TDA 594/DIT 593 - Assignment 4 - Modularization, Dynamic Variability, and Testing

Due Date: Sunday, December 18, 11:59 PM
Submission: Via Gitlab and Canvas

Overview

Previously, your company successfully created a small product line by integrating three bots into a preprocessor-based architecture. You have played a bit with different configurations and given the generated bots to test customers who like the idea. Your company realizes that this way, it can shorten time-to-market for new robot firmware builds, deliver more value to individual stakeholders, and offer increased customization. In summary, your company is now more confident that establishing a configurable platform is the way to go.

Unfortunately, it is quite clear that the preprocessor annotations will clutter the source code and lead to maintenance nightmares when more bots are integrated into a platform. You have also decided that load-time configuration will enable greater flexibility (perhaps, in the future, you will also support dynamic runtime binding).

Preprocessors were a great start. However, now, a proper architecture is needed, facilitating more modularity of features. Your company has decided to create a framework and refactor the existing features into modules adhering to the framework. You are shifting from a compile-time, tool-based, annotation-based implementation (preprocessors) into a load-time, language-based, composition based approach based on a framework and design patterns.

In consequence, your company will establish a flexible framework for modularizing features (to the extent possible) and enabling/disabling them via configuration and parameters. After that, once it has a working implementation and has solved various technical issues, there is the need for proper quality assurance through testing, to sustain the platform in the long term, since many more features are planned. Two types of testing are essential to ensuring that features work well and are well-modularized: *Unit tests* are used to verify the correctness of individual features. *System (integration) tests* are used to examine combinations of features. You have decided to explore unit tests as a first way to establish your testing pipeline.

Following this assignment, you will be able to abstract lower-level variation into an object-oriented framework, judge the applicability of design patterns and apply them when beneficial, and be able to modularize concerns (features) to the extent possible. Furthermore, you will be able to create test cases (using the JUnit framework) that test individual features (unit tests).

Your Tasks

Your task is to establish an object-oriented framework for bots and realize your platform from the previous assignment with it. Specifically, you will start with the same feature model you refined in Assignment 3 and the three bots that we provided code for.

1. Switch to dynamic variability using parameters. This requires switching the project's FeatureIDE composer to RuntimeParameters instead of Antenna. FeatureIDE will then create a .properties file that you can use to configure a bot on launch (see hints below!).
2. Think about abstractions in terms of classes according to what you learned in this and other courses. Create a framework based on these bots and your feature model in a dedicated package (e.g., groupX.framework).
3. Integrate the codebases by refactoring the bots into your framework. Modularize your existing features as much as possible using design patterns. Your integrated codebase should implement at least one design pattern. This can be one of the design patterns discussed in class, or any other that supports variability in some form. As a rule of thumb, we expect to see at least one design pattern applied, but it is possible to incorporate more.
    a. Only apply design patterns if they make sense and their use can be justified[1]. Do not keep adding design patterns if they would make the design less effective!
    b. Consider, for example, the use of the strategy pattern to compose a bot with particular feature options, or the factory pattern to instantiate a requested bot based on selected options.
4. Implement one more (non-trivial) feature of your choice, either from your domain analysis (when you know the implementation exists), from the RoboWiki, or from another RoboCode source (i.e., a GitHub project).
    a. We recommend you find existing code and refactor it into your framework, rather than starting from nothing and coding yourself. Give credit to the original source in your code comments and submission
5. Create a set of unit tests (using either JUnit 4 or 5) for one of the individual features implemented in Assignments 3 and 4.
    a. This should not be a trivial feature. The feature should have complex conditional behavior in its code (multiple if-statements or switch statements, especially nested if-statements), indicating multiple possible outcomes when it is executed. If unsure of what to focus on, discuss your choices with your supervisor.
    b. For this feature, create a set of unit tests that cover each of the major outcomes of that feature (i.e., if there is conditional behavior, try to cover each condition that affects the outcome). If there is exception handling, try to ensure that exception-triggering input is handled. We do not expect exhaustive testing, but try to write enough test cases to ensure thorough testing. There is not a correct "number" of tests, but try to aim for 5 or more.

