RI Development Principles:

We have a draft version of Engineering Principles for RI (Sept 2019). This is in 'feedback mode' but once agreed with narrative they will be officially launched.

There are also specific Development Principles that are more specific to development of code and features that should also relate to an over-arching Engineering Principle. We need to add to these and also provide some narrative.

- Engineering Principles (draft):
  - Cloud First.
  - Serverless preferred.
  - Expose all Business Events.
  - Maximise Data Accessibility.
  - Avoid building new features in Core Systems.
  - Migrate Core Functionality to Microservices.
  - Modularity and De-coupled Solutions.
  - Buy V Build if commodity and no differentiator.
  - Prefer Event Driven to Batch.
  - API as a Product.
- Development Principles (draft and very incomplete):
  - SOLID
  - Security Aware
  - Reusable UI Components
  - YAGNI
  - TDD
  - BDD
  - Clean Code
  - Release Early, Release Often
  - DRY
  - Pair programming
  - Pomodoro
  - Rubber Duck Debugging
  - Test doubles
  - Design patterns

Engineering Principles (draft):

- **Cloud First.**
- **Serverless preferred.**
- **Expose all Business Events.**
- **Maximise Data Accessibility.**
- **Avoid building new features in Core Systems.**
- **Migrate Core Functionality to Microservices.**
- **Modularity and De-coupled Solutions.**
- **Buy V Build if commodity and no differentiator.**
- **Prefer Event Driven to Batch.**
- **API as a Product.**

Development Principles (draft and very incomplete):

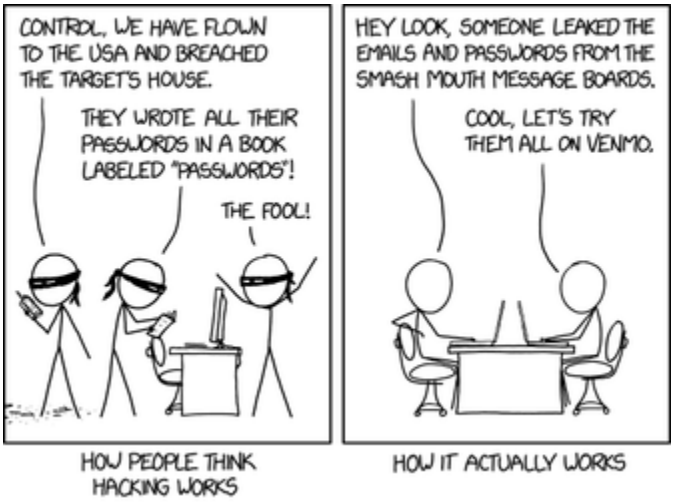
SOLID

- **Single responsibility principle**: "Do one thing and do it well." Classes, software components and microservices that have only one responsibility are much easier to explain, understand and implement than the ones that provide a solution for everything
- **Open / closed principle**: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

A class should be easily extendable without modifying the class itself; In doing so, we stop ourselves from modifying existing code and causing potential new bugs. Implement Open / Closed via inheritance and / or implementing interfaces that enable classes to polymorphically substitute for each other.

- **Liskov Substitution**: "If ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced (or substituted) with objects of type ***S***." Every subclass / derived class should be substitutable for their base / parent class.
- **Interface segregation**: "A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use." It is better to have many smaller interfaces, than fewer, fatter interfaces; Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them. No client should be forced to depend on methods it does not use
- **Dependancy Inversion**: "Entities must depend on abstractions not on concretions. High level module must not depend on the low level module, but they should depend on abstractions." This is a way to decouple software modules via dependency injection: 'inject' any dependencies of a class through the class constructor as an input parameter.

