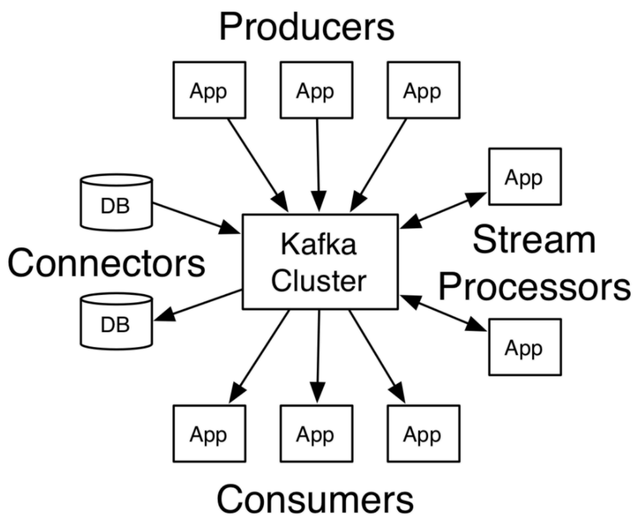
# SWE215 Project Report

## 1.Introduction

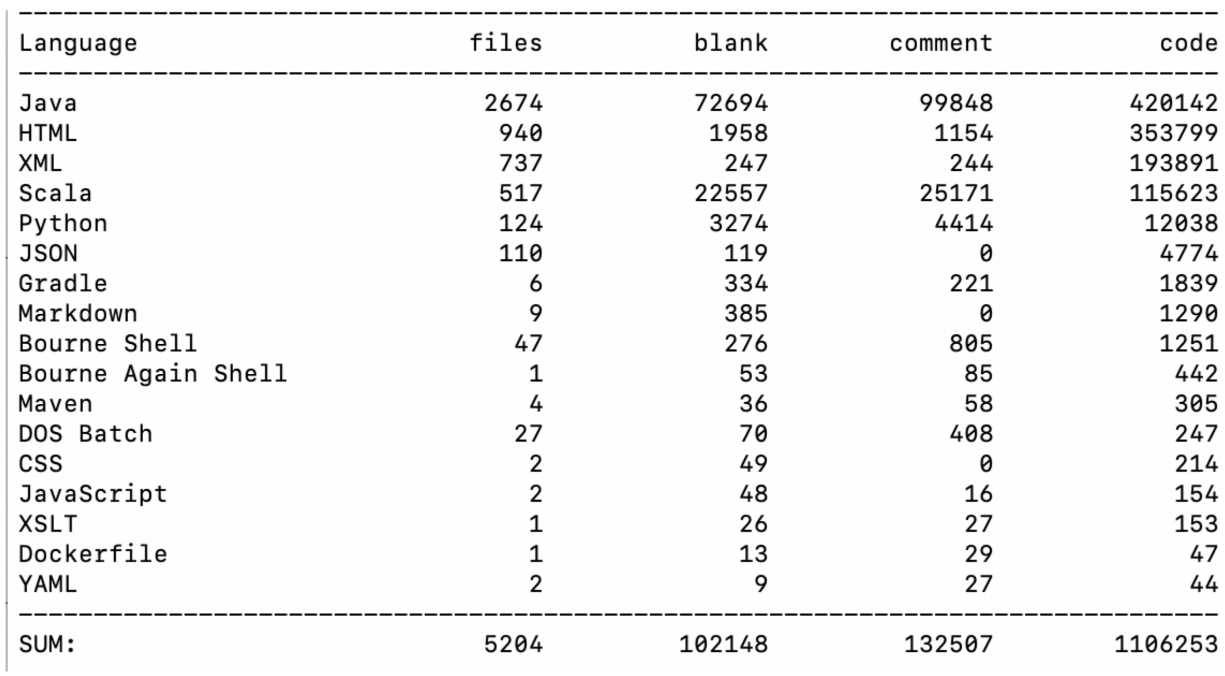
The project I'm testing this quarter is Apache Kafka. Apache Kafka is a distributed streaming platform. It can be used to build real-time streaming data pipelines that reliably get data between systems or applications and build real-time streaming applications that transform or react to the streams of data (Quote from https://kafka.apache.org/intro). The main feature of Kafka is supporting real-time data. Kafka stores streams of records on categories called topics. There are five core APIs shows as follow:



With Producer API, application can publish a stream of records to one or more Kafka topics. With Consumer API, application can subscribe to one or more topics ad process the stream of records produced to them. Stream API provides the ability for an application to act as a stream processor. The Connector API allows Kafka to connect with other applications such as data base. The Admin API allows managing and inspecting topics, brokers and other Kafka objects.

It is written in JAVA and built with Gradle. I use a tool called cloc to count the lines of code in Kafka project. Here is the result:

As shown above, most files are written in Java.



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                          2674          72694          99848         420142
HTML                           940           1958           1154         353799
XML                            737            247            244         193891
Scala                          517          22557          25171         115623
Python                         124           3274           4414          12038
JSON                           110            119              0           4774
Gradle                           6            334            221           1839
Markdown                         9            385              0           1290
Bourne Shell                    47            276            805           1251
Bourne Again Shell               1             53             85            442
Maven                            4             36             58            305
DOS Batch                       27             70            408            247
CSS                              2             49              0            214
JavaScript                       2             48             16            154
XSLT                             1             26             27            153
Dockerfile                       1             13             29             47
YAML                             2              9             27             44
-------------------------------------------------------------------------------
SUM:                          5204         102148         132507        1106253
```

In this report I will use different testing techniques to test Apache Kafka.

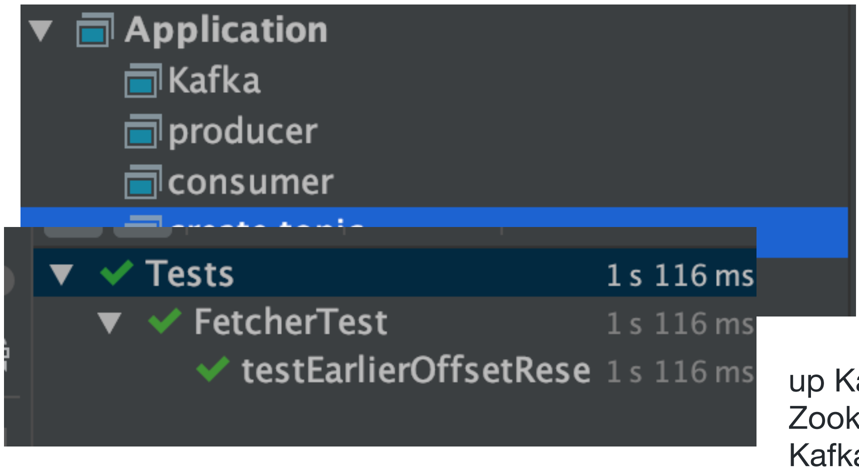
## 2. Set up

I build Kafka with IntelliJ-IDEA. In order to run Kafka, first we need to install some dependencies. Java of course, we also need to install Zookeeper and Scala and Gradle. Since the Kafka project uses Gradle to build. We need to add some dependencies into the build.gradle file:

```
compile libs.slf4jlog4j

compile(libs.zookeeper) {

        …

        exclude module: ‘slf4j-simple'

}

testCompile group: 'org.slf4j', name: 'slf4j-simple', version: '1.7.21'
```

This is for supporting log4j. Currently I still have some problems with this tool. It is because the log4j configuration file: log4j.properties. However this has no impact on the project itself. I will fix this problem in the future.

As mentioned in the Introduction section, Kafka has five core APIs, we need to add some configurations to the Run/Debug Configurations.

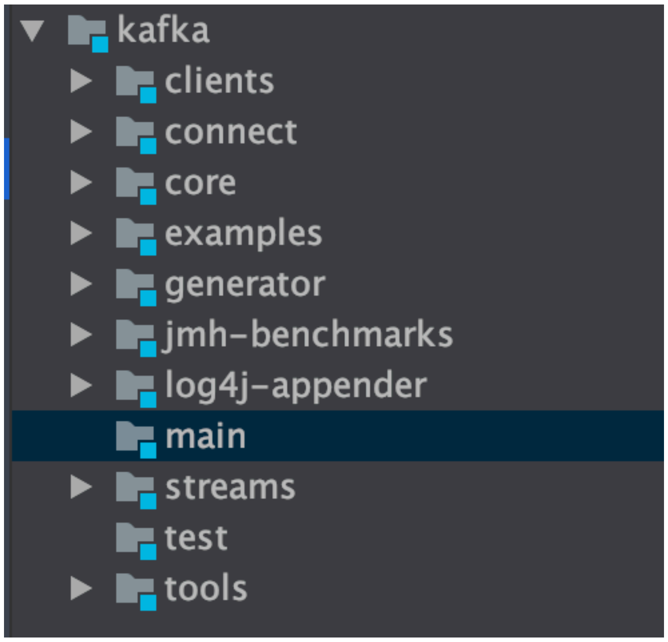


```
▼ Application
    Kafka
    producer
    consumer
```

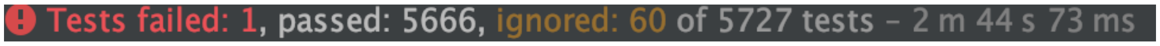
```
▼ ✓ Tests                              1 s 116 ms
    ▼ ✓ FetcherTest                    1 s 116 ms
        ✓ testEarlierOffsetRese        1 s 116 ms
```

When start up Kafka, we can run Zookeeper, then run Kafka, then run creat_topic, run producer and run consumer. Then we can type something in the producer console and in the consumer console we can see the message.(We can also use the Docker file in the project to run or test the application)

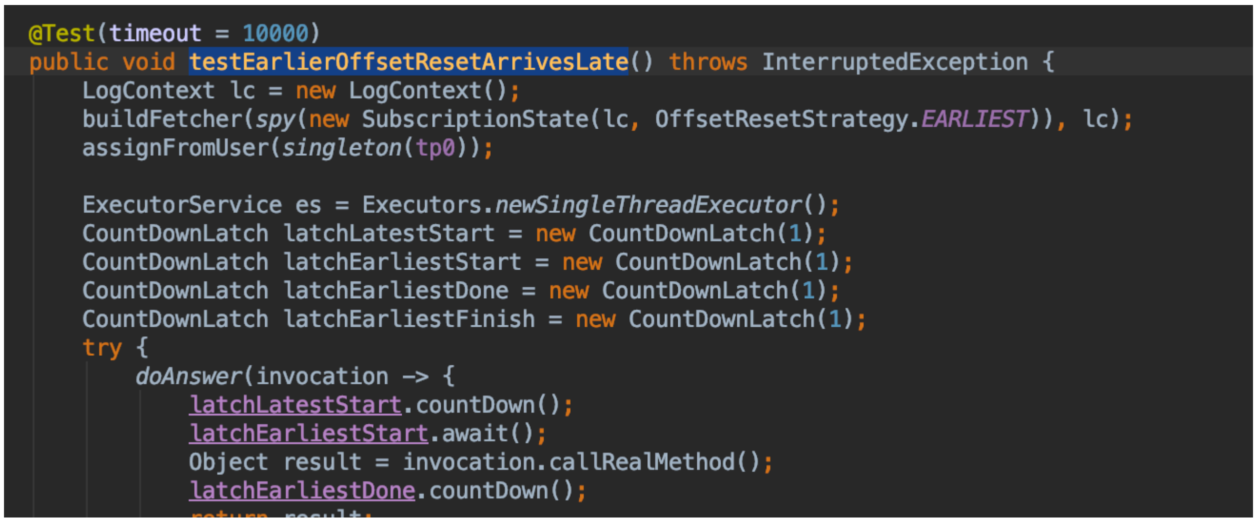
Kafka is a large application contains many modules. Most tests are written with JUnit. Besides, it also uses python script to test the whole application. There are also some test tools for Kafka. But right now I'm focused on the unit test written in JUnit. There are lots of JUnit test cases for most modules. I choose the clients module this time.



