Statistical Data Cleaning for Official Statisitcs with R

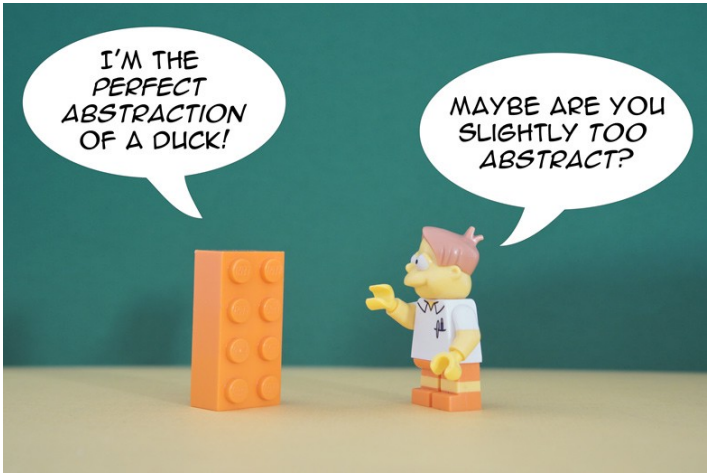
Mark van der Loo, Statistics Netherlands

CBS R&D Methodology

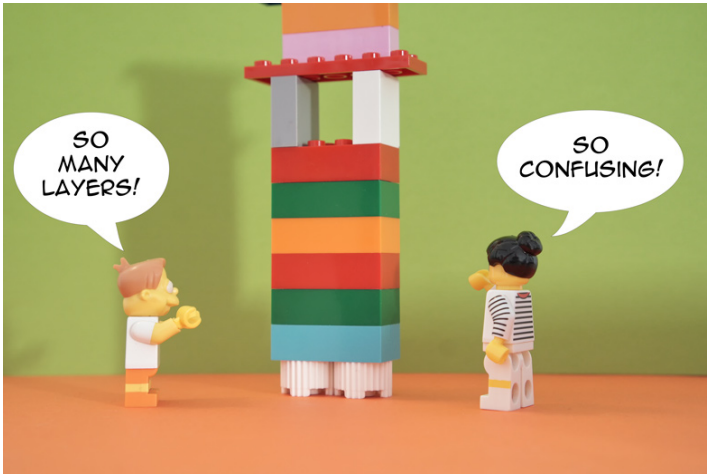
UNECE Meeting on Statistical Data Editing 2020

UNECE

UNECE

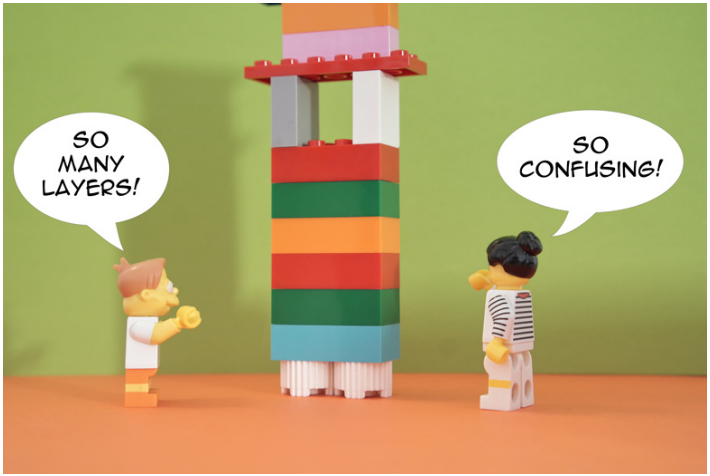

I'M THE PERFECT ABSTRACTION OF A DUCK!
MAYBE ARE YOU SLIGHTLY TOO ABSTRACT?

Abstraction?



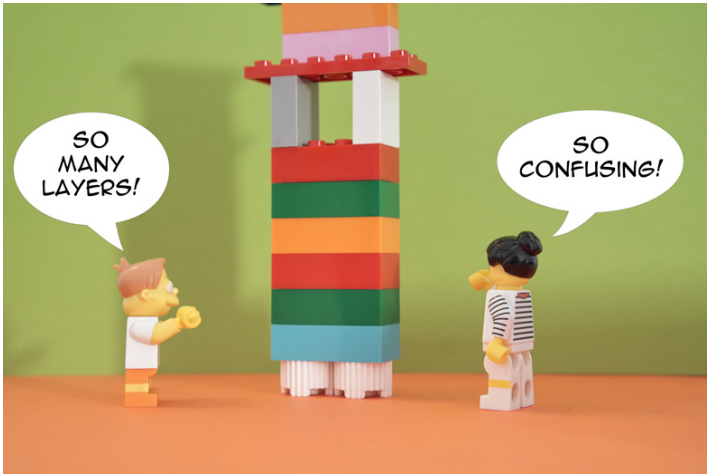
source: https://thevaluable.dev/abstraction-type-software-example/

Abstraction?