Security Aware



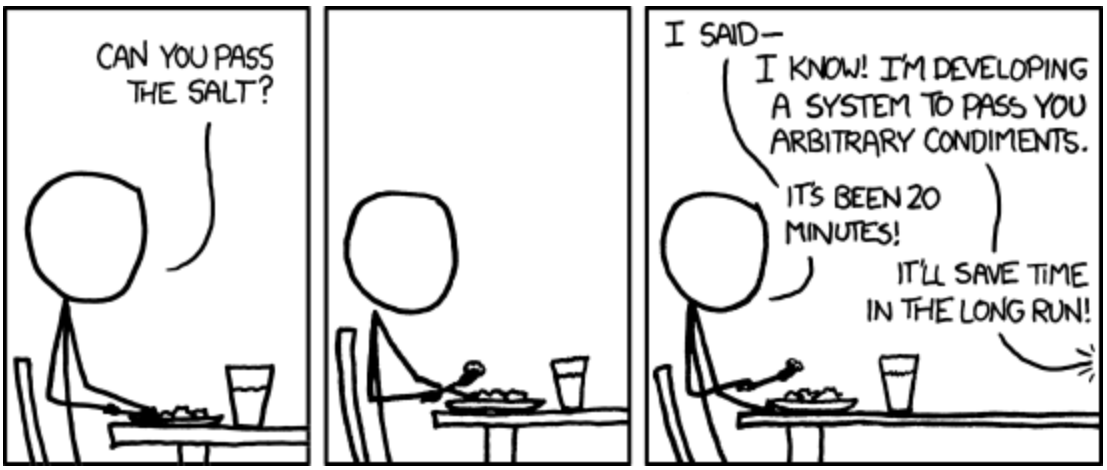
- Zero Trust security
  - A security model that requires strict identity verification for every person and device trying to access resources on a private network, regardless of whether they are sitting within or outside of the network perimeter
- 2FA
  - A security process in which the user provides two different authentication factors to verify themselves to better protect both the user's credentials and the resources the user can access.
- OWASP
  - Open Web Application Security Project is a worldwide not-for-profit charitable organisation focused on improving the security of software. The annually published OWASP Top 10 consists of the 10 most seen application vulnerabilities.
- CWE Top 25
  - The Common Weakness Enumeration Top 25 Most Dangerous Software Errors (*CWE Top 25*) is a demonstrative list of the most widespread and critical weaknesses that can lead to serious vulnerabilities in software.
- Static code analysis
  - A method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards.

Reusable UI Components

- Components let you split the UI into independent, shareable, reusable pieces. These reusable components are self-contained and have well-defined interfaces; as such, they can be reused to build more than one UI instance.

YAGNI

- “Always implement things when you *actually* need them, never when you just foresee that you *may* need them.”
- Don't waste time and resources on what you might need; Build only what you *do* need



TDD

- Red, green, refactor:
  - Create a unit test that fails.
  - Write production code that makes that test pass.
  - Tidy up the code
- 3 laws of TDD:
  - You are not allowed to write any production code unless it is to make a failing unit test pass.
  - You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
  - You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

BDD

- Behaviour Driven Design is a collaborative approach to software development that helps create business requirements that can be understood by the whole team.
- The Given-When-Then formula is a template intended to guide the writing of acceptance tests for a User Story:
  - (Given) some context
  - (When) some action is carried out
  - (Then) a particular set of observable consequences should obtain

Clean Code

- Follow standard conventions.
- Keep It Simple Stupid: Simpler is always better. Reduce complexity as much as possible.
- Boy scout rule: Leave the campground cleaner than you found it.
- Always find root cause: Always look for the root cause of a problem, don't just treat the symptoms.
- Always try to explain yourself in code / your code should be self-documenting. Heavy use of comments implies needless code complexity
- Choose descriptive and unambiguous naming conventions
- Keep functions small and obey SRP
- One assert per test
- Look for code smells:
  - Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.

- Fragility. The software breaks in many places due to a single change.
- Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
- Needless Complexity.
- Needless Repetition.
- Opacity. The code is hard to understand.

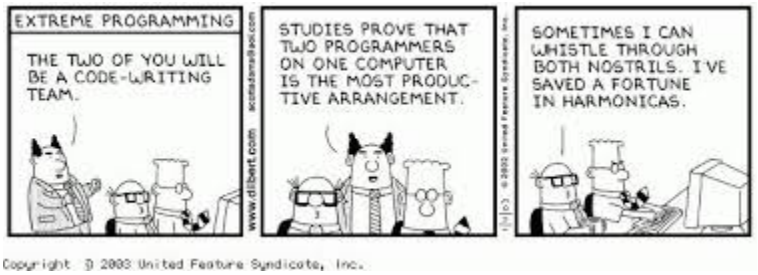
Release Early, Release Often

- Early and frequent releases creates a tight feedback loop between developers and testers or users, contrary to a feature-based release strategy.
- This allows software development to progress faster, enables the user to help define what the software will become, better conforms to the users' requirements for the software, and ultimately results in higher quality software.
- The development philosophy attempts to eliminate the risk of creating software that no one will use.

DRY

- The Don't Repeat Yourself (DRY) principle states that duplication in logic should be eliminated via abstraction; duplication in process should be eliminated via automation.
- Adding additional, unnecessary code to a codebase increases the amount of work required to extend and maintain the software in the future.
- Duplicate code adds to technical debt.
- Whether the duplication stems from 'Copy Paste Programming' or poor understanding of how to apply abstraction, it decreases the quality of the code.
- Duplication in process is also waste if it can be automated. Manual testing, manual build and integration processes, etc. should all be eliminated whenever possible through the use of automation.

