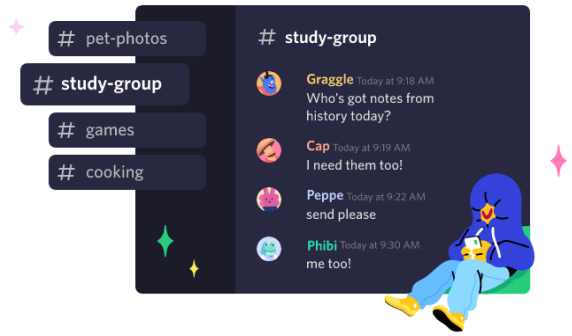
Testing Analysis of Java Discord API

Danny Do, You-Wen Luo, and Kevin Bo-Yang Song
16 March 2023

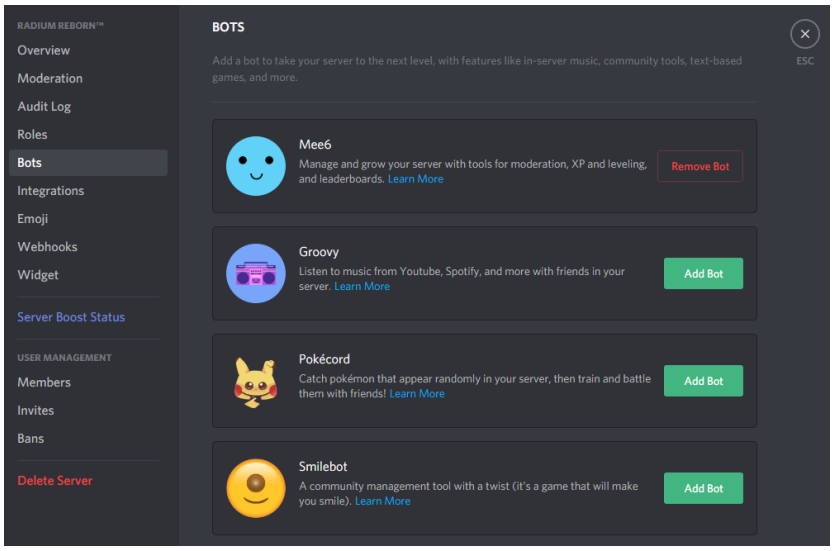
Table of Contents

An Introduction to Java Discord API (JDA)

We have decided to test the free open source widely used project "JDA." The Java Discord API allows users to create their own Discord bot. Discord bots range from small hobby bots created for personal use, to large bot projects that expand over 100,000 users. **What is Discord?** Discord



is a communication application that allows its users to message, call, and stream to each other. Users may install custom made bots into their servers of communications that add various utility to the users community. Examples include ProBot, a music playing bot that allows everyone in the same voice channel to listen to the same songs in high quality, and Karuta, a bot that facilitates a virtual card collection game where users can trade, purchase, and upgrade their e-trading cards. This API provides a clean and full wrapping of the Discord REST api and its Websocket-Events for Java, leading to easy functionality for creating a bot. The programming language in use is, of course, Java. There are over 166,000 total lines of Java code in the JDA repository.



Building with and Utilizing JDA

In order to build a Discord bot with this API, you have to first download the latest Java JDK, and an IDE, preferably IntelliJ. Create a new Java Maven project in the folder of your choice. Once created, add the dependency into the pom.xml folder, and reload the Maven project so you can use the JDA in your work. The following text that should be added can be found in the JDA repo page:

```
<dependency>
    <groupId>net.dv8tion</groupId>
    <artifactId>JDA</artifactId>
    <version>VERSION</version>
</dependency>
```

Change the above VERSION to your working JDA version. Once you have reloaded the project you are ready to start building a bot in your IDE.

Visit the Discord developer portal here: https://discord.com/developers/applications. You will have to sign in with your Discord account. Here you create the new bot application, and then get a token that will let your Java code point to the bot.

There are two approaches when coding your bot: sharding and non-sharding. Sharding allows your bot to run on multiple Discord servers at once, so multiple people can use it.

Here is an example of a simple bot code:

```java
package org.example;

import ...

2 usages
public class NewBot {

    1 usage
    private final ShardManager shardManager;
    1 usage
    public NewBot() throws LoginException {
        String token = "TOKEN";
        DefaultShardManagerBuilder builder = DefaultShardManagerBuilder.createDefault( token: "TOKEN");
        builder.setStatus(OnlineStatus.ONLINE);
        builder.setActivity(Activity.watching( name: "exampleTV"));
        shardManager = builder.build();
    }
    no usages
    public static void main(String[] args) {

        try{
            NewBot bot = new NewBot();
        } catch (LoginException e){
            System.out.println("ERROR: Invalid token!");
        }
    }
}
```

TOKEN will be your specific bot token created when you use the Discord Application portal mentioned above.

Once you have created your bot and are ready to add it to your server, go to the Discord Application portal > OAuth2 >URL Generator. Check the boxes “bot” and “application commands.” Then, select the permissions your bot has access to in the servers it is added to. You can finally use the given URL to add bots to servers.

Existing Test Cases

The existing test cases can be found in the test folder of the project repo.

I.E. (JDA/src/test)

Currently, the test cases are all JUnit tests. There's a separate folder containing test cases for testing entity strings. With AnEntity.java creating the entity object, and ASnowFlake.java that gets the ID of the snowflake. As of now, the ID of the snowflake is set at 42. The final file in the Net folder is the entity string test, which includes 9 test methods ordered from 1 to 9. The tests are:

- testSimple() - test simple entity string
- testClassNameAsString() - test class name as entity string
- testType() - test type of entity string
- testMetaData() - test the metadata stored inside the entity string
- testAll() - test all the functionalities of entity string
- testSimpleSnowflake() - testing a simple snowflake object
- testTypeSnowflake() - testing type of snowflake
- testMetaSnowflake() - testing metadata of snowflake object
- testAllSnowflake() - testing all data from snowflake

Outside of the net folder, we have the current test cases for different functionalities of the code. They are all included in separate test files. These files are

- CommandDataTest.java

- DataPathTest.java
- HelpersTest.java
- JsonTest.java
- MarkDownTest.java
- MarkDownUtilTest.java

Break Down of the files

CommandDataTest.java

- 7 tests
- testChoices()
- testNameChecks()
- testRequiredThrows()
- testSubcommandGroup()
- testSubcommand()
- testNormal()
- testDefaultMemberPermission()

DataPathTest.java

- testSimple()
- testSimpleMissing()
- testObjectInArray()
- testArrayInObject()
- testArrayInArray()
- testComplex()

HelpersTest.java

- testIsEmpyt()
- testContainsWhiteSpace()
- testIsBlank()
- testCountMatches()
- testTruncate()
- testLeftPad()
- testRightPad()
- testIsNumeric()
- testDeepEquals()

JsonTest.java

- testParse()
- testJsonToString()

MarkDownTest.java

EscapeMarkdownAlltest

- testQuote()
- testStrike()
- testSpoiler()
- testCodeBlock()
- testUnderscore()
- testAsterisk()

EscapeMarkdownTest

- testComplex()
- testBold()
- testTrivial()
- testItalics()
- testBoldItalics()
- testUnderline()
- testStrike()
- testMono()
- testSpoiler()
- testMonoTwo()
- testBlock()
- testQuote()

IgnoreMarkdownTest

- testComplex()
- testBold()
- testTrivial()
- testItalics()
- testBoldItalics()
- testUnderline()
- testStrike()
- testMono()

- testSpoiler()
- testMonoTwo()
- testBlock()
- testQuote()

MarkdownTest

- testComplex()
- testBold()
- testTrivial()
- testItalics()
- testBoldItalics()
- testUnderline()
- testStrike()
- testMono()
- testSpoiler()
- testMonoTwo()
- testBlock()
- testQuote()

MarkDownUtilTest.java

- testBold()
- testItalics()
- testBoldItalics()

- testUnderline()
- testStrike()
- testMonospace()
- testSpoiler()
- testCodeBlock()
- testQuote()
- testQuoteBlock()
- testMaskedLink()

Running of the above test cases are relatively simple. By using VSCode, we are able to run the test cases through the testing files by using the run button next to each test case.

Testing

The Java Discord API (JDA) allows users to add slash commands to control the robot they created. The systematic functional testing and partition testing part will focus on the command data that are parsed by the JDA. The boundaries of the slash commands can be

(1) Is there any option/choices available in the slash command?

(2) If there is, how many options/choices are there?

(3) Are the options required or optional?

(4) What happens when adding invalid choices?

Therefore, we partitioned the command input test into 4 parts.

-`public void testSlashCommand()`

-`public void testSlashCommandWith1RequiredOption()`

-`public void testSlashCommandWith1RequiredOption1OptionalOption()`

-`public void testCommandWithChoices()`

First, simply test the slash command input "/bark" without the option input. This command allows the bot to say the content is hardcoded in the JDA program, so the user can not change in Discord. The `testSlashCommand()` will specify whether the JDA parses the correct string in `CommandData`.For example, the command name and its description.

Second, test the slash with one required option. Take the new added slash command "/say" as an example, the user can make a bot send a message in the current Discord channel. The message option is required. For instance, `testSlashCommandWith1RequiredOption()` will check not only the command name and its description, but also the option name, description, and whether it is set to required.

Third, test the slash command with two options, one is required, and the other is optional. For example, add a new "/chat" command using JDA. The first option is content, which is required, and the second is channel, which is optional. If the user specifies a channel in the command, the bot will send chat content to the assigned channel. If not, the default channel is the one that the user uses the command. The test function

`testSlashCommandWith1RequiredOption1OptionalOption()` will check the command, command description, the first option's name, whether this option is set to required like the previous test, and the second option's name, description, and whether it is set to optional. It seems to be redundant to test the previous feature again and again; however, redundancy is an important element in software testing that allows us to find existing bugs at a higher chance.

Finally, test the command with choices by adding valid and invalid format of choices. For example, add a new "/mood" command with the required "type" option, and add choices to that option. To begin with, the `testCommandWithChoices()` tests whether the "type" is set to a required option as previously stated. Second, try to add choices with invalid types (The expected type of the option choices should be string), with int and float values, and see whether the JDA throws an `IllegalArgumentException`. Finally, add three valid choices into the option and the `choices` array respectively, and see whether the option choices size and content equals the array.

In conclusion, the JDA passed all of the partition tests mentioned above. We can assume JDA is a reliable API at this time.

The link below is my github repo including JUnit test. The main/java/org/example contains the TestBot.java, which is the bot we created to test in a discord app, and the commands/CommandManage.java for adding new commands to the bot. The test/java/org/example/commands contains CommandDataTest.java, which includes 4 JUnit tests mentioned above.

Our project repository: https://github.com/vicluo96/SWE261_JDA_bot_testing

Functional Testing Java Discord API

How Finite Models Can Be Useful for Testing?

Finite models can be useful for software testing because they provide a way to rigorously and systematically examine the behavior of a system. By representing the system as a finite model, it becomes possible to automate the testing process and ensure that all relevant scenarios and conditions have been considered.

Finite models can also help to identify corner cases, edge cases, and other scenarios that might not have been considered during the design and development of the software. By exposing these scenarios through testing, it becomes possible to identify and fix bugs or other issues before the software is released to the public.

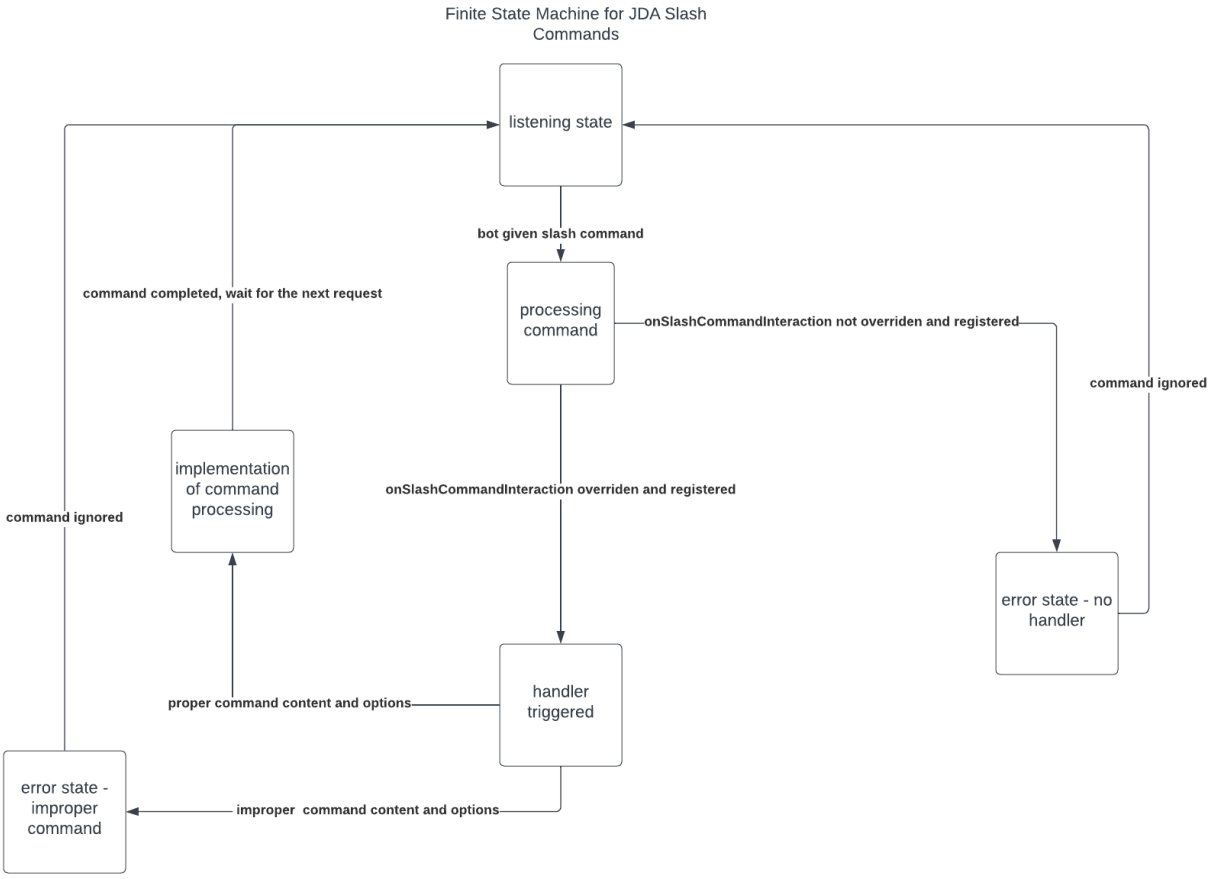
Furthermore, finite models can provide a way to test software in a controlled and predictable environment, which is particularly important in critical systems such as those used in aviation, finance, or healthcare. By using finite models, it is possible to isolate specific parts of the system and test them in isolation, which can help to reduce the risk of unexpected interactions or dependencies between different components.

In summary, finite models can help to improve the reliability and quality of software by providing a systematic and automated approach to testing. By using finite models, software developers and testers can identify and fix bugs and other issues early in the development process, ensuring that the software meets the required specifications and works as expected in all relevant scenarios.

Choose a feature or component that lends itself well to being described by a non-trivial functional model (specifically or ideally by a finite state machine).

Describing an API feature as a Finite State Machine

We choose the ListenerAdapter component and its use for slash commands in Java Discord API (JDA) as the functional model described as a finite state machine.



In JDA, we can create our own robot with slash commands. To do this, we built the CommandManager class that inherits from the ListenerAdapter. First, we implemented the onSlashCommandInteraction method, to check whether the command name from the event

equals the ones we have specified in the commandData array in onGuildReady method. If the command is valid, check whether there are required or optional options, and wait for the user to enter the content. When all the required options have content, the command is complete, and the robot will do the action. If the command does not exist or the input format is invalid, throw exceptions.

In the finite state machine point of view, at first, the robot is in the listening state, waiting for the slash command. After the user enters a command, the robot will switch to processing command state, and the API will search for the matching command handler. If the matching command handler does not exist, it will switch to the error state, the command will be ignored, and the robot back to listening state. Otherwise, the matching handler will be triggered, the robot switches to the handler triggered state, waiting for the user to enter required or optional options. When the user completes all the options, the handler will check whether all the required options are filled, and whether the input format is valid. If not, the robot switches to the error state, ignores the command, and returns to the listening state. Otherwise, the command is complete, the robot switches to the implementing command state, does the action corresponding to the command, and returns to the listening state, waiting for the next command. This is the general case of finite state machine description of the CommandManager that extends the LinstenerAdapter component in JDA.

Testing using the Functional State Model

For the FSM testing portion, we decided to test out the slashBotExample.java, which contained the contents and the details of the bot as well as their respective functionalities.

The goal of testing is to ensure that our prescriptive architecture drawings, the FSM drawings previously shown. Here's a detailed explanation of the states and their expected next states.

Listening state - when the slash bot is open to receiving commands

Processing command - taking a dive into current commands

Error no handler - when the command is nonexistent or invalid, FSM comes back to the listening state to anticipate another command

Handler triggered - when the command is valid so that the bot chooses the specific handler to process the command

Implementation of command processing - the result of the processing of the valid command handler when there is a valid command content, outputs the command

Error improper command - when the command given is not proper, therefore we output the error output command.

The testing files of the slashbot example have been included in the testing file section of the project. We included a slasher test file which will test the basic functionalities with respect to the different commands as well as if they are able to provide the correct output. The testing for the slasher bot would have to be done through the discord application - creating a discord slasher bot and then giving it input commands.

Testing file: SlasherTest.java

Testing functions:

testSubCommands() - testing if the bot responds correctly to commands within commands

testSubCommandGroup() - testing if the bot responds correctly to groups of sub commands

testChoices - testing if the bot responds correctly to command choices

testNameChecks() - testing if the bot responds correctly to checking of names

testPrunePermissions() - testing the permissions of prune action

testLeavePermissions() - testing the permissions of leave action

testSlashSay() - test command say for slash bots

testSlashLeave() - test command leave for slash bots

testSlashCommandAssignUserRole() - test command when assigning user role for slash bots

testCommandError() - error in ban commands

testSubCommandError() - test to see if there are errors in sub commands

Github link: https://github.com/dalo390/JDA

White Box Testing Java Discord API

Structural Testing(White-box testing)

Structural testing, or white-box testing, is a testing technique that judges the testing suite's thoroughness based on the structure of the program itself.

Key question to answer when implementing structural testing is asking "What is missing in our test suite?" If we have parts of the program that are not covered by the current test suite, then we can not find faults if that part of the system is faulty. Parts, in this case, are considered to be control flow elements or combinations of control flow elements.

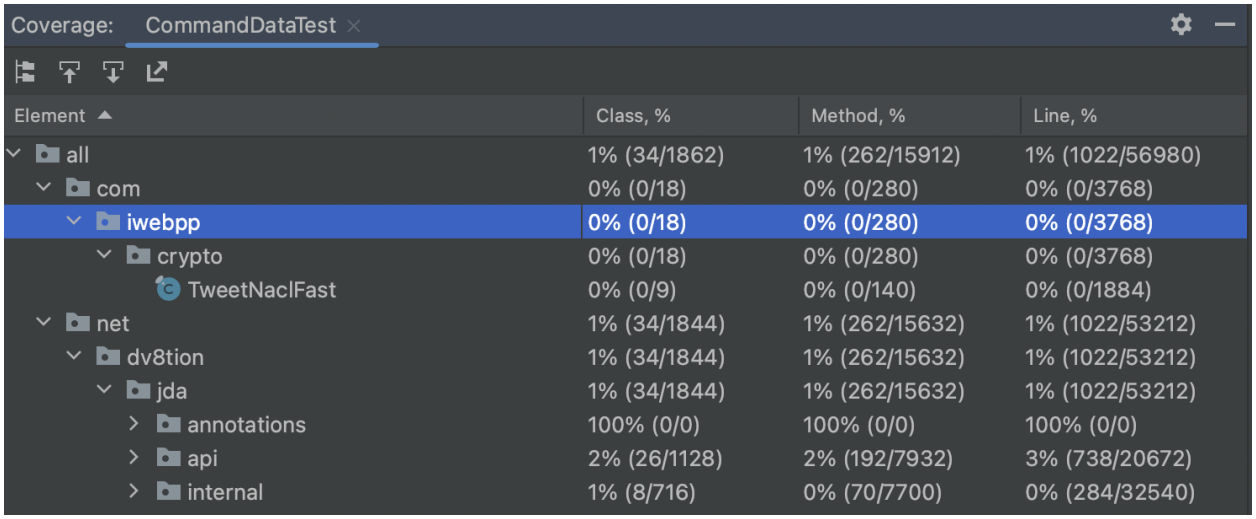
Although executing all control flow elements does not guarantee finding faults, structural testing is sufficient to increase the confidence of the thoroughness of the existing testing suite by removing obvious inadequacies.

