TSDetect

ANTI PATTERN PATTERN CLUB

Joseph Saltalamacchia Ben Radley

Cameron Riu Spencer Stissi

Project Sponsors

Anthony Purrma
Mohamed Mkaouer
Christian Newman

Faculty Coach

Zhe Yu

Project Overview

Previously developed by Anthony Peruma[1] and his team; TSDetect is an open source tool used for detecting various code smells found in Junit tests, specifically that of Java projects utilizing Junit4 or Junit5. Test smells are defined as bad programming practices in unit test code (such as how test cases are organized, implemented and interact with each other) that indicate potential design problems in the test source code[2].

Our team was tasked with improving this project with the following features: user data analysis and storage of relevant user data, results checking, and possible usage against python projects. This feature set changed slightly as the team researched and developed solutions, and discovered the limitations of both the tools and their schedule.

The goal of this project was to deliver well designed solutions to each of the feature sets provided to our team. In addition, we aimed to traverse all steps of a software engineering project from the ideation phase to the delivery stage.

Basic Requirements

The Basic requirements of the system are broken up into two categories:

1. Existing functionality that must be maintained.
2. New or expanded functionality added on to the existing project.

As it would be impossible to accurately describe the new functionality without also describing the original functionality, both will be detailed here

Original Functionality

1. TSDetect shall be accessible to IntelliJ users in the form of a plugin, and will integrate into their normal workflow.
2. TSDetect shall be runnable from the Intellij Plugin pane.
    a. When run, TSDetect will scan the test files of the project, and analyze then to detect a subset of test smells
        i. A complete list of test smells accounted for include:
            1. Assertion Roulette[2]
            2. Conditional Test Logic[2]
            3. Constructor Initialization[2]
            4. Default Test[2]

5. Duplicate Assert[2]
6. Eager Test[2]
7. Empty Test[2]
8. Exception Handling[2]
9. General Fixture[2]
10. Ignored Test[2]
11. Lazy Test[2]
12. Magic Number Test[2]
13. Mystery Guest[2]
14. Redundant Print[2]
15. Redundant Assertion[2]
16. Resource Optimism[2]
17. Sensitive Equality[2]
18. Sleepy Test[2]
19. Unknown Test[2]
20. Verbose Test
    a. A Test is considered Verbose if it contains too many statements.
    b. There is currently no information provided on the testsmells.org website in regards to verbose Tests.

ii. Once analyzed, the results shall be displayed for the user in the plugin pane via two graphical tools:
    1. A table containing the test smells and the quantity of each found in the test suite.
    2. A pie chart showing the distribution of test smells found in the test suite.

iii. The plug in shall provide the user with the test file name and class name of any test that is determined to contain a test smell

iv. Additionally, a CSV file containing the results of the scan can be created and saved to the user's machine.

3. Users will have the ability to toggle specific test smells to be included/excluded from future runs of TSDetect via the plugin settings.

New and Expanded Functionality

New functionality in the system is representative of the majority of effort completed in this project, consisting of a command line implementation (Known as "Headless Mode"), a centralized database for storing test smell statistics of consenting users, a dashboard for accessing that information, and the communication links necessary to connect these different components together.

1. Users shall be able to use the plugin in "Headless Mode" (from the command line).

   a. The User shall be able to run the plugin against multiple test suites in rapid succession in headless mode.
   b. The user will run a command line script with the path(s) of the project(s) as command line arguments
      i. The user shall receive the plugin results as a CSV file stored in the root directory of the project(s)
      ii. The command line script will set a new Gradle project argument triggering Gradle to run the application in a headless environment.
   c. The IntelliJ Platform Plugin SDK Documentation[4] shall be followed for development of headless mode
2. Users of the TSDetect plugin shall be asked for consent to collect non-identifying information for research purposes
   a. Polling for user consent shall take place via a popup that is displayed whenever the plugin is opened and a consent choice has not already been made.
   b. If a user does not consent to send their usage data, they will still have full access to TSDetect features as described, but no data will be sent to the TSDetect Centralized Database.
   c. The user shall have the ability to revoke consent by toggling a consent setting in the Plugin settings menu.
      i. If consent is revoked, all previously sent data will still exist in the database
      ii. Once consent is revoked the user will still have full access to TSDetect features as described, but no data will be sent to the TSDetect Centralized Database.
   d. Users who do consent to send their usage data shall see no noticeable difference in their normal use of TSDetect.
3. A user who has consented to send their usage data to the TSDetect Database shall have their plugin attempt to contact the TSDetect Server via the TSDetect Plugin API whenever they run the plugin against their test suite.
   a. Usage data sent to the TSDetect Database shall include a “UUID,” (a (unique user ID”), a Timestamp for the test run, and a list of test smells detected, along with the quantities of those test smells that appeared in the run of the plugin.
      i. A UUID shall be randomly assigned to a user when they initially consent to send their data.
      ii. The Usage data sent shall contain no information that can be used to identify the user without their consent.
   b. Usage data shall be sent to the TSDetect Database as a JSON-wrapped HashMap of the following form:
      i. "Test Smell Name" : "X",
      "uuid" : "abcd-efgh-ijkl-mnop-qrst-uvwx-yzabcd",
      "timestamp" : "yyyy-mm-dd hh:MM:SS.SSS"
      (Where X is an integer representing the number of the given smell detected, and "abcd-efgh-ijkl-mnop-qrst-uvwx-yzabcd" is any 36 character string)

c. The plugin API shall have a single endpoint, reached with the following call:
    i. POST /test-results
        1. This endpoint's Header information shall consist of the following:
            a. content-type: application/json
                i. This content includes the data mentioned in requirement 3.b.i
    ii. This POST message shall return a 200 code on when data is successfully inserted into the database.
        1. In the event of a non-200 response, the TSDetect Plugin shall attempt to establish a connection up to five times.
            a. The attempt to send usage data shall be considered a failure when the TSDetect plugin receives 5 consecutive non-200 responses from the TSDetect plugin API.
d. The usage data shall be stored on the user's machine in the form of a JSON file when the TSDetect plugin fails to send that usage data to the TSDetect Database.
    i. If that JSON file exists on the user's machine when a connection is attempted to the TSDetect server, the TSDetect plugin shall attempt to send the usage data in that JSON file to the TSDetect Database
        1. The usage data JSON file shall be deleted when the data it contains is successfully sent to the TSDetect server.
        2. Data older than one month shall be removed if an attempt to send it fails.
        3. The JSON file shall be limited to 10,000 lines of data to preserve system storage.

Updates to the existing code consisted predominantly of adding the functionality to send the test run results to a database for collection, along with the necessary changes to allow headless mode. That database, the server that owns the API that allows users to make queries on it, and the dashboard for viewing the data are all completely new additions to the project. The requirements for these additions are as follows:

1. There shall be a centralized database to store test run data from various consenting users.
    a. This Database shall be a MySQL database.
    b. The Database shall be consistent of three tables, as follows:
        i. "test_smells", containing:
            1. "test_smell_id", an auto-incrementing integer, and the primary key of the table
            2. "name"", the string name of the test smell associated with the test_smell_id
        ii. "test_runs", containing:
            1. "run_id", an auto-incrementing integer, and the primary key of the table

2. “uid”, a 36 character string representation of the user id associated with that run
3. “Timestamp”, a Timestamp of the time the run was performed, with the following format: “yyyy-MM-dd hh:mm:ss.SSS”

iii. “test_run_smells”, containing:

1. “run_id”, as described above
2. “test_smell_id”, as described above
3. “quantity”, an integer representing the number of times the test smell associated with the test_smell_id was found in the run associated with the given run_id.

c. The Database shall be hosted in a Docker container
d. The Database shall be accessible to an administrator by using a PHPMyAdmin account provided by the Docker configuration.

