CodeMash 2019

By Andrew Hinkle



This was my first year at [CodeMash](http://www.codemash.org/) and thankfully the weather behaved unlike the inches of snow I've heard from years past. It was an awesome event and I left with loads of info that will benefit me and my team immediately.

Contents

[Conference Overview 2](#_Toc293218)

[Kidzmash 3](#_Toc293219)

[Slideshows and Code Repositories 3](#_Toc293220)

[You will learn 3](#_Toc293221)

[Thursday 01/10/2019 4](#_Toc293222)

[Practical Steps to Modernizing Legacy Applications by Gregory Beamer 4](#_Toc293223)

[Leadership Guide for the Reluctant Leader by David Neal 5](#_Toc293224)

[Hold Up, Wait a Minute, Let Me Put Some Async In it By Matthew Jones 6](#_Toc293225)

[Learning the Three Types of Microservices by Mike Amundsen 6](#_Toc293226)

[Does Agile Need Architects by Kevin Fox (ICC – Sponsor Session) 8](#_Toc293227)

[Becoming an Effective Mentor by Olivia Liddell 9](#_Toc293228)

[Metaprogramming in .NET by Jason Block 9](#_Toc293229)

[Friday 01/11/2019 10](#_Toc293230)

[DISCovering your team: A Guide to Effective Communication by Patrick Badley and Jared Faris 10](#_Toc293231)

[Versioning 1.0.1 by Jon Skeet 11](#_Toc293232)

[Data management in a Microservices world by Gerald Venzl 12](#_Toc293233)

[Patterns and Architectures Beyond Microservices by Stephan Shary 13](#_Toc293234)

[How to Improve Your Task Estimates by 0% to 1,000% by Dave Sweeton (Stout Systems – Sponsor Session) 13](#_Toc293235)

[The Two Question Code Quiz: How to Interview Programmers Effectively by Scott Connerly 15](#_Toc293236)

[If Your Tests Could Talk by James Balmert 16](#_Toc293237)

[Conclusion 17](#_Toc293238)

# Conference Overview

The first 2 days of Pre-Compiler sessions have pre-requisites that require you to bring your own device (laptop/tablet/etc.), download/install apps and source code where you'll get hands on workshops and training. JD always attends these sessions ([2019](https://www.danylkoweb.com/Blog/review-codemash-v2019-LY), [2018](https://www.danylkoweb.com/Blog/codemash-v2018-LR), [2017](https://www.danylkoweb.com/blog/codemash-2017-new-friends-old-friends-great-times-IS), [2016](https://www.danylkoweb.com/Blog/codemash-2016-my-exhaustive-week-DP), and [2015](https://www.danylkoweb.com/Blog/how-i-used-codemash-2015-to-determine-my-technology-plans-for-2015-97)), so I'll let him cover them.

The last 2 days consist of standard 1 hour sessions with a 15 minute break in-between. On Thursday it started at 8:00AM EST and ended at 5:45PM EST. Friday started at 8:30AM EST and ended at 5:00PM EST with closing remarks, thanks to staff and presenters, and sponsor giveaways finished before 6:00PM EST.

There was no Keynote sessions which is fine as it gave everyone time for an additional session of their choice. CodeMash 2019 had 8 sessions on Thursday and 7 sessions on Friday. That's definitely more than any conferences I've attended to date. There were roughly a dozen sessions available at each time slot, so there were plenty of options. The conference doesn't target a specific technology. This led me to choose soft skills in some time slots which I've had a hard time justifying to myself in the past. One attendee, a developer, told me he almost exclusively chooses the soft skill sessions to work on personal growth, because the sessions here are that good. He was right, they were good.

The [Kalahari Resorts & Conventions in Sandusky, OH](https://www.kalahariresorts.com/ohio/things-to-do/weddings-and-meetings/meetings-and-conventions/) was an awesome venue for CodeMash 2019. The meals were wonderful and worth the time to eat them. There was plenty of room to walk between sessions while still supporting sponsor booths in the hallways for the roughly 2500 attendees present. The audio and video in each conference room was excellent. Oddly enough, the only complaint I heard from the main suite were the bright stage lights irritating the guests at my table. We may have just been in the anti-sweet spot for the lights. The main suite is huge with multiple screens making it pretty easy to see what was going on even from the back of the room.

Overall, the only thing I heard negative about the conference is the choice of lunch or session. Whether you left a session early, got to one late, skipped a session or lunch, something doesn't feel right here in the pit of my stomach. Yeah, I went there. I missed one session on Thursday for this very reason.

I saw many people in the main suite eating, skipping the session. I wonder how much better the session attendance would be if the conference set aside a full half hour for lunch instead of the 15 minutes to run between sessions. They could have a 15 minute sponsor presentation while attendees ate. This could also give us more time to speak with the sponsors in the hallways.