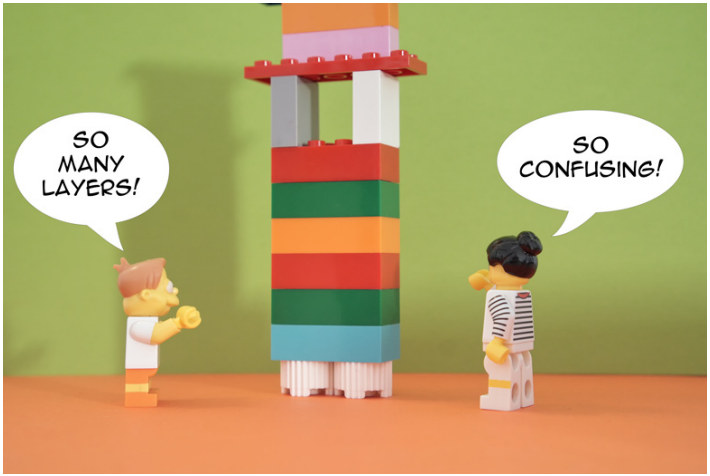
source: https://thevaluable.dev/abstraction-type-software-example/

Abstraction?



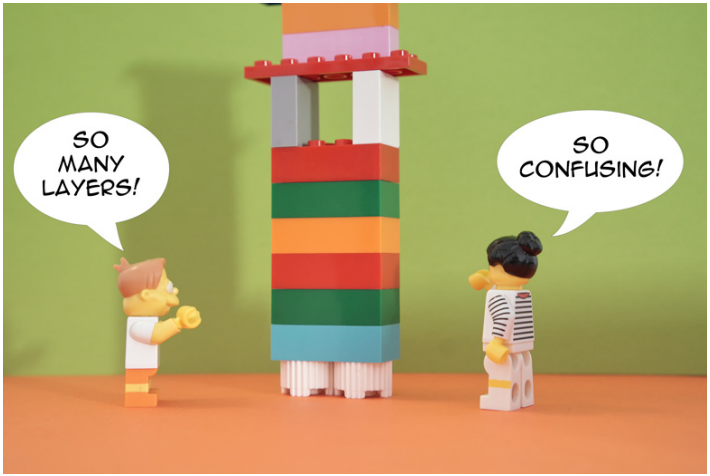
source: https://thevaluable.dev/abstraction-type-software-example/

Abstraction?



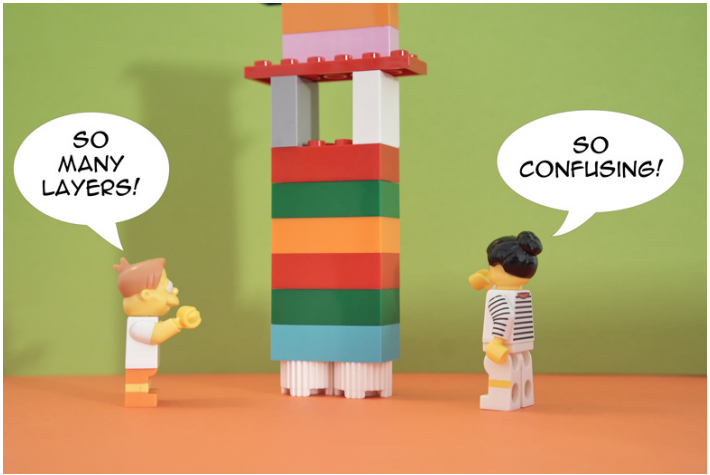
source: https://thevaluable.dev/abstraction-type-software-example/

Abstraction?



