CITS5501 Software Testing and Quality Assurance
Syntax-based testing

Unit coordinator: Arran Stewart

Overview

- Syntax-based models
  - Keeping a note of the rules and symbols we need is useful here (or bring your pre-reading notes)
- Mutation testing

Grammars and syntaxes

Grammars, syntax and language

Developers use grammars and syntax of all the time, though they may not realize it.

Whenever we see a requirement like “a date should be in the format `YYYY-MM-DD`”, we’re making use of a grammar (though only very informally expressed).

Analysing a date

If some requirement says that a parameter to a function, or an item in a database, should be "in the format `YYYY-MM-DD`", what it (usually) means, but more explicitly stated is:

- Any of the Y's, M's or D's can be replaced by a digit in the range 0–9 – if you provide a date that can't be generated in such a fashion, we might say you've provided a *syntactically ill-formed* date.
- There are other rules about validity (e.g. if the first "M" is replaced by a 1, then the second "M" can only be in the range 0–2), but we usually don't consider those to be *syntax* errors.
    - Dates which violate these rules are usually said to violate the *semantics* of dates, or *semantic constraints*

Analysing a date

If some requirement says that a parameter to a function, or an item in a database, should be "in the format `YYYY-MM-DD`", what it (usually) means, but more explicitly stated is:

- Any of the Y's, M's or D's can be replaced by a digit in the range 0–9 – if you provide a date that can't be generated in such a fashion, we might say you've provided a *syntactically ill-formed* date.
- There are other rules about validity (e.g. if the first "M" is replaced by a 1, then the second "M" can only be in the range 0–2), but we usually don't consider those to be *syntax* errors.
  - Dates which violate these rules are usually said to violate the *semantics* of dates, or *semantic constraints*

(What is it that distinguishes between the two? We'll come to that later.)

Grammars

Grammars just give us a way of formally specifying what things are and are not syntactically correct.

Every grammar defines what is called a *language* (though not always a very interesting one) – a set of acceptable strings.

A grammar for the date might look like this:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"

<date>  ::=  <digit> <digit> <digit> <digit> "—"
             <digit> <digit>  "—"
             <digit> <digit>
```

Grammars

The following grammar is *equivalent* to the previous one – in that they define the exact same set of strings – but provides a few hints as to the *semantics* of bits of the string (and is probably a bit easier to read).

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"

<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The notation is a simplified form of what is called BNF (Backus-Naur Form).

The following symbols are used in this notation:

We read “::=” as “is defined as” or “can be expanded to”, and “|” as “or”.

So the first line says, “A ‘digit’ is defined as being either the string”0”, or the string “1”, or . . . ”

(These symbols are sometimes called “meta-syntactic symbols”, meaning symbols used to define a syntax.)

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The things in strings are called *terminal symbols* – they are the equivalent of “words” in our language.

They are like atoms, in that they are the smallest, indivisible parts of our language.

In our case, the terminals are all strings containing a single digit.

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The things between angle brackets are called *non*-terminal symbols.

The above grammar contains five *rules* (also called “productions”, in the textbook).

In the sorts of grammar we will consider,[1] every rule is of the form:

non-terminal “::=” *sequence of terminals and non-terminals*

[1] Called *context-free grammars* or CNFs (see https://en.wikipedia.org/wiki/Context-free_grammar).

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

To be precise: the simplest possible right-hand side (RHS) of a rule will be a sequence of terminals and non-terminals, meaning "these strings, concatenated together".

(For example – the RHS of the last rule, which means "an expansion of the 'year' rule, then a hyphen, then an expansion of the 'month' rule, then a hyphen, then an expansion of the 'day' rule.)

Notation

But we can also insert on the RHS the following symbols, between or after terminals and non-terminals:

- bars to indicate "or" (alternatives)
- an asterisk (called the "Kleene star") to indicate "zero or more of the preceding thing"
- a plus sign to indicate "one or more of the preceding thing"
- a range of numbers (e.g. "3–4") to indicate a number of possible instances of the preceding thing.

Notation

Examples

- `"dog" | "cat"` – either the string `"dog"`, or the string `"cat"`
- `"dog"*` – zero or more instances of the string `"dog"`
- `"dog"+` – one or more instances of the string `"dog"`
- `"0"-"7"` – digits from "0" to "7" inclusive

And we can use parentheses to group things.

- `( "dog" | "cat" )+` – one or more instances of these two alternatives

Notation – asterisk

An example: the following is a fairly typical way of defining valid *identifiers* in many programming languages:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<letter> ::= "a" | "b" |    ...
<underscore> ::= "_"
<identifier> ::= ( <letter> | <underscore> )
                  (<letter> | <underscore> | <digit>)*
```

