How to add a new validator to *linteddata*

Contents

1 Structure

- describe the structure of this document

2 General architecture

In this section at first the architecture is displayed. Additionally some of the classes and their functionality are explained.
The general structure of the classes, used to implement the *linteddata*, is displayed in figure 1. The classes belong to three different packages:

checks: This package contains all the validators.

JUnitXML: All classes in this package are used to build the structure of [**?**].

Main: The classes from this package are used to execute the tool.

The different classes in the package `checks` are described in section 3. In general all of these classes have a `execute` that must be implemented. Each of the abstract classes overwrites the `execute` of its superclass, except for `FileCheck` as the first check. `Check` is a superclass of `FileCheck` and used to capture the attributes of a check. It can be used to extend the tool later with different types of checks.
Each of the `FileChecks` has one level it applies to, this could also be modelled via the subclass relation, but with the enum `Level`, the user can pass as an argument what kind of validations he wants to execute. A new validator mustn't have the `Level ALL`, this value is only for the user input. Also each one has a `TargetLanguage`. This attribute is used to assign the validator to the corresponding Testsuite in the result.

The abstract subclasses of the different levels each implement the abstract `execute` method of its superclass. In these implementations the argument, that is not `failureDescription`, is further processed and along with the `failureDescription` passed to the new abstract method `execute`.
The package `JUnitXML` contains classes that are needed to represent the elements from [**?**]. To prevent misunderstandings, elements of *Testsuites* are represented with instances of the class `TestsuiteManager`. When adding a new validator, no changes need to be done to this package.

The `Severity` has the three values `ERROR, WARN` and `INFO`. Where `ERROR` is used for critical failures that must be fixed to ensure correctness and functionality. `WARN` represents failures that are non critical but should be fixed for correct behaviour. The weakest value is `INFO` which is for non critical failures that don't affect the correctness but should be fixed.
The `main`-method is contained in `LintedData`. Within this class the command-line arguments are processed and a new instance of `Runner` is created. `Runner` executes the selected checks and stores the result to the destination file. When a new validator is added, it must be added to the method `Runner.createAllChecks`, there are no further changes needed.

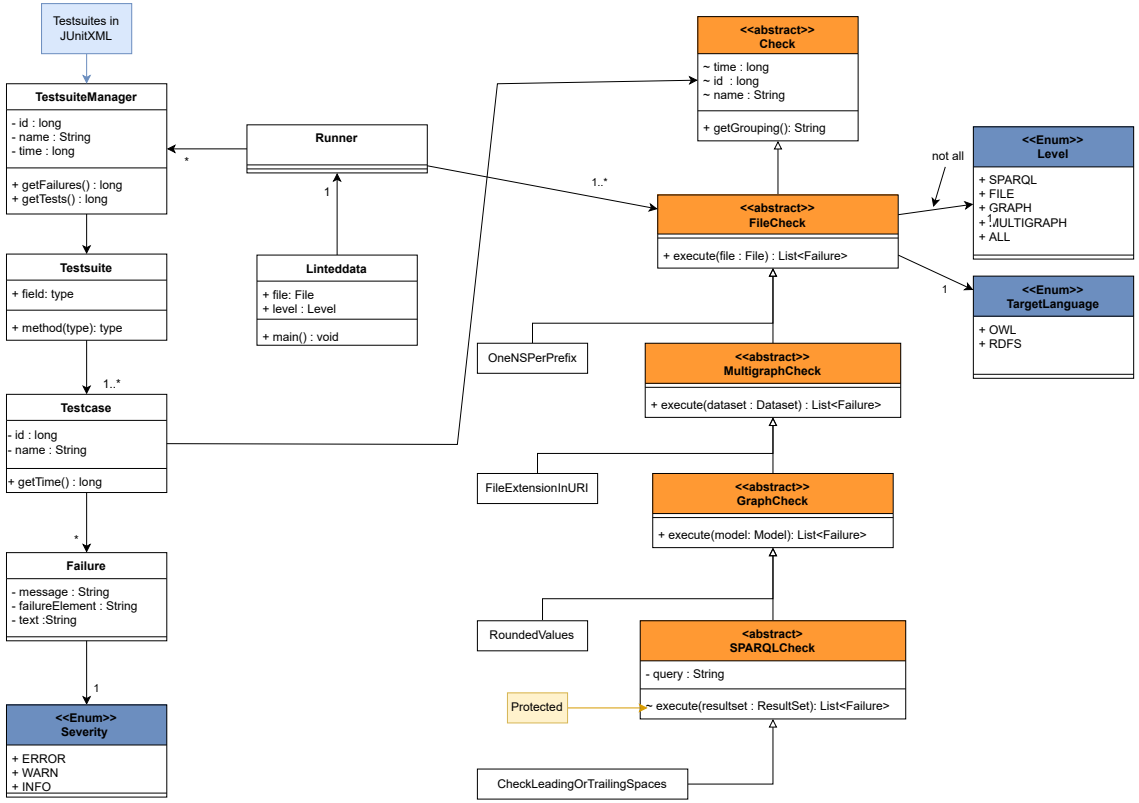

Figure 1: Class diagram of *linteddata*.

In the following section the classes of the package `check` are described more in detail.

- image of the class diagram
- should the classes be described more in detail?

3 Check types

In this section the different types of validators are explained. The aim of this section is it, to understand, when to implement which type of check.

As seen in the class diagram (figure 1), the different types of validators are implemented as subclass from `Check`.

`FileCheck` is the first 'real' check that applies to the structure of the tool. All non abstract subclasses of this class need to process the file in its raw format. The failures detected in those classes can't be detected after parsing the file into a Jena dataset [**?**].
A validator should be realised as a subclass of `MultiGraphCheck` if the problem can be detected after parsing the file, but it is necessary to have access to all models contained in the dataset.
When a problem affects only a single model from Jena contained in the dataset. At this level, it is not possible to access any other model. If this is necessary, the validator should be a subclass of `MultiGraphCheck`.
Although implementing a validator as a subclass of `SPARQLCheck` might not be the best efficient solution, it should be the superclass to choose. If the validator is implemented as SPARQL query, this query can be reused in other frameworks as well.

- describe the different types of test
- SPARQL checks might be implemented more efficient as a different type, but should be implemented as them → can be used in other frameworks
- multi graph level
  - file parsed as dataset
  - information extracted from all models
- graph level
  - executed on default model and all named models
  - check always performs on only one model at the same time → no information about other models at this point
- SPARQL level
  - executed on a single model
  - query = string → attribute
  - `execute` → only processing of the query result into failures

4 Implementation of a new validator

1. choose a corresponding level
2. implementation of the constructor
    - description of the attributes
    - choose severity
3. implementation of the `execute`
    - how to customise `failureDescription`
    -
4. test check
    - JUnit Test
    - test the 'lowest' `execute` function
5. add test to the list of all checks
    - add check in `Runner.createAllChecks()`

5 Example

- documentation of creating a check