All this facts make data validation especially important during this phase.

Certification can also require a complex data acquisition system to provide proper evidence to authorities, guaranteeing not only the aircraft meets specifications, but also and most importantly flight safety.

Both development and certification have to deal with non-aircraft native sensors, added on purpose to provide further data to analysts. These sensors must be treated with special care since, by their nature (exposition, installation limits, etc.), they are susceptible to failures.

Delivery and in service phases normally rely on simpler acquisition systems, usually no instrumentation is present, and analysts rely on bus recorded data.

All this data is later on used for data analysis, be it a design change needed to fulfil specifications, some kind of evidence needed to certify the aircraft or fleet analysis for predictive maintenance. Of course if data quality is no good all these analysis will be impacted.

Data acquisition plays a major role in aircraft development but, how can we guarantee data quality? Manual validation is costly and error prone, especially in modern developments due to the sheer amount of acquired data; **some way of automatic data validation is needed.**

Solution Overview

Beholder is Airbus Defence and Space solution for rule analysis and anomaly detection.

In an ideal world every system's expert would check for data quality not only after the test but before and during the test itself, meaning that if the data is not up to the quality levels needed, the test could be stopped, resulting in improved safety conditions and reducing expenses in unproductive tests. Due to current acquisition systems complexity and aircraft testing pace, where several flights are performed in a single day, this is utopic.

Beholder answers this problem by leveraging expert knowledge, in the form of validation rules, and produces a simple to read log where every rule occurrence during the flight is recorded.

This methodology guarantees experts are not encumbered by their computing skills and can focus on defining a good-enough rule set that grants recorded data is up to required quality levels for later analysis.

Beholder makes test validation an autonomous task, meaning the expert does not need to visualize or manually analyse the test recording to validate data, but this process has to be executed manually whenever a new test is recorded.

Data recording in Flight Test normally happens on-board during the flight, meaning the whole data set is not available until the aircraft lands and the hard drives are processed. This task is time consuming and can take place during night time; the experts may not be there to validate their data when the data is available.

Airbus Defence and Space has developed a Daemon Automation System intended to execute tasks whenever a new test shows in the servers.

This system provides several advantages to manual systems:

- Tasks can be executed without human interaction, they can be executed during night hours and their results are normally available next day in the morning.
- No human interaction diminishes failure risk and improves timing by not requiring long working hours or turns.
- Validation sharing, which is sometimes overseen if user does not see immediate profit from it, is improved as it is a natural product of standardised rule sets and validation procedures.

Beholder

The rationale behind Beholder is that an expert can write a rule set that defines the data quality level for the operation of the system, detecting anomalies, strange working conditions, and situations where limits or thresholds surpassed.

Manually detecting these conditions is possible, but it is not a desirable approach since it is tedious, error prone and time consuming.

Beholder is based around the following concepts:

- A rule is a logic condition which can be evaluated by a common compiler into a Boolean value.
- Rules are external to Beholder; no rule logic is stored inside the application.
- Each rule can be triggered in every instant.
- All rules are evaluated at the same pace; this eases rule & parameter synchronization. This is what commonly known as CVT: Current Value Table.
- Rules can be grouped in sets; a set defines a system's behaviour.
- Beholder's output is a time line describing when rule's conditions are met: the rule is true.

As a software application Beholder's main characteristics can be summed up as follows:

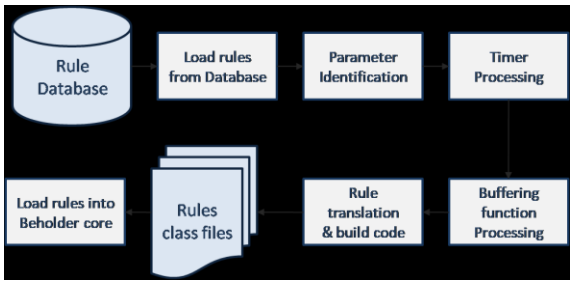
- Multiplatform: developed in Java, OpenJDK 1.8 [2] [3] compliant code.
- Based on Flight Test Multiplatform Analysis Software (FT+) and Dataserver technology.
- UI and BATCH processing modes. Normally the data validation process takes place in an automatic, unmanned way, but a UI offers the possibility to run Beholder on purpose over a set of tests.
- Dataserver is used as the data providing technology, meaning IRF, PFF, or CDF files are readily accessible.
- Since it is Data Server based Client – Server technology offers several benefits including: scalability, heterogeneous systems, etc.
- Automatic code generation. Beholder does not store rules inside its code, they are defined externally to the application. In order to get the best performance user rules are automatically transformed into Java code, which is compiled and linked dynamically in execution time.