Documentation of existing test file coverage in Java Discord API (JDA):

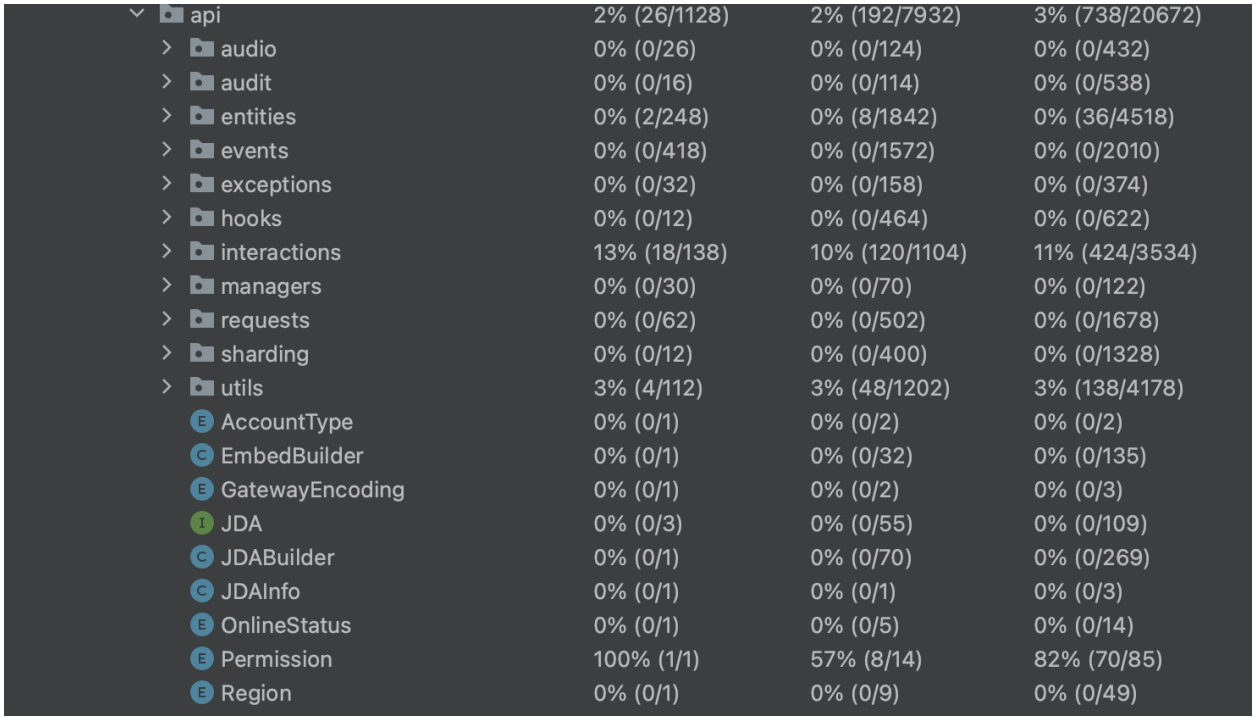
1. CommandDataTest
2. DataPathTest
3. HelpersTest
4. JsonTest
5. MarkDownTest
6. MarkDownUtilTest

CommandDataTest:

In this very first test file, we will go through each sub package to show the hierarchy and structure of the JDA source code:

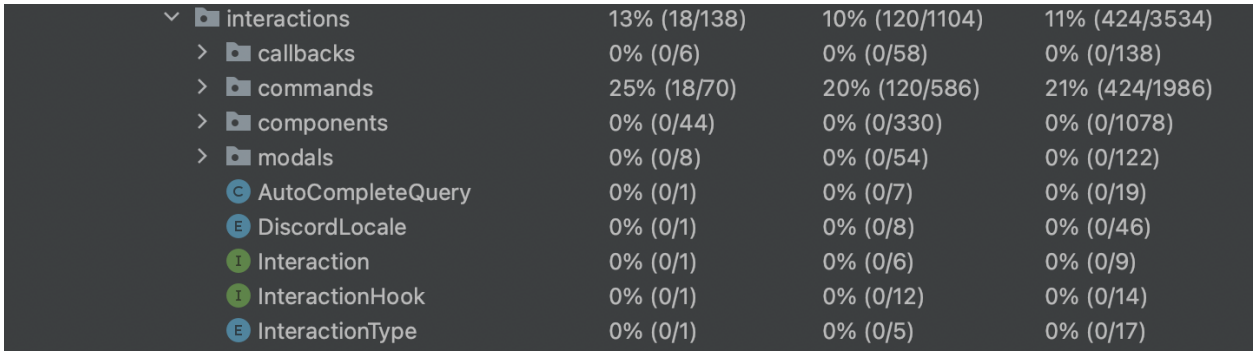


As we can see in the overall coverage, CommandDataTest mainly focuses on testing the components in the api package. Here is the detail of class, method, and line coverage in api package:



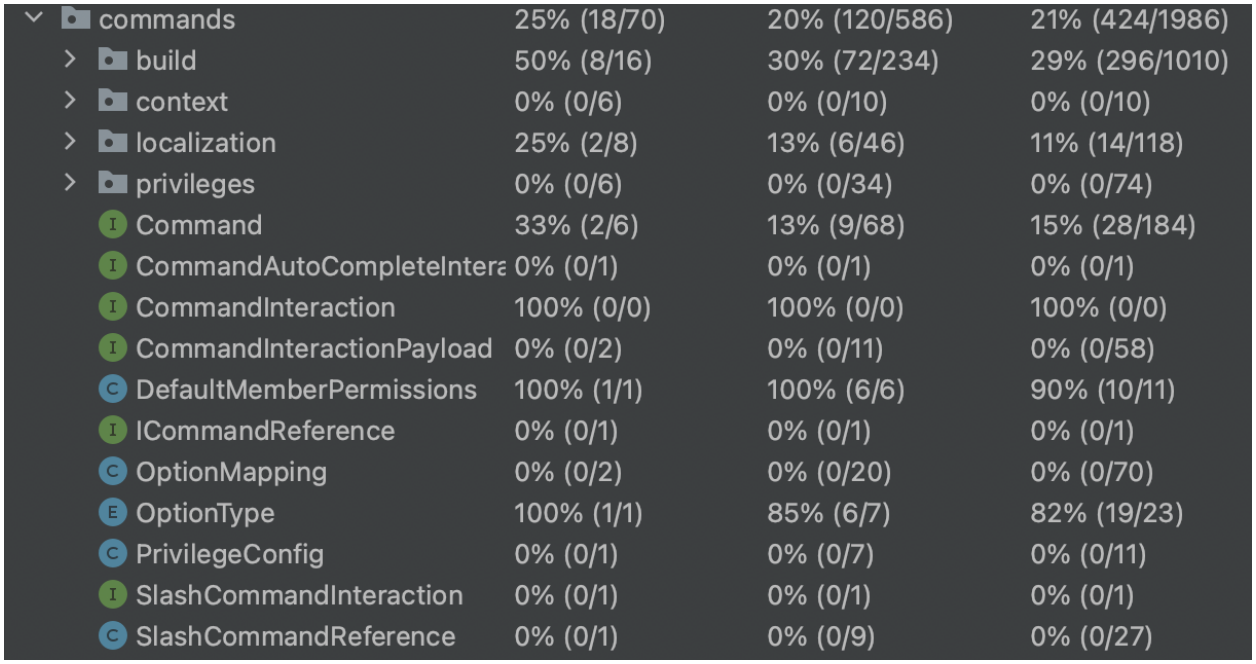
api	2% (26/1128)	2% (192/7932)	3% (738/20672)
audio	0% (0/26)	0% (0/124)	0% (0/432)
audit	0% (0/16)	0% (0/114)	0% (0/538)
entities	0% (2/248)	0% (8/1842)	0% (36/4518)
events	0% (0/418)	0% (0/1572)	0% (0/2010)
exceptions	0% (0/32)	0% (0/158)	0% (0/374)
hooks	0% (0/12)	0% (0/464)	0% (0/622)
interactions	13% (18/138)	10% (120/1104)	11% (424/3534)
managers	0% (0/30)	0% (0/70)	0% (0/122)
requests	0% (0/62)	0% (0/502)	0% (0/1678)
sharding	0% (0/12)	0% (0/400)	0% (0/1328)
utils	3% (4/112)	3% (48/1202)	3% (138/4178)
AccountType	0% (0/1)	0% (0/2)	0% (0/2)
EmbedBuilder	0% (0/1)	0% (0/32)	0% (0/135)
GatewayEncoding	0% (0/1)	0% (0/2)	0% (0/3)
JDA	0% (0/3)	0% (0/55)	0% (0/109)
JDABuilder	0% (0/1)	0% (0/70)	0% (0/269)
JDAInfo	0% (0/1)	0% (0/1)	0% (0/3)
OnlineStatus	0% (0/1)	0% (0/5)	0% (0/14)
Permission	100% (1/1)	57% (8/14)	82% (70/85)
Region	0% (0/1)	0% (0/9)	0% (0/49)

As the image shown above, the CommandDataTest mainly tests the interactions package, which is in charge of the interaction between the robot and the user. To dive in more detail, here is the coverage of the components inside this package.



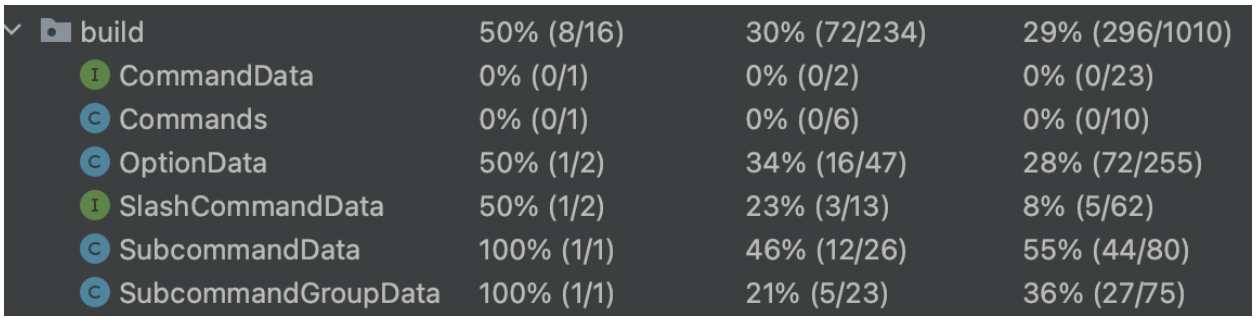
interactions	13% (18/138)	10% (120/1104)	11% (424/3534)
callbacks	0% (0/6)	0% (0/58)	0% (0/138)
commands	25% (18/70)	20% (120/586)	21% (424/1986)
components	0% (0/44)	0% (0/330)	0% (0/1078)
modals	0% (0/8)	0% (0/54)	0% (0/122)
AutoCompleteQuery	0% (0/1)	0% (0/7)	0% (0/19)
DiscordLocale	0% (0/1)	0% (0/8)	0% (0/46)
Interaction	0% (0/1)	0% (0/6)	0% (0/9)
InteractionHook	0% (0/1)	0% (0/12)	0% (0/14)
InteractionType	0% (0/1)	0% (0/5)	0% (0/17)

To explore more detail, we can see the coverage in command package:



commands	25% (18/70)	20% (120/586)	21% (424/1986)
build	50% (8/16)	30% (72/234)	29% (296/1010)
context	0% (0/6)	0% (0/10)	0% (0/10)
localization	25% (2/8)	13% (6/46)	11% (14/118)
privileges	0% (0/6)	0% (0/34)	0% (0/74)
Command	33% (2/6)	13% (9/68)	15% (28/184)
CommandAutoCompleteIntera	0% (0/1)	0% (0/1)	0% (0/1)
CommandInteraction	100% (0/0)	100% (0/0)	100% (0/0)
CommandInteractionPayload	0% (0/2)	0% (0/11)	0% (0/58)
DefaultMemberPermissions	100% (1/1)	100% (6/6)	90% (10/11)
ICommandReference	0% (0/1)	0% (0/1)	0% (0/1)
OptionMapping	0% (0/2)	0% (0/20)	0% (0/70)
OptionType	100% (1/1)	85% (6/7)	82% (19/23)
PrivilegeConfig	0% (0/1)	0% (0/7)	0% (0/11)
SlashCommandInteraction	0% (0/1)	0% (0/1)	0% (0/1)
SlashCommandReference	0% (0/1)	0% (0/9)	0% (0/27)

Finally, we found the relatively high coverage in Enum OptionType and class DefaultMemberPermisssions. On the other hand, the build package may also be the target of this test file, let's take a look:



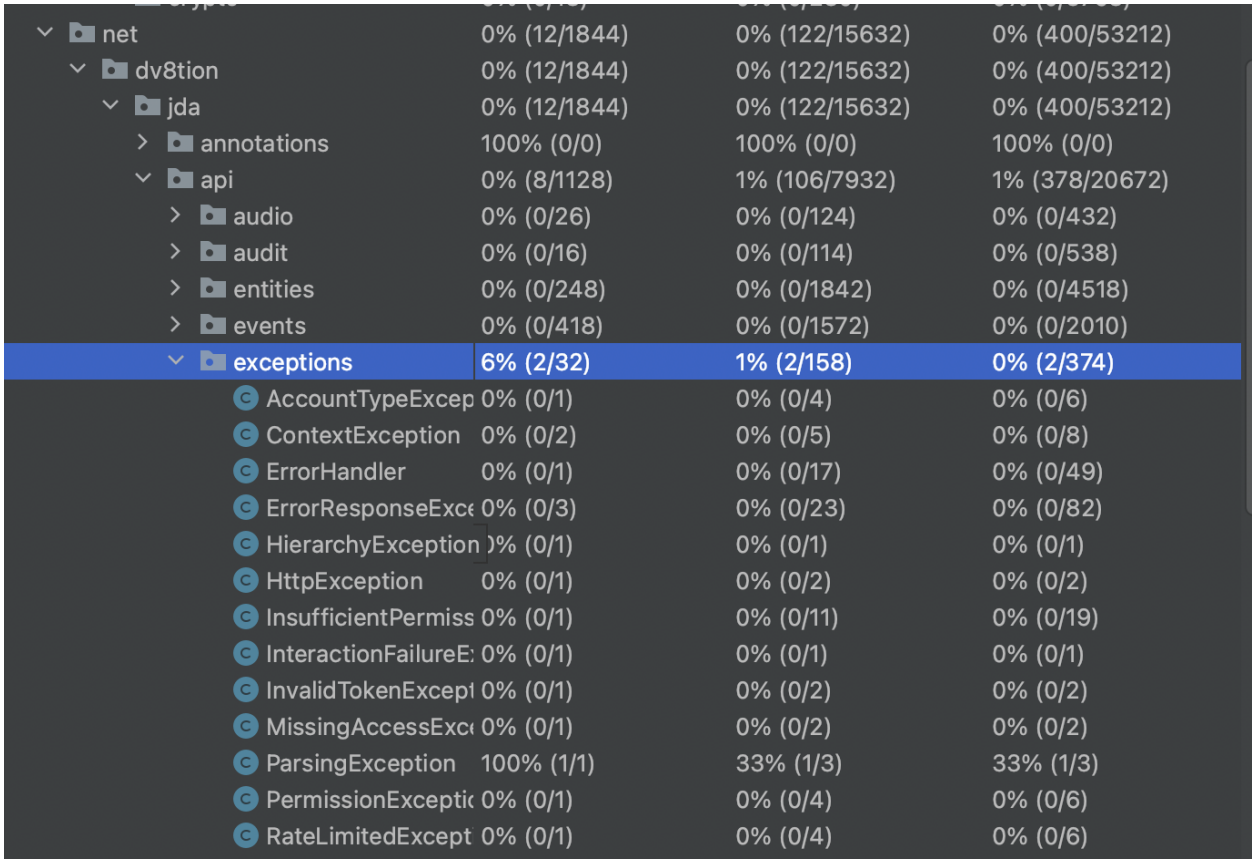
build	50% (8/16)	30% (72/234)	29% (296/1010)
CommandData	0% (0/1)	0% (0/2)	0% (0/23)
Commands	0% (0/1)	0% (0/6)	0% (0/10)
OptionData	50% (1/2)	34% (16/47)	28% (72/255)
SlashCommandData	50% (1/2)	23% (3/13)	8% (5/62)
SubcommandData	100% (1/1)	46% (12/26)	55% (44/80)
SubcommandGroupData	100% (1/1)	21% (5/23)	36% (27/75)

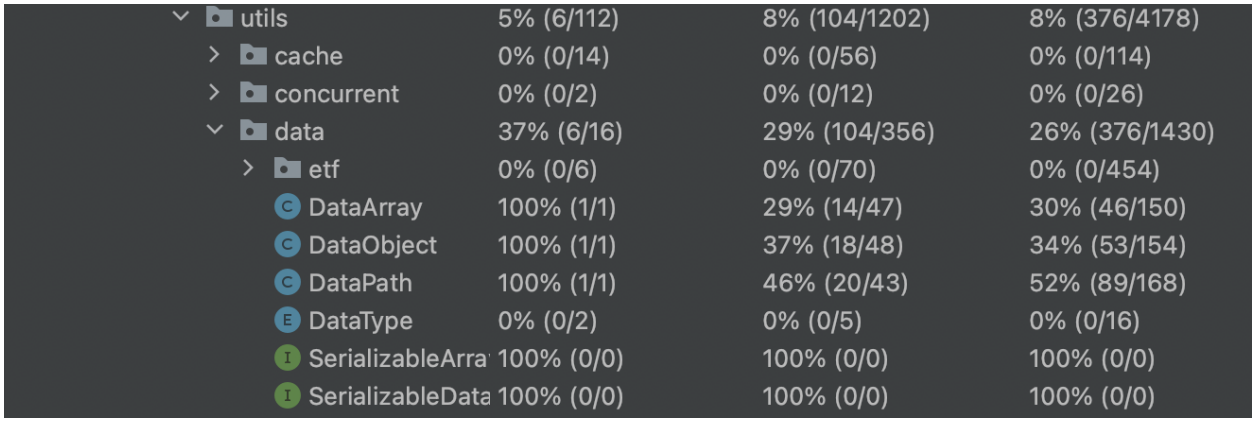
See the last four components also have relatively high coverage.

Now we can conclude that this test file aims at testing commands with option types, permissions, and its option data, slash command data, subcommand data, and subcommand group data while building. Obviously, there is still room to improve the coverage of CommandDataTest()

DataPathTest:

This test covers relatively low in overall coverage. We used the same method to find its testing target as we did previously:





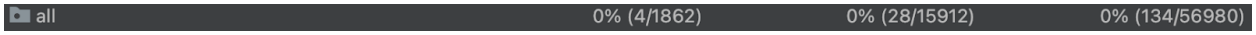
utils	5% (6/112)	8% (104/1202)	8% (376/4178)
cache	0% (0/14)	0% (0/56)	0% (0/114)
concurrent	0% (0/2)	0% (0/12)	0% (0/26)
data	37% (6/16)	29% (104/356)	26% (376/1430)
etf	0% (0/6)	0% (0/70)	0% (0/454)
DataArray	100% (1/1)	29% (14/47)	30% (46/150)
DataObject	100% (1/1)	37% (18/48)	34% (53/154)
DataPath	100% (1/1)	46% (20/43)	52% (89/168)
DataType	0% (0/2)	0% (0/5)	0% (0/16)
SerializableArra	100% (0/0)	100% (0/0)	100% (0/0)
SerializableData	100% (0/0)	100% (0/0)	100% (0/0)

It shows that this test file put emphasis on api/exceptions and api/utils/data

In the data package, classes such as DataPath, DataObject, DataArray have higher coverage percentage.