2. Access to the Database shall be controlled by two APIs hosted on a Spring Boot server (known as the TSDetect Server).
   a. Server Administrators shall be able to check the health of the server via a Spring Boot Actuator.
      i. Single endpoint shall be accessible and reports the current status of the server:
         1. Path: ‘/health’
         2. Expected response: { “status”: “UP” }
   b. The TSDetect Server shall be runnable directly in Intellij.
      i. The server shall build successfully through use of the Gradle task ‘:assemble’
      ii. The server shall run when using the Gradle task ‘:bootRun’
   c. The TSDetect Server shall be runnable through the command line using Gradle.
      i. The server shall build successfully through use of the following command:
         1. ./gradlew assemble
      ii. The server shall runwhen using the following command:
         1. ./gradlew bootRun
   d. The TSDetect Server shall be containerized (runnable in a Docker container).
      i. The server shall build successfully through use of the Gradle task ‘:assemble’
      ii. The server image shall be builty through use of the Gradle task ‘:buildBootImage’
      iii. The server shall run without crashing when using the following docker command:
         1. docker run --net tsdetect-network -p 8080:8080 -t tsdetect-server:1.0.0
   e. The TSDetect Server shall own two API endpoints:
      i. test run data shall be inserted into the test_run table using the following endpoint (known as the Plugin API):
         1. POST /test-results
            a. Headers:

i. content-type: application/json

b. Body:

i. Json-wrapped HashMap of the following form:

1. "Test Smell Name" : "X",

a. (Where X is an integer representing the number of the given smell detected)

2. "uuid" : 36 character string,

3. "timestamp" : "yyyy-MM-dd hh:mm:ss.SSS"

2. The Plugin API shall return a 200 response if at least one entry was added to the test_run table.

3. The Plugin API shall return a 400 response if no entries were added to the test_runs table.

ii. test run data can be retrieved from the test_run table using the following endpoint (known as the Dashboard API):

1. GET /test-smells/query-params/?datetime=[integer]&smell_type=[List]

a. Parameters:

i. DateTime - the number of days before the current date that data is being retrieved from

1. Eg. If a “3” is received, it will look for entries with a timestamp within the last three days.

2. If not given, will retrieve timestamps over all time.

ii. Smell Types - List of smells to include in query

1. If not given, will retrieve all smells

b. Response: A JSON string containing the smell quantities for each smell type currently in the database that align with the given parameters.

2. All query parameters in the Get request shall be optional

a. The Dashboard API shall return a list of test run data for all given tests smells going back a number of days specified by “datetime” when both a “datetime” and “smell_type” parameter are given.

b. The Dashboard API shall return a list of test run data for all given tests smells in the entire database when only the “smell_type” parameter is given.

c. The Dashboard API shall return a list of all test run data for all smells going back a number of days specified by “datetime” when only a “datetime” parameter is given.

d. The Dashboard API shall return a list of all test run data for all smells available in the database when no parameters are given.

3. The Dashboard API shall return a list of test smell names associated with the number of times that test smell was found when any data matching those parameters are found.
4. The Dashboard API shall return a 400 response if no test smell data is found that matches the given parameters.

3. There shall be a Dashboard where test smell data collected by the centralized database can be easily viewed.
    a. The total number of smell counts in the database shall be visible along with “new” smells based on a date filter.
    b. The current number of smell counts and their associated smell names should be visible via a graph.
        i. The graph will be a bar graph where the x-axis is smell names and the y-axis is smell count.
        ii. The graph shall have options to edit the font size of the axes
        iii. The graph shall have options to edit the color of the graph, or add new colors entirely..
            1. Options shall be resettable to the default value.
            2. Options shall be stored in browser local storage for persistence on reload.
    c. The current number of smell counts and their associated smell names shall be visible via a table
        i. This table shall have smell names in the left column and the associated count in the right column.
    d. The presented smell data from the database can be filterable by day
        i. Users shall have an option to display collected data from 1 day ago to present.
        ii. Users shall have an option to display collected data from 30 day ago to present.
        iii. Users shall have an option to display collected data from 365 day ago to present.
        iv. Users shall have an option to display collected data over all time.
    e. The presented smell data from the database can be filterable by the type of smell (smell name)
        i. Users shall have access to a filter to enable the display smell counts from one or more test smells.

Constraints

A major constraint surrounding the project was the adherence to open source principles. As a result of this, the team went out of their way to avoid incurring any monetary cost, specifically in regards to acquiring a means to host the server, as the average user cannot be expected to have money to spend on sure resources. When weighing their options for hosting the server portion of the application, the team reached a resolution on where to host the API services, database, and server components of the plugin: using an RIT hosted Linux virtual machine

(VM). This was an augmentation of their original solution, which was to directly host the server and database on Azure using student credits. The idea to use azure was scrapped due to difficulties surrounding server configuration, specifically the establishment of a static IP, and cost concerns. After being provided the VM credentials by the RIT IT department, the team migrated the database and the server to that VM.
The project was also, predictably, restricted to the Java language, and needed to fit into the framework of an Intellij plugin. This did limit some of the tools the team had available, sometimes out of necessity, sometimes out of convenience. The decision to use MySQL was made based on the availability of support that existing Java libraries provided, and the implementation of a Spring Boot application was done for similar reasons. The constraint helped in some ways, allowing the team to narrow down the potential options for different features. Ultimately this constraint was not one that the team minded much, as they all had a reasonably high level familiarity with the language and the tools surrounding it.

Another major constraint the team was forced to navigate was the implementation of headless mode. In short, the sponsors wanted the IntelliJ plugin to be able to run without needing to actually start IntelliJ to do so. Upon further investigation, this task proved to be more difficult that originally assumed, as many of the plugin's operations were reliant on resources that are not started until IntelliJ starts. Some ad hoc solutions were provided, albeit with some restrictions which are expanded upon in the system design.

Development Process

The development methodology the team decided to use is that of a Modified Scrum.

Why Modified Scrum?

As a team, we expected our requirements to evolve throughout the duration of the senior project journey and to have a constant loop of feedback from our sponsor to improve the new TSDetect features. As such, we looked at agile methodologies that would offer us the ability to accept our changing requirements and adjust our stories, along with their point values, based on the work needed throughout each sprint. The core elements of modified scrum, and the principles of the scrum process in general, lent themselves to the evolving nature of the project.

Listed below are the key phases of the scrum process the team followed throughout the project. These phases align with the phases of a standar Scrum process:

1. Product Backlog creation
    a. This stage is used to break the list of features the sponsors requested down into epics, stories, and spikes.
2. Sprint Planning
    a. In general, the plan was to claim the stories generated from the breakdown of one of these larger tasks and turn that into our sprint backlog.
    b. Each sprint lasted four weeks.
3. Working on Sprint

    a. The backlog was worked piecemeal, with the goal of having a working prototype by the end of each sprint.
4. Testing and Product demonstration
    a. Stories were broken up into software and test tasks, with each component being assigned to different team members. These were developed concurrently, and were reviewed separately by another team member.
5. Retrospective and next sprint planning
    a. A small amount of time was allocated at the end of each sprint to review metrics from the previous sprint and use this data to assist in allocating work for the following sprint.

What Makes This Modified From Core Scrum?

The main change to the Scrum methodology is that instead of utilizing daily standups, the team implemented weekly standups that occured during the scheduled team meetings. The reason this change was made to this scrum element is because the daily element of the meetings became redundant given the frequent communication between team members via their communication channels (Slack, Discord). The project scope is one that is so small that changes that would be made would be difficult to be do without the other team members noticing, covering accountability.

Why Not Other Methodologies?

Methodologies such as Kanban contained elements that were of interest to the team, such as the structure for tracking the work being done. This would have offered the team a high level method for tracking who was working on what feature throughout the project, but lacks the key element of story points. Scrum offering story points as a measure for the effort that a piece of work will require allows the team to better estimate their timeline and create a divisible set of work where no one team member will be given too little or too much work at any given time.

Extreme programming (XP) was another considered methodology that appealed to the team in that programming could begin at an earlier point in time. Although this was appealing in the sense that it would feel like more progress is being made, this was deemed not suitable for this project as it would require a strong design for each feature prior to this development, which XP does not offer. With this modified version of scrum, the team was able to do iterations of design and implementation giving them a balance of strong design both in terms of architecture and implementation of features.

Roles

The three key roles that exist in scrum are that of the product owner, developers, and scrum master. Compared to standard scrum, this team preserved the scrum master role and utilized newly created roles that better suit the need of the team and capitalize on each team member’s

expertise. These roles include the following: sponsor communication lead, website manager, and technical lead.

Cameron - Scrum Master
Ensures senior project submissions and development deadlines are met

Leads the team in creating and completing stories, defining requirements, and manages the Kanban board

Ben - Sponsor Communication Lead
Maintained communication with the sponsor

Leads the sponsor meetings

Spencer - Website Manager
Maintains the senior project website

Joseph - Technical Lead
Leads code reviews and approves pull requests

Leads testing efforts

Activities & Artifacts

At the core, the team tracked thier effort using stories, epics, and spikes that have a point value associated with each. These epics were be broken down into various stories, spikes, and other relevant subtasks that the team deems necessary, each of which have a scored effort tied to them called “Story Points”.

During the implementation phase, code changes underwent code reviews prior to any merge into the main codebase to ensure two developers will have viewed any new changes prior to them reaching production.

Similarly to implementation, the deliverables for the senior project will require reviews from the sponsor and/or the coach depending on the submission.

