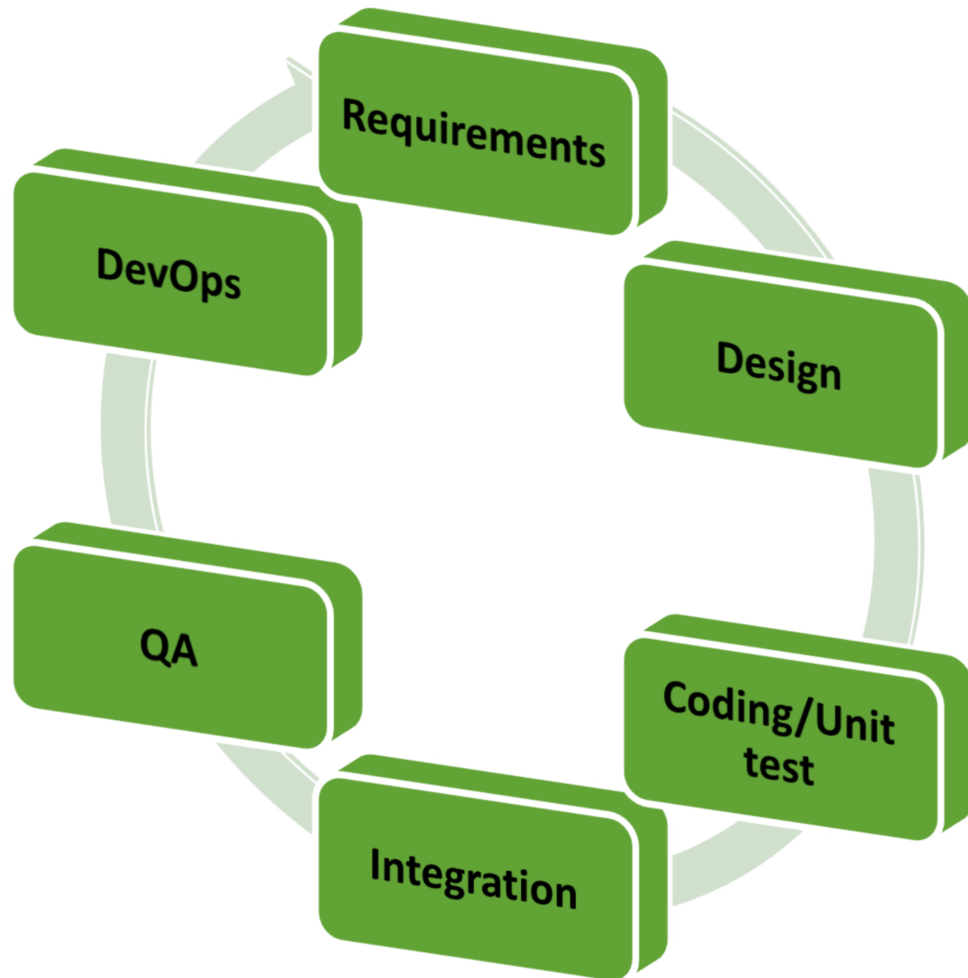
Performance throughout the Development Life Cycle

Performance throughout the Development Life Cycle

- Why aren't performance concerns better integrated into the prevailing software development methodologies (SDMs)?
  - Waterfall (e.g., Grady Brooch (see video), Rational Unified Process)
  - TDD
  - Agile
  - etc.
  - One clue is that Performance is classified as a "non-functional" requirement
    - Functional requirements specify the "behavior" of the software
    - Other non-functional requirements include Maintainability, Accessibility, Reliability, etc.

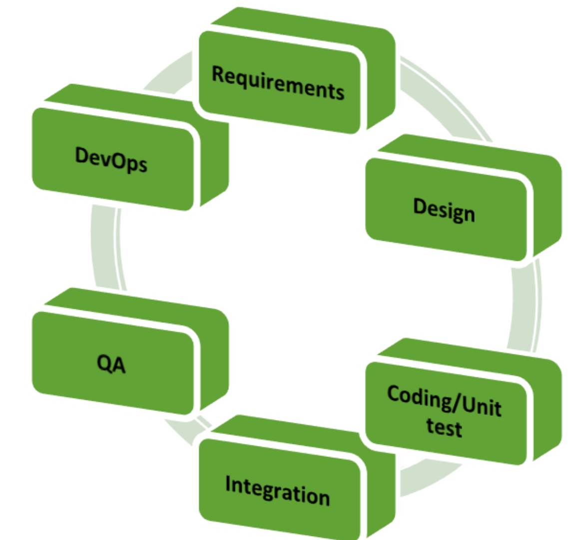
Performance throughout the Development Life Cycle



- Software Development methodologies distinguish distinct phases:
  - Requirements
  - Design
  - Coding/unit test
  - Integration testing
  - QA
  - Maintenance/Operations
- What performance-related activities are associated with each phase?

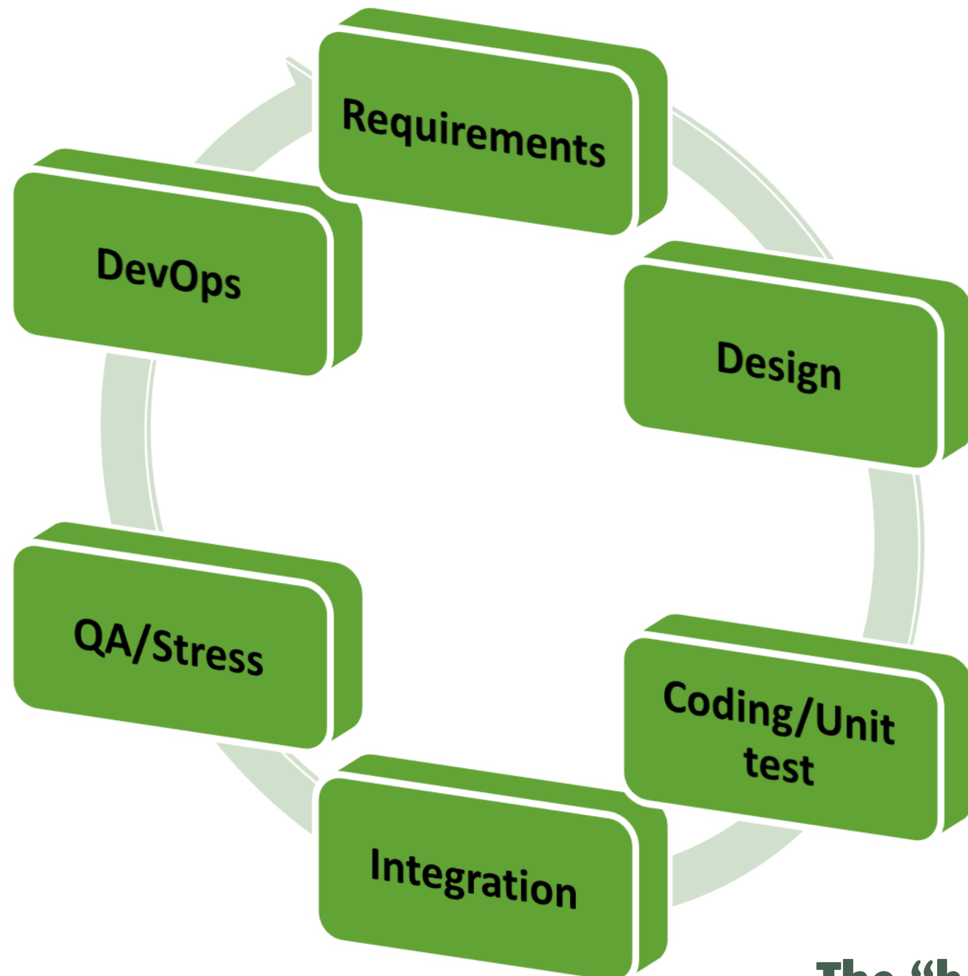
Performance throughout the Development Life Cycle

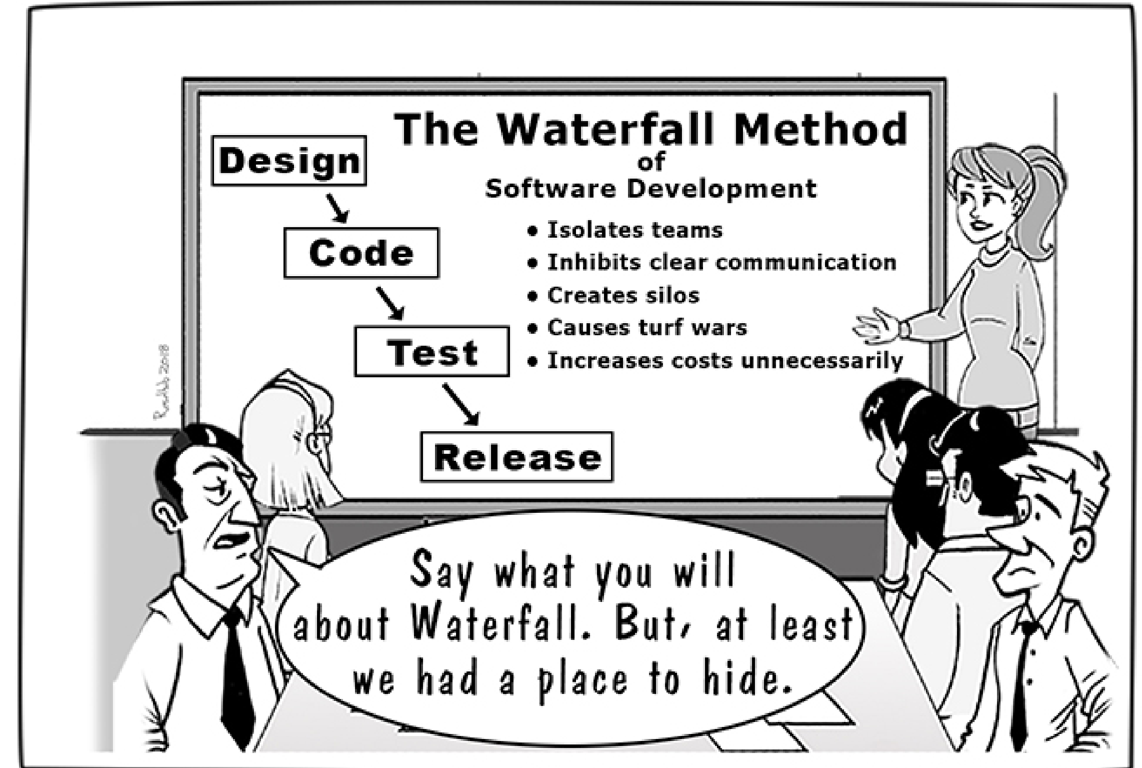
- **Performance-related activities associated with each software development phase:**



Requirements	Understand/Establish performance requirements; Establish a performance budget for each scenario
Design	Design reviews that ensure proposed designs meet performance requirements; Modeling; Prototyping
Coding	Instrumentation; Build and run Timing tests; Performance Quality gates
Integration	End-to-end performance testing
QA	Load testing; stress testing
DevOps	Performance monitoring

