Concurrent programs have a degree of non-determenism that sequential programs do not, increasing the number of potential interactions and failure modes that need to be planned for and analyzed.

The same techniques for testing correctness and performance in sequential programs can be applied to concurrent programs:

- but the space of things that can go wrong is much larger
- and potential failures may be rare probabilistic occurences rather than determenistic ones

Tests of concurrent classes usually fall into one or all of the classic categories of safety, liveness and performance

- **tests for safety** verify that class's behavior conforms to its specification, they usually take form of testing invariants
  this may require locking a concurrent collection for exclusive access
  employing some sort of atomic snapshot
  or by using "test points" provided by the implementation that let tests asset invariants or execute test code atomically
  test code can unfortunately introduce timing or synchronization artifacts that that can mask bugs that might otherwise manifest themselves
- **tests for liveness**
  tests for progress and non-progress are hard to quantify — how do you verify that a method is blocking and not merely running slowly?
  how do you test that an algorithm does not deadlock?
- **performance tests**
  throughput tests a rate at which a set of concurrent tasks is completed
  responsiveness (latency) tests the delay between a request for and completion of some actiod
  scalability tests the improvement in throughput (or lack thereof) as more resources are made available

Testing for correctness

Unit testing a concurrent class starts with the same analysis as for a sequential class — identifying invariants and post-conditions that are amendable for mechanical checking.

Basic unit tests

Write basic unit test for sequential scenarios, assert invariants and post-conditions. These tests can disclose when a problem is not related to concurrency issues.

Testing blocking operations

Testing that a thread blocks is similar to testing wether an exception is thrown: if finishes successfully, then the test failed.

Testing that a method blocks has **two complications**:

- once a method is blocked, you have to unblock it, which can be solved if the method supports interruptions
- waiting until thread is blocked, have to make a decision about how long the few instructions could possibly take

Instructions:

- create a new test thread which performs a blocking operation and either registers failure right after (calling fail()) or cathes an **InterruptedException** which is a success
- test thread waits for specified time, interrupts the test thread, joins it and asserts **assertFalse(newThread.isAlive())**
  use timed join — for the case when blocking gets stuck; the scenario when it is useful to subclass Thread
  also tests that blocking method throws **InterruptedException** when blocked and interrupted

Caution: don't use **Thread.getState** to verify that the thread is actually in **WAITING** or **TIMED_WAITING** states:

- the JVM can choose to implement blocking by spin-waiting
- spurious wakeups from **Object.wait** or **Condition.await** are permitted, a thread in **WAITING** or **TIMED_WAITING** state may transition to **BLOCKED** and then **RUNNABLE**.
- besides all these reasons, it just may take some time for the target thread to settle into a blocking thread.

Testing safety

- Identifying easily checked properties that will, with high probability, fail if something goes wrong.
- Not letting the failure auditing code limit concurrency artificially. It is best if checking the test propety does not require any synchronization.

Testing consumer-producer designs. Check that everything put into a queue comes out of it, and nothing else does.

- Using shadow list: add elements when added to concurrent collection, remove when removed, check that nothing is left.
  This approach would distort the scheduling of the test threads because modifying the shadow list would require synchronization
- Compute checksums of the elements that are enqueued and dequeued using an order-sensitive checksum function, and compare them. Test passes if they match
  Works best when there is a single producer and consumer
- For multiple producer-consumer scenario. Compute checksums of the elements that are enqueued and dequeued using an order-insensitive checksum function

We can't check order but don't have to when multuple producers and consumers are used
Either multiple checksums must be calculated or a single shared counter used (which may become a bottleneck)
Ensure that the checksums are not guessable by the compiler
* Test data must be randomly generated
* Provide each thread with custom generator: allows using non-thread safe generators; most are thread-safe and introduce synchronization
* Medium quality fast random generator is enough, no need for high quality randomness, only ensure numbers change from run to run (hashCode and nanoTime, for instance)
If threads are short-running and you start a number of threads in a loop, threads run earlier have a head start on those run later, you can get less real interleavings
* Use CyclicBarrier, make the worker thread and test driver thread wait the barrier at the beginning and the end of their run.
* Scheduling may still run worker threads sequentially
Use a deterministic termination criterion so that no additional interthread communication is needed to figure out when the test is finished
Threads vs cores requirements
* Tests should be run on multiprocessor systems
* There should be more active threads than CPUs so at any given time some threads are running and some are switched out, reducing the predictability of interactions between threads
It is possible that the test never finishes due to a bug or an exception
* Abort tests that do not terminate within some amount of time
* Should distinguish between failed tests and "not waited long enough"

Testing resource management

Should not only test that

- code adheres to specification (does what it's suppose to do)
- but also that it doesn't do what it's not supposed to, such as leak resources: does not hold resources (objects in memory, threads, file handles, sockets, databases connections, other limited resources)

Make snapshot of heap size, run code under resource test, make new snapshot of heap size, compare sizes.

Using callbacks

Callbacks to client-provided code are often made at known points in an object's lyfecycle that are good opportunities to assert invariants.

ThreadPoolExecutor makes calls to **Runnables** and **ThreadFactory**

- addition threads are created when they are supposed to, not when they are not supposed to (using TestingThreadFactory can accomplish that)

- idle threads get reaped when they are supposed to (custom Thread class could record when threads terminate)
- the correct number of created tests

Generating more interleavings

Besides running tests on multiprocessors system and using more threads than CPUs you can:

- use of **Thread.yield** encourages more context switches during operations that access shared state, though the effectiveness of this aproach is platform specific
- use of **Thread.sleep** is slower but more relieable

Testing for performance

Performance tests:

- seek to measure end-to-end performance metrics for representative use cases
- to select sizings emperically for various numbers of threads, buffer capacities and so on

Building performance tests

- Use **CyclicBarrier** to start and stop worker threads
- Provide it with **Runnable** to measure start when starting and finishing a job on barrier
- Build “normalized throughput”/”number of threads” graph for different parameters

Comparing multiple algorithms

Build performance tests for different algorithms.