Tools

The tools used throughout the project are as listed:

- Github
- Github Kanban board
- IntelliJ IDE
- Java programming language
- Springboot, for server applications
- MySQL
- Docker

Standards and Quality Practices

To ensure the team produced strong designs and well-thought out deliverables, the following practices were integrated into thier process to manage the aforementioned components of the project:

- Development of features done outside the master branch
- Code reviews conducted before any branch is merged, or story closed.
- Testing done as a part of each story, and run continuously when applicable.

Project Schedule: Planned and Actual

The team initially planned for a total of six sprints with each delivering either a major deliverable, in our case this was features such as the TSDetect database, web server, or headless mode. Feature work was the focal point of the sprints that made up the fall semester. Following winter break, which occurred throughout the final half of December and early half of January, the team aimed to prepare documentation and activities that would be utilized during usability testing planned for sprint five. Given that this timeline was created early on in the fall semester, each deliverable seemed to be achievable based on our set of skills, planned time investment, and knowledge at the time of planning. At the time, this timeline was also supported by the team's story point values that were associated with each story that were tied to these deliverables. This allowed for a more accurate measurement in the fall portion of the timeline as the team focused on scoring these features early on.

Sprint 1 end	Sprint 2 end	Sprint 3 end		Sprint 4 end	Sprint 5 end	Sprint 6 end	Sprint 7 end
10/20/2022	11/17/2022	12/15/2022		2/16/2023	3/16/2023	4/13/2023	5/4/2023
Database, pop up, dashboard page	Server-side DB tools setup	Headless mode implemented	**BREAK**	Testing and cleanup Prep for usability testing	Usability Testing Code Freeze at end of sprint	Documentation cleanup	Cleanup

Figure 1: Project Schedule

Throughout the fall semester the team was able to deliver upon the majority of the deliverables they had planned on, such as the server application, initial headless mode, and the dashboard web application. Although the fall semester aligned well with the established timeline, that timeline became problematic upon entering the spring semester, as there was a heavy focus on usability testing, and a large unforeseen time investment was necessary to resolve the hosting situation for the TSDetect server application. As such, the effort of both sprint five and six became larger than expected and contained the aforementioned investigations and usability testing preparations.

As the team underwent usability testing, the sponsor did not express any issues with the progress being made on the usability testing but requested a slight shift in priority as there was a need for a fully functioning headless mode version of the TSDetect tool. At the time the team did not understand the high priority of this feature and continued to resolve the hosting difficulties alongside the in-class testing portion of the usability testing. As they began their review of usability testing results, the team realized that the remaining time within the semester would not allow for a well developed headless mode that would achieve the exact requirements defined by the sponsor. The team then constructed a plan to address the important elements of the headless mode feature that would be feasible in the remaining time while also completing the college deliverables associated with the project and it quickly became

a game of choosing which tasks to prioritize, which ended up being the college deliverables towards the end of the semester while working in parallel on review of usability testing results and the issues that were discovered with the headless mode solution the team had decided upon.

Overall, the timeline had a drastic shift towards the end of the spring semester given the change in prioritization and unforeseen issues that would be discovered during the development and implementation of features like headless mode. Although the spring semester timeline was less aligned with the actual work that was completed in addition to the deliverables, this resulted from a small grouping of reasons. One being the lack of time being allocated for each remaining feature as well as usability testing. There could have been a form of task breakdown to organize these deliverables and have a better understanding as to which team member was completing each task and the time invested into each. The one benefit of the spring semester was the shift in time tracking from a spreadsheet to an application known as Clockify which aided in keeping the team aligned on time commitments and accurately track the amount of time the team was spending per week on sprint work and project work.

System Design

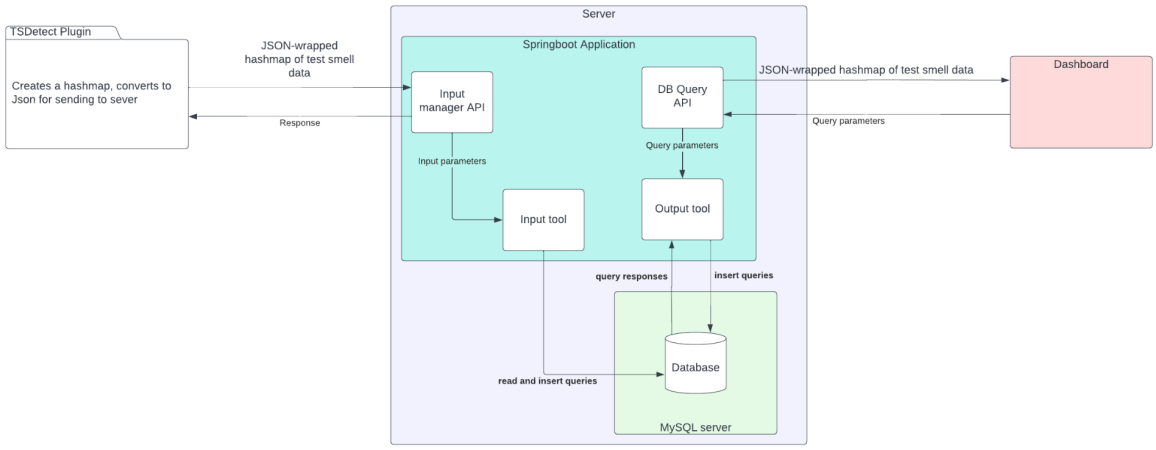


Image 2: TSdetect Data Collection Architecture Diagram

<u>Data collection and the Opt-in popup</u>

For polling users as to whether or not they want to share their anonymous data, the team decided the best approach would be to have a popup register on startup. A picture of the popup that was designed is included below

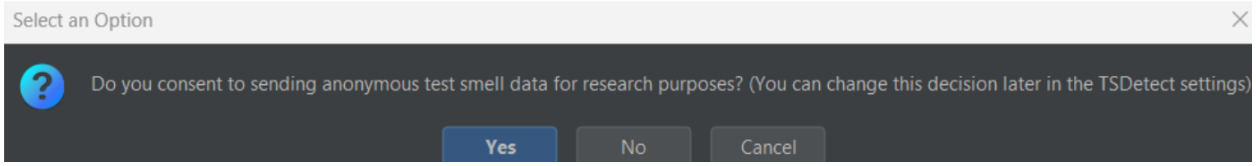


Image3: Opt-in Popup Window

Upon selecting “Yes”, a setting is saved in the application’s settings to remember the user's decision and set to “True”. Upon selecting “No”, “Cancel”, or closing the popup the aforementioned setting is populated as “False”. This is an example of the basic plugin’s functionality being augmented, as the application settings at the time of the team starting the project was purely to filter the smells the tool is testing against. Furthermore, within these settings a user can also toggle their choice originally set from the popup if they change their mind. The actual process by which the data is collected is also a basic augmentation of the tool’s original functionality: where before the tool scans through all of the files in a project’s directory, then populates the tool windows with the results, it now stores that data in a class, AnonymousData, and saves then sends the data to a centralized database.

The data collected by the plugin includes a tool generated 36 character unique user ID, the timestamp of the test run, and a list of the test smells detected with a number per each occurrence. The data is sent via a JSON wrapped HashMap, passed to a client side API to unwrap and verify the input, then passed to another server side API to be written to the database. In the event the tool is unable to connect to the database, a reconnection is attempted 5 times then saved to a local JSON file. Once the tool is able to make a successful connection, the old data is sent before the data from the new run is sent. The old data expires upon the JSON file exceeding 10,000 lines or being older than 1 month.

Database

The schema and implementation of the database was a significant consideration for the team and sponsors, so much so that a Spike was added to the schedule for the sole purpose of researching database solutions that would best fit the team’s constraints. During this spike several different options for a relational database were considered, chief amongst them being MySQL, Microsoft SQL Server, PostgreSQL, SQLite, and Oracle. Microsoft SQL Server and Oracle were both removed from consideration when the potential costs became apparent, and SQLite was discounted soon afterwards due to it’s serverless nature, leaving MySQL and PostgreSQL as the two main competitors. In the end the team decided to utilize MySQL due to its multiple Java libraries and API support, specifically its Java Database Connectivity (JDBC) support that greatly simplified the connection between the database and the server program that the team had been working on to handle the inputting and outputting of data from the database. This, combined with some simple integration with PHPMyAdmin to allow for easy database maintenance and administration, made MySQL a very appealing option.

Once the technology that the database was going to be built upon was decided, a schema for that database needed to be made. The team briefly considered making a simple collection of test smells, a table that was composed of a row for the UUID, the timestamp, and a row for each smell type. This was dismissed almost immediately in favor of a more extensible solution, as this structure would require recreating the database from scratch every time a new test smell is added to the database.