source: https://thevaluable.dev/abstraction-type-software-example/

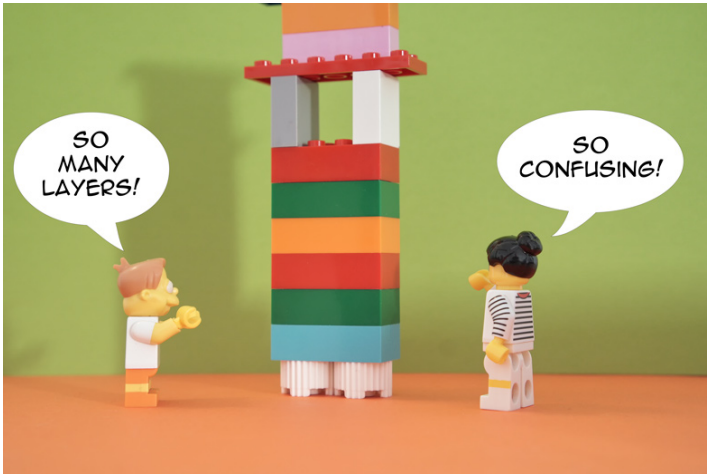
Abstraction?



source: https://thevaluable.dev/abstraction-type-software-example/

Abstraction?

?



source: https://thevaluable.dev/abstraction-type-software-example/

UNECE

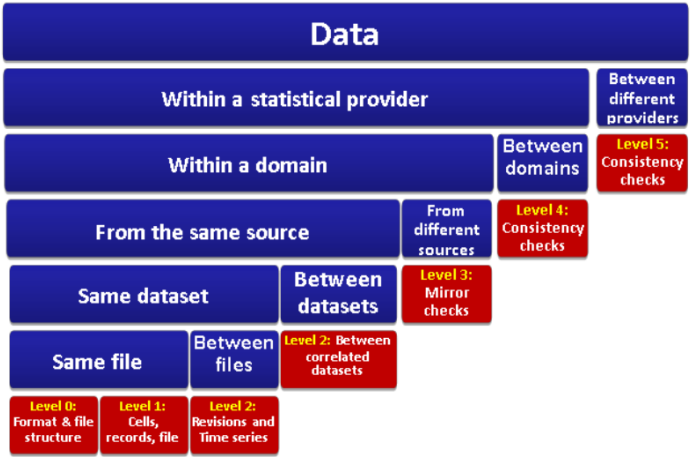
Tree by Piet Mondriaan

1909

1911

1912

Example: Data validation



A. Simon (2013) *Exhaustive and detailed typology of validation rules.* ESTAT working document.

Example: Data validation

Definition 3. *A* data validation function *is a surjective function*

$$v : D^K \twoheadrightarrow \{\text{False}, \text{True}\}$$

MPJ van der Loo and E. de Jonge (2020) *Data Validation*. Wiley StatsRef: Statistics Reference Online, DOI: 10.1002/9781118445112.stat08255

Wiley StatsRef:
Statistics Reference Online

Data Validation

Mark P.J. van der Loo and Edwin de Jonge

Keywords: *data quality, data cleaning*

Abstract: Data validation is the activity where one decides whether or not a particular data set is fit for a given purpose. Formalizing the requirements that drive this decision process allows for unambiguous communication of the requirements, automation of the decision process, and opens up ways to maintain and investigate the decision process itself. The purpose of this article is to formalize the definition of data validation and to demonstrate some of the properties that can be derived from this definition. In particular, it is shown how a formal view of the concept permits a classification of data quality requirements, allowing them to be ordered in increasing levels of complexity. Some subtleties arising from combining possibly many such requirements are pointed out as well.

Informally, data validation is the activity where one decides whether or not a particular data set is fit for a given purpose. The decision is based on testing observed data against prior expectations that a plausible data set is assumed to satisfy. Examples of prior expectations range widely. They include natural limits on variables (weight cannot be negative), restrictions on combinations of multiple variables (a man cannot be pregnant), combinations of multiple entities (a mother cannot be younger than her child), and combinations of multiple data sources (import value of country A from country B must equal the export value of country B to country A). Besides the strict logical constraints mentioned in the examples, there are often softer constraints based on human experience. For example, one may not expect a certain economic sector to grow more than 5% in a quarter. Here, the 5% limit does not represent a physical impossibility but rather a limit based on past experience. Since one must decide in the end whether a data set is usable for its intended purpose, we treat such assessments on equal footing.