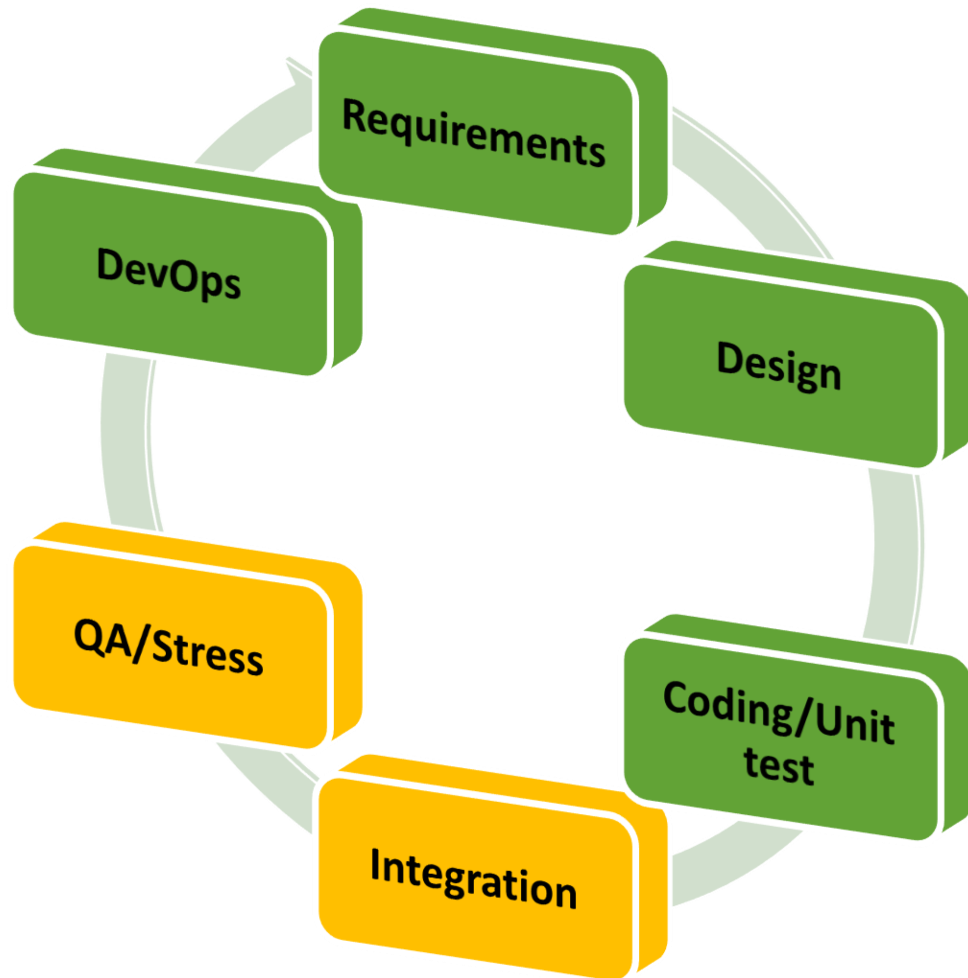
Performance throughout the Development Life Cycle





The "benefits" of Waterfall, according to advocates of Agile & DevOps

Performance throughout the Development Life Cycle



- **Organizational Immaturity:**
  - Performance testing is often relegated to later stage heroics, aka, "fire-fighting" by senior developers
  - Performance stress testing serves as a final obstacle that must be hurdled prior to release
- **Prevailing Trends challenge this status quo:**
  - Continuous integration
  - Agile

Performance throughout the Development Life Cycle

- Why aren't performance concerns better integrated into the software development methodologies?
  - not recognized in "Design Patterns" either
  - with the exception of Loosley & Douglas:
    - Workload
    - Efficiency
    - Locality
    - Sharing
    - Parallelism
    - Trade-off

Performance throughout the Development Life Cycle

- Loosley & Douglas, *High Performance Client/Server*, 1998.
  - proposed Design Patterns for performance

Workload	Minimize the total processing load
Efficiency	Maximize the ratio of useful work to overhead
Locality	Group-related components based on their usage
Sharing	Share resources without creating bottlenecks
Parallelism	Use parallelism when the gains outweigh the overhead/costs
Trade-off	Reduce delays by substituting faster resources

Performance throughout the Development Life Cycle

- Smith & Williams, “More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot”, 1998.
  - some of the proposed anti-Patterns that reduce performance

Empty Semi-Trucks	When an excessive number of requests is required to perform a task
Roundtripping	Maximize the ratio of useful work to overhead
The Ramp	Occurs when processing time increases as the system is used.
One-Lane Bridge	Processes are delayed while they wait for their turn at a single-use bottleneck
Traffic Jam	Occurs when one problem causes a processing backlog that persists long after the initial cause
More is Less	when a system “thrashes” rather than accomplishing real work because there are too many processes relative to available Resources

Performance and the Development Life Cycle

- Concern about "premature optimization" that defers tuning efforts until the code base is stable is valid – up to a point
- Senior technical staff are heroes that parachute in to investigate & fix performance problems in the latter stages of a project
- But if there is a fundamental design flaw that was baked in early…
  - More expensive to fix it in the later stages
  - a serious enough "flaw" can delay (or even torpedo) the release

Performance and the Development Life Cycle

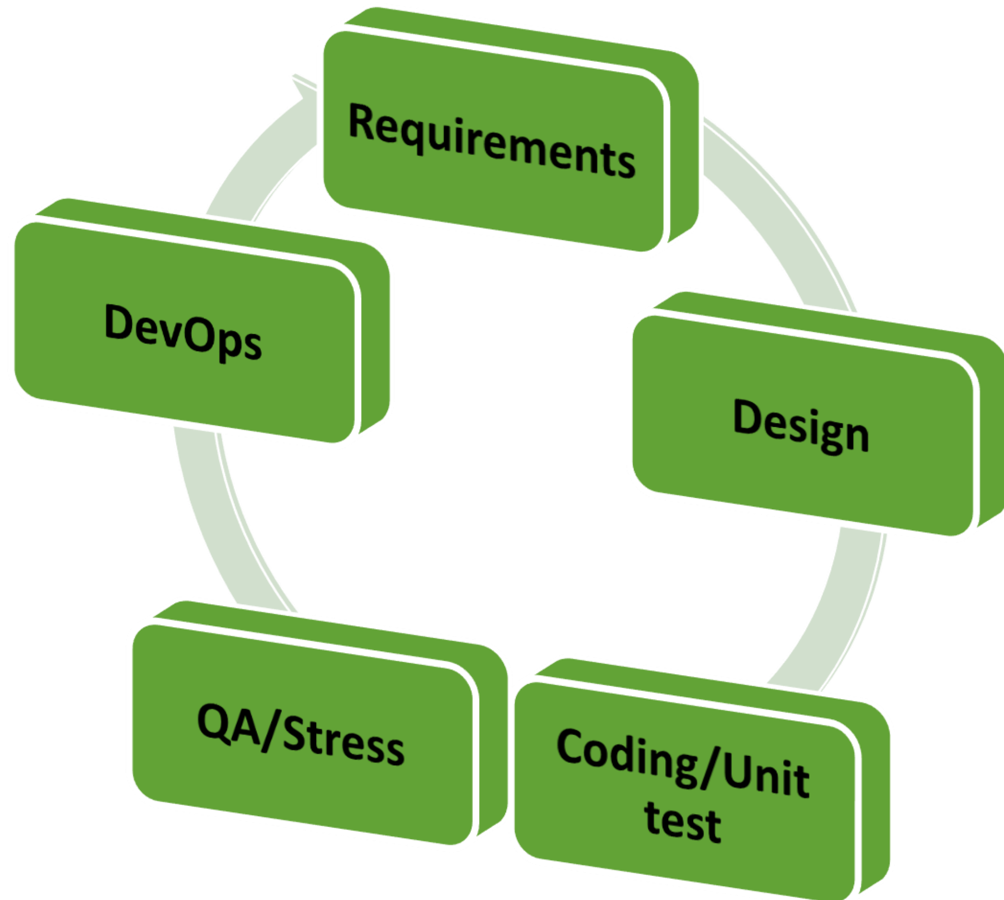
- Here are some things that have been tried:
  - Performance “anti-patterns” approach
    - single-lane bridge
    - long path
    - resource bottlenecks
      - e.g., Resource R is a candidate bottleneck if:
        1. it is used by the majority of scenarios,
        2. many scenarios that use it are too slow,
        3. it is near saturation (>80% of its units are busy),
        4. resources that are acquired earlier and released later are also near saturation
    - layered software bottlenecks

Performance throughout the Development Life Cycle

- Other attempts:
  - Generate a model from the specification
    - Markov models of sequence, to queueing models
    - annotated UML ⇨ Queueing model

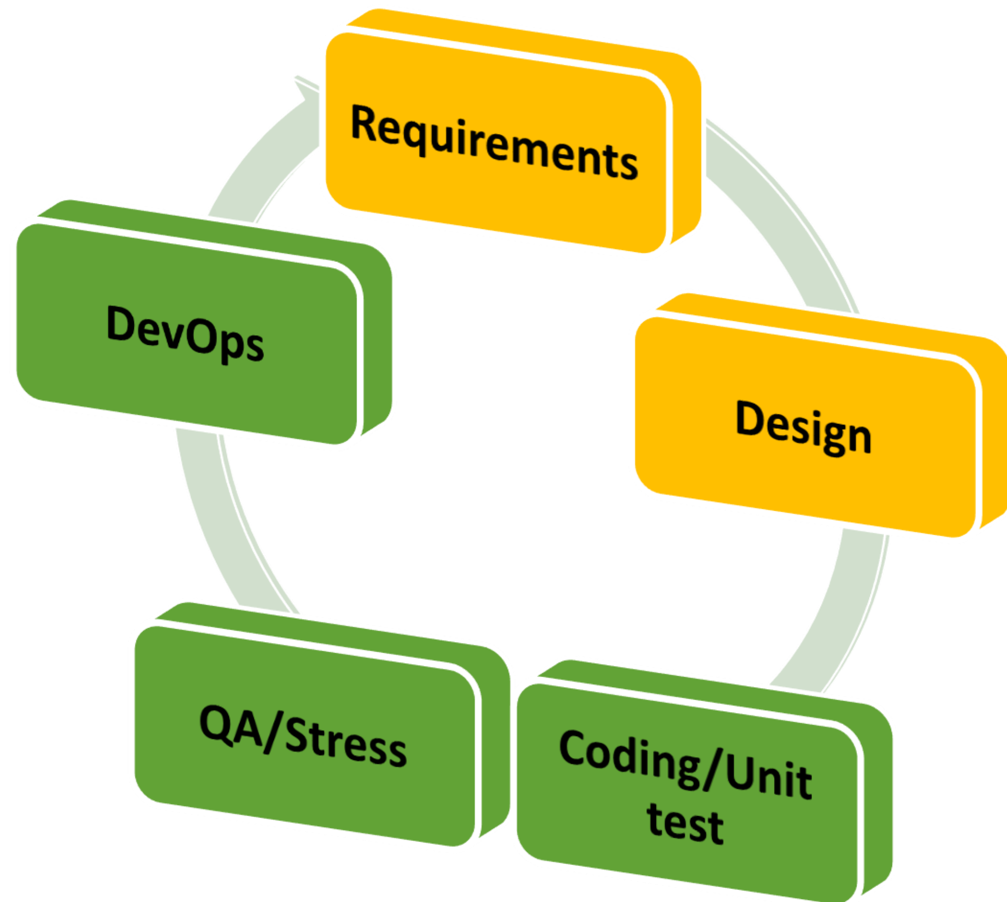
    - Issues:
      - Validation
      - Using static analysis to parameterize a model rather than wait for actual run-time measurements

Performance throughout the Development Life Cycle



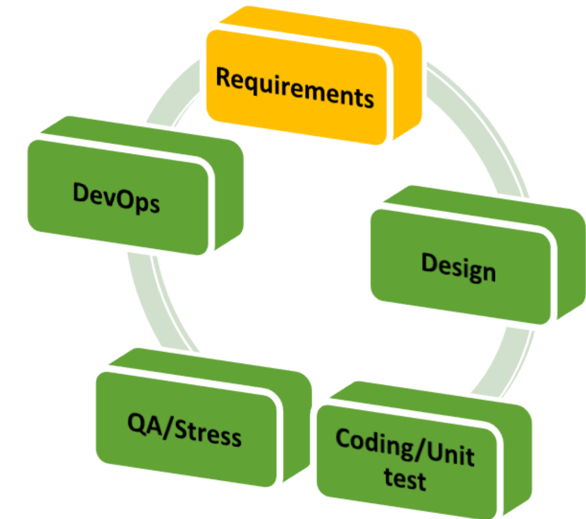
- Day of Reckoning when a major release misses its performance objectives by a wide margin
- *Proactive* performance management
- Continuous improvement model
  - Monitor and report on progress/risk against performance objectives throughout the life cycle