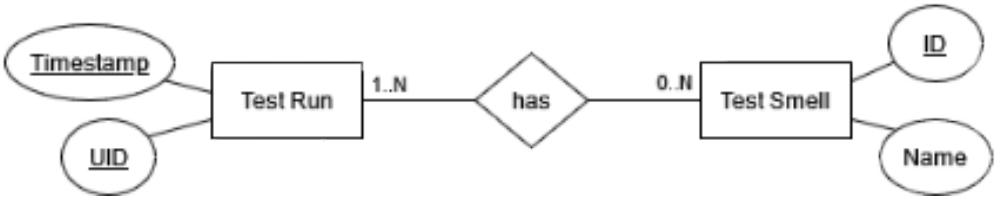


Image 4: TSDetect Database schema

This more extensible solution, detailed in the image above, broke the database up into three core tables. First the test_runs table, which owns the timestamp and uuid, and has a unique identifying integer (this integer automatically increments whenever a new test_smell entry is added). Second, the test_smells table, which contained a list of test smell names, and a unique identifier for each test smell name. Finally, a joining table called test_run_smells which has three rows: the run_id (which is associated with one of the test_runs), the smell_id (which is associated with one of the test_smells), and a quantity (representing the number of times the given test_smell was found in the given test_run). This schema makes it so that adding a test smell is as simple as making a new entry in the test_smells table, no old entries need to be purged, and the supporting server will automatically begin accounting for the new smell, so there should be minimal impact on the system, and, more importantly, there should be no need to rebuild the server.

For adding a new test_smell, or for other on-the-fly administration, there is also a PHPMyAdmin page associated with this database. This provides a quick and easy method for users (who have the username and password assigned when the database is spun up) to add new test_smells, remove entries from tables, or execute quick queries to find specific information. While it would be impractical to display all of the functionality in this document, an image of the main table hub is provided below for context:

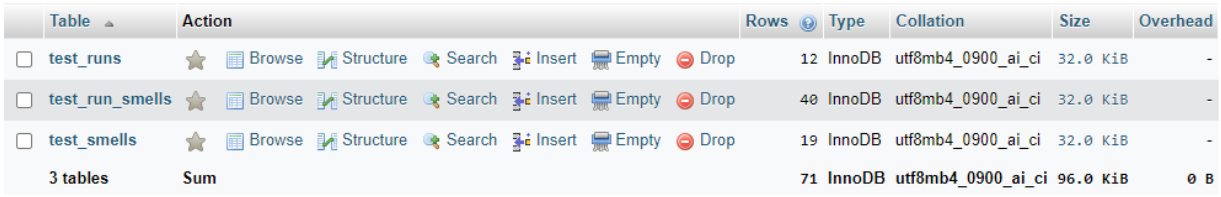


Image 5; PHPMyAdmin Core Tables View

Server

The TSDetect server application was initially conceptualized as a centralized web server that both the TSDetect tool and the dashboard could issue requests to. To be aligned with our existing requirements, the team aimed to utilize Java and had past experience using Spring Boot specifically. As such, we opted for a Spring Boot based web server that would expose various endpoints that both the tool and dashboard could access. There had not been much consideration for alternative solutions in regards to frameworks and languages given that the team aimed for consistency by sticking to the Java language and finding a suitable, well-documented solution for web servers.

Once the server solution was decided upon, configuration and hosting became the two primary components of the team's design choice. Focusing on configuration first, the server required a single configuration file for the development and production version of the server in addition to a separate set of configurations for the testing environments. For testing purposes, there exists both unit test and integration tests and the databases (or datasources) that the internal API references which brought about a simple solution backed by Spring Boot. There exists a file known as "application.properties" which is a provided configuration file that is capable of automatically being referenced within the server code and wires into the database access component. As such, we were able to create a single version of this file for the development/production that primarily focused on changing the source of each database and the testing version which performed identical configuration with the exception being database sources being pointed at the docker containers hosting the temporary testing databases.

Although not directly tied to the configuration, the Spring Boot platform enabled a simplistic approach to "containerizing" our application and running it within a Docker container. In addition to this, we may opt to utilize technologies such as Kubernetes for hosting multiple instances of the TSDetect web server providing a highly scalable solution for those requiring such setup in future development.

Server API

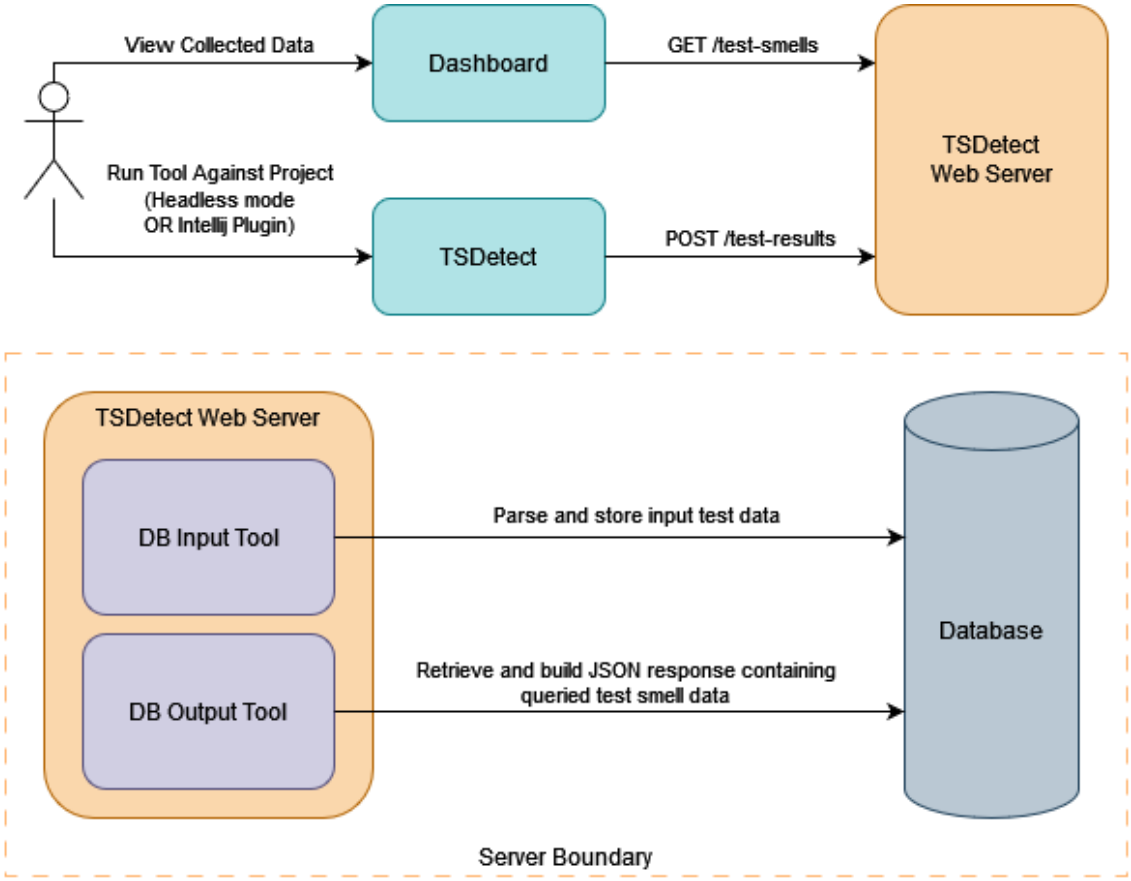


Image 6: Simplified TSDetect Database Storage and Retrival System Data Flow Diagram

With a server solution in place, the team then needed a method to handle the data being sent to and requested from the server. The obvious solution was to create an API with a pair of endpoints: one for accepting test run data from the plugin instances and inserting that data into the database, and a second for accepting requests for data and generating the queries required to retrieve the desired data from the database. The final iteration of this portion of the architecture was more-or-less unchanged from our initial design. The nature of the project locked us into a Java-based solution, and the architecture of the implementation is standard for Springboot applications, so there was not much debate to be had. The only other major attribute of the system that needed to be determined was how to construct the queries. The team approached the problem from a security mindset: we needed to use request inputs in order to form the queries, but did not want bad actors to take advantage of the system by injecting bad code. Luckily, Java provides a library for creating prepared statements with binding variables, allowing the team to easily filter incoming data and greatly reduce any vulnerabilities in the system. The two endpoints follow the standard Spring Boot architecture, and so are made up of three classes each: a controller class, a service class, and a repository-layer class; each of these with separate, specific responsibilities.

POST /test-results

The first of these is the “POST/test-results” endpoint, which is responsible for intaking test run data from various users of TSDetect. The controller class for the test-results endpoint, “TestResultController” accepts the initial POST request, then passes the responsibility of parsing the incoming data from the request body off to the service class “TestResultService”, using the returned value to determine if the input action was a success or not. If the action succeeded, a 200 response is returned back to the requesting entity, otherwise a 400 response is returned.