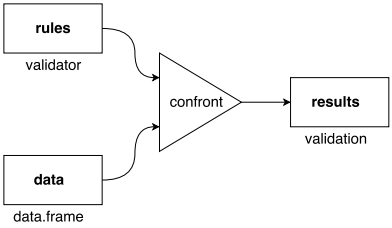
The purpose of this article is to formalize the definition of data validation and to demonstrate some of the properties that can be derived from this definition. In particular, it is shown how a formal view of the concept permits a classification of data validation rules (assertions), allowing them to be ordered in increasing levels of "complexity." Here, the term "complexity" refers to the amount of different types of information necessary to evaluate a validation rule. A formal definition also permits development of tools for automated validation and automated reasoning about data validation[1–9]. Finally, some subtleties arising from combining validation rules are pointed out.

Statistics Netherlands, The Hague, The Netherlands

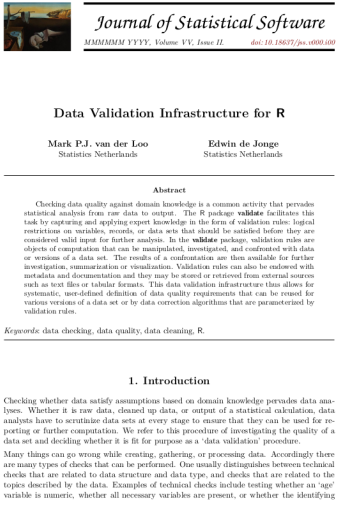
DOI: 10.1002/9781118445112.stat08255

UNECE

Example: Validate



MPJ van der Loo and E. de Jonge (2020) *Data Validation Infrastructure for R*. JSS (Accepted) https://arxiv.org/abs/1912.09759



Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi:10.18637/jss.v000.i00

Data Validation Infrastructure for R

Mark P.J. van der Loo
Statistics Netherlands

Edwin de Jonge
Statistics Netherlands

Abstract

Checking data quality against domain knowledge is a common activity that pervades statistical analysis from raw data to output. The R package **validate** facilitates this task by capturing and applying expert knowledge in the form of validation rules: logical restrictions on variables, records, or data sets that should be satisfied before they are considered valid input for further analysis. In the **validate** package, validation rules are objects of computation that can be manipulated, investigated, and confronted with data or versions of a data set. The results of a confrontation are then available for further investigation, summarization or visualization. Validation rules can also be endowed with metadata and documentation and they may be stored or retrieved from external sources such as text files or tabular formats. This data validation infrastructure thus allows for systematic, user-defined definition of data quality requirements that can be reused for various versions of a data set or by data correction algorithms that are parameterized by validation rules.

Keywords: data checking, data quality, data cleaning, R.

1. Introduction

Checking whether data satisfy assumptions based on domain knowledge pervades data analyses. Whether it is raw data, cleaned up data, or output of a statistical calculation, data analysts have to scrutinize data sets at every stage to ensure that they can be used for reporting or further computation. We refer to this procedure of investigating the quality of a data set and deciding whether it is fit for purpose as a 'data validation' procedure.

Many things can go wrong while creating, gathering, or processing data. Accordingly there are many types of checks that can be performed. One usually distinguishes between technical checks that are related to data structure and data type, and checks that are related to the topics described by the data. Examples of technical checks include testing whether an 'age' variable is numeric, whether all necessary variables are present, or whether the identifying

Learning to use validate

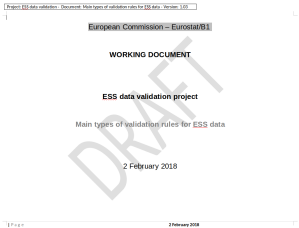
```
library(validate)
retailers <- read.csv("supermarkets.csv")

rules <- validator(
    total.rev - total.costs == profit
  , mean(profit) >= 10
)
result <- confront(retailers, rules, key="id")

head(as.data.frame(result), 3)
```

UNECE

We are extending validate

Increased support for

- long-form (transmission) data formats
- time series data
- grouped checks



Example: checking for gaps in (time) series

```
library(validate)

is_linear_sequence(c(1,2,3))
```

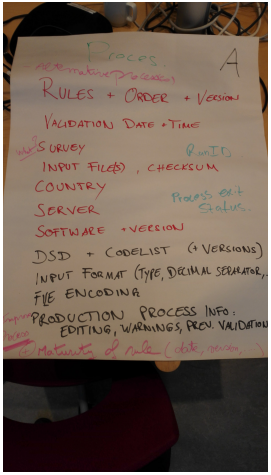
```
## [1] TRUE
```

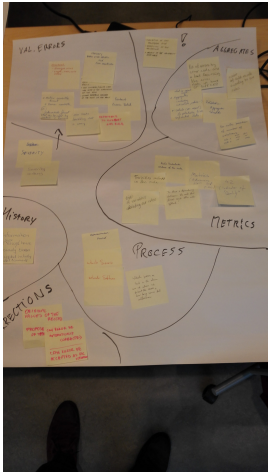
```
is_linear_sequence(c(1,4,5))
```

```
## [1] FALSE
```

