Assignment: Software Tool Chain

Introduction

Estimated time: 2 days. We suggest splitting this into four 4-hour meetings. Work in teams of 2-3 persons.

You will develop a small Java product, and gradually extend a software development tool chain for it. The tool chain will include

- Version control using *Git*
- Version control server, issue tracking, and code review using *GitHub*
- Automated test using *JUnit*
- Automated build using *Ant*
- Continuous integration using *Travis*
- Optionally additional tools, for example for code coverage.

You may choose another programming language, for example C++ or Python, as long as you can find suitable replacements for the Java based tools (on your own).

If you are already very experienced in this area, take the opportunity to try out some new tools, and to coach your fellow team members.

You will use agile development, working in iterations and in very small development steps. For each of the iterations, work together as a team, and sit co-located. We expect each iteration to take around four hours. At the end of each iteration, you should reflect briefly on your work, and document this on your GitHub repository wiki, for examiners to review.

As soon as you have created your GitHub repository, you should register your team and repo url on a form. You should have received a link to this form in an email, and also a link to a page where you can see all registered teams.

As a result of the assignment, we expect a very tiny product that is well designed, well tested, well documented, easy and possibly fun to use. As the product, you will build a tool/game that can translate English words to *Pig Latin*. See `https://en.wikipedia.org/wiki/Pig_Latin`.

An example project that you may use as inspiration:

- `https://github.com/gorelhedin/minimal-tool-chain`

Some pointers to on-line documentation for the tools:

- **GitHub** Hello World: `https://guides.github.com/activities/hello-world/`
- **Ant** Hello World: `https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html`
- **Ant** tasks for JUnit: `https://ant.apache.org/manual/Tasks/junit.html`
- **JUnit** main site: `http://junit.org`
- **JUnit** getting started: `https://github.com/junit-team/junit4/wiki/Getting-started`
- **JaCoCo**: Java Code Coverage Library `http://www.eclemma.org/jacoco/`
- **codecov.io**: Online code coverage server `https://codecov.io`
- **Travis**: A continuous integration server `https://travis-ci.org`

Iteration 1 (Getting Started)

The goal of the first iteration is to get into the normal state of a working product, albeit with extremely little functionality. Everybody in the team should be able to build and run the product on their own computer.

1. Set up a public git repository for the product on GitHub. (We have chosen GitHub to make it possible to later use *Travis* for continuous integration.) Add a `README` and a `.gitignore` file. Make sure there is an issue tracker associated with the repository.
2. Implement version `v0.1` of your product. It should be the simplest possible version of the product that you can think of and it should be executable. For example, just read one word from the command line and output a constant pig latin word, disregarding the input.
3. Implement a JUnit test for one of the methods in the product.
4. Implement an *Ant* build script from which you can
   - compile the code and build a jar file
   - run all tests
   - clean away generated files
5. The `README` file should explain how to build, run, and test the product. Make a new clone of the project and check that the instructions work. The `README` should also document any platform dependencies and dependencies on installed tools. For any subsequent changes to the product, make sure to update the `README` so that it is consistent.
6. Check that everyone on the team can a) commit, push, and pull from the common GitHub repository, b) build and run the product on their computer, c) understand all the files.
7. Tag the resulting version with `v0.1`.
8. Have a joint brainstorming session to identify around 10 possible features to add to the product, and add these to the issue tracker. Try to make the features as small as possible, yet represent something of use for the end user. You will implement some of these issues in later iterations. For bigger features, you can break them into smaller features later. Some example features:
   - translate a single word to Pig Latin
   - handle many words in a row, not just a single word
   - read from an input file
   - treat punctuation correctly
   - support a reverse translator
   - extend the Pig Latin rules
   - make it into a game
   - ...
9. Discuss briefly, and reflect on what you have done as a team in this iteration. What was your background expertise on version control systems? What tasks were easy? Were there problems that were difficult to sort out? What surprised you? Document your reflections on the wiki.
10. Register your team and the GitHub reposiotory url on the registration form. You should have received an email with a link to this form.

Time left? Start on or prepare for iteration 2.

Iteration 2 (Test-Driven Development)

In the second iteration, you will implement some of the features you have listed on your issue tracker. You should practice working in *small steps* using *test-driven development*, and *integrating often*. Try to commit often, after completing a small step. As soon as you have a small improvement and all tests work, push your changes to the common repository. After this iteration, we expect that

- each team member should have pushed several times
- the product should have a little more functionality than `v0.1`
- there should be good test cases with what you believe is good coverage.
- each version should build and test without failures.

