DataForge

SCIENTIFIC DATA PROCESSING FRAMEWORK

Alexander Nozik

Institute for Nuclear Research

MIPT MOSCOW INSTITUTE OF PHYSICS AND TECHNOLOGY

NPM Nuclear Physics Methods

History and motivation

History and motivation. Personal experience

Troitsk nu-mass

- Search for masses of active and sterile neutrinos by the means of analyzing shape of tritium beta-spectrum.
- Complicated many-dimensional spectrum fitting with tight correlated (badly conditioned) covariance matrices.
- Complicated many-step data filtering and transformation requiring a lot of additional experimental parameters and calibrations.

Best results in the world for direct boundary on neutrino mass.

History and motivation. Scientific software

Scientific data analysis is strongly dependent on analysis software

The current scientific software have two major flaws:

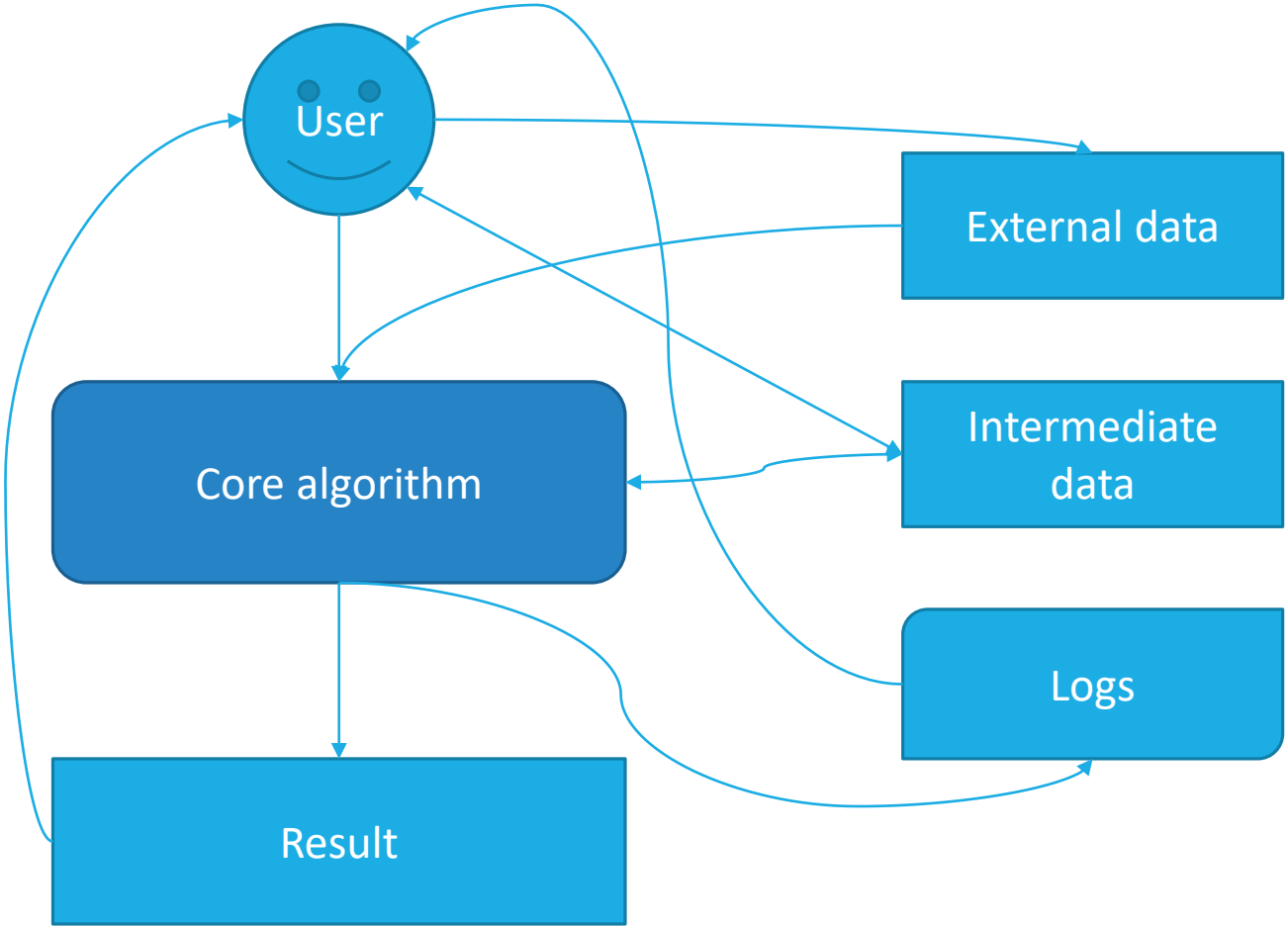
- Legacy platform (FORTRAN/C/C++)
- Legacy architecture (procedural / purely designed OOP)

Mostly developed in 1980-s or early 1990-s!

These programs was good 20 years ago. But programming industry and computer science made a huge progress since then.

Nowadays scientists waste a lot of time doing job the could and should be automated. The only thing they need is a tool, that makes this automation convenient.

History and motivation. The problem



And if you need not one, but 100 different results for different models?..