```
▼ kafka
    ▶ clients
    ▶ connect
    ▶ core
    ▶ examples
    ▶ generator
    ▶ jmh-benchmarks
    ▶ log4j-appender
      main
    ▶ streams
      test
    ▶ tools
```



```
Tests failed: 1, passed: 5666, ignored: 60 of 5727 tests – 2 m 44 s 73 ms
```



```java
@Test(timeout = 10000)
public void testEarlierOffsetResetArrivesLate() throws InterruptedException {
    LogContext lc = new LogContext();
    buildFetcher(spy(new SubscriptionState(lc, OffsetResetStrategy.EARLIEST)), lc);
    assignFromUser(singleton(tp0));

    ExecutorService es = Executors.newSingleThreadExecutor();
    CountDownLatch latchLatestStart = new CountDownLatch(1);
    CountDownLatch latchEarliestStart = new CountDownLatch(1);
    CountDownLatch latchEarliestDone = new CountDownLatch(1);
    CountDownLatch latchEarliestFinish = new CountDownLatch(1);
    try {
        doAnswer(invocation -> {
            latchLatestStart.countDown();
            latchEarliestStart.await();
            Object result = invocation.callRealMethod();
            latchEarliestDone.countDown();
```

Here is the result of running all the test cases in client module:

The fail test case is the FetcherTest/testEarlierOffsetResetArrivesLate. Because of the test case is time out, it failed. However when running this single test case. It won't fail.

Since I'm using IDE, we can simply right click the test folder or any test files to run the tests.

Most test cases are using @Test notation with simple assertEquals() method. Since it is client module, most test cases are about client configuration such as IP address or host name. It can be categorize into five32 categories: Configuration testing, network testing, security testing, mechanism testing and example testing.

## 3. Testing

### 3.1 Partitioning

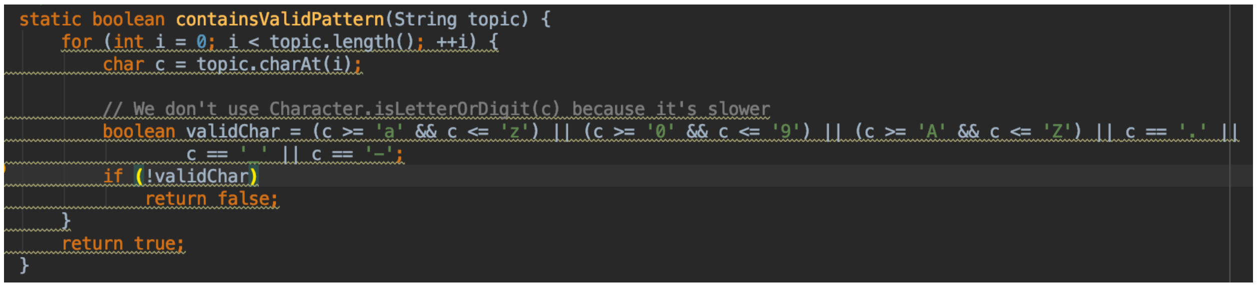
As for a large project like Kafka, looking into the code and understand the code is difficult. Kafka is not a independent project. It relies on Scala, Zookeeper and other projects. The dependencies between these projects makes it hard to run white box testing. In order to do white box test we need to understand the code first, which is hard. However, Kafka has simple functions. In the Kafka website, it describes some popular use cases: Messaging, Website Activity Tracking, Metrics, Log Aggregation, Stream Processing, Event Sourcing, Commit Log, which makes it easier to do functional testing.

Systematic functional testing is with the test cases we deriving from the SRS, we use some valuable input to test some representative classes which are more likely to fail or not fail at all. Speaking of input, in a large software the input space would be large and hard to enumerate. In order to address this difficulty, we need partition testing. We split the entire input space into several partitions. With partitions we can assume that failure will appear densely in some partitions.

For Apache Kafka, the input space is not large enough to run a reasonable partition test, because its inputs are messages or some runtime arguments. I choose a feature which is the most reasonable one for partition testing ——Topics.

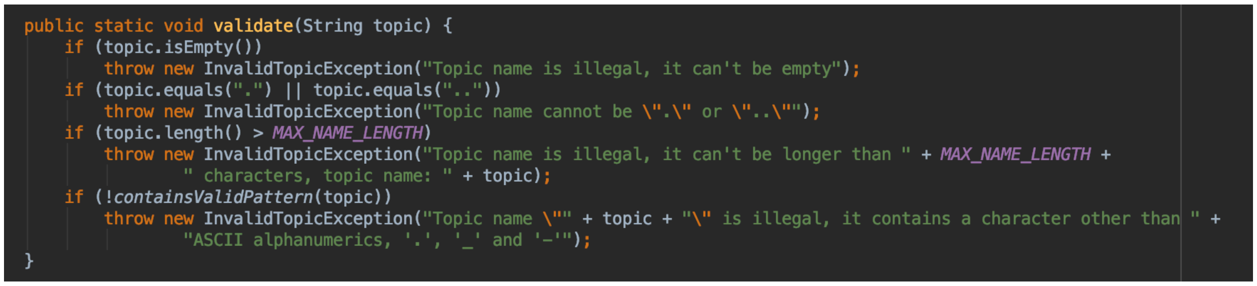
In Apache Kafka, a topic is a category or feed name to which records are published. Each topic would have a name from users' input. We can partition this input space to validate if it's a valid topic name.

In the class Topic we can see a valid topic name has a certain pattern which can only contain the ASCII alphanumerics, '.', '_', and '-'. And it can not be empty or longer than  Maximum length. Besides, due to the metric limitation, topic names contain '.' or '-'  at the same position in the string will be considered as the same topic name, which will be considered as collision.



```
static boolean containsValidPattern(String topic) {
    for (int i = 0; i < topic.length(); ++i) {
        char c = topic.charAt(i);

        // We don't use Character.isLetterOrDigit(c) because it's slower
        boolean validChar = (c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || (c >= 'A' && c <= 'Z') || c == '.' ||
                c == '_' || c == '-';
        if (!validChar)
            return false;
    }
    return true;
}
```

It also provides a method validate( ) to help us check if a topic name is valid.



```
public static void validate(String topic) {
    if (topic.isEmpty())
        throw new InvalidTopicException("Topic name is illegal, it can't be empty");
    if (topic.equals(".") || topic.equals(".."))
        throw new InvalidTopicException("Topic name cannot be \".\" or \"..\"");
    if (topic.length() > MAX_NAME_LENGTH)
        throw new InvalidTopicException("Topic name is illegal, it can't be longer than " + MAX_NAME_LENGTH +
                " characters, topic name: " + topic);
    if (!containsValidPattern(topic))
        throw new InvalidTopicException("Topic name \"" + topic + "\" is illegal, it contains a character other than " +
                "ASCII alphanumerics, '.', '_' and '-'");
}
```

With the knowledge of Topic name, I make several partitions:

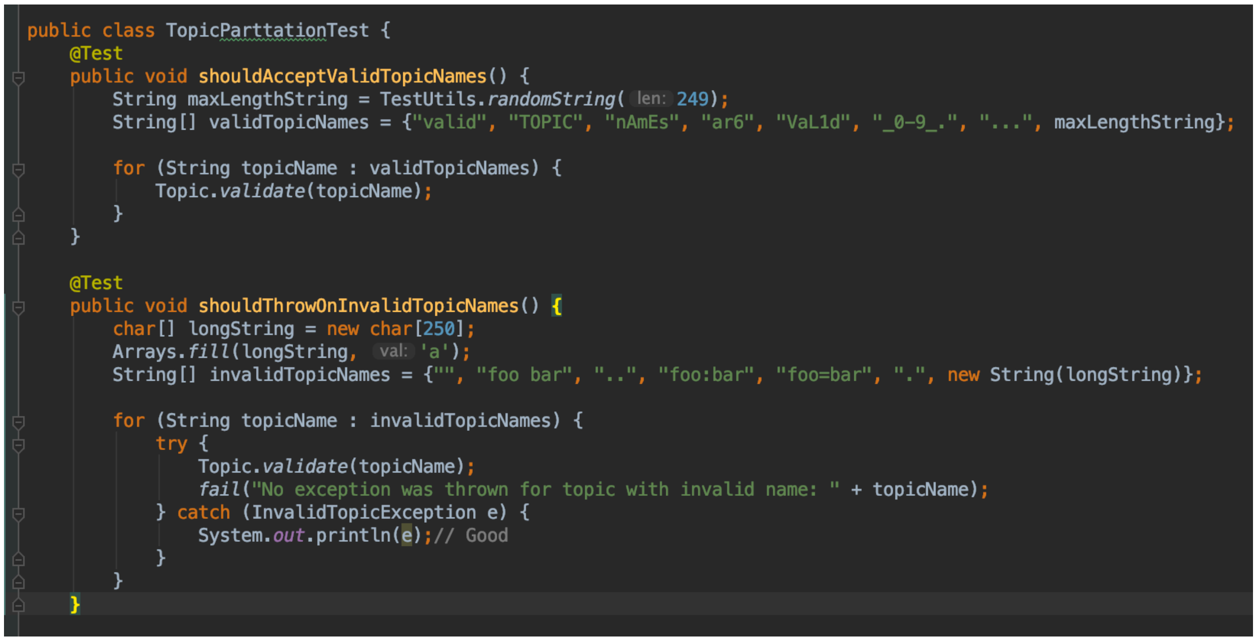
ValidTopicNames: no invalid chars(All char is between a-z||A-Z||0-9||.||_)

InvalidTopicNames: contains invalid chars(Some chars is outside the range above)

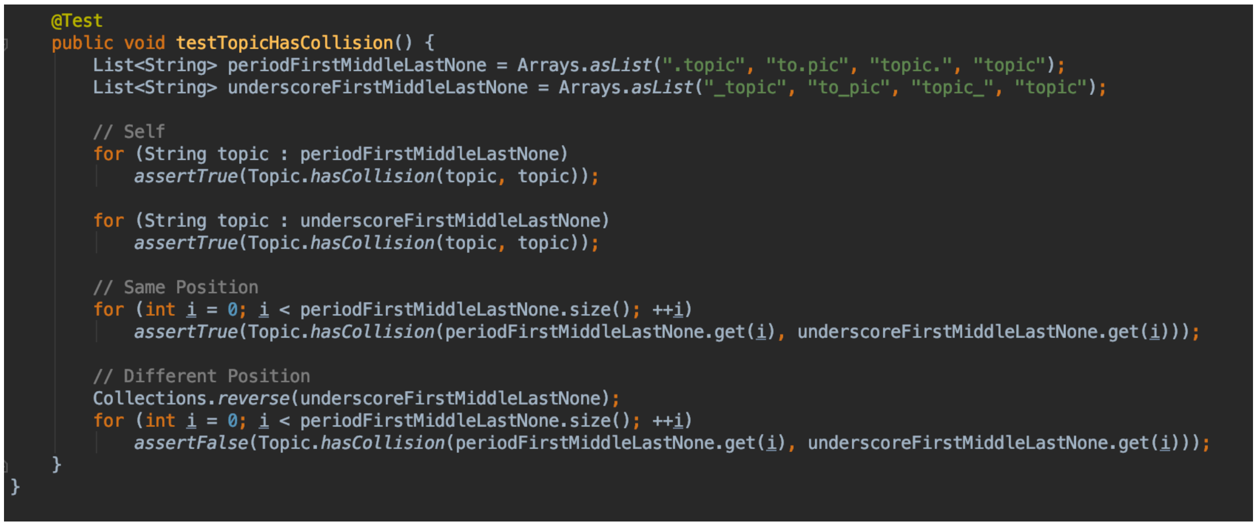
CollisionTopicNames: topic names are valid but there are collision(contains pattern .string && _string)

Here is my partition test class and result.

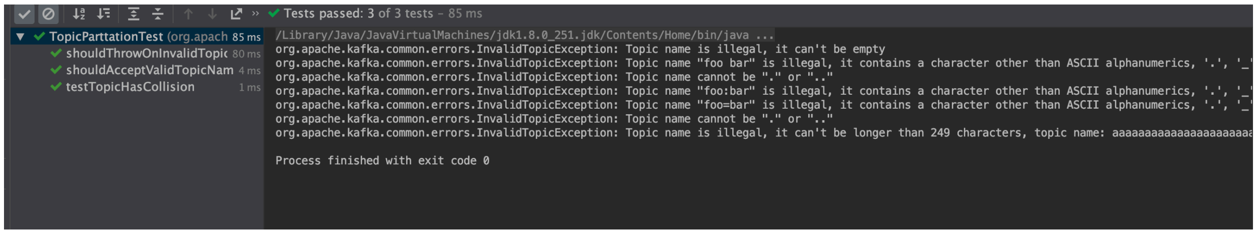
There are not failures cause I use assertFalse to test collisions and throw exceptions when encounter invalid inputs.