This means, “An identifier always starts with a letter or underscore; it is followed by any number (possibly zero) of characters drawn from the set of letters, digits and the underscore character”.

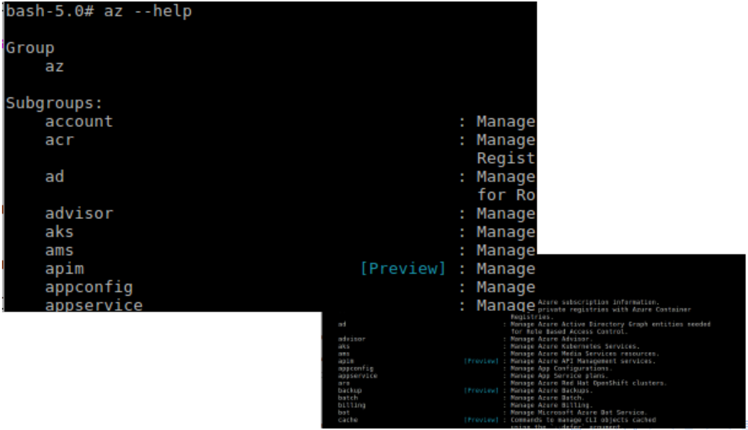
Use in development

Much of the software we rely on makes use of grammars (though not always explicitly).

Whenever we *validate* entries into web forms or databases, we are often are defining a syntax to do so. (One common way is to use what are called *regexes* – we will discuss them more later.)

Use in development

Command-line programs often take arguments – sometimes adhering to very complex rules, as we saw in the first lecture.



```
bash-5.0# az --help

Group
    az

Subgroups:
    account                                       : Manage
    acr                                           : Manage
                                                    Regist
    ad                                            : Manage
                                                    for Ro
    advisor                                       : Manage
    aks                                           : Manage
    ams                                           : Manage
    apim                               [Preview] : Manage
    appconfig                                     : Manage
    appservice                                    : Manage
```

```
                                                  Azure subscription information.
                                                  private registries with Azure Container
                                                  Registries.
    ad                                : Manage Azure Active Directory Graph entities needed
                                        for Role Based Access Control.
    advisor                           : Manage Azure Advisor.
    aks                               : Manage Azure Kubernetes Services.
    ams                               : Manage Azure Media Services resources.
    apim                    [Preview] : Manage Azure API Management services.
    appconfig                         : Manage App Configurations.
    appservice                        : Manage App Service plans.
    aro                               : Manage Azure Red Hat OpenShift clusters.
    backup                  [Preview] : Manage Azure Backups.
    batch                             : Manage Azure Batch.
    billing                           : Manage Azure Billing.
    bot                               : Manage Microsoft Azure Bot Service.
    cache                   [Preview] : Commands to manage CLI objects cached
                                        using the `--defer` argument.
```

Use in development

For very simple programs, we might analyse the arguments “by hand”.

For complex programs – we typically use a *command-line argument parser* to work out whether a user has supplied a valid set of arguments (and what we should do with them).

Use in development

Grammars are used to define whether something is a valid

- email
- HTML page
- email address

and many other formats.