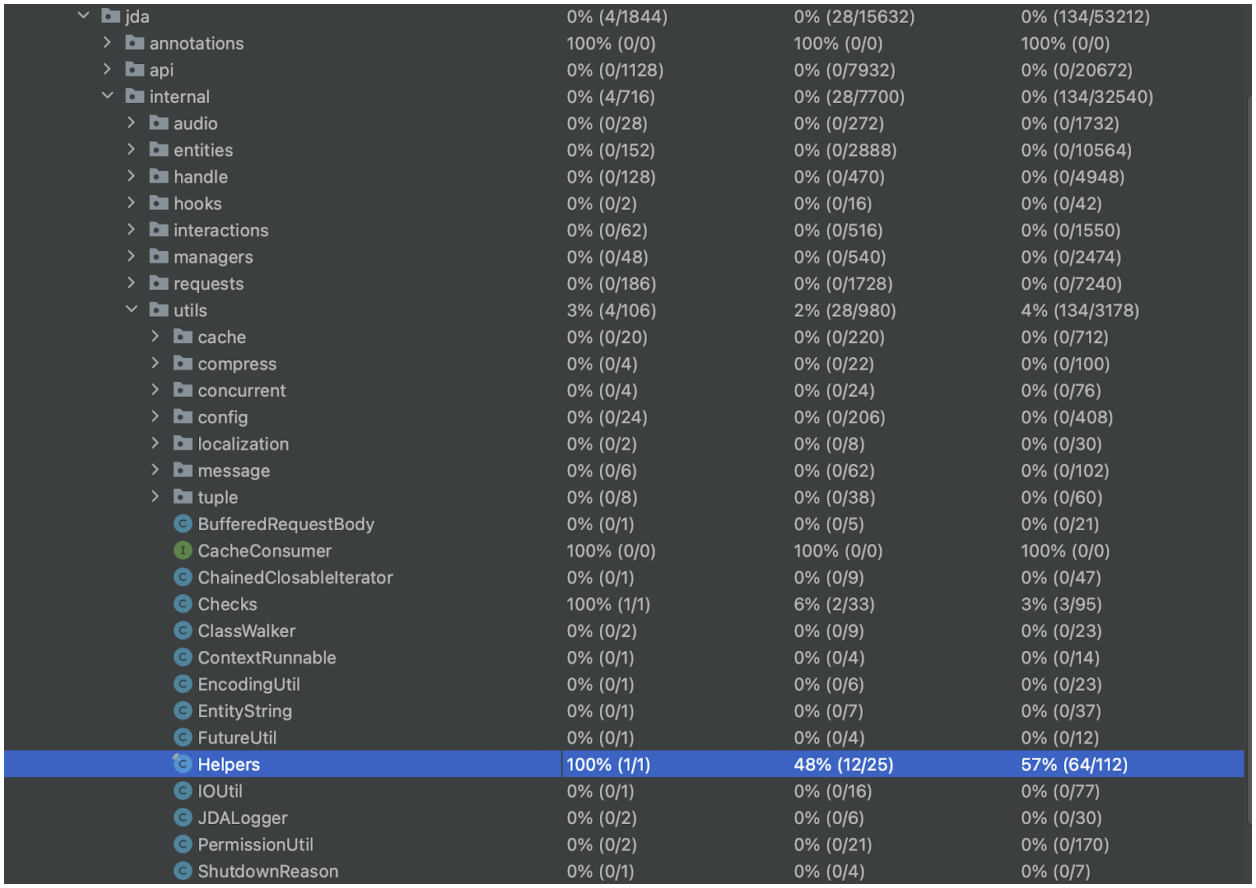
HelpersTest:

Overall overall coverage:



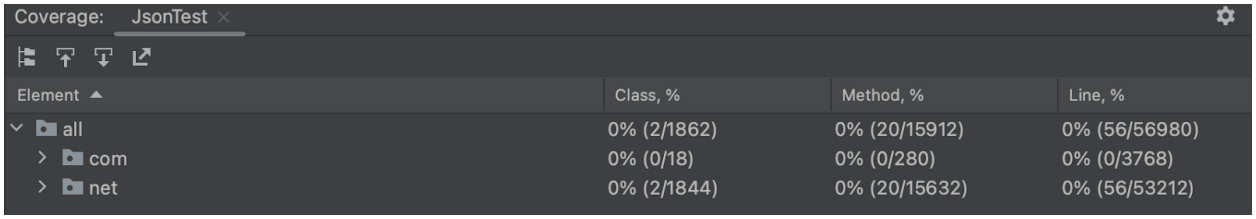
all	0% (4/1862)	0% (28/15912)	0% (134/56980)

As the image shown below, this test focuses on the internal package instead of api. The highest coverage lies in the internal/utils/Helper class, which have 48% method coverage and 57% lines coverage.



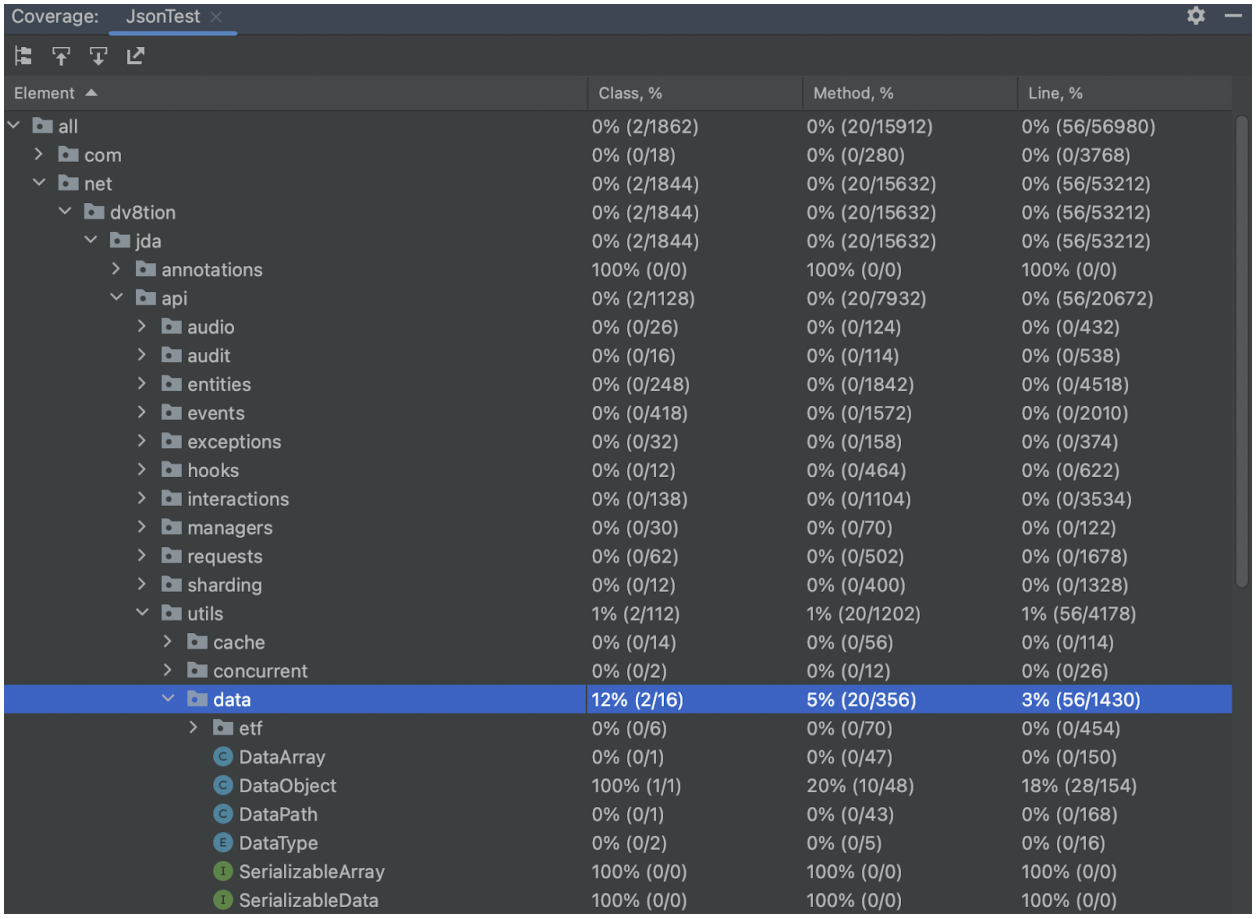
jda	0% (4/1844)	0% (28/15632)	0% (134/53212)
annotations	100% (0/0)	100% (0/0)	100% (0/0)
api	0% (0/1128)	0% (0/7932)	0% (0/20672)
internal	0% (4/716)	0% (28/7700)	0% (134/32540)
audio	0% (0/28)	0% (0/272)	0% (0/1732)
entities	0% (0/152)	0% (0/2888)	0% (0/10564)
handle	0% (0/128)	0% (0/470)	0% (0/4948)
hooks	0% (0/2)	0% (0/16)	0% (0/42)
interactions	0% (0/62)	0% (0/516)	0% (0/1550)
managers	0% (0/48)	0% (0/540)	0% (0/2474)
requests	0% (0/186)	0% (0/1728)	0% (0/7240)
utils	3% (4/106)	2% (28/980)	4% (134/3178)
cache	0% (0/20)	0% (0/220)	0% (0/712)
compress	0% (0/4)	0% (0/22)	0% (0/100)
concurrent	0% (0/4)	0% (0/24)	0% (0/76)
config	0% (0/24)	0% (0/206)	0% (0/408)
localization	0% (0/2)	0% (0/8)	0% (0/30)
message	0% (0/6)	0% (0/62)	0% (0/102)
tuple	0% (0/8)	0% (0/38)	0% (0/60)
BufferedRequestBody	0% (0/1)	0% (0/5)	0% (0/21)
CacheConsumer	100% (0/0)	100% (0/0)	100% (0/0)
ChainedClosableIterator	0% (0/1)	0% (0/9)	0% (0/47)
Checks	100% (1/1)	6% (2/33)	3% (3/95)
ClassWalker	0% (0/2)	0% (0/9)	0% (0/23)
ContextRunnable	0% (0/1)	0% (0/4)	0% (0/14)
EncodingUtil	0% (0/1)	0% (0/6)	0% (0/23)
EntityString	0% (0/1)	0% (0/7)	0% (0/37)
FutureUtil	0% (0/1)	0% (0/4)	0% (0/12)
Helpers	100% (1/1)	48% (12/25)	57% (64/112)
IOUtil	0% (0/1)	0% (0/16)	0% (0/77)
JDALogger	0% (0/2)	0% (0/6)	0% (0/30)
PermissionUtil	0% (0/2)	0% (0/21)	0% (0/170)
ShutdownReason	0% (0/1)	0% (0/4)	0% (0/7)

JsonTest:



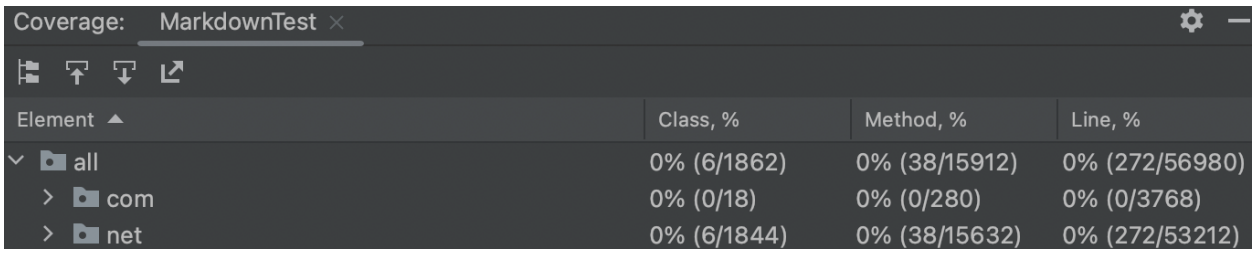
Coverage: JsonTest

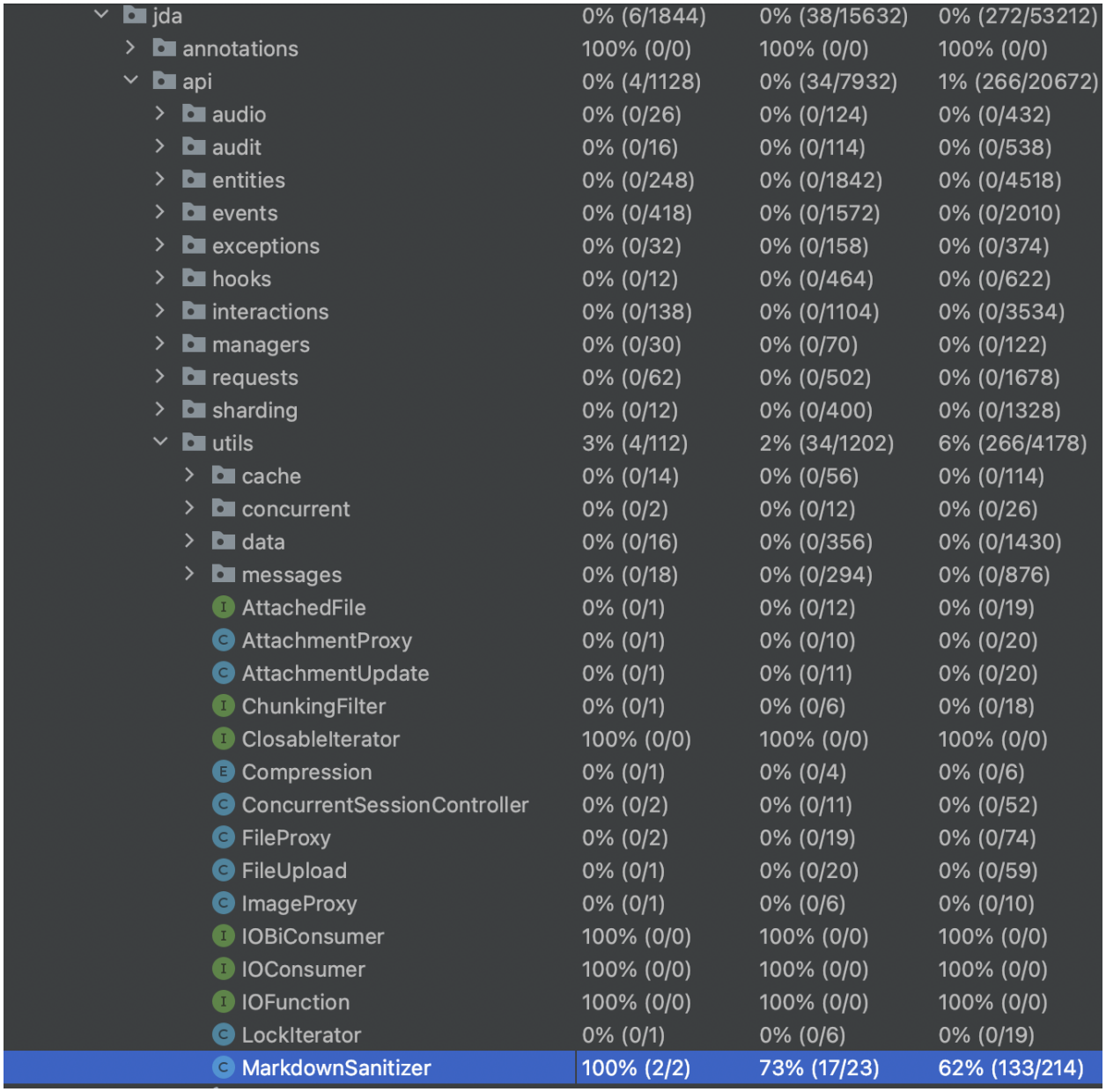
Element	Class, %	Method, %	Line, %
all	0% (2/1862)	0% (20/15912)	0% (56/56980)
com	0% (0/18)	0% (0/280)	0% (0/3768)
net	0% (2/1844)	0% (20/15632)	0% (56/53212)



The jsonTest mainly tests api/utils/data, and the DataObject has the highest coverage.

MarkDownTest:



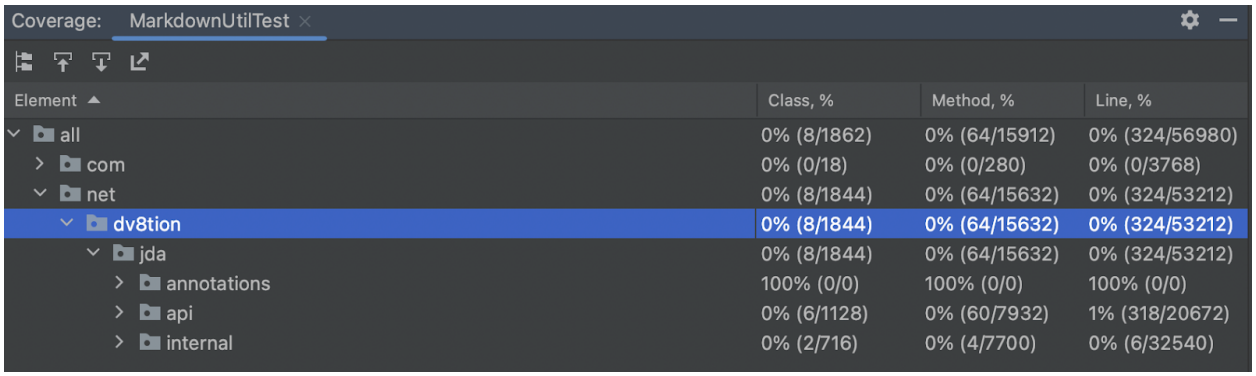


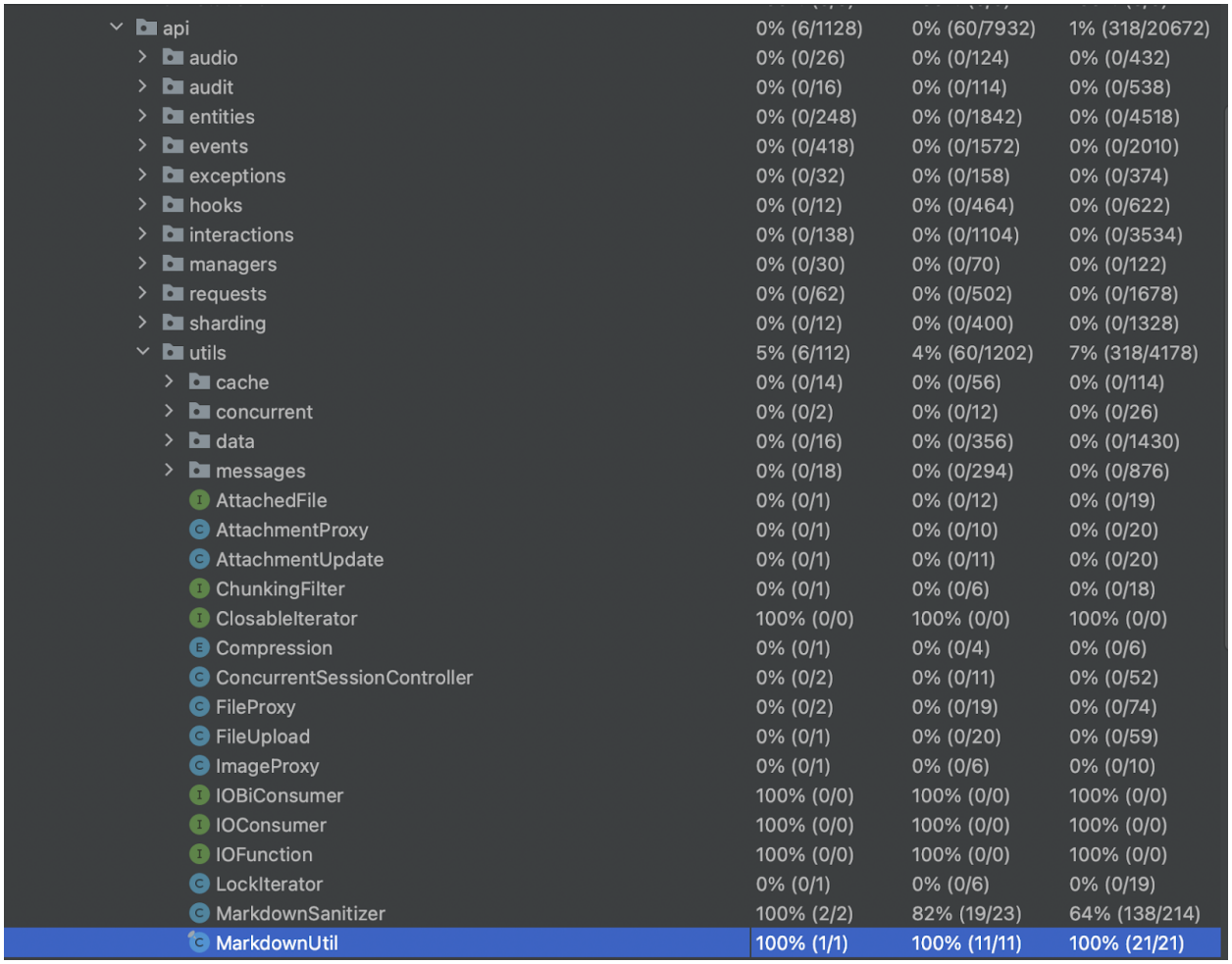
jda	0% (6/1844)	0% (38/15632)	0% (272/53212)
annotations	100% (0/0)	100% (0/0)	100% (0/0)
api	0% (4/1128)	0% (34/7932)	1% (266/20672)
audio	0% (0/26)	0% (0/124)	0% (0/432)
audit	0% (0/16)	0% (0/114)	0% (0/538)
entities	0% (0/248)	0% (0/1842)	0% (0/4518)
events	0% (0/418)	0% (0/1572)	0% (0/2010)
exceptions	0% (0/32)	0% (0/158)	0% (0/374)
hooks	0% (0/12)	0% (0/464)	0% (0/622)
interactions	0% (0/138)	0% (0/1104)	0% (0/3534)
managers	0% (0/30)	0% (0/70)	0% (0/122)
requests	0% (0/62)	0% (0/502)	0% (0/1678)
sharding	0% (0/12)	0% (0/400)	0% (0/1328)
utils	3% (4/112)	2% (34/1202)	6% (266/4178)
cache	0% (0/14)	0% (0/56)	0% (0/114)
concurrent	0% (0/2)	0% (0/12)	0% (0/26)
data	0% (0/16)	0% (0/356)	0% (0/1430)
messages	0% (0/18)	0% (0/294)	0% (0/876)
AttachedFile	0% (0/1)	0% (0/12)	0% (0/19)
AttachmentProxy	0% (0/1)	0% (0/10)	0% (0/20)
AttachmentUpdate	0% (0/1)	0% (0/11)	0% (0/20)
ChunkingFilter	0% (0/1)	0% (0/6)	0% (0/18)
ClosableIterator	100% (0/0)	100% (0/0)	100% (0/0)
Compression	0% (0/1)	0% (0/4)	0% (0/6)
ConcurrentSessionController	0% (0/2)	0% (0/11)	0% (0/52)
FileProxy	0% (0/2)	0% (0/19)	0% (0/74)
FileUpload	0% (0/1)	0% (0/20)	0% (0/59)
ImageProxy	0% (0/1)	0% (0/6)	0% (0/10)
IOBiConsumer	100% (0/0)	100% (0/0)	100% (0/0)
IOConsumer	100% (0/0)	100% (0/0)	100% (0/0)
IOFunction	100% (0/0)	100% (0/0)	100% (0/0)
LockIterator	0% (0/1)	0% (0/6)	0% (0/19)
MarkdownSanitizer	100% (2/2)	73% (17/23)	62% (133/214)

MarkDownTest tests api/utils/MarkdownSanitizer class with 100% class coverage, 73% method coverage and 62% lines coverage.

