CITS5501 Project 2022 – model answers

Question A: Parser – ISP test design

One of your colleagues, Anastasia, is writing a test plan for the `Parser` class using input space partitioning. She suggests that the `Parser` class can be viewed as a function which takes as input a string (i.e., a document), and outputs a `List<Chunk>`. She proposes to use the interface-based approach to input space partitioning, with the aim of achieving Base Choice Coverage. She suggests the following, single characteristic for the input string:

- Is the string empty? (This divides strings into two partitions, empty vs non-empty.)

For empty strings, there is only one possible value. For non-empty strings, Anastasia suggests a simple, two-line string (`"This is a sample document.\nIt contains two lines\n"`) as the base choice, as well as several non-base choices:

- A very short document (just the string `"This"`)
- A document containing non-English Unicode characters
- A document containing a hundred lines (consisting of the base choice string, repeated 50 times)

Answer the following question in your report:

- Is your colleague's approach to applying the ISP technique an appropriate one? Why or why not? How would you apply the technique? You should give each of the steps involved in applying ISP, except that for the last step (refining into test values), you need give only 3 test cases. (10 marks)
- Your 3 test cases may *not* include the case where the input string is the empty document.

Competent model solution

Fuctionality- vs interface-based modelling

The approach suggested by Anastasia is not ideal. Often, a functionality-based approach to ISP (which considers the intended meaning and functionality of a function and its arguments) will give better results than Anastasia's proposed interface-based approach (which mechanically considers each parameter in terms of its type), and that is the case here. While it is possible to view the input can be viewed as "just a string" for testing purposes, we are likely to obtain a larger number of more useful characteristics if test designers bear in mind that the string is intended to be parsed into a list of docchunks and codechunks. Then we can create characteristics based on that functionality, and more thoroughly test the Parser class.

Modelling the Parser as a function

Anastasia's suggestion of modelling the class as a function from Strings to lists of Chunks seems reasonable:

- As long as the model is useful, it's acceptable to model methods, classes, components or entire systems as functions from an input domain to an output.
- Any model is a simplification of what is being modelled. In this case, the actual type used to construct a `Parser` is a `BufferedReader`. But all we ever do with the `BufferedReader` is read its contents using `readLines`, so we can simplify our model by assuming we have the contents of the `BufferedReader` as a `String`.

[Answers might in addition to or instead of the above, adopt or discuss one of the following alternatives – but Anastasia's "string -> list" model is the preferred one.]

Possible alternatives to modelling the class as a function from Strings to lists of Chunks are as follows:

- One alternative would be to model the domain as a list of `Strings` (with each element of the list representing a single line). But a counterargument to that is that we might want to write tests which depend on exactly where newline characters are placed, so a `String` is probably the better option.
- Another alternative would be to model the `Parser` constructor, the `parse` method, and the `getChunks` method as distinct functions (Anastasia's model combines them all into one). It is probably true that we should design tests for the constructor on its own (though we won't do that here), so for the constructor, this argument has some weight; but `getChunks` is such a trivial method that there is probably no point writing tests for it. (A colleague performing a code review would be enough.)
- It may be worth noting that Anastasia perhaps could have got better results by modelling the parser class in terms of *syntax*, than as a function – but we will restrict ourselves here to a discussion of ISP.

Characteristics

Anastasia's characteristic is fine, as it goes – it is a useful characteristic, and does indeed divide the input domain into distinct partitions that cover the domain – but should be augmented with other characteristics, as described below.

Test values

Anastasia's test values don't seem to have been chosen in any especially principled way; we can do better with a functionality-based approach (described below).

Applying ISP

1. The first step in ISP is to identify testable functions. Here, we consider the functionality of the Parser class as a whole as being a function from the domain of `String`s, to the set of lists of `Chunk`s. We'll call this the "parser" function, for convenience.

2. We then need to come up with characteristics which partition the input space of the parser function.

   Some possible partitions are listed below.