Pair programming



- "Two heads are better than one": Pair programming is two programmers working together at one workstation: One programmer is the driver and writes code. The other is the observer or navigator who reviews each line of code as it is typed in. While it may seem that this would reduce the total productivity, it has the following benefits:
  - Fewer mistakes and bugs
  - Easier to keep going — Moral support
  - Shared best practices
  - Faster on-boarding
  - Harder to procrastinate

Pomodoro

- A means to timebox tasks, using a real or virtual 'pomodoro' (tomato) timer.
- Each 25 minute segment is followed by a short break - the aim being to reduce the impact of internal and external interruptions on focus and flow.
- A pomodoro is indivisible; when interrupted during a pomodoro, either the other activity must be recorded and postponed or the pomodoro must be abandoned.

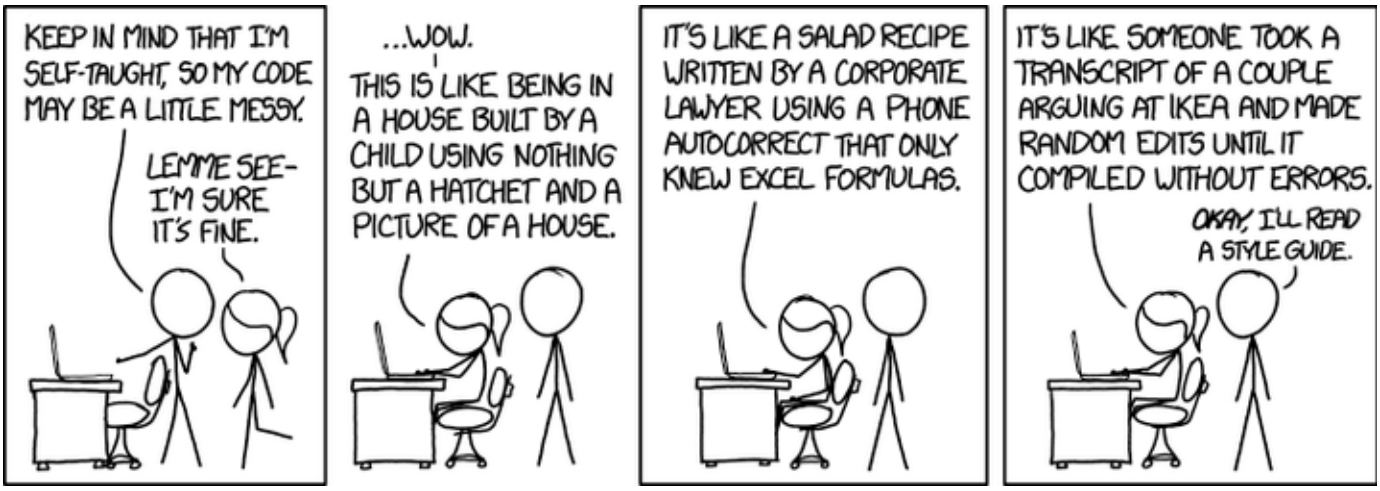
Rubber Duck Debugging

- From https://rubberduckdebugging.com/
  - Beg, borrow, steal, buy, fabricate or otherwise obtain a rubber duck (bathtub variety).
  - Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.
  - Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.
  - At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.
- Having to explain your code thoroughly and exactly forces you, by the need to be precise while helping someone else understand your problem, to pay very careful attention to all that you were previously just taking for granted (and so help find the route cause of your bug).

Test doubles

- Use objects that are designed to look and sometimes behave like their Production equivalents. There are 4 main types:
  - Fakes: returns the same value or performs the same behaviour all the time.
  - Stubs: Provide canned answers to calls, where the behaviour being testing depends on what an input does – return a fixed value, throw an exception, calculate a return value based on parameters, pull from a sequence of values, etc.
  - Mocks: objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed: A mock has expectations about the way it should be called, and a test should fail if it's not called that way. E.g. simulating a service
  - Spies: Stubs that also record some information based on how they were called.

Design patterns



- There are a number of design patterns - they can generally be categorised as creational, structural or behavioural.
- Of these, the creational patterns 'factory method', 'builder' and 'singleton' should be familiar, or if you have not seen them before then you should find them straight-forward.
- Likewise the 'command', 'observer' and 'chain of responsibility' behaviour patterns should be familiar to you.
- It is always good to have knowledge of other patterns, but the above are the most commonly applied.