The DataForge fundamentals

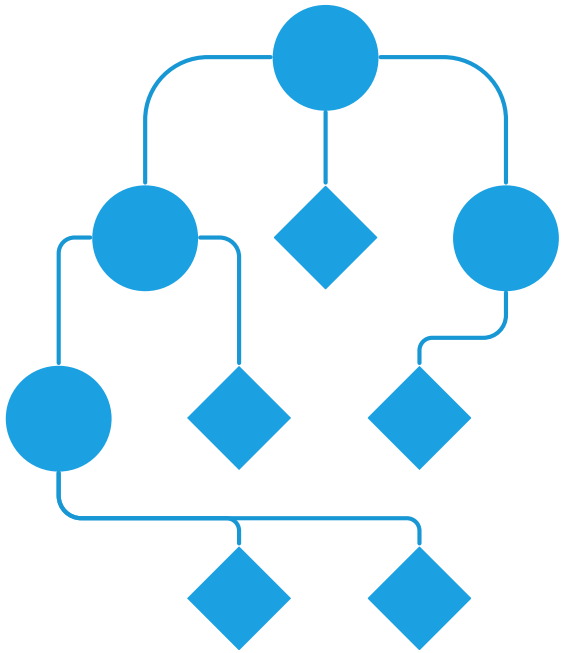
Meta-data processor

1. Data is immutable
2. Everything aside from data itself is meta-data:
   - Information about how data was obtained.
   - Any additional parameters.
   - Information about how data should be processed.
3. Meta-data could have different sources:
   - Intrinsic meta-data. Immutable, generated with data or when data is initialized.
   - Process meta-data. Specific for what you do with the data. Immutable after process started.
   - Context meta-data. Specific for a system. Immutable after framework is initialized.
4. Meta-data layers could be used simultaneously (layering)

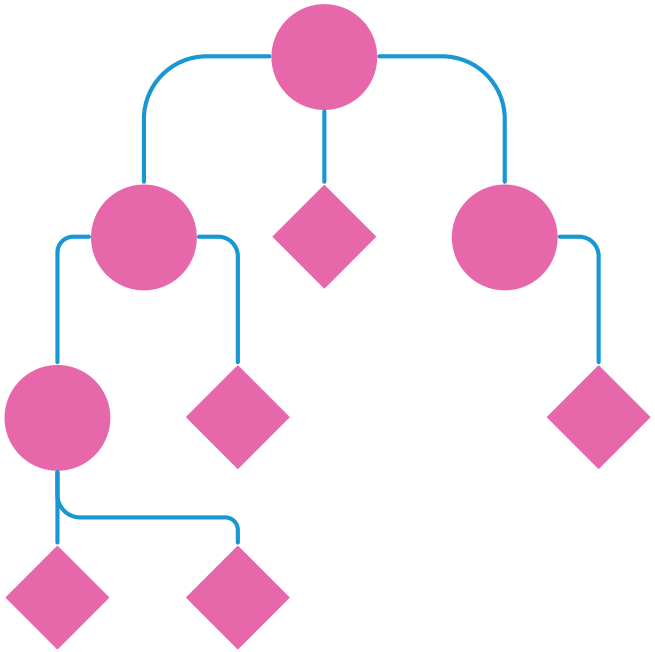
You need only data and meta-data, nothing else!

Meta layering

LAYER 1

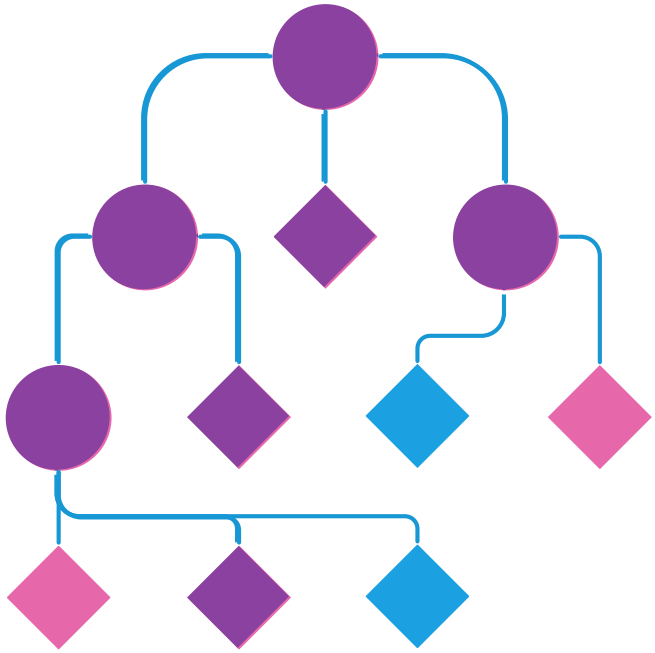


LAYER 2



Meta layering

LAYER 1



LAYER 2

Layering mechanism allows to automatically merge any number of configurations.

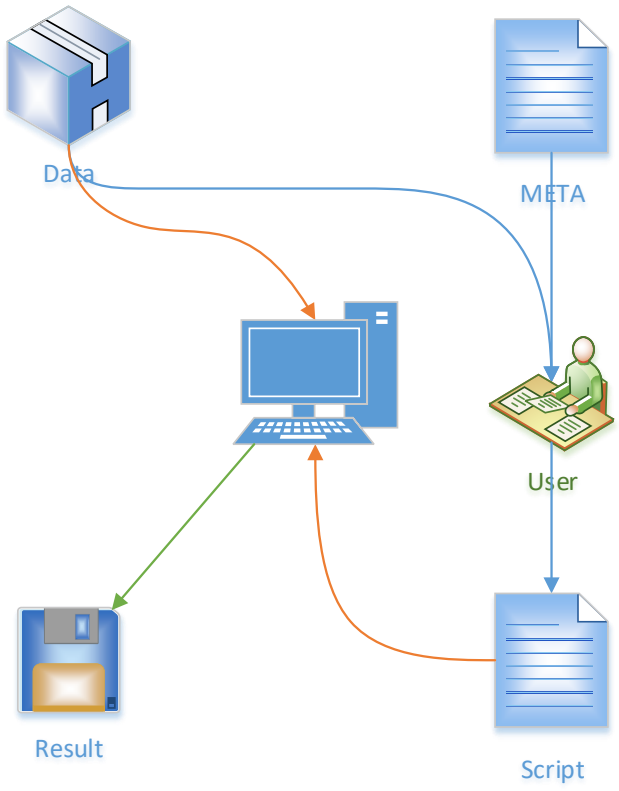
If some value is present at least in one layer, it is used.