Part of ESS Grant agreement 882817–NL–validation

Example: Validation Report Standard





Example: Validation Report Standard

- **Event**
  - timestamp
  - actor (who/what)
  - *agent, trigger*
- **Rule**
  - Language
  - Expression
  - Severity (information/warning/error)
  - *Description*
- **Data**
  - *UruX* (population, measurement, population element, variable)
  - *Description*
- **Value** $(0, 1, \text{NA})$

Design of a generic machine-readable validation report structure

Mark van der Loo and Olav ten Bosch
Statistics Netherlands

Version 1.0.0 August 20, 2019

M. van der Loo and O. ten Bosch (2017) *Design of a generic machine-readable report structure* Deliverable of ESSnet ValidatFOSS ec.europa.eu/eurostat/cros/system/files/wp2-genericvalidationreport.pdf

Example: Validation Report Standard


```
[
  {
    "id": "agg1",
    "type": "aggregation",
    "event": {
      "time": "20150930T104052+0200",
      "actor": "Triton demo 2",
      "agent": null,
      "trigger": null
    },
    "aggregate": {
      "language": "VBA",
      "expression": "COUNT",
      "severity": "information",
      "description": "Totalt antal fel
      "status": ""
    },
    "data": {
      "source": null,
      "target": null,
      "description": ""
    },
    "value": "17"
  },
  {
    "id": "A_22",
    "type": "validation",
    "event": {
      "time": "20150930T104041+0200",
      "actor": "Triton demo 2",
      "agent": null,
      "trigger": null
    },
    " , , ,
```

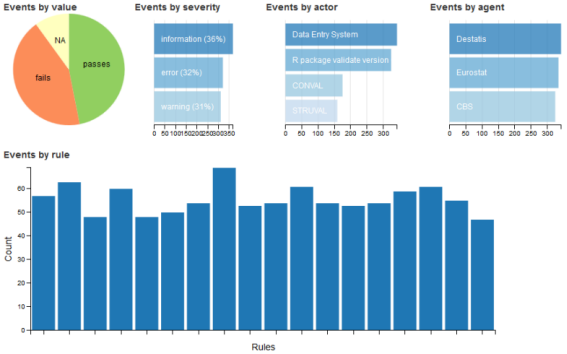

Triton (SW)


```
[
  {
    "id": "aggr",
    "type": "aggregation",
    "event": {
      "time": "",
      "actor": "",
      "agent": null,
      "trigger": null
    },
    "expression": {
      "language": "VTL 1.1",
      "aggregate": "COUNT",
      "description": "Total nr of revisions above threshold",
    },
    "data": [
      "source": ["WC001", "WC002", "WC003", "WC004"],
      "target": null,
      "description": "EU member states"
    },
    "value": "4"
  },
  {
    "id": "WC001",
    "type": "validation",
    "event": {
      "time": "20150930T104041+0200",
      "actor": "Laura Vignola",
      "agent": null,
      "trigger": null
    },
    "rule": [
      "language": "VTL 1.1",
      "expression": "
        ds_esa_current_filt =
          ds_esa_current [
            filter (PRICE = 'V' and PRICE = 'Y') ];

        ds_esa_previous_filt =
```


VTL

UNECE

Example: Validation Report Standard



events selected. Click on the graphs to apply filters.

Id	Value	time	severity	language	change	actor	agent	trigger
id_0073ixekhk	0	2017-09-01T07:51:44.943Z	I	VTL 1.0	down	STRUVAL	Eurostat	Jim Statistician
id_017n0zovl6	1	2017-09-01T07:51:44.933Z	W	validate	up	R package validate version 0.2.0	CBS	John Statistician
id_01p9wjze4l	0	2017-09-01T07:51:44.943Z	W	validate	up	R package validate version 0.2.0	CBS	John Statistician
id_03cd692qy6	1	2017-09-01T07:51:44.943Z	E	Estatistik	up	Data Entry System	Destatis	Lucas Statistician
id_05adv7979x	0	2017-09-01T07:51:44.933Z	W	validate	up	R package validate version 0.2.0	CBS	John Statistician