- All Java [2] syntax functionalities are allowed in rule definition: simple mathematical functions, more complex functions as defined by Math class, etc.
- Internal Beholder functions. In addition to Java functions Beholder provides a set of custom, internal functions to be used in rules:
- Timers: a timer is a special function intended to measure the amount of time a rule has been in certain state.
- Buffering functions: some functions need a time slice to be usable, relying on past data to provide a meaningful value. Examples of this kind of functions are maximum, minimum o mean values.
- Macros. There are certain macro values that are usable within Beholder rules, for example: current time, analysis starting or ending times, etc.
- External mathematical functions: the idea of these functions is extend Beholder functionalities to Machine Learning systems, including classification algorithms, etc.

From a design [4] [9] [7] [8] point of view Beholder is divided in several modules:

1. Rule analysis and transcription
2. Beholder core
3. Loggers
4. Interface

Prior to rule execution Beholder needs to transcript natural language syntax into a computer intelligible language, in this case Java code.

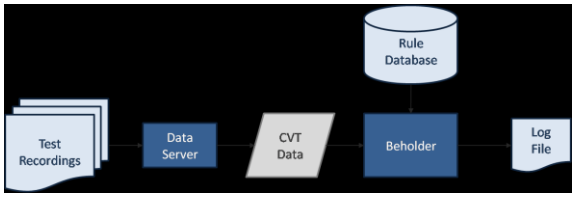


Schema 1: Beholder rule preparation workflow.

The process follows the following steps:

1. Rules are loaded from a user selected database
2. Parameters are identified and replaced with a placeholder. This placeholder is a link (pointer) to a memory location that is going to hold requested data, this data changes every sample.
3. Timers are identified and extracted, for every timer a new condition and rule is created. Original timers are replaced, like parameters before them, by a pointer to a memory location; this location will hold the timer value after the rule has been evaluated.
4. Buffering functions are identified and replaced by pointers, like parameters and timers before them.
5. Rules are translated into plain Java language, every rule is written as a different class, compiled, linked and loaded dynamically [5] [6] into Beholder core.

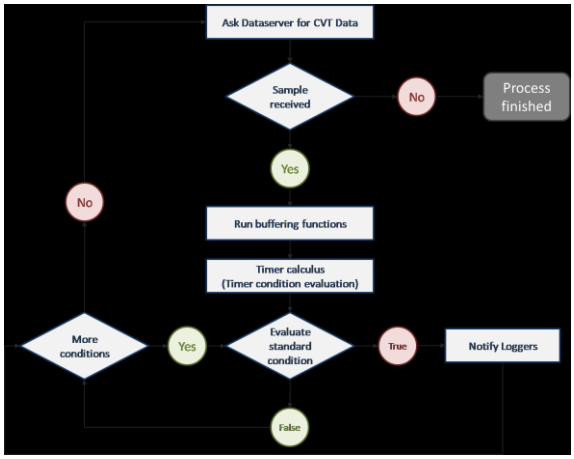
Beholder core is the processing nucleus of the program, all other parts attach to it, preparing rules, providing data or consuming rule events.



Schema 2: Beholder core diagram. Once rules are loaded into the core they are evaluated using Dataserver provided CVT data.

Beholder is based in a processing pipeline; this pipeline is repeated for every sample read from Dataserver (CVT data in Schema 2).

1. Data is read from the Dataserver, one DBT at a time. This DBT contains all parameter information for a current time.
2. Data is copied into a static data array. This is needed to ensure all rules get to access their parameter information properly.
3. Elaborated functions are executed using this static data array.
4. Timers rules are evaluated and the static times array updated with their values
5. Rules are evaluated, in this step all variable values (buffered function values, timers, etc. are known).
6. An event is thrown for every rule state change. These events contain relevant information such as time, change status (true to false or false to true), etc.



