1 Basic structure

```
1 checking = Checking(environment=detect_environment())
2 checking.add_steps(
3     compile=CompileCpp('solution.cpp'),
4     run_solution=RunSolution(stdout='out'),
5     diff=Diff(),
6 )
7 status_code, detailed_result = checking.run()
```

We can distinguish three key parts:

- Initialization (line 1)

  The only argument to the `Checking` constructor is the environment responsible for running the subsequent tasks. `detect_environment()` is an utility function which automatically detects the environment based on the command line arguments. For more information refer to the Environments section.

- Configuration (lines 2–6)

  This section is responsible for defining the judge pipeline. The steps will be executed in the same order as they were specified in the calls' arguments. The usage of keyword over regular arguments in `add_steps()` is purely optional, designed to provide an option to specify the custom step identifiers.

- Launching the checker (line 7)

  The `run()` method runs the configured pipeline, and returns a 2-tuple, consisting of the final judge verdict code, and a detailed log of the steps ran.

2 Examples

2.1 CMake

```
checking.add_steps(
    cmake=CMake(),
    make=Make(build_dir='build'),
    solution=RunSolution(executable='build/untitled', stdout='out'),
    diff=Diff(),
)
status_code, detailed_result = checking.run()
```

`CMake` without any arguments sets the source directory to the current directory, and a build directory to `build`. `Make` receives the build directory name as the argument, since its default is set to the current directory. Then, assuming the project name in `CMakeLists.txt` is set to `untitled`, the `Makefile` (generated via the `cmake` command) builds the executable named `untitled`. Path to this executable is then passed to the `RunSolution`, and the `stdout` is redirected to the file called `out`. Finally, the `Diff` command – which defaults to `wzo` and `out` as the file names – compares the output with the expected answer. The `wzo` file needs to be provided by the test case, just as the source files required by `CMakeLists.txt`, and the `CMakeLists.txt` file itself.

2.2 JAR files

```
checking.add_steps(
    rename=RenameJavaFile('solution.java'),
    list_jars=ListFiles('**/*.jar', variable_name='jars'),
    compile=CompileJava('**/*.java', compilation_options=[
        '-cp',
        DependentExpr('jars', func=lambda x: '.:' + ':'.join(x))
    ]),
    create_jar=CreateJar('**/*.class', entrypoint='Main'),
    run_jar=RunJarSolution(interpreter_options=[
        DependentExpr(
            'jars',
            func=lambda x: '-Xbootclasspath/a:' + ':'.join(x)
        )
    ], stdout='out'),
    diff=Diff(),
)
status_code, detailed_result = checking.run()
```

`RenameJavaFile` takes the file submitted to the judge – whose name might have been mangled in the process – and renames it, to correspond with the class name it contains. `ListFiles` takes the glob matching all `*.jar` files recursively, and stores all of them in the environment's variable called `jars`. `CompileJava` compiles all matching `*.java` files, while overriding the classpath with a `DependentExpr`. This expression is evaluated just before compilation, using a lambda function that takes only one argument – since there's only one variable being evaluated, `jars` – and produces an output of the form `.:<jar_1>:<jar_2>[...]`. Therefore the executed command will be `javac -cp .:<jar_1>:<jar_2>[...]`. `CreateJar` takes all compiled `*.class` files, and creates a JAR file named `archive.jar` containing a manifest with the entrypoint set to `Main`. `RunJarSolution` runs the JAR archive with the `-Xbootclasspath/a` option generated in a very similar way as previously, and redirects the output to the `out` file, which is then compared with `wzo` file using the `Diff` class. This test case assumes the existence of the `solution.java`, `Main.java`, `wzo`, and any supporting `*.jar` files.

2.3 Automatic source type detection and a custom checker

```
checking.add_steps(
    compile_checker=AutoCompile('checker.cpp'),
    compile_solution=CompileJava('Solution.java'),
    solution=RunJavaClassSolution(class_name='Solution', stdout='out'),
    check=RunChecker('autocompile/run.sh', stdin='out'),
)
status_code, detailed_result = checking.run()
```

`AutoCompile` detects the source file type (in this case `.cpp`), compiles it using `g++`, and creates the `autocompile/run.sh` file, which contains the command needed to run the compiled file. `CompileJava` compiles the `.java` file representing the solution. `RunJavaClassSolution` runs the compiled `.class` file, and redirects the output to `out` file. `RunChecker` finalizes the checking by running the shell file generated by the `AutoCompile` class, with the `stdin` redirected from the solution's output file. Existence of the `checker.cpp` and `Solution.java` files needs to be assured by the test case.