Example: imputation

Example: imputation

Example: (s)imputation

An imputation prodedure is specified by

1. The variable to impute
2. An imputation model
3. Predictor variables

The simputation interface

```
impute_<model>(data
  , <imputed vars> ~ <predictor vars>
  , [options])
```

UNECE

Example: simputation

```
head(retailers, 3)
```

```
##   staff turnover other.rev total.rev
## 1    75       NA        NA      1130
## 2     9     1607        NA      1607
## 3    NA     6886       -33      6919
```

Example: simputation

```
retailers %>%
  impute_lm(other.rev ~ turnover) %>%
  head(3)
```

```
##   staff turnover other.rev total.rev
## 1    75       NA        NA      1130
## 2     9     1607  5427.113      1607
## 3    NA     6886   -33.000      6919
```

UNECE

Example: simputation

```
retailers %>%
  impute_lm(other.rev ~ turnover) %>%
  impute_lm(other.rev ~ staff) %>%
  head(3)
```

```
##   staff turnover other.rev total.rev
## 1    75       NA  4114.065      1130
## 2     9     1607  5427.113      1607
## 3    NA     6886   -33.000      6919
```

UNECE

Example: simputation

```
retailers %>%
  impute_rlm(other.rev ~ turnover) %>%
  impute_rlm(other.rev ~ staff) %>%
  head(3)
```

```
##   staff turnover other.rev total.rev
## 1    75       NA  64.88174      1130
## 2     9     1607  17.25247      1607
## 3    NA     6886 -33.00000      6919
```

UNECE

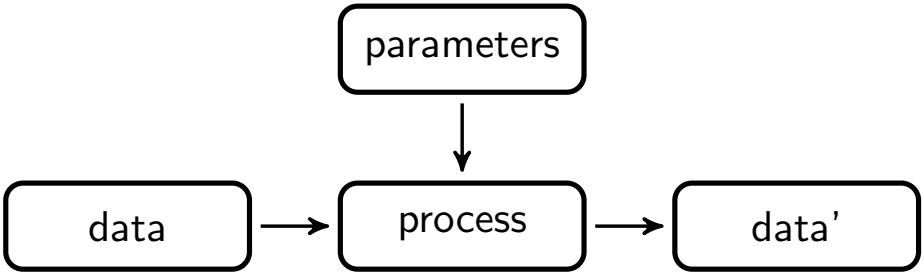
Modularity and composability

*We call a system **modular** when it is composed of various parts that can be linked to each other.*

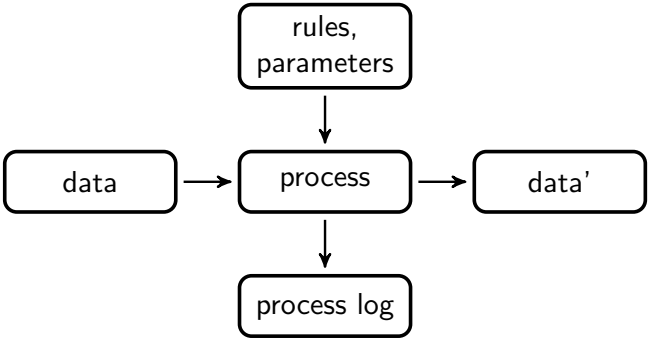
*We call a system **composable** when it shows no emergent behaviour.*