Schema 3: Internal Beholder core workflow.

Events are processed asynchronously by different classes, external to the calculus core; some of these classes provide a graphical representation of the events whereas others write rule events in different formats (log files, databases, etc.).

Some examples of loggers include, but are not limited to:

- Database logger, where each event is inserted into a results table.
- Simple text logger, output is a plain text file.
- Summary logger, where rules are grouped by type.

Beholder user interface is a detachable module intended to offer flexibility in test selection and process execution. From a programmatically point of view the design follows a MVC paradigm, separating control agents from graphical representations and model structures.

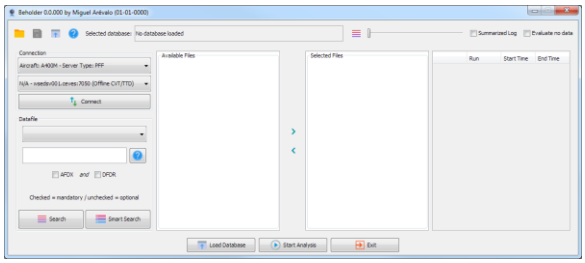


Image 1: Main graphical interface.

Interface is divided in three rows:

- Top row holds an icon bar for opening and closing sessions, selection the amount of threads used for analysis or customizing outputs (logger types).
- Middle row is dedicated to test selection.
- Bottom row has three buttons:
    - Load database
    - Start analysis
    - Exit application

As can be seen middle row holds most of the options, the idea is for the user to proceed from left to right, top to bottom selecting options:

- Dataserver
- Aircraft model and serial number
- Data file filter
- Acquisition system type (DFDR and or AFDX)

Once this information is filled in user can look up for files matching this criteria. Files are presented in the available file list, and can be transferred into the selected file list.

For each file selected user is offered the option to specify a custom time slice or set of time slices. User is offered a default time slice covering the whole test.

Once all this information is selected the analysis is ready to proceed, user must only provide a database containing a rule set.

When in UI mode there are several loggers active intended to provide enhanced visual information to user, whereas in batch mode graphical loggers are reduced to save up resources and status information is shown in a command window.

In order to properly display rules and events a dedicated graphical logger is shown, the Detected Events window (see image 3).

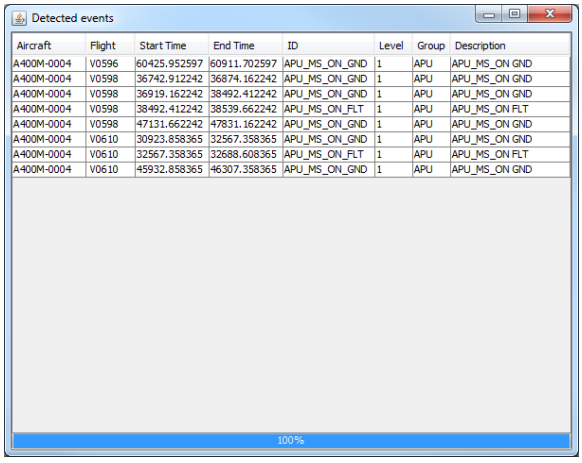


Image 2: Detected events window.

This logger displays a table with a single row for every condition met, this table shows rule ID, aircraft information, starting and ending times and some rule information such as rule group, description or severity level.

While Beholder is running Analysis status windows displays information about the whole process, seen as a data processing task. Conditions are show when they start or end, but the main task of the window is to provide information about the data transfer status.