Measuring responsiveness

Latency may be a more important characteristic than throughput:

- predictible service time
- can answer questions “how many responses are provided within N units of time?”

To measure latency build histograms

- keep track of per-task completion in addition of average time
- timer granularity can be a factor in measuring individual task time; to avoid measurement artifacts we can measure run time of small batches

Size of blocking collection

- small —- heavy context switching, poor throughput

- large (enough) —- low context switching
  fair mode synchronization —- less throughput, but more predictable latency
  non-fair mode synchronization —- larger throughput, less predictable

Avoding performance testing pitfalls

Watch out for a number of coding pitfalls that prevent performance tests from yielding meaningful results

Garbage collection

The timing of GC is unpredictable, there is always the possibility that the GC will run during a mesured test run, which can have a big and spurious impact on effect on the measure time per iteration.

Use 1 of 2 approaches:

- ensure that garbage collection does not run at all during your test (invoke JVM with -verbose:gc to find out)
- make sure that the garbage collector runs a number of times during your run; takes longer test and more adequately reflects real-world performance

Dynamic compilation

JVM first interpretes loaded classes and may compile them and then execute native code. This can bias tests results in two ways:

- compilation consumes CPU resources
- comparing combination of interpreted and compiled code is not a meaningful performance metric
- code may be decompiled or recompiled with different optimizations due to invalidated assumptions about prior optimizations: loading a new class or gathering sufficient profiling data

Prevent compilation from biasing your results:

- **run your program for a long time** (several minutes) so that compilation and interpreted execution represent a small fraction of the total run time
- **use unmeasured "warm-up"** —- execute code long enough to be full compiled. Running HotSpot with -XX:+PrintCompilation prints out a message when dynamic compilation runs, so you can verify that this is prior to, rather than during, measured test runs
- **run tests several times** in the same JVM, **discard the first results as warm-up**, inconsistent results in the remaining groups suggests that the test should be examined further
- running multiple unrelated CPU intensive activities in a single run, it is a good idea to **place explicit pauses between trials** to give JVM a chance to catch up with

background tasks
pauses for **related activites** may give **unrealistically optimistic results**

Unrealistic sampling of code paths

JVM can use information about execution to provide better code, which may generate **different code for the same bytecode** over time

- approximate the usage patterns of a typical application
- approximate the set of code paths used by such an application

Unrealistic degrees of contention

Two very different sorts of work:

- accessing shared data
- thread local computations

Depending on relative proportions of the two types of work, the application will experience different levels of contention and exhibit different performance and scaling behaviours. Concurrent performance tests should try to approximate

- the thread-local computation done by a typical appliction
- concurrent coordination under study

Dead code elimination

Problem

- optimizing compilers are adept at spotting and eliminating dead code (that has no effect on the outcome)
- since benchmarks often don't compute anything they are an easy target for the optimizer
- both in statically and dynamically compiled languages, but more difficult to analyze in the latter (hard to access compiled code)

HotSpot's **—server** compiler is more adept at optimizing dead code. You should still prefer it (more realistic) to **—client** and write code that is not susceptible to dead-code elimination.

- write code so that every computed result is used somehow by your program
- computed results should be unguessable

Complementary testing approaches

Code review

- Expert concurrent programmers are better at finding subtle races than are most test programs

- Platform issues such as JVM implementation details or processor memody models can prevent bugs from showing up on a particular hardware or software configurations
- Can improve quality of comments describing the implementation details

Static analysis tools

FindBugs includes detectors for the following concurrency-related bug-patterns:

- **Inconsistent synchronization** —- a field is accessed frequently but not always with **this** lock held: synchronization policy may not be consistently followed
- **Invoking Thread.run** —- almost always a mistake and must be **Thread.start**.
- **Unreleased lock** —- locks must be released in a finally block
- **Empty synchronized blocks** —- do have semantics under JVM memory model, but frequently used incorrectly, there are better solutions
- **Double checked locking** —- broken idiom for reducing sychronization overhead in lazy initialization that involves reading a shared mutable field wihtout proper synchronization
- **Starting a thread from a constructor** —- risk of subclussing problem, escape of **this** reference
- **Notification errors** —- a synchronized block that calls **notify** or **notifyAll** but does not modify the state is likely to be an error
- **Condition wait errors** —- waiting on a condition (**Object.wait** or **Condition.await**) should be called in a loop, with appropriate lock held and after testing some predicate; anything missing probably indicates an error
- **Misuse of Lock and Condition** —- using a lock as an argument to **syncrhonized** block, calling **Condition.wait** instead of **Condition.await**.
- **Sleeping or waiting while holding a lock** —- sleeping while holding a lock, calling **Object.wait** or **Condition.await** with 2 locks held is a liveness or performance hazard
- **Spin loops** —- code that spins checking for a field for an expected value wastes CPU time, if field is not volatile then the thred is not guaranteed to terminate.

Aspect-oriented testing techniques

AOP can be applied to assert invariants or some aspects of compliance with synchronization policies. It's possible to wrap all calls to non-thread-safe Spring methods with the assertion that the call is occuring in the event thread.

Profilers and monitoring tools

- **Profiling tools**. Timeline for each thread with different colors with the various thread states, where CPU resources are utilized.
- **JMX**
  ThreadInfo class includes thread's current state, if thread is blocked, the lock or condition queue on which it is blocked.
  If the "thread contention monitoring" is enabled, **ThreadInfo** also included the

number of times that the thread has blocked waiting, and the cumulative time it has spent waiting