The 2-day conference cost of $350 is more than other conferences I've attended, but comes with more amenities such as access to the Kalahari waterpark, breakfast, lunch, (Thursday night dinner), credit towards drinks and snacks, game nights, and more. The Kalahari is a wonderful hotel, but if it's a bit more than what you want, there are multiple good hotels nearby, such as the [Great Wolf Lodge Sandusky, OH](https://www.greatwolf.com/sandusky) which is where I stayed with my family. CodeMash even provided shuttle service between several of the hotels adding a nice convenience.

Overall, CodeMash 2019 was an awe inspiring conference with a great location and sessions. It's considered a National conference where people come from all over the nation and other countries. It's a reunion of friends and colleagues. If you've never been before I recommend trying it out.

## Kidzmash

My kids (13 and 14) had a great time at [Kidzmash](http://www.codemash.org/kidzmash/). They enjoyed playing laser tag, twice. They participated in a team building exercise with the [Artemis Spaceship Bridge Simulator](https://artemisspaceshipbridge.com/info). They created a board game from scratch. The CAD session taught them the basics of the application where they designed an object that would later be digitally printed in 3D. My kids enjoyed the experience and would like to go back and if possible attend a session for the next level of CAD training.

They were a couple of the older kids that attended, so I asked if they could add another session, what it would be. My son recommended adding an intro to web development such as HTML, CSS, and JavaScript. Of course with CodeMash being so diversified, another session could be an intro to C#, Java, PHP, Ruby, etc. My daughter is a fan of computer graphics and would love a training session on that. Just a few ideas to possibly further grow this great opportunity.

## Slideshows and Code Repositories

I dread looking for slideshows and code repositories after a conference. Some are included in the presentation or they are promised to be linked in twitter or slack. Unfortunately they are never in one place where session goers may find them quickly. I get the extra maintenance required to post this information on the conference's web site or a simple github repository that just contains a spreadsheet as an example with the {year}.{conference}.{xlsx}. Thankfully, while searching on [Twitter #CodeMash](https://twitter.com/hashtag/CodeMash?src=hash) I found that [@ikeif](https://twitter.com/ikeif) compiled a list of [CodeMash 2019 Presentations](https://docs.google.com/spreadsheets/d/1omYTaGvLAbkJ-AGphM-WO90iE5O5s5-1dG2srKbNFYo/edit#gid=0) with links in a nice spreadsheet for us. Thanks!

The last session I attended has a slideshow created in Keynote (.key) which is an Apple program and therefore is only available for Apple Mac and iDevices only, there is no Windows version. I've never had troubles opening slideshows in the past, since they were saved in generic Microsoft formats or in slideshow sites. Something I hadn't considered in the past, but presenters should save their slideshows in a format that users can open without much effort or they won't bother trying. The last session stored the presentation as a .key file and I gave up trying to open it after five minutes of looking for a reader.

## You will learn

If you've read my last [Dog Food Con 2018](https://www.danylkoweb.com/Blog/review-dog-food-con-2018-NZ), then you'll know I recommended adding a "You will learn" section. In an effort to refine the concept further I've added that section to each session I attended. This time I added more information to the "You will learn" section and for some sessions included a requirements section. For some sessions I felt the need to identify when certain parts were beginner or advanced. Some sessions started off with introductions to beginner concepts like "what are microservices" and ended with such advanced ideas like cloud design patterns that you were left wondering if you were still in the same session.

# Thursday 01/10/2019

## Practical Steps to Modernizing Legacy Applications by Gregory Beamer

[Slideshow](https://github.com/gbworld/LegacyModernization)

**You will learn: Practical steps to modernizing legacy applications**

Legacy is code that is no longer properly serving business in an acceptable manner. Legacy is recognized as old technology, old techniques/methods. Non-Legacy is bad design/architectural decisions, impossible to maintain. Modernize Legacy, Refactor Non-Legacy. Determining how to approach modernizing a legacy app is all about TIME.

Tolerate: Function required, not worth investment. It's rarely used functionality. It's a stable product.

Eliminate: Not worth keeping or investment. It's legacy code. The function is duplicated. It's the most expensive app fulfilling the function

Invest: Worth keeping and investing in. It's modern code. It's not following best practices. There are newer methodologies/libraries. Think: Refactor.

Modernize: Worth investing in, but not keeping current code. Migrate to a new application for the same functionality. Think: Rebuild.

Know where you are going by identifying the application's current state, future state, and a map on how to get there. Use the TIME rationalization by function. Work with business to determine what existing functionality is required, what should be added, versus what is nice to have. Discard the nice to haves to meet a minimum viable product. The discarded items may come back in a later phase.

Keep in mind that you can apply TIME at the function level. You may find yourself spending time in refactoring business logic into modern libraries and then rebuild old WebForms to MVC pages.

Default to Invest (Refactor). In general don't rebuild as it's more expensive. Impossible to Invest: Migrate (Rebuild). More expensive: Migrate (complexity of code compared to functionality). More time consuming to invest: maybe Migrate (utilize products to speed time to market).