Use in development

Often, it will be useful to define what are called “domain-specific languages” (**DSLs**) which describe entities in a domain and things to do with them – e.g. Makefiles are an example of this.

Syntaxes are typically used to define such languages.

Use in development

And of course, every **programming language** is defined by a grammar or syntax – when we violate the syntax, the compiler tells us we've committed a "syntax error".

Syntactically well-formed Java class:

```
class MyClass { }
```

Syntactically ill-formed:

```
class { MyClass  }
```

Questions

- **Q.** What's the dividing line between what we call "syntax" ("Student numbers are of the form: NNNN-NNN-NN, where N is a digit") and semantics ("If the first digit of the month is 1, the second can only be '0' or '1' or '2' ")?

Questions

- **Q.** What's the dividing line between what we call "syntax" ("Student numbers are of the form: NNNN-NNN-NN, where N is a digit") and semantics ("If the first digit of the month is 1, the second can only be '0' or '1' or '2' ")?
- **A.** Usually, if a rule can be described using BNF, it's called a syntactical rule; if not, it's a semantic rule.

  Languages using such rules are called **context-free languages**.

  (Why "context-free"? Because we can't have rules like the one just described, that say "*If* the previous item was '1', then *this* item can only be '0' or '1' or '2' – the grammars we use never require us to supply *contextual* information of this sort.)

Questions

- **Q.** Can we describe binary formats, as well as text?

Questions

- **Q.** Can we describe binary formats, as well as text?
- **A.** Yes, though BNF is not especially suited to describing binary formats.
    - BNF works well for things in textual format (including the source code of programming language files, HTML documents, JSON documents, and so on).
    - For data in binary format (for instance, TCP packets or JPEG files), a commonly-used formalism is ASN.1 ("Abstract Syntax Notation One").
    - We won't be examining ASN.1 in detail, but similar considerations apply.

Questions

- **Q.** Are there any other artifacts we can describe using a grammar?

Questions

- **Q.** Are there any other artifacts we can describe using a grammar?
- **A.** Yes – we can often use grammars to describe data structures.
    - This follows on from the previous question.
    - If we can use a grammar to describe, say, the JPEG format; and if JPEGs can be stored as data structures; it follows we could use grammars to describe the permissible instances of data structures.
    - In particular, **recursive data structures** can often be described using a grammar.

Using the Syntax to Generate Tests

- Syntactic descriptions can be obtained from many sources:
  - program source code
  - design documents
  - input descriptions (e.g. file formats, network message formats, etc)
- Tests are created with two general goals
  - Cover the syntax in some way
  - Violate the syntax (invalid tests)

Using the Syntax to Generate Tests

- Should we apply the techniques we see in this lecture to *every* example of syntactic validation / use of grammars?
- Usually not – we will usually focus on areas of high risk (e.g. that are easy to get wrong, or have bad impacts when we get them wrong).
- Parsing command-line arguments is sufficiently important that we should probably test it.

Example – arithmetic expressions

Another example – we'll define a language to represent simple arithmetic expressions.

Some strings will be *valid* in our language (like "`(3 + 2) - 5`") and some will not (like "`3++-(`").

Our *terminal symbols* will consist of the numerals 0-9, and the symbols "`+ - ( )`".

Example – arithmetic expressions

As before, we define a digit:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

Example – arithmetic expressions

And we can say, “An *expression* is either a digit, or, a smaller expression plus some other smaller expression.”

```
<expression> ::= <digit> | <expression> "+" <expression>
```

Example – arithmetic expressions

Our whole grammar:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"

<expression> ::= <digit>
               | <expression> "+" <expression>
               | <expression> "—" <expression>
               | "(" <expression> ")"
```

More on BNF grammars

When we specify a grammar, there will normally be a *start symbol*, representing the "top level" of whatever construct we're specifiying.

- e.g. for some programming language:

```
<program_file> ::=
        <import_statements><declarations><definitions>
```

- Each possible rewriting (i.e., each alternative) of a non-terminal is often called a *production*.

Use of grammars

- Grammars can be used to build **recognizers** (programs which decide whether a string is in the grammar – i.e., parsers) and also **generators**, which produce strings of symbols.

Coverage criteria

- If we're developing tests based on syntax . . .
- The most straightforward coverage criterion:
  use every terminal and every production rule at least once

Terminal Symbol Coverage (TSC) Test requirements contain each terminal symbol t in the grammar G.

Production Coverage (PDC) Test requirements contain each production p in the grammar G.

Coverage criteria (cont'd)

- Production coverage subsumes terminal symbol coverage;
  if we've used every production, we've also used every terminal.

Coverage criteria – an impractical one

- We could aim to cover all possible strings

Derivation Coverage (DC) Test requirements contain every possible string that can be derived from the grammar G.

- But except in special cases, this will be impractical

Bounds on coverage

- Example grammar:

```
<integer> ::= <digit>|<integer><digit>

<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

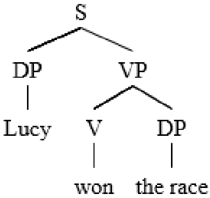
- The number of tests to get TS coverage is bounded by the number of terminal symbols (ten, here)
- To get production coverage, that depends on the number of productions (here: 2 for the first rule, 10 for the second – so, 12)
- Whereas the number of strings that can be generated – needed for derivation coverage – is actually infinite.
  - (likewise for, say, the set of all possible Java programs)
- Even for finite grammars (e.g. some file formats), DC will usually require an infeasibly large number of tests

Data structures

- Typically, for any format we specify syntactically (like JPEG, GIF etc.), we'll have an accompanying data structure that mirrors that the structure of the syntax, in order to manipulate in-memory objects representing that format.
- E.g. see the `JpegImageData` class from the Apache Commons Imaging library for Java, or the `png_struct_def` for the `libpng` C library.

Trees

- We can draw a *tree* structure for an expression adhering to some particular syntax called a *parse tree*:[2]



(Here, "S" stands for "sentence", "VP" for "verb phrase", "V" for "verb", "DP" for "determiner phrase" – basically something that picks out a particular entity.)

[2] Image from https://commons.wikimedia.org/wiki/File:Precedent_example_1_decl_sent.png

Trees

- The parse tree shows what productions should be followed to parse (or alternatively, to generate) a particular string.

Generators

- Suppose we had the grammar:

```
<Sentence> ::= <NounPhrase><Predicate>
<NounPhrase> ::= "Alice" | "Bob" | "the hacker"
<Predicate> ::= <Verb><NounPhrase>
<Verb> ::= "hires" | "defeats"
```

- Then we can see that “Alice hires Bob” and “Bob defeats the hacker” are valid strings in the language this grammar defines (modulo some whitespace).
- And we can see how we could easily generate random valid sentences that conform to these rules.
- Being able to generate things that follow a syntax-like structure is extremely useful for testing.

Generators – network traffic

- We can use it to create traffic generators, for instance – we could generate random valid TCP traffic with which to test a router.
- TCP packets follow a syntax-like structure, so it's fairly straightforward to generate them randomly.
  A TCP packet consists of: 2 bytes representing a source port (0 through 65535), 2 bytes representing a destination port, then 4 bytes representing a "sequence number", then . . . (see the TCP specification for detailed rules).
- Not all the validity rules for a TCP packet can be expressed in a syntactical way – for instance, it contains a checksum towards the end, which is calculated based on previous information – but quite a bit can.
- This is very handy for "stress" or "load" or "performance" testing – generating large amounts of data, and seeing how our system performs under the load.

