**Prepare Your Mind**

**Somewhere Along The Line**

* We became impressed with programs that contain large amounts of code.
* We strived to create large abstractions in our code base.
* We forgot that the hardware is the platform.
* We lost the understanding that every decision comes with a cost.

**These Days Are Gone**

* We can throw more hardware at the problem.
* We can throw more developers at the problem.

**Open Your Mind**

* Technology changes quickly but people's minds change slowly.
* Easy to adopt new technology but hard to adopt new ways of thinking.

**Interesting Questions - What do they mean to you?**

* Is it a good program?
* Is it an efficient program?
* Is it correct?
* Was it done on time?
* What did it cost?

**Aspire To**

* Be a champion for quality, efficiency and simplicity.
* Have a point of view.
* Value introspection and self-review.

**Mental Models**

You must constantly make sure your mental model of your projects are clear. When you can't remember where a piece of logic is or you can't remember how something works, you are losing your mental model of the code. This is a clear indication that refactoring is a must. Focus time on structuring code that provides the best mental model possible and code review for this as well.

How much code in that box do you think you can maintain a mental model of in your head? I believe asking a single developer to maintain a mental model of more than one ream of paper in that box (~10k lines of code) is asking a lot. If you do the math, then it takes a team of 100 people to work on a code base that hits a million lines of code. That is 100 people that need to be coordinated, grouped, tracked and in a constant feedback loop of communication.

**Productivity vs Performance**

Productivity and performance both matter, but in the past you couldn’t have both. You needed to choose one over the other. We naturally gravitated to productivity, with the idea or hope that the hardware would resolve our performance problems for free. This movement towards productivity has resulted in the design of programming languages that produce sluggish software that is outpacing the hardware’s ability to make them faster.

By following Go’s idioms and a few guidelines, we can write code that can be reasoned about by anyone who looks at it. We can write software that simplifies, minimizes and reduces the amount of code we need to solve the problems we are working on. We don’t have to choose productivity over performance or performance over productivity anymore. We can have both.

**Correctness vs Performance**

You want to write code that is optimized for correctness. Don't make coding decisions based on what you think might perform better. You must benchmark or profile to know if code is not fast enough. Then and only then should you optimize for performance. This can't be done until you have something working.

Improvement comes from writing code and thinking about the code you write. Then refactoring the code to make it better. This requires the help of other people to also read the code you are writing. Prototype ideas first to validate them. Try different approaches or ask others to attempt a solution. Then compare what you have learned.

Too many developers are not prototyping their ideas first before writing production code. It is through prototyping that you can validate your thoughts, ideas and designs. This is the time when you can break down walls and figure out how things work. Prototype in the concrete and consider contracts after you have a working prototype.

Refactoring must become part of the development cycle. Refactoring is the process of improving the code from the things that you learn on a daily basis. Without time to refactor, code will become impossible to manage and maintain over time. This creates the legacy issues we are seeing today.

**Code Reviews**

You can't look at a piece of code, function or algorithm and determine if it smells good or bad without a design philosophy. These four major categories are the basis for code reviews and should be prioritized in this order: Integrity, Readability, Simplicity and then Performance. You must consciously and with great reason be able to explain the category you are choosing.

**Integrity**

***We need to become very serious about reliability.***

There are two driving forces behind integrity:

* Integrity is about every allocation, read and write of memory being accurate, consistent and efficient. The type system is critical to making sure we have this micro level of integrity.
* Integrity is about every data transformation being accurate, consistent and efficient. Writing less code and error handling is critical to making sure we have this macro level of integrity.

**Write Less Code:**

There have been studies that have researched the number of bugs you can expect to have in your software. The industry average is around 15 to 50 bugs per 1000 lines of code. One simple way to reduce the number of bugs, and increase the integrity of your software, is to write less code.

Bjarne Stroustrup stated that writing more code than you need results in Ugly, Large and Slow code:

* Ugly: Leaves places for bugs to hide.
* Large: Ensures incomplete tests.
* Slow: Encourages the use of shortcuts and dirty tricks.

**Error Handling:**

When error handling is treated as an exception and not part of the main code, you can expect the majority of your critical failures to be due to error handling.

48 critical failures were found in a study looking at a couple hundred bugs in Cassandra, HBase, HDFS, MapReduce, and Redis.

* 92% : Failures from bad error handling
  + 35% : Incorrect handling
    - 25% : Simply ignoring an error
    - 8% : Catching the wrong exception
    - 2% : Incomplete TODOs
  + 57% System specific
    - 23% : Easily detectable
    - 34% : Complex bugs
* 8% : Failures from latent human errors