Verification

We wish to verify that ArrayQueue and CircularQueue are correct implementations. What are our options for doing this?

Correctness Proof. An ideal approach would be to prove the code is correct. In particular, if you assume the method preconditions are satisfied (eg. each "requires" or "must be" condition stated in the method comments), then can you prove that the representation invariant is always satisfied? And, from the representation invariant, can you then prove the methods are all correct? Unfortunately, this is usually much tougher than writing a correct program in the first place.

Automatic Verification. Why don't we get the computer to prove the code is correct? This is a current research issue in Computer Science. At this point it is clear that we cannot expect a complete, practical, automatic verification tool. There will be limits imposed by the computational intractability of these problems.

Verification by Testing. Moreover, for most programs of interest, we cannot even verify their correctness by testing the output for every possible input. There are usually far too many possible input combinations.

Systematic Testing

Systematic Testing. A practical approach to testing is to consider only systematically selected test cases. The selected cases are typically an insignificant fraction of all possible test cases.

Such partial testing increases our confidence in a program's correctness. But it cannot actually verify correctness (i.e., correct without a doubt). For example, it is possible for the program to break on some cases we didn't test. Nevertheless, with a carefully selected test set, we can be reasonably confident the program is correct.

The key to systematic testing is in the intelligent choice of the tests. We will practice this throughout this course.

In this course we primarily consider **white-box testing**. In white-box testing we can "see inside" the class to the code or, at least, to a detailed description of the algorithm that the code implements.

Black-box testing refers to testing a program for which you do not know how it is implemented. You only know what outputs it should provide for given inputs (eg. you might only know the interface it is supposed to implement). It is typically much tougher to come up with intelligent and thorough tests in black-box testing.

Possible Test Cases

What's a test? A test first puts the program being tested into **a particular state**. Say we are testing just one class, then the state is just the set of values held by all member variables of that class.

The test then provides **possible input values** the class needs in order to execute each of the methods (i.e., all the method parameters, along with the return values and side-effects of any method calls invoked within the class being tested).

Finally, for the specified state and input, the test **verifies the result of each method call**.

For example, in order to test ArrayQueue we need to set the instance variables size and contents[*] to particular values. The only input the test case needs to specify is the argument of the enqueue method. The various methods in ArrayQueue can then be called, and the subsequent state of **all** the instance variables checked for the appropriate change (including no change). For example, in the case of the enqueue method, if the queue wasn't initially full before the method call, then it needs to be verified that the new object is enqueued at the end of the queue, the size is incremented, and the previous elements within the queue are left in their arrival order.

Divide the Possible Tests into Categories

Systematic testing involves the careful selection of which tests are important to run. To do this we attempt to identify categories of tests for which the performance of the program is likely to be very similar.

For the ArrayQueue class definition, for example, one huge category of tests could involve situations where:

capacity (i.e., contents.length) is reasonably large,

size has a moderate value (i.e., size is significantly bigger than 0 and significantly smaller than capacity),

contents has the correct items in contents[0..size-1] (see the Representation Invariant), and contents[size..contents.length-1] can contain any possible references (including null),

enqueue's parameter o is any reference (including null).

The key property here is that each method in ArrayQueue **executes the same lines of code** for every test within this category. It is therefore reasonable to believe that only a few test examples need to be sampled from this category to convince us of the correctness of the implementation for the whole category of tests. (Now that's divide and conquer!)

Choose Categories to Include all Possible Tests

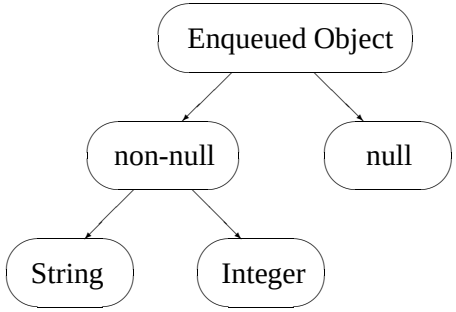
For ArrayQueue, special situations occur when the queue is empty or nearly empty, and when it is full or nearly full. In particular, different lines of code are executed in these special cases. Therefore these two special cases (i.e. nearly empty, and nearly full) should be included in test categories of their own.

We should continue to add test categories until all the special cases are covered.

The **general rule of thumb** is that we should choose enough categories so that every line of code is executed within at least one category.

We show examples of the design of systematic tests next.

Systematic Testing: ArrayQueue



Classes of Objects. For ArrayQueue, a quick scan of the code indicates that it is unlikely that the class of the objects being enqueued and dequeued matters.