2.4 Limits and privilege deescalation

```
checking.add_steps(
    solution=RunJavaClassSolution(
        class_name='Solution',
        stdout='out',
        limits={
            'time': 2.5,
            'memory': 1024 * 1024 * 10,
        },
        user='nobody',
    ),
)
status_code, detailed_result = checking.run()
```

The `RunJavaClassSolution` class declares the `UsedTimePostcondition` and `UsedMemoryPostcondition`, therefore if the real running time will exceed 2.5 seconds, the exit status returned will be `TLE`. Similarly, if the total RAM usage exceeds 10 MB, checking will return `MEM`. If the script is running as a superuser, dropping privileges is often required – in this case, user called `nobody` will be assigned as the process owner via the `setuid` call.

3 Checking

`Checking` class's responsibility is to provide a readable and clean interface for binding together the two abstractions: Steps and Environments.

`Checking`

`__init__(self, environment)`

Initializes the instance with a given environment, which will be used during the `run()` call.

`add_steps(self, *args, **kwargs)`

Takes the steps that will be run during the `run()` call as the arguments. If the arguments are passed as `kwargs`, the argument names will be used as step names in the execution details and logs. Otherwise, consecutive integers will be assigned as step names. If a step with a given name was already added (i.e. during a previous `add_steps()` call), `TypeError` will be raised.

`run(self)`

Calls the `run_steps()` method on `self.environment` with the previously registered steps.

`format_result(self, result)`

Takes the result of `run()` as an argument, and returns a JSON-compatible dictionary using the `format_execution_status()` method on `self.environment`.

4 Steps

Steps together form a checking *pipeline*, each of them being responsible for a logically separate assignment. They are divided into two categories: *commands* (specifying the program to run) and *tasks* (arbitrary Python code). Each of them returns an optional object with execution statistics. Gathered data varies greatly between the environments, but there are a few properties that should always be accessible. Those include `time`, `memory`, and `cpus` (with subproperties `system` and `user` indicating the time that was used by the processor).

4.1 Commands

Commands are wrappers for the programs normally used in the pipeline: `g++`, `diff`, etc. They take care of making sure that every dependency is properly set up, and reporting the execution status back to the environment.

The base class of every command step is `CommandBase`.

`CommandBase`

`__init__(self, limits, user, group)`

Constructor. The `limits` argument allows the caller to specify the limits for the command execution without having to override `get_limits()` method. The `user` and `group` arguments perform the same function, as per `get_user()` and `get_group()` methods, respectively.

`get_env(self)`

Used to specify step-specific environment variables that should be present during the command execution. Values specified here take precedence over those from the `ExecutionEnvironment.get_env()` method.

`get_limits(self)`

Defines the dictionary containing the limits active during the command execution. It will by called by the `ExecutionEnvironment.run_command_step()` method. The default implementation simply returns the limits passed to the constructor.

`get_command(self)`

Abstract method. Returns a list consisting of an executable and its arguments that will be run.

`get_stdin_file(self)`

Returns a path to the file that will be redirected as the `stdin` stream to the program. Default: `None`.

`get_stdout_file(self)`

Returns a path to the file to which the `stdout` of the program will be redirected to. Default: `logs/<step_name>_stdout.txt`.

`get_stderr_file(self)`

Returns a path to the file to which the `stderr` of the program will be redirected to. Default: `logs/<step_name>_stderr.txt`.

`postconditions(self)`

Returns a list of validators, consisting of the 2-tuples: `(validator, exit_status)`. Each of the validators will be called after the command execution. Refer to the postconditions description for more information about their interaction.
Default: `[]`

`verify_postconditions(self, result)`

Iterates over the list returned from the `postconditions()` method, calls each one of them with the execution statistics as an argument, and returns the appropriate exit status if they return `False`.