If it is present in two or layers, then upper layer value is used.

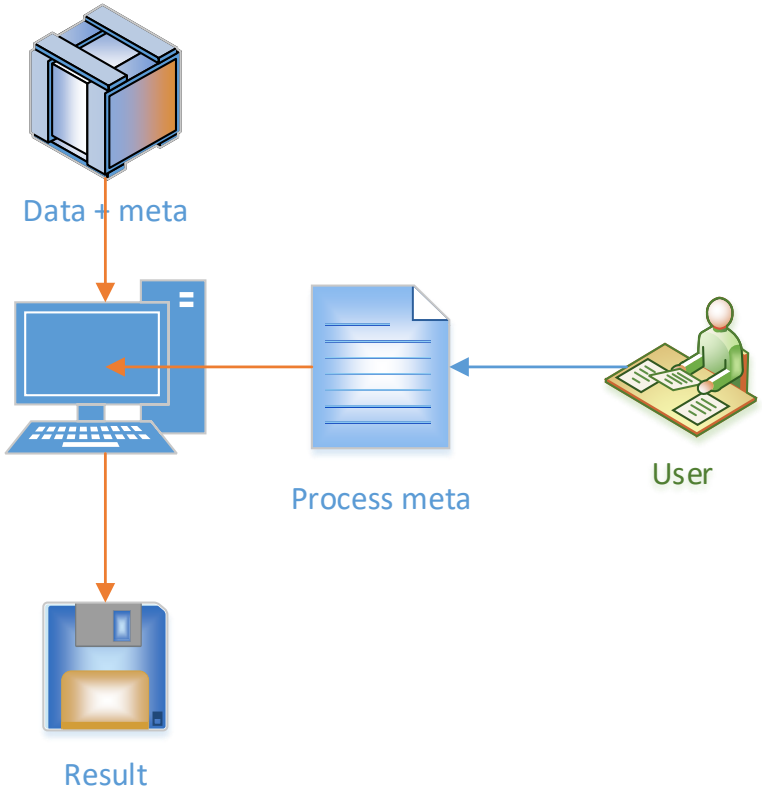
If it is not present, default value is used.

Classic Vs Meta-data processor

CLASSIC



METADATA PROCESSOR



Context encapsulation

How to make framework scalable?

Avoid global variables and states.

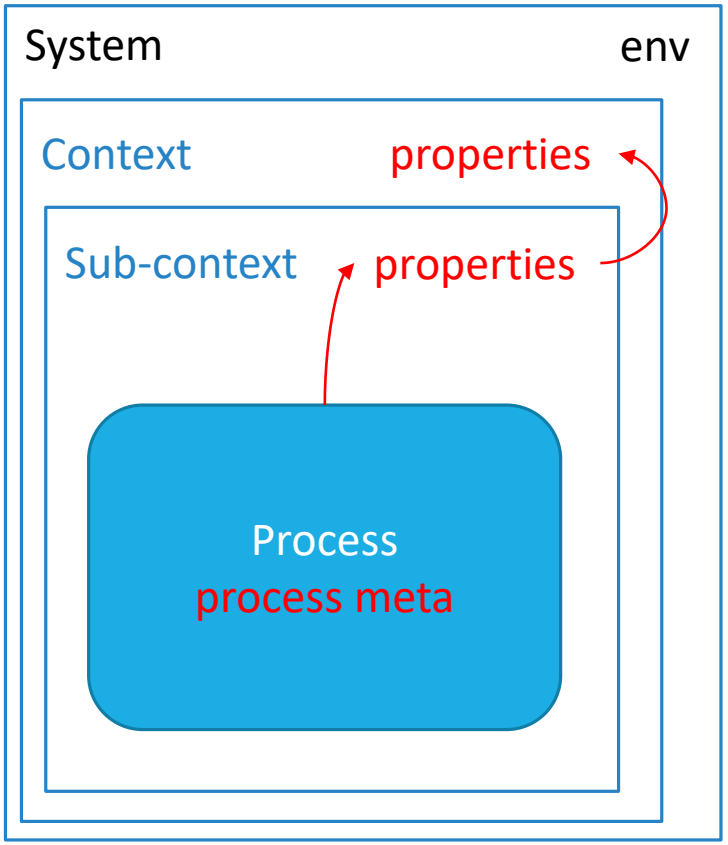
Use Context!