Once the service receives the request body, it will first ensure that the body is of the correct format. The body must take the form of a String representation of a HashMap containing at least the following “key”:”value” pairs:

- “uuid”:[A 36 character unique user ID]
- “Timestamp:[a valid timestamp with the format “yyyy-MM-dd hh:mm:ss.SSS”]

If either of these “key”:”value” pairs are missing or not formatted correctly the service will immediately return a failure response to the controller. Otherwise the service will return the same value the repository-layer class returns.

Typically the HashMap received by the service will also contain a series of “key”:”value” pairs comprised of a known test smell name as the key and an integer representing the number of times that type of test smell had been found in that particular run of TSDetect as the value. In those cases where the uuid and timestamp are present and properly formatted they will be passed on the the repository-layer class, the “DBInputTool” along with a list of those test smells and quantities. The DBInputTool will then attempt to establish a connection with the TSDetect database and use a set of prepared statements with binding variables to generate the queries necessary to input the test run data into the relevant tables; first creating a new entry in the Test-Run table with the uuid and timestamp of the run, and then creating one entry for each of the provided test smells in the Test-Smells table, keyed by the identifying number allocated to the entry made in the test-run table.

The DBInputTool will return a HashMap with a similar format to the one that was sent to it, containing the uuid, a timestamp (if provided an accurate), and each of the tests smells that were successfully added to the database. If the tool failed to make a new entry to the Test-Runs table, it will simply return a map with a key of “uuid” and a value of zero. The Service will receive this HashMap, and check if it is the “‘uuid’:0” case, if so it will return a false to the controller, informing it that the add was unsuccessful, otherwise it will return a true, informing the controller of the successful addition. Whereupon the controller will return the appropriate response message to the TSDetect user and close the connection.

GET /test-smells

The GET /test-smells endpoint was created for the purposes of allowing the dashboard to collect data from the database in real time. It follows a similar structure to the POST /test-results endpoint: it has a controller (DashboardController) that makes and manages the

connections to the Dashboard, a service (DashboardService) that handles the incoming data and directs it to the repository (DBOutputTool), which then forms the queries required to gather all of the data requested. The major difference between this GET endpoint and the POST endpoint above is in how they receive the data. Where the POST endpoint expects to receive the incoming data in the form of a request body, this GET endpoint expects a set of request parameters, each of which are optional. The parameters are as follows:

- datetime- an integer representing the number of days before the current date that data is being retrieved from.
    - eg. If given a "3", the query will look for entries with timestamps from the past three days.
    - If left blank, the query will look for test runs over all time.
- smell_type- List of smells to include in query.
    - if left blank, the query will look at all test smells.

So an example request may look like this:

- GET /test-smells/query-params/?datetime=3&smell_type=[Assertion Roulette,Verbose Test]

In an effort to both accommodate for the varying potential number of parameters, and in an effort to be extensible, the DBOutputTool's primary method used for collecting data (outTestSmells), is overloaded several times, allowing any combination of DateTime and SmellType, and allowing for multiple different data types for the time stamp (allowing future developers to either use String timestamps or the Timestamp datatype). Once the DBOutputTool has collected all of the test smells and the total number of times each has appeared given the filters, it will return a HashMap, keyed by test smell names with values equalling the quantity of that smell that appeared back up to the DashboardService. The Service then passes this value up to the DashboardController, which returns that HashMap to the requestor.

Headless mode

The ability to run the plugin in a headless environment (no user interface) did not have much initial design before development of the feature began. There is very little documentation from JetBrains or the community on running IntelliJ plugins in a headless mode. Early on, the team was not sure if it was possible to run TSDetect in a headless environment as we had found no helpful sources. With some direction from the sponsors, we were able to refine our research and find some documentation from JetBrains on running a plugin in a headless environment. To accomplish this feature, a class implementing the ApplicationStarter interface from the JetBrains Plugin OpenAPI was needed. The implementation of this class involved running the analysis functionality from the plugin and displaying the results via a generated CSV that mimics the interface of the normal plugin results. This class is run when an argument is passed to the Java Virtual Machine (JVM) specifying to run the Java process in a headless environment. The plugin utilizes Gradle as a build tool and that argument can be invoked in the Gradle command that starts the plugin. The issue with this design was that the JVM argument to run in a headless environment cannot be added at run time and must be present in the Gradle command by

default. This forces a headless environment every time the plugin is run. To mitigate this, the team utilized the project properties of Gradle which could be added to at run time. The script to run headless mode provided to the user invokes the Gradle command to start the plugin and adds any arguments provided from the script as a project property called ‘paths’. This script accepts arguments as absolute paths to projects to run the plugin on, allowing users to run the TSDetect analysis on multiple projects from one command. The Gradle command to run the plugin was modified to only invoke headless mode if the ‘paths’ project property exists, which prevents a normal run of the plugin from running in a headless environment.

There are currently some limitations to the headless mode feature. While multiple projects can be analyzed, the results will not be accurate if the project being analyzed has not been indexed by IntelliJ already. This is due to IntelliJ only indexing projects when they are opened for the first time. Running headless mode does not open an IntelliJ window and therefore the project is not indexed and cannot be analyzed. This issue could be fixed by forcing the indexer to run programmatically, but the team has been unable to accomplish this at this time. As a temporary fix, the output was updated to inform users of this issue and recommend they open the project to index it first.

Dashboard

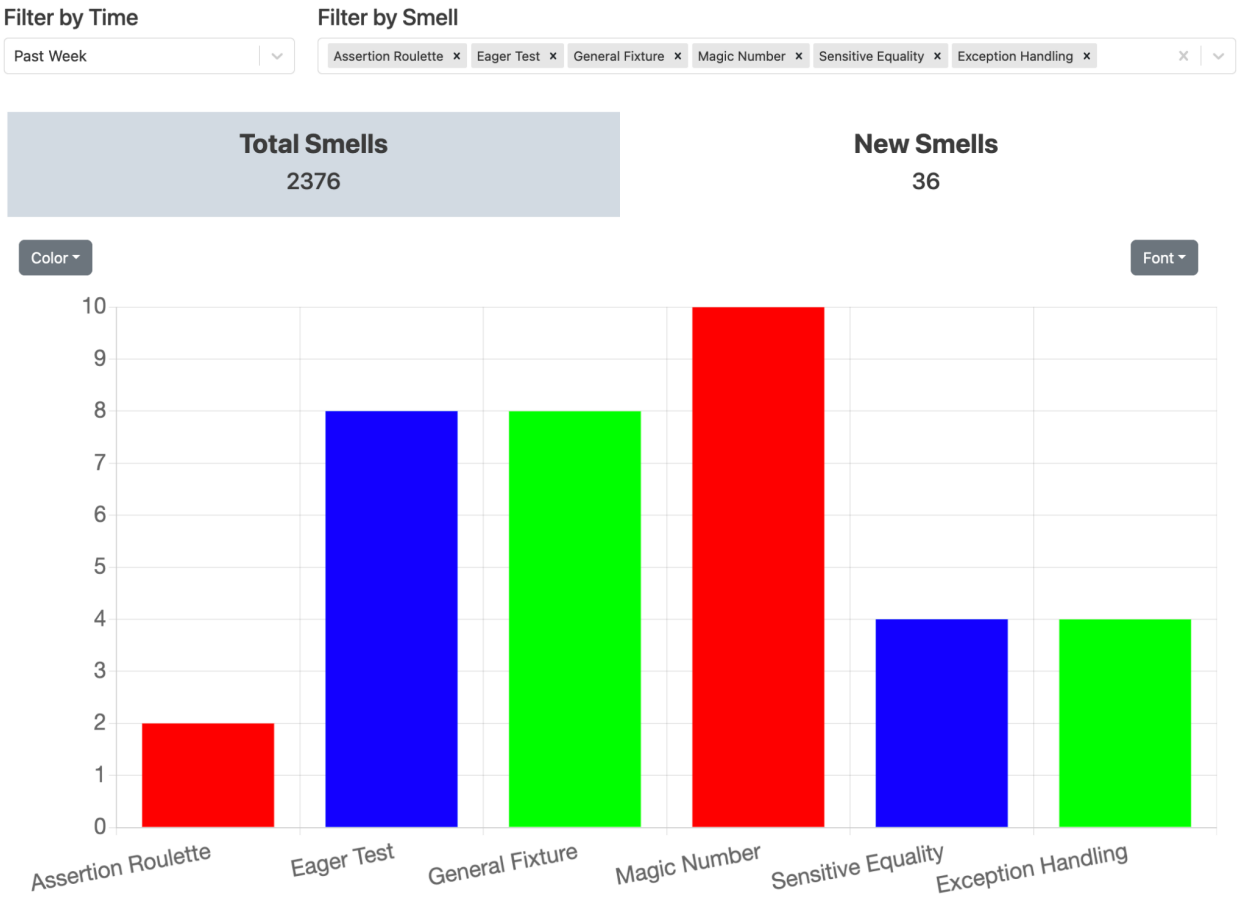
The dashboard for our system was designed as a web application to display the anonymous user data collected from the plugin. Our initial design for the dashboard included a graph and table for displaying the test smells with the ability to filter by time. The dashboard would also display the total smells in the database, new smells based on the time filter set, and the total and new users, by determining the different UID’s in the database. The layout of the dashboard was finalized after a few iterations of wireframes. The team, with the help of the sponsors, decided that while anonymous, the dashboard should not display the number of users as that could lead some to believe their data was not anonymous. The ability for users to filter by smell type along with time was added. The web app was developed using React.js with Chart.js (a graphing tool), and Reactstrap (a Bootstrap based UI library). With Chart.js, we can easily display a bar graph of the test smells, which is a main focus of the dashboard. The data is displayed on the dashboard by performing an HTTP request to the server to receive the test smell names and associated counts via a JSON string. Filtering by time adds a specification to the request, asking for smells from ‘X’ days ago to now. Filtering by smell includes the desired smell types in the request to the server.