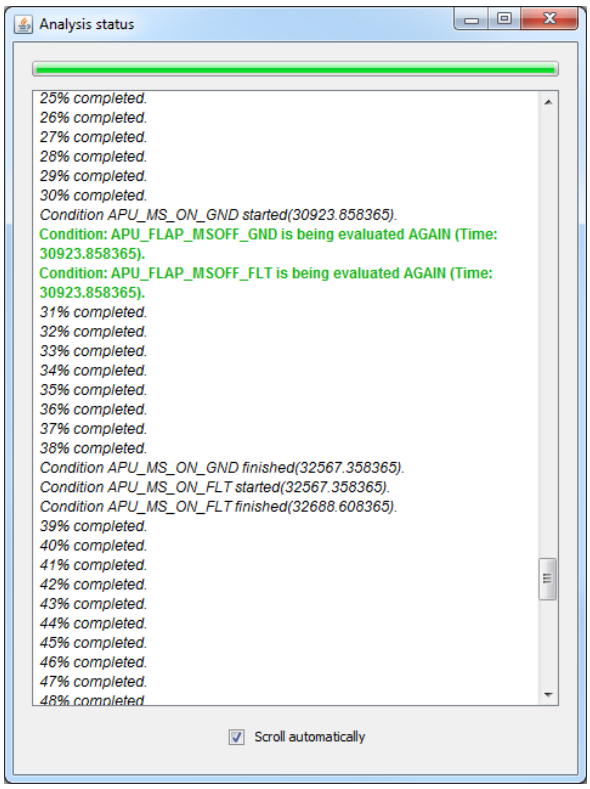


Image 3: Analysis status dialog

Relevant data retrieval information such as aircraft, data transfer status and connection settings are shown in this logger.

Real Time Validation

Data corruption can sometimes happen during testing, due to the large amount of systems and parameter sometimes it is not feasible to check all these data during testing phase, moreover, even in the event of being able to do so, doing it manually is error prone and stressing.

Beholder can act as a real time validation system.

Under this working scenario Beholder works exactly in the same way as described before, but there are differences in parameter inputs and capabilities.

- Beholder limits itself in the amount of rules being analysable. This is a consequence of having to run in real time, if rules cannot be analysed in a timely manner it is better to split the ruleset into smaller sets and execute multiple Beholder instances.
- Data is not provided by an offline Dataserver but an IENA packet one. This type of Dataserver is a real time based server offering the same access protocol to real time telemetry data. The fact that data access is homogeneous between real time and files greatly simplifies the application.
- Due to real time Dataserver way of working there is no need to select an aircraft or data file, the Dataserver already knows which aircraft, MSN and test number it is receiving data from.
- A simplified UI is used, only detected events are offered and a telemetry status monitor is provided. This interface is controlled by the RTMS monitoring system, ensuring it is auto relaunched if it fails.

Real time autonomous validation is a great addition to any telemetry system; typically this was done with ad-hoc applications meaning small changes in sensors or parameters impacted monitoring software.

Beholder splits parameters from its validation system and the software used for it, improving validation opportunities.

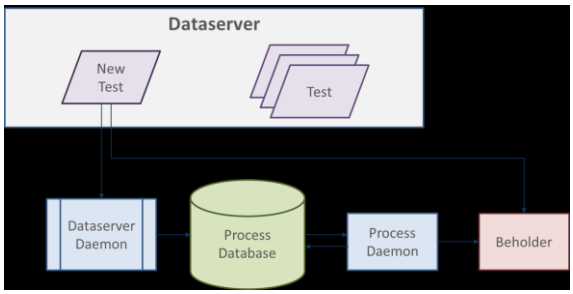
A relevant real time monitoring use scenario, that showcases the advantages of this separation, is adding new rules in real time while the test is being performed. In this use case user can define a new rule, add it to Beholder's database and restart the application. Beholder will analyse the rule and compile it into working code, add it to the rule pool and continue the monitoring process.

In order to improve performance real time mode introduces a key characteristic over standard Beholder, compiled rules are kept from one execution to another, meaning that if there are no rule changes between executions Beholder does not need to recompile the whole ruleset. This optimization is only used in real time because it may lead to lazy condition checking problems; on the other hand it improves application start time, which is necessary if the application is to be managed by RTMS (Real Time Monitoring System).

Daemon System

Daemon system is an external application used to automatically run analysis tasks whenever a new test is detected in Flight Test data servers.

Daemon concept is similar to the one used in computer science, where a program, the daemon, is always running in the background. This program is normally in sleeping state, not consuming resources, but it awakes periodically and runs a predefined task.



Schema 4: Daemon system

The rationale behind Flight Test Daemon System is that whenever a new flight is uploaded into the server a daemon, the Dataserver Daemon, will find it and upload its meta-information into a database, the Process Database. This information describes a test univocally:

- Aircraft model
- Manufacturing serial number
- Test identification: test type plus number.