```java
public class TopicParttationTest {
    @Test
    public void shouldAcceptValidTopicNames() {
        String maxLengthString = TestUtils.randomString( len: 249);
        String[] validTopicNames = {"valid", "TOPIC", "nAmEs", "ar6", "VaL1d", "_0-9_.", "...", maxLengthString};

        for (String topicName : validTopicNames) {
            Topic.validate(topicName);
        }
    }

    @Test
    public void shouldThrowOnInvalidTopicNames() {
        char[] longString = new char[250];
        Arrays.fill(longString,  val: 'a');
        String[] invalidTopicNames = {"", "foo bar", "..", "foo:bar", "foo=bar", ".", new String(longString)};

        for (String topicName : invalidTopicNames) {
            try {
                Topic.validate(topicName);
                fail("No exception was thrown for topic with invalid name: " + topicName);
            } catch (InvalidTopicException e) {
                System.out.println(e);// Good
            }
        }
    }
```



```java
    @Test
    public void testTopicHasCollision() {
        List<String> periodFirstMiddleLastNone = Arrays.asList(".topic", "to.pic", "topic.", "topic");
        List<String> underscoreFirstMiddleLastNone = Arrays.asList("_topic", "to_pic", "topic_", "topic");

        // Self
        for (String topic : periodFirstMiddleLastNone)
            assertTrue(Topic.hasCollision(topic, topic));

        for (String topic : underscoreFirstMiddleLastNone)
            assertTrue(Topic.hasCollision(topic, topic));

        // Same Position
        for (int i = 0; i < periodFirstMiddleLastNone.size(); ++i)
            assertTrue(Topic.hasCollision(periodFirstMiddleLastNone.get(i), underscoreFirstMiddleLastNone.get(i)));

        // Different Position
        Collections.reverse(underscoreFirstMiddleLastNone);
        for (int i = 0; i < periodFirstMiddleLastNone.size(); ++i)
            assertFalse(Topic.hasCollision(periodFirstMiddleLastNone.get(i), underscoreFirstMiddleLastNone.get(i)));
    }
}
```



Tests passed: 3 of 3 tests – 85 ms
TopicParttationTest (org.apach 85 ms
shouldThrowOnInvalidTopic 80 ms
shouldAcceptValidTopicNam 4 ms
testTopicHasCollision 1 ms

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_251.jdk/Contents/Home/bin/java ...
org.apache.kafka.common.errors.InvalidTopicException: Topic name is illegal, it can't be empty
org.apache.kafka.common.errors.InvalidTopicException: Topic name "foo bar" is illegal, it contains a character other than ASCII alphanumerics, '.', '_'
org.apache.kafka.common.errors.InvalidTopicException: Topic name cannot be "." or ".."
org.apache.kafka.common.errors.InvalidTopicException: Topic name "foo:bar" is illegal, it contains a character other than ASCII alphanumerics, '.', '_'
org.apache.kafka.common.errors.InvalidTopicException: Topic name "foo=bar" is illegal, it contains a character other than ASCII alphanumerics, '.', '_'
org.apache.kafka.common.errors.InvalidTopicException: Topic name cannot be "." or ".."
org.apache.kafka.common.errors.InvalidTopicException: Topic name is illegal, it can't be longer than 249 characters, topic name: aaaaaaaaaaaaaaaaaaaaa

Process finished with exit code 0
```

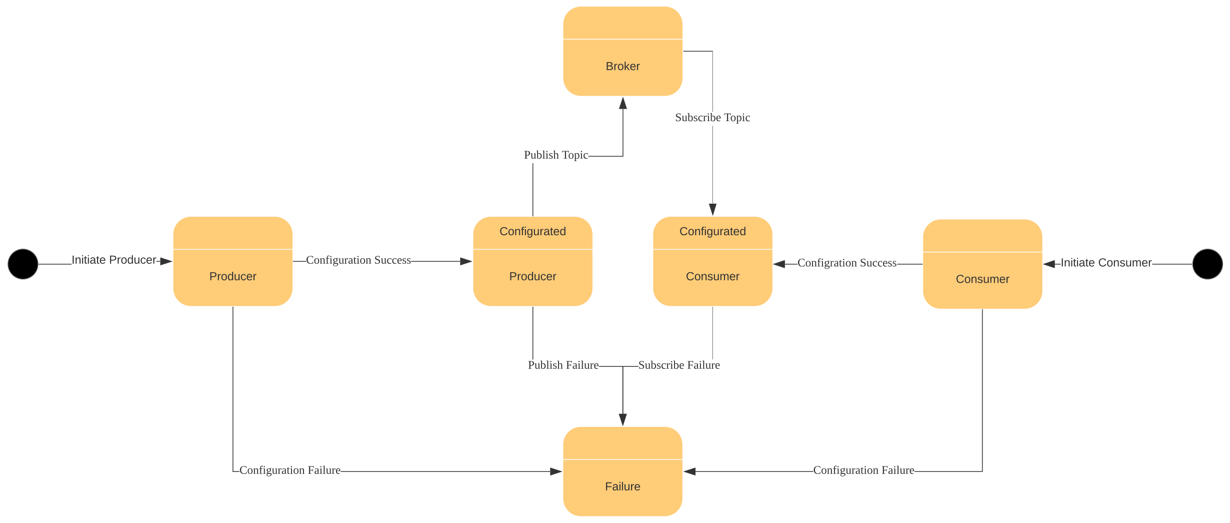
## 3.2 Functional Models

### 1. Functional Models

Functional Model is a structured representation of the functions. In Software Testing, when we are doing black box testing, Functional Models are useful. Functional Models can capture the requirement specifications with a visual representation. For example, finite state machine can help us understanding how system reacts with some certain inputs, which can help us test the system .

### 2. Kafka's functional model

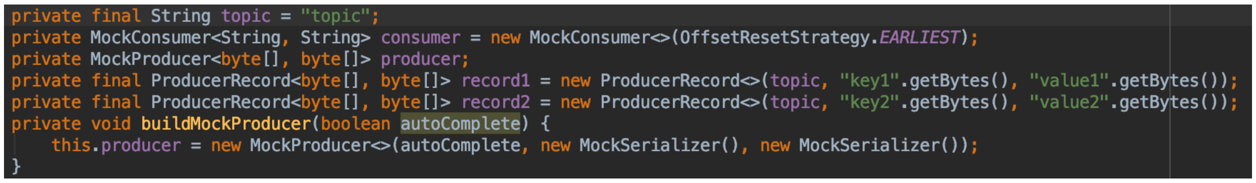
For Apache Kafka, the main feature is data streaming between multiple nodes in a distributed system. There are two main roles in this procedure: Producer and Consumer. Producer is a client or a program, which produces the message and pushes it to the Topic. Consumer is a client or a program, which consumes the published messages from the Producer. In Kafka, a Topic is a category or a stream name to which messages are published and a Kafka Broker is a server where message data is replicated and persisted on. When a producer publish a topic, if a consumer subscribe the same topic, the consumer will receive the message. Before this procedure, both producer and consumer need to initiate and do some configuration. This procedure needs so many methods so I'm not going to describe it in test cases. However it can be described in the functional model.



## 3. Test Cases

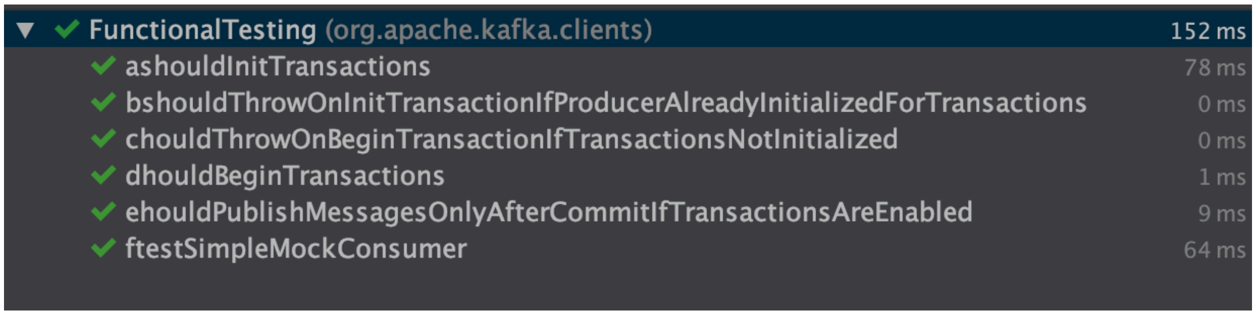
For test cases, I write six test cases for this functional model(Only for the publish/subscribe part, not including the configuration part). For producer, the procedure for publish a topic is: Initiate transaction -> Begin transaction -> Publish Message. For consumer, it can just pull the record/topic from the broker.

In the test class, first we need to initiate a producer, consumer, topic and a record.



```
private final String topic = "topic";
private MockConsumer<String, String> consumer = new MockConsumer<>(OffsetResetStrategy.EARLIEST);
private MockProducer<byte[], byte[]> producer;
private final ProducerRecord<byte[], byte[]> record1 = new ProducerRecord<>(topic, "key1".getBytes(), "value1".getBytes());
private final ProducerRecord<byte[], byte[]> record2 = new ProducerRecord<>(topic, "key2".getBytes(), "value2".getBytes());
private void buildMockProducer(boolean autoComplete) {
    this.producer = new MockProducer<>(autoComplete, new MockSerializer(), new MockSerializer());
}
```

The MockProducer and MockConsumer API are provided by Kafka for testing only. Then I write five test cases for producer and one test case for consumer. I use @FixMethodOrder annotation to let the test cases run orderly.



```
▼ ✓ FunctionalTesting (org.apache.kafka.clients)                                  152 ms
    ✓ ashouldInitTransactions                                                      78 ms
    ✓ bshouldThrowOnInitTransactionIfProducerAlreadyInitializedForTransactions      0 ms
    ✓ choulдThrowOnBeginTransactionIfTransactionsNotInitialized                     0 ms
    ✓ dhouldBeginTransactions                                                       1 ms
    ✓ ehouldPublishMessagesOnlyAfterCommitIfTransactionsAreEnabled                  9 ms
    ✓ ftestSimpleMockConsumer                                                      64 ms
```

The shouldInitTransactions test case assures that the producer has initiated a transaction. The shouldThrowOnInitTransactionIfProducerAlreadyInitializedForTransactions test case assures that a producer only initiates a transaction once before committing it. The shouldThrowOnBeginTransactionIfTransactionsNotInitialized test case assures that a producer has to initiate a transaction before beginning it. The shouldBeginTransactions test case assures that a producer can begin a transaction. The shouldPublishMessagesOnlyAfterCommitIfTransactionsAreEnabled test case assures that a producer can publish its message. The testSimpleMockConsumer test case assures that the consumer can get the message.