Smell Type	Number of Occurrences
Assertion Roulette	81
Eager Test	153
Conditional Test	34
Sleepy Test	21

Image 7: Dashboard with default options. Page headers and table cut off for readability.

The use of Chart.js primarily drove our further design decisions as some limitations applied. Since we have around 20 test smells to display, the option to select multiple colors for readability was needed. However, Chart.js requires multiple data sets to allow for specific color selection on each bar. This led the team to utilize a background color array that contains hex color codes. The first hex code in the array is the primary/default color of the entire graph, which can be changed by the user via a color picker. To add more colors to the graph, users can select secondary colors. If the user adds a secondary color, it will be added to the array and that color will be displayed on every other bar. If a third color is added, that color is displayed on every third bar and so on and so forth. While this is not ideal functionality, the team and sponsors decided it allowed for readability just as well. The default font size of Chart.js can be too small to read on smaller devices. The team added a font size picker with options from 12pt to 36pt font. Both the colors and font size selected by the user are saved in the browser's local

storage for persistence upon reload of the page. Reset buttons were also added for each to allow the user to return to the default options.



Smell Type	Number of Occurrences
Assertion Roulette	2
Eager Test	8
General Fixture	8
Magic Number	10

Image 8: Dashboard with filters, multiple colors, and larger font.

Process and Product Metrics

The team utilized two metrics related to our process of Modified Scrum and three metrics related to the development of the project:

Process Metrics

- Team satisfaction: Gauging the motivation levels and enjoyment of the team members at varying stages of the project. This would be recorded in each sprint retrospective in a

dedicated sprint retrospective document that could comment on our satisfaction with the current process.
- Estimation accuracy: Planned vs actual effort for each story/task represented by story points

Project Metrics
- Code volume: # of lines of code changes each sprint, automatically tracked by Git commits
- Sprint burndown chart: Graphically shows the amount of work that has been completed in an epic or sprint, and the total work remaining
- Test metrics: Coverage, # of test cases per requirement (use case, user story etc), % of tests passed successfully

Team Satisfaction

Team satisfaction is an unconventional metric but one that we found useful. The team's desire to implement new features was directly related to our effort. We felt that when the project was going well and each team member understood the why and how of the task at hand that our productivity and communication improved.

Estimation Accuracy

Estimation accuracy tracked how well we planned for an upcoming sprint by comparing the planned effort and the actual effort it took to complete the work. Using our story points system, we are able to convert the total story points for each sprint into planned hours of effort. This metric was not useful during the fall semester, as we were not tracking our time spent on sprints and senior project work separately. We rectified this in the spring semester by using a time tracking tool called Clockify to track individual sprint effort. During Sprint 4, the team dedicated just under 75 hours of effort to development. In that time we completed four user stories and made significant progress on two others. For the completed stories, we estimated it would take 60 hours of combined effort. Given the additional progress made on two other user stories, the team feels the estimation accuracy of Sprint 4 was in line with the actual effort. Sprint 5 was by far the most time consuming as we spent just under 95 hours completing five user stories. The estimated effort for those stories was 70 hours. The difference in planned and actual effort comes from a severe underestimate of the testing involved for the stories involving the server. Sprint 6 was dedicated to the usability testing of the plugin and served as a cleanup sprint, where remaining stories were completed and some additional functionality requested by the sponsors was added. We did not estimate the planned effort for the usability testing as it was not related to the development of the plugin. However, the actual effort for Sprint 6 was just over 27 hours, which the team feels was an appropriate amount of time to conduct the usability testing and provide some last minute changes to the product.

Code Volume

The code volume for each sprint can be found in the table below:

Sprint	Additions	Deletions
1	+115	-28
2	+8709	-336
3	+438	-70
4	+4239	-1512
5	+2507	-1088
6	+685	-834

At first glance, it seems the team had varying levels of effort from sprint to sprint but that is not the case given the nature of the features we added. Sprint 1 had low additions to the TSDetect project since we were developing the database and the pop-up during that time. The database is run from a docker container and the only addition to the project was a small build file to start the database. The pop-up required more mental work as the team was still learning how IntelliJ plugins are developed. The large jump in additions from Sprint 1 to Sprint 2 was caused by a ~7000 line lock file being committed to the repository which prevented dependencies from updating and introducing breaking changes. Sprint 3 was focused on implementing the headless mode feature which required a lot of research for a relatively small code addition to the project. Sprint 4 and 5 was where the team completed the bulk of code added to the project. During this time we implemented the server and the dashboard which were more code heavy features especially considering the extensive testing involved. Sprint 6 is where the team did some clean up work on the project which resulted in more deletions than additions due to the refactoring of some components. This metric was hit or miss for the team as a lot of development was configuration which resulted in less code volume.

Sprint Burndown Chart

The sprint burndown chart metric proved to be our hardest to track. As explained above, the team did not track user story effort separate from senior project tasks like meetings in the fall semester. This led to relatively unusable burndown charts as the actual effort depicted in the graphs below includes senior project tasks.