MarkDownUtilTest:

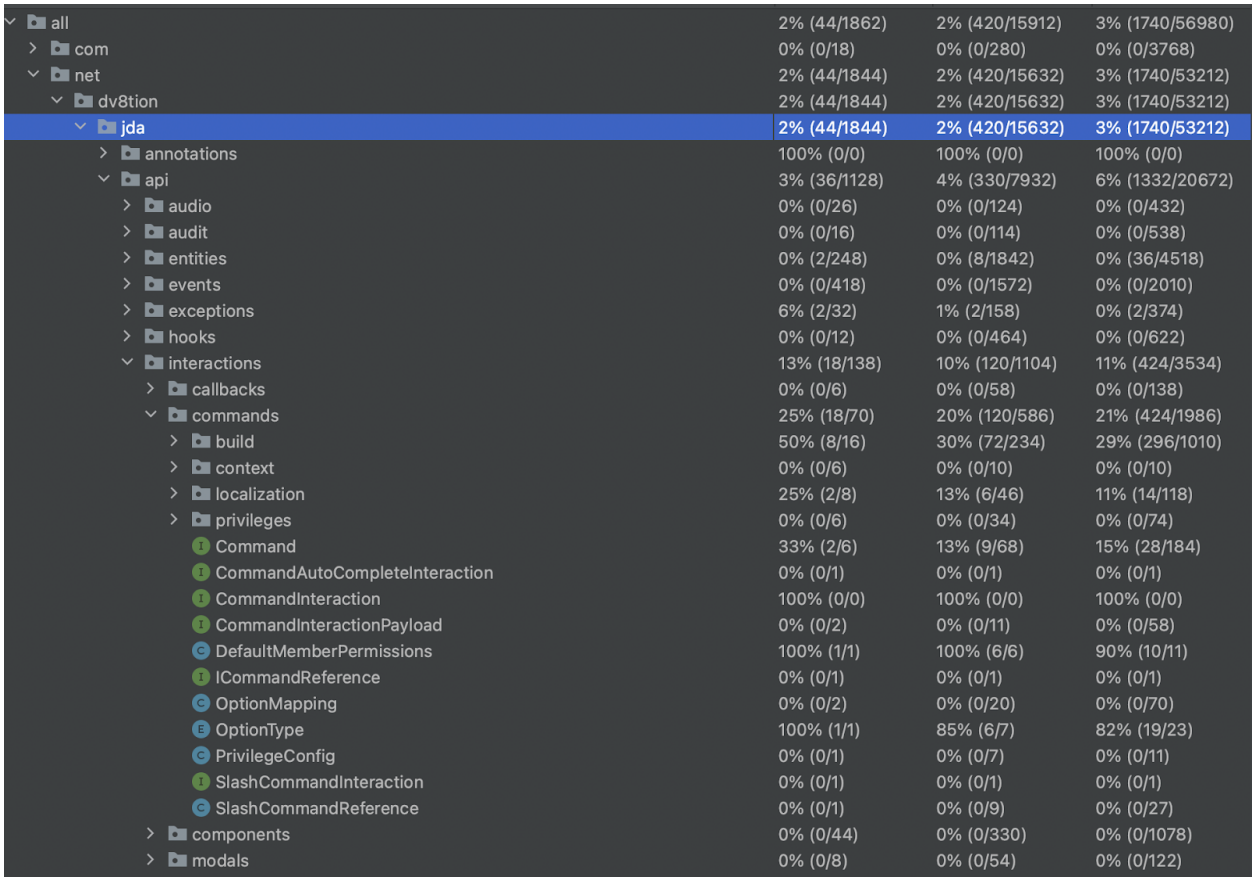
Overall coverage:





Like the previous test file, this one also focuses on MarkdownSanitizer. In addition, it covers all the class, lines, and methods in MarkdownUtil class.

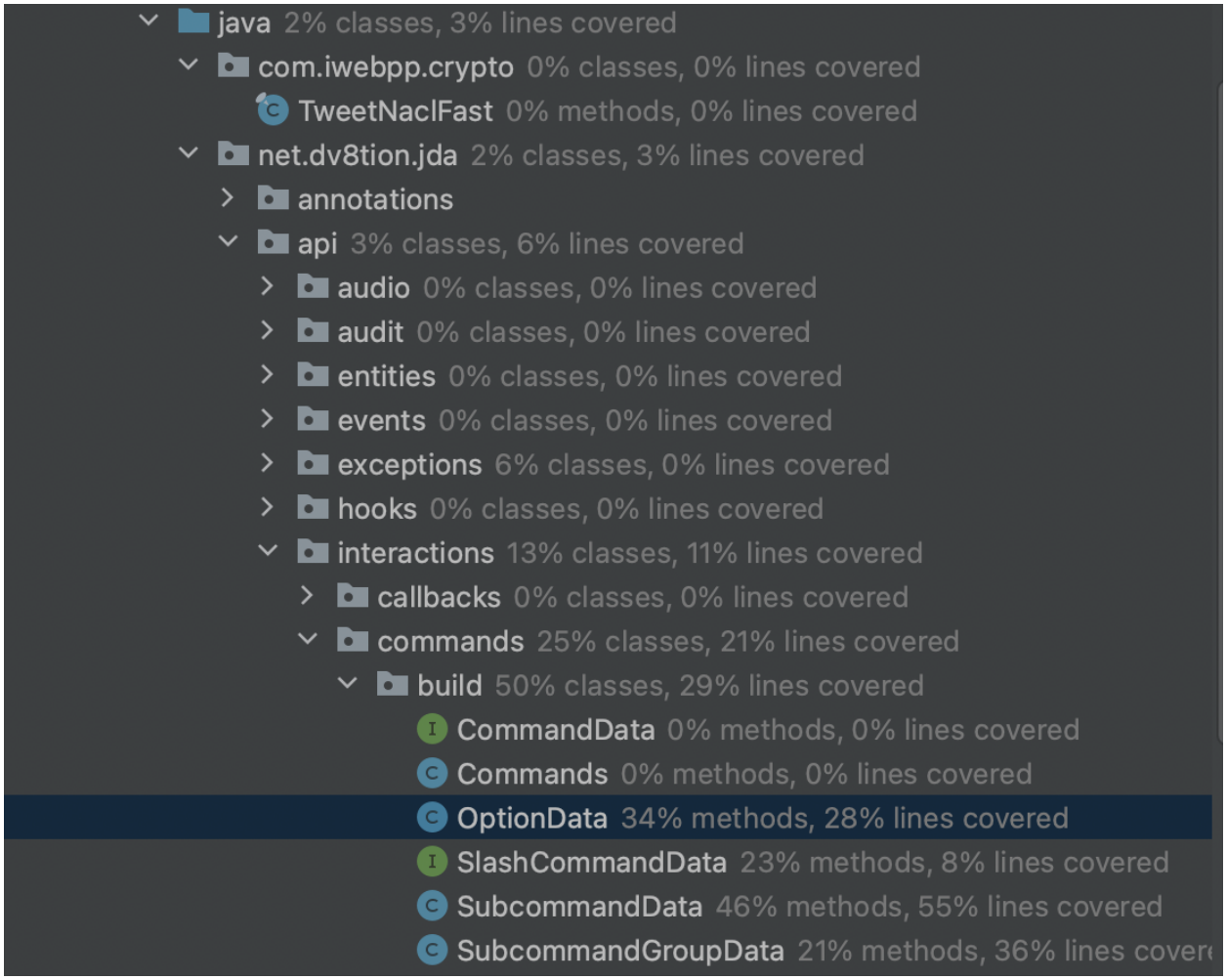
The total coverage of all existing test files:



all	2% (44/1862)	2% (420/15912)	3% (1740/56980)
com	0% (0/18)	0% (0/280)	0% (0/3768)
net	2% (44/1844)	2% (420/15632)	3% (1740/53212)
dv8tion	2% (44/1844)	2% (420/15632)	3% (1740/53212)
jda	2% (44/1844)	2% (420/15632)	3% (1740/53212)
annotations	100% (0/0)	100% (0/0)	100% (0/0)
api	3% (36/1128)	4% (330/7932)	6% (1332/20672)
audio	0% (0/26)	0% (0/124)	0% (0/432)
audit	0% (0/16)	0% (0/114)	0% (0/538)
entities	0% (2/248)	0% (8/1842)	0% (36/4518)
events	0% (0/418)	0% (0/1572)	0% (0/2010)
exceptions	6% (2/32)	1% (2/158)	0% (2/374)
hooks	0% (0/12)	0% (0/464)	0% (0/622)
interactions	13% (18/138)	10% (120/1104)	11% (424/3534)
callbacks	0% (0/6)	0% (0/58)	0% (0/138)
commands	25% (18/70)	20% (120/586)	21% (424/1986)
build	50% (8/16)	30% (72/234)	29% (296/1010)
context	0% (0/6)	0% (0/10)	0% (0/10)
localization	25% (2/8)	13% (6/46)	11% (14/118)
privileges	0% (0/6)	0% (0/34)	0% (0/74)
Command	33% (2/6)	13% (9/68)	15% (28/184)
CommandAutoCompleteInteraction	0% (0/1)	0% (0/1)	0% (0/1)
CommandInteraction	100% (0/0)	100% (0/0)	100% (0/0)
CommandInteractionPayload	0% (0/2)	0% (0/11)	0% (0/58)
DefaultMemberPermissions	100% (1/1)	100% (6/6)	90% (10/11)
ICommandReference	0% (0/1)	0% (0/1)	0% (0/1)
OptionMapping	0% (0/2)	0% (0/20)	0% (0/70)
OptionType	100% (1/1)	85% (6/7)	82% (19/23)
PrivilegeConfig	0% (0/1)	0% (0/7)	0% (0/11)
SlashCommandInteraction	0% (0/1)	0% (0/1)	0% (0/1)
SlashCommandReference	0% (0/1)	0% (0/9)	0% (0/27)
components	0% (0/44)	0% (0/330)	0% (0/1078)
modals	0% (0/8)	0% (0/54)	0% (0/122)

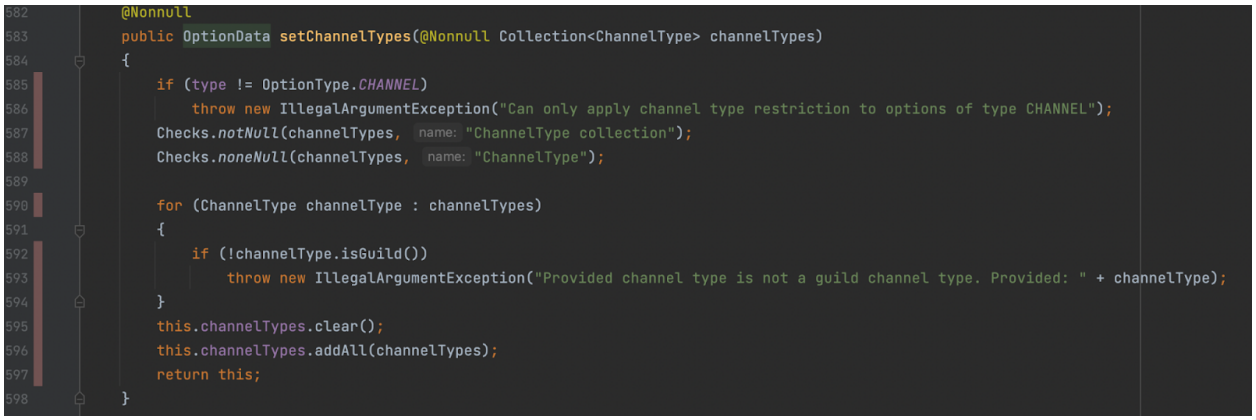
Documentation of some uncovered part of existing test files:

Take api/interactions/commands/OptionData form example:

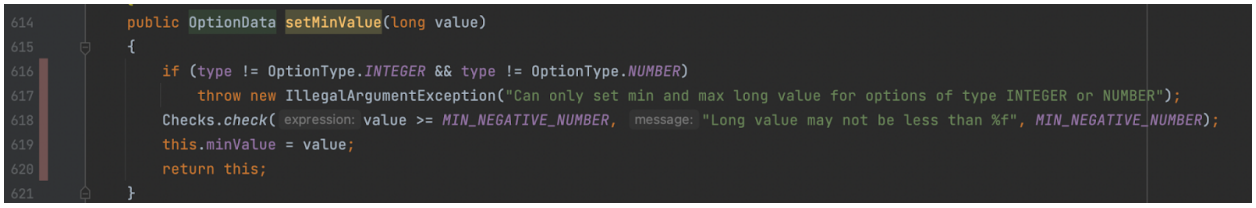


The coverage of OptionData is only 34% in method, 28% in lines.

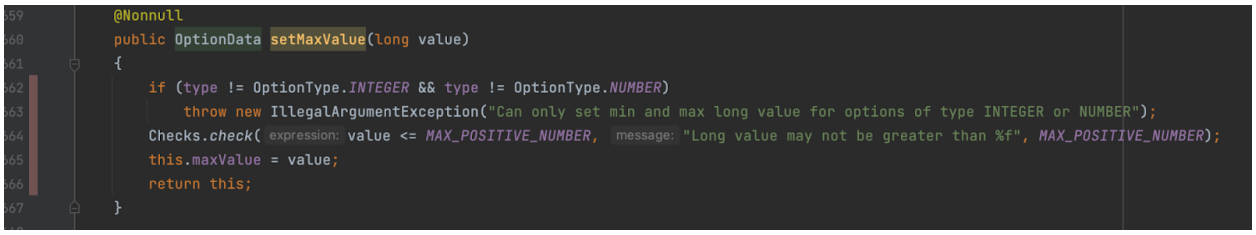
Here are some of the examples for the fully uncovered code:



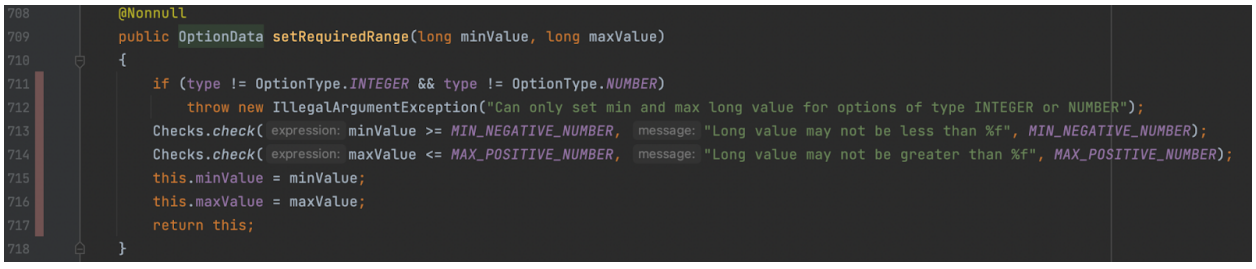
```
@Nonnull
public OptionData setChannelTypes(@Nonnull Collection<ChannelType> channelTypes)
{
    if (type != OptionType.CHANNEL)
        throw new IllegalArgumentException("Can only apply channel type restriction to options of type CHANNEL");
    Checks.notNull(channelTypes, name: "ChannelType collection");
    Checks.noneNull(channelTypes, name: "ChannelType");

    for (ChannelType channelType : channelTypes)
    {
        if (!channelType.isGuild())
            throw new IllegalArgumentException("Provided channel type is not a guild channel type. Provided: " + channelType);
    }
    this.channelTypes.clear();
    this.channelTypes.addAll(channelTypes);
    return this;
}
```



```
public OptionData setMinValue(long value)
{
    if (type != OptionType.INTEGER && type != OptionType.NUMBER)
        throw new IllegalArgumentException("Can only set min and max long value for options of type INTEGER or NUMBER");
    Checks.check( expression: value >= MIN_NEGATIVE_NUMBER,  message: "Long value may not be less than %f", MIN_NEGATIVE_NUMBER);
    this.minValue = value;
    return this;
}
```



```
@Nonnull
public OptionData setMaxValue(long value)
{
    if (type != OptionType.INTEGER && type != OptionType.NUMBER)
        throw new IllegalArgumentException("Can only set min and max long value for options of type INTEGER or NUMBER");
    Checks.check( expression: value <= MAX_POSITIVE_NUMBER,  message: "Long value may not be greater than %f", MAX_POSITIVE_NUMBER);
    this.maxValue = value;
    return this;
}
```



```
@Nonnull
public OptionData setRequiredRange(long minValue, long maxValue)
{
    if (type != OptionType.INTEGER && type != OptionType.NUMBER)
        throw new IllegalArgumentException("Can only set min and max long value for options of type INTEGER or NUMBER");
    Checks.check( expression: minValue >= MIN_NEGATIVE_NUMBER,  message: "Long value may not be less than %f", MIN_NEGATIVE_NUMBER);
    Checks.check( expression: maxValue <= MAX_POSITIVE_NUMBER,  message: "Long value may not be greater than %f", MAX_POSITIVE_NUMBER);
    this.minValue = minValue;
    this.maxValue = maxValue;
    return this;
}
```

```java
public OptionData addChoice(@Nonnull String name, double value)
{
    Checks.notEmpty(name, name: "Name");
    Checks.notLonger(name, MAX_CHOICE_NAME_LENGTH, name: "Name");
    Checks.check( expression: value >= MIN_NEGATIVE_NUMBER, message: "Double value may not be less than %f", MIN_NEGATIVE_NUMBER);
    Checks.check( expression: value <= MAX_POSITIVE_NUMBER, message: "Double value may not be greater than %f", MAX_POSITIVE_NUMBER);
    Checks.check( expression: choices.size() < MAX_CHOICES, message: "Cannot have more than 25 choices for an option!");
    if (isAutoComplete)
        throw new IllegalStateException("Cannot add choices to auto-complete options");
    if (type != OptionType.NUMBER)
        throw new IllegalArgumentException("Cannot add double choice for OptionType." + type);
    choices.add(new Command.Choice(name, value));
    return this;
}
```

```java
@Nonnull
public OptionData addChoices(@Nonnull Collection<? extends Command.Choice> choices)
{
    Checks.notNull(choices, name: "Choices");
    if (choices.size() == 0)
        return this;
    if (this.choices == null || !type.canSupportChoices())
        throw new IllegalStateException("Cannot add choices for an option of type " + type);
    Checks.noneNull(choices, name: "Choices");
    if (isAutoComplete)
        throw new IllegalStateException("Cannot add choices to auto-complete options");
    Checks.check( expression: choices.size() + this.choices.size() <= MAX_CHOICES, message: "Cannot have more than 25 choices for one option!");
    this.choices.addAll(choices);
    return this;
}
```

```java
@Nonnull                                                                                                    ⚠ 22  ✓ 5
public static OptionData fromData(@Nonnull DataObject json)
{
    String name = json.getString( key: "name");
    String description = json.getString( key: "description");
    OptionType type = OptionType.fromKey(json.getInt( key: "type"));
    OptionData option = new OptionData(type, name, description);
    option.setRequired(json.getBoolean( key: "required"));
    option.setAutoComplete(json.getBoolean( key: "autocomplete"));
    if (type == OptionType.INTEGER || type == OptionType.NUMBER)
    {
        if (!json.isNull( key: "min_value"))
        {
            if (json.isType( key: "min_value", DataType.INT))
                option.setMinValue(json.getLong( key: "min_value"));
            else if (json.isType( key: "min_value", DataType.FLOAT))
                option.setMinValue(json.getDouble( key: "min_value"));
        }
        if (!json.isNull( key: "max_value"))
        {
            if (json.isType( key: "max_value", DataType.INT))
                option.setMaxValue(json.getLong( key: "max_value"));
            else if (json.isType( key: "max_value", DataType.FLOAT))
                option.setMaxValue(json.getDouble( key: "max_value"));
        }
    }
    if (type == OptionType.CHANNEL)
    {
        option.setChannelTypes(json.optArray( key: "channel_types") Optional<DataArray>
                .map(it -> it.stream(DataArray::getInt).map(ChannelType::fromId).collect(Collectors.toSet())) Optional<Set<ChannelType>>
                .orElse(Collections.emptySet()));
    }
```

```
        if (type == OptionType.STRING)
        {
            if (!json.isNull( key: "min_length"))
                option.setMinLength(json.getInt( key: "min_length"));
            if (!json.isNull( key: "max_length"))
                option.setMaxLength(json.getInt( key: "max_length"));
        }
        json.optArray( key: "choices").ifPresent(choices1 ->
                option.addChoices(choices1.stream(DataArray::getObject) Stream<DataObject>
                        .map(Command.Choice::new) Stream<Choice>
                        .collect(Collectors.toList())
                )
        );
        option.setNameLocalizations(LocalizationUtils.mapFromProperty(json, localizationProperty: "name_localizations"));
        option.setDescriptionLocalizations(LocalizationUtils.mapFromProperty(json, localizationProperty: "description_localizations"));
        return option;
```