Building through cycles:

1. Plan
2. Find seams in the code
3. Organize the code base
4. Add tests
5. Add code (Assumes [TDD](https://en.wikipedia.org/wiki/Test-driven_development)/[BDD](https://en.wikipedia.org/wiki/Behavior-driven_development) Approach) (4 and 5 may flip in some instances)
6. Refactor
7. Release

Find seams in code base. Follow SOLID principles. Use interfaces to separate concerns so you may reverse dependencies. Write contracts (interfaces in most language). Your UI, business, and data should be separate seams. Understand Domain Driven Design ([DDD](https://en.wikipedia.org/wiki/Domain-driven_design)).

Organize the code base based on the Code AS Application (CASA) model. See CASA diagram and explanation on slide 40. Architectures that fit the CASA model: [Hexagonal](https://github.com/jschairb/sandbox/wiki/HexagonalArchitecture), [Onion](https://www.infoq.com/news/2014/10/ddd-onion-architecture), [Clean Architecture](http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html).

Test based on acceptance criteria, ensure you cover the fringes, and consider white and black box testing, test negative cases. Testing IS YOUR JOB, even if you have a QA department! Focus on behavior. Be sensible about code coverage.

Releases can be deployed staggered by function or a big bang, default to staggered. A Big Bang release should only be performed when there are a small number of functions, functions are intertwined, and heavy database refactoring requires a single move, heavy retraining.

Greg's presentation was an excellent start to the conference with tons of knowledge. Check out his slides for additional info and examples. The session was well paced except the family vacation analogy which I liked but could have been cut back a little to leave more time for questions at the end. This session described all of the challenges as developers we're all experiencing with practical guidance I agree with fully. Great job!

## Leadership Guide for the Reluctant Leader by David Neal

[Slideshow](https://speakerdeck.com/reverentgeek/leadership-guide-for-the-reluctant-leader-ndc-sydney-2018)

**You will learn: Leadership guide for the reluctant leader**

David is an excellent artist and we were all blessed with the plethora of hilarious renderings in his slideshow. Seriously, load up the slideshow and laugh hard as you page through it. The session was well paced and enjoyable to attend.

Leadership does not mean management. Negotiation involves research, a BATNA (Best Alternative to a negotiated agreement), and establishing an anchor. Be smart: active listening, sensitivity, empathy.

[The Ideal Team Player by Patrick Lencioni](https://www.amazon.com/Ideal-Team-Player-Recognize-Cultivate-ebook/dp/B01B6AEJJ0) is a great resource. Be humble: no ego, not status, team shares > self-credit. Be hungry: self-motivated, diligent, passion for more: learning, responsibility. Be aware of DISC behaviors (see session review below for more details). [Strengths Finder by Tom Rath](https://www.amazon.com/StrengthsFinder-2-0-Tom-Rath/dp/159562015X) is another great resource.

Avoid multi-tasking, just perform one task at a time and stay focused. Look beyond flaws. Allow the freedom to fail. Keep trying. Ask yourself the following questions. Have you done everything you can do? If you left now, would you have any regrets? Life is too short to let things stay the way they are. People want to know they have significance and value. Give positive reinforcement. Inspire others.

"People will forget what you said, people will forget what you did, but people will never forget how you made them FEEL." - Maya Angelou

## Hold Up, Wait a Minute, Let Me Put Some Async In it By Matthew Jones

[Slideshow](https://exceptionnotfound.net/downloads/slides/CodeMash2019/LetMePutSomeAsyncInIt.pptx), [Code](https://github.com/exceptionnotfound/AsyncAwaitRefactoringDemo)

**Requirements: ASP.NET C# (Beginner)  
You will learn: Async fundamentals, refactoring guidelines, troubleshooting, and cancellation tokens**

The goal of the session was to take a WebClient and update it to work asynchronously with live code. With minor assistance from the watchful eyes of the attendees we all were delighted to watch it complete successfully.

[Async](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/) Fundamentals

* Async does not equal multi-threading
* I/O-Bound, not CPU-bound
* Spreads like a virus
* Does not make code faster, it improves throughput

Refactoring guidelines

* Refactor "bottom up"
* Less dependencies
* Easier testing
* Promotes all async all the way down

Anti-Pattern: Async-over-sync

* Task.Run(() => …).Result
* Now uses 2 threads instead of 1
* Less efficient and still sync

Between the slideshow and the code repository you'll learn that in the end you'll need to add async/await throughout the entire app hence the reference to async being a virus. You'll also learn about [CancellationTokens](https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads) and how you should use the built in ones. I'm usually wary of live coding sessions as the quality is variable, but there was nothing to fear here as it worked well to keep the attendees attention. Great session!