   We start by considering characteristics where there are at most one appearance of the start and end markers – these represent very simple documents, but if the implementer has introduced a defect, it will hopefully turn up in one of these.

   a. Does the input string contain a single instance of the start marker sequence, `"=<<<"`? (Gives 2 partitions)
      - If so, we can consider sub-partitions. Is the start marker sequence
        - … at the start of the document? (Gives 2 partitions)
        - … at the start of a line? (Gives 2 partitions)
        - … at the end of the document? (Gives 2 partitions)
        - … at the end of a line? (Gives 2 partitions)
        - … on a line by itself? (Gives 2 partitions)

   b. Does the input string contain a single instance of the end marker sequence, `">>>"`?
      - If so, we can consider sub-partitions – see the previous characteristic (a), since we can use the same ones.

   The previous two characteristics will include various badly-formed start and end markers (e.g. a line like `"=<<<something"`, which is not a valid start marker). In the interests of brevity, the remaining characteristics will focus on only well-formed start and end markers, but in a thorough test design, we'd consider badly-formed ones as well.

   c. Does the input string contain a single start marker line and a single end marker line?
      - If so, we can consider sub-partitions; do they appear with
        i. … the start marker line first, and then the end marker line? Or,
        ii. … the end marker line first, and then the start marker line?

More sub-partitions and variations can be created similar to the previous ones.

But they become very repetitive. One useful way of simplifying our characteristics is to note that many of them are to do with where a start or end sequence sits inside some "container" of text – either a document or a line. Consider the following alternative approach to characteristics:

Assuming we're considering the case where we have a non-empty document, and it contains a single start or end sequence (possibly malformed, due to not being on a line of its own):

We can model the input space by imagining that it consists of start or end sequences (that is, `"=<<<"` or `">>>"` sitting inside a "container" of text, where the container is either a *document* or a *line*.

In all cases, there's the possibility of some text appearing *before* the marker, and some text appearing *after* the marker.

i.e.: `<start_text> <marker> <end_text>`

We can then partition based on what the *container* is:

- Is the container
  - ... a line?
  - ... the whole input document?

And we can partition each of `<start_text>` and `<end_text>` based on their length – for instance, whether they have

- 0 characters (i.e., they are empty)
- 1 character
- more than character

This gives us a set of partitions and sub-partitions like the following:

- Is the container a *line* or a *document*? (gives 2 partitions)
  - In each case, we can introduce sub-partitions based on the size of the `<start_text>` and `<end_text>` – 3 partitions each.

For cases where we have both a start marker and an end marker, we can regard the container as consisting of:

```
<start_text> <start_marker> <middle_text> <end_marker>
<end_text>
```

Once again, we can create partitions based on whether the container is a *line* or a *document*, and we can partition `<start_text>`, `<middle_text>` and `<end_text>` based on their size, the same as before.

The final steps of ISP are to choose partitions and values from those partitions, and refine these into test values.

There are many possible solutions here depending on what partitions and values are chosen; here we show one example.

The test cases we will implement are as follows:

a. The case where the document contains a single start-sequence on a line, with empty text before it and one character after it, a space.

This would represent what could be a common possible error when creating an input document – accidentally adding some space after the the start marker (which will often be invisible in a text editor).

So our input document is: `"=<<<_"` (where the underscore is used here to represent a space).

The expected output is that we get a single `DocChunk` with contents `"=<<<_"`. (As an aside: we might want to discuss with a team lead or senior developer whether there should be any special handling of this situation. Would it be useful to users if the system emitted a *warning* when it encountered this case, telling them that they might have mis-typed? But for current testing purposes, we simply work off the spec as given.)

To be precise, we expect to get a `DocChunk` with contents `"=<<<_"` and `startLineNumber` 1.

b. The case where the document contains a well-formed start marker line and a well-formed end marker line, with non-empty text before the start marker line, between the two markers, and after the end marker line. The "non-empty text" here would represent a "base" or typical case.

   To refine this further into an exact test value, we just need to make a choice as to what the non-empty text will be.

   In this case, we will make our input document:

   ```
   "start\n=<<<\nmiddle\n>>>end\n"
   ```

   The expected output is to get the following list of chunks:

chunk no.	chunk type	contents	start line no.
0	doc	`"start\n"`	1
1	code	`"middle\n"`	2
2	doc	`"end\n"`	5

c. A variant on test case (b). This will represent a situation where we keep most values the same, and vary one of them to a non-base case. We will vary the "middle" text, and make it empty, instead.