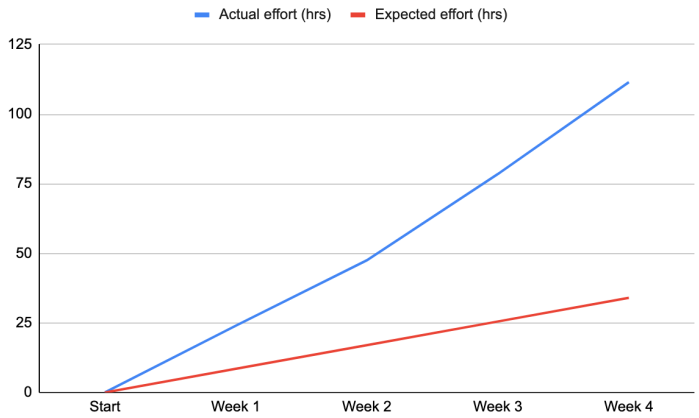


Image 9: Sprint 1 Burndown Chart

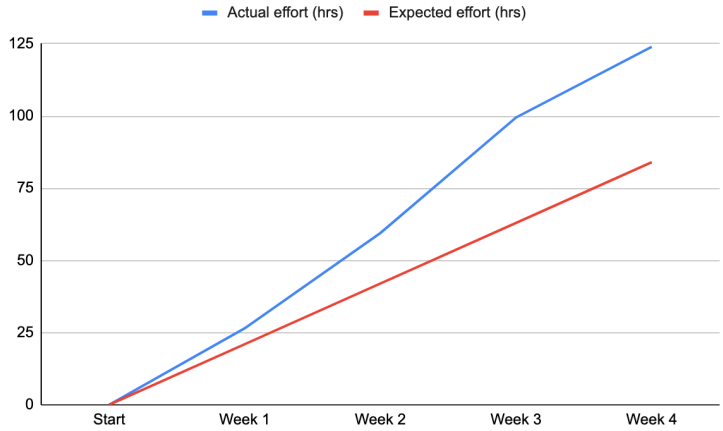


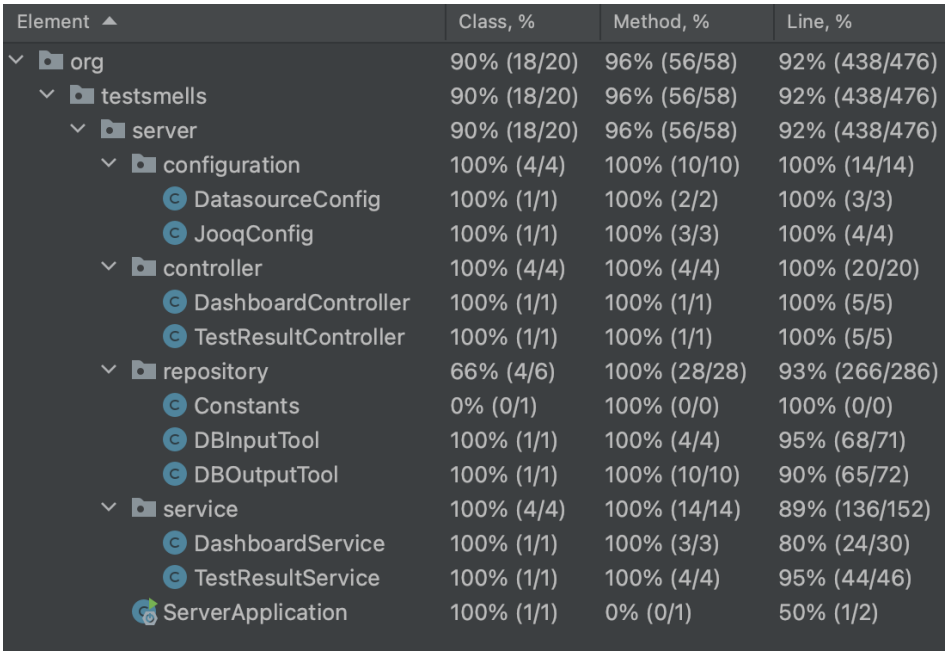
Image 10: Sprint 2 Burndown Chart

At the beginning of the spring semester, we made the choice to switch our time tracking to a web based system called Clockify. We chose this due to the ability to generate reports based on the time tracked for individual tasks. This allowed us to view how our time was spent differently on sprints vs senior project work and made our estimation accuracy metric useful. However, the way we added our time entries made us unable to generate useful burndown charts for the spring semester. While this was disappointing to the team we are still able to gather the data from the estimation accuracy to show the amount of work captured in a sprint and the total remaining work left. For sprint 4, we completed an estimated 60 hours of story point work with 70 hours of remaining estimated story point time. While we did not have story points allocated to the usability testing, we estimated it would take about 15-20 hours of our time to complete. We completed the remaining stories in sprint 5, totalling our work for the spring semester to be 130 estimated hours completed with roughly 15-20 hours remaining for the usability testing.

Test Metrics

Test metrics was an obvious choice at the beginning of the project to verify how well the team was testing the implementations. However, it is only useful from a unit test perspective for the

implementation of the server as that was the only feature able to be unit tested. The following is the code coverage of the server provided by the unit tests:



Element	Class, %	Method, %	Line, %
org	90% (18/20)	96% (56/58)	92% (438/476)
testsmells	90% (18/20)	96% (56/58)	92% (438/476)
server	90% (18/20)	96% (56/58)	92% (438/476)
configuration	100% (4/4)	100% (10/10)	100% (14/14)
DatasourceConfig	100% (1/1)	100% (2/2)	100% (3/3)
JooqConfig	100% (1/1)	100% (3/3)	100% (4/4)
controller	100% (4/4)	100% (4/4)	100% (20/20)
DashboardController	100% (1/1)	100% (1/1)	100% (5/5)
TestResultController	100% (1/1)	100% (1/1)	100% (5/5)
repository	66% (4/6)	100% (28/28)	93% (266/286)
Constants	0% (0/1)	100% (0/0)	100% (0/0)
DBInputTool	100% (1/1)	100% (4/4)	95% (68/71)
DBOutputTool	100% (1/1)	100% (10/10)	90% (65/72)
service	100% (4/4)	100% (14/14)	89% (136/152)
DashboardService	100% (1/1)	100% (3/3)	80% (24/30)
TestResultService	100% (1/1)	100% (4/4)	95% (44/46)
ServerApplication	100% (1/1)	0% (0/1)	50% (1/2)

Image 11: TSDetect Code Coverage

The team achieved 90% class coverage, 96% method coverage, and 92% line coverage. The class coverage is technically 100% as the ‘Constants’ class only defines constant variables used by the server and does not contain any implementation details. This metric was extremely helpful during development as it informed the team of where in the code the tests needed to be more extensive. The database, dashboard, headless mode, and pop-up all required code inspections or manual testing. The team emphasized extensive code inspections and manual testing and believe we have close to one hundred percent “coverage” for those features, although this metric does not really apply.

Product State at Time of Delivery

At time of delivery the team has managed to deliver on nearly all expected functionality.

A database configuration was made in a containerized fashion, allowing it to be set up and torn down as needed. A Spring Boot server application was created that hosted the API that controls dataflow into, and out of, the Database. This aspect has also been made to be deployable in a variety of environments. Specifics of how they will be deployed are ambiguous, so both the database and server components were designed to be configurable, and to work with Docker containers for added predictability.

TSDetect has been updated to prompt users if they want to opt-in to data collection for research purposes, and includes options for users to change their minds in the future through the application settings. TSDetect has also been updated to collect the test smell data of users who have opted in, and send that collected data to the server, and by extension the database. The application also has the capacity to save test run data in the event that it was unable to submit it, avoiding as much loss of data as possible.

At time of writing there is one promised component that did not quite make it to completion: the headless mode option. While it does function, it is currently restricted in its use, both requiring that the tested projects have been indexed by Intellij and requiring that a particular set of application settings be disabled in the code due to the fact that they cannot be instantiated in the command line. The team believes that this will still serve its purpose in some capacity, but will require refactoring in the future to reach its full potential.

The only feature that is completely missing is a conversion of the tool to work with the python language as a Pycharm application. This was an expected miss, as both the sponsors and the team agreed early on in the project that this was likely to be out of scope, more likely being the work of a future senior project team entirely.

Usability Testing

Per the request of the project's sponsors, a "usability test" of the TSDetect plugin was executed towards the end of the spring semester. Due to some legal and practical restrictions, this test more resembled a closed beta of the plugin, but the same general principles still applied. In preparation for this task, the team was instructed to design instructions to install the plugin, and an accompanying activity for users to complete. Following the completion of the activity, the users would be asked to complete a Google Form to rate the usability of TSDetect, detail any defects they found, and generally give feedback on the plugin. Within this survey, users would be asked to indicate details relating to their education, their programmatic experience, and a range of details relating to the usability of the plugin. The planning process began internally around March, and the test itself was executed the week of April 18th.

The test was executed in SWEN-352, a Software Testing class taught by our Faculty Coach Zhe Yu, who offered that class as a subject for the usability test. A slidedeck was presented by the senior project team, which segued into the aforementioned activity. There were some groups that reported issues either installing the plugin or getting a tool to correctly run, and most troubleshooting was during the class period. Initially the suspicion was that JUnit 5 was causing the projects the tool was being run against, but that assumption was debunked after comparing our own test code and its respective JUnit version. While the variety of systems and configurations made the issue difficult to pin down, the more reliable theory at this point is that some steps had been skipped or disregarded during the installation process, as the majority of these issue were resolved after a second, guided, instalation

After completing the activity, which involved having the students run the tool against a test suit that they had made previously, accessing the test smells present in that test suit, and attempting to correct for some of those test smells, participants were asked to complete a survey designed and provided by the seniors project team. In this survey participants were asked about their academic and software engineering experience levels, and their prior experience with IntelliJ if any. Following this, participants were asked about the overall usability of TSDetect, as the tool had not undergone any usability testing up until this point. Questions were asked regarding the tools overall usability, any bugs or defects encountered, and suggestions on how the tool could be improved in the future. Results were generally positive, with most users indicating that the tool was currently fairly usable, with some reported bugs and suggestions to improve general usability (mostly graphical or quality of life improvements). 80% of users reported encountering bugs or defects in their usage of the tool, but this number may be inflated as a result of the assignment requiring users report at least one bug or defect to earn credit after the assignment was modified by our coach. 64% of our users rated the usefulness of the tool as a 4 out of 5 or greater, with 68% of our users reporting a 3 out of 5 or greater chance of wanting to use the tool again.

Project Reflection

Requirements Gathering

The fact that this project's sponsors were also software engineers proved to be both very helpful and something of an obstacle in requirements gathering. Their understanding of software engineering principles allowed us to ask them technical questions and receive technical answers and advice, expediting the discovery phase and allowing the team to firmly define certain requirements. This background also meant that sponsor meetings could easily devolve into discussion of very low-level aspects of the system, with otherwise simple components being caught up in deliberation for weeks.

