# TESTed: a universal judge for evaluating software in an educational context

Niko Strijbol

## Abstract

Writing programming exercises is a time-consuming activity. In this article we focus on one particular aspect: using the same exercise in multiple programming languages. Currently, achieving this requires a manual translation of the exercise to each programming language that the author wishes to support, although most exercises do not make use of language specific constructs. We introduce TESTed, a prototype of a judge for the Dodona platform (an online exercise platform) that is capable of evaluationg submissions in multiple programming languages using one exercise. We describe how the prototype works and discuss limitations and future work.

## 1. Introduction

Technology is becoming increasingly important in our society: the computer is used to solve tasks and problems in more and more domains. As a result of this evolution, students have to become familiar with computational thinking: translating problems from the real world into problems understood by a computer [1]. Computational thinking is broader than just programming, yet programming is a very good way to train students in computational thinking. However, learning to program is often perceived as difficult by students [2]. As solving exercises is beneficial to that learning process, it is important to provide students with exercises that are high in both quality and quantity. This requirement imposes two challenges on educators:

- Educators have to write suitable exercises: they need to take into account what concepts the students are familiar with, how long it takes to solve an exercise, etc.
- The students' submissions must be provided with high quality feedback. This feedback allows students to learn from their mistakes and improve their programming skills (and by proxy their computational thinking skills).

The second challenge often leads educators to use a platform that provides automatic feedback, with at least the correctness of the submission being automatically determined. One such platform is Dodona, which was created by and is maintained by a team at Ghent University.

In this article, we focus on one aspect of the first challenge. There are a lot of programming languages, and each of those languages needs exercises if it will be taught. Currently, educators have to manually translates the exercises from one programming language to the next, even though most exercises don't use programming language specific concepts. In this article we provide an answer the question: is it feasible to create a system where the exercise is written once, but is solvable in multiple programming languages? We start by giving some more background on Dodona, after which we introduce the answer to previous question in the form of TESTed and we finish by discussing limitations and future work.

## 2. Dodona

Dodona is an online platform for programming exercises. It supports multiple programming languages and provides real-time automatic feedback for submissions. Dodona is freely available for schools and is used by multiple courses at Ghent University.

Dodona internally uses the concept of a judge to denote the piece of software responsible for evaluating a submission in a given programming language. This judge is run in a Docker container and communicates with Dodona via a JSON interface. The judge system is flexibel: other types of feedback, aside from correctness, are possible. For example, some existing judges run a linter on the submissions.

## 3. TESTed

TESTed is a prototype of a judge for the Dodona platform, capable of evaluation submissions in multiple programming languages for the same exercise. Although it is a judge for Dodona, it does not depend on Dodona and is independently usable. TESTed does use the Dodona format for input and output, but we believe this not a hindrance in using TESTed elsewhere: the formats are sufficiently flexible to accommodate a wide variety of scenario's.

To discuss the internals of TESTed, it is useful to split it into three main area's of interest:

- The *test plan* and serialization format, which is a specification on how to evaluate a submission for a given exercise.
- The *core* of TESTed, which takes care of communicating with Dodona, generating and running testcode, and evaluating the results.

- The *language configurations*, which are the subsystem responsible for translating the testplan into actual code.

### 3.1. *The test plan*

The test plan is a programming language independent format that specifies how a submission for an exercise must be evaluated. It contains elements such as the different tests, the inputs, the expected outputs, etc. The structure of the test plan is heavily inspired by the Dodona output format and consists of following elements:

**Tabs** A testplan contains multiple tabs, which are the top-level grouping of the tests.

**Contexts** Each tab consists of one or more contexts. A context is an independent execution and evaluation of a submission.

**Testcase** A context consists of one or more testcases. A testcase the evaluation of one input and the resulting outputs. We distinguish two types of testcases:

> **Context testcase** The testcase containing the call to the main function (or script execution). A context has one context testcase at most.
>
> **Normal testcase** Testcases with other inputs, such as function calls or assignments.

**Test** Finally, a testcase contains multiple tests, each for a different output channel. For example, there are tests for `stdout`, `stderr`, return values, exceptions, etc.