Generators – http traffic

- HTTP requests for web pages also follow a syntax, so we could easily generate random HTTP traffic (for instance, to stress-test a web-server, and see how it performs under high load).
- The full syntax for HTTP requests is larger than this,[3] but the start of a simplified version of it would look something like:

```
<request> ::= <GETrequest> | <POSTrequest>
<GETrequest> ::=  "GET" <space> <URI> <space> <HTTPversion>
                  <lineend> <getheaders> <getbody>
...
```

(i.e., HTTP requests are either `GET` or `POST` requests, and `GET` requests start with the keyword `GET` then a space, then a URI, and so on...)

[3] See IETF RFC 2616, https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

Generators – http traffic

- The vast majority of randomly generated HTTP requests would not be for valid URIs, and would result in 404 errors.
- If we wanted to generate, not just random HTTP requests, but requests that actually hit part of a website, we can add in additional constraints to ensure that happens.
- (E.g. We might start by only generating URLS that begin with `https://myblog.github.io/`, if we were testing a blog site hosted on GitHub.)

Generators

- Likewise, HTML and XML documents, JSON, and many other formats all follow syntactical rules, so we can randomly generate them.
- Likewise for custom formats we may come up with.
  - e.g. If we were writing a word processor, we might want to be able generate very large random documents in our word-processor format, to see how our program holds up.

Generators

- For common formats, there are often already data generators with many capabilities:
  - Tools for constructing and generating network traffic: Ostinato, Scapy Traffic Generator, flowgrind, jtg . . . see this list for many more.
  - HTTP request generators: see for example httperf
  - Random bitmap generators: see for example random.org
- If not, it is perfectly possible to write our own.

Generators and data structures

Things to note when generating data structures:

- In languages with pointers or references, it may be possible to have data structures that contain *cycles*, meaning they are no longer trees but graphs.
- For instance, we could have two linked list nodes A and B, and make A's `next` reference point to B, and B's point to A. (A cyclic linked list.)
- It's still possible to generate random data of that sort, but doing so takes us beyond our current scope.

More complex rules for validity

- There may be rules for validity of a format (like the existence of checksums) that can't be captured by a grammar.
- This is frequently the case, actually. BNF lets us describe what are known as "context-free" grammars, and a specification for a format may include requirements that are impossible or inconvenient to specify using BNF.
    - e.g. In a valid Java program, variables have to be declared before they are used; it's an error to assign a string literal to an int; and many other rules.
- We may be able to use simple calculations to generate or verify those.

  (e.g. to verify or generate a checksum)
- Or we may have to apply more complex rules – these are outside the scope of this unit.

Using generators for testing

- Generating random, valid values is useful for performance testing, as just described – but it is also useful for *property-based testing*, which we will see more of later.
- What is property-based testing? It's a sort of (usually randomized) testing which checks that invariants about functions hold.

Applications of syntax-based testing

- Mutation-based testing
- Generators used for **property-based testing** and **fuzzers**

Mutation testing

Mutation testing

Mutation testing (also called “mutation analysis”) is a technique for evaluating the quality of a suite of software tests.

- Suppose we have some program under a test, and a suite of tests designed to identify defects in it.
- Mutation testing works by modifying the program under test in small ways (e.g. flipping a less-than sign to a greater-than; changing a hard-coded number from 0 to 1).
    - These are usually designed to mimic typically programming errors, such as typograhical errors, wrong choice of operator, or off-by-one errors
- If our test suite doesn’t detect and reject the mutated code, we consider it defective.

Mutation testing – terminology

- Ground string: A string in the grammar
  - (The term "ground" basically means "not having any variables" – in this context, not having any non-terminals)
- Mutation operator: A rule that specifies syntactic variations of strings generated from a grammar
  - (e.g. "If the string has a less-than symbol in it, flip that intro a greater-than symbol")
- Mutant: The result of one application of a mutation operator
  - A mutant is a string

Killing Mutants

- An example of a ground string is – our program under test.
- . . . since it's a string in the grammar of "syntactically valid Java programs"
- We apply our mutation operator to the ground strings to generate *mutants*, new valid strings
- Killing mutants: If we have some mutant generated from the original ground string, and we look at one or several of our tests, we can ask: do they "**kill**" the mutant?
  - i.e. Does the test(s) give a different result for the mutant, compared to the original?
  - If it does, it's said to "kill" the mutant.