   So our input document will be:

   ```
   "start\n=<<<\n>>>end\n"
   ```

   The expected output is to get the following list of chunks:

chunk no.	chunk type	contents	start line no.
0	doc	`"start\n"`	1
1	code	`"\n"`	2
2	doc	`"end\n"`	3

Poor answers

The following indicate a poor understanding of ISP:

- Stating that `Parser` cannot be tested using ISP, because it is a class, not a method. (This is incorrect; methods, classes, or whole systems can be tested using ISP.)
- Failing to consider the problem of "identifying a testable function" at all.
- Claiming that it's not permissible for Anastasia to have selected, as one of her test values, a document containing non-English Unicode characters. It is perfectly acceptable; nothing in the project description precludes documents from containing (for instance) Greek or Chinese or Cyrillic characters.
- Incorrectly assuming that requirements of the Processor class (e.g. that chunk types must strictly alternate) are relevant to ISP testing of the Parser class. (They are not.)
- Failing to point out the problems with an interface-based approach, or failing to adopt a functionality-based approach (and to state that that is the approach being adopted). The interface-based approach will only give a very few characteristics, at best, and is inappropriate here.
- Poor justification of a functionality-based approach.
- Claiming to apply an interface-based approach, but actually applying a functionality-based approach (or vice versa).
- Incorrectly stating that Anastasia's characteristic does not result in a partition. (It does.)
- Incorrectly stating that Anastasia's characteristic is not a useful or sensible one. (It is.)
- Incorrectly stating that Base Choice Coverage is not appropriate here. (It is; there is no reason not to use BSC.)
- Deriving tests based on what the sample implementation code in the `src` directory does, rather than what the project specs and JavaDoc comments say it should do. (This will be penalised heavily, since this error has now been pointed out numerous times in lectures.)
- Trying to derive tests based on undefined behaviour (such as the input document containing Unicode line-end characters such as "vertical tab" or "form feed"). By definition, it is impossible to write tests for undefined behaviour. This is a major misunderstanding of how testing works and one which has been discussed multiple times in lectures, so it will be penalized heavily.
- Nonsensical or incomprehensible answers or reasoning.

Question B: Parser – ISP test code

The `parsers.ParserTest` contains partial code for testing the `Parser` class. Implement your test cases from section A by adding them to the `ParserTest` class. (10 marks)

Competent model code

What your JUnit tests look like will depend on what tests you came up with for question

(A), but in general, they will be similar to the provided `testDoc` method, but with different text in the `inputTextLines` array.

(Since many of the tests will contain similar "setup" code like this, one could factor it out into its own method; but it's not required for this assignment.)

So test case (a) from section A would look like this:

```java
@Test
public void testMalformedStartLine() {
  String inputTextLines[] = { "=<<<_" };

  String inputText = StringUtilities.arrayLinesToString(inputTextLines);
  BufferedReader reader = StringUtilities.stringToBufferedReader(inputText);
  Parser parser = new Parser(reader);
  parser.parse();
  List<Chunk> chunks = parser.getChunks();

  Chunk expectedArr[] = { new DocChunk("=<<<_", 1) };
  List<Chunk> expected = Arrays.asList(expectedArr);

}
```

Poor answers

The following indicate minor issues in a student's understanding of how to develop a good test:

- Leaving `print` or `println` statements in a test. There are two situations here: either the println statements are providing useful information about the test, or they are not. If they are not, they should be removed – it's poor practice to leave in unnecessary code, *and* the `println`s will clutter the output when tests are run from the command-line. If the `println` statements *are* providing useful information about the test, then that's even worse, since a human needs to visually scan the output to obtain this information. The `println` statements should be turned into *assertions*, which the test framework can check automatically.

The following indicate major issues in a student's understanding of how to develop a good test, and will result in no or few marks being awarded.

- Failure to provide code that compiles. In general, this will result in 0 marks being awarded.
- Failure to properly carry out all parts of the "Arrange-Act-Assert" pattern, which requires you to set up the test environment ("Arrange"), invoke the behaviour under test ("Act"), and assert something about the result ("Assert").

  For example:

  – Failure to construct appropriate objects to run the test on (i.e., the "Arrange" part). This will likely result in compiler errors or meaningless tests.

  - Failure to invoke the desired behaviour (e.g., never actually invoking the `parse` method; this means the "Act" part of the pattern has been missed).
  - Failure to include a call to one of JUnit's `assert` methods. This makes the test of no use at all.