Other daemons, known as Process Daemons, consume this information; they add their own information to the database and launch a user defined analysis process.

This process can be configured when creating the daemon, in the case we are analysing it is a Data Acquisition Validation Process which is uses Beholder.

The information added to the database is a register, with the same meta-information added by the Dataserver Daemon, adding some additional data:

- Process Id.
- Processing machine.
- Processing date.
- Process status.

When a new process is launched it updates its status to ONGOING, if it fails it will automatically change to ERROR and when it finishes it is updated to FINISHED.

There are control mechanisms in place to ensure daemons are alive, checking daemon status and working times. These guarantees there are no zombie processes and all tasks are executed appropriately.

When a task is updated to an error status other daemons running in different machines to the one that executed this process previously can re-execute the process, this leads to execution redundancy.

One important fact about daemons is that multiple daemons can be configured to take care of the same task type. We may have multiple Validation Daemons so when several flights are uploaded into the Dataserver at the same time they can be processed concurrently by different machines. This kind of redundancy ensures that if a machine is down for any reason others can take its place and keep with the data analysis tasks.

Airbus Daemons have the following characteristics:

- Multiplatform, developed in Java using OpenJDK 1.8 [3] compliance.
- Centralized database based [10]. This ensures multiple daemons, multiple machine synchronization.
- UI is web based.
- Daemons can be tied to a user group, meaning users can be notified whenever a daemon is down or a new test has been analysed.

Daemons interface has been developed using HTML technology. This web interface communicates with daemon database providing information and control over existing daemons, showing their status (sleeping, working or down).

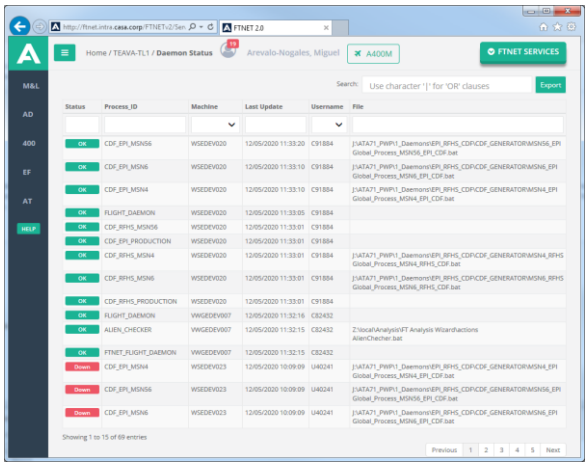


Image 4: Daemon view

Interface offers not only daemon status but also process information. From the interface any user can order a test reprocessing, check its status or see when it was processed and what machine performed the analysis.

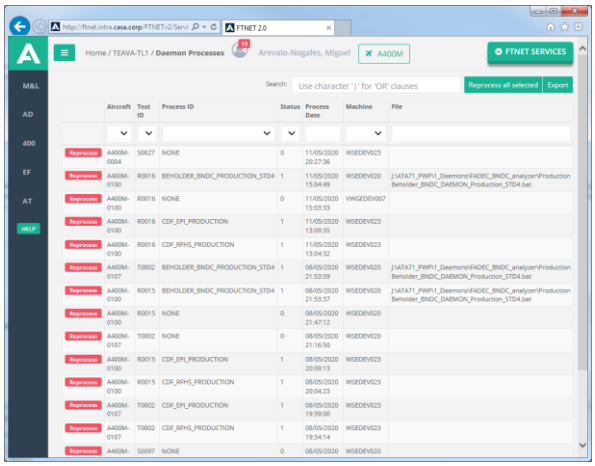


Image 5: Process view

Parameter Validation

Previous chapters showcased different aspects of Beholder, but they did not delve deep into how it can be used as a parameter validation tool and what are the benefits associated to its usage.

First thing to do is define what we should understand by parameter validation. Sometimes, especially when dealing with complex system, we incur in a misconception, mixing system validation with parameter validation, it is important to make a difference between these two concepts.

- Parameter validation only focuses in parameter value, checking if it is within limits and its value is reasonable considering contour conditions.
- System validation checks if the parameter is right considering system's current state.