One important part of the test plan that deserves further attention is the input. As we've mentioned, each testcase has a single input. There are basically two types of input: `stdin` and a statement. Since our goal is not to create a universal programming language, statements have intentionally been kept simple. A statement is either an expression or an assignment (by which we mean giving a name to the result of an expression). An expression can be a function call, an identifier (if we previously used an assignment) or a literal value.

Additionally, TESTed defines a serialization format for values. This format consists of two pieces: the data type of a value and the encoding of the value. Since the serialization format is also defined in JSON, the encoding is simply a JSON type. The data type of the value is complexer: since we support multiple programming languages, we must support generic data types. To this end, TESTed defines two kinds of data types:

- Basic types, which include integral numbers, rational numbers, booleans, strings, sequences, sets and maps. These are abstract types, and we don't concern ourselves with implementation details for these types. For example, all integer types in C (`int`, `long`, etc.) map to the same integer type in the serialization format.

- Advanced types, which are more detailed (`int64`) or programming language specific. Here we do concern ourselves with implementation details.

The advanced types are associated with a basic type, which acts as a fallback. For example, a `tuple` in Python will be considered equal to an `array` in Java, since they both have the basic type `sequence`. Programming languages can also indicate that there is no support for certain types (e.g. no support for sets in C), in which case the exercise will not be solvable in that programming language if the exercise uses that data type.

### 3.2. *Running an evaluation*

A first part of the core of TESTed is responsible for running the evaluation of an exercise. This evaluation consists of a few steps, as illustrated by TODO and described below:

1. The Docker container for TESTed is started by Dodona, and the submission and configuration are made available to TESTed.
2. The test plan is checked to verify that the exercise is solvable in the programming language of the submission.
3. For each context in the test plan, the test code is generated.
4. The test code is compiled (this step is optional) in one of two ways:

   **Batch compilation** The test code of every context is bundled and compiled together in one compilation. We say that this step results in one executable (even though this is not the case for languages such as Java).

   **Context compilation** The test code for each context is compiled separately. For $n$ contexts, there will be $n$ compilations, resulting in $n$ executables.
5. The result of the previous step (the compilation or the test code itself if there is no compilation) is executed. Each context is executed in a new subprocess, to combat sharing information between contexts.
6. The collected results of the execution are then evaluated. For example, the results contain the produced `stdout`, which will now be evaluated.
7. In the final step, TESTed collects all results and sends them to Dodona.

One might ask why we need two compilation modes. The reason we employ these modes is performance. Since the evaluation of a submission happens in real-time, it is desirable to keep the evaluation short. One bottleneck is the compilation step, which can be quite slow in languages such as Java or Haskell. As a solution, we don't want to compile each context independently: instead we compile all contexts in one go. Note that we still execute the contexts independently, since they are independent of each other.

### 3.3. *Evaluating the results*

A second part of the core is evaluating the results of the previous execution step. As we've mentioned, each output channel is denoted by a different test in the test plan. Currently, TESTed has support for following output channels: `stdout`, stderr, exceptions, return values, created files and the exit code. In most cases, the testplan would specify for each relevant output channel how it should be evaluated, which in most cases boils down to recording the expected value. If an output channel is not relevant, TESTed provides sane defaults (e.g. not specifying stderr means there should be no output on stderr).

Generally, there are three ways in which an output channel can be evaluated:

**Language-specific evaluation** The evaluation code is included in the test code generated for the context and is executed directly in the same subprocess. This mode is intended to check programming language specific aspected of an exercise.

**Programmed evaluation** The evaluation code is executed separately from the test code, in a different process. The results pass through TESTed and are serialized and deserialized. This means the programming language of the submission and the evaluation code does not have to be the same. For example, the evaluation code can be written in Python, and used to evaluate the results of submissions in Java, JavaScript, Haskell, etc.