Mutation example

- A sample method to test (in a Java-like language):

  ```
  int mult(int a, int b) { return a * b; }
  ```

- A possible test:

  ```
  assertEquals( 1, mult(1,1) );
  ```

- Is this a useful test?

Mutation example (2)

- ans.: No, it's terrible.
- Consider the following mutation:

  ```
     int mult(int a, int b) { return a * b; }

  ⇒ int mult(int a, int b) { return a ** b; }
  ```

  (where `**` is a "power" operator)
- `1*1 == 1**1` so our test will still pass -
  - so it's a pretty poor test

Frameworks for mutation testing

Some example mutation testing frameworks are:

- PIT, for Java (originally stood for “Parallel Isolated Test”)
- Mutpy, for Python
- Stryker, for C#
- Mutagen, for Rust (motto: “Breaking your Rust code for fun & profit”)

Advantages and disadvantages

Advantages and disadvantages of mutation testing:

- Identifies weak/ineffective tests
- Very effective at finding problems
- Helps quantify how useful your tests are

- Can be time-consuming (large number of mutants to generate, whole test suite needs to be run many times)
- Results require some familiarity with mutation testing to be properly understood.

Syntax-based coverage criteria – mutant coverage

- We can define a coverage criterion in terms of killing mutants:

Mutation Coverage (MC) For each mutant m, the test requirements contains exactly one requirement, to kill m.

- Coverage in mutation equates to number of mutants killed
- The amount of mutants killed is called the mutation score

Coverage criteria – creating invalid strings

- When creating invalid strings, two simple criteria –
- It makes sense to either use every operator once or every production once

Mutation Production Coverage (MPC) For each mutation operator, TR contains several requirements, to create one mutated string *m* that includes every production that can be mutated by that operator.

Mutation Operator Coverage (MOC) For each mutation operator, TR contains exactly one requirement, to create a mutated string *m* that is derived using the mutation operator.

Mutation testing example[4] – IOT cat door

Suppose we have an Internet of Things (IoT)–enabled cat door. The final cat door will be implemented as hardware with embedded software; but we can still check our logic using testing techniques we have seen before.

We have the following user story describing the purpose of the cat door:

> *Using my home automation system (HAS),*
> *I want to control when the cat can go outside,*
> *because I want to keep the cat safe overnight.*

[4] Adapted from Alex Bunardzic, "Mutation testing by example: Failure as experimentation" (2019)

Cat door interface

We represent the cat door using the following interface:

```
public interface ICatDoor {
  /** When "day" is supplied, unlock the door;
    * when "night" is supplied, lock the door */
  public void control(String dayOrNight);
  /** Returns either "locked" or "unlocked" */
  public String getStatus();
}
```

Testing scenario

We want to write tests revolving around the following scenario:

Scenario #1: Disable cat trap door during nighttime

Given that the day/night detector detects that it is nighttime
When the day/night detector notifies the HAS
Then HAS disables the IoT-capable cat door

(We won't worry about how the day/night detector is implemented. Perhaps it uses ambient light levels; perhaps it consults a database of sunrise/sunset times for its current geographical location.)

Cat door code under test

```
public class CatDoor implements ICatDoor {
  // ...
  public void control(String dayOrNight) {
    if (dayOrNight.equals("night")) {
      this.lock();
    } else {
      this.unlock();
    }
  }
}
```