## Learning the Three Types of Microservices by Mike Amundsen

[Slideshow](http://amundsen.com/talks/2019-01-codemash/2019-01-codemash-three-types-msc.pdf), [Webpage](http://amundsen.com/talks/2019-01-codemash/index.html)

**You will learn: Introduction to microservices (Beginner); Stateless, Persistence, and Aggregator microservices; Microservice stability cloud patterns (Advanced)**

Stateless Microservices

* No dependency on other microservices
* No local data storage (disk I/O)
* The most common, but least useful
* No shared state
* Easy to replace
* Easy to scale up

Persistence Microservices

* Simple (local) storage (reads and/or writes)
* Disk I/O dependent
* Possibly VM or one-U dependent
* Commonly needed difficult to implement
* System of record/source of truth
* Relatively easy to scale for reads (CQRS)
* No cross-service two-phase commits (Saga)

Aggregator Microservices

* Depends on other ("distant") microservices
* Network dependent
* Usually Disk I/O dependence, too
* The most often needed, hardest
* Sequence vs. Parallel calls
* Timing is everything
* Easy to scale (should be…)

Building upon the [Nygard Stability Patterns](https://speakerdeck.com/mtnygard/stability-patterns-and-antipatterns?slide=28) described in his book [Release It!](https://www.oreilly.com/library/view/release-it/9781680500264/)

* Use Timeouts – stop waiting for an answer once you think it will not come
* [Circuit Breaker](https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker) – degrade functionality when the system is under stress; closes/open/half-open
* [Bulkhead](https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead) – enforces a principle of damage containment
* Steady State – the system should be able to handle itself without human intervention; avoid fiddling; purge data with app logic; limit caching; roll the logs
* Fail Fast – if the system knows something will fail, then fail fast
* Handshaking – server protects itself by [throttling](https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling) its own workload; call [health check](https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring) to know response times and determine if you should continue
* [Cache](https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside) – caching can reduce load on the server

For each microservice type here are the recommended stability patterns:

* Stateless: Fail fast
* Persistence: Fail fast, use timeout, circuit breaker, steady state
* Aggregator: Use all stability patterns

A reminder about [HTTP Methods](https://restfulapi.net/http-methods/): PUT is Idempotent, POST is not, consider writing your code as PUT when possible.

The first quarter of the session was an introduction to microservices, the second described the three types of microservices, and the last half was a high level overview of the advanced topic of stability patterns. There were plenty of questions at the end left unanswered due to time constraints. I recommend making it an advanced topic and cut the intro to microservices out or at least down to a few minutes to allow for more time to answer questions at the end. Mike knows his stuff and is great speaker.

## Does Agile Need Architects by Kevin Fox (ICC – Sponsor Session)

No slideshow links at the time of writing.

**You will learn: How to transform the standard waterfall architect into an agile architect**

If I got nothing else out of this session, it was worth it just to get a new nickname for our architect the "Whiteboard Warrior"! He wears it as a badge of honor.

Architects provide the benefits to coordinate across teams, identify common patterns, continuous improvements, and rapid improvement. Architects are often associated with the [waterfall development model](https://en.wikipedia.org/wiki/Waterfall_model) where everything is designed upfront before any work is done. Some changes in thought will help transition architects to [Agile software development](https://en.wikipedia.org/wiki/Agile_software_development). Taking a page from the [Agile Manifesto](http://agilemanifesto.org/) Kevin listed out his proposal for an Agile Architect Manifesto. In the following while both the left and right items are valued, the left items are valued more.

1. Capability over Architecture
   1. People centric, value-focused, actionable, reproductive, adaptable, improvable.
2. Organic over Imposed
   1. Change the math, change the economics, change the perception, requires cultivation not only of solutions but relationships as well
3. Improvement over Adherence
   1. Emergent design feeds into standards, improvement is fostered through engagement with the team
4. Guidance over Governance
   1. Enablement is attractive, enforcement is expensive, experience is invaluable

In conclusion, Agile needs architects that work with the team. The [Scaled Agile Framework](https://www.scaledagileframework.com/) was given as a good resource too and I thought the [Agile Architecture](https://www.scaledagileframework.com/agile-architecture/) was definitely of interest.

Overall, the session was fine and a decent start at an Agile Architect Manifesto and Kevin or ICC (the sponsor) should get the discussion started to fine tune it.

## Becoming an Effective Mentor by Olivia Liddell

[Slideshow](https://www.slideshare.net/OliviaLiddell/olivia-liddell-codemash-2019-becoming-an-effective-mentor)

**You will learn: Becoming an effective mentor**

If you're considering being a mentor and you don't know what to mentor on, then follow Olivia's "Reflected Best Self Exercise" found in her slideshow. These steps will help you identify your specific skills and strengths that you could mentor on. Ask others to answer the following question about you. "I am at my best when…" Identify a pattern in their response and you might just learn more about yourself and what you could mentor on.

Informal mentoring involves the mentors and mentees to choose to work together, goals are specified, and outcomes are usually not tracked or measured. Offer support when you can, but don't force it. Be open and approachable.