## 3.3 Structural Testing

### 1. Structural Testing

Contrast to Black-Box Testing, Structural Testing is the type of testing known as White-Box testing. It requires the knowledge of the code. It shifts the focus from the functionality of the system to the actual implementation of the system. It is a complementary to Functional Testing. Using this technique the test cases drafted according to system requirements can be first analyzed and then more test cases can be added to increase the coverage. It can be automated so it's useful when it comes to thorough testing. With Structural Testing, we always want to increase the coverage. There are several coverage criterion: Statement Coverage, Branch Coverage, Condition Coverage and Path Coverage.

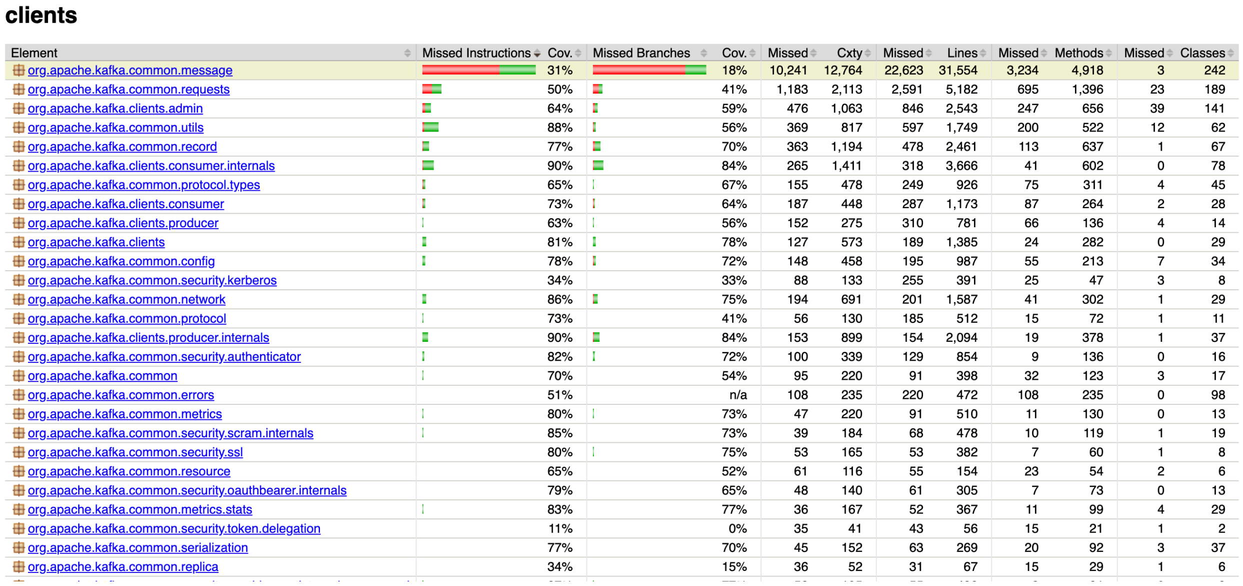
Structural Testing is important mostly because it focuses more on the implementation of the system. It can reveal errors hidden in the code. Besides, since it can be automated, it provides a way to do a thorough testing of the software. Last but not least, there are lots of tools support Structural Testing, which makes it easier for developer to do Structural Testing.

### 2. Coverage of Existing Test Cases

For the Apache Kafka, it has lots of test cases. I tried to run all the test cases with coverage tool and my both my device's memory and disk ran out and I can't finish the testing. So I decide to run the coverage on the client module(The core module contains more than 8000 test cases. Each of them consumes lots of time and resources. This module contains the core classes and methods in Kafka. Though I think I should choose this module, I can't run all the test cases on my device. Besides, the core module uses Scala. All the test cases in core module are using Scala. I think maybe that's the reason why it consumes so much time and resources.) I tried different coverage tools. I'm using Intellij IDEA with Emma plugin as coverage tool. EMMA is an open-source toolkit for measuring and reporting *Java code coverage*. Kafka uses Gradle with Jacoco as a plugin.

First I use Jacoco. In the build.gradle file, Kafka uses Jacoco as plugin.

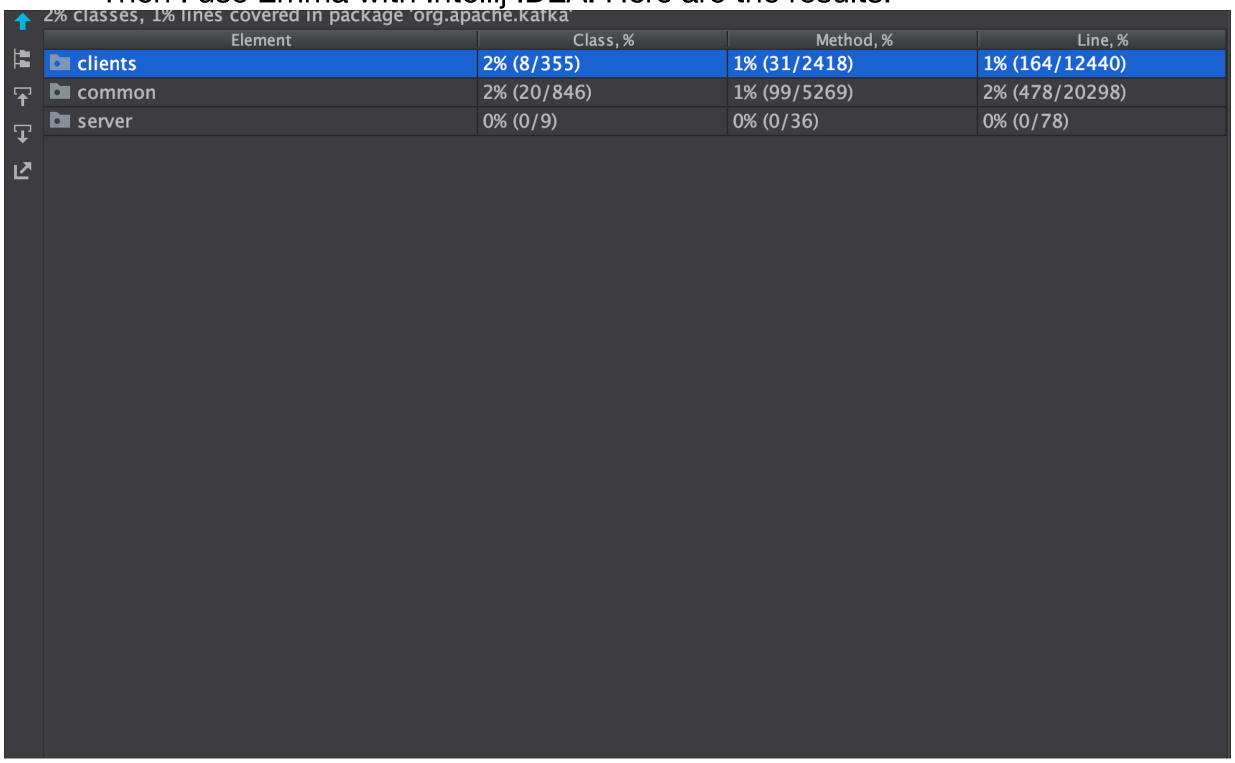
We use the command "./gradlew client:reportCoverage" in the console to run all the test cases in Client module with Jacoco as coverage tool. Here are the results:



## clients

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| org.apache.kafka.common.message | | 31% | | 18% | 10,241 | 12,764 | 22,623 | 31,554 | 3,234 | 4,918 | 3 | 242 |
| org.apache.kafka.common.requests | | 50% | | 41% | 1,183 | 2,113 | 2,591 | 5,182 | 695 | 1,396 | 23 | 189 |
| org.apache.kafka.clients.admin | | 64% | | 59% | 476 | 1,063 | 846 | 2,543 | 247 | 656 | 39 | 141 |
| org.apache.kafka.common.utils | | 88% | | 56% | 369 | 817 | 597 | 1,749 | 200 | 522 | 12 | 62 |
| org.apache.kafka.common.record | | 77% | | 70% | 363 | 1,194 | 478 | 2,461 | 113 | 637 | 1 | 67 |
| org.apache.kafka.clients.consumer.internals | | 90% | | 84% | 265 | 1,411 | 318 | 3,666 | 41 | 602 | 0 | 78 |
| org.apache.kafka.common.protocol.types | | 65% | | 67% | 155 | 478 | 249 | 926 | 75 | 311 | 4 | 45 |
| org.apache.kafka.clients.consumer | | 73% | | 64% | 187 | 448 | 287 | 1,173 | 87 | 264 | 2 | 28 |
| org.apache.kafka.clients.producer | | 63% | | 56% | 152 | 275 | 310 | 781 | 66 | 136 | 4 | 14 |
| org.apache.kafka.clients | | 81% | | 78% | 127 | 573 | 189 | 1,385 | 24 | 282 | 0 | 29 |
| org.apache.kafka.common.config | | 78% | | 72% | 148 | 458 | 195 | 987 | 55 | 213 | 7 | 34 |
| org.apache.kafka.common.security.kerberos | | 34% | | 33% | 88 | 133 | 255 | 391 | 25 | 47 | 3 | 8 |
| org.apache.kafka.common.network | | 86% | | 75% | 194 | 691 | 201 | 1,587 | 41 | 302 | 1 | 29 |
| org.apache.kafka.common.protocol | | 73% | | 41% | 56 | 130 | 185 | 512 | 15 | 72 | 1 | 11 |
| org.apache.kafka.clients.producer.internals | | 90% | | 84% | 153 | 899 | 154 | 2,094 | 19 | 378 | 1 | 37 |
| org.apache.kafka.common.security.authenticator | | 82% | | 72% | 100 | 339 | 129 | 854 | 9 | 136 | 0 | 16 |
| org.apache.kafka.common | | 70% | | 54% | 95 | 220 | 91 | 398 | 32 | 123 | 3 | 17 |
| org.apache.kafka.common.errors | | 51% | | n/a | 108 | 235 | 220 | 472 | 108 | 235 | 0 | 98 |
| org.apache.kafka.common.metrics | | 80% | | 73% | 47 | 220 | 91 | 510 | 11 | 130 | 0 | 13 |
| org.apache.kafka.common.security.scram.internals | | 85% | | 73% | 39 | 184 | 68 | 478 | 10 | 119 | 1 | 19 |
| org.apache.kafka.common.security.ssl | | 80% | | 75% | 53 | 165 | 53 | 382 | 7 | 60 | 1 | 8 |
| org.apache.kafka.common.resource | | 65% | | 52% | 61 | 116 | 55 | 154 | 23 | 54 | 2 | 6 |
| org.apache.kafka.common.security.oauthbearer.internals | | 79% | | 65% | 48 | 140 | 61 | 305 | 7 | 73 | 0 | 13 |
| org.apache.kafka.common.metrics.stats | | 83% | | 77% | 36 | 167 | 52 | 367 | 11 | 99 | 4 | 29 |
| org.apache.kafka.common.security.token.delegation | | 11% | | 0% | 35 | 41 | 43 | 56 | 15 | 21 | 1 | 2 |
| org.apache.kafka.common.serialization | | 77% | | 70% | 45 | 152 | 63 | 269 | 20 | 92 | 3 | 37 |
| org.apache.kafka.common.replica | | 34% | | 15% | 36 | 52 | 31 | 67 | 15 | 29 | 1 | 6 |

Then I use Emma with Intellij IDEA. Here are the results:



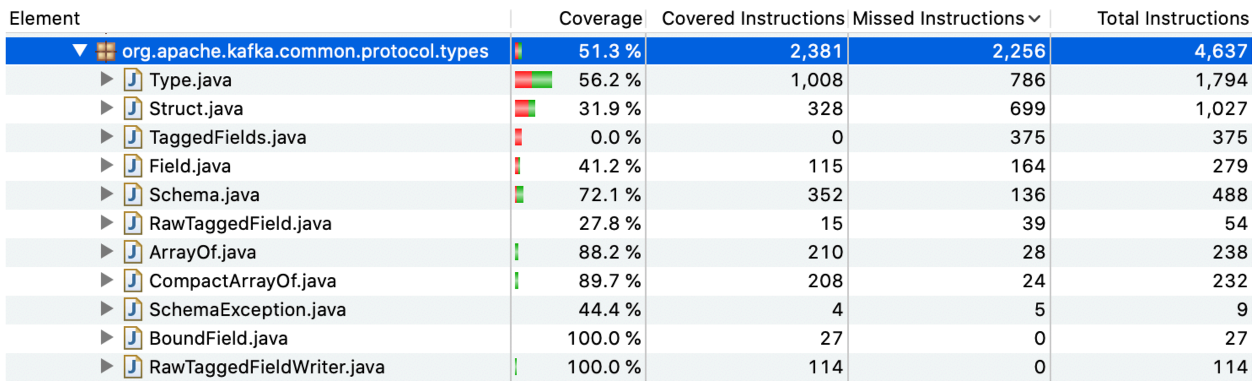
2% classes, 1% lines covered in package 'org.apache.kafka'

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| clients | 2% (8/355) | 1% (31/2418) | 1% (164/12440) |
| common | 2% (20/846) | 1% (99/5269) | 2% (478/20298) |
| server | 0% (0/9) | 0% (0/36) | 0% (0/78) |

I only show the code coverage in the org.apache.kafka package, which is the source folder of the client module.