Increasing The Testing Coverage

Many parts of the OptionData class are uncovered by the current testing suite. We have proceeded to increase the coverage through white box testing. The results can be seen below.

Before:

build	50% (8/16)	30% (72/234)	29% (296/1010)
CommandData	0% (0/1)	0% (0/2)	0% (0/23)
Commands	0% (0/1)	0% (0/6)	0% (0/10)
OptionData	50% (1/2)	34% (16/47)	28% (72/255)
SlashCommandData	50% (1/2)	23% (3/13)	8% (5/62)
SubcommandData	100% (1/1)	46% (12/26)	55% (44/80)
SubcommandGroupData	100% (1/1)	21% (5/23)	36% (27/75)

After:

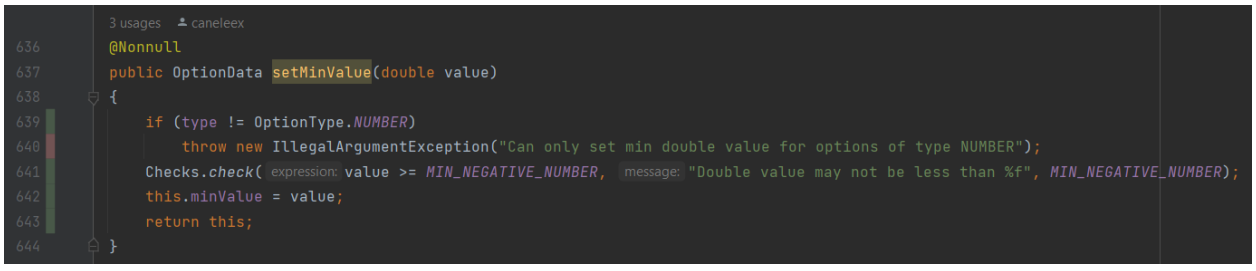
build	62% (10/16)	49% (116/234)	49% (498/1010)
CommandData	0% (0/1)	0% (0/2)	0% (0/23)
Commands	0% (0/1)	0% (0/6)	0% (0/10)
OptionData	100% (2/2)	80% (38/47)	67% (173/255)
SlashCommandData	50% (1/2)	23% (3/13)	8% (5/62)
SubcommandData	100% (1/1)	46% (12/26)	55% (44/80)
SubcommandGroupData	100% (1/1)	21% (5/23)	36% (27/75)

The coverage for OptionsData was only 28% of the lines of the class. White box testing allowed us to increase this coverage to 67%, over twice the original coverage. By expanding the test coverage for OptionsData we ensure that options, similar to parameters, for Discord user commands work properly. The testing asserts that these features work:

- Setting minimum and maximum values and lengths for options, as well as getting these properties.
- Getting option descriptions and the option autocomplete properties.
- Adding multiple choices to an option at once.
- Parsing serialization of an option back into an OptionData instance.
- Converting a provided Command.Option into an OptionData instance.

The code for this testing can be found in this repository, added as WhiteBoxTesting.java in src/test/java/: https://github.com/dalo390/JDA

Here you can see where some of the previously uncovered code is now covered as shown through the IDE coverage tool:



```java
3 usages  caneleex
@Nonnull
public OptionData setMaxValue(long value)
{
    if (type != OptionType.INTEGER && type != OptionType.NUMBER)
        throw new IllegalArgumentException("Can only set min and max long value for options of type INTEGER or NUMBER");
    Checks.check( expression: value <= MAX_POSITIVE_NUMBER,  message: "Long value may not be greater than %f", MAX_POSITIVE_NUMBER);
    this.maxValue = value;
    return this;
}
```

```java
@Nonnull
public OptionData setRequiredRange(long minValue, long maxValue)
{
    if (type != OptionType.INTEGER && type != OptionType.NUMBER)
        throw new IllegalArgumentException("Can only set min and max long value for options of type INTEGER or NUMBER");
    Checks.check( expression: minValue >= MIN_NEGATIVE_NUMBER,  message: "Long value may not be less than %f", MIN_NEGATIVE_NUMBER);
    Checks.check( expression: maxValue <= MAX_POSITIVE_NUMBER,  message: "Long value may not be greater than %f", MAX_POSITIVE_NUMBER);
    this.minValue = minValue;
    this.maxValue = maxValue;
    return this;
}
```

```java
@Nonnull
public OptionData addChoice(@Nonnull String name, double value)
{
    Checks.notEmpty(name,  name: "Name");
    Checks.notLonger(name, MAX_CHOICE_NAME_LENGTH,  name: "Name");
    Checks.check( expression: value >= MIN_NEGATIVE_NUMBER,  message: "Double value may not be less than %f", MIN_NEGATIVE_NUMBER);
    Checks.check( expression: value <= MAX_POSITIVE_NUMBER,  message: "Double value may not be greater than %f", MAX_POSITIVE_NUMBER);
    Checks.check( expression: choices.size() < MAX_CHOICES,  message: "Cannot have more than 25 choices for an option!");
    if (isAutoComplete)
        throw new IllegalStateException("Cannot add choices to auto-complete options");
    if (type != OptionType.NUMBER)
        throw new IllegalArgumentException("Cannot add double choice for OptionType." + type);
    choices.add(new Command.Choice(name, value));
    return this;
}
```

```java
@Nonnull
public OptionData addChoices(@Nonnull Collection<? extends Command.Choice> choices)
{
    Checks.notNull(choices,  name: "Choices");
    if (choices.size() == 0)
        return this;
    if (this.choices == null || !type.canSupportChoices())
        throw new IllegalStateException("Cannot add choices for an option of type " + type);
    Checks.noneNull(choices,  name: "Choices");
    if (isAutoComplete)
        throw new IllegalStateException("Cannot add choices to auto-complete options");
    Checks.check( expression: choices.size() + this.choices.size() <= MAX_CHOICES,  message: "Cannot have more than 25 choices for one option!");
    this.choices.addAll(choices);
    return this;
}
```

Continuous Integration With Java

Discord API

What is Continuous Integration and Why is it Important?

Development becomes more complicated the longer one goes without testing. New problems inevitably arise when development and merging takes place. The cost for fixing bugs grows exponentially as a project goes further down its development. Therefore, it is important and more efficient to integrate and test components for all small changes to a system, rather than waiting until near completion of separate components before integrating and testing.

This practice of commonly merging and testing components together after small changes is called continuous integration. Through continuous integration bugs are caught when they are smaller, easier to diagnose, and easier to fix. Furthermore, prediction for the development timeline is more easily maintained through this methodology. Developers have constant feedback for their code, and users will also be able to have access to more features as they are constantly being built and deployed.

Continuous integration involves maintaining a single source code repository. Every day commits should be built on the main line. Automatic building and testing keeps this process efficient, and makes it easy for someone to grab the latest executable. The latest version should always build and pass the tests. There is also a goal of high visibility with continuous integration. Users should be able to discern what process is currently taking place, whether it is building or testing, what is passing or failing, and the identity of user actions. There are many tools for users to follow the continuous integration strategy- this includes but is not limited to GitHub Actions, TravisCI, CircleCI, Jenkins, etc..

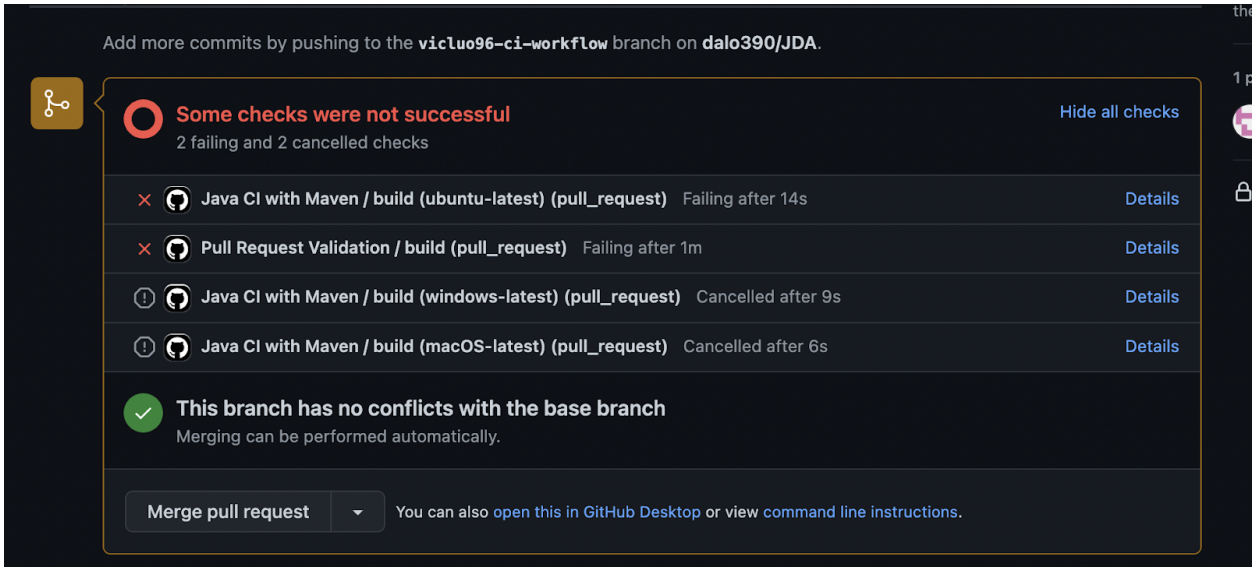
CI system testing:

We chose Github Actions as the continuous integration tool. At first, we choose Java with Maven as a new workflow, then create a ci-maven.yml configuration file shown below:

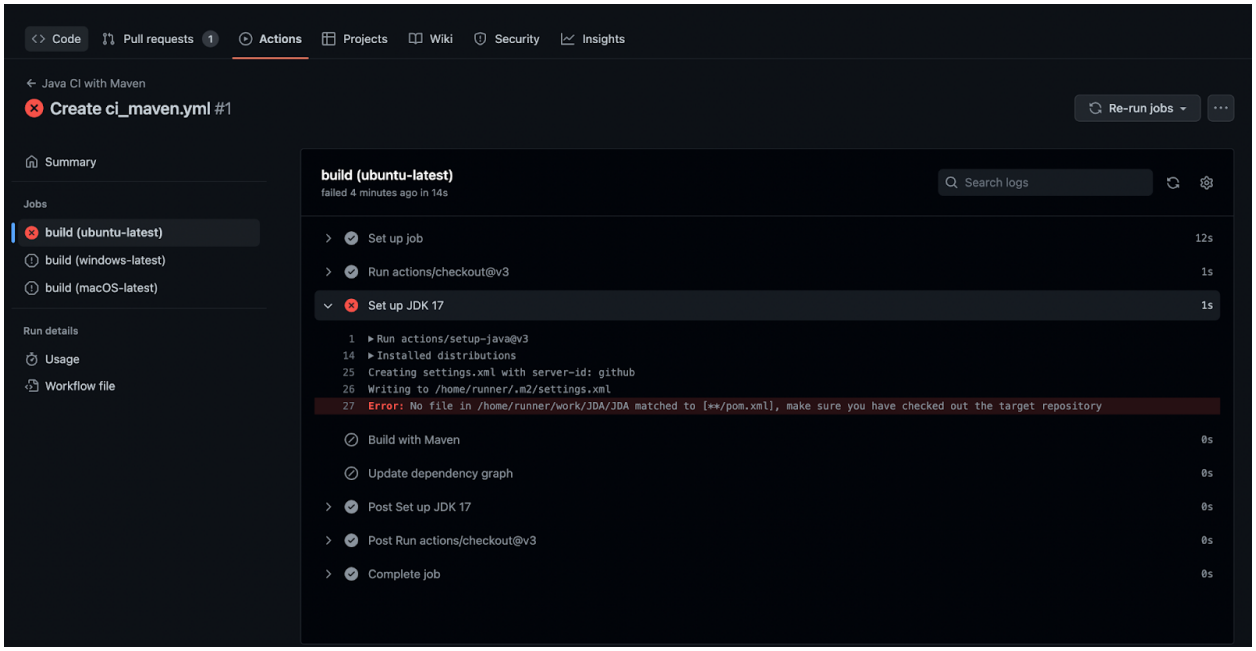
```yaml
# This workflow will build a Java project with Maven, and cache/restore any dependencies to improve the workflow execution time
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven

# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.

name: Java CI with Maven

on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]

jobs:
  build:

    runs-on: ${{matrix.os}}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macOS-latest]

    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file pom.xml

    # Optional: Uploads the full dependency graph to GitHub to improve the quality of Dependabot alerts this repository can receive
    - name: Update dependency graph
      uses: advanced-security/maven-dependency-submission-action@571e99aab1055c2e71a1e2309b9691de18d6b7d6
```

Then we created a new branch for this workflow called [vicluo96-ci-workflow] and made a pull request. Since making a PR is considered a change, Github Actions builds and runs the tests automatically. The Github Actions uses the environment, actions, and the commands specified in the configuration file to build and run the projects. Specifically, it will create a Github server for this workflow, set up the Ubuntu, Windows, MacOS environments respectively, download Java

17, and use actions specified in Github Actions source code to build and run the Java project with Maven. The result is shown below:

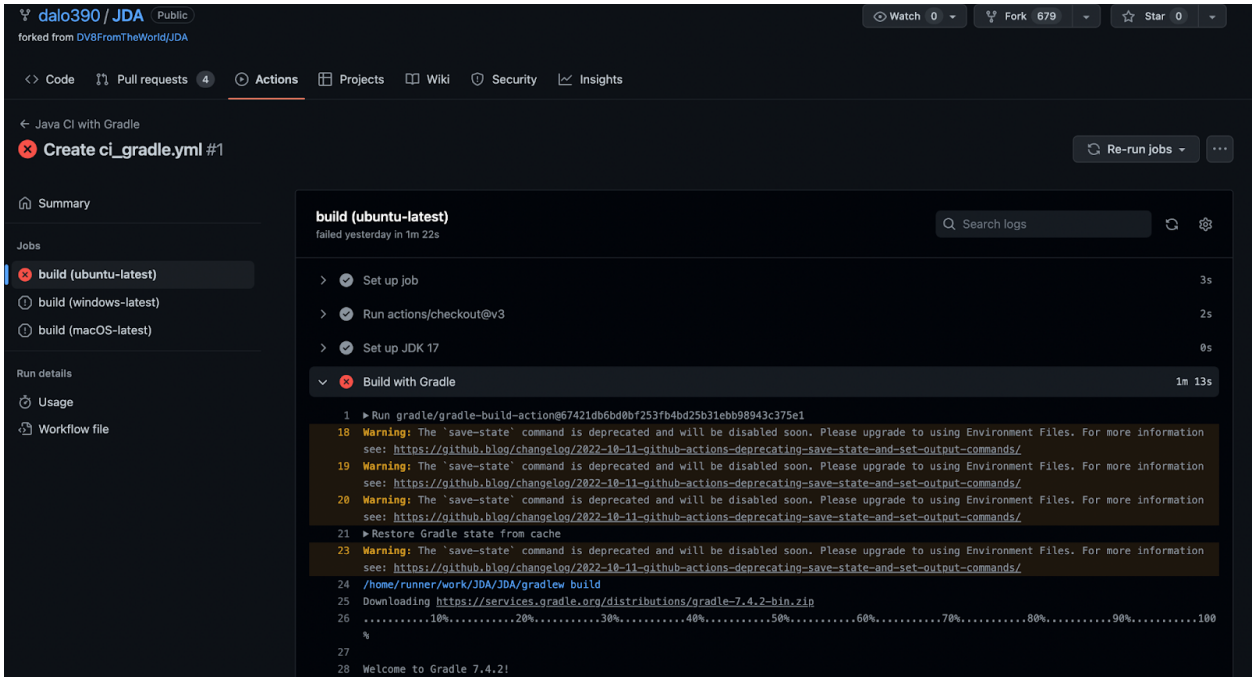


There are some problems in the build process, so we clicked the details to see what happened:



The problem is: there is no pom.xml file for the Maven project in source code. On the other hand, we saw there are Gradle files in the source code directory, so we assume the project should use Gradle as default. Then, we create another workflow branch called [vicluo96-ci-gradle-workflow](), used Java with Gradle this time, then generate new JDA/.github/workflows/ci_gradle.yml

However, this time there are issues in the building process:



we found the problems lies in the Slasher.java testcase

```
SlasherTest > testSlashSay() FAILED
    org.opentest4j.AssertionFailedError at SlasherTest.java:52

SlasherTest > testSlashLeave() FAILED
    org.opentest4j.AssertionFailedError at SlasherTest.java:76

108 tests completed, 2 failed

> Task :test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///home/runner/work/JDA/JDA/build/reports/tests/test/index.html
```