Performance and the Development Life Cycle



- Continuous improvement model
  - Set *achievable* Performance goals initially based on requirements
    - Hardware limitations
    - Scalability limitations
    - costs of Parallelism
  - So they can inform design decisions (and early stage scouting)

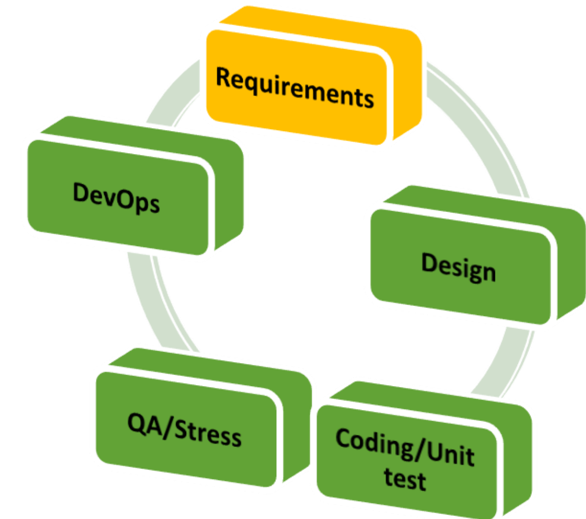
Performance Requirements/Goals



- Establishing Performance requirements
  - New features
    - scalability goals
    - response time goals (focus on the User Experience)
  - Improving existing features where performance is currently a dissatisfier
    - Is this a competitive issue?

Performance Requirements/Goals

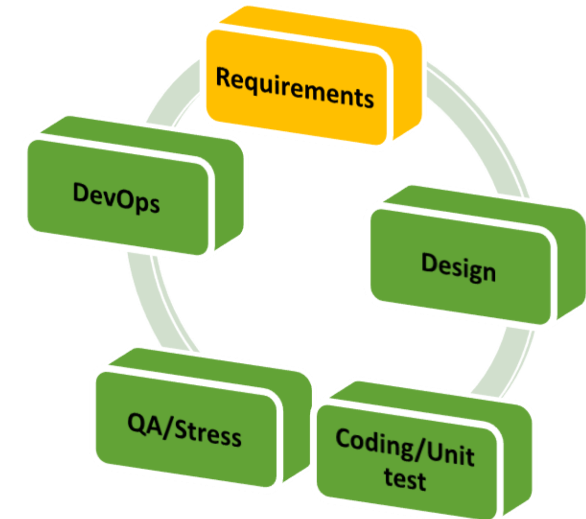
- Introduction to Scalability Testing
  1. Identify the ***scalability factors*** that impact the performance of your feature
  2. Develop a performance budget for the feature based on the costs associated with these scalability factors
  3. Built a set of tests that measure the performance of the feature across the key scalability factors (coverage)
  4. Run the tests and ensure the performance of the feature remains within the allotted budget



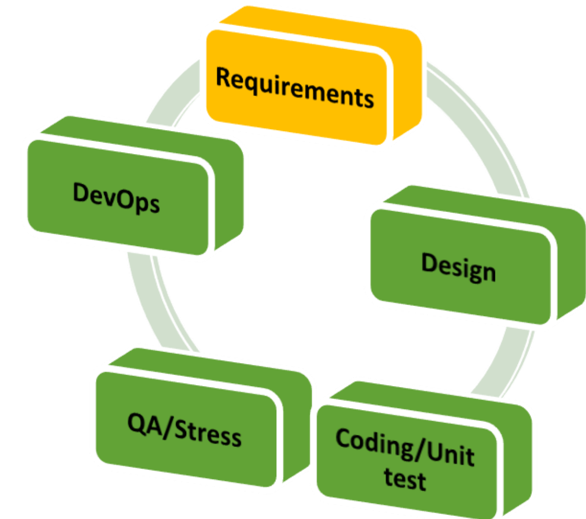
Scalability Testing

- ***Scalability factors*** that you believe impact the performance of your feature
  - Most of us have an implicit mental model for how the application performs at scale
  - It is only a theory until you prove it, so test this Hypothesis!

  - e.g., consider a Compiler:
    - number of lines of source code in a file,
    - number of files in a Project
    - number of local variables in a procedure,
    - number of external variables
    - number of Projects
    - etc.



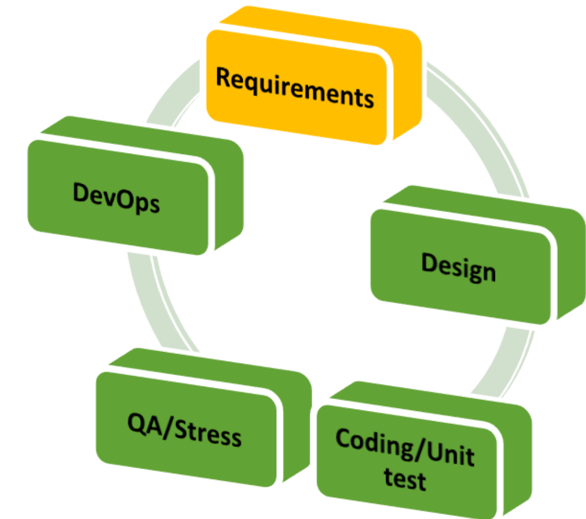
Scalability Testing



- ***Scalability factors*** that you believe impact the performance of your feature
  - It is a theory until you prove it, so test your Hypothesis!
  - More than one scalability factor?
    - Number of controls on a web or native application form (**C**)
    - Number of elements in a list of tree control (**E**)
    - What test coverage do you need?

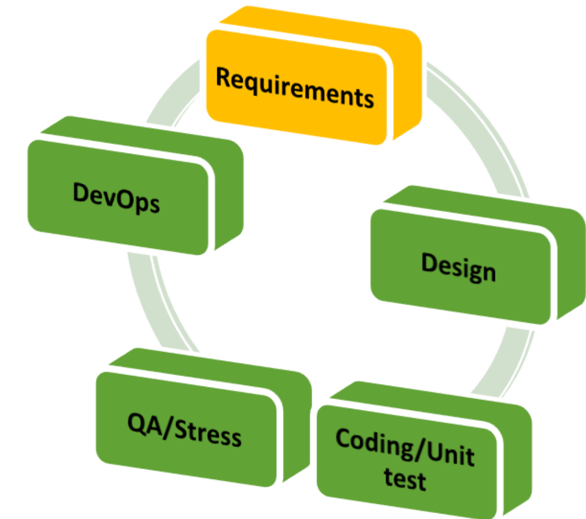
Is the full test matrix C *x* E ?

Scalability Testing



- **Identifying the Scalability factors associated with your feature?**
  - **e.g. consider the performance of .NET Collection classes**
    - `Dictionary`
    - `List`
    - `SortedList`
    - `SortedDictionary`
    - `Queue`
    - `Stack`
    - **etc.**

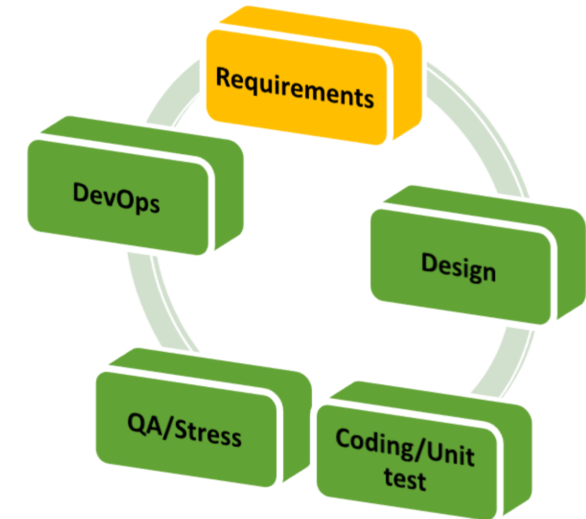
Scalability Testing



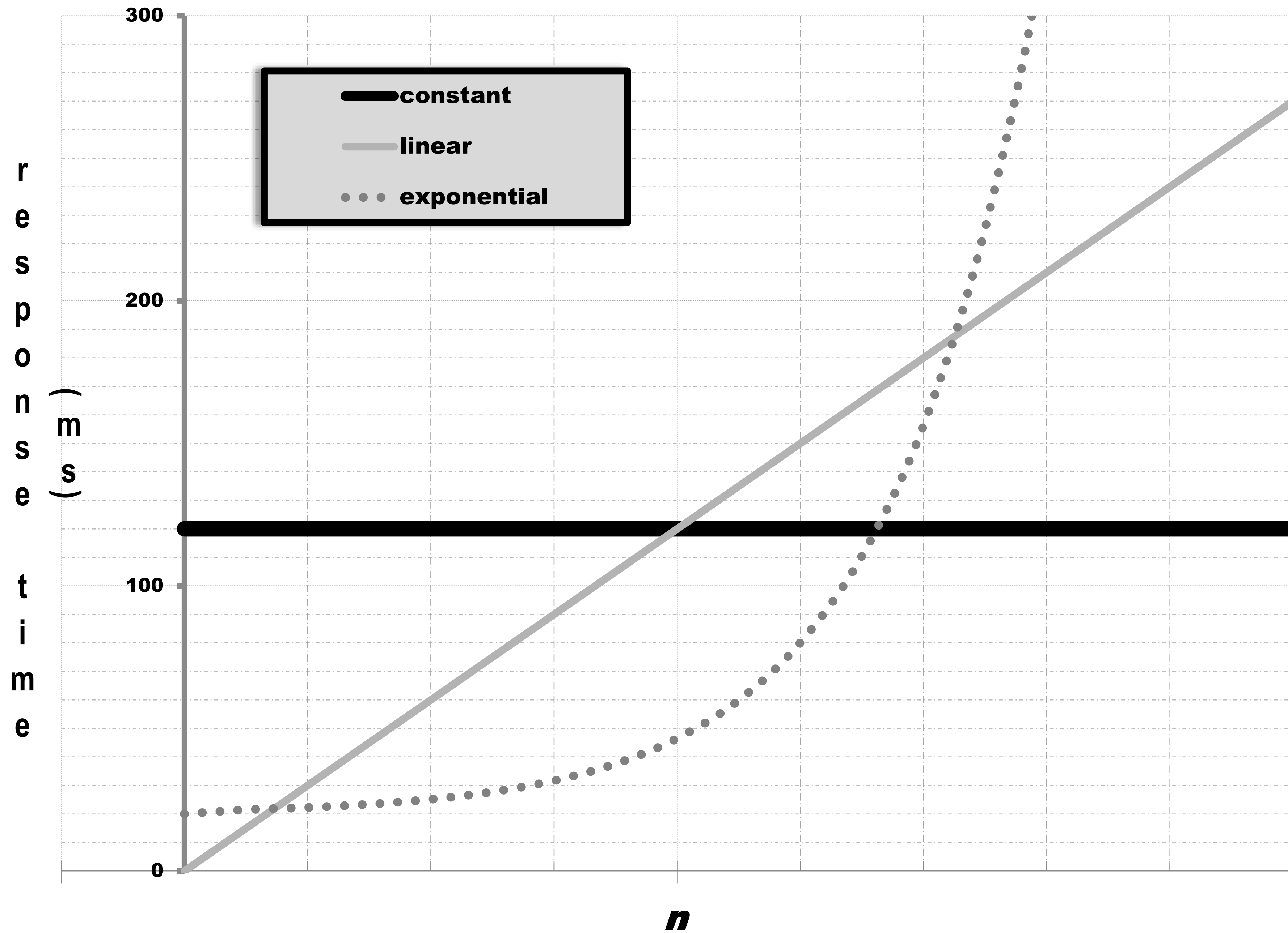
- Identifying the Scalability factors associated with your feature?
  - e.g. consider the performance of .NET Collection classes
  - scaling Factors:
    - # of elements in the collection (i.e., **cardinality**)
    - Access pattern:
      - inserts
      - deletes
      - searches
      - enumerate elements