Formal mentoring involves the mentors and mentees being paired, goals are established, and outcomes are tracked and measured. Start with the end goals in mind and work backwards. Identify what measurable evidence proves the mentoring is working. Meet with the mentee to continue their education. Set regular check-in intervals.

Let the mentee drive the relationship. Listen before you speak. Your body language should be encouraging and open. When they speak, pay attention. Restate the thoughts differently to reflect that you are paying attention and not just repeating their words. Ask for clarification when you don't quite understand. Consider their perspective. Be specific and direct. Offer suggestions and explain why. Olivia included in her slideshow and presentation phrases to consider using to address these concerns.

She concluded the presentation with helpful advice on getting started by acknowledging the fear of being a mentor, lack of strong communication skills, not enough time, and feeling you are a beginner yourself. She's an excellent speaker and held our attention throughout the session. If you're interested in becoming a mentor Olivia provided additional resources near the end of the slideshow.

## Metaprogramming in .NET by Jason Block

[Slideshow](https://github.com/JasonBock/Presentations/blob/master/Metaprogramming%20in%20.NET.pptx)

**Requirements: C# .NET (Beginner)  
You will learn: Snippets, expressions, dynamic, reflection (Beginner); DynamicMethod, intermediate language (MSIL) (Advanced)**

The C# .NET [dynamic](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic) option is a form of [Metaprogramming](http://en.wikipedia.org/wiki/Metaprogramming) that takes advantage of [reflection](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection) to allow you to bypass the strongly typed requirement prevalent in C#. [Snippets](https://docs.microsoft.com/en-us/visualstudio/ide/code-snippets?view=vs-2017) are another form of metaprogramming that allows you to map a simple string such as "prop" or "ctor" to pieces of code to save time.

Unit tests use attributes to define the test classes and methods and use reflection to identify them. Use [Expressions](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expressions) like lambda's to simplify readability while behind the scenes additional code replaces the lambda behind the scenes. One interesting tidbit he added was that "var" can't tell the difference between an Expression and a Func.

Use [System.Reflection.Emit](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit?view=netframework-4.7.2) and [DynamicMethod](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.dynamicmethod?view=netframework-4.7.2) to code the Microsoft Intermediate Language ([MSIL](https://en.wikipedia.org/wiki/Common_Intermediate_Language)) directly. Working with the MSIL is very dangerous. If you make a syntax error in MSIL you will not get an error at compile, but you'll get an exception you've never seen before. Dynamic is better than MSIL. Expressions are better than dynamic.

Roslyn is a compiler written in C# for C# and it exposes an API to modify it further. Jason took advantage of this feature to write his own mock library called [Rocks](https://github.com/JasonBock/Rocks). It works like [Moq](https://github.com/Moq/moq4/wiki/Quickstart) except instead of using reflection to manipulate the MSIL, this uses the Roslyn API. Running Rocks opens a console to "warm up" the Roslyn API. Rerunning the test adds the test results to the console.

This was an interesting session that gave insight into another way of approaching programming to prevent muddying your code by injecting code via Metaprogramming in a similar contrast to [Aspect-Oriented Programming](https://www.postsharp.net/aop.net). While there are definitely some quick takeaways with using snippets and expressions, the rest goes down a scary alley way too close to MSIL for my liking. Still, it's pretty cool what we can do with C# and the .NET framework. Jason did a great job and you can tell he's passionate about coding.

# Friday 01/11/2019

## DISCovering your team: A Guide to Effective Communication by Patrick Badley and Jared Faris

[Slideshow](https://www.slideshare.net/PatrickBadley/discovering-your-team-codemash)

**You will learn: DISC behavior type and how they interact with each other**

[DISC](https://discinsights.com/disc-theory):

1. D: Dominant
   1. They are: Self-Starter, Task-oriented, Direct, Fast-paced, Decisive, Strong-willed
   2. They are not: Sensitive, Patient
   3. Motivators: Results, Challenges, Power/Authority
2. I: Influence
   1. They are: Fun, Story-teller, Captivating, Charismatic, Chatty, Inspiring
   2. They are not: Detail-Oriented, Focused
   3. Motivators: Relationships, Fun and Socializing, Connecting People
3. S: Steadiness
   1. They are: Friendly, Determined, Even-tempered, Sincere, Sensitive, Reliable
   2. They are not: Assertive, Confrontational
   3. Motivators: Stability, Routines and Plans, Avoiding Conflict
4. C: Conscientious
   1. They are: Patient, Careful, Focused, Organized, Logical, Solitary
   2. They are not: Outspoken, Impulsive
   3. Motivators: Quality, Logic and Reason, Recognition

Instead of the Golden Rule, DISC instead promotes the Platinum Rule.

1. Golden Rule: Do unto others as you would have them do unto you.
2. Platinum Rule: Treat others as they want to be treated.