Since Emma can't see which line or branch is missing during the execution. I use Eclipse to execute certain test case to see which lines are missing.

In the package org.apache.kafka.common.protocol.types package. There are several test cases. Here is the coverage detail:

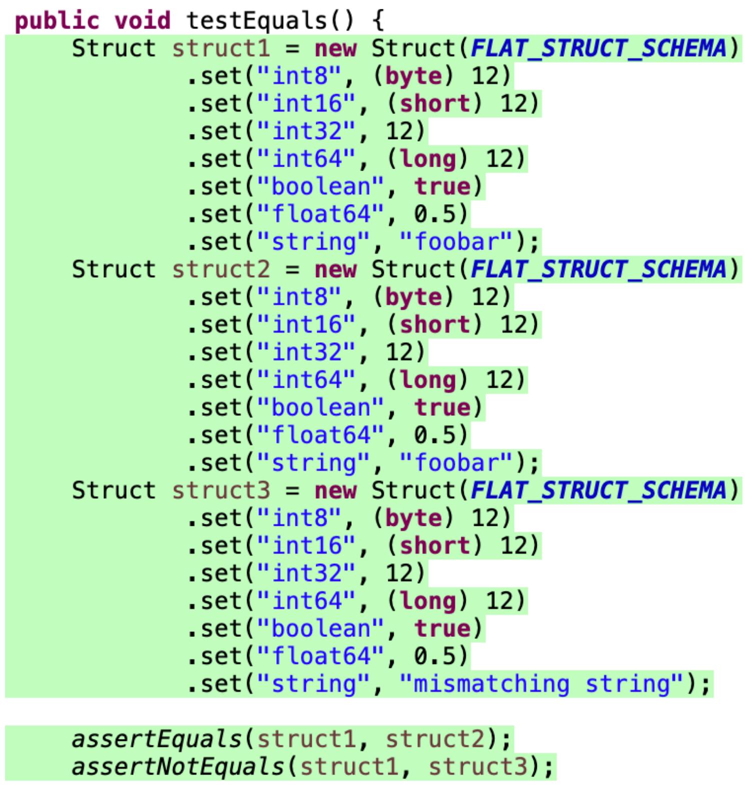


| Element | Coverage | Covered Instructions | Missed Instructions | Total Instructions |
|---|---|---|---|---|
| org.apache.kafka.common.protocol.types | 51.3 % | 2,381 | 2,256 | 4,637 |
| Type.java | 56.2 % | 1,008 | 786 | 1,794 |
| Struct.java | 31.9 % | 328 | 699 | 1,027 |
| TaggedFields.java | 0.0 % | 0 | 375 | 375 |
| Field.java | 41.2 % | 115 | 164 | 279 |
| Schema.java | 72.1 % | 352 | 136 | 488 |
| RawTaggedField.java | 27.8 % | 15 | 39 | 54 |
| ArrayOf.java | 88.2 % | 210 | 28 | 238 |
| CompactArrayOf.java | 89.7 % | 208 | 24 | 232 |
| SchemaException.java | 44.4 % | 4 | 5 | 9 |
| BoundField.java | 100.0 % | 27 | 0 | 27 |
| RawTaggedFieldWriter.java | 100.0 % | 114 | 0 | 114 |

I check the Struct.java file for coverage detail. This class defines many different get/set methods for different field in the struct of Kafka DocumentedTypes. Some get/set methods haven't been called in the test cases so there is no coverage on them.

## 3. Coverage with New Test Cases

I didn't write new test cases. Instead I modified the StructTest.java file.

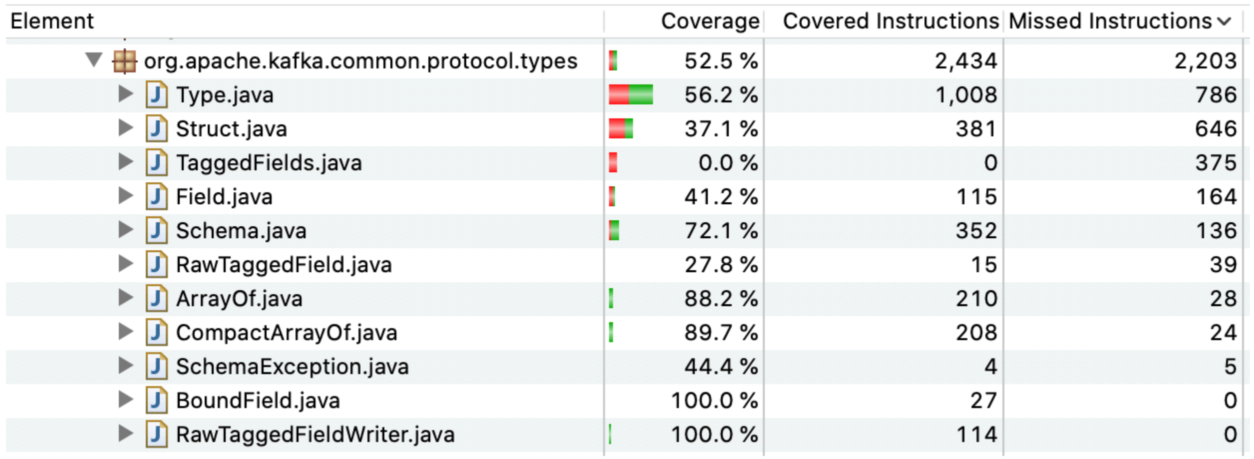


```
public void testEquals() {
    Struct struct1 = new Struct(FLAT_STRUCT_SCHEMA)
            .set("int8", (byte) 12)
            .set("int16", (short) 12)
            .set("int32", 12)
            .set("int64", (long) 12)
            .set("boolean", true)
            .set("float64", 0.5)
            .set("string", "foobar");
    Struct struct2 = new Struct(FLAT_STRUCT_SCHEMA)
            .set("int8", (byte) 12)
            .set("int16", (short) 12)
            .set("int32", 12)
            .set("int64", (long) 12)
            .set("boolean", true)
            .set("float64", 0.5)
            .set("string", "foobar");
    Struct struct3 = new Struct(FLAT_STRUCT_SCHEMA)
            .set("int8", (byte) 12)
            .set("int16", (short) 12)
            .set("int32", 12)
            .set("int64", (long) 12)
            .set("boolean", true)
            .set("float64", 0.5)
            .set("string", "mismatching string");

    assertEquals(struct1, struct2);
    assertNotEquals(struct1, struct3);
```

This test case initiates three structs with different fields. It uses set method to set the value for each field. However in the Struct.java file it offers a method called setIfExist() for different fields. This method has never been called. So I called this method to set the field. For example:

```
Object[] array = {(byte) 1, (byte) 2};

struct1 = new Struct(NESTED_SCHEMA)
        .setIfExists("array", array)
        .set("nested", new Struct(NESTED_CHILD_SCHEMA).set("int8", (byte)
Object[] array2 = {(byte) 1, (byte) 2};
struct2 = new Struct(NESTED_SCHEMA)
        .set("array", array2)
        .set("nested", new Struct(NESTED_CHILD_SCHEMA).set("int8", (byte)
Object[] array3 = {(byte) 1, (byte) 2, (byte) 3};
struct3 = new Struct(NESTED_SCHEMA)
        .set("array", array3)
        .set("nested", new Struct(NESTED_CHILD_SCHEMA).set("int8", (byte)

assertEquals(struct1, struct2);
assertNotEquals(struct1, struct3);
```

Here is the result after I made these changes:



| Element | Coverage | Covered Instructions | Missed Instructions |
|---|---|---|---|
| org.apache.kafka.common.protocol.types | 52.5 % | 2,434 | 2,203 |
| Type.java | 56.2 % | 1,008 | 786 |
| Struct.java | 37.1 % | 381 | 646 |
| TaggedFields.java | 0.0 % | 0 | 375 |
| Field.java | 41.2 % | 115 | 164 |
| Schema.java | 72.1 % | 352 | 136 |
| RawTaggedField.java | 27.8 % | 15 | 39 |
| ArrayOf.java | 88.2 % | 210 | 28 |
| CompactArrayOf.java | 89.7 % | 208 | 24 |
| SchemaException.java | 44.4 % | 4 | 5 |
| BoundField.java | 100.0 % | 27 | 0 |
| RawTaggedFieldWriter.java | 100.0 % | 114 | 0 |

The Covered Instruction increases from 328 to 381. I could improve it a lot but it's just the same step as above. I just need to write the same test cases for setIfExist() method for every field.

## 3.3 Testable Design

### 1.What makes Testable Design

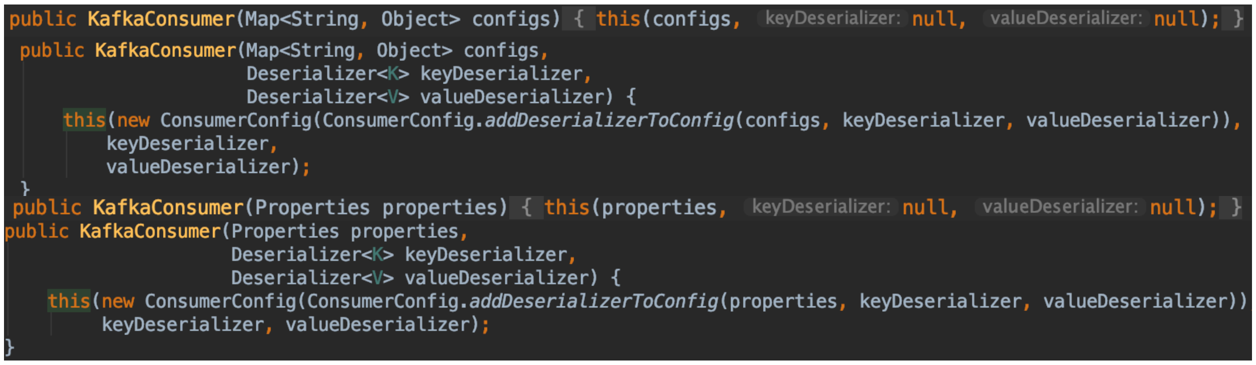
The fundamental value proposition of testable design is being better able to test code. Quoting Roy Osherove, the author of *The Art of Unit Testing with Examples* *in .NET* (Manning Publications, 2009), “a given piece of code should be easy and quick to write a unit test against.”

In order to make the program more testable, there are some certain ways to improve it. For example, we should avoid complex private method. Private method usually can’t be tested because in Java it can’t be accessed by other classes. So there is no way to test it. We should also avoid static method.  For functionality that has side effects or that has randomness (anything that we may want to stub out), static makes it difficult or impossible to test. We better have a more simple constructor especially if we want to change something in the test case. We can move something in the constructor to other methods.