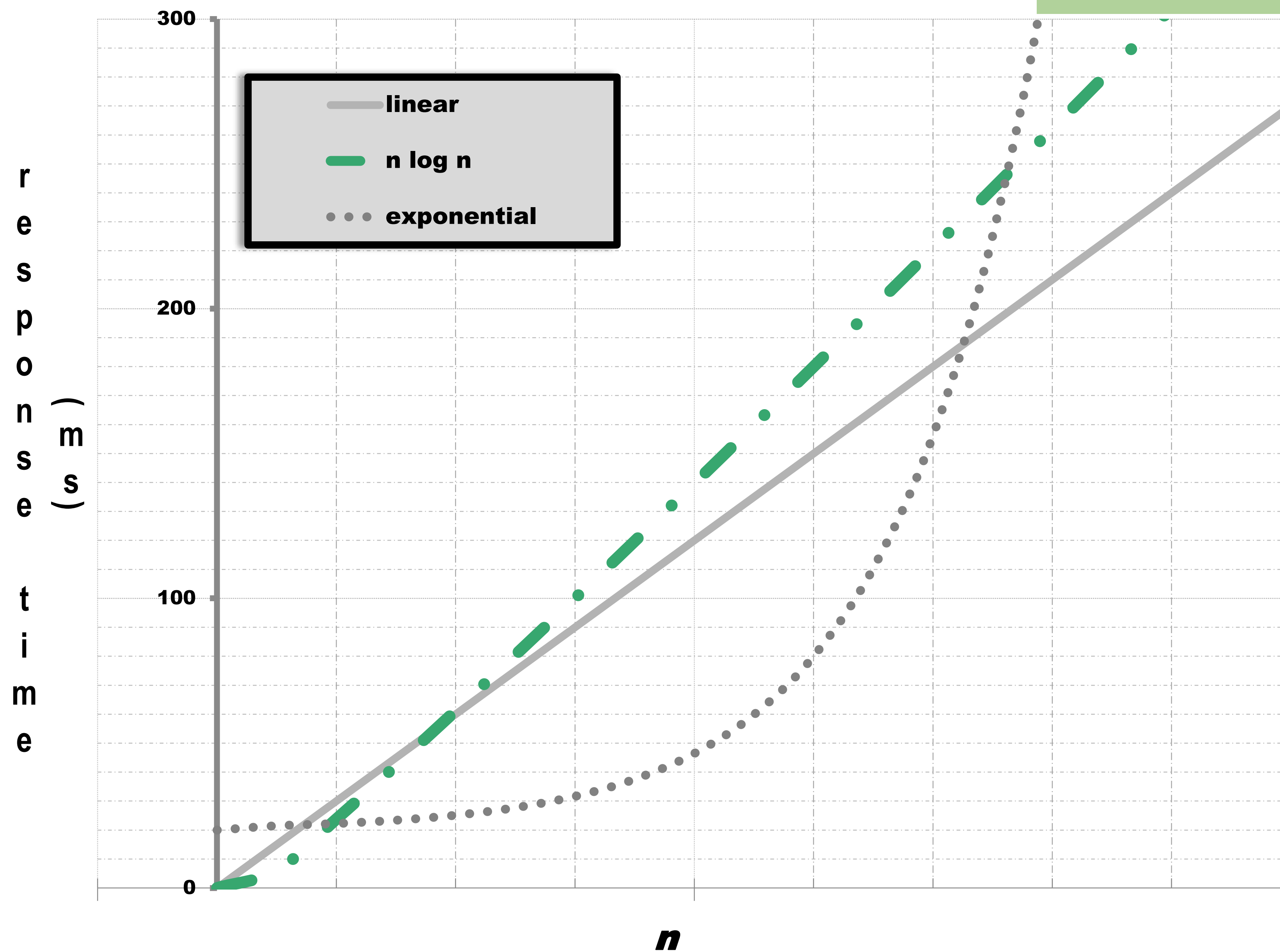
Scalability Testing

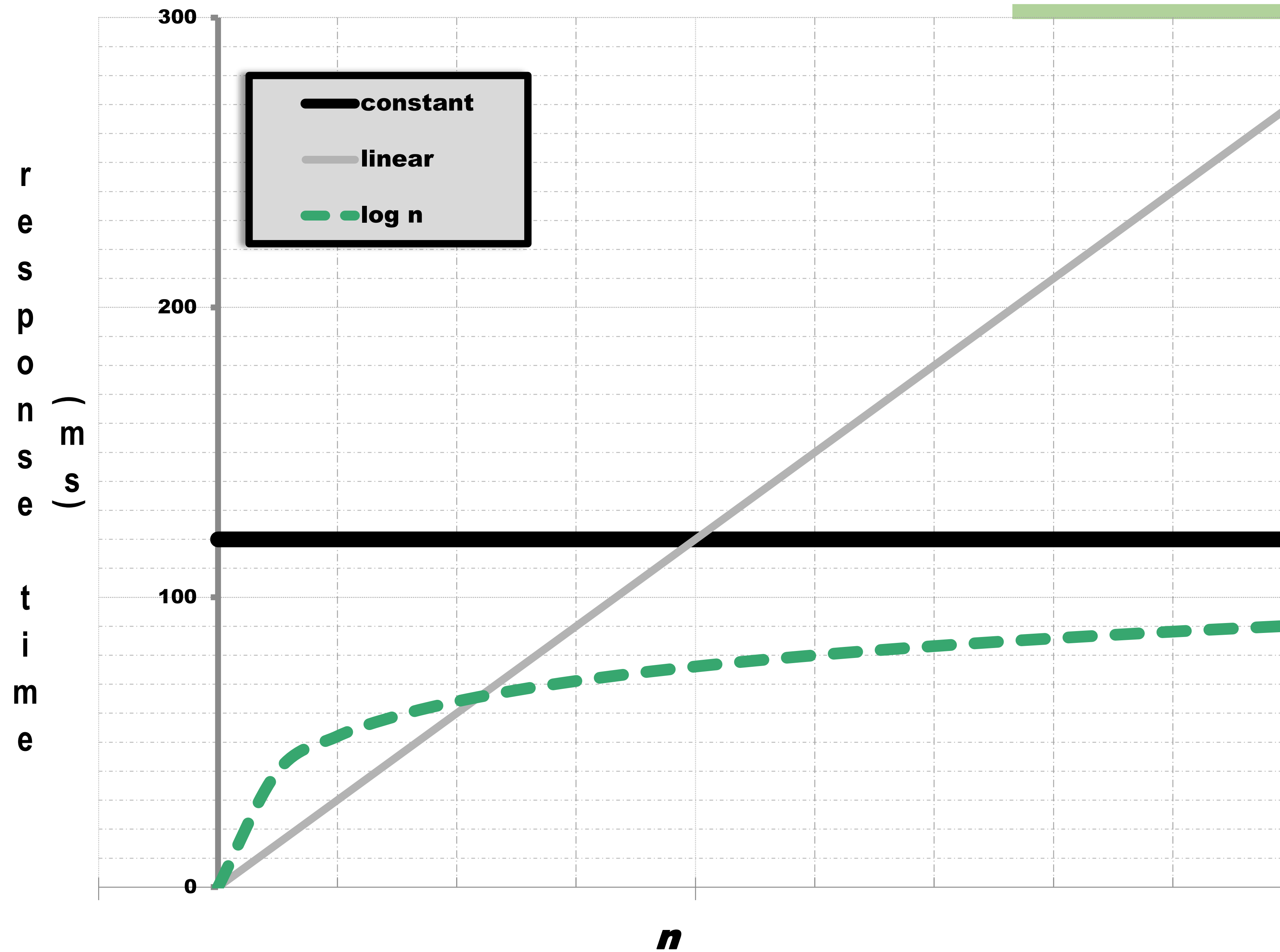
- What are the Scalability factors associated with your feature?
  - Hypothesis that requires validation/testing
  - scalability Models*
    - uniform or constant
    - linear
    - exponential
    - combinatorial
    - log linear
    - $\log_n$



* Algorithms and Complexity

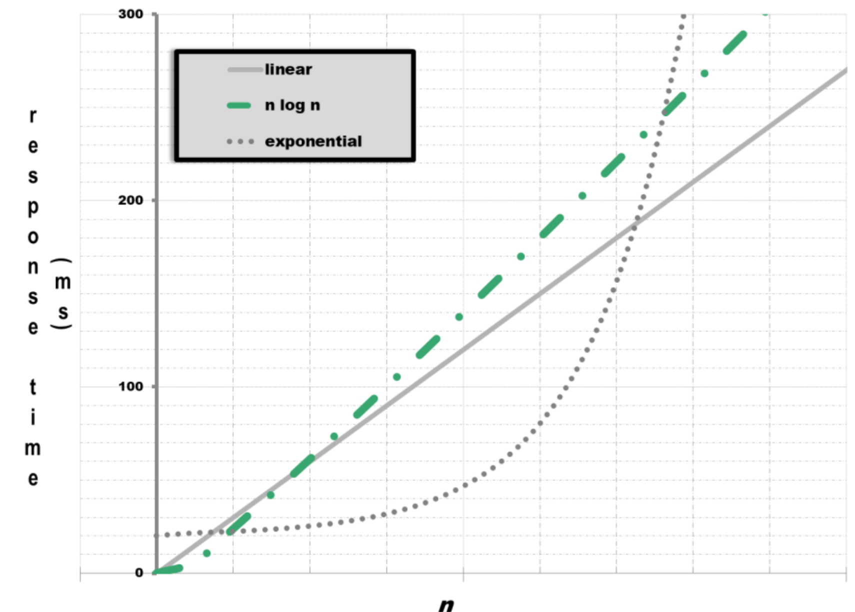

300
200
100
0
constant
linear
exponential
response time (ms)
n


300
200
100
0
linear
n log n
exponential
response time
(ms)
n

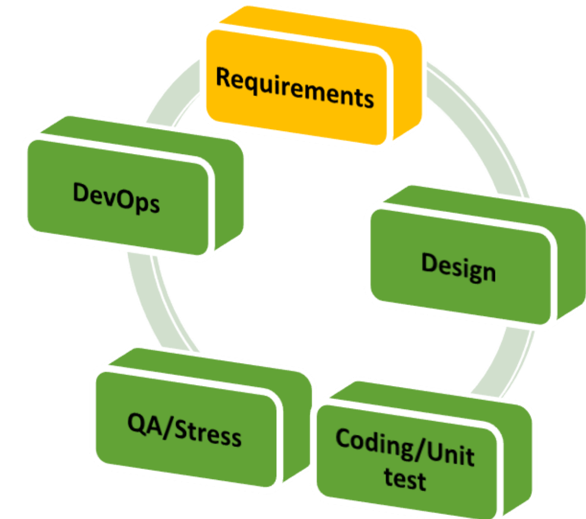

constant
linear
log n
300
200
100
0
response time (ms)
n

Scalability Testing

- Scalability Models
  - uniform/constant/deterministic : Table look-up using hash codes
  - linear : search an unordered list or array
  - $\log_n$: binary search of an ordered list/tree
  - log linear: sorting
  - exponential : NP-completeness
  - combinatorial : NP-completeness