Then we fixed the bugs of the two aforementioned functions,

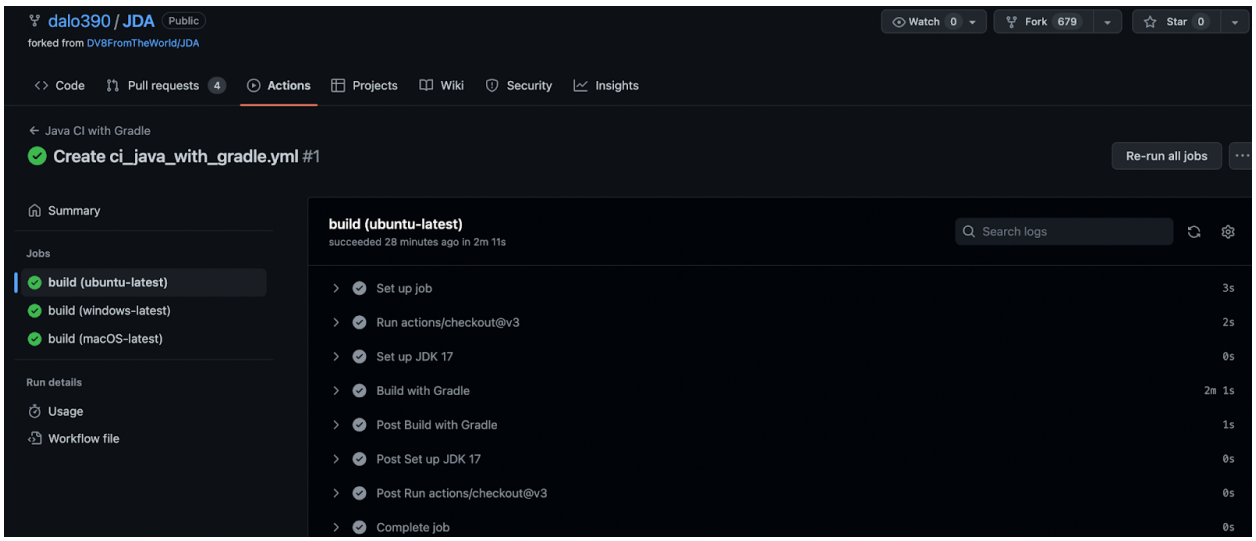
Before:

```
@Test
public void testSlashSay(){
    CommandData command = new CommandDataImpl( name: "say",  description: "make the bot say what you want")
            .addOptions(new OptionData(OptionType.STRING,  name: "content",  description: "What the bot should say",
    DataObject data = command.toData();
    Assertions.assertEquals( expected: "say", data.getString( key: "name"));
    Assertions.assertEquals( expected: "make the bot say what you want", data.getString( key: "description"));

    DataArray options = data.getArray( key: "options");
    DataObject option = options.getObject( index: 0);
    option = options.getObject( index: 0);
    Assertions.assertTrue(option.getBoolean( key: "required"));
    Assertions.assertEquals( expected: "content", option.getString( key: "name"));
    Assertions.assertEquals( expected: "the saying content back itself", option.getString( key: "description"));

}


no usages  Kevin-the-man
@Test
public void testSlashLeave(){
    CommandData command = new CommandDataImpl( name: "prune",  description: "prune messages from the channel")
    .setGuildOnly(true)
    .setDefaultPermissions(DefaultMemberPermissions.enabledFor(Permission.MESSAGE_MANAGE))
    .addOptions(new OptionData(OptionType.INTEGER,  name: "amount",  description: "How many messages to prune (Default 100
    DataObject data = command.toData();
    Assertions.assertEquals( expected: "prune", data.getString( key: "name"));
    Assertions.assertEquals( expected: "prune messages from the channel", data.getString( key: "description"));

    DataArray options = data.getArray( key: "options");
```

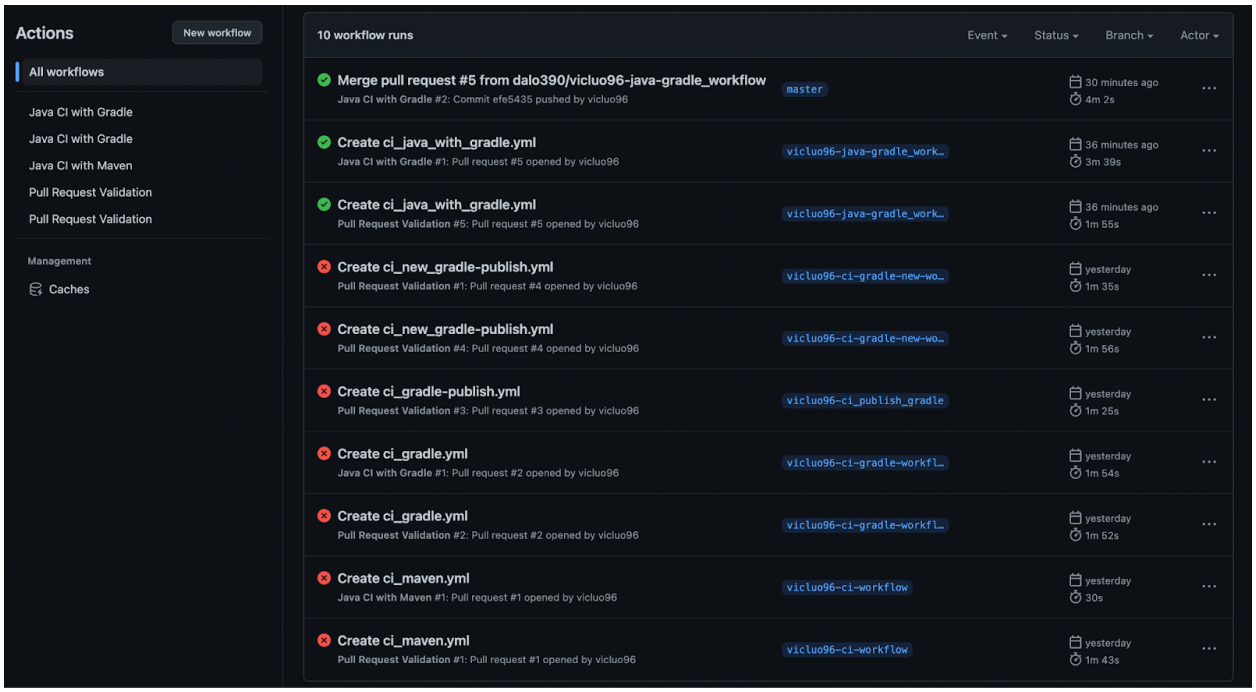
After:

```java
@Test
public void testSlashSay(){
    CommandData command = new CommandDataImpl( name: "say",  description: "make the bot say what you want")
            .addOptions(new OptionData(OptionType.STRING,  name: "content",  description: "What the bot should say",  isRequired: true));
    DataObject data = command.toData();
    Assertions.assertEquals( expected: "say", data.getString( key: "name"));
    Assertions.assertEquals( expected: "make the bot say what you want", data.getString( key: "description"));

    DataArray options = data.getArray( key: "options");
    DataObject option = options.getObject( index: 0);
    option = options.getObject( index: 0);
    Assertions.assertTrue(option.getBoolean( key: "required"));
    Assertions.assertEquals( expected: "content", option.getString( key: "name"));
    Assertions.assertEquals( expected: "What the bot should say", option.getString( key: "description"));

}


no usages  Kevin-the-man +1
@Test
public void testSlashLeave(){
    CommandData command = new CommandDataImpl( name: "prune",  description: "prune messages from the channel")
    .setGuildOnly(true)
    .setDefaultPermissions(DefaultMemberPermissions.enabledFor(Permission.MESSAGE_MANAGE))
    .addOptions(new OptionData(OptionType.INTEGER,  name: "amount",  description: "How many messages to prune (Default 100)",  isRequired: true));
    DataObject data = command.toData();
    Assertions.assertEquals( expected: "prune", data.getString( key: "name"));
    Assertions.assertEquals( expected: "prune messages from the channel", data.getString( key: "description"));

    DataArray options = data.getArray( key: "options");
    DataObject option = options.getObject( index: 0);
    Assertions.assertTrue(option.getBoolean( key: "required"));
    Assertions.assertEquals( expected: "amount", option.getString( key: "name"));
    Assertions.assertEquals( expected: "How many messages to prune (Default 100)", option.getString( key: "description"));
```

Then everything works fine this time; the set up, build, and test processes passed in all of the environments:



Then we merge this branch to the master, the Github Actions runs the test again, no conflicts happens and everything works fine:



To sum up, the continuous integration tool runs every time we commit a change, including a PR and merge in repo. Every time we make a PR, it will automatically run all the tests to check whether there is any problem before merging a branch. This automation saves a lot of effort for developers to maintain and manage the repo, making the error more likely to be found as early as possible.

To see the final version of the configuration file, check:

JDA/.github/workflows/ci_java_with_gradle.yml

In the “Actions” tab of the repo, you can see all of the workflows we have created, including some of which are labeled as bugs in the branch.

Testable Design and Mocking With Java Discord API

What makes a Testable Design?

A testable design in software engineering is a design that facilitates the creation of automated tests for the software system. Such a design ensures that the software components are loosely coupled, that dependencies are clearly defined and can be easily substituted, and that the software has a clear separation of concerns. Some of the characteristics that make a design testable include:

Loose Coupling: Components should be loosely coupled, meaning that they should depend on each other as little as possible, to make it easier to test them separately. Some standard design patterns, such as dependency injection and inversion of control, can make the system more testable by reducing coupling between components and making dependencies more explicit. Take dependency injection for example, instead hardcode in a “new” (when appropriate), allow the object reference to be created outside the method and passed in.

Separation of concerns (SoC): Separating a computer program into distinct sections. Each section addresses a separate concern, a set of information that affects the code of a computer program. A program that embodies SoC well is called a modular program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface. A clear SoC between components ensures that the software system can be tested more easily, with each component focusing on a specific task or functionality.

Avoid complex private methods and static methods: Private methods are not testable, so complex logic in private methods can be a source for bugs that cannot be found by direct testing. On the other hand, static methods operate on the class rather than the object, for functionality that has side effects or that has randomness (anything that we may want to stub out), static makes that difficult or impossible.

Avoid real work in constructor:

Constructors are difficult to bypass because a subclass's constructor always triggers at least one of the superclass's constructors. Make sure that any code that happens in a constructor is not something that we might want to substitute in a test case. If it were moved to a method, it can be overridden. For example, anything more than field assignment in constructors, control flow (conditional or looping logic) in a constructor, and code does complex object graph construction inside a constructor rather than using a factory or builder are warning signs.

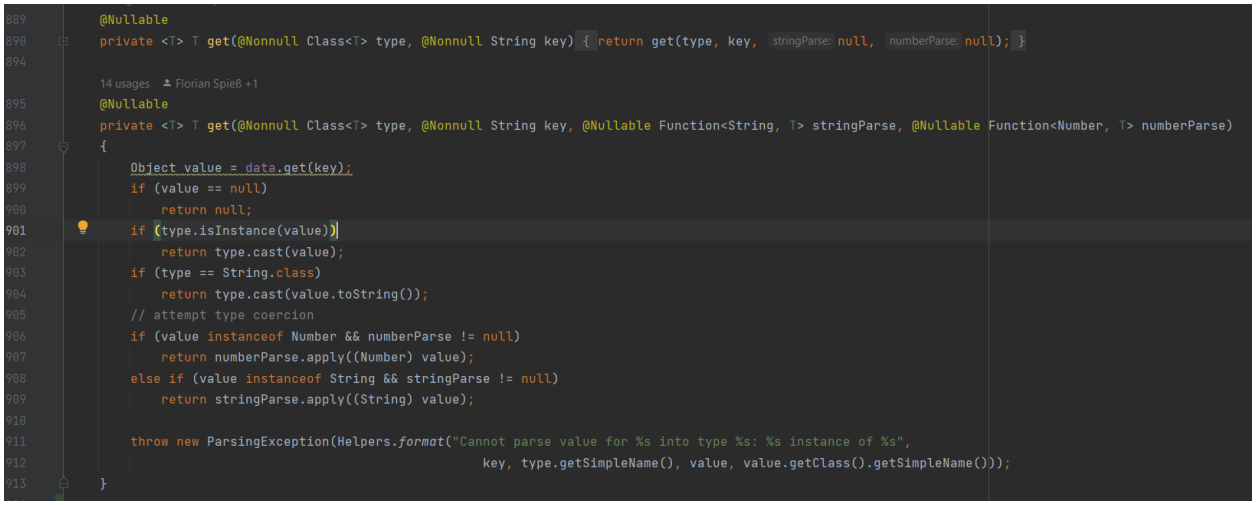
Overall, a testable design promotes software quality and reduces the risk of errors and bugs, making it easier to maintain and improve the software over time.

Code in the Java Discord API With Poor Testable Design

Within the java/net/dv8tion/jda/api/utils/data/DataObject.java class there is a private method called get() with a signature of:

```
private <T> T get(@Nonnull Class<T> type, @Nonnull String key, @Nullable
Function<String, T> stringParse, @Nullable Function<Number, T> numberParse)
```

DataObject represents a map of values that is used in communication with the Discord API. This method returns a value of a key value pair within the DataObject. An aspect of having a highly testable design is having simple private methods, because private methods can not be tested. However, complex private methods can be a source of bugs. This get method in the DataObject class does a series of type checking and type coercion. Here is the code:



```
889     @Nullable
890     private <T> T get(@Nonnull Class<T> type, @Nonnull String key) { return get(type, key,  stringParse: null,  numberParse: null); }
894
        14 usages   Florian Spieß +1
895     @Nullable
896     private <T> T get(@Nonnull Class<T> type, @Nonnull String key, @Nullable Function<String, T> stringParse, @Nullable Function<Number, T> numberParse)
897     {
898         Object value = data.get(key);
899         if (value == null)
900             return null;
901         if (type.isInstance(value))
902             return type.cast(value);
903         if (type == String.class)
904             return type.cast(value.toString());
905         // attempt type coercion
906         if (value instanceof Number && numberParse != null)
907             return numberParse.apply((Number) value);
908         else if (value instanceof String && stringParse != null)
909             return stringParse.apply((String) value);
910
911         throw new ParsingException(Helpers.format("Cannot parse value for %s into type %s: %s instance of %s",
912                                                   key, type.getSimpleName(), value, value.getClass().getSimpleName()));
913     }
```

As one can see, the first get method is simple, but the second get method has multiple lines of code that can be better implemented for modularization and thorough direct testing.

How We Fixed This code

To improve this code for better testing, we created two new methods. Both methods are public so they will be able to be tested directly and independently. The getNew() method is essentially the first half of the get method. We then modularized the type casting by creating a

getNewCoercion() helper method that would handle the type coercion with parsing functions. These two new methods are dummy methods that will not break existing code.

This approach allows us to test the functionality of this function independently with different argument values, and to check if we are getting the appropriate results from different types of the object value.

Here is the implementation of this new testable design:

```
915     @Nullable
916     public <T> T getNew(@Nonnull Class<T> type, @Nonnull String key, @Nullable Function<String, T> stringParse, @Nullable Function<Number, T> numberParse)
917     {
918         Object value = data.get(key);
919         if (value == null)
920             return null;
921         if (type.isInstance(value))
922             return type.cast(value);
923         if (type == String.class)
924             return type.cast(value.toString());
925         // attempt type coercion
926         return getNewCoercion(type, key, value, stringParse, numberParse);
927     }
928
        1 usage  new *
929     @Nullable
930     public <T> T getNewCoercion(@Nonnull Class<T> type, @Nonnull String key, @Nonnull Object value, @Nullable Function<String, T> stringParse, @Nullable Function<Number, T> numbe
931     {
932         if (value instanceof Number && numberParse != null)
933             return numberParse.apply((Number) value);
934         else if (value instanceof String && stringParse != null)
935             return stringParse.apply((String) value);
936
937         throw new ParsingException(Helpers.format("Cannot parse value for %s into type %s: %s instance of %s",
938                 key, type.getSimpleName(), value, value.getClass().getSimpleName()));
939     }
940 }
```

Testing The New Testable Code

We created unit tests in a file named TestableDesignTest.java. These unit tests ensure that the correct output is being given by both functions, and that exceptions are being thrown when appropriate. Here is the code:

```java
@Test
public void testGetNewNormal()
{
    CommandData command = new CommandDataImpl( name: "ban", description: "Ban a user from this server")
            .setGuildOnly(true) CommandDataImpl
            .setDefaultPermissions(DefaultMemberPermissions.enabledFor(Permission.BAN_MEMBERS))
            .addOption(OptionType.USER, name: "user", description: "The user to ban", required: true) SlashCommandData
            .addOption(OptionType.STRING, name: "reason", description: "The ban reason");

    DataObject data = command.toData();
    DataArray options = data.getArray( key: "options");

    //test if getting string value works
    DataObject option = options.getObject( index: 0);
    Assertions.assertEquals( expected: "user", option.getNew(String.class, key: "name", stringParse: null, numberParse: null));
    Assertions.assertNull(option.getNew(String.class, key: "nonexistentKey", stringParse: null, numberParse: null));

    //test if getting integer value works
    OptionData optionTest = new OptionData(OptionType.STRING, name: "example_option", description: "this is an option!");
    optionTest.setMinLength(2);
    DataObject op = optionTest.toData();
    Assertions.assertEquals( expected: 2,op.getNew(Integer.class, key: "min_length", stringParse: null, numberParse: null));

    //test if typecasting to string is working
    Assertions.assertEquals(String.class, op.getNew(String.class, key: "min_length", stringParse: null, numberParse: null).getClass())
}
```

```java
@Test
public void testGetNewCoercionNormal()
{
    OptionData optionTest = new OptionData(OptionType.STRING, name: "example_option", description: "this is an option!");
    optionTest.setMinLength(2);
    DataObject op = optionTest.toData();

    //test that converting to long from int works
    Assertions.assertEquals(Long.class, op.getNewCoercion(Long.class, key: "min_length", value: 2, MiscUtil::parseLong, Number::longValue).getClass());

    //test that converting to double from int works
    Assertions.assertEquals(Double.class, op.getNewCoercion(Double.class, key: "min_length", value: 2, Double::parseDouble, Number::doubleValue).getClass());

    //test that converting to string from int works
    Assertions.assertEquals( expected: "2", op.getNewCoercion(String.class, key: "min_length", value: 2, UnaryOperator.identity(), String::valueOf));
}
```

```java
@Test
public void testGetNewCoercionException()
{
    OptionData optionTest = new OptionData(OptionType.STRING, name: "example_option", description: "this is an option!");
    optionTest.setMinLength(2);
    DataObject op = optionTest.toData();

    Assertions.assertThrows(ParsingException.class, () -> op.getNewCoercion(Integer.class, key: "min_length", value: 2, stringParse: null, numberParse: null));

}
```

Mocking

Mocking is a testing technique that creates a fake object that decides whether a unit test has passed or failed by watching interactions between different objects. The goal of mocking is to determine the behavior of the tested object/method - to see if this method is able to interact with other objects/methods correctly.

Purpose of Mocking

- Significantly improves testing efficiency since we would not rely on external sources
- Mocking makes sure that the test we run are non-repetitive
- Since there aren't requirements for third-party databases or APIs, we can run test cases independently through mocking

Examples of Mocking in the source code

Below is an example of mocking in the source code. This example can be found following this link:https://github.com/dalo390/JDA/blob/master/src/main/java/net/dv8tion/jda/api/EmbedBuilder.java

```
     *
     * @param  color
     *         The {@link java.awt.Color Color} of the embed
     *         or {@code null} to use no color
     *
     * @return the builder after the color has been set
     *
     * @see    #setColor(int)
     */
    @Nonnull
    public EmbedBuilder setColor(@Nullable Color color)
    {
        this.color = color == null ? Role.DEFAULT_COLOR_RAW : color.getRGB();
        return this;
    }

    /**
     * Sets the raw RGB color value for the embed.
     *
     * <p><b><a href="https://raw.githubusercontent.com/DV8FromTheWorld/JDA/assets/a
     *
     * @param  color
     *         The raw rgb value, or {@link Role#DEFAULT_COLOR_RAW} to use no color
     *
     * @return the builder after the color has been set
     *
     * @see    #setColor(java.awt.Color)
     */
    @Nonnull
    public EmbedBuilder setColor(int color)
    {
        this.color = color;
        return this;
    }
```

These two separate methods both trigger the setColor function from the JDA bot. Depending on the response, an exception can be made.