Inspired by

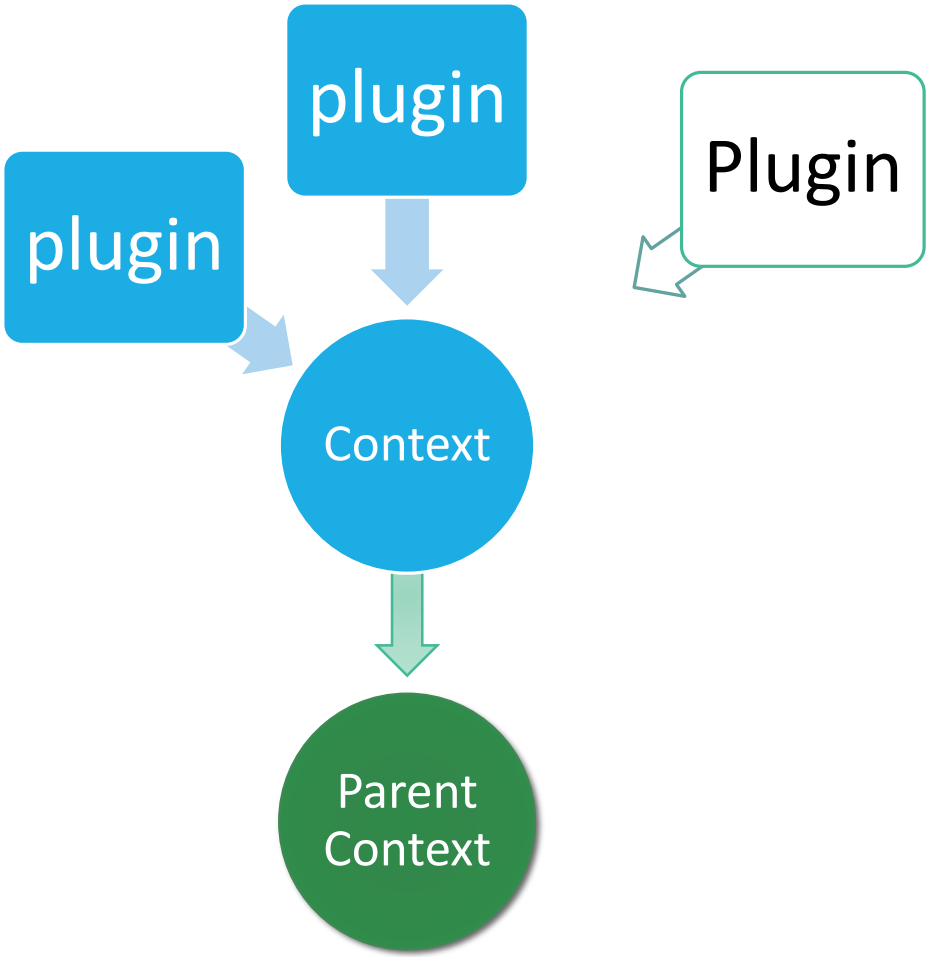
Context is a lightweight way to isolate user logic from environment as far as possible.

Contains current environment properties.

A general Provider implementation. Provider is a class which could provide an object for given Path (see naming conventions).



Context-based modularization



Framework features are stored in context plugins, so application logic can request feature from context.

If feature is not present in the context, parent context is searched. If feature is not present in Global context, it is automatically loaded from plugin repository.

Some features could have multiple implementations.

In other words Context works as a dependency injection server.

Data flow models

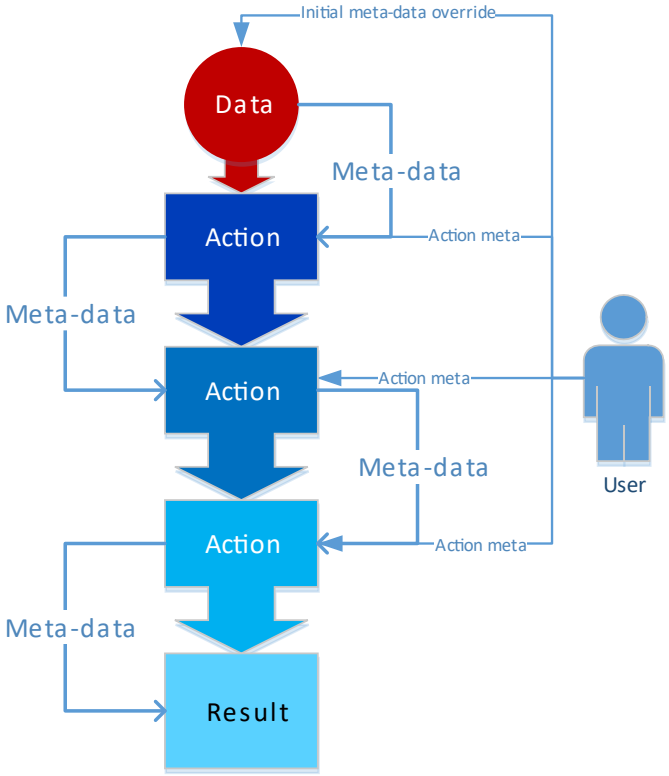
“Push” data flow

User defines which (annotated) data to be used and the order of actions to be performed on this data.

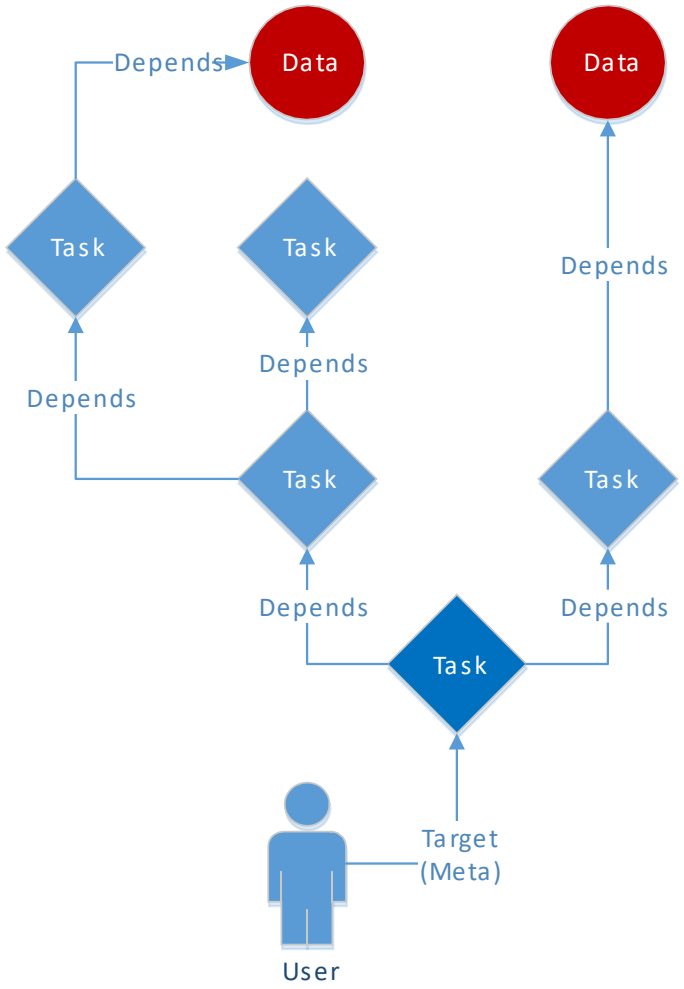
The meta-data is transmitted from one action to another automatically.