Scalability Testing



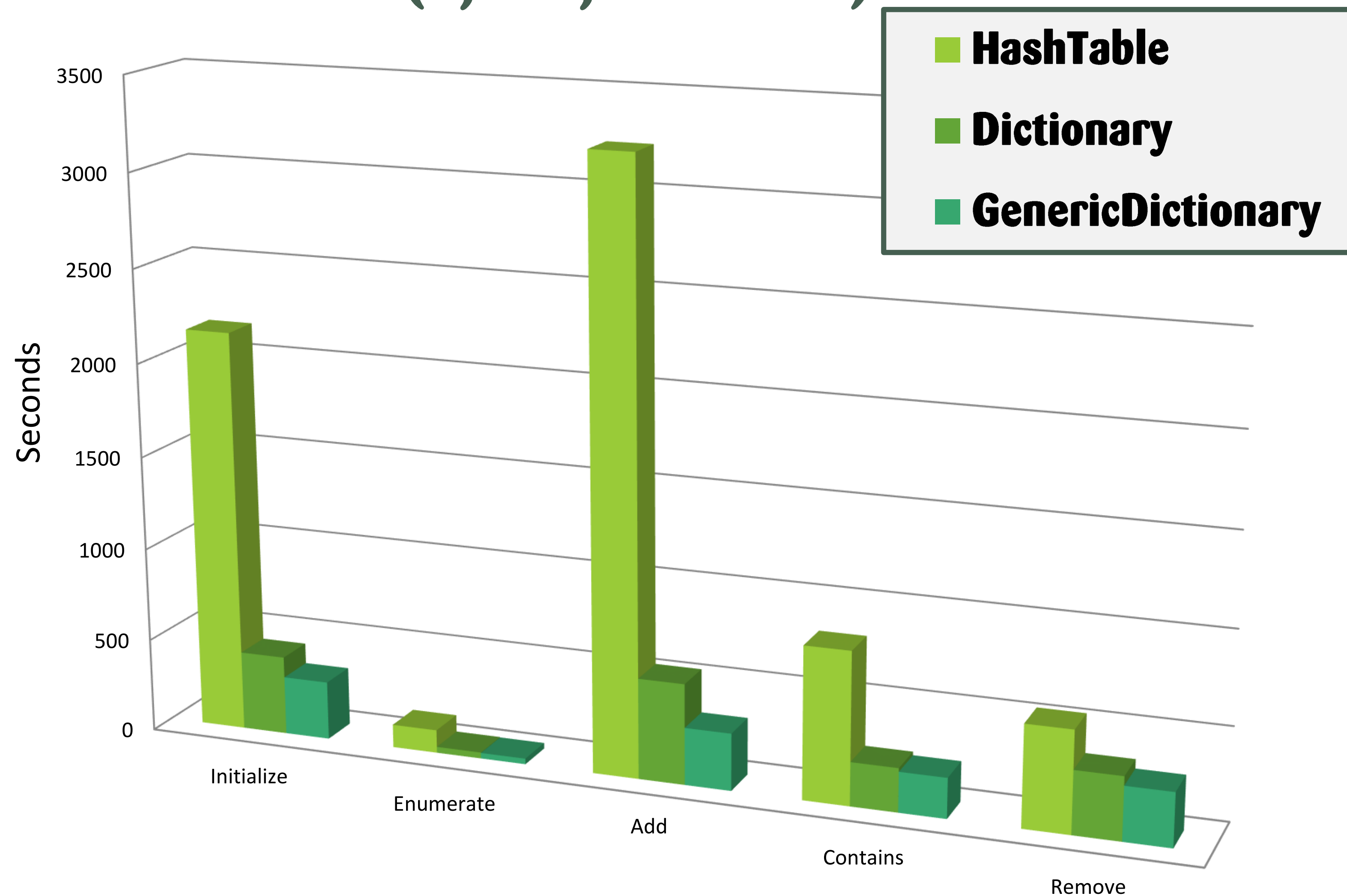
- **Identify the Scalability factors**
  - **e.g. consider the performance of .NET Collection classes**
    - `ICollection<T>Interface`
      - `Add()`
      - `Remove()`
      - `Clear()`
      - `Contains(); ContainsKey()` – **equivalent to Search()**
      - `GetEnumerator();`
        - `Current`
        - `MoveNext()`

Scalability Testing

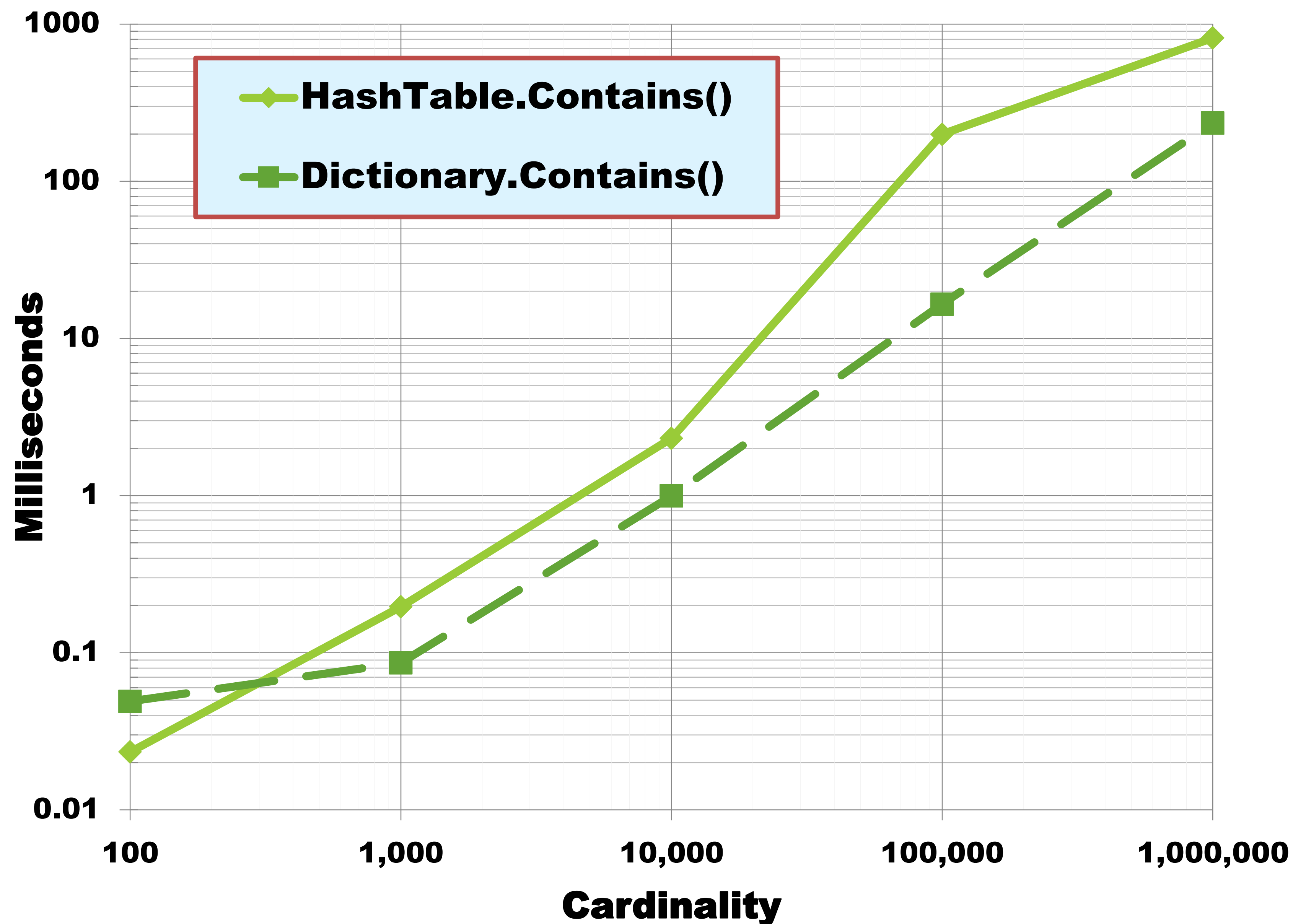
- Scalability factors associated different .NET Collection classes

Size	Method				
	Init	Add	Delete	Contains	Enumerate
100					
1000	√	√	√	√	
10,000					
100,000	√	√	√	√	
1,000,000					
10,000,000	√	√	√	√	√