Do the following in this iteration:

1. Work in parallel so that you get some merge conflicts. Communicate with each other so everyone knows what goes on, who implements what features, etc.
2. Use Test-Driven Development.
3. Whenever relevant, add more issues. This could be bugs or refactorings that are related to other things than the issue you are currently working on, or it could be new features that you come to think of.
4. Commit and push often, but only when your sandbox is "green" (all tests pass). For each commit, relate to the relevant issue in the commit comment (e.g., using "see ..." or "fixes ...").
5. Pull often to get the latest changes from the common repository, and merge with your work.
6. If you didn't get any merge conflicts naturally, provoke some, just to get the hang of how to handle them.
7. Tag the resulting version with `v0.2`.
8. Reflect and discuss briefly what you have done in this iteration. What went well? What would you like to improve or do differently? What surprised you? Document your reflections on the project wiki.

Time left? Try out additional features in JUnit. See Usage and Idioms, and Third-Party Extensions at `http://junit.org/junit4/`:

- **Parameterized tests** Experiment with creating parameterized tests. For example, add support for tests with input/output files which can be used as acceptance tests.
- **Test fixtures** Use `@Before` annotations to set up the same data structure ("fixture") before each test in a class is run.
- **Mock objects** Experiment with creating mock objects to test parts of the code in isolation.
- **Matchers** Experiment with `matchers` and `assertThat` to get more readable test cases, and better error messages when tests fail.
- **Random testing** Experiment with random testing using property-based "Theories" and QuickCheck-style random input generators.

Iteration 3 (Code Review)

In iteration 3, you will continue doing test-driven development, but now also focus on *clean code* (well designed, easy to understand). You will extend the tool chain with *code review*, and work with *feature-branches* in the git workflow.

After this iteration, each team member should have been involved in both sides of a code review, and the code should be beautiful and well tested.

1. Discuss the current design. Is the code clean? Is there a need for larger refactorings? Do you need to agree on some design issues or coding conventions?
2. Continue the development, but now all changes should be done on separate *feature branches*, and merged into the master branch only after code review. After pushing your commits on the feature branch, ask for code review by creating a so called *pull request* on GitHub. Experiment with some discussion back and forth between the reviewer and the author.
3. Tag the resulting version with `v0.3`.
4. Reflect and discuss briefly what you have done in this iteration. What went well? What would you like to improve or do differently? What surprised you? Document your reflections on the project wiki.

Time left?

- **Merge vs. Rebase** Experiment with using `git merge` and `git rebase` to understand the difference.
- **Code coverage** Try out the tool JaCoCo to check your code coverage.
- **Metrics** Use JaCoCo to compute the cyclomatic complexity (CC) of your code.

Iteration 4 (Continuous Integration)

In iteration 4, you will continue doing test-driven development with code review. You will extend the tool chain with a *continuous integration* server. In this iteration you will also review another team's product and propose a small change to it.

1. Set up online continuous integration using *Travis*. Travis should run all the tests after each commit to the GitHub repository. If there are any problems, Travis should email the team. You do this by registering your repo on Travis, and creating an short `.travis.yml` script in your repository to instruct Travis what to do on each commit.

2. Continue the development. Experiment with pushing code with errors so you see that the continuous integration works.

3. Review another team's product. Pick the team that is right after your team on the page with the registered teams. (You should have received an email with a link to this page.) Do this by forking their repository and try to run their product. Is it easy to install? Does it work? Is there something in the code or documentation that could be improved? Identify some detail you would like to improve. Make the change to your forked repository and send them a pull request, asking them to integrate your changes. If they like your proposal they can integrate it, and otherwise just turn it down. Write a brief review on your project wiki, and email the team you have reviewed about it.

4. When you think your product is finished, tag the resulting version with `v0.4`, and upload an executable version of it (a jar file) to your GitHub repository.

5. Reflect and discuss briefly what you have done in this iteration. What went well? What would you like to improve or do differently? What surprised you? Document your reflections on the project wiki.

6. Discuss the assignment as a whole. What is your general opinion on the assignment: Was it worthwhile? Too easy or too difficult? Do you have suggestions for improvements? How much of the optional tasks did you do? Document your reflections on the project wiki.

7. Double check that you have completed everything necessary for this assignment, and documented your reflections for each iteration on the wiki. Write a final note on the wiki that you consider your assignment complete.

Time left?

- **Continuous code coverage** Use the online server `https://codecov.io` to integrate JaCoCo code coverage into your tool chain.
- **Container** To help others build and test your product without having to install separate tools, package it into a Docker container and upload it to your GitHub repo. See `https://www.docker.com` for more information.