`prerequisites(self)`

Returns a list of validators, that will be called before the command execution. See prerequisites description for more information.
Default: `[]`

`verify_prerequisites(self, environment)`

Iterates over the list returned from the `prerequisites()` method, calls each one of them with the environment as an argument, and raises the `PrerequisiteException` if they return `False`.

`set_name(self, name)`

Allows the step to store the name that it was assigned at `Checking.add_steps()` call. It can then be used to better identify the outputs of the command execution (e.g. prefixing filenames).

`get_configuration_status(self)`

Returns a 2-tuple containing the information if the command can be run, and an exit status if it cannot. Useful for the aggregate steps, where the command that needs to be run occurs to be undefined (e.g. because the file type passed as an argument is not recognized).
Default: `(True, None)`

`get_user(self)`

Returns the name of the user that the command should be run as.

`get_group(self)`

Returns the name of the group that the command should be run as.

4.1.1 Pre-defined commands

`CompileBase`

Base class of every compilation command. Receives the compiler, files to be compiled, and compilation options as the constructor arguments, and prepares appropriate prerequisites (`ProgramExistsPrerequisite`, `FileExistsPrerequisite`, `NonEmptyListPrerequisite`), postconditions (`ExitCodePostcondition` → `CME`), and a command to be called.

`CompileNasm`

Inherits from the `CompileBase` class, specifies the `nasm` compiler and `-felf64` as the default compilation options.

`CompileC`

Inherits from the `CompileBase` class, specifies the `gcc` compiler.

`CompileCpp`

Inherits from the `CompileBase` class, specifies the `g++` compiler.

`CompileCSharp`

Inherits from the `CompileBase` class, specifies the `mcs` compiler and `-t:exe -out:main.exe` as the default compilation options.

`CompileGo`

Inherits from the `CompileBase` class, specifies the `gccgo` compiler.

`CompileHaskell`

Inherits from the `CompileBase` class, specifies the `ghc` compiler.

`CompileJava`

Inherits from the `CompileBase` class, specifies the `javac` compiler.

`CreateJar`

Takes the files that will be packed into a JAR archive, output file, manifest and an entrypoint as the constructor arguments. Prepares the standard prerequisites (`ProgramExistsPrerequisite`, `FileExistsPrerequisite`, `NonEmptyListPrerequisite`), postconditions (`ExitCodePostcondition` → `CME`), and uses the `jar` program in the returned command.

`Link`

Takes the object files that will be linked together (using the `ld` program) and the output file as the constructor arguments. Specifies the standard prerequisites (`ProgramExistsPrerequisite`, `FileExistsPrerequisite`, `NonEmptyListPrerequisite`), and postconditions (`ExitCodePostcondition` → `CME`).

`Make`

Takes the target and the build directory as the constructor arguments. Specifies only the `ProgramExistsPrerequisite` prerequisite, and sets the `ExitCodePostcondition` → `CME` postcondition.

`CMake`

Takes the source and the build directories as the constructor arguments. Specifies only the `ProgramExistsPrerequisite` prerequisite, and sets the `ExitCodePostcondition` → `CME` postcondition.

`Run`

Base class for all of the commands that are running a program. Receives the executable, its command line arguments, and paths to the files representing the standard I/O streams. Sets the `ProgramExistsPrerequisite` on the executable, and `FileExistsPrerequisite` on the `stdin`. No postconditions are set.

`RunSolution`

Inherits from the `Run` class and sets `./a.out` as the default executable. This class, and also all other following the `*Solution` convention define three postconditions: `UsedTimePostcondition` → `TLE`, `UsedMemoryPostcondition` → `MEM`, and `ExitCodePostcondition` → `RTE`.

`RunCSharp`

Inherits from the `Run` class and sets `mono` as the default executable. Takes an EXE file to be run and the interpreter options as the arguments, and sets the `FileExistsPrerequisite` on that EXE file.

`RunPSQL`

Inherits from the `Run` class and sets `psql` as the default executable. Receives a SQL file and the connection configuration: user, password, host and database name as the arguments.