Scalability of the Collection classes…

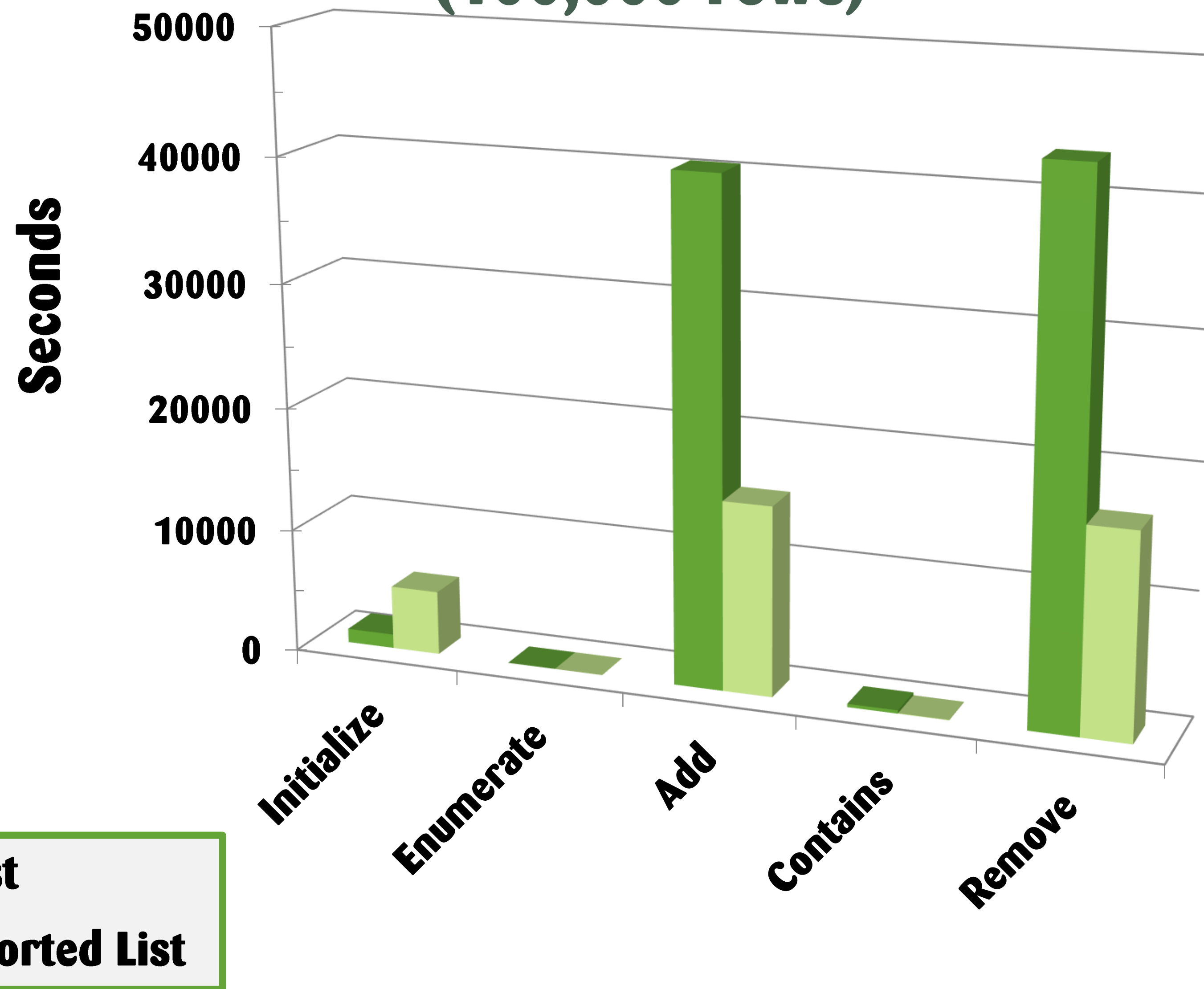


Scalability of the Collection classes…



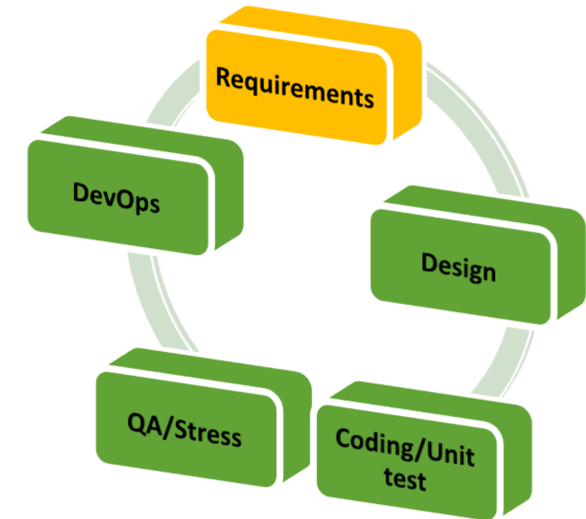
Note:
Plotted against a Log/Log scale

Scalability of the Collection classes…



Scalability Testing

- **Discussion:**

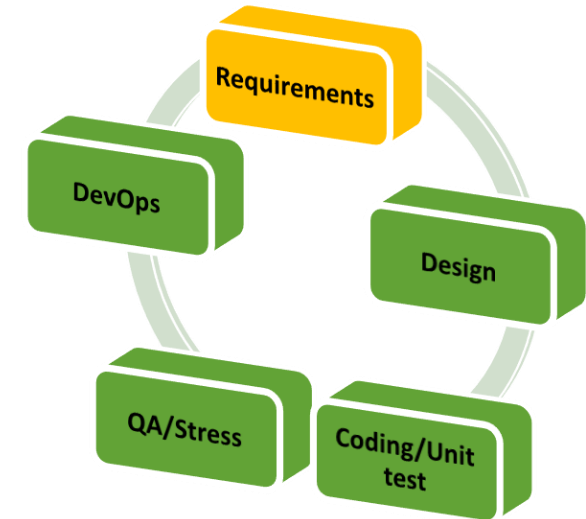


1. How much of the actual response time do the factors from the scalability Model explain?

2. Are more factors required for a better model?
   - Note: 3 factors makes the full test matrix C x D x E ?

3. Why not just use curve-fitting?

Response Time (UX)



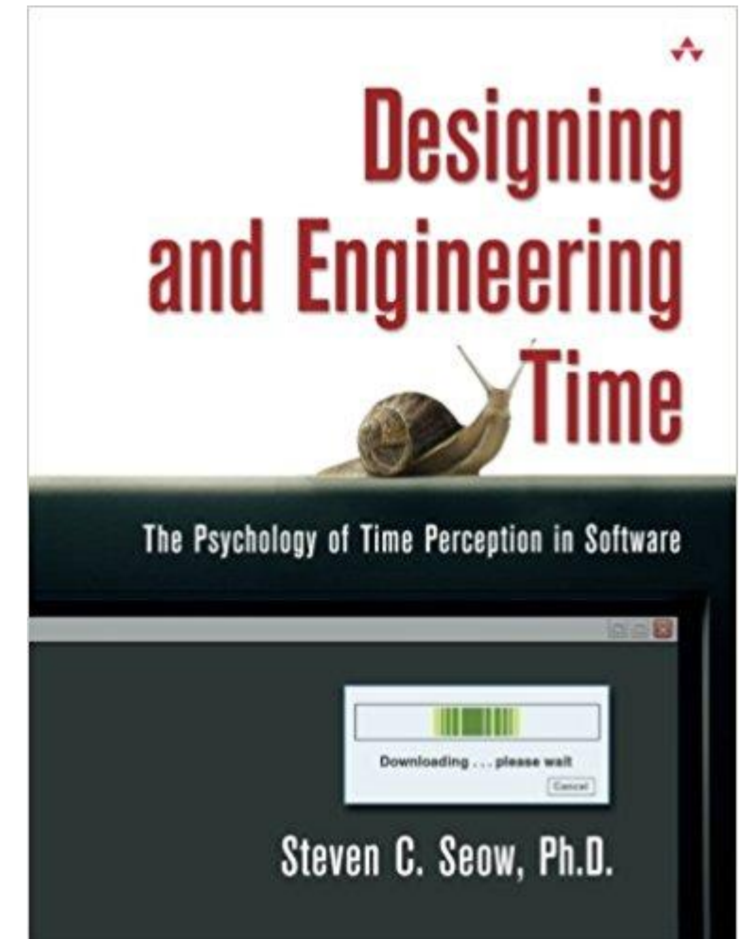
- Application response time is an important aspect of the User Experience
  - often highly correlated with User Satisfaction, Fulfillment rates and Abandonment rates
- Given a new UX scenario, how can I set an achievable Response Time goal?
- Are there industry Standards and Best Practices, derived from Human Factors research, that can inform this decision?

Psychology of Human Time Perception

- Application response time is an important aspect of the User Experience
- Given a new UX scenario, how can I set an achievable Response Time goal?
- Are there industry Standards and Best Practices, derived from Human Factors research, that can inform this decision?

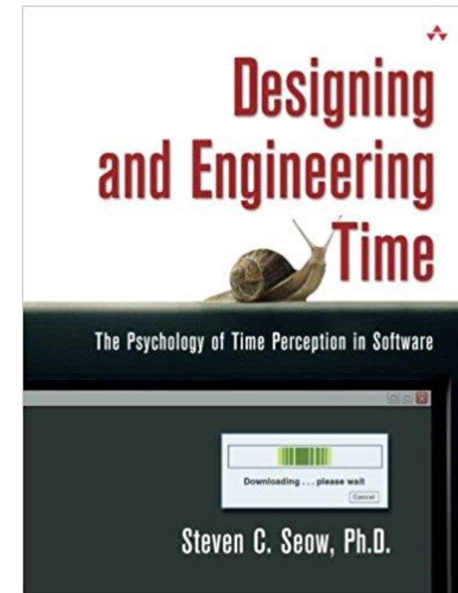
Psychology of Human Time Perception

- **Ben Schneiderman, "Response time and display rate in human performance with computers," *ACM Computing Surveys*, Sept. 1984.**
- **Barber & Lucas, "System Response Time Operator Productivity, and Job Satisfaction", *CACM*, 1983.**
- **Steve Seow, *Designing and Engineering Time*, 2008.**



Available at https://www.amazon.com/Designing-Engineering-Time-Psychology-Perception/dp/0321509188

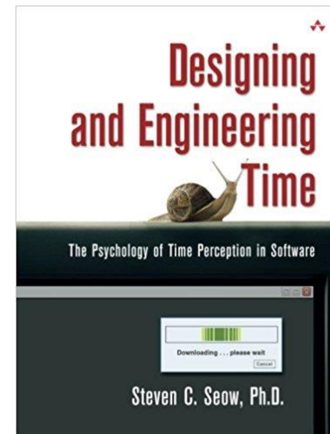
Psychology of Human Time Perception



- While Wait Time **duration** can be measured objectively, waiting is experienced subjectively.
- Time perception has physiological components, but time perception is based mainly on expectations.
  - Seow argues that Weber's Law of a **Just Noticeable Difference** (JND) also applies to time perception

Duration JND ~ 20%

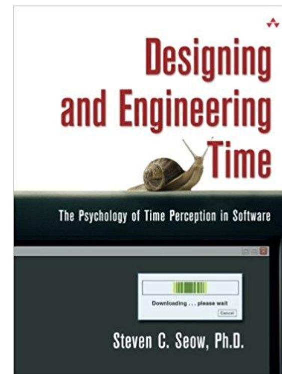
Psychology of Human Time Perception



- Other Human Factors research shows Users experience a higher state of "anxiety" when response times are poor or erratic.
  - Error rates increase in the face of unexpected variability
  - for additional insight, see : Kahneman, *Thinking, Fast and Slow*

- Consider some long running scenario:
  1. Can we speed it up?
  2. Can we make it appear faster?
  3. Can we get customers to tolerate better the current level of performance?

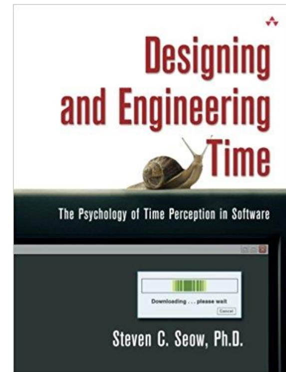
Psychology of Human Time Perception



- **Seow suggests four categories of interactive response time:**

Category	Range in Seconds	
Instantaneous	0.1 – 0.2	Applies when you are simulating a physical operation; e.g., a key down event or a button push; all animations, any real-time, interactive gaming application
Immediate	0.5 – 1	Seamless because it is similar to human-human interactions
Continuous	2 – 5	Tolerable because it is still within the limits of normal human-human interaction
Captive	7 – 10	Slow, but still within an acceptable range. However, response time > 10 seconds cause loss of attention and users start to drift away

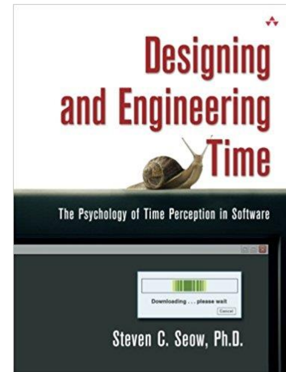
Psychology of Human Time Perception



- Seow's four (actually 5) categories are consistent with "industry standard" guidelines

- *Department of Defense Design Criteria Standard: Human Engineering*. MIL-STD 1472F. Available at http://hfetag.dtic.mil/docs-hfs/mil-std-1472f.pdf.)
- *Department of Defense Technical Architecture Framework for Information Management* (TAFIM). Volume 8: DoD Human Computer Interface Style Guide.
- Smith, S. L and J. N. Mosier (1986). *Guidelines for Designing User Interface Software: ESD-TR-86-278*. Bedford, MA: The MITRE Corporation.

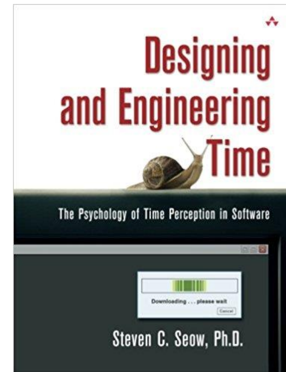
Psychology of Human Time Perception



- Performance is relative to expectations
- “Captive” category:
  - Instead of being held in captivity, Users need to be able to escape out of interactions experiencing long delays

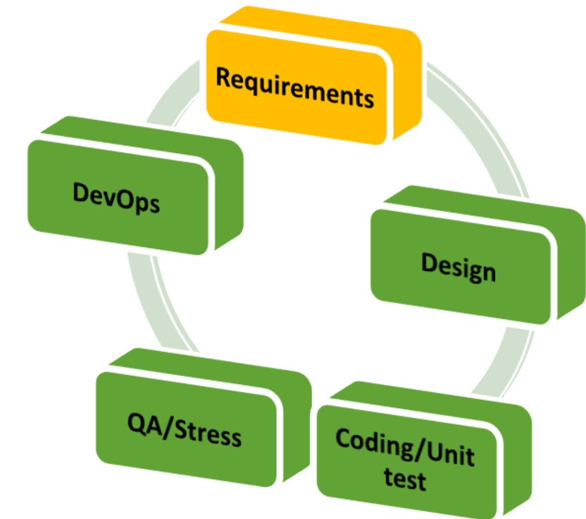
- > 10 seconds Response Time
  - Seow recommends trying to rein in User dissatisfaction at this point:
    - use of a Progress bar
    - send other feedback

Psychology of Human Time Perception



- What if the customer is captive to your app?
  - e.g.,
    - using an online banking application to pay a bill
    - purchase something on Amazon using an Amazon app
  - Customers are often captive to your application!
  - So, if you cannot improve it, you can at least try to make the waiting experience (relatively) more pleasant
    - Even if it takes more than 10 seconds to process the Request!
  - Best Practice is to assume an intelligent, adaptive customer…
    - Someone just like you ☺
  - Remember, familiarity with the application creates a set of expectations with regard to its performance
    - Corollary: Instrument your application!