One special case is the null reference which, due to the absence of any stated preconditions to the contrary, could be enqueued. The null reference is also used to initialize the contents[] array. Otherwise any reference is treated the same way.

We test this by considering only two (non-null) classes, String and Integer. We will enqueue random selections of String and Integer objects, along with null references.

Testing ArrayQueue: Decision Points

Similarly, we cannot test all combinations of possible sizes and capacities for the queue. Rather, we need to identify particular values of these parameters which represent decision points, that is, places where the execution changes in some way. These occur, for example, when different if-clauses are executed. For ArrayQueue this happens when the queue is either nearly empty or nearly full. Away from decision points, we will only test a few sample values.

Therefore we identify the following special categories for testing the capacity and size:

capacity (i.e. contents.length):

- small (say $\{0, 1, 2\}$),
- medium (say 10),
- large (say 100, at least for how we intend to use it).

size:

- empty or nearly so (say $\{0, 1, 2\}$),
- partially filled (say three random integers from the interval $(2, \text{capacity} - 2)$, if they exist),
- filled or nearly so (say 0, 1, or 2 below capacity).

Testing ArrayQueue: State of contents[]

So far we have considered the contents of the actual queue along with its size and capacity. This almost completely characterizes the state of any ArrayQueue object.

All we have missed are the values of contents[*] **beyond** the current extent of the queue. These are initialized to null, but do not remain null. We therefore consider two cases,

cold start: The ArrayQueue reaches this size for the first time (so contents[k] is null for $k \geq$ size),

warmed up: The ArrayQueue has been full at least once before.

As a result, we have one random mix of objects for input, 5 different capacities, up to 9 sizes, and 2 warmed-up states (i.e. cold or warm).

Note that the number of combinations here is roughly $1 \times 5 \times 9 \times 2 = 90$ test cases. (Actually it is somewhat less because the small capacities $\{1, 2, 3\}$ can only be tested with correspondingly small sizes.) These tests could be read from a hand-constructed file, or generated automatically.

Combinatorics of Tests

In general, the number of test cases can grow rapidly with the number of parameters and the number of states that need to be tested within each parameter. The total number of test cases can be as large as the **product** of the number of selected values within each of the parameters (eg. n^p for n values within each of p different parameters).

The growth of n^p with increasing p is called exponential. Only for relatively small values of p and n will a systematic test of all these cases be feasible.

For example, consider the increase in the number of test categories required for CircularQueue, over those required by ArrayQueue.

Note that many of the test categories chosen above for ArrayQueue should be further subdivided for CircularQueue. In particular, there are new decision points within the code for CircularQueue that correspond to the instance variables tail or head wrapping around. Another decision point is when tail and head are nearly equal. But this is equivalent to size being nearly 0 or capacity, which we have already taken into account.

Additional Test Categories for CircularQueue

Special values for the head and tail instance variables are:

head:

- nearly 0, say $\{0, 1, 2\}$,
- nearly capacity-1, say 1, 1, or 3 below capacity,
- nearly equal to tail, say within 2, taking wrap-around into account (note this is equivalent to size being nearly 0 or capacity),
- other values away from these special cases, say three random choices.

tail:

- nearly 0, say $\{0, 1, 2\}$,
- nearly capacity-1, say 1, 2, or 3 below capacity,
- nearly equal to head, say within 2, taking wrap-around into account (note this is equivalent to size being nearly 0 or capacity),
- other values away from these special cases, say three random choices.

Some of these special values have already been taken into account by the special values we selected above for size. But there remains 9 new cases for each of these two new instance variables. Therefore, for each test category selected for ArrayQueue, there are up to 18 new subcategories that need to be tested for CircularQueue. This pushes the total number of categories towards 18×90!

Unit and Integration Tests

As a consequence of this rapid growth of the number of tests with the number of different state or input parameters, we cannot expect to thoroughly test large programs in one shot.

Unit Testing. Instead, the plan is to thoroughly test the individual components of large programs, so-called units. For our purposes here, a unit could be a particular method, or a particular class. Since good design dictates that these units be relatively simple, it is possible to thoroughly test each of them using systematic testing, as described above.

Integration Testing. Once the units have been tested, small combinations of them can be tested using the same systematic approach. In testing these combinations, we can assume each of the units themselves are correct. This assumption keeps the number of categories that need to be tested from exploding. Finally, we can test larger and larger combinations of the units, assuming the previously tested combinations work correctly. This overall process is called integration testing.

Required Reading: Diane Horton, Software Testing, DCS, Univ. of Toronto, 1999. (Available from the course web-page.)