- Failing to implement the test cases described in section A. (A student is not double-penalised for poor choices of tests made in section A, however; as long as those tests are implemented properly, a student can get 100% for section B.)

Question C: Parser – syntax-based testing

Your supervisor, Yennefer, suggests that because input documents use a special syntax (start marker lines and end marker lines) to demarcate code chunks, syntax-based testing could be used to come up with tests, and assess the coverage level of existing tests.

In your report,

- Give a grammar which describes the syntax of input documents. Explain how it works and how you derived it. (7 marks)

Hint: You may assume you are allowed to use a special non-terminal symbol, "`<nonmarker_text>`". It expands into a string of arbitrary length which never contains the character sequence `"=<<<"` nor the character sequence `">>>"`.

Sample solution

We assume that the grammar is intended to describe input documents which will be consumed by the Parser class. (The Processor class, in any case, takes as input a `List<Chunk>`, not a document/`String`.) Therefore, we need not concern ourselves with the requirements of the Processor class (e.g. requiring that chunk types strictly alternate).

We use Extended BNF in our grammar – we use a question mark ("?") to indicate optional elements.

```
<newline> ::= "\n"

<code_start_marker> ::= "=<<<"

<code_end_marker> ::= ">>>"

<code_chunk> ::= <code_start_marker> <newline>
                 ( <nonmarker_text> <newline> )?
                 <code_end_marker> <newline>
```

```
<document> ::= ( (<nonmarker_text> <newline>)? <code_chunk> )* <nonmarker_text>?
```

The terminal symbols of our grammar are the strings on the right-hand side of the `<newline>`, `<code_start_marker>`, and `<code_end_marker>` rules. We define the non-terminal symbols `<newline>`, `<code_start_marker>`, and `<code_end_marker>` to make later rules easier to understand. We can combine those non-terminals into a `<code_chunk>`, which contains a start marker line and end marker line.

For a document to be valid, we need the start marker and end marker always to appear at the start of a line, and to be followed by a newline. Our `<code_chunk>` non-terminal mostly ensures that this is the case, except that it doesn't pin down the `<code_start_marker>` as being at the start of a line. But we handle that in our `<document>` non-terminal: valid `<document>`s will always have a `<code_chunk>` appearing at the very start of a document (which is fine), or else after a `<nonmarker_text> <newline>` sequence. In either case, the code chunk's `<code_start_marker>` will be in a valid position.

Our "start" symbol is `<document>`, which can be seen as a (possibly empty) list of code chunks, possibly with non-marker text before, between and after them.

Other possible answers

- Answers which try to model the Processor class's requirement of strictly alternating chunk types won't be penalised; but it's quite difficult to model this correctly.
- Answers which try to define `<nonmarker_text>` will have a mark deducted for irrelevance – there is no need to define it, and in any case the task is impossible using BNF.
- Answers which do things other than what was asked for (namely, writing a grammar) will have marks deducted for irrelevance. (e.g. writing test cases. This was not asked for. Don't provide things that weren't asked for; just answer the questions, no more and no less.)

Poor answers

The following indicate major issues in a student's understanding of grammars and/or syntax-based testing.

- Making incorrect claims about the syntax given. e.g. claiming that `( <doc_chunk> | <code_chunk> )*` represents a strictly alternating list of doc chunks and code chunks. (It doesn't.)
- Failing to specify repeated elements (using "`*`" or "`+`") or optional elements (using "`?`"). Any correct answer will have to, at a minumum, specify repeated elements.
- Nonsense or incomprehensible grammars.

D. ShellProcessor tests

Question D(1):

The `processors.ShellProcessorTest` contains a sample test, `testOneLineDoc` for the `ShellProcessor` class.

- Do you think `testOneLineDoc` is a unit test, or an integration test? Why? (3 marks)

Competent solutions

Possible good answers here might state that:

- `testOneLineDoc` is an integration test, since it uses multiple classes, and tests whether the output of the `Parser` works properly with a `ShellProcessor`.
- `testOneLineDoc` is a unit test, since it seems to focus on testing the processor class – note that the only assertion included in the test (which is currently commented out) is do with observing the output produced by the `ShellProcessor` class. However, it's not a very *good* unit test, since it needlessly makes use of the `Parser` class, and a property of good unit tests is that they test only the "class under test". The `Chunk`s which in `testOneLineDoc` are output by the parser should have instead been constructed "by hand" (or by using a utility method).