**Generic evaluation** TESTed has built in support for simple evaluations, like comparing a produced value against an expected value contained in the testplan. For example, if a function call with argument $a$ should result in value $b$, there is no need to write evaluation code, since TESTed can take care of it. There is built-in support for evaluating textual results (stdout, stderr), return values, exceptions and the exit code. The evaluator for values is intelligent and takes the data types into account. If the test plan specifies the return value should be a tuple, TESTed will apply strict comparisons in language supporting tuples (Python and Haskell), but loose comparisons in other languages. This means that for Python and Haskell solutions, only tuples will be accepted. In other languages, all data types with the corresponding basic type will be accepted (such as arrays and lists).

Not all modes are available for all output channels. For example, the language-specific evaluation mode is only available for return values and exceptions.

## 4. Configuring programming languages

Support for a programming language in TESTed consists of three parts:

1. A configuration file
2. A configuration class
3. Templates

The configuration file is used to record properties of the programming language, such as the file extension or which data structures are supported.

The configuration class handles the language-specific aspects of the compilation and execution step. TESTed expects a command to perform during those steps (e.g. for C, the command would along the lines of `gcc -std=c11 file1.c file2.c`).

The third component are the templates. TESTed uses the mako[1] templating system to generate the test code and translate language independent concepts from the test plan into actual code (such as literal values). The templating system works similar to the one used by webapps (such as ERB in Rails, Blade in Laravel or EEX in Phoenix), but generates code instead of HTML/JSON.

## 5. Future work

A first area in which future improvements are possible is the test plan. The test plan currently does not support dynamic scheduling of contexts: the contexts to execute are decided before the execution happens. The test plan can be extended to allow some form of decision, for example deciding if a context should run depending on the result of the previous context. Related are repeated executions of the same context. This is useful for exercises with randomness, where the output is not deterministic. For example, generating a list of random numbers with a minimum and maximum constraint would need multiple runs to validate that random numbers are used.

Another aspect of the test plan are the statements and expressions. While these are currently intentionally simple, this does impose some limitations. For example, an expression with mathematical operators (`5 + 5`) is not possible. The expressions and statements could be extended to provide more capabilities. It is also worth investigating if, when more capabilities are added, it might be worth switching to an existing language and transpile this language to the various programming languages TESTed supports.

The test plan is also fairly verbose. We believe this could be solved by introducing a preprocessing step, that translates a more compact format to the test plan. A promising upcoming format is [3]. We also envision different formats for different exercise types. For example, an IO exercise and exercise with function calls have different requirements.

Secondly, TESTed does not translate programming paradigms. For example, a given exercise might be solved using two top-level functions in Python, but would often be solved with an object in Java. Functional languages also have different paradigms compared to object-oriented languages. It might be worth researching common patterns and their equivalent in other programming languages and providing translation for those common patterns.

Lastly, while the performance is not bad, there is still room for improvements. One area in particular is the programmed evaluations. Each evaluation is currently compiled and executed, even though it is common for multiple contexts to use the same evaluation code.

[1] https://www.makotemplates.org/

## 6. Conclusion

We have presented TESTed, a judge for the Dodona platform, capable of evaluating submissions in multiple programming languages for the same exercise. At the moment, TESTed supports Python, Java, Haskell, C and JavaScript. While there still some limitations on the type of exercises TESTed can evaluate, it can already be used for a variaty of exercises. We believe TESTed can be particularly useful in two scenarios:

- When using simple exercises, for example when teaching programming to novices. Due to their simple nature, these exercises often don't use language specific constructs, making them a good candidate for TESTed. Such exercises might also be useful for students with more programming experience who want to learn a new programming language.
- When the programming language is not important. This is the case, for example, in courses teaching algorithms. The focus of the exercises is learning the algorithmic techniques, not the specifics of a programming language.

## References

[1] B. Bastiaensen, J. De Craemer, Zo denkt een computer, Vlaamse Overheid, 2017.

[2] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, C. Szabo, Introductory programming: A systematic literature review, in: Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018 Companion, Association for Computing Machinery, New York, NY, USA, 2018, p. 55–106. doi:10.1145/3293881.3295779.

[3] P. Conrad, C. Bart, S. Edwards, Programming exercise markup language. URL https://cssplice.github.io/peml/