Actions could be dynamically type checked before start of the process.

Actions are automatically parallelized if more then one token of data is present.



"Pull" data flow



Task structure is completely declarative. It is similar to build system functioning.

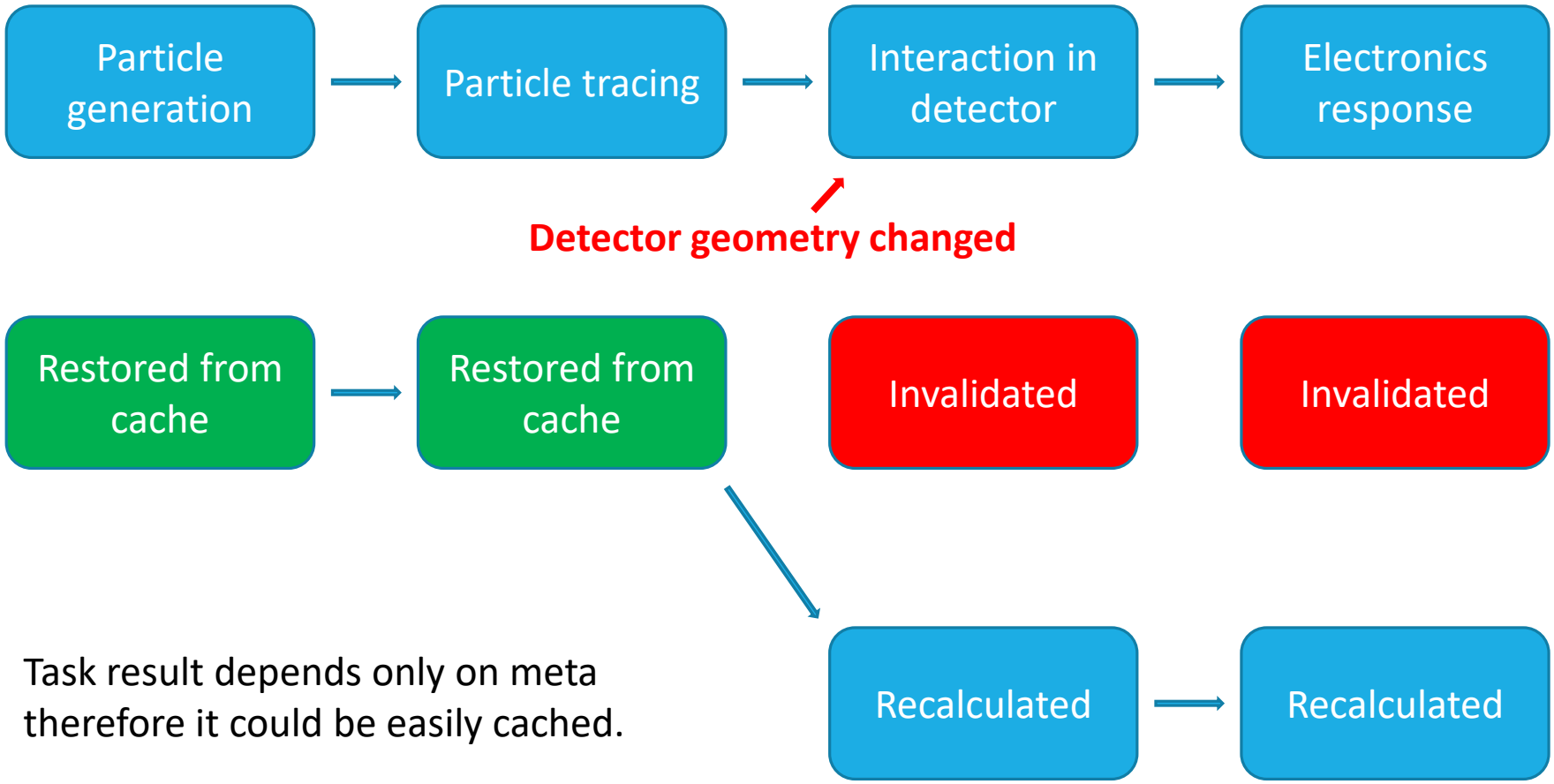
Task is defined by its name and meta-data.

Task infers which other tasks or data it needs to complete and calls it, passing meta-data over.

Task results could be cached!