A reasonable, although not as good answer:

- `testOneLineDoc` is a unit test, since it seems to focus on testing the processor class – note that the only assertion included in the test (which is currently commented out) is do with observing the output produced by the `ShellProcessor` class. However, it's not a very *good* unit test, since it needlessly makes use of the `Parser` class, and a property of good unit tests is that they test only the "class under test". The `Parser` object should have been replaced by a mock.

The reason this isn't as good is that if `testOneLineDoc` is a unit test of `ShellProcessor`, there's no need for `Parser` at all, not even a mock of it; all we need is a `List` of `Chunk`s.

A less-than-reasonable answer:

- `testOneLineDoc` is a unit test, since it seems to focus on testing the processor class – note that the only assertion included in the test (which is currently commented out) is do with observing the output produced by the `ShellProcessor` class. However, it's not a *good* unit test, since it needlessly makes use of the `Parser` and `Chunk` classes. All classes other than `ShellProcessor` should be replaced by mocks.

This is not a very sensible approach to testing. `DocChunk` and `CodeChunk` have very little logic in them (objects like these are sometime called "data objects"); simulating their "behaviour" (which is minimal) with mocks would result in needlessly complex tests.

Poor answers

Answers which fail to demonstrate a good understanding of unit tests and integration tests may:

- Provide an answer ("It's a unit test" or "It's an integration test") with no justification. These will be awarded 0 marks.
- Give incomprehensible or nonsense justifications for their conclusion. These will be awarded 0 marks (or at most a few marks, if they make partial sense).

Question D(2):

In the `processors.ShellProcessorTest` class:

- Create a JUnit *parameterized* test in the `ShellProcessorTest` class, which executes on 4 different sets of test values. You need not follow any particular technique for choosing your test values, but we will assess how likely it is that your test values will identify a wide range of defects. (10 marks)

Competent answers

Good answers will:

- Define a method or variable which acts as a *source* of data for a parameterized test. Typically, this will be a static method that returns a `Stream<Arguments>`.
- Define a parameterized test which uses the source. It will need a `@ParameterizedTest` annotation and an annotation like `@MethodSource("mySourceMethod")`.
- Perform the typical "arrange, act, assert" portions of a test. See section (B) for marking details.

Sample code for a data source:

```java
// provide (input, expected output) pairs for
// a parameterized test.
private static Stream<Arguments> expectationsSource() {
  List<Arguments> expectations = List.of(
      Arguments.of("foo", "foo\n"),
      Arguments.of("foo\n", "foo\n"),
      Arguments.of("foo\nbar\n", "foo\nbar\n"),
      Arguments.of("foo\n=<<<\necho XXX\n>>>\nbar\n", "foo\nXXX\nbar\n"),
      Arguments.of("foo\n=<<<\necho XXX\n>>>", "foo\nXXX\n"),
      Arguments.of("foo\n=<<<\necho XXX\n>>>\nbar", "foo\nXXX\nbar\n")
  );

  return expectations.stream();
}
```

Sample code for a parameterized test using that source:

```java
@ParameterizedTest
@MethodSource("expectationsSource")
public void testMergeParameterized(String input, String expectedOutput)
    throws ExecutionException
{
  BufferedReader reader = StringUtilities.stringToBufferedReader(input);
  Parser parser = new Parser(reader);
  parser.parse();
  List<Chunk> chunks = parser.getChunks();

  MemIO mio = new MemIO();
  Processor processor = new ShellProcessor(mio.out, chunks);
  processor.merge();

  String actual = mio.getContents();

  assertEquals(expectedOutput, actual);
}
```

The exact values used will be the ones explained/justified in question D(3).

Question D(3):

In your report, answer the following question:

- For your 4 different sets of test values, why did you choose them? What sort of defect do they aim to expose? (5 marks)

Competent answers

There are many possible answers here – good answers will contain a sensible justification of the values chosen, usually on the grounds that they are:

- degenerate and/or boundary cases, where the system implementer is likely to make a mistake
- unusual input cases, which the system implementer might not have considered.

There is no requirement that your tests use `String`s as input; they could use a `List<Chunk>` instead (e.g. if you decide to do unit-testing of `ShellProcessor`, rather than integration testing.)