Using a mockito test, we will examine the behavior of the functions and see if their intended results are shown. Sample of the file can be found following this link:

https://github.com/dalo390/JDA/blob/master/src/test/java/MockitoTest.java

```java
public class MockitoTest {
    @Mock
    Color color;
    int col;
    EmbedBuilder mockEmbed;

    @BeforeEach
    public void setup() {
        color = mock(classToMock: Color.class);
        col = (int)Math.random();
        mockEmbed = mock(classToMock: EmbedBuilder.class);
    }

    @Test
    public void mockitoTestColor() throws Exception {
        assertNotNull(color);
        mockEmbed.setColor(color);
        verify(mockEmbed).setColor(color);
    }

    @Test
    public void mockitoTestColor1() throws Exception {
        assertNotNull(color);
        EmbedBuilder realEmbed = new EmbedBuilder();
        assertNotEquals(mockEmbed.setColor(color), realEmbed.setColor(color));
    }


    @Test
    public void mockitoTestColorInt() throws Exception {
        assertNotNull(col);
        mockEmbed.setColor(col);
        verify(mockEmbed).setColor(col);
    }

    @Test
    public void mockitoTestColorInt1() throws Exception {
        assertNotNull(col);
        EmbedBuilder realEmbed = new EmbedBuilder();
        assertNotEquals(mockEmbed.setColor(col), realEmbed.setColor(col));
    }
}
```

As the setColor object sets the color for the EmbedBuilder class, we first create a mockEmbed object that acts as the mocking version of the EmbedBuilder object, then we pass on the arguments in both Integer class as well as Color class and observe the correct outputs. When the outputs obtained from the tests are consistent with our expectations, we then compare the outputs of the mocked object to the real object and compare their results, obtaining different outcomes due to one object not being real.

Static Analyzers With Java Discord API

Static Analysis

Static analysis, or static program analysis, is when the tester takes a look and develops an analysis of the system's source code without executing the application. On the contrary, dynamic analysis is when we analyze the source code of the application by executing the program

Static analysis is an approach that falls under the realm of code reviews. Oftentimes the people performing the analysis are other software developers as well as software testing engineers. Technical reviews and structured walkthroughs of the implementation are common forms of static analysis.

Static analysis can be used to determine if the application is adhering to its prescriptive architecture, to detect potential or existing vulnerabilities in the software, to update methods with newer standards as well as to find performance bottlenecks and provide solutions. Thus static analysis plays an important role in the software's development life cycle.

Why Static Analysis?

There are plenty of benefits of applying static analysis to a software application. Namely,

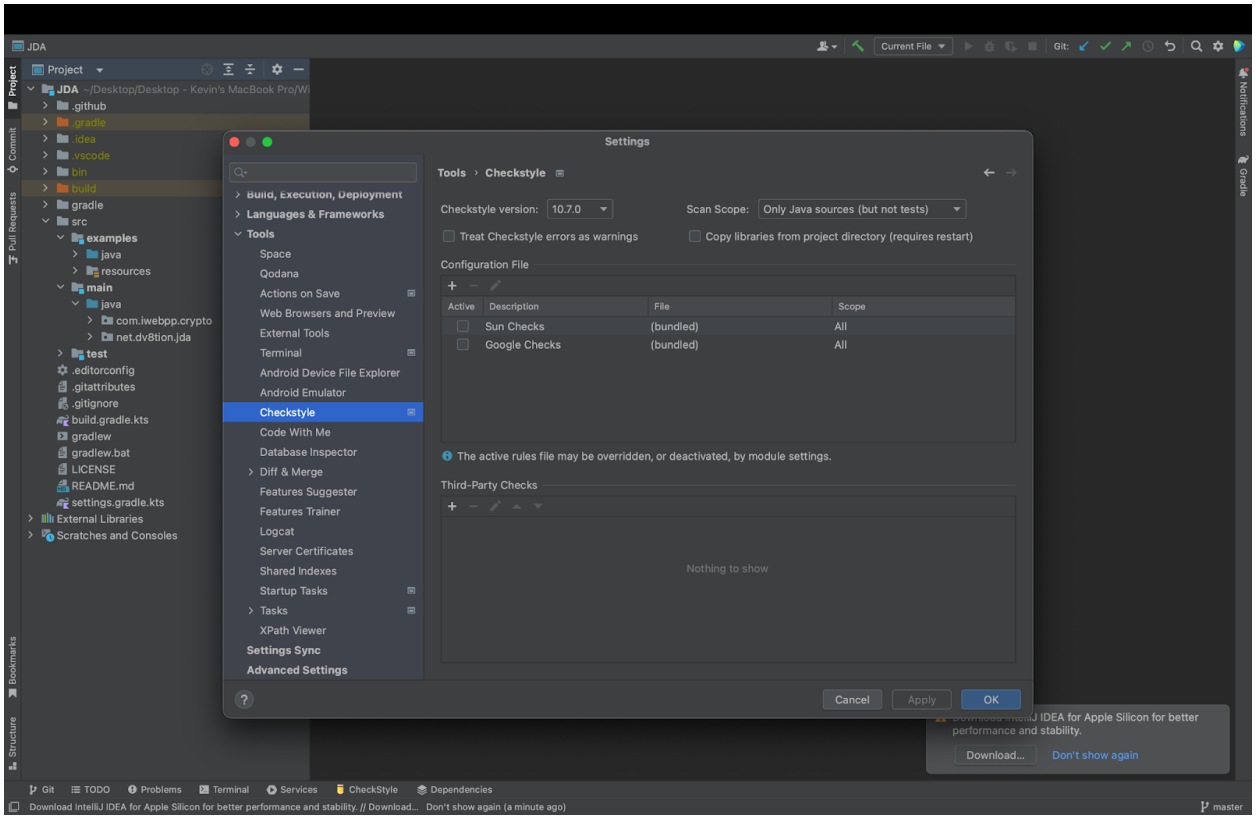
- Early bug detection
- Higher efficiency

With static analysis, we are able to detect bugs early on in the development process. Dynamic analysis would require us to execute the program, thus only exposing potential bugs later on, which could lead to potentially consequential monetary as well as time costs.

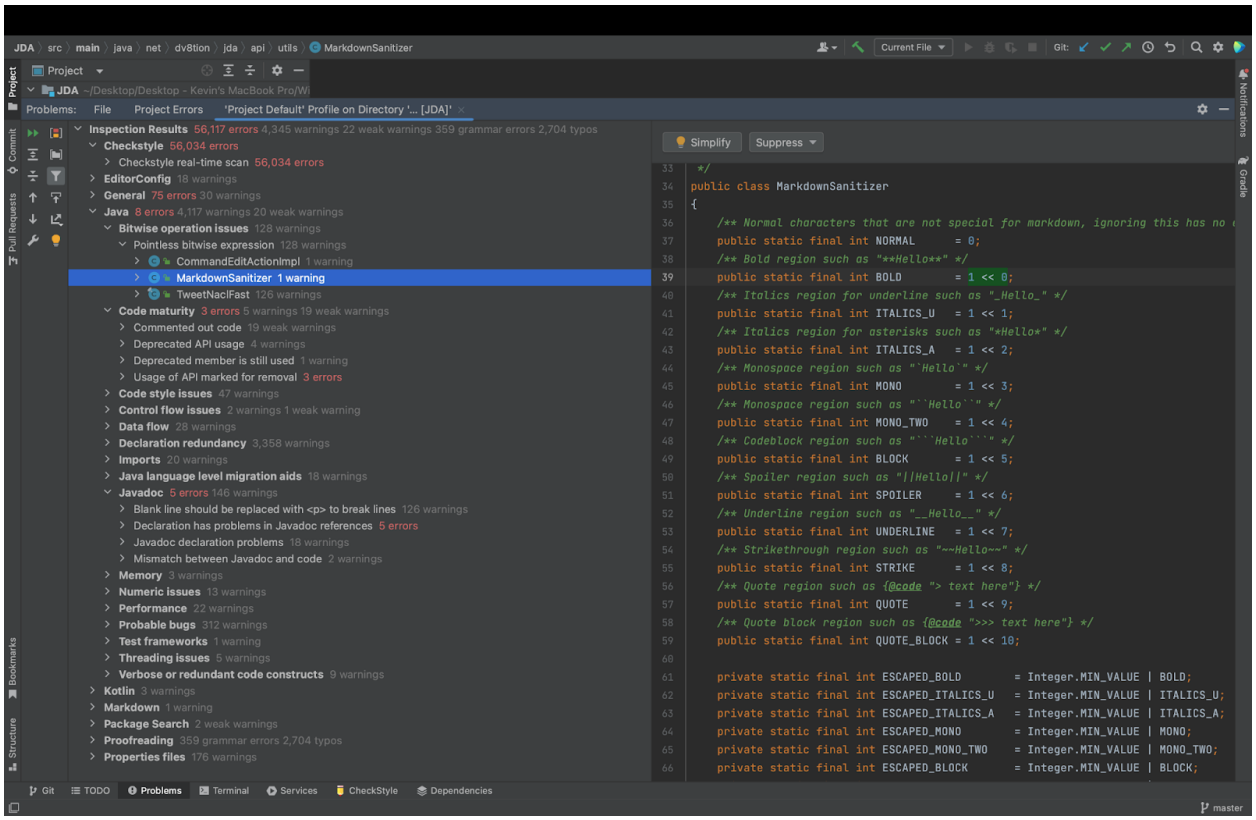
Once we are able to locate bugs early in the development process, we are then able to correct them timely as well as locating exactly where the bug is. This is crucial as it saves time and improves efficiency so we can directly look at the root cause of bugs, avoiding deep digging from testers.

Tools

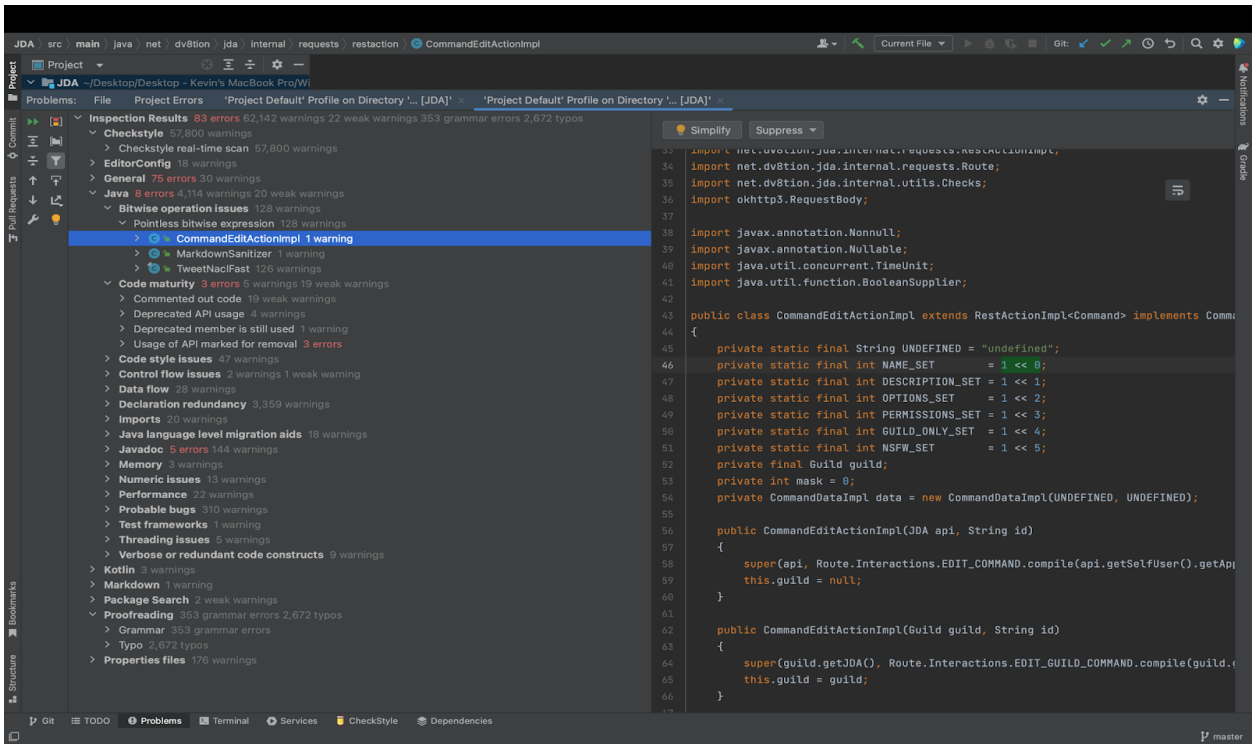
For the automatic analyzers, we decided to choose CheckStyle as well as Spotbugs. The results of both tools can be seen here.



The image below shows the results with Sun Checks, with 56034 errors in total:

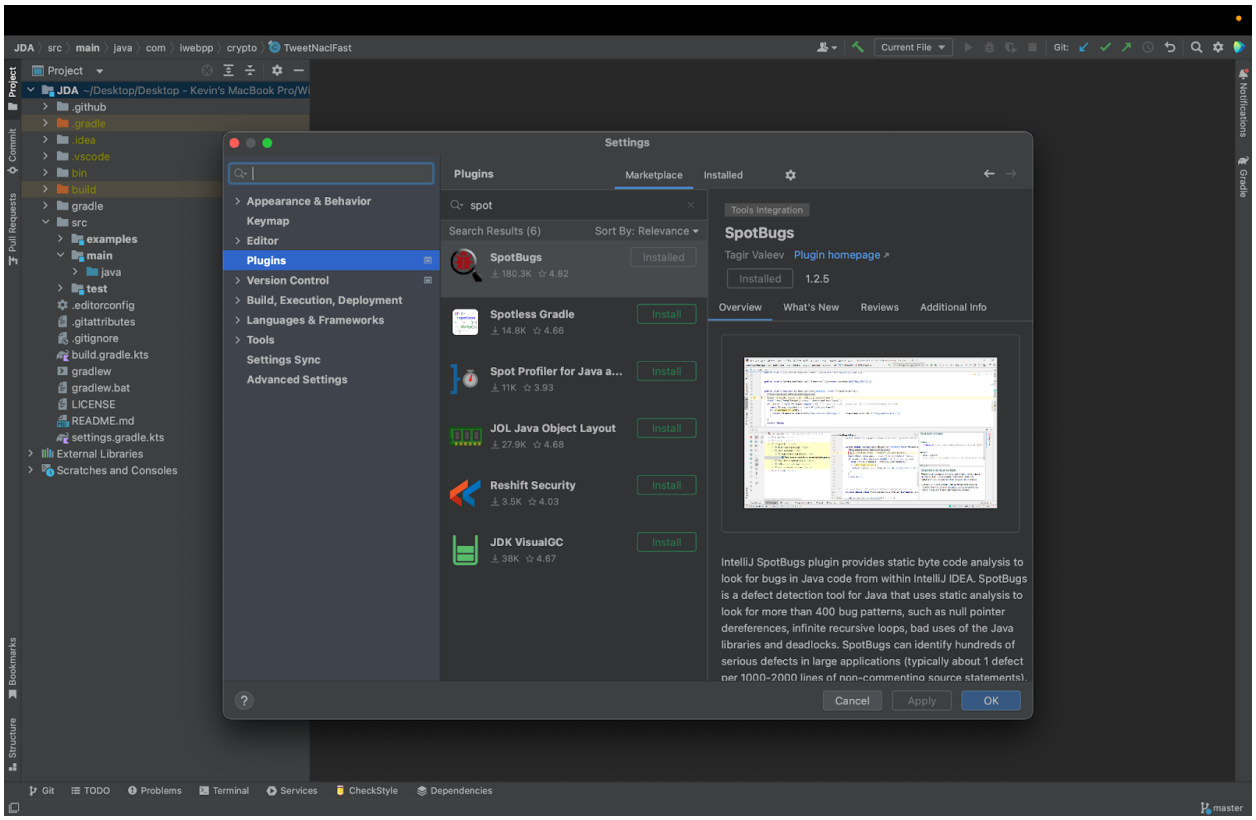


The image below shows results with Google Checks

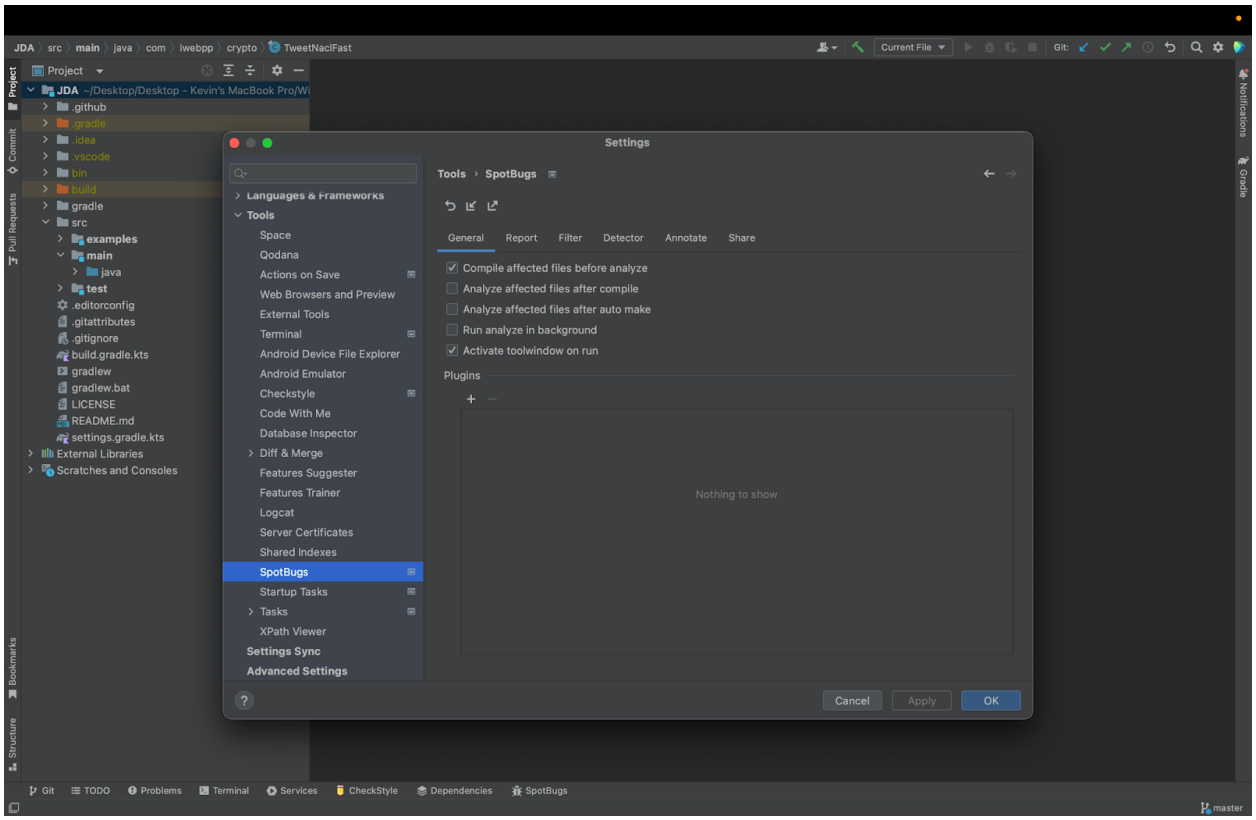


As the picture shown above, Google checks found 57800 warnings in the whole program.