`RunJavaClass`

Inherits from the `Run` class and sets `java` as the default executable. Takes a class file to be run and the interpreter options as the arguments, and sets the `FileExistsPrerequisite` on that class file.

`RunJar`

Inherits from the `Run` class and sets `java` as the default executable. Takes a JAR file to be run and the interpreter options as the arguments, and sets the `FileExistsPrerequisite` on that JAR file.

`RunPython`

Inherits from the `Run` class and sets `python` as the default executable. Takes a Python script to be run and the interpreter options as the arguments, and sets the `FileExistsPrerequisite` on that Python script.

`RunShell`

This class is synonymous to the `Run` class.

`Diff`

Command to compare two files while ignoring trailing spaces. Takes these two files as the arguments, sets the `FileExistsPrerequisite` on them, and defines the `ExitCodePostcondition` → `ANS` postcondition.

`RunChecker`

Inherits from the `Run` class and sets the `ExitCodePostcondition` → `ANS` postcondition.

`ExtractArchive`

Command to extract various types of archives. Receives the archive, optionally its type and a directory to extract to as the arguments. Declares the `ProgramExistsPrerequisite` on the appropriate executable (e.g. `unzip` or `tar`), and the `ExitCodePostcondition` → `EXT` postcondition. Return the `EXT` exit status if the archive type is not recognized.

4.2 Tasks

Tasks are supposed to perform the work that can't be easily done using shell commands, like renaming `*.java` files. They also make passing unknown (before runtime) arguments to the commands possible, using the `DependentExpr` syntax.

The base class of every task step is `TaskBase`.

`TaskBase`

`execute(self, environment)`

Represents the Python code part of the checking pipeline. It will be called by the `ExecutionEnvironment.run_task_step()` method. Returns a 2-tuple: `(exit_status, execution_statistics)`. Both fields can be `None`.

`set_name(self, name)`

Allows the step to store the name that it was assigned at `Checking.add_steps()` call. It can then be used to better identify the outputs of the command execution (e.g. prefixing filenames).

`prerequisites(self)`

Returns a list of validators, that will be called before the command execution. See prerequisites description for more information.
Default: `[]`

`verify_prerequisites(self, environment)`

Iterates over the list returned from the `prerequisites()` method, calls each one of them with the environment as an argument, and raises the `PrerequisiteException` if they return `False`.

4.2.1 Pre-defined tasks

`AutoCompile`

Detects the compiler and compiles the file passed as an argument, based on its extension. Creates an `autocompile/run.sh` file that contains the shell command that can be then run. Useful when creating custom checkers - code written in one language can then be quickly replaced with another written in second language, with no or only minimal changes in the judge script itself. The prerequisites set are the same as of the detected command that will be used to compile the file internally.

`RenameJavaFile`

Renames the Java class file given as the argument, to correspond to the public class and package contained in that file. Implementation was based on one of the existing Satori judges.

`ListFiles`

Takes glob patterns and a environment's variable name (from where it can be retrieved using `DependentExpr`) as the arguments. Evaluates the glob expressions and stores the result using `ExecutionEnvironment.set_variable()` method.

`CreateUser`

Creates the user with name received as an argument, using the `useradd` program.

`Setuid`

Receives a file as an argument and sets the `setuid` bit on it.

4.3 DependentExpr

Commands' arguments support a special mechanism, called `DependentExpr`. It represents a lazy object, that takes an unknown number of variable names, and a function which is used to reduce the contents stored (via the `ExecutionEnvironment.set_variable()` method) under those names to a single object. For the most common use case, where only one variable name is given and no reduction is needed, the function argument can be omitted. `DependentExpr` will be evaluated only just before the command execution.

5 Validators

Validators provide a way to verify that the state of an environment is as expected. There are two types of validators: those executed before a step (*prerequisites*), and those executed after (*postconditions*).

Prerequisites are called with an environment as the only argument. They return `True` when they find the state of this environment acceptable, and `False` otherwise. If any of the conditions is not met, `PrerequisiteException` is raised from the `verify_prerequisites()` method, and the checking terminates. Implementation of a prerequisite can be either a function, or a class implementing the `__call__()` method. Classes are preferred because of their ability to be passed arguments in their constructors, and the possibility of overriding the `__repr__()` behaviour, resulting in a better exception message in case the requirement is not satisfied.