[1] The chosen pattern(s) should be applied for an intuitive reason, e.g., they efficiently solve a problem in the design, support customization, or support maintenance and evolution of the code. As an example, within RoboCode, there are many `onScanned` events that occur when one robot "sees" another robot. These events are sent by the simulator to all appropriate "subscribers". This is an example of the Observer pattern, which efficiently enables a publisher/subscriber relationship.

Hints

For your implementation. We recommend that you use the same combination of tools and versions as in Assignments 2 and 3: FeatureIDE 3.9, Java 17, and Eclipse 4.20. If you encounter issues, report them to your supervisor.

While you could start with your Assignment 3 code and try to change the FeatureIDE composer via the project properties, you may find it easier to create a new project and copy the feature model and the code over incrementally when creating the framework.

Chapter 17 of *Mastering Software Variability with FeatureIDE* offers some information on getting started with Runtime Parameters[2]. The full book is available digitally from the Chalmers library.

We recommend that you enable continuous integration to avoid making commits that break the build. Continuous integration is a process where developers are encouraged to integrate their changes into a shared code repository on a frequent basis. Often, a set of scripted actions take place automatically each time you commit changes to the code repository[3]. This can be used to compile the project each time you commit to ensure that the project still builds successfully. Once set up, it will execute all JUnit test cases in your project automatically. The following tutorial explains how to enable continuous integration[4].

RoboCode's security manager may issue an error when trying to read the properties file in the simulator. For reading the properties file, do not use FeatureIDE's generated PropertyManager class, since that will try to read the file from the filesystem, which Robocode's security manager will not allow. The solution is to put the properties file in the bin folder either manually, or using a build script. Instead:

1. Use the provided class ConfigurationManager[5] to manage the runtime parameters.
2. Copy the properties file into the bin/ folder as well as the folder where ConfigurationManager is stored. You can copy the file manually or use[6] a build file[7].
3. In RoboCode, navigate to options -> preferences -> development options and add your project. Alternatively, follow the instructions at https://robowiki.net/wiki/Robocode/Eclipse/Running_from_Eclipse

For testing. If you have not used JUnit before, there are several excellent tutorials. Some options:

- https://www.vogella.com/tutorials/JUnit/article.html
- https://www.tutorialspoint.com/junit/index.htm
- https://www.guru99.com/junit-tutorial.html

---

[2] https://chalmers.instructure.com/courses/16077/files/folder/Resources?preview=1813372
[3] For more information, see https://en.wikipedia.org/wiki/Continuous_integration
[4] https://chalmers.instructure.com/courses/16077/pages/continuous-integration-with-gitlab-ci
[5] https://chalmers.instructure.com/courses/16077/files/folder/Assignments?preview=1813373
[6] https://rohitprabhakar.com/2010/02/03/how-to-run-ant-build-from-eclipse
[7] https://chalmers.instructure.com/courses/16077/files/folder/Assignments?preview=1813377

A quick JUnit reference can be found at:
https://docs.google.com/document/d/1_rrUlDcA7E9UcVY7mZLa64ZlWtDDEfgpsfs6N51QBto/edit?usp=sharing

To test a feature, you must write JUnit tests that call associated methods and make assertions on their return values. Then, you can create Mockup robots by giving them dummy values, and even instantiate their movement, targeting, etc. with dummy objects. The goal is to run the methods and see if the values they calculate/return are the same values as expected. For that, you don't have to actually "run" the robots in RoboCode. As an example, here is a file containing sample test cases. These test cases make use of a "mock" (fake) robot, which can be found here. Do not simply use these test cases as your own - you may use them as inspiration, but you must do more than just change the values.

