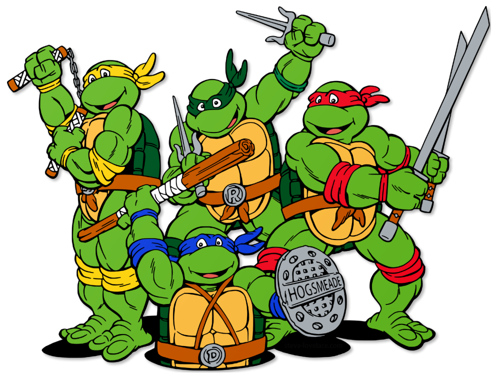

F23 - Project - **Super-Mutant Challenge:** Applying Systematic Testing Strategies to an Open-Source Project

What you will do

- Apply systematic spec-based, structural, and unit testing strategies to real software.
- Share your insights with other teams.
- Compare your experiences with other teams' experiences.

Schedule (F23) - ***Strict***

Milestone	**Due Date (Due Time: 11:59pm PT)**
Selection: Form a team (3-5 members) and choose an open-source project	**10/02**
Approval: Teaching team reviews and approves project selection	**Quick approval: 10/03** **Changes: 10/3-10/5** **Final approval: 10/5**
Handover: Send selected project to rival team	**10/06**
Mutation: Rival team mutates victim team's target code and returns it to victim team	**10/10** **Late and non-compliance penalties apply**
Testing complete	**11/05 (29 days from last milestone, including the break)**
Revelation: Rival team reveals mutations applied to victim team's target code	**11/06**
Sharing and Canvas Deliverables: Teams post team findings on shared board and upload all deliverables to Canvas	**11/08** **Late penalties apply**
Presentation and Discussion	**11/09** **Team Schedule on Canvas Item** **Teams will be scheduled randomly - *may extend class time and ask teams to arrive at different times***

Step 1 - Each team ideally has 4 members. 3 or 5 are also acceptable. Select your teammates, form a team, and assign yourself to a project team on Canvas (If you email us your team we'll create the team).

You must form your team before the work on the project starts. Email the name of your team members to your TA Divyam once your team is complete. If you cannot find a team, contact your Divyam, and we will assign you to one.

Step 2 - Choose open-source project and class/module to test

This is the most important step since if you don't choose an appropriate project, your task may turn out to be more challenging than it needs to be. Identify an **active** open-source project with suitable classes to test. Project must:

- be implemented in **Java;**
- have existing unit tests that you can run locally (we'll call these **native tests**);
- have a reasonably well-documented API;
- present some challenges for testing (e.g. not a math library with only numerical inputs);
- have complex enough behavior worthy of comprehensive testing.

Select a small set of classes/modules to test. The classes/modules should:

- come with unit tests;
- be testable with the strategies presented in class;
- implement important or core capabilities.

Make sure you can run the unit tests in your local environment.
Determine your **target code**. This is how you can scope your work:

- Identify a subset of methods/functions to test within the selected classes or modules. Methods/functions should be:
  - public;
  - implement a coherent set of core capabilities
  - as diverse as possible in terms of their signatures, with a variety of parameters of different types (numeric, enumerated, string, collection, objects of varying complexity).
- To start with, each team member should roughly have **100 Lines of Code** (not counting comments or whitespace) to test. Better to have a reduced scope and do a thorough job than have a wide scope and do a half-baked job.

When choosing the **target code**:

- Exclude simple methods/functions that need not be tested, such as setters or getters.
- Include some methods/functions that track state or work together (collectively fulfill a capability by requiring a sequence of calls).

You should expect to create about 50 tests per team member -- this is your budget. Scope the work, optimizing as necessary, to reach and stay within this goal. If necessary, expand your **target code** to reach your budget.
Enter your selections in the Project Signup Sheet. **First-come first-served:** if your selected project is taken by another team on the signup sheet, you will need to find another one!