The discussion of how to keep data anonymous is a good example of a requirement that devolved into a time sink. The discussion began as a question as to how a user could potentially have their data removed from the database without collecting some identifying information. Discussions ranged from using the user's IntelliJ ID to create an identifiable UUID to attach to their results, to using their IP, not just not allowing users to identify their data at all. This topic was raised and debated in three different sponsor meetings before we found a UUID library that would eliminate the problem altogether.

Had the team taken it upon themselves to find a solution and submit it for approval instead of involving the sponsors in the conversation at that level it likely would have saved a few hours worth of meeting time. Interactions like these taught the team that it is most often better to simply take responsibility for the technical details of a project like this, even when the product owners are technically minded, as too many heads thinking on a problem will likely result in its over complication.

Sponsor Communication

Sponsor communication was a very smooth process. Electing one member of the team to be the main point of contact with the sponsors forced the entire team to be in alignment on a subject before it was brought to the sponsor's attention. This process forced the team to discuss complicated matters in-depth before committing to a course of action, and also ensured that all members of the team had at least a surface level knowledge of every component of the system. This also gave the sponsors a clear line of communication to the team, allowing them to consistently communicate with the same member of the team, and trust that he will be able to get any necessary information from the team member(s) who are most knowledgeable on the subject.

The process of sending out a weekly agenda proved to be effective, as it allowed for the sponsors to remain aware of what the team was working on. This agenda would also act as an itinerary for the sponsor meeting that followed two days after it was disseminated.

Largely, any immediate discussions with the sponsors occurred in our weekly meetings, with only a few exceptions of discussions in asynchronous formats via email and discord. These, for the most part, were only for immediate problems the teams had discovered, as the conversations that transpired in meetings were productive and insightful. Sponsor meetings dictated what the workflow would be for the following week. Originally, in the fall, our sponsor meetings were on Tuesdays with our team meetings on Thursday, but these were switched in the spring semester for scheduling reasons, but it was the opinion of the team that made our team meetings more productive, as there was now a four-day gap separating the sponsor meeting and the weekly team meeting instead of just one day.

Ultimately communication between the sponsors and the team was one of the largest strengths that this team had. Sponsors made themselves available and friendly, and the project team did not shy away from owning their mistakes or pushing back on aspects of the project that they felt were of scope or impossible. Any roadblocks were brought to all stakeholder's attention in a timely manner, discussions were held in a format that allowed everyone to feel heard, and all parties felt that interactions were constructive and pleasant.

Process

The team's process was a significant point of growth, starting first as a loose set of guidelines for conducting the senior project, and steadily growing into a more stringent set of rules that helped us to recognise and correct for deficiencies. When the deliverable for deciding on a process came up, the team decided on using scrum almost reflectively, as that is the process used in every class we had each taken thus far. Scrum did not quite agree with the level of activity we were expected to maintain however, so instead of either moving to another methodology, or agreeing to use Scrum but not adhering to all of its processes, we decided to modify it. This was fruitful, as we still had a structure that could guide our development efforts appropriately, and we did not get into the habit of saying that we don't need to follow some aspects of the methodology (a slippery slope).

In general our metric tracking was very weak in the beginning, hours and stories were tracked haphazardly and without concern for the work being done, meaning there was no clear indication of what our hours-to-story-point ratio was. By the close of the first semester the team recognized this as a major problem and took the effort required to change tools and processes (and in some cases the metric themselves) in order to better facilitate their tracking. Now, at the end of the project, the team has a much more firm understanding of both how to track valuable metrics, and how to make use of them.

Finally, the process for testing and code review was very rigid in an effort to eliminate any defects before they were introduced to the final product. This system worked extremely well for ensuring a quality product, but did delay features on more than one occasion. Ultimately we feel that this tradeoff was more than worth it, as nearly every delay was the result of having found some potential critical vulnerability and the efforts required to fix it. This was easily the most successful portion of our development plan. Ultimately, the process worked the best when it was followed strictly, and the largest holes always appeared in areas where we grew lax. This taught us practically the values of a strong process, and the support that it can provide, even when it seems annoying.

System Design

The system design ended up being one of the more stable portions of the project: the final version is very similar to the initial proposed design, with the only major deviations being made when it was discovered that the tools we were employing were more capable of handling the functionality of some proposed features than we could. The most stand-out example of this was a server-side storage that existed in early iterations of the design, the purpose of which was to hold the data from requests while they wait to be processed, so that the requestor does not need to maintain a connection, theoretically allowing for more requests to be accepted in a shorter period of time. This was completely removed when the Spring Boot application was made, as it was able to manage those connections far more efficiently than we could manually.

A few other, smaller adjustments were made over the course of the project as the team gained a better understanding of the domain and how best to utilize the resources available to us, but the broad strokes of the system remained the same. The team did learn to adjust their expectations as more spikes were completed and research was done, and were not afraid to revisit architectural documentation whenever necessary. We remain confident in the architecture designs we made, they are extensible and reasonably secure, and leave plenty of room for future adjustments.

Usability Testing

Defining the parameters for and organizing the usability test was both a logistical nightmare and an excellent learning opportunity. The desire of the sponsors to run a usability test was expressed very early on (in one of the first sponsor meetings), and at the time the team relegated it to a lower priority, justifying it by arguing that there would be nothing to test if we

failed to implement the desired features. Despite this, the idea sat in the back of our minds, and we made it a point to bring up in sponsor meetings periodically, attempting to understand the desired scope of the test. The more it was discussed, however, the more we realized that no one, not the team members, coach, or sponsors, had a firm idea of what the test should look like. As the proposed start time for the usability tests drew closer we entered a cycle of going into sponsor meetings with an idea of what the usability test requirements were, and coming out having a completely different understanding of what they should entail. It was not helped by the fact that the subject was discussed in somewhat vague terms (eg. no firm start or end dates were confirmed, nor were the desired testing subjects, or test criteria). These generalities lead everyone to fill in the blanks with their own ideas of how the tests should be run. This, combined with issues with remaining compliant with the Institutional Review Board (IRB), led to eventually scrapping the idea of a proper usability test in favor something less strict: a closed beta test.

While this new approach cleared up much of the confusion, preparing for it became another beast. The project, specifically the database, server, and dashboard, are meant to be easily deployable to anyone with the resources available. This was done deliberately for two reasons: first, because the sponsors were unsure how it would be deployed, and secondly because the sponsors did not have the resources to deploy it. Now, with this usability test coming up, the team needed to find a way to deploy the data collection assets for free, or risk not being able to test the data collection features at all. The first attempt to do so involved using Microsoft Azure to host a virtual machine, server, database, and all of the ancillary objects required to make those run in that environment. This was met with two difficulties: Microsoft Azure is very obtuse, making any efforts to use its various tools a challenge in their own right, Microsoft Azure is also not free. The team pooled together what student credits they had left in an attempt to get everything running in time, but the cost was too great, and the final result was too unreliable. Just two weeks before the testing was to begin however, the sponsors connected us with a member of RIT's IT department, who helped us requisition a virtual machine on RIT's network. Utilizing this, and the lessons learned from attempting to use Azure, the server, database, and dashboard were all set up just in time for the testing.

The results of the testing can be seen in more detail in the Usability Testing section above, but broadly speaking, the event went better than we had expected. There were some difficulties, the majority of which were resolved quickly, and the feedback gave some excellent guidance for the direction the plugin might want to go in the future. The majority of complaints involved minor graphical issues, such as the plugin not matching the color scheme of custom Intellij themes, or not being able to resize the pie chart. The data collection components were massively successful, receiving a tremendous amount of data in rapid succession. Ultimately, we consider the execution of the usability test to be a huge success.

The leadup to the usability test was full of lessons. The team learned the importance of firmly defining requirements early on, in writing, to avoid any miscommunication or confusion. We learned new technologies such as Microsoft Azure, we learned to interact with outside

organizations, and most importantly, how to have those difficult conversations with our employers.

Final Thoughts

Working on TSDetect afforded our team a myriad of opportunities to expand our professional skills, both the technical and “soft” skills. We were placed into a situation where we needed to be able to communicate efficiently in order to produce the product that the sponsors desired in the short amount of time we had available. We bonded fairly quickly, fell into the roles that best fit each of us, trusted each other, and did what was expected from each of us. That is to say, we learned how to integrate into a new team, on a new project, and deliver on the promises we made. This project gave us an introductory experience to working on a production project, and we believe, as do our sponsors, that we have proven ourselves ready to take our software engineering careers to the next level.

References

1. https://github.com/shehan
2. https://testsmells.org/
3. https://github.com/TestSmells
4. https://plugins.jetbrains.com/docs/intellij/welcome.html