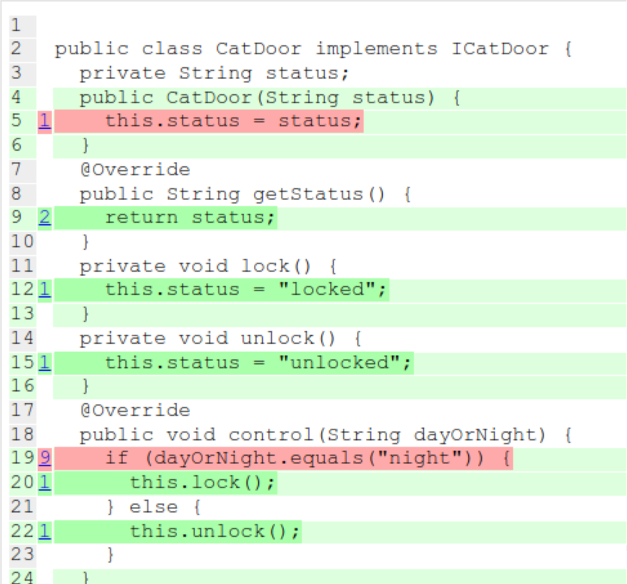
Cat door test code

```
public class TestGivenNighttimeDoorLocked {
  @Test
  public void test() {
    ICatDoor door = new CatDoor();
    door.control("night");
    String expected = "locked";
    String actual   = door.getStatus();
    assertEquals(expected, actual, "status should be locked");
  }
}
```

start PIT running

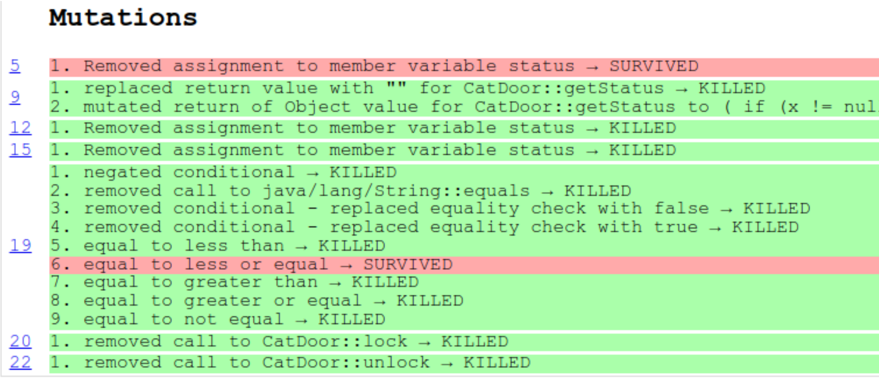
```
27:46 am PIT >> INFO : Completed in 2 seconds
===============================================================
— Mutators
===============================================================
> org.pitest.mutationtest.engine.gregor.mutators.rv.ROR3Mutator
>> Generated 1 Killed 1 (100%)
> KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
---------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.VoidMethodCallMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
<more output snipped>
```

PIT output



```
  
   public class CatDoor implements ICatDoor {
     private String status;
     public CatDoor(String status) {
1      this.status = status;
     }
     @Override
     public String getStatus() {
2      return status;
     }
     private void lock() {
1      this.status = "locked";
     }
     private void unlock() {
1      this.status = "unlocked";
     }
     @Override
     public void control(String dayOrNight) {
9      if (dayOrNight.equals("night")) {
1        this.lock();
       } else {
1        this.unlock();
       }
     }
```

PIT output



```
    Mutations

5   1. Removed assignment to member variable status → SURVIVED
9   1. replaced return value with "" for CatDoor::getStatus → KILLED
    2. mutated return of Object value for CatDoor::getStatus to ( if (x != nul
12  1. Removed assignment to member variable status → KILLED
15  1. Removed assignment to member variable status → KILLED
    1. negated conditional → KILLED
    2. removed call to java/lang/String::equals → KILLED
    3. removed conditional - replaced equality check with false → KILLED
    4. removed conditional - replaced equality check with true → KILLED
19  5. equal to less than → KILLED
    6. equal to less or equal → SURVIVED
    7. equal to greater than → KILLED
    8. equal to greater or equal → KILLED
    9. equal to not equal → KILLED
20  1. removed call to CatDoor::lock → KILLED
22  1. removed call to CatDoor::unlock → KILLED
```