Step 3 - Hand selected project over to rival team

Hand your project over to your assigned **rival team** for mutation. Tell them about your team's **target code**, which classes/modules and methods/functions you'll be testing. The **rival team** will hand you back a mutated project, but won't tell you what they have changed. *You can find out by a bit of code forensics... but we trust that you won't, for the sake of good sportsmanship and to get the most out of the project.* ***Number of mutations/faults found is not a factor in your project grade. What counts are your team's (1) process to create a strategy and a well-specified, comprehensive plan that follows that strategy, (2) diligence to collaborate and follow the plan (and tweak it as necessary), and (3) ability to communicate the plan, evaluate its effectiveness, and interpret the results.***

Step 4 - Secretly mutate the victim team's selected project to seed defects

This step is difficult. Do not underestimate it. Get the project selected by your **victim team**. The **victim team** will tell you which classes and methods they will be testing (their **target code**). Mutate **these classes/modules only** using mutation operators listed on pitest.org that inject faults. Document your mutations in a **Mutation Sheet**. A template with examples is provided here (copy this template, but don't share it with the **victim team** yet).

- Select a variety of different mutations to apply. If you don't succeed with mutation operators from pitest.org, as a last resort you can devise your own mutation to inject faults. Try to avoid reusing mutation operators. Introduce **10 mutations**, reusing as few operators as possible**.** You'll see that this is hard with the constraints given below.
- Apply the mutations to different parts of the **target code** to create a **super-mutant**.
- Make sure you don't change APIs (interfaces, method/function signatures) when applying mutations. This is really important.
- Make sure that the classes/modules compile. This is really important.
- Make sure that when a class/module is mutated using a mutation operator, the resulting mutant is not *equivalent* to the original class/module (see pitest.org and lecture slides for explanation -- an equivalent mutation does not change the external behavior of the system). This will ensure that each mutation injects a *real* defect into the **target code**.
- Do not ever modify the provided **native tests**. You may remove them if necessary (see below).
- Make sure that the **native tests** for the classes/modules under test can be run after mutation. **Native tests** may or may not fail with the mutations. Try to find mutations that prevent **native tests** from failing, while still introducing real bugs. Make sure that your mutations are not so drastic that they end up breaking all or a massive amount of **native tests**, or produce a cascade of runtime errors**.** If you cannot find mutations that don't break any **native tests**, as a last resort, you may delete failing **native tests** to prevent revealing the mutations. You should delete as few **native tests** as possible, if any.
- Your **super-mutant** should not be too broken or too obvious. The mutations should be as localized, subtle, and targeted as possible without being equivalent.
- Do not mutate functions that manipulate or generate externally undetectable internal representations, such as methods that have to do with hash functions, hash values, or

unique identities. These kinds of mutations are likely to be equivalent to the original code.

- Do not create mutants that only affect system performance or manipulate the ordering of processing without any visible outside effects. These kinds of mutations are likely to be equivalent to the original code.
- Record the mutations (and if applicable, any **native tests** deleted) in a copy of the **Mutation Sheet**, but keep them secret for now.
- Hand over the resulting **super-mutant** to the **victim team**.
- **A 25% penalty will apply** to the **rival team**'s grade for not following mutation rules, changing **native tests**, mutating code outside the **target code**, handing in mutated code that doesn't compile, changing method signatures in the **target code**, handing in mutated code that fails undeleted **native tests** or breaks the system in a catastrophic way, deleting a large number of **native tests**, or handing in mutated code that causes runtime errors.
- **An additional 25% penalty will apply** to the rival **team**'s grade for being late.
- **Issues with mutation:** sometimes it may be difficult to mutate a piece of code without causing runtime errors or a massive meltdown with **native tests** even after removing some **native tests**. In these cases, as a last resort, you can use two strategies (please contact your TA or instructor to check first):
  - If you are still getting runtime errors after applying mutations and removing some **native tests**, try masking the mutation with a *try-catch* clause and doing nothing in the *catch* clause. This may be acceptable as a last resort.
  - If you are still stuck, try reusing more mutation operators (thus sacrificing diversity of mutations), or try to mutate a portion of the target code that you haven't tried to mutate yet.