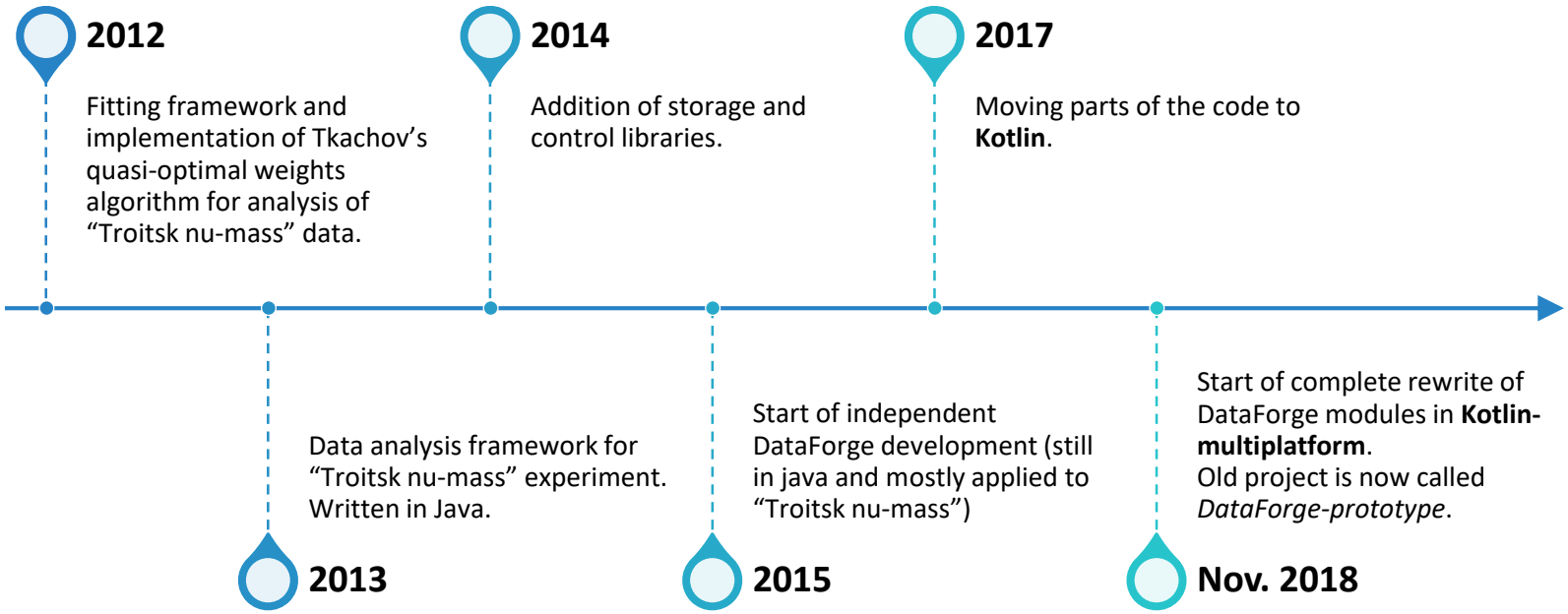
Inspired by Gradle

Task caching



Task result depends only on meta therefore it could be easily cached.

History


2012
Fitting framework and implementation of Tkachov's quasi-optimal weights algorithm for analysis of "Troitsk nu-mass" data.
2013
Data analysis framework for "Troitsk nu-mass" experiment. Written in Java.
2014
Addition of storage and control libraries.
2015
Start of independent DataForge development (still in java and mostly applied to "Troitsk nu-mass")
2017
Moving parts of the code to **Kotlin**.
Nov. 2018
Start of complete rewrite of DataForge modules in **Kotlin-multiplatform**.
Old project is now called *DataForge-prototype*.

Feature map

Legend

- Release version
- Work in progress
- Implemented in prototype
- Planned

DataForge-core	DataForge-storage	Visualization
Core meta	Envelope API	Universal renderer
Meta io	Storage API	Chart API
Contexts	File storage	3d visualization API
Data nodes	SQL storage	GUI utilities
Push model (actions)	NoSQL storage	Miscellaneous
Workspaces (tasks)	HDF5 storage	ROOT/AIDA interop
Output handling	DataForge-control	Remote call
Table API	DAQ API	Cross-language calls
Chart API	DAQ GUI	BAT.jl integration
Scripting support	Message passing	Cluster computing/Spark

Conclusion

- The DataForge is a scientific framework based on modern trends and solutions in programming.
- It introduces a few new concepts into scientific (hep-physics) software:
  - The analysis as a metadata process
  - Declarative description of analysis process (the analysis as a build system)
  - Context isolation of analysis processes
- It is completely and “true” cross-platform (not “compile wherever you want on your on risk”).
- It is modular!
- It has a few very important ideological effects that could be expanded further and can open a whole new world of possibilities for scientific data processing.

Thank you for your attention.

We are looking forward for your feedback and contributions.

Site: http://npm.mipt.ru/dataforge/

Prototype: https://bitbucket.org/Altavir/dataforge

Current: https://github.com/altavir/dataforge-core/tree/dev

Examples

Example: workspace definition

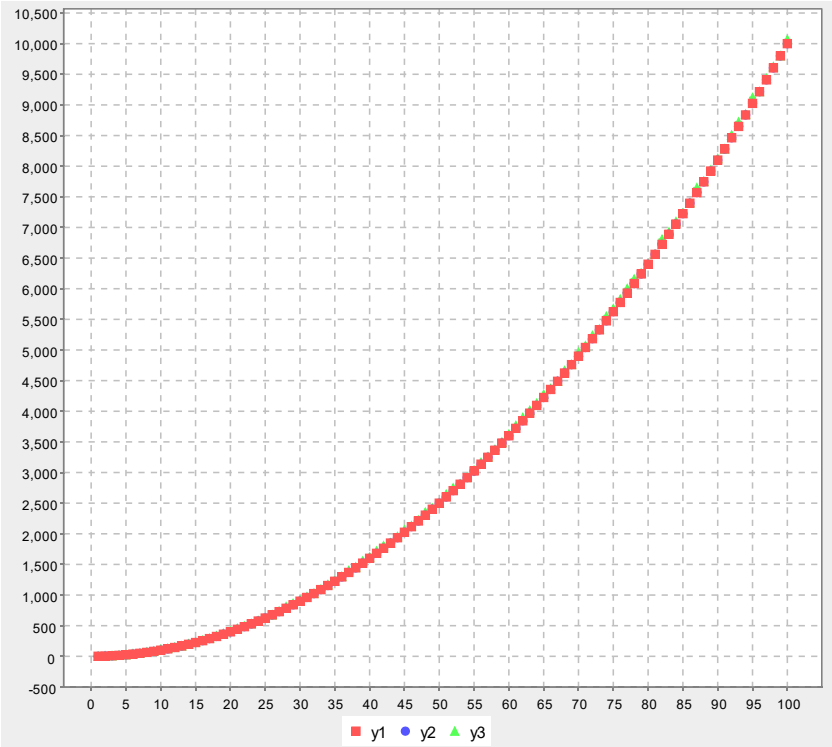
```
context {
    name = "TEST"
    properties{
        power = 2d
    }
}

data {
    item("xs") {
        (1..100).asList() //generate xs
    }
    node("ys") {
        Random rnd = new Random()
        item("y1") {
            (1..100).collect { it**2 }
        }
        item("y2") {
            (1..100).collect { it**2 + rnd.nextDouble() }
        }
        item("y3") {
            (1..100).collect { (it + rnd.nextDouble() / 2)**2 }
        }
    }
}
```