In the end, realize that people are different, recognize your differences, and adjust your behavior. The slideshow and DISC link above provide excellent examples on how to respond from the perspective of each type. The "C" in me really wanted a bullet point list explaining basic rules to follow when you are a "D" speaking with an "S" and so on. Jared fell into the category of "D" and "I", while Patrick was an "S" and "C". Having them perform the session together is what made it work so well.

## Versioning 1.0.1 by Jon Skeet

No slideshow links at the time of writing. Here's a quick shout out to [Gene Gotimer](https://twitter.com/CoverosGene) for tweeting out some great [versioning notes](https://twitter.com/CoverosGene/status/1083737167518711811) on this session.

**You will learn: Semantic versioning and other standards; Versioning based on dependencies**

Be kind to others. If you took nothing else away from his presentation, be kind to others. [Jon Skeet](https://twitter.com/jonskeet) is a legend on [Stack Overflow](https://stackoverflow.com/users/22656/jon-skeet), so I wanted to see his presentation in person at least once.

The first ten minutes consisted of how versioning works in different media and some of it was referenced later in the session to bring it all together.

Minor versions can tell the order of when a package was created, but when you have different major versions, it's not obvious as older versions may still be maintained.

Versioning should address compatibility. Non-breaking changes go into minor versions. Adding class names (usings could be breaking) that a consumer app also has can be considered a breaking change. However, there's a difference between theoretical and realistic breaking. If you follow the [names of namespaces](https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces) standards, then namespaces should theoretically not be an issue.

You can't make a patch to fix a breaking change, because now you're breaking the minor version.

Diamond dependencies via app dependencies

App 1.0 depends on LibX 1.1 which depends on LibZ 1.2  
 App 1.0 depends on LibZ 3.2

While many times you can take the latest version, sometimes it may not work especially over major versions. Between versions, functionality may have been moved or removed. You'll need to update your app to work with the same versions to avoid issues.

If you take a new patch of a dependency, you'll need to bump your patch. If you take a new minor, bump your app's minor. If you take a new major, bump your app's major.

I came to the session in regards to Network API (Web Services) versioning. He asked the following questions and proposed a standard that Google is experimenting with.

1. How is the client calling the API?
2. What does the client care about?
3. What are the rules for compatible changes?
4. How can we perform user testing for a change?

I was hoping for more concrete examples such as answering if adding a field to a service contract is a breaking change that would require a patch version number change?

In the end, Jon Skeet recommended that in most cases [Semantic](https://docs.microsoft.com/en-us/dotnet/csharp/versioning) [Versioning](https://semver.org/), while not perfect, is still the best versioning technique available at this time.

## Data management in a Microservices world by Gerald Venzl

[Slideshow](https://www.slideshare.net/gvenzl/data-management-in-a-microservices-world-128345289)

**You will learn: Polyglot persistence; cloud design patterns built in by database providers**

Rule 1: You are not Netflix, Google, or Facebook!

Per Martin Fowler, the future is not NoSQL Databases, the future is [Polyglot Persistence](https://martinfowler.com/articles/nosql-intro-original.pdf). This means the data can be stored in Oracle for one microservice, MySql for another, and SQL Server for a third. Each microservice should have its own data storage.

Rule 2: Just because you can doesn't mean you should.

Data may be persisted in the following formats. Some databases support a single-model (only one of the formats), while others are multi-model and may support some up to all of the formats. While you may choose different database providers, by selecting a multi-model from a single provider simplifies maintenance, standard administrator, and skill base.

1. Relational data
2. Key/Value data
3. Document (Json/XML)
4. Graphs
5. Free Text and Docs (PDF, DOC, …)
6. Spatial data

There's plenty of hype for Event Sourcing which has apps send commands to a central location that is referenced by every app. This pattern makes it much harder to understand the structure from other domains. The more data you have, the more complex this system gets.

Rule 3: Just because it sounds cool doesn't mean it fits your use case!

Gerald proceeded to explain that many patterns are already handled by the popular database providers. [Event Sourcing](https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing) is managed via [Change Tracking](https://docs.microsoft.com/en-us/sql/relational-databases/track-changes/enable-and-disable-change-tracking-sql-server?view=sql-server-2017). Containerization is managed via [Docker containers for SQL Server](https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-container-ha-overview?view=sql-server-2017). [Sharding](https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding) allows for linear scalability, high availability, and geographic distribution.

The session was well done overall. At the end the patterns described had concrete examples on how to resolve those using Oracle features. I'd expect a presenter to choose a provider to demonstrate with and he was up front about it from the beginning. While I didn't get to see the examples in the provider of my choice, in this case SQL Server, the terminologies translated well and I provided a few links above as reference.

## Patterns and Architectures Beyond Microservices by Stephan Shary

No slideshow links at the time of writing.