Some possibilities for test values:

- An empty `List<Chunk>`, or an empty document. It's a degenerate case (and there's nothing in the section D questions that precludes us from using it). It would highlight defects that occur when the system implementer incorrectly assumes the input must have length greater than zero.
- Documents containing a `CodeChunk` with empty contents – again, a degenerate case the system implementer may not have allowed for.
- Documents containing a `CodeChunk` with "non-typical" contents – e.g. invoking a non-existent command, or a command that is likely to result in an error. The JavaDoc for the `IProcessor` interface (which `Processor` and `ShellProcessor` implement) states what should happen in such a case: namely, an `ExecutionException` should be thrown.
- Documents with malformed start- or end- markers for a code chunk. (Strictly speaking, these are probably better as tests for the `Parser` class; but we might be interested in seeing what happens when the Processor encounters them.)

Poor answers

Answers which fail to demonstrate a good understanding of testing, and parameterized tests in particular, may do the following:

- Provide test values with no explanation of why they are being used or what defects they aim to expose. These will be awarded 0 marks.
- Contain incomprehensible or nonsense justifications. These will be awarded 0 or only a few marks.
- Attempt to test undefined behaviour (e.g., the situation where the input contains 2 contiguous code chunks). By definition, undefined behaviour cannot be tested for, so this is a very serious mistake (and one that has been outlined in lectures multiple times). Such answers will be awarded 0 or only a few marks.

E. Logic-based tests

Your colleague Anastasia has been reading about logic-based testing, and wonders whether it would be a good use of time to assess what level of coverage the tests for the JDocGen system have (e.g. Do they have Active Clause Coverage)?

In your report, answer the following question:

- Would measuring logic-based coverage for the JDocGen be a good use of the team's time? Why or why not? Explain your reasoning. (5 marks)

Competent answer

To answer this, we need some idea of how much effort would be required to measure the level of logic coverage of the JDocGen system, and how much benefit we might gain from it.

As far as the time required is concerned: it should take very little time. Logic-based testing models the system as a set of predicates, and we can inspect the JDocGen codebase to see exactly how many predicates there are in it: the answer is 12.

Here is a breakdown of where they occur:

Class	No. of predicates
`core.Chunk`	2
`parsers.Parser`	4
`processors.CommandRunner`	1
`processors.ReverseProcessor`	2
`processors.ShellProcessor`	3

All of them consist of single-clause predicates that are part of an "`if`" statement (e.g. line 76 of `Parser.java`, which contains the line "`if (currentLine.equals( CodeChunk.START_MARKER) )`"). This means achieving "Active Clause Coverage" would be trivial: since there's only one clause, it automatically determines the value of the whole predicate. (In other words: ACC for the JDocGen system will have exactly the same number of tests as Predicate Coverage or Clause Coverage would.)

Because all the predicates are single-clause, ACC should also end up being almost equivalent to Edge Coverage in a control-flow graph. (The graph structure would also have some edges representing the two `"for"` loops in the codebase; but other than that, every branch in the control-flow graph is equivalent to a predicate.)

So the amount of effort involved to *check* whether we have ACC is very small. A developer just needs to ensure we've exercised each of the 12 predicates/"`if`" statements in the program. Even without code instrumentation, this could be checked just by inserting some logging/print statements at various points in the program. It could be done in a couple of hours, at most.

So at worst, we'll spend a few hours and perhaps discover we already have ACC; but at best, we'll identify that our existing tests don't execute some paths through the system, and will have the opportunity to write some new tests and improve our code coverage.

So the answer is "Yes": we *should* check to see whether we have ACC. The cost is minimal, and it could have a useful payoff.

Poor answers

Answers which fail to demonstrate a good understanding of testing, and logic-based testing in particular, may do the following:

- Assume without justification that checking for Active Clause Coverage would take a great deal of time. This just doesn't make sense: the whole system consists of only 13 classes, none of which have very complex methods; why would checking ACC take a great deal of time?
- Assume without justification that checking for Active Clause Coverage would take minimal time. Although the conclusion here is true (checking won't take long), the question requires all students to explain their reasoning. Failure to justify your conclusions is a major error.
- Provide a poor or unfounded justification for their assessment of how long checking for ACC would take.
- Not bother to assess how long checking for ACC would take at all.
- Provide incomprehensible or nonsense explanations.