Example: Task definition

```
task custom("table") {
    def xs = input.optData("xs").get()
    def ys = input.getNode("ys")
    ys.dataStream().forEach {
        //yield result
        yield it.name, combine(xs, it, Table.class, it.meta) { x, y ->
            new ColumnTable()
                    .addColumn("x", ValueType.NUMBER, (x as List).stream())
                    .addColumn("y", ValueType.NUMBER, (y as List).stream())
        }
    }
}

task pipe("dif", dependsOn: "table") {
    def power = meta.getDouble("power", context.getDouble("power"))
    return (input as ColumnTable).buildColumn("y", ValueType.NUMBER) {
        it["y"] - it["x"]**power
    }
}
```

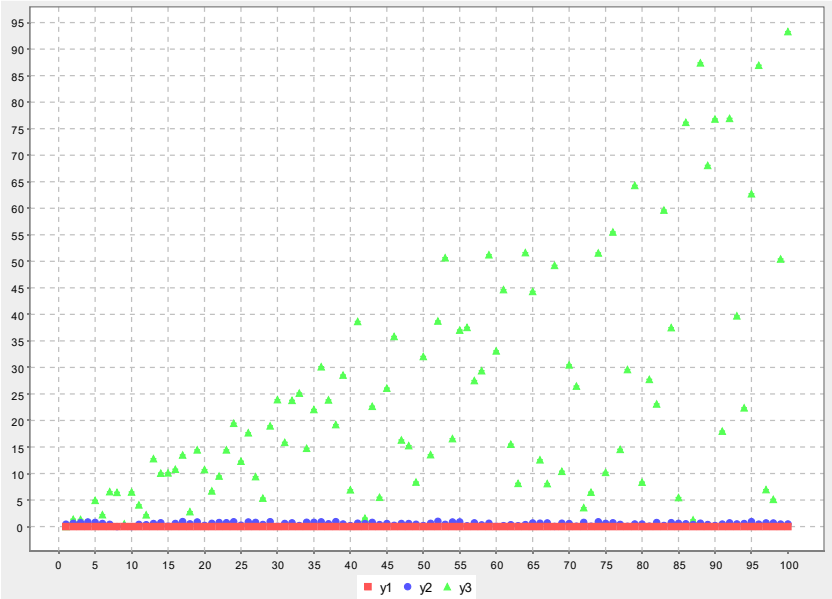
Example: run script

```
//loading plot feature
PlotManager pm = context.getFeature(PlotManager)
PlotFrame frame = pm.getPlotFrame("demo");

workspace.run("table").dataStream().forEach {
    frame.add new PlottableData(it.name, it.meta).fillData(it.get() as Table)
}
```



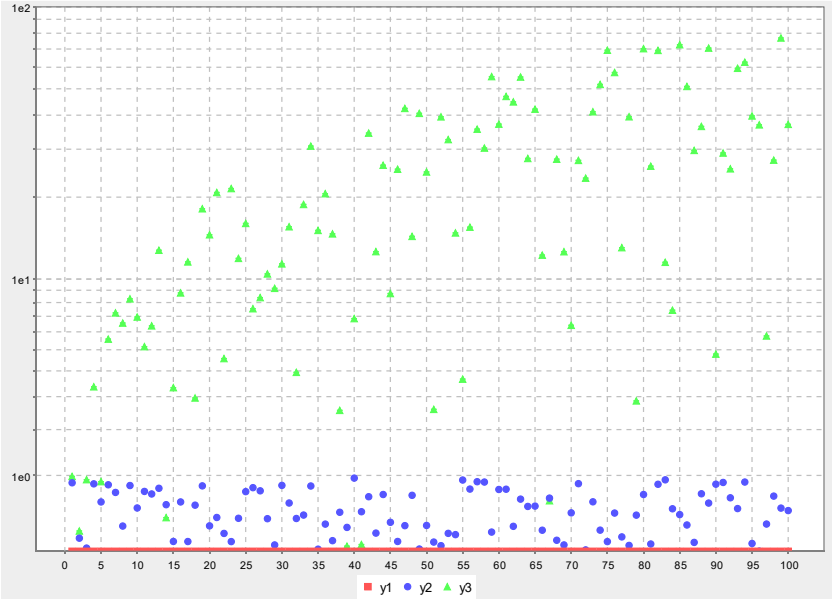
Example: change task

```
//loading plot feature
PlotManager pm = context.getFeature(PlotManager)
PlotFrame frame = pm.getPlotFrame("demo");

workspace.run("dif").dataStream().forEach {
    frame.add new PlottableData(it.name, it.meta).fillData(it.get() as Table)
}
```