Response Time (UX) requirements

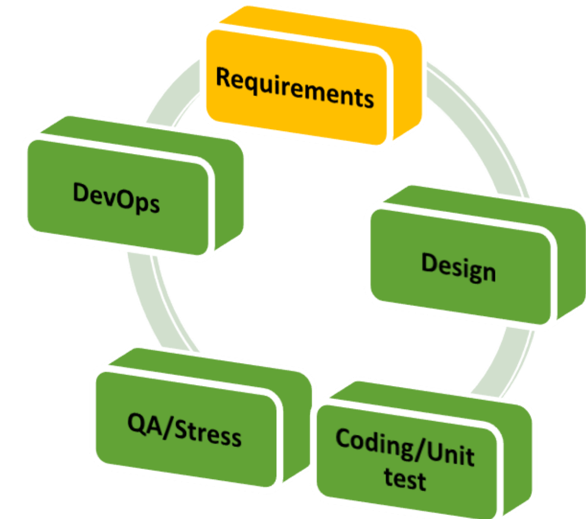


- For new UX scenarios,
  - Break into separate Request:Response sequences
  - For each Request:Response sequence,
    - prepare a resource budget (UI, CPU, IO, Network)
    - if the budget exceeds 1-2 seconds, then decompose the scenario further
  - Also, prepare an "expert" interface where the experienced User can execute the scenario with fewer, but more time-consuming interactions

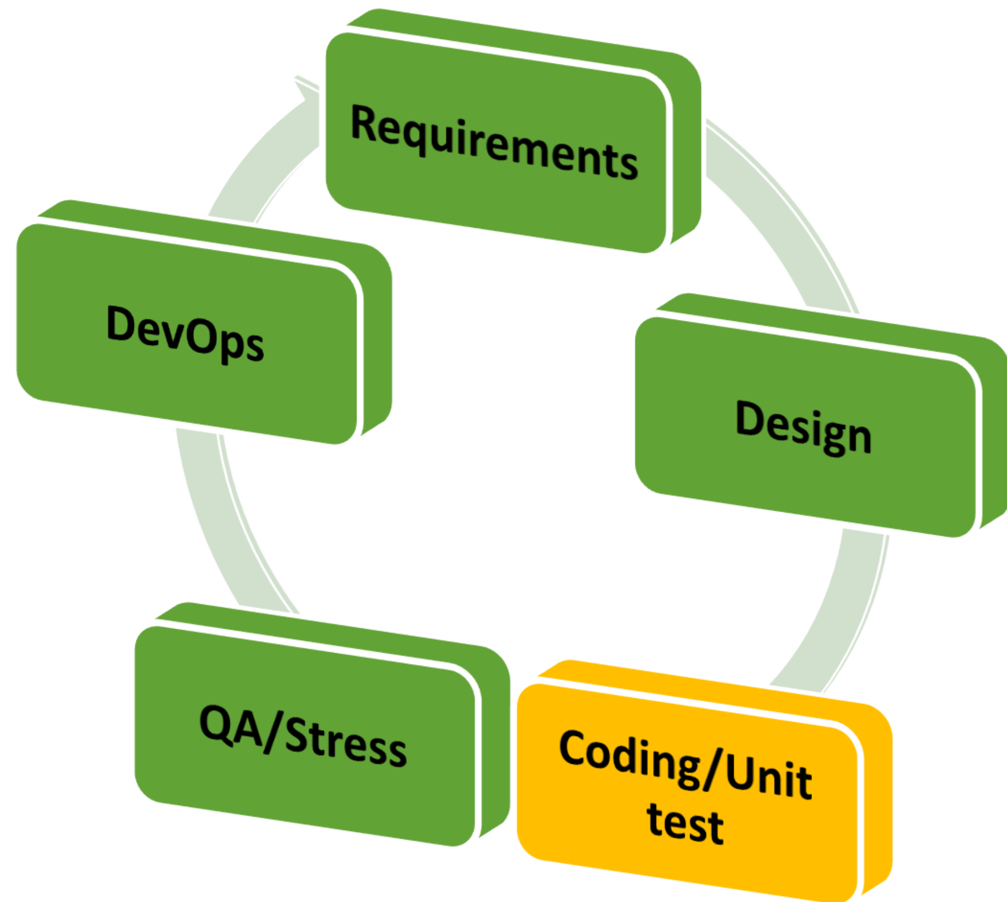
Response Time (UX)

- **For an existing scenario,**
  - Gather timings for the current system: these establish a **baseline**, setting current customer expectations
  - Any performance improvement must be noticeable (Weber's Law);
    - i.e., the new performance baseline must be at least 20% faster than before

- **Weber's Law cuts both ways!**
    - Performance regressions that are < 20% are also not perceptible

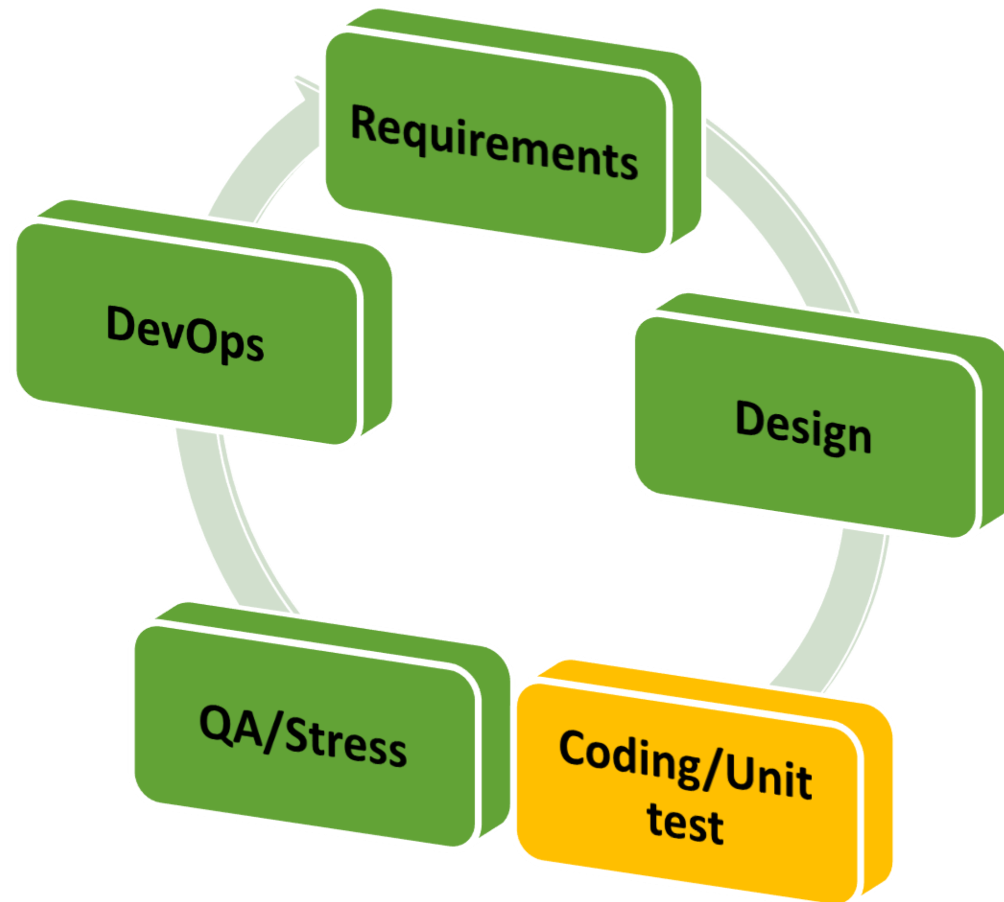


Performance and the Development Life Cycle



- Continuous improvement model
  - Performance tests performed early and often so progress against goals can be monitored
  - Automated performance testing
    - Instrument early and test often
    - Every unit test can also be a Timing test!
  - Performance Quality gates to detect problems ***prior to integration***

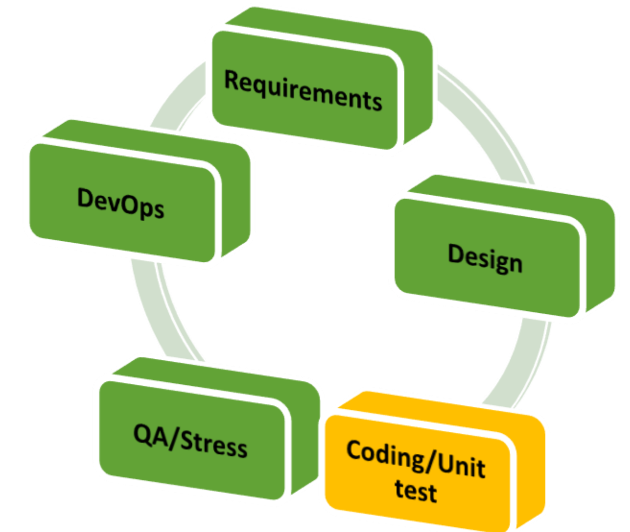
Performance and the Development Life Cycle



- Two types of performance tests
  - Timing tests
    - shows current performance levels, compared to the performance objectives
    - every unit test can also be a timing test
  - Performance Quality gates to detect problems ***prior to integration***
    - Understand how any new code impacts current performance levels
    - Block integration of new code into Main branch that impairs performance

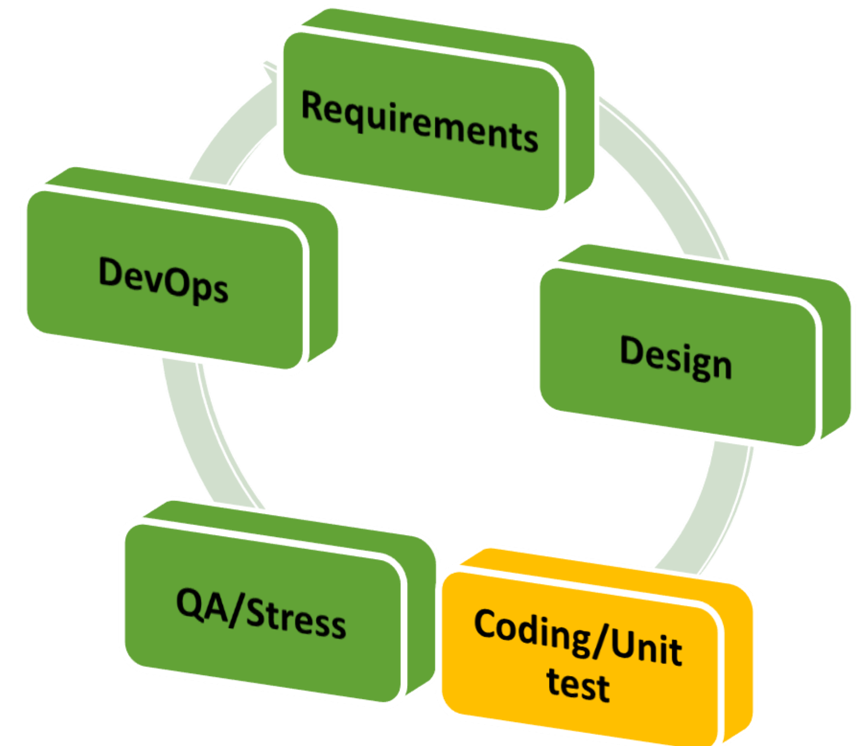
Performance and the Development Life Cycle

- Instrument the application
  - **Minimally intrusive!**
  - Easily understood semantics
  - Simple, intuitive interface
  - Embedded timing code should be standardized
    - enables tool development (e.g., service level reporting)
  - Needs to work in the lab, but being available to work in production (dynamically) is a major benefit



Performance and the Development Life Cycle

- Instrument the application
  - Lightweight, but accurate CPU & Execution time
  - Event-oriented:
    - `Scenario.Start() :Scenario.End()`
  - Minimally intrusive
- Test early and test often
- Goal:
  - Make every unit test a timing test!