**Required: Microservices (Beginner)  
You will learn: Microservice architecture recommendations with maintenance in mind; Cloud design patterns (Advanced)**

The first highlight as noted by JD in his [CodeMash 2019 review](https://www.danylkoweb.com/Blog/review-codemash-v2019-LY) was when you should not use micro-services. If you group is smaller than 15 people or your data size is less than 100GB of data, and most importantly lacks discipline, then you should not use microservices, important advice indeed.

Each service must provide good documentation that is always up-to-date, focuses on examples, doesn't have too much to read, provides a way to try out examples, and is easy to find. There are multiple solutions that create documentation based on your code such as [Swagger](https://swagger.io/docs/specification/about/)/[OpenAPI](https://github.com/OAI/OpenAPI-Specification). Tests can be provided as examples on how to implement the service.

Don't build a library to be used by multiple microservices. You need to keep them loosely coupled. Avoid mixing multiple architecture types by having the microservices follow a standard pattern. Keep maintenance in mind front and foremost.

Gerald explained several [cloud design patterns](https://docs.microsoft.com/en-us/azure/architecture/patterns/) such as the [Sidecar](https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar), [Bulkhead](https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead), [CQRS](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs), and [Event Sourcing](https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing) that were reviewed in the rest of the session.

Overall, the session went well and I learned more about some cloud design patterns I haven't worked with much yet. The intro to microservices went by quick enough given this was an advanced session.

## How to Improve Your Task Estimates by 0% to 1,000% by Dave Sweeton (Stout Systems – Sponsor Session)

[Slideshow](https://drive.google.com/file/d/14qlSVn0jQqMb7csvs_pYV-hEPzIONWnl/view)

**You will learn: Precision vs accuracy; estimation by hours; ballparks and guesses**

This was another great session. We've all been plagued with inaccurate estimates to the point they're considered a joke until we're held to them as deadlines at which point we're crying. I'm not speaking from experience {sniff} really I'm not {sob}.

Precision vs accuracy: accuracy is hitting the target and precision is the size of the target. The goal is to be 100% accurate and only precise enough. Don't give estimates that you aren't comfortable with. Do not say you don't know how long a task will take, because that's part of your job. You can say you don't know, let me get back to you.

Use appropriate units. The units tell how certain you are. Weeks, months, and years are rough estimates. Estimates by hours are more accurate.

Ballparks and guesses are useful planning tools. Never call them estimates. However, only give them to those you trust to use them wisely. Otherwise, they turn into accusations and deadlines. Eventually the ballparks and guesses will need to be broken down into estimations.

For your estimations use ranges that convey uncertainty. The low number is optimistic but the work can reasonably be done in that time. The high number is based on actual risks. Don't use ranges just to range, they must represent actual risk. Example: 7 – 10 hours. The range represents the target which may increase your accuracy of meeting the estimate.

Don't double or triple your estimations to buffer for issues. Be honest and accurate. If you are not, then your manager will just halve or third your estimations and no one is better off.

Break down projects into small tasks and estimate those tasks. Tasks should be 16 hours or smaller. That number should be adjusted by the team.

Track your actual time spent on a task. The only way to improve at estimation is to:

1. Make detailed estimates
2. Keep those estimates
3. Track your actual time spent in more detail than your estimates
4. Compare when you are done
5. Use insights to improve future estimates
6. Repeat until your retire or are replaced by AI
   1. I loved this line!

Learn from your actual time vs estimates. If the estimation was wrong, what time was not accounted for? Here are some examples of time that should be tracked for a task. I added a few more that I seem to miss. Usually I'll create separate tasks for deployment and QA tests. I'd like to follow up sometime with Dave to see if he assigns time for all of these in the same task or if he breaks some of them up like the deployments as I do. Check out his slideshow for his examples.

1. Meetings
2. Requirements gathering
3. Project setup
4. Familiarizing yourself with legacy code
5. Coding work
6. Database work
7. Code Reviews
8. Unit Tests
9. Integration Tests (manual)
10. Integration Tests (automated)
11. QA tests
12. Deployment

Your goal is not to be 100% accurate on every task. Instead aim for 100% over a block of tasks. Overtime, they'll average out.

In regards to Scrum estimation, Dave can't help. You cannot estimate user stories. Story points may help give an idea of how long a project may take, but it's not much different than a ballpark or guess.

Developers should estimate their own tasks. Do not estimate someone else's tasks. The best you can give them is a ballpark.

This was a great session on a topic that I would like to learn and improve upon. Great job!

## The Two Question Code Quiz: How to Interview Programmers Effectively by Scott Connerly

[Slideshow](https://slides.com/scottconnerly/2questioncodequiz)

**You will learn: Screening and Technical interview guidelines**

This session started off with watching the classic "[Monty Python's flying circus – Interview and salesman](https://www.youtube.com/watch?v=1dWMIuipn_c)". While very entertaining and setting the mood it took up some time that may have been better used to answer the multitude of follow-up questions attendees had at the end. The goal of the video was to humorously demonstrate how behavioral interviewing has not been used effectively in the past. The lesson learned is that we want to use open-ended questions where the candidate can tell a story that will give us more insight into who they are and if they would be a good cultural fit.