Step 5 - Create a test plan

Pretend that the **native unit** tests don't exist for your **target code**. You will test it from scratch. Write down which capabilities (functionality you will be focusing on) you will be testing in the **target code**: what does the **target code** do? Evaluate the complexity of the methods/functions you will be testing -- are they too difficult to test, or too easy? Determine underlying characteristics and attributes in the **target code**. Create an input space model, or a set of models for different parts. Identify the strategies you need from the course for each distinct capability or method/function. Define test case specs for each input case model by using a proper combinatorial, black-box strategy, and if appropriate (e.g., for methods with complex control logic) by using a structural, white-box strategy. Remember the *behavior-first* principle: first black-box, then white-box. The strategies should encompass both of these techniques:

- black-box, or spec-based techniques that employ a variety of (at least two) combinatorial strategies(test the majority of the **target code** with these strategies); and
- structural, or white-box, techniques that aim for specific code coverage criteria (test selected parts of the **target code** with these strategies, in particular if there is complex logic and/or black-box strategies were found to be inadequate in terms of coverage

results (these strategies include various coverage-based techniques, branch testing, condition testing, MC/DC, and cyclomatic or basis-path testing).

In applying your strategy you should adhere to unit testing and general testing principles, including the use of test doubles when necessary to isolate the capabilities you are testing.

As stated above, you may apply different techniques to different parts of the **target code**. Describe the design process for your **test plan**. Be systematic in your approach. Optimize your plan to stay within the given scope goal. Optimization may involve changing your input space models to be coarser or finer. Document your **test plan** using tables, *like you did in the Test Design Exercises Lab*.

Check: Your **test plan** should be precise, straightforward to allow generation of individual test cases without any extra information. Imagine that a different person or an automated tool will be generating the test cases: the tool or person should have all the information required to generate the test cases. Did you use the right terminology (attribute, characteristic, semantic vs. syntactic characteristic, base choice, all-choice, all-pairs, each choice, capability, test case spec vs. actual test cases, oracle, basic condition, MC/DC, compound condition, white-box/structural vs. black-box/spec-based, etc.)?

A **test plan template** for spec-based testing is provided here. There is no template for white-box or coverage-based testing: explain your white-box strategy clearly in any format you wish.

Step 6 - Apply test plan to the super-mutant

Design test cases using the **test plan** and apply them to the **super-mutant** created by the **rival team**. Follow the process used in the *Test Design Exercises Lab* (Lab 2).

- Use test doubles to isolate the classes/modules under test from their collaborators if necessary.
- Organize groups of related test cases into separate test classes/modules and test suites.
- Translate test cases to actual executable tests: write the tests using the testing framework of the OS project. These are your **team tests**.
- Execute your **team tests** on the **super-mutant**.

Tip: Do not divide work strictly along capabilities/methods. Testing the capabilities are likely to share attributes, characteristics, input domains or whole input space models unless you pick entirely unrelated or disjoint capabilities. You need to develop these shared models *as a team* and reuse them in testing the selected capabilities individually. If your input space models are redundant or uncoordinated, deductions will apply on your project grade.

Step 7 - Ask the rival team to reveal the mutations applied

The **rival team** shares the **Mutation Sheet** with the **victim team** to reveal the list of mutations applied to the **victim team**. Execute your **team tests** on the original, unmutated code. Observe the differences.

Step 8 - Evaluate the test plan

Fault analysis

Record any faults found in the **super-mutant** by your **team tests**.

- Indicate for each fault whether the fault is a **seeded fault** (a **mutant**) or whether it is a **native fault** (existed in the original unmutated class). If the original unmutated code breaks a **team test**, mark the associated root cause as a **native fault**.
- For each fault found, describe the type and consequence of the fault using the taxonomy provided in the **Shared Board**.

Coverage analysis

You will calculate structural coverage metrics for the **target code**.

- Calculate statement and branch coverage of the **super-mutant** with the **team tests**. Compute coverage only for the **target code**.
- Calculate statement and branch coverage of the **super-mutant** with the **native tests** supplied with the selected project. Compute coverage only for the **target code**.
- Compare and interpret the results.

To be able to compute coverage, you'll need a coverage tool. You may use **Cobertura** as it works in many environments and with many build tools.