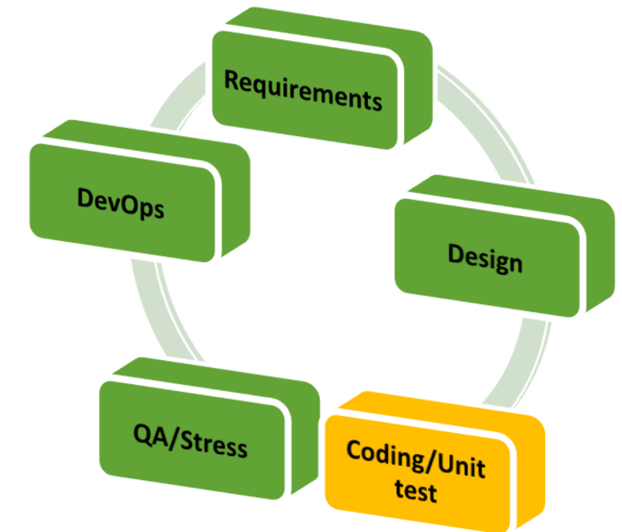


Performance and the Development Life Cycle

- Instrument the application
  - Examples:
    - ARM (HP and IBM joint initiative)
    - Custom Instruments in MacOS (link)
    - HTTP timing interface: RUM
    - Visual Studio MeasurementBlock
    - Scenario class library (.NET)
      - Event-oriented

```
Scenario.Start() :Scenario.End()
```

      - hierarchical (i.e., parent : child)
      - dynamic (leveraging ETW)



Tools for instrumenting your application

- Hardware clocks
  - e.g.,
- **rdtsc**
  - mnemonic for **read timestamp counter** instruction
  - Introduced with the Pentium II
  - latency < 100 cycles (much faster on AMD and current Intel hardware)
    - pipeline effects make it unsuitable as a reliable timer for very small numbers of instructions
  - originally implemented as a Cycle Count, incremented each processor cycle: uint64
  - see my blog for details
- OS Timers
- .NET base classes

Tools for instrumenting your application

- Hardware clocks
  - Intel rdtsc instruction
- Windows OS Timer services
  - standardized 100 nanosecond clock counters
  - GetTickCount()
    - the number of milliseconds that have elapsed since the system was started
    - (ticks actually occur every 15.6 ms.)
  - multimedia timer
  - QueryPerformanceCounter() & QueryPerformanceFrequency()
  - QueryThreadCycleTime(*hThread,CycleTime*)
    - instrumented thread Dispatcher: rdtsc issued at every context switch

- .NET base classes

Tools for instrumenting your application

- Hardware clocks
  - Intel rdtsc instruction
- OS Timers
  - QueryPerformanceCounter()

- .NET base classes
  - Stopwatch()
    - Use case: embed object in your application; remove prior to shipping
    - `Stopwatch.StartNew`, `Start`, and `Stop` Methods
    - `Stopwatch.Elapsed` Property (TimeSpan object)
    - thin wrapper around QueryPerformanceCounter() and QueryPerformanceFrequency()

Tools for instrumenting your application

- .NET base classes
  - **Stopwatch()**
    - Use case: from Vance Morrison's blog

```
CodeTimer timer = new CodeTimer(1000);
string myString = "aString";
string outString;

timer.Measure("Measurement Name", delegate {
        outString = myString + myString; // measuring concatenation.
});
```

Tools for instrumenting your application

- .NET base classes
  - Problem: no provision in **Stopwatch()** class to log measurement data external to the program
  - Solution: **MeasurementBlock()** wrapper class around Stopwatch() to fire an ETW event containing the timing data

```
MeasurementBlock mb
      = new MeasurementBlock();

mb.Begin();

…


mb.End()
```

Tools for instrumenting your application

- MeasurementBlock() wrapper fires an ETW event containing the timing data
  - Less test noise!

    - Writing an event to ETW when there is an active Listener functions as an asynchronous RPC

```
MeasurementBlock mb
      = new MeasurementBlock();

mb.Begin();
      this.stopwatch.Start
```

… [execute test scenario]

```
mb.End()
      this.stopwatch.Stop
      ETW.write()
```

Timing Test "Noise"

- Test automation infrastructure executes each timing test multiple times
  - Tests that provide persistent, repeatable results are the most valuable ones
    - across time, across multiple machines, etc.

- Noisy tests ⇨ test scenarios with excessive **variability** in the CPU and/or execution time of performance timing tests
  - Difficult to interpret the results
  - Difficult to use the test results to drive optimization efforts

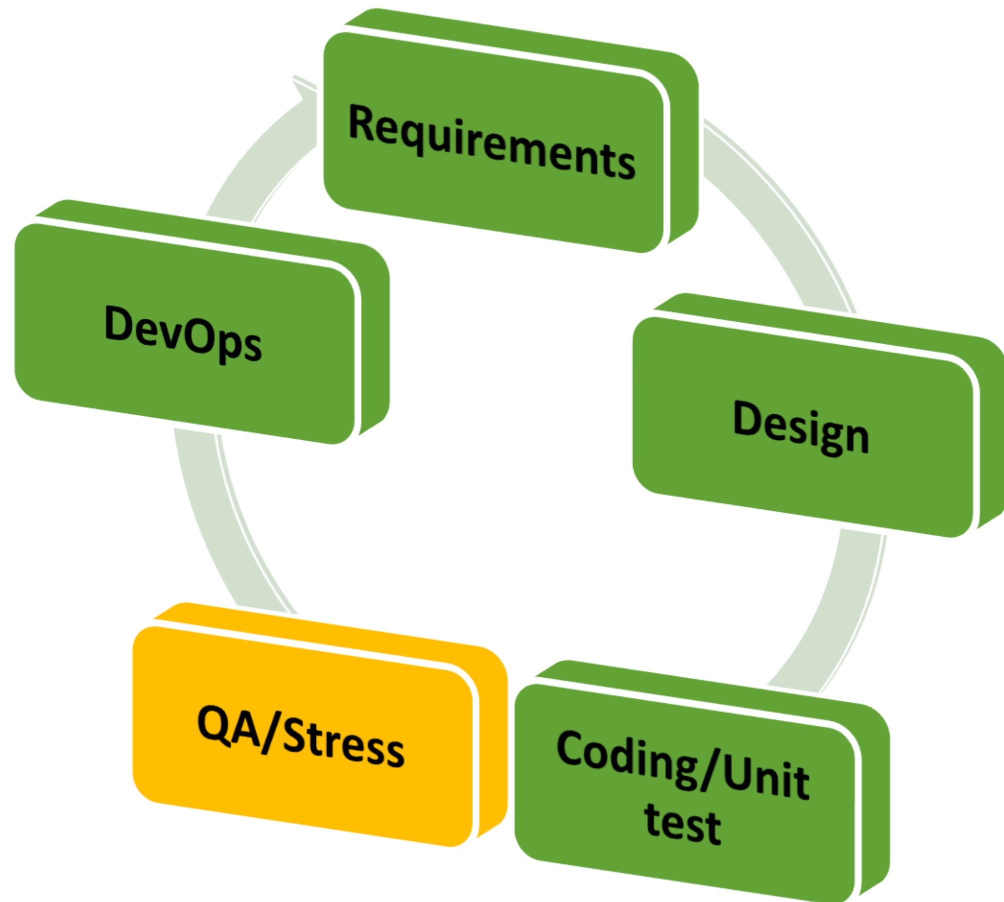
Timing Test “Noise”

- Noisy tests ⇨ excessive variability in the CPU and/or execution time of performance timing tests

- Strategies for minimizing noise during timing tests
  - Isolate test machines from the rest of the network
  - Clear all caches prior to executing the test scenario
  - Often a good practice to throw away results from the first test iteration
  - etc.

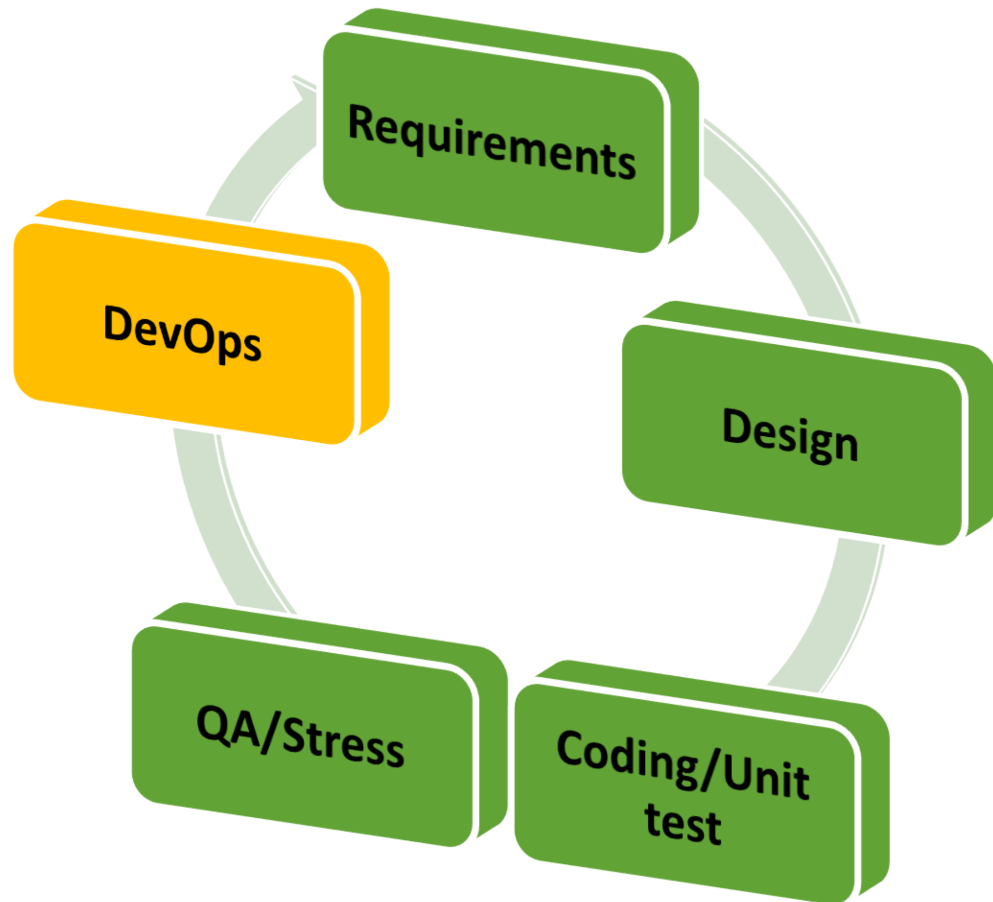
- These strategies for dealing with noise may also make the timing test results less realistic!
  - investigating the sources of “noise” for a particular test often proves valuable!

Performance and the Development Life Cycle



- Continuous improvement model
  - Full scale load/stress testing
  - Evaluate the cost of embedded instrumentation

Performance and the Development Life Cycle



- Continuous improvement model
  - Service level reporting
  - Management by Exception
    - Statistical Quality Control techniques
  - Embedded instumentation
  - Diagnostic tools to drill into problems on demand

Questions

References

- Woodside, et.al., "The Future of Software Performance Engineering," *Proceedings Future of Software Engineering*, IEEE, 2007.