Postconditions are given the step's execution statistics as their argument, and return `True` if they find them acceptable, and `False` otherwise. Each step declares a list of applicable postconditions, consisting of the 2-tuples in the form `(postcondition, exit_status)`. When first from them fails to meet its criteria, `exit_status` is returned from the `verify_postconditions()` method, and consequently as the final exit status of the step. This in turn will cause the checking to halt immediately, and no following steps will be run. Similarly to the prerequisites, their implementation can be either a function, or a class implementing the `__call__()` method – with the same constructor advantages as previously.

The following validators are implemented:

`FileExistsPrerequisite(file)`

Calls the `file_exists()` validator from `ExecutionEnvironment.Validators`.

`ProgramExistsPrerequisite(file)`

Calls the `program_exists()` validator from `ExecutionEnvironment.Validators`.

`NonEmptyListPrerequisite(list)`

Makes sure that the given list is not empty. This is especially useful when dealing with source files using glob patterns, which tend to expand to an empty list in case of a mistake - even when there are no wildcard characters. Adopting this validator can result in an earlier detection of the problem.

`ExitCodePostcondition(allowed_codes)`

Checks whether the command have terminated with a code that is present in the `allowed_codes`. The argument defaults to `[0]`, i.e. all non-zero exit codes are forbidden.

`UsedTimePostcondition(time)`

Checks whether the command took less than `time` seconds to execute.

`UsedMemoryPostcondition(memory)`

Checks whether the command allocated less than `memory` bytes during the execution.

`PSQLErrorPostcondition()`

Opens the `stderr` file from execution statistics, if it exists, and ensures that the `'^.*ERROR:'` expression is not matched. Implementation was based on one of the existing Satori judges.

6 Environments

Environments define how the steps are run, what limits are applied to them, and how the processes are monitored. They are also responsible for implementing various checks that validators might want to use, but their exact implementation would depend on the environment's specification.

There are three predefined environments, `LocalComputer`, `PsutilEnvironment` and `KolejkaObserver`.

`LocalComputer` is supposed to provide a minimal support for running the judge without installing any additional packages (provided `/usr/bin/time` is available). It doesn't support any limits, and should be used solely for debugging/testing purposes.

`PsutilEnvironment` is an slightly enhanced environment, which uses the `psutil` module and a separate thread to continuously query the step being run for used resources.

`KolejkaObserver` uses the `kolejka-observer` package, and is recommended for any serious checking systems, as it provides the greatest flexibility in terms of the available limits.

The utility function `detect_environment()` can be used to automatically select the environment, based on the command line arguments.

In order to create an own environment, its implementation has to implement all abstract methods, i.e. `run_command()` and `format_execution_status()`.

`ExecutionEnvironment`

This is the base class, defining common methods for the environments.

`__init__(self, output_directory)`

Constructor, which takes as an argument the directory where all files created by the checking run will be stored. Commands will have this directory set as a current working directory. Note that the executed programs aren't limited by this setting when they don't follow the working directory, especially if they use absolute paths. The `Validators` nested class is instantiated with the environment as an argument and assigned to a variable, creating a circular dependency between these two.

`set_limits(self, **kwargs)`

Filters the limits passed as the arguments, based on the `self.recognized_limits` variable, and prints out a warning on `stderr` for each unrecognized one. The identified limits are saved to be used during the following `run_command()` calls, until the next `set_limits()` invocation.

`run_steps(self, steps)`

Executes the received steps one by one, halting immediately when any of the steps returns an exit status (e.g. `CME`). Returns a 2-tuple containing the final status (either one of the exit statuses, or `OK`), and a dictionary with execution statistics for each ran step.

`run_command_step(self, step, name)`

Responsible for running the command step, which consists of the following parts:

- verifying that step is configured correctly
- verifying the prerequisites are met
- setting the limits requested by the step (see `CommandBase.get_limits()`)
- evaluating the `DependentExpr` expressions
- calling the `run_command()` method
- checking the postconditions
- restoring the old limits

`run_command(self, command, stdin, stdout, stderr, env, user, group)`

Abstract method. Responsible for running the specified command within the appropriate launch configuration, consisting of standard input/output files (handles opened from `stdin`, `stdout`, `stderr` arguments, all of type `pathlib.Path`), environment variables (`env`), process permissions (`user` and `group`) and limits (from previous `set_limits()` call). Returns the optional execution statistics object.