---

## 2.Difficult feature in Kafka for testing

In Apache Kafka, the Consumer class is KafkaConsumer class in the client module. In this class file, there are several constructor with different parameters.



```
public KafkaConsumer(Map<String, Object> configs) { this(configs,  keyDeserializer: null,  valueDeserializer: null); }
 public KafkaConsumer(Map<String, Object> configs,
                      Deserializer<K> keyDeserializer,
                      Deserializer<V> valueDeserializer) {
     this(new ConsumerConfig(ConsumerConfig.addDeserializerToConfig(configs, keyDeserializer, valueDeserializer)),
         keyDeserializer,
         valueDeserializer);
 }
public KafkaConsumer(Properties properties) { this(properties,  keyDeserializer: null,  valueDeserializer: null); }
public KafkaConsumer(Properties properties,
                     Deserializer<K> keyDeserializer,
                     Deserializer<V> valueDeserializer) {
    this(new ConsumerConfig(ConsumerConfig.addDeserializerToConfig(properties, keyDeserializer, valueDeserializer))
        keyDeserializer, valueDeserializer);
}
```

All these methods eventually call a private constructor to create the instance.

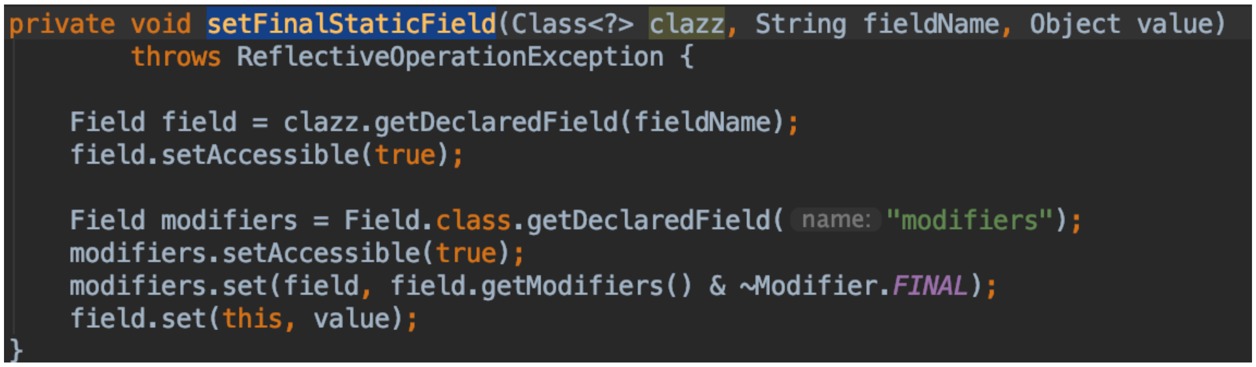
In this constructor, it uses ConsumerConfig as a parameter and initialize all the fields in KafkaConsumer with the ConsumerConfig’s field. All the fields in the ConsumerConfig class is static, which can be changed in the test. However since the constructor is a private method. We can not test it. For example, if some fields in the Consumer are null, there could be a NullPointerException in the constructor. Surely we want to make sure the constructor has no such bugs.

---

## 3.New Version and Testing

In order to fix it, I write another class called KafkaConsumerNew. I write a method called initialize() to initialize all the fields. And in the original constructor I call this method to initialize all the fields. The difficulty I’m facing is that nearly all the fields in the KafkaConsumer are final fields, which can not be updated and can only be assigned value in the constructor.

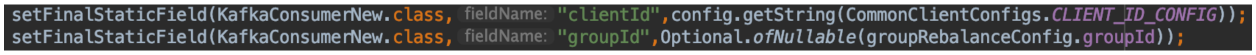
In order to change the final field. I write a new method called setFinalStaticField:



```
private void setFinalStaticField(Class<?> clazz, String fieldName, Object value)
        throws ReflectiveOperationException {

    Field field = clazz.getDeclaredField(fieldName);
    field.setAccessible(true);

    Field modifiers = Field.class.getDeclaredField( name: "modifiers");
    modifiers.setAccessible(true);
    modifiers.set(field, field.getModifiers() & ~Modifier.FINAL);
    field.set(this, value);
}
```

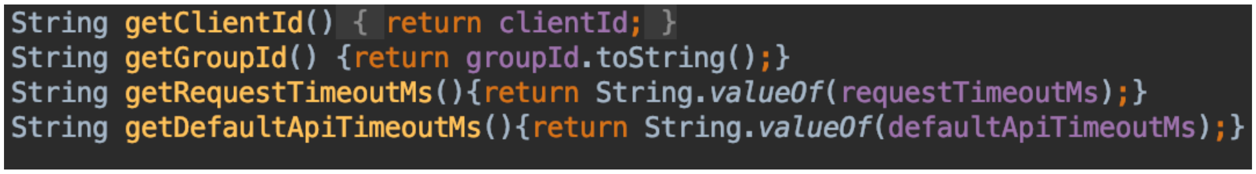
This method uses Java reflection to get the final fields and calls setAccessible() method so that the final field can be changed. At the end it calls the field.set() method to set the corresponding field to the value you want.

In the initialize() method I call this setFinalStaticField() method several times to set the corresponding field. The method takes values in the CommonClientConfigs class to set the fields.



```
setFinalStaticField(KafkaConsumerNew.class, fieldName: "clientId",config.getString(CommonClientConfigs.CLIENT_ID_CONFIG));
setFinalStaticField(KafkaConsumerNew.class, fieldName: "groupId",Optional.ofNullable(groupRebalanceConfig.groupId));
```

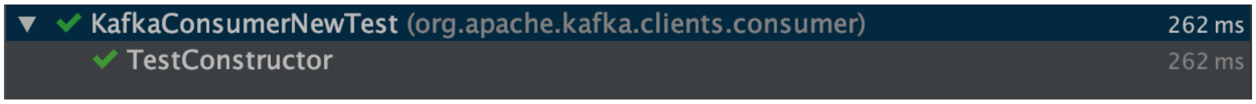
In order to get the fields value for testing, I write some get methods to get the field value.



```
String getClientId() { return clientId; }
String getGroupId() {return groupId.toString();}
String getRequestTimeoutMs(){return String.valueOf(requestTimeoutMs);}
String getDefaultApiTimeoutMs(){return String.valueOf(defaultApiTimeoutMs);}
```

Here are the test cases:

I use the properties class to set some certain config values for the fields in KafkaConsumerNew. The KafkaConsumerNew constructor takes the property as a parameter to set the CommonConsumerConfig. Then in the initialize() method the setFinalStaticField() method will update the field using the value in the CommonConsumerConfig class. Here are the results:



```
▼ ✓ KafkaConsumerNewTest (org.apache.kafka.clients.consumer)      262 ms
      ✓ TestConstructor                                            262 ms
```

## 3.4 Mocking

Mocking is useful for interaction testing. Mock Object is a fake object that decides whether a unit test has passed or failed by watching interactions between objects. In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.

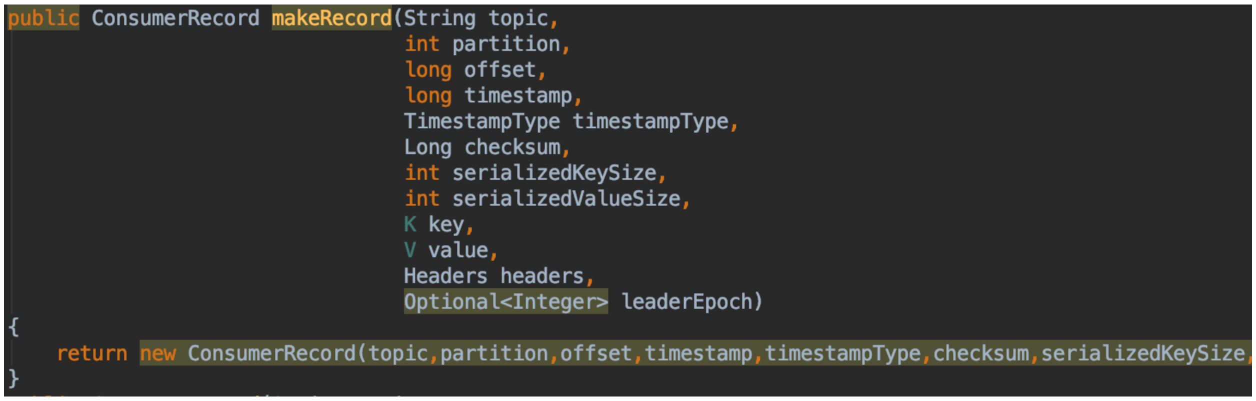
When doing unit test, we want to test functionality in isolation, which means we want to eliminate side effects from other classes or systems. We can achieve this goal through several techniques. With mock object, we can define some certain method calls. It can be configured to perform a certain behavior during a test. There are many tools support mock object such as Mockito, so it is preferred in many scenarios. 1

---

### 1.Feature could be mocked

In Kafka, each Consumer has a list of topics that it's interested in. These information is held in the ConsumerRecords class. Each ConsumerRecords instance has a list of ConsumerRecord objects. In order to test ConsumerRecords class alone with no side affect from the ConsumerRecord class. I created a mock object of ConsumerRecord.

In order to do so, I have to modify some codes in the ConsumerRecord class. Because the constructor in ConsumerRecord can not be mocked. So I add this makeRecord() method. In this method I call the constructor so I only need to mock this method.



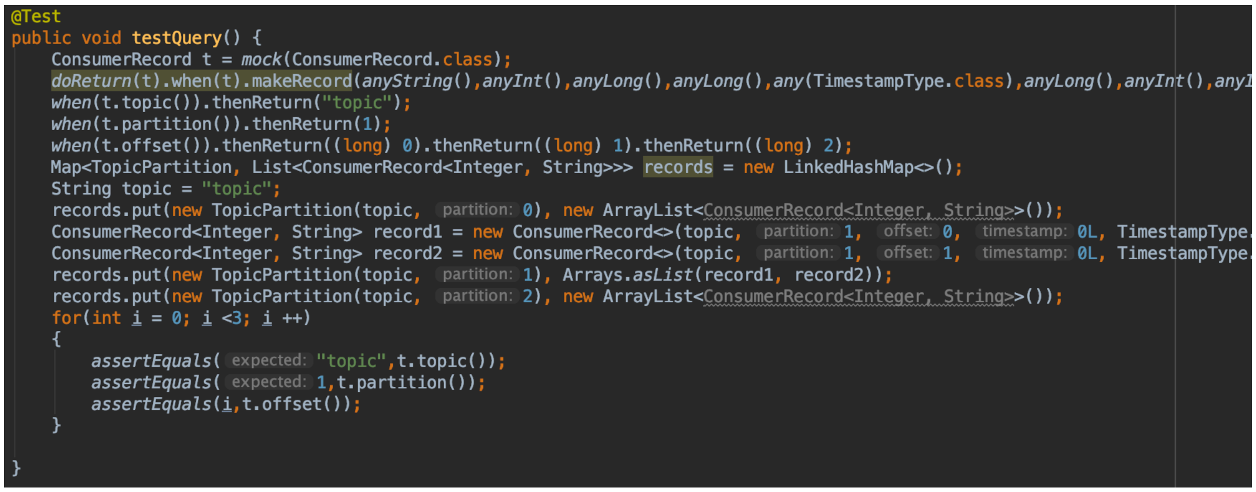
```
public ConsumerRecord makeRecord(String topic,
                                  int partition,
                                  long offset,
                                  long timestamp,
                                  TimestampType timestampType,
                                  Long checksum,
                                  int serializedKeySize,
                                  int serializedValueSize,
                                  K key,
                                  V value,
                                  Headers headers,
                                  Optional<Integer> leaderEpoch)
{
    return new ConsumerRecord(topic,partition,offset,timestamp,timestampType,checksum,serializedKeySize,
}
```

With mock object of ConsumerRecord, I can test ConsumerRecords class with no idea of whether the ConsumerRecord class is right or not.

---

### 2.New Test Case and results

I write a new test case called ConsumerRecordsTestMock.