UNECE

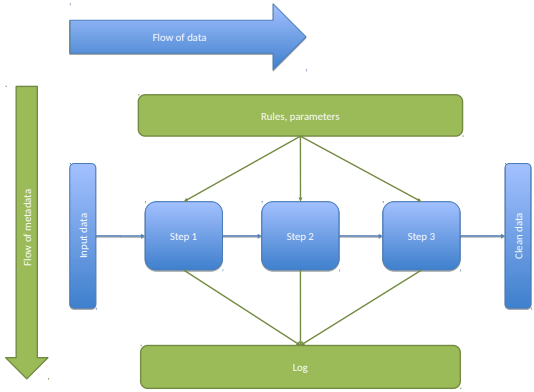
A hard composability problem



A hard composability problem



A hard composability problem

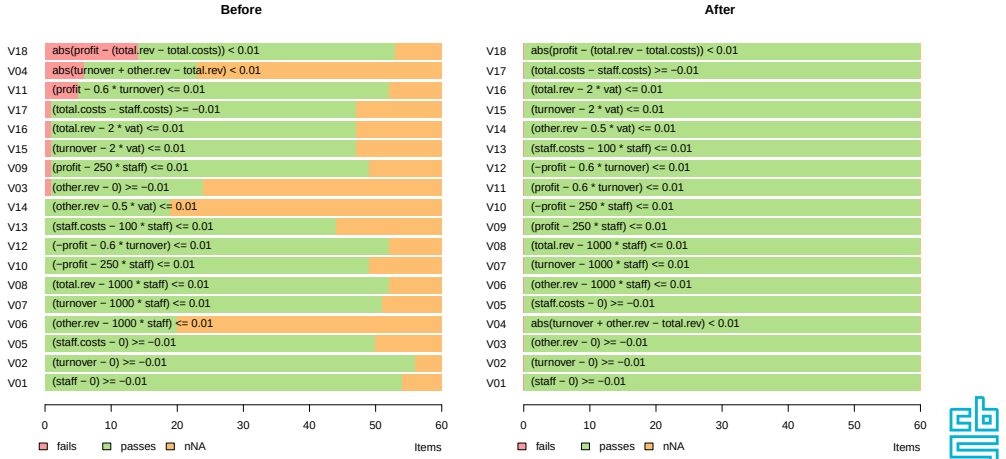


Example: cleaning SBS data

```
# read data, rules
supermarkets <- read.csv("data/input.csv")
rules        <- validator(.file="data/rules.yaml")

# error localization, imputation, adjusting
supermarkets <- replace_errors(supermarkets, rules)
A <- is.na(supermarkets)
supermarkets <- impute_mf(supermarkets, . ~ .)
supermarkets <- match_restrictions(supermarkets, rules, adjust=A)

# write output
write.csv(supermarkets, "data/clean_data.csv", row.names=FALSE)
```

UNECE

Example: cleaning SBS data



UNECE

Adding the second data stream

A method for deriving information from running R code

by Mark P.J. van der Loo

Abstract It is often useful to tap information from a running R script. Obvious use cases include monitoring the consumption of resources (time, memory) and logging. Perhaps less obvious cases include tracking changes in R objects or collecting output of unit tests. In this paper we demonstrate an approach that abstracts collection and processing of such secondary information from the running R script. Our approach is based on a combination of three elements. The first element is to build a customized way to evaluate code. The second is labeled *local masking* and it involves temporarily masking a user-facing function so an alternative version of it is called. The third element we label *local side effect*. This refers to the fact that the masking function exports information to the secondary information flow without altering a global state. The result is a method for building systems in pure R that lets users create and control secondary flows of information with minimal impact on their workflow, and no global side effects.

Introduction

The R language provides a convenient language to read, manipulate, and write data in the form of scripts. As with any other scripted language, an R script gives description of data manipulation activities, one after the other, when read from top to bottom. Alternatively we can think of an R script as a one-dimensional visualisation of data flowing from one processing step to the next, where intermediate variables or pipe operators carry data from one treatment to the next.

We run into limitations of this one-dimensional view when we want to produce data flows that are somehow 'orthogonal' to the flow of the data being treated. For example, we may wish to follow the state of a variable while a script is being executed, report on progress (logging), or keep track of resource consumption. Indeed, the sequential (one-dimensional) nature of a script forces one to introduce extra expressions between the data processing code.

As an example, consider a code fragment where the variable x is manipulated.

```
x[x > threshold] <- threshold
x[is.na(x)]  <- median(x, na.rm=TRUE)
```

In the first statement every value above a certain threshold is replaced with a fixed value, and next, missing values are replaced with the median of the completed cases. It is interesting to know how an aggregate of interest, say the mean of x, evolves as it gets processed. The instinctive way to do this is to edit the code by adding statements to the script that collect the desired information.

```
meanx <- mean(x, na.rm=TRUE)
x[x > threshold] <- threshold
meanx <- c(meanx, mean(x, na.rm=TRUE))
x[is.na(x)]  <- median(x, na.rm=TRUE)
meanx <- c(meanx, mean(x, na.rm=TRUE))
```

This solution clutters the script by insterting expresssions that are not necessary for its main purpose. Moreover, the tracking statements are repetitive, which validates some form of abstraction.

A more general picture of what we would like to achieve is given in Figure 1. The 'primary data flow' is developed by a user as a script. In the previous example this concerns processing x. When the script runs, some kind of logging information, which we label the 'secondary data flow' is derived implicitly by an abstraction layer.