`run_task_step(self, step, name)`

Responsible for running the task step, which consists of the following parts:

- verifying the prerequisites are met
- calling the `execute()` method

`get_env(self)`

Returns the dictionary of environment variables that will be passed to the spawned process. Steps can expand and modify the mapping by overriding the `CommandBase.get_env()` method.

`set_variable(self, variable_name, value)`

Used by the tasks to store any value that should be accessible by the command steps, but is undetermined before run-time.

`format_execution_status(cls, status)`

Abstract classmethod. Responsible for serializing the execution statistics data into a dictionary containing solely JSON-compatible types. Useful for logging.

`get_path(self, path)`

Responsible for returning a path uniquely determined by the argument, that is a subdirectory of `self.output_directory`.

`get_file_handle(file, mode)`

Creates all required parent directories of `file`, if they don't exist. Returns a file handle opened with the specified mode.

`Validators`

Class specifying the environment-specific validators, available for use mainly in the prerequisites. When requesting an unknown validator, an no-op function is returned instead, to ensure maximum compatibility while switching between multiple environments.

`LocalComputer`

`recognized_limits = []`

`run_command(self, command, stdin, stdout, stderr, env, user, group)`

Runs the command using the `/usr/bin/time` tool to measure the time and memory used. Returns the `LocalComputer.LocalStats` object containing execution statistics.

`format_execution_status(cls, status)`

Implements the `ExecutionEnvironment.format_execution_status()` method.

`LocalStats`

Object representing the execution statistics. Contains three properties: `time`, `memory` and `cpus`.

`Validators`

Inherits all validators from the `LocalExecutionEnvironmentValidatorsMixin`.

`PsutilEnvironment`

`recognized_limits = ['cpus', 'cpus_offset', 'time', 'memory']`

`run_command(self, command, stdin, stdout, stderr, env, user, group)`

Runs the command using the `psutil.Popen` function, and starts a separate thread monitoring the resource usage of the launched process (see `monitor_process()`). Returns the `PsutilEnvironment.LocalStats` object containing execution statistics.

`monitor_process(self, process, execution_status)`

Sets the `cpu_affinity` limit on the process passed as an argument, then proceeds to query the time and memory usage each 0.1s. Kills the process if it exceeds the `time` or `memory` limits. After the program finishes its execution, sets the gathered statistics on the `execution_status` argument.

`format_execution_status(cls, status)`

Implements the `ExecutionEnvironment.format_execution_status` method.

`LocalStats`

Object representing the execution statistics. Contains three properties: `time`, `memory` and `cpus`.

`Validators`

Inherits all validators from the `LocalExecutionEnvironmentValidatorsMixin`.

`KolejkaObserver`

`recognized_limits = ['cpus', 'cpus_offset', 'pids', 'memory', 'time']`

`run_command(self, command, stdin, stdout, stderr, env, user, group)`

Runs the command using the `observer.run()` function from the `kolejka-observer` package. Returns an enriched `CompletedProcess` object, containing execution statistics.

`format_execution_status(cls, status)`

Implements the `ExecutionEnvironment.format_execution_status()` method.

`Validators`

Inherits all validators from the `LocalExecutionEnvironmentValidatorsMixin`.

`detect_environment()`

Returns the environment based on the command line arguments.

`--local` (*default*) – `LocalComputer`

`--psutil` – `PsutilEnvironment`

`--kolejkaobserver` – `KolejkaObserver`

Remaining arguments are then passed to the environment-specific parsers and, if recognized, to the environment constructor as `kwargs`.

`LocalExecutionEnvironmentValidatorsMixin`

Defines the methods that are shared between all environments running on a local file system. Currently two validators are implemented:

`file_exists(self, file)` - checks if the file exists in the file system

`program_exists(self, file)` - checks if the program exists in the file system