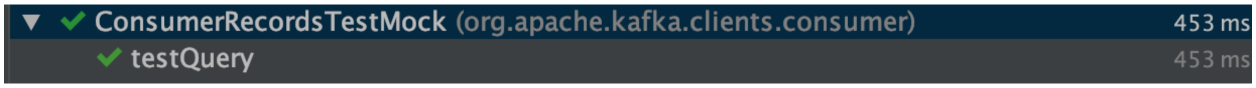


```
@Test
public void testQuery() {
    ConsumerRecord t = mock(ConsumerRecord.class);
    doReturn(t).when(t).makeRecord(anyString(),anyInt(),anyLong(),anyLong(),any(TimestampType.class),anyLong(),anyInt(),anyI
    when(t.topic()).thenReturn("topic");
    when(t.partition()).thenReturn(1);
    when(t.offset()).thenReturn((long) 0).thenReturn((long) 1).thenReturn((long) 2);
    Map<TopicPartition, List<ConsumerRecord<Integer, String>>> records = new LinkedHashMap<>();
    String topic = "topic";
    records.put(new TopicPartition(topic,  partition: 0), new ArrayList<ConsumerRecord<Integer, String>>());
    ConsumerRecord<Integer, String> record1 = new ConsumerRecord<>(topic,  partition: 1,  offset: 0,  timestamp: 0L, TimestampType.
    ConsumerRecord<Integer, String> record2 = new ConsumerRecord<>(topic,  partition: 1,  offset: 1,  timestamp: 0L, TimestampType.
    records.put(new TopicPartition(topic,  partition: 1), Arrays.asList(record1, record2));
    records.put(new TopicPartition(topic,  partition: 2), new ArrayList<ConsumerRecord<Integer, String>>());
    for(int i = 0; i <3; i ++)
    {
        assertEquals( expected: "topic",t.topic());
        assertEquals( expected: 1,t.partition());
        assertEquals(i,t.offset());
    }

}
```

In this test case, I create a mock object of ConsumerRecord named t. When the makeRecord method was called, it returns instance t.

The method topic(), partition() and offset() are used to return the value of corresponding fields. I use mock to return some certain values for testing. And for offset value, it is the location of ConsumerRecord in the ConsumerRecords list. So I need to return different values. So I add several thenReturn() after when(t.offset()).

Here is the result:



```
▼ ✓ ConsumerRecordsTestMock (org.apache.kafka.clients.consumer)    453 ms
      ✓ testQuery                                                   453 ms
```

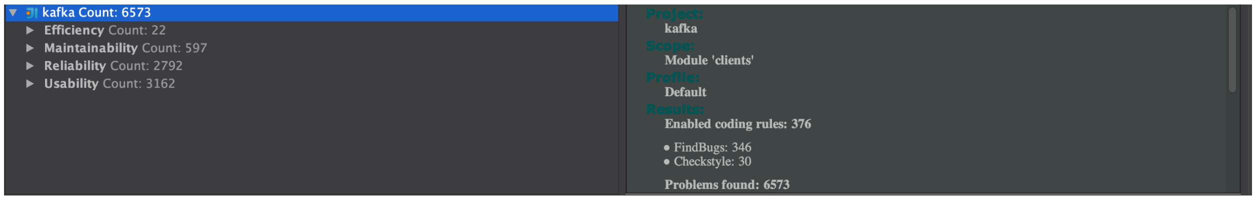
## 3.5 Static Analyzers

### 1.Static Analysis Tools

Code Review is important in software development. It can help us find bugs at the early stage of software development process. This method is done without running the program so it is called static analysis. Static Analyzers can help us do code review automatically so we could save much more time.

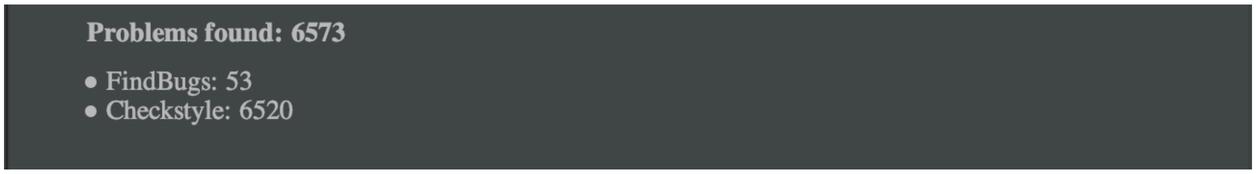
Different static analyzers have different focuses. For this project I choose FindBugs and Checkstyle. Checkstyles can check many aspects of the source code. It can find class design problems, method design problems, code layout and formatting issues. FindBugs can find potential bugs in the software.

I use the above tools on the Client module of Apache Kafka. I'm using Intellij IDEA with QA-FindBugs and QA-Checkstyle plugins. Here are the overall results:



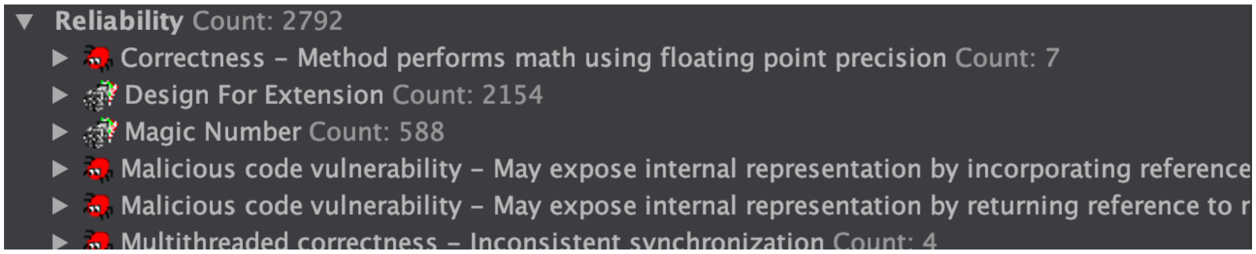
```
▼ kafka Count: 6573
  ▶ Efficiency Count: 22
  ▶ Maintainability Count: 597
  ▶ Reliability Count: 2792
  ▶ Usability Count: 3162
```

```
Project:
    kafka
Scope:
    Module 'clients'
Profile:
    Default
Results:
    Enabled coding rules: 376

    • FindBugs: 346
    • Checkstyle: 30

    Problems found: 6573
```



```
Problems found: 6573

• FindBugs: 53
• Checkstyle: 6520
```

In the followed picture, the red icon represents FindBug results and the other icon represents Checkstyle results.

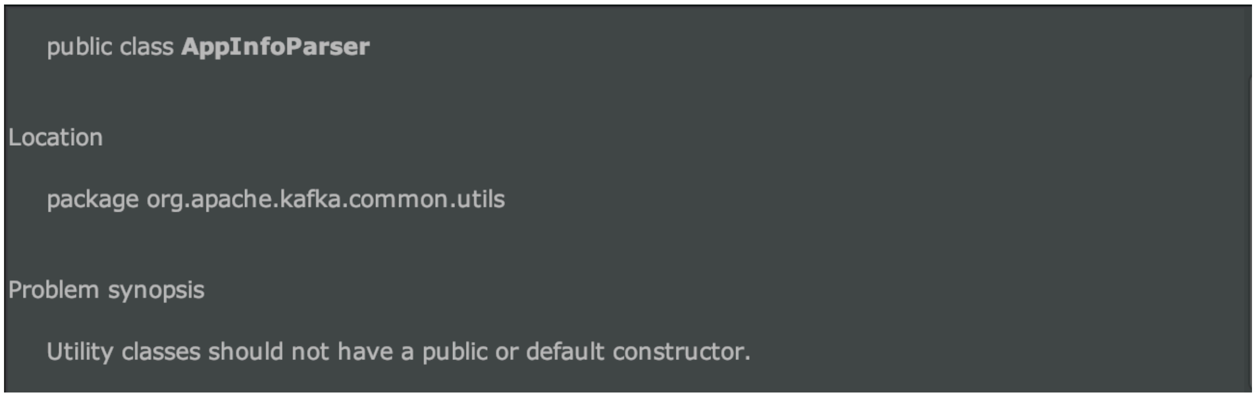


```
▼ Reliability Count: 2792
  ▶ Correctness – Method performs math using floating point precision Count: 7
  ▶ Design For Extension Count: 2154
  ▶ Magic Number Count: 588
  ▶ Malicious code vulnerability – May expose internal representation by incorporating reference
  ▶ Malicious code vulnerability – May expose internal representation by returning reference to r
  ▶ Multithreaded correctness – Inconsistent synchronization Count: 4
```

The results are divided into four categories: Efficiency, Maintainability, Reliability and Usability. These categories can be set in the setting page. These are coding rules for the projects. From the results, some code may violate these rules.

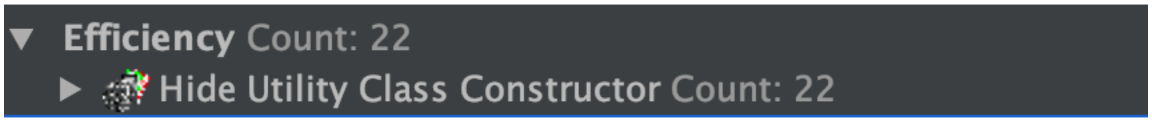
## 2.Checkstyle

Here are some warnings provided by FindBug:



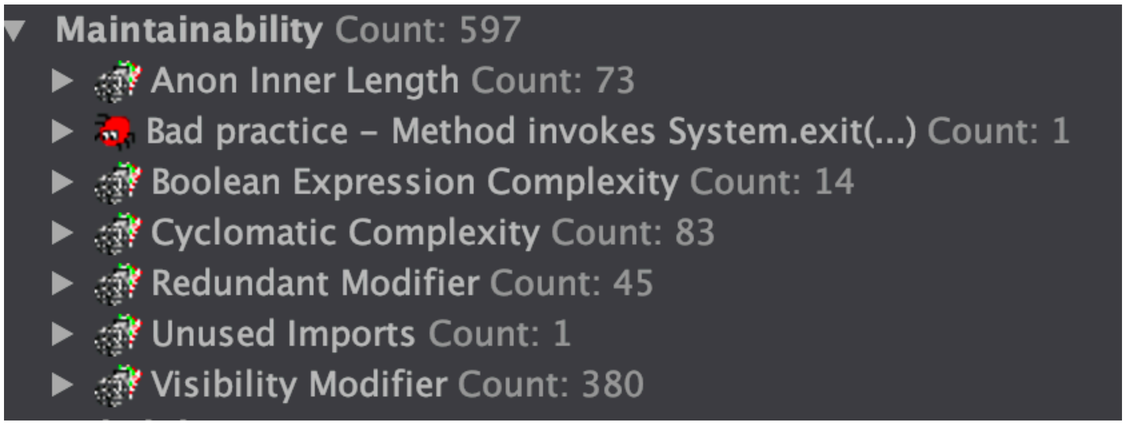
```
    public class AppInfoParser

Location

    package org.apache.kafka.common.utils

Problem synopsis

    Utility classes should not have a public or default constructor.
```

This problem was found in several classes in the org.apache.kafka.common.utils package. Since most Utility classes are static so there is no need to instantiate a instance.


▼ Efficiency Count: 22
▶ Hide Utility Class Constructor Count: 22

However for this problem I look into the code and there aren't any constructors in these classes. These are false warnings.