Creating an abstraction layer means that concerns between primary and secondary data flows are separated as much as possible. In particular, we want to prevent the abstraction layer from inspecting or altering the user code that describes the primary data flow. Furthermore, we would like the user to have some control over the secondary flow from within the script, for example to start, stop or parameterize the secondary flow. This should be done with minumum editing of the original user code and it should not rely on global side effects. This means that neither the user, nor the abstraction layer for the secondary data flow should have to manipulate or read global variables, options, or other environmental settings to convey information from one flow to the other. Finally, we want to treat the availability of a secondary data flow as a normal situation. This means we wish to avoid using signaling conditions (e.g. warnings or errors) to convey information between the flows, unless there is an actual exceptional condition such as an error.

R Journal (Accepted) Arxiv:2002.07472

Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

Monitoring data in R with the lumberjack package

Mark P.J. van der Loo
Statistics Netherlands

Abstract

Monitoring data while it is processed and transformed can yield detailed insight into the dynamics of a (running) production system. The **lumberjack** package is a lightweight package allowing users to follow how an R object is transformed as it is manipulated by R code. The package abstracts all logging code from the user, who only needs to specify which objects are logged and what information should be logged. A few default loggers are included with the package but the package is extensible through user-defined logger objects.

Keywords: Data Quality, Process Monitoring, Logging, Debugging, R.

1. Introduction

It is common practice to monitor a data analyses process while it is running. Especially in production environments where analyses are run repeatedly on different but structurally comparable data sets. Following a running procedure is usually done with some form of logging system, where the running process updates a log that can be tracked by users as it proceeds.

One can distinguish two types of monitoring. On the one hand there is *process logging*, or just *logging* for short. Here, the running system notifies users of progress and significant events, usually by writing short time-stamped messages to a file (where 'file' can be a flat text file, database, screen or any other device accepting such input). The aim of these messages is to signal whether procedures have concluded successfully, and if they haven't, to report what went wrong. Such information is highly valuable in *post-mortem* investigations, for example when a production script has crashed. On the other hand there is *tracing* where the state of variables is followed over the course of the process. Tracing is usually applied at the development stage as a debugging tool, often using an interactive interface tool to run the code line by line while inspecting the state of variables. One of the purposes of this paper

JSS (Accepted) Arxiv:2005.04050

UNECE

Example: cleaning SBS data

```
# read data, rules
supermarkets <- read.csv("data/input.csv")
rules        <- validator(.file="data/rules.yaml")

start_log(supermarkets, logger = cellwise$new()) # <- ADD ONE LINE

# error localization, imputation, adjusting
supermarkets <- replace_errors(supermarkets, rules)
A <- is.na(supermarkets)
supermarkets <- impute_mf(supermarkets, . ~ .)
supermarkets <- match_restrictions(supermarkets, rules, adjust=A)

# write output
write.csv(supermarkets, "data/clean_data.csv", row.names=FALSE)
```

Example: cleaning SBS data

```
lumberjack::run_file("cleanup.R")
```

```
## Dumped a log at supermarkets_cellwise.csv
read.csv("supermarkets_cellwise.csv")[100:102, ]
```

```
##     step                      time          srcref
## 100    4 2020-08-31 11:29:01 CEST cleanup.R#20-20
## 101    4 2020-08-31 11:29:01 CEST cleanup.R#20-20
## 102    4 2020-08-31 11:29:01 CEST cleanup.R#20-20
##                                       expression      key  variable old
## 100 supermarkets <- impute_mf(supermarkets, . ~ .) 70799197 total.rev  NA
## 101 supermarkets <- impute_mf(supermarkets, . ~ .) 70799197  turnover  NA
## 102 supermarkets <- impute_mf(supermarkets, . ~ .) 71774143 other.rev  NA
##          new
## 100 626.8672
## 101 598.3333
## 102  50.7900
```

UNECE

Conclusions

UNECE

Take-home messages

Abstraction $\neq$ hiding

Don't hide the details, get rid of them!

Modularity is not enough

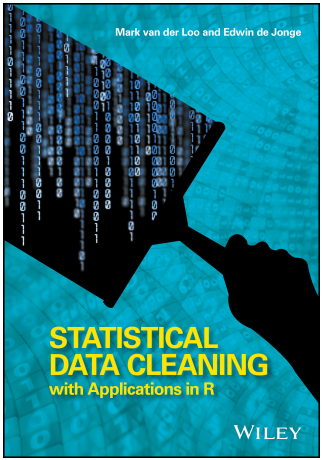
Composability is key!

UNECE

References, tutorials

Tutorials:

- uRos2019
- useR2019
- EESW 2020
- ISM 2020



data-cleaning.org

UNECE