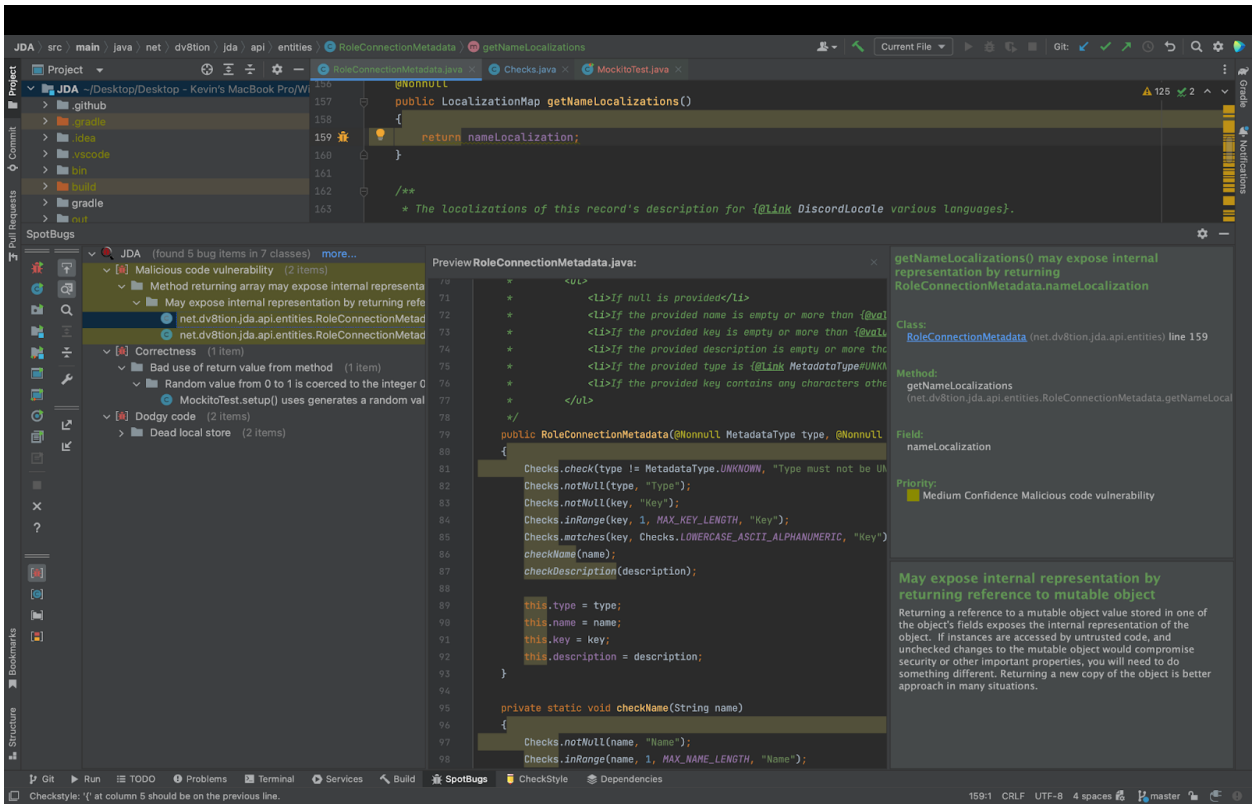
Spotbugs installation:



SpotBugs configuration:



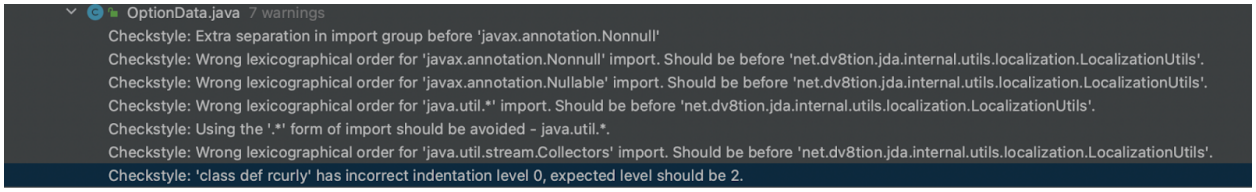
Spotbugs result:



Dive Into Bugs Found by the Static Analyzing Tools:

Checkstyle:

For Checkstyle Google Checks, take `OptionData.java` for example:



The first warning says there is a redundant separation in the import group, specifically, it suggest us delete line 31 before java.annotation.Notnull:

```
25  import net.dv8tion.jda.api.utils.data.DataArray;
26  import net.dv8tion.jda.api.utils.data.DataObject;
27  import net.dv8tion.jda.api.utils.data.DataType;
28  import net.dv8tion.jda.api.utils.data.SerializableData;
29  import net.dv8tion.jda.internal.utils.Checks;
30  import net.dv8tion.jda.internal.utils.localization.LocalizationUtils;
31
32  import javax.annotation.Nonnull;
33  import javax.annotation.Nullable;
34  import java.util.*;
35  import java.util.stream.Collectors;
```

This is not an actual problem because it will not cause any compile, runtime error, or any potential failures. The program's behavior remains the same with or without that space. On the other hand, it is more likely to be a preference of coding style. Some people may think the space doesn't make sense, some may think the space makes the import group code chunk more readable.

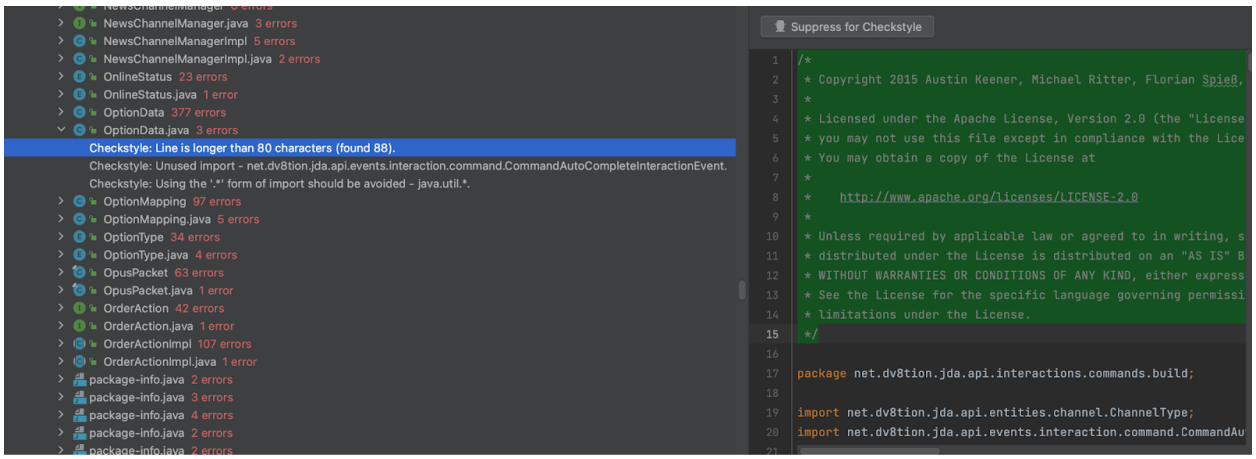
The second to the fourth warnings are also the same, checkstyle suggests we group the import statements by lexicographical order. These are not actual problems that would negatively influence the program behaviors either, but they are style preferences. Some people prefer lexicographical order, it may be easier for them to find a certain statement, while others think grouping the statement by the import libraries makes more sense.

The fifth warning, avoid using “*” in the import statement, can be an actual problem.

The main benefit of using a wildcard in import statements is that it saves typing and makes the code easier to read. Instead of having to list out every individual class that we want to import, we can use a single line with the wildcard character to import all the classes in a package. However, using a wildcard can also lead to potential naming conflicts and performance issues. When we import all classes from a package, we may inadvertently import classes with conflicting names, which can cause errors or unexpected behavior in your code. Additionally, importing unnecessary classes can increase the memory usage of our program and slow down compilation times. Therefore, it's generally recommended to use wildcard imports sparingly and only in cases where it does not create naming conflicts or negatively impact performance. In most cases, it's better to import only the specific classes that we need.

The last warning, class definition of incorrect indentation, may not be an actual problem. Since in Java, indentations won’t affect the behavior of the program. However, in the human perspective, incorrect indentations may make the code hard to read.

Using Checkstyle Sun Checks to analyze OptionData.java, there are 3 potential errors:



The first one may not be an actual problem. Although the line with long characters makes the program hard to read, it doesn't negatively affect the program's behavior, plus, in this case, the long line is in the comment.

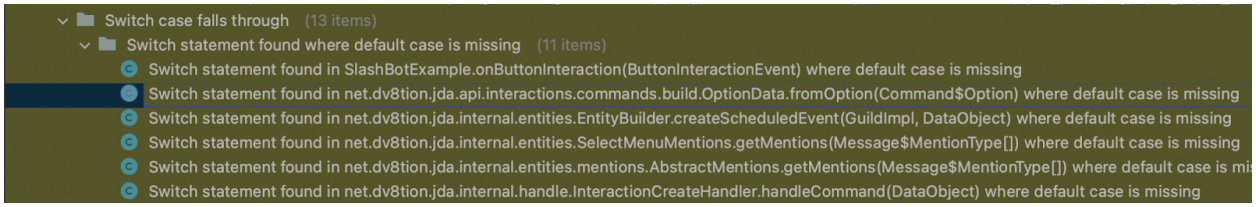
The second error, used import, can be an actual problem, since unnecessary import will slow the compilation process and negatively affect the program's performance.

The third error, avoid using "*" in the import statement, is also a problem, which is caught by the Google Check as well, like we mentioned before.

SpotBugs:

In SpotBugs, we will dive into several categories of bugs found in the software:

1. Switch case fall through



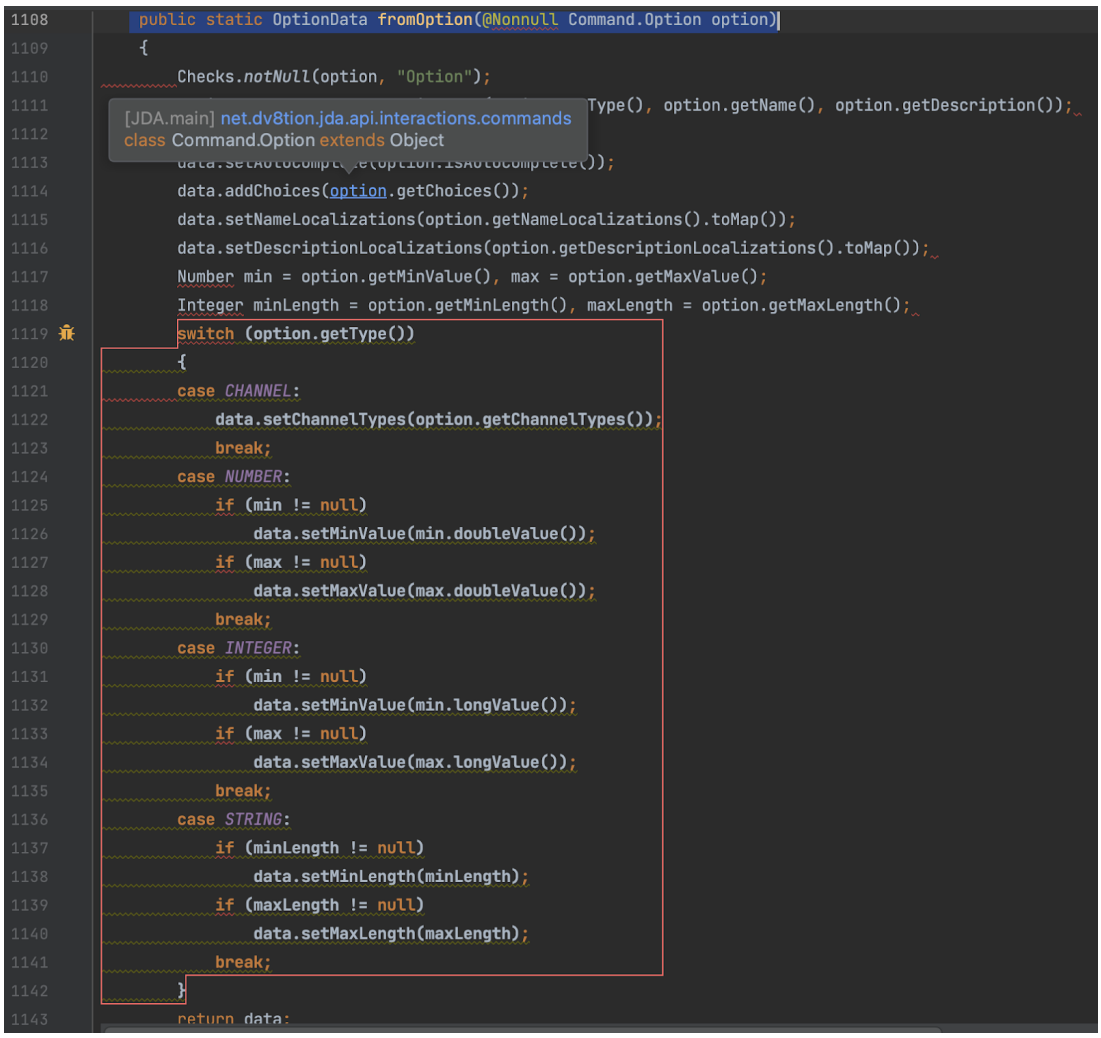
This issue can be an actual problem sometimes. It is not strictly required to include a default case in a switch statement, but it is considered a best practice to do so in order to handle unexpected inputs or situations that were not explicitly accounted for in the code.

If there is no default case in a switch statement, and the input value does not match any of the cases, then the switch statement will simply exit without executing any code. This could potentially lead to unexpected behavior or bugs in the code.

However, sometimes the switch statement just lists some special cases that need to be handled, and does not need to do anything in other cases, so not including the default case may not be a problem.

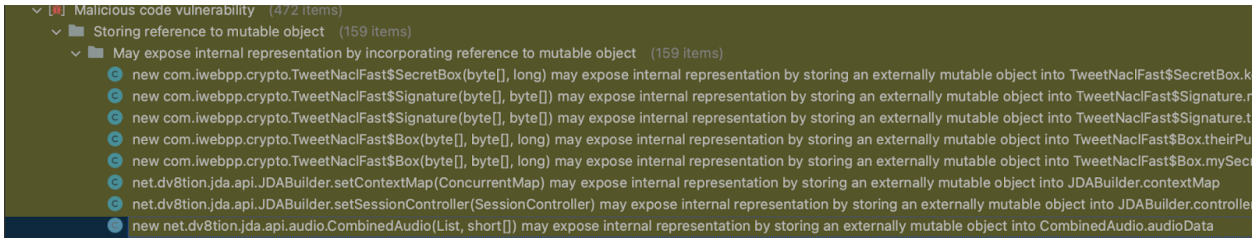
Take public static OptionData fromOption(@Nonnull Command.Option option)

in OptionData.java for example:



In this case, the four cases are the types that need to do some extra operations before returning data. Hence, missing default is not an actual problem in this case.

2. May expose internal representation by incorporating reference to mutable object:



This code stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Storing a copy of the object is a better approach in many situations.

For example, in the CombinedAudio class in CombinedAudio.java:

```
28  public class CombinedAudio
29  {
30      protected List<User> users;
31      protected short[] audioData;
32
33      public CombinedAudio(@Nonnull List<User> users, @Nonnull short[] audioData)
34      {
35          this.users = Collections.unmodifiableList(users);
36          this.audioData = audioData;
```

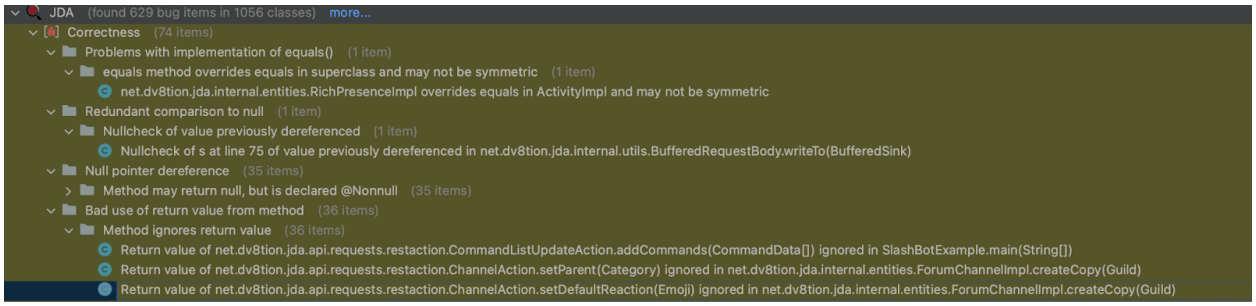
new CombinedAudio(List, short[]) may expose internal representation by storing an externally mutable object into CombinedAudio.audioData. This can potentially lead to issues where audioData is exposed and modified outside of the class, which can result in unexpected behavior.

To avoid this issue, it is generally recommended to make a defensive copy of any mutable objects that are passed into a class constructor or method. This ensures that any changes made to the object outside of the class do not affect the internal state of the class.

In this case, it would be better to make a defensive copy of the short[] array by creating a new array and copying the values from the input array into it. For example:

```
this.audioData = Arrays.copyOf(audioData, audioData.length);
```

3. Method ignore return value



This sometimes can be problematic as well.

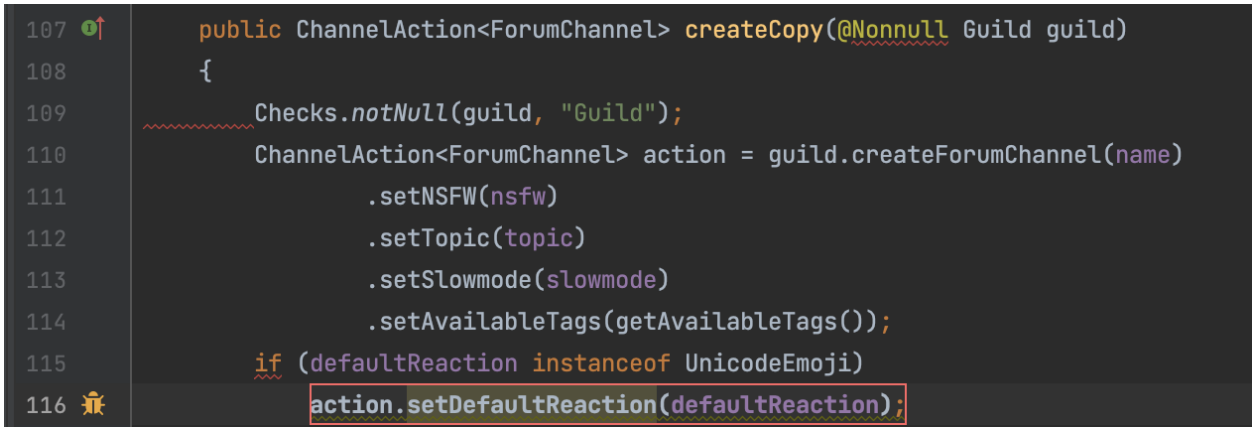
The return value of this method should be checked. One common cause of this warning is to invoke a method on an immutable object, thinking that it updates the object. For example, in the following code fragment,

```
String dateString = getHeaderField(name);

dateString.trim();
```

the programmer seems to be thinking that the trim() method will update the String referenced by dateString. But since Strings are immutable, the trim() function returns a new String value, which is being ignored here. The code should be corrected to:

```
String dateString = getHeaderField(name);

dateString = dateString.trim();
```

In the following case, it has potential to be an actual problem:



Since setDefaultReaction() is not a void function setter:

```
/**
 * Sets the <b><u>default reaction emoji</u></b> of the new {@link ForumChannel}.
 * <br>This does not support custom emoji from other guilds.
 *
 * @param  emoji
 *         The new default reaction emoji, or null to unset.
 *
 * @return The current ChannelAction, for chaining convenience
 *
 * @see    ForumChannel#getDefaultReaction()
 */
2 usages  1 implementation  Florian Spieß
@Nonnull
@CheckReturnValue
ChannelAction<T> setDefaultReaction(@Nullable Emoji emoji);
```

Moreover, there is a @CheckReturnValue annotation, which means that the developers of JDA may make a careless mistake in this part of the code

Therefore, we should always check the return value while invoking this method, by creating a new variable to store and check the return value.

Differences Between Checkstyle and Spotbugs:

Checkstyle seems to focus more on how clean and efficient the application is. Many of the issues highlighted by Checkstyle have more to do with the way the code looks, such as the extra spacing in the import statements. All of the issues highlighted by Checkstyle are not code-breaking. This tool seems to check more for how well the application follows a standard "style" or "rules" of programming. Users of this tool can utilize it to swiftly clean and make code more understandable, as well as ensuring the full uniformity of the code. We tested this tool with Google and Sun Checks, but one can use a configuration file that fits the user's preference.

Google and Sun are two different sets of rules that Checkstyle can use. Google Checkstyle analyzes the code based on the standard that Google practices in code. Meanwhile, Sun Checkstyle analyzes the code based on a code guide curated by Sun Microsystems; a company that was acquired by Oracle. Both are useful for molding a project to a coding standard for Java development. Which one to use depends on developer preference and project requirements. They are different configurations, but they can also provide overlapping results. The results of running these two different Checkstyle standards was shown previously. For example, both styles gave a warning against using the "*" symbol for import statements.

Running SpotBugs on the JDA API resulted in more serious warnings in the code. These faults have the possibility of becoming errors causing the program to stray from expected behavior. For example, exposing the internal representation of CombinedAudio is a problem the developers would definitely want to avoid. Being able to identify potential bugs and vulnerabilities is the highlight of this tool and can help developers prevent compilation and runtime errors.

These tools are similar in nature but are different. The warnings given by these tools do not overlap. They complement each other and work well to fully encompass a static analysis of code. While Checkstyle provides the common conventions and structure for code, SpotBugs identifies the vulnerabilities that may compromise the reliability of a program. Between the two, SpotBugs gives more pertinent information because having bugs can break a program or cause security issues. However, both tools are valuable to fully encompass static analysis.