Step 9 - Post your team's findings on Shared Board

Fill out the **Shared Board** to summarize and record your results. There are multiple tabs (*Baseline Info* and *Summary of Results*) you should fill out. Don't miss any of the tabs.

Step 10 - Prepare a presentation

Your presentation must have all of these elements:

- **Context and Rationale**. Describe the selected project and methods/functions targeted and justify their selection with respect to project requirements. Give usage examples of capabilities tested. *Indicate the URL for your project repo -- your repo should include your* **team tests***.*
- **Plan**. Describe and justify your **test plan**: what the testing focused on, which specific testing strategies/techniques were used, what their goals were, why they were chosen. Summarize the input space models used, with attributes and characteristics identified, partitions and blocks formed. Explain the structural testing strategies and coverage criteria used and why they were selected. Use tables and charts.

- **Execution**. Explain how the **test plan** was implemented. Focus on practical considerations, for example, how you have used test doubles or adhered to testing principles. **List all tools used in applying the strategies.**
- **Results.** Summarize your findings and insights by addressing the following questions in your presentation:
    - How effective was your **test plan** in terms of finding **native faults**?
    - How effective was your **test plan** in terms of detecting **seeded faults**?
    - How effective was your **test plan** in terms of the number of **team tests** created compared to the number of **native tests**?
    - How effective was your **test plan** in terms of code coverage compared to **native tests**?
- **Lessons Learned.** What were the challenges and benefits compared to ad-hoc unit testing? What would you do differently if you had to start over? This section is important. Lessons learned should be **specific**, **meaningful, specific, and deep**.. It should not just repeat platitudes, known facts, or course advice, but rather show how that advice was confirmed/refuted in the project.

Step 11 - Present to class

We will record the presentations so that you can

- Limit your presentation to **12 mins /* corrected */**. I will cut you off when your time is up.
- Come to the class having rehearsed your presentation as a team.
- Know who should speak when and for how long.
- **Every team member should speak** and contribute equally to the presentation.
- Cover all the elements.
- Coordinate your presentation -- it should not look like a bunch of mini presentations stitched together.
- Use one laptop.

Step 12 - Submit your presentation and test plan to Canvas

Submit your **presentation slides** and **test plan** each as a pdf file and submit to Canvas under Project. Upload two files with these file naming conventions: **Project-TestPlan<Team-ID>.pdf** and **Project-Presentation-<Team-ID>.pdf.** Please do not upload a zip file.

Rival and victim team pairs

T**X** mutates team T(**X+1**)'s code: T**X** is the **rival team** and T(**X+1**) is the **victim team**. If there are **N** teams, the last team T**N** mutates T**1**'s code. Team assignments and emails of team members will be available on Canvas.

Tools you will use

You should use the same testing framework used in the original OS project as your test driver. This will likely be JUnit. **Jacoco/EclEmma** is recommended for performing coverage analysis.

In addition to these tools, you may use any testing or test case generation tools you wish. Report any tools used in the **Shared Board** and discuss their usage in your presentation. For all-pairs combinatorial testing, a list of tools can be found here.

Github Team Repos for Project

Make sure to have a **real photo** attached to your github profile used for the course. You may create private repos in this organization for your project by forking and cloning the OS project chosen. These rules apply:

- Public repos are forbidden.
- Name any repo you create for your team using this convention: **[F23]-[TeamId]-[ProjectName]**. Example: **F23-T1-someOSProject**.

Bonus points

You have a chance to contribute your tests and any fixes for **native faults** found back to the OS project you tested. Your contribution must be **meaningful and demonstrably of value** to the project. They should not consist of trivial or frivolous changes. To do this, you may fork the OS project (see the Atlassian Forking Workflow). Later you can request from the maintainer that your tests and changes be integrated back to the original project. If your changes are useful, accepted, and you succeed in contributing to the original OS project in a meaningful way, your team will receive one bonus point that can be used to reach the maximum Project 1 score. Your total score will not exceed the maximum score allocated to Project 1.

No pull requests with trivial or bogus changes. No pull requests performed after the fact: the pull request must be issued **before** project presentation. Bonus will be awarded after the pull request has been accepted by the OS project core developers and after the process has been reviewed/verified by the teaching team.