Software Engineering

WS 2022/23, Assignment 02

Prof. Dr. Sven Apel
Annabelle Bergum
Sebastian Böhm
Christian Hechtl

Handout: 02.12.2022
Handin: 16.12.2022 23:59 CET

Organizational Section:

- The assignment must be accomplished by yourself. You are not allowed to collaborate with anyone. Plagiarism leads to failing the assignment.
- The deadline for the submission is fixed. A late submission leads to a desk reject of the assignment.
- We provide a project skeleton that must be used for the assignment. The skeleton can be downloaded from the CMS.
- The submission must consist of a *ZIP* archive containing only the project folder (i.e., the folder included in the provided skeleton)
- Any violation of the submission format rules leads to a desk reject of the assignment.
- Questions regarding the assignment can be asked in the forum or during tutorial sessions. Please do not share any parts that are specific to your solution, as we will have to count that as attempted plagiarism.
- If you encounter any technical issues, inform us immediately.

Task 1 [30 Points]

Your task is to implement a simple configurable database in SCALA[1] using different implementation techniques. The database is a simple key-value store that allows storing a value given some key. We provide an implementation using C preprocessor directives which acts as a template for the other implementations. The logic for all features is already present in this implementation so you can focus on the different implementation techniques rather than on the logic. We highly recommend to solve the three tasks in order since this gives you a more gentle introduction to SCALA before using more advanced concepts. If you are unfamiliar with the language we recommend taking the *tour of Scala*[2].

a) Implement the configurable database using **runtime parameters** in the file `runtime/Database.scala`. We provide the database's interface and a configuration class that gets passed to the constructor of the database class. Your implementation must change its behavior based on the values in the configuration object.[10 Points]

b) Implement the configurable database using the **decorator pattern** in the file `decorator/Database.scala`. We provide the component interface as well as tests that specify how the concrete component and decorator classes must be named and how they can be combined. [10 Points]

c) Implement the configurable database using **traits**[3] **and class composition with mixins**[4] in the file `traits/Database.scala`. We provide the database's interface as well as tests that specify how the traits must be named and how they can be combined.
Note: The logging feature is cross-cutting to the other features and, hence cannot be implemented as a single trait with the given `Database` interface. Instead, you should create one separate logging-trait for each of the other traits you create (e.g. `trait LoggingWithRead extends Read`). [10 Points]

[1] `https://www.scala-lang.org/`
[2] `https://docs.scala-lang.org/tour/tour-of-scala.html`
[3] `https://docs.scala-lang.org/tour/traits.html`
[4] `https://docs.scala-lang.org/tour/mixin-class-composition.html`

Grading Your submission will be graded based on the following criteria:

- Your submission must be in the correct format, i.e., a *ZIP* archive with the same layout as the project skeleton (results in 0 points if violated).
- Your submission must compile, i.e., the command `sbt compile` must succeed (results in 0 points if violated).
- Each subtask must be implemented using the specified implementation technique (results in 0 points for the subtask if violated).
- We run unit tests (the ones provided + additional tests) against your submission. Points will be awarded based on passed tests.

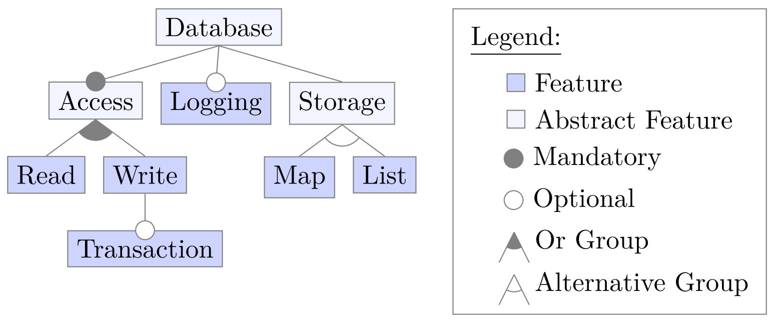
Project Skeleton You must implement your solution based on the provided project skeleton. The skeleton has the following structure:

```
 DatabasePreprocessor.scala.txt
├── build.sbt
└── src
    ├── main
    │   └── scala
    │       ├── decorator
    │       │   └── Database.scala
    │       ├── runtime
    │       │   └── Database.scala
    │       ├── traits
    │       │   └── Database.scala
    │       └── utils
    │           ├── Exceptions
    │           └── Storage
    └── tests
        └── scala
            └── *DatabaseTest.scala
```

The file `DatabasePreprocessor.scala.txt` contains a implementation of the configurable database using C preprocessor directives. We also provide some interfaces and tests for each task which can be found in the respective subfolders of `src`. The `utils` package contains the different storage implementations (*Map* and *List*) that have to be used by the database, as well as the `ConfigurationError` exception that should be thrown by the runtime and decorator implementations when a function is called that is not part of the current configuration (e.g., if `write()` is called but the feature *Write* is disabled.). This is necessary because with these implementation techniques it is not possible to change the interface depending on the selected features. The file `build.sbt` contains a build script that we use to build your submission and run the tests.[5] You must not edit this file.

The build script should also enable you to import the project skeleton into any IDE with SCALA support (e.g., INTELLIJ with the SCALA plugin). SCALA can be quite sensitive regarding the used language version and things might break if you use the wrong one. For the assignment, we use *SCALA 3.2.1* which is also specified in the build script. You can also run the tests included with the project skeleton using the build script. To run all tests execute the command `sbt test`.

Feature Model The configurable database we implement follows the following feature model:



The following table explains each feature in more detail:

[5]https://www.scala-sbt.org/1.x/docs/

Table 1: Explanation for all features.

Read	Enable read operations, i.e., given a key retrieve its value.
Write	Enable write operations, i.e., allow storing key-value pairs.
Transaction	Enable transactions. This adds new functions for committing or rolling back a series of write operations.
Logging	Log all function calls.
Map	Use a map for storage.
List	Use a list of tuples for storage.