Deliverable

Submit, via Canvas, the following (one submission per team):

- A document in PDF format, containing:
    - A link to a “release” of your GitLab repository containing the source code of this assignment[8].
    - A description of your framework, including a class diagram outlining the architecture. Your diagram can be created in any tool of your choice, and can even be generated from your code using, for example, Eclipse plug-ins.
    - A description of your integration strategy and design decisions, including explanation of how you integrated and refactored the code, which design patterns were used, why you applied those specific design patterns, and the challenges encountered during this process.
    - A description of the two feature implementations.
    - Your updated feature model, if any further changes were made.
    - The property files for three example variant configurations (at least one should be different from the previous assignment).
    - Screenshots of your variants operating in the simulator (screenshots can be individual or show combinations of the bots).
    - A list of the test cases designed and brief explanations of each:
        - State which feature is being tested.
        - List the file where this test is located in the source code.
        - Explain the purpose of the test (e.g., what functional outcomes are covered, what types of errors it could identify).
        - Document why the specific input was chosen that you applied in the test case.
        - Document why the assertions/fail statements used in the test are sufficient to show that the feature’s behavior is correct or incorrect.

[8] For information on creating a release, see https://docs.gitlab.com/ee/user/project/releases/

Grading Guidelines

Note, these guidelines are intended to give some guidance, but are not exhaustive. Each supervisor will assign a grade based on the correctness and quality of your work.

Grade	Guidelines
5	● Design is clearly described, features are cleanly separated, and a class diagram is present. ● Consistent and well-explained integration strategy that thoroughly describes the process, including challenges encountered. ● Consistent and correct use of design patterns, with clear and detailed rationale for their use. ● LinearTargeting and another new feature are implemented and their implementation is described in clear detail (describe the functionality of the techniques, including the different outcomes of that functionality). ● Feature model is consistent with the implementation. ● New variants are not equivalent to original bots, and are consistent with the feature model. ● Refactored bots and new variants still function as expected in the simulator. ● The chosen feature is thoroughly tested, covering the full range of outcomes and exception handling. ● All tests are well explained, covering purpose, input choices, and assertion choices in detail and with included rationale. Why these tests are sufficient to show correctness of a feature is clearly explained. ● Test code is commented and can be understood without having written the code.
4	● Design is described and a class diagram is present, features are mostly separated (some overlap may be present). ● Consistent integration strategy. Detailed explanation of the integration process. ● Correct use of design patterns, with detailed rationale for their use. ● LinearTargeting and another new feature are implemented and their implementation is described. ● Feature model is consistent with the implementation. ● New variants are not equivalent to original bots, and are consistent with the feature model. ● Refactored bots and new variants still function as expected in the simulator. ● The chosen feature is well-tested, covering a wide range of outcomes and exception handling. ● All tests are explained, covering purpose, input choices, and assertion choices with included rationale. Some attempt is made to

	show why these tests are sufficient to show correctness of a feature. ● Test code is commented.
3	● Design is described, features are mostly separated (some overlap may be present). ● Some explanation is made of the integration strategy. ● Correct use of design patterns, with some rationale for their use. ● LinearTargeting and another new feature are implemented and their implementation is described. ● Feature model is consistent with the implementation. ● New variants are not equivalent to original bots, and are consistent with the feature model. ● Refactored bots and new variants still function as expected in the simulator. ● The chosen feature is tested, covering a range of outcomes and exception handling. ● All tests are explained, covering purpose, input choices, and assertion choices with included rationale.
U	● Design is not described. ● Significant overlap between features. ● Integration strategy not properly explained. ● Design patterns used incorrectly, or not used at all. Lack of justification for their use. ● Implementation inconsistent with feature model. ● New bots do not exist, are equivalent to original bots, are not integrated successfully. ● Refactored or new bots do not function in the simulator. ● Testing effort is inadequate. ● Tests are not explained or poorly explained. No or little rationale for choices made.