When asking technical questions avoid asking Google-bait tech-facts that can be uncovered with a simple search. Research studies have shown that if you can find the information quickly, you won't memorize it. Brain Teasers and trick questions don't test your creativity or your ability to think outside the box. Don't ask questions that aren't related to the job. Always go for 3-month exit clause for both the company and the candidate regardless of Contract to Hire (CTH) or Full-Time Employee (FTE).

Don't give the candidate homework assignments. Unless you're watching them you don't know how they came to the solution. They could have just searched for an answer on the web or had someone else do it. Don't ask for code examples that implement a good performing sorting algorithm, since there are dozens of example on the web. Don't ask how to use a technology, ask why they use it.

There is plenty of great advice in the slideshow and I highly recommend any hiring manager or interviewer read through them. Understand that the session only went in depth on the screening and technical interviews.

In the end the two questions came down to two code samples with 5 minute timers. Each had some code in place with a setup and expected results. The candidate would then code the solution, compile, fix errors, and repeat until done or time runs up. Browsing the web is fine, but you shouldn't be Googling the answer, it should be used for syntax issues and such. That of course is part of the test.

Question 1: Sum the values in an array. The real question: do you even code? There may be fancy methods part of the language that does it better, but a simple foreach statement is perfectly acceptable. In fact if they chose the fancy version they may have Googled it.

Question 2: Uppercase the values in a 5-deep nested array. The real question: Can you handle abstraction with ease such as recursion. Let the candidate know most people don't finish this question within the time slot. Stop when time is up. Review with the candidate what was done and what were the next steps.

The session was great along with the follow-up questions. As mentioned earlier the session would have benefited from another five minutes at the end. Some great questions were as follows. Do you give a different set of questions based on the position? No, give the same two questions to the juniors, mids, and seniors. Is it demeaning to ask a senior to write code? No, if they are expected to code, then they should answer the questions.

## If Your Tests Could Talk by James Balmert

[Slideshow](https://github.com/jbalmert/if_your_tests_could_talk) (It is a ".key" file)

**You will learn: Mockist TDD; Refactoring based on SOLID principles via examples**

The Classic [Test-Driven Development (TDD)](https://technologyconversations.com/2014/09/30/test-driven-development-tdd/) follows the pattern of red/green/refactor that eventually breaks up the responsibilities into separate classes. Here's an [example walkthrough](https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/) written in Java that even C# developers like myself can follow.

Jimmy promoted the Mockist TDD variation during this session. He created a tic-tac-toe game and defined the first scenario for a simple win with 3 X's in a row. He defined the classes he thought he would need such as players, game board, and a game evaluator to detect when the game was over. He created the classes and stubbed methods that he could then mock and inject into the other classes.

After each refactoring he would review his work and record feedback by evaluating the following three types. Most of this feedback falls back to following the [SOLID](https://en.wikipedia.org/wiki/SOLID) principles.

1. Abstraction: ex. Is this class exposing internal state?
2. Responsibility: ex. Too many responsibilities?
3. Usability: ex. Is it easy to understand what this object does?

The thought exercise was an intriguing point of view. After reviewing business requirements for a project and deciding between new or updating an existing system, I usually have a good idea of the objects I want to create similar to this approach. As I create each object, I'll test just the logic in it. If there are any dependencies, I'll inject mocks of them through the constructor and verify that they were called when expected, with the expected values, and have them return an expected result necessary for the object I'm testing.

I'm not interested in testing the behaviors of the dependencies, just that they were called. The only issue I ever run into with my variation is that sometimes integrations may be missed, so at some point an actual integration test will need to be performed, but that's not a unit test. By creating a full scenario with all mocked data all the way down, Jimmy avoided this issue. There's definitely value here to be considered further.

Overall Jimmy did a fine job going through all of the feedback loops, explaining the reasons for each change, and performing the refactoring. There was some repetition, but it was reasonable considering we are talking about TDD and sometimes you start a refactoring only to realize it didn't work and go back and reevaluate. That just reinforced the need for quicker feedback.

Additional References:

1. [Mocks Aren't Stubs by Martin Fowler](https://martinfowler.com/articles/mocksArentStubs.html#ClassicalAndMockistTesting) which includes a discussion on Classical and Mockist Testing
2. [Should I practice "mockist" or "classical" TDD? - StackOverflow](https://stackoverflow.com/questions/184666/should-i-practice-mockist-or-classical-tdd)

# Conclusion

This was my first time attending Codemash. I didn't attend the first two days of pre-compiler sessions, but the last two days were filled with great sessions of even greater variety than I am used to from the other conferences I've attended. It was an awe-inspiring experience that I will treasure.