Here are some other kinds of warnings:


▼ Maintainability Count: 597
▶ Anon Inner Length Count: 73
▶ Bad practice – Method invokes System.exit(...) Count: 1
▶ Boolean Expression Complexity Count: 14
▶ Cyclomatic Complexity Count: 83
▶ Redundant Modifier Count: 45
▶ Unused Imports Count: 1
▶ Visibility Modifier Count: 380

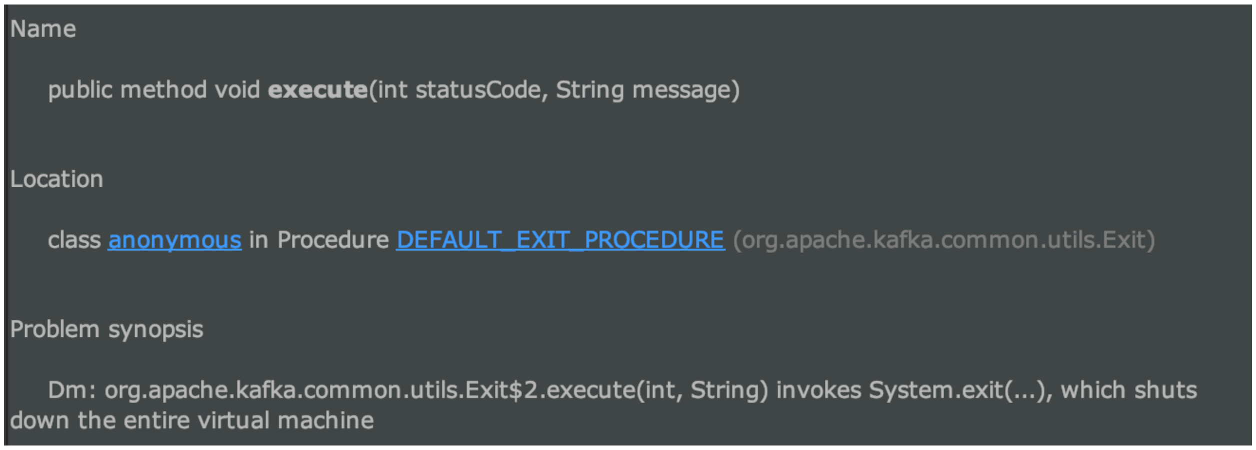
For the "Anon Inner Length" problem it means in the coding rules, anonymous

Inner classes should not have more than 20 lines. But I have 73 such classes exceed 20 lines.

For the "Visibility Modifier" problem it means some fields in the code are are modified by "private" or "protected" modifier. It should have accessor methods.

## 3. FindBugs

Here are some warnings provided by FindBug:


Name

public method void **execute**(int statusCode, String message)

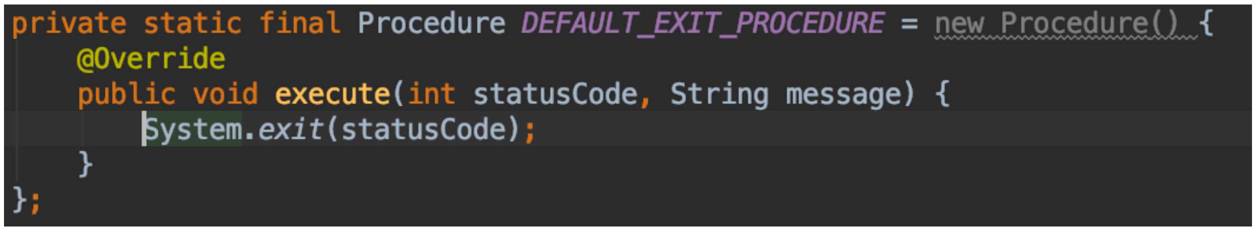
Location

class anonymous in Procedure DEFAULT_EXIT_PROCEDURE (org.apache.kafka.common.utils.Exit)

Problem synopsis

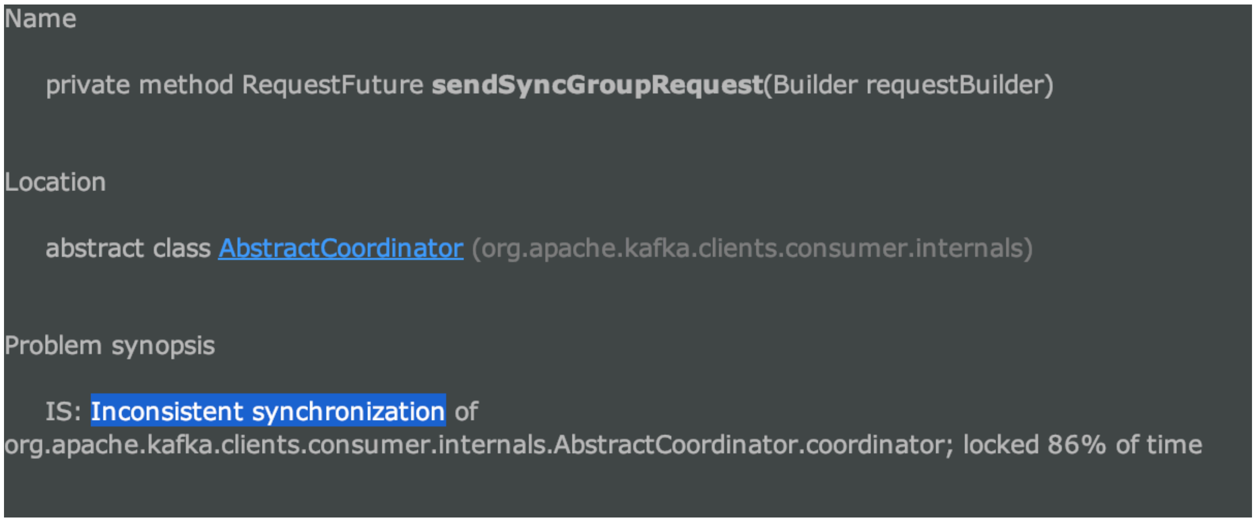
Dm: org.apache.kafka.common.utils.Exit$2.execute(int, String) invokes System.exit(...), which shuts down the entire virtual machine

This problem means in the code it calls System.exit() method which will shut down the entire system. However the code bellow is used to shut down the system. So this is not a bug.



```
private static final Procedure DEFAULT_EXIT_PROCEDURE = new Procedure() {
    @Override
    public void execute(int statusCode, String message) {
        System.exit(statusCode);
    }
};
```

Here is another example:



Name

private method RequestFuture **sendSyncGroupRequest**(Builder requestBuilder)

Location

abstract class AbstractCoordinator (org.apache.kafka.clients.consumer.internals)

Problem synopsis

IS: Inconsistent synchronization of org.apache.kafka.clients.consumer.internals.AbstractCoordinator.coordinator; locked 86% of time

This problem means if we want to synchronize some field or method, we should synchronize it on every usage. This will make sure the correctness.

## 4. Conclusion

These two tools provide different warnings. Checkstyle basically checks design decisions in the code and FindBug checks if there are any bugs in the code.

Checkstyle can help us code more efficiently and easy to understand. It not help us correct the code but improve the code. FindBug fins potential bugs in the code. It might not be a bug in some scenarios but it will cause some issues such as security issue or bugs in other scenarios.

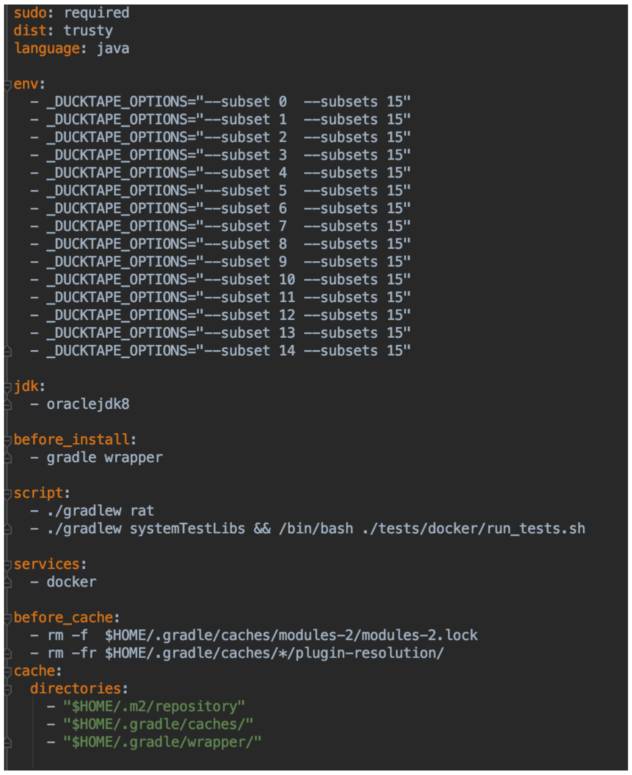
## 3.6 Continuous Integration

### 1. Continuous Integration

In modern software engineering, continuous integration is the practice of merging all developers' working copies to a shared mainline several times a day. It makes sure that the latest version of the code always builds and passes all tests. It has lots of advantages. It allows us to find bugs earlier; It makes it easier for developer to estimate development time; Developer can get feedback more quickly.

### 2. .travis.yaml file

Apache Kafka supports TravisCI. Here is the .travis.yaml file:



```yaml
sudo: required
dist: trusty
language: java

env:
  - _DUCKTAPE_OPTIONS="--subset 0  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 1  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 2  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 3  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 4  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 5  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 6  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 7  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 8  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 9  --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 10 --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 11 --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 12 --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 13 --subsets 15"
  - _DUCKTAPE_OPTIONS="--subset 14 --subsets 15"

jdk:
  - oraclejdk8

before_install:
  - gradle wrapper

script:
  - ./gradlew rat
  - ./gradlew systemTestLibs && /bin/bash ./tests/docker/run_tests.sh

services:
  - docker

before_cache:
  - rm -f  $HOME/.gradle/caches/modules-2/modules-2.lock
  - rm -fr $HOME/.gradle/caches/*/plugin-resolution/
cache:
  directories:
    - "$HOME/.m2/repository"
    - "$HOME/.gradle/caches/"
    - "$HOME/.gradle/wrapper/"
```

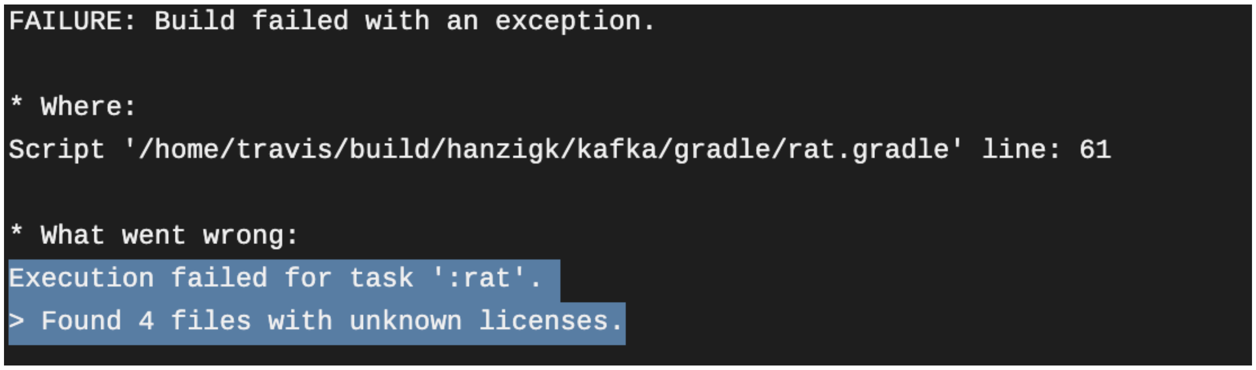
However with this yaml file I can not build Kafka correctly. It is due to the Gradle version on the virtual machine is not compatible with the version specified in the project. So I use ./gradlew instead of gradle command.

In this file, it specified the language is Java and use jdk8.0. It also specified that the virtue machine should run linux trusty.

Then in the before_install part it uses gradle wrapper to build the project. Then it runs rat task and run the tests. Then at last it clears the cache.

## 3. Build with TravisCI

I encountered a problem with task rat. Rat is implemented by the Apache RAT (Release Audit Tool) Gradle Plugin. It checks license for existing files. Since I added some classes and test cases this task will fail.



```
FAILURE: Build failed with an exception.

* Where:
Script '/home/travis/build/hanzigk/kafka/gradle/rat.gradle' line: 61

* What went wrong:
Execution failed for task ':rat'.
> Found 4 files with unknown licenses.
```

I haven't fix this problem yet. But I can assure the rest parts of the project are right. I will try to fix it in the future.