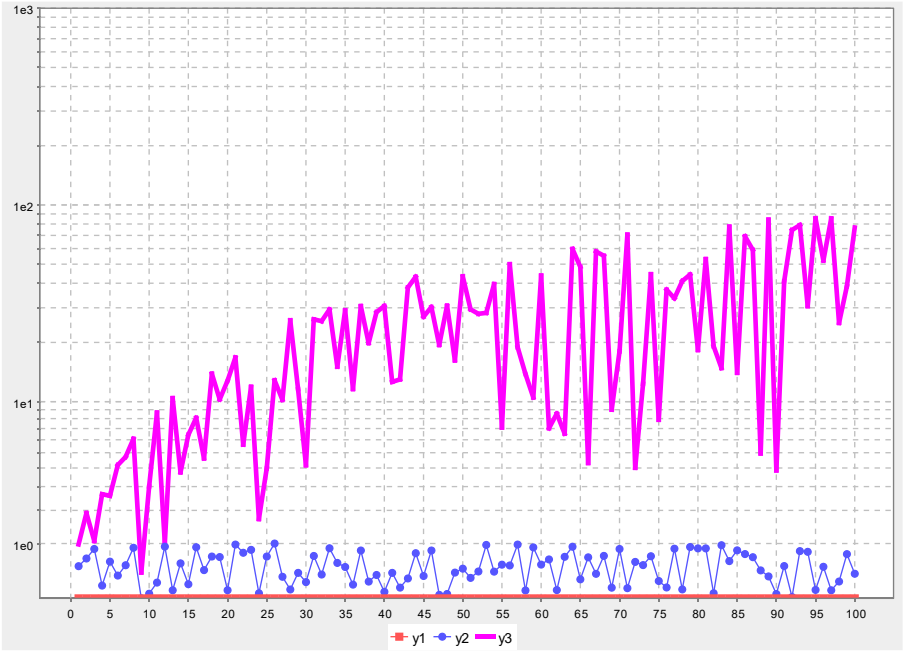
Example: add log scale

```
//loading plot feature
PlotManager pm = context.getFeature(PlotManager)
PlotFrame frame = pm.getPlotFrame("demo");

frame.configure("yAxis.type": "log")

workspace.run("dif").dataStream().forEach {
    frame.add new PlottableData(it.name, it.meta).fillData(it.get() as Table)
}
```



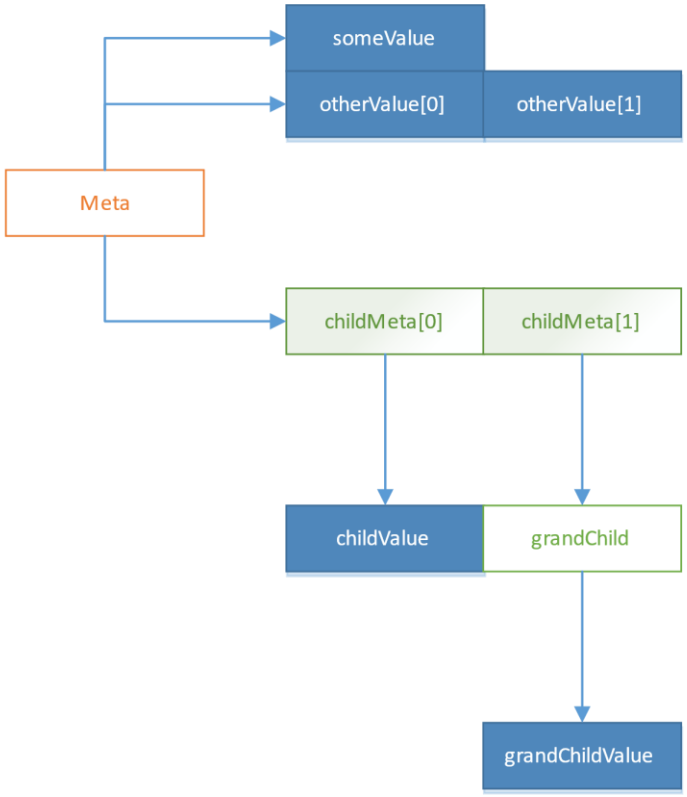
Example: changing plot parameters in data

```
node("ys") {
    meta(showLine: true)
    …
    item("y3") {
        meta(thickness: 4, color: "magenta", showSymbol: false, showErrors: false)
        (1..100).collect { (it + rnd.nextDouble() / 2)**2 }
    }
}
```



Add-ons

Meta-data structure



Meta-data is a tree-like structure.

Each meta node could have named child nodes and named values.

Same-name values and same-name sub-nodes are ordered lists.

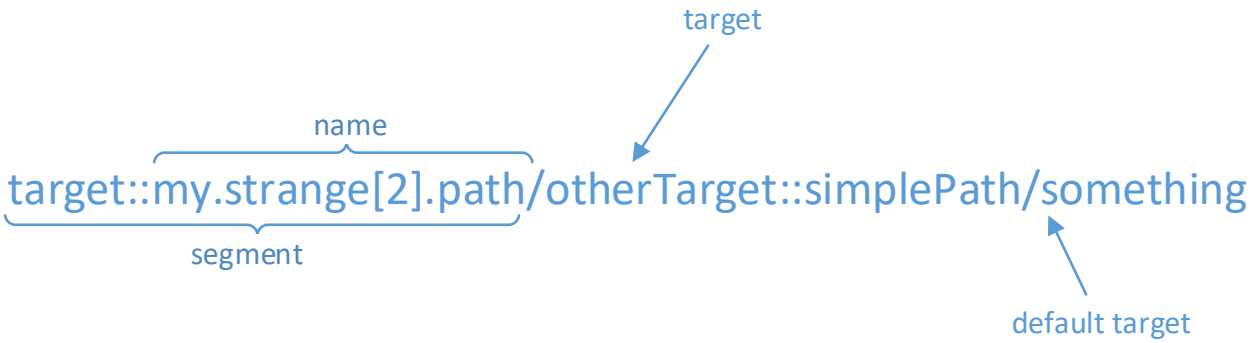
Meta-data does not have fixed textual or binary representation, but could be transformed in and from most popular formats (XML, JSON, etc).

Naming conventions and providers

Simple path: ***token1.token2.token3***

Tokens could include query string in ***[]***

In case the path directs to Provider object, than chain path could be implemented in a following way:



Context is provider. Meta is also provider.