As can be seen sometimes the difference can be very small, since a parameter out of bounds may not mean our sensor is misbehaving but the system providing erroneous data.

There are several approaches regarding parameter validation:

- Threshold rules. In this case the user defines an upper and lower limit for the parameter. If the parameter exceeds these limits then a warning is issued.

  A especial case are defined state rules, in this case a parameter can only take a small set of predefined values, for example Open-Close o True-False.

- Slope rules. These rules are based not in parameter's value, but how the parameter changes from one instant to another.

  Beholder offers several possibilities in this situation; since we are computing a derivative we can do so taking into account a predefined number of points, this buffer can be selected by user when creating the rule. We can also select the maximum or minimum values for a given time buffer, or even the mean.

- Cross system validation. It is common in aeronautical environments to have different systems provide different parameters for the same measurement. A very useful rule for parameter validation is to compare the difference between parameters or the delay between signals corresponding to the same parameter.

- Data alive check. Some straightforward validation consists on checking data moves and reacts to stimulus.

Once rules have been defined the instrumentation validation phase starts. During this phase systems are started, whenever possible, and data is recorded into small test files, these files are processed and added to Flight Test Dataservers.

As seen in previous chapters Daemons detect these tests and run specific rule detection tasks on them, looking for sensor misbehaviours, parameter exceedances, etc.

The result of this analysis is automatically uploaded to web servers and communicated to system owners and flight test instrumentation personnel, when something strange is found these groups can take proper action to fix the issue or investigate it even further.

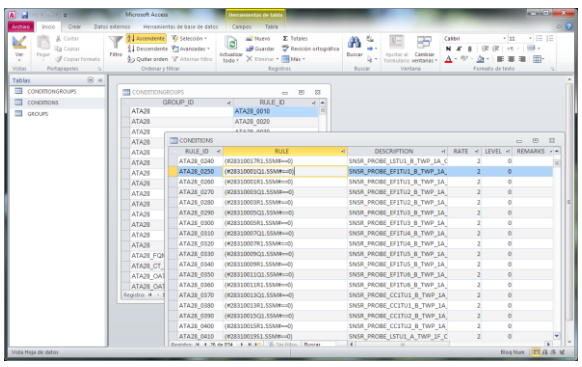


Image 6: Rule database example

Conclusion

Current aircrafts mandate for a large amount of FTI sensors, sometimes more than on thousand. These sensors are added to an already large parameter dataset provided by the aircraft.

Validation of all these parameters is a huge task for which an automated system is necessary, manual validation does not guarantee good enough quality standards.

Making an automated system, open for end users usage, like design offices, not only improves Flight Test data validation, but helps users to get involved in their system's acquisition task.

Since data is automatically processed large amounts of test can be used for validation, meaning a new realm of validation techniques opens to end user: Big Data and machine learning. Using machine learning algorithms for data validation can help to identify upcoming problems, meaning FTI can make predictive maintenance to the acquisition system.

An additional benefit of automated data validation is react time improvement. Current technologies such as 3D printing or Data Science applied to system analysis allow for fast paced development, sometimes referred to as fast prototyping. This methodology mandates for improvements in FTI installation and validation, due to the prototype nature of the modifications being installed in the aircraft.

References

[1] Avner Engel. *Verification, Validation, and Testing of Engineered Systems (Wiley Series in Systems Engineering and Management Book 73)*. Wiley (2010)

[2] Oracle. "Java SE". *Oracle Technology Network*. (18 December 2014).

[3] Alex Kasko. *OpenJDK Cookbook.* Packt Publishing (2015).

[4] Robert Lafore. *Data Structures and Algorithms in Java*. SAMS (2003)

[5] Cay S. Horstmann. *Core Java, Volume II - Advanced Features*. Prentice Hall (2017)

[6] Glen McCluskey. Using Java Reflection. Oracle. https://www.oracle.com/technical-resources/articles/java/javareflection.html (January 1998)

[7] Trygve Reenskaug. *Working with objects. The OOram Software Engineering Method*. Manning/Prentice Hall. (1996)

[8] Gamma, Erich et al. *Design Patterns*. Addison-Wesley (1995)

[9] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Really Media. (2017)

[10] Michael J. Hernandez. *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design*. Addison-Wesley (2007)