General

Meyer's Uniform Access Principle

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation

This is a clear recommendation for using getters and setters with classes. You should not see method calls outside of the class, they should appear as properties of the object.

Don't Repeat Yourself

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

The Principle of Orthogonality

This notion comes from geometry. Two lines are orthogonal to each other if they form a right-angle when they meet.

Their meeting isn't the important part. Think of a simple x, y graph:

> If you move along one of the lines, **your position projected onto the other doesn't change**

In computing this is expressed in terms of **decoupling** and is implemented through modular, component-based architectures. As much as possible code should be scoped narrowly so that a change in one area does not cause changes in others. By keeping components discrete it is easier to make changes, refactor, improve and extend the codebase.

> We want to design components that are self-contained: independent and with a single, well-defined purpose. When components are isolated from one another, you know that you can change one without having to worry about the rest. As long as you don't change that component's external interfaces, you can be comfortable that you won't cause problems that ripple through the entire system.

Benefits of orthogonality: productivity

- Changes are localised so development time and testing time are reduced
- Orthogonality promotes reuse: if components have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned by their original implementors. The more loosely coupled your systems, the easier they are to reconfigure and reengineer.
- Assume that one component does M distinct things and another does N things. If they are orthogonal and you combine them, the result does M x N things. However if the two components are not orthogonal, there will

be overlap, and the result will do less. You get more functionality per unit effort by combining orthogonal components.

Benefits of orthogonality: reduced risk

- Diseased sections of code are isolated. If a module is sick, it is less likely to spread the symptoms around the rest of the system.
- Overall the system is less fragile: make small changes to a particular area and any problems you generate will be restricted to that area.
- Orthogonal systems are better tested because it is easier to run and design discrete tests on modularised components.

  Building a unit test us itself a an interesting test of orthogonality: what does it take to build and link a unit test? Do you have to drag in a large percentage of the rest of the system just to get a test to compile or link? **If so, you've found a module that is not well decoupled from the rest of the system**

Relationship between DRY and orthogonality

With DRY you're looking to minimize duplication within a system, whereas with orthogonality, you reduce the interdependency among the system's components. If you use the principle of orthogonality combined closely with the DRY principle, you'll find that the systems you develop are more flexible, more understandable and easier to debug, test, and maintain.

Reversibility

The principles of orthogonality and DRY result in code that is reversible. This means it is able to change in an agile way when the circumstances of its use and deployment change. This is important because when developing software in a business setting, the best decisions are not always made the first time around. By following the principles it should be relatively easy to change your program's interfaces, platform and scale. In other words, with the principle of orthogonality and DRY, refactoring becomes less of a chore.

Prototyping and Tracer Bullets

'Tracer bullets' are used by the military for night warfare. They are phosphorous bullets that are included in the magazines of guns alongside normal bullets. They are not intended to kill but instead light-up the surrounding area making it easier to see the terrain and target more efficiently.

The authors use the notion of tracer bullets as a metaphor for developing software at the early stages of a project. This is **not** the same thing as prototyping. A tracer bullet model is useful for building things that haven't been built before. They exist to 'shed light' on the project's needs and to help the client understand what they want.

They differ from prototypes in that they include integrated overall functionality but in a rough state. Whereas prototypes are more for singular, specific subcomponents of the project. Because tracer bullet models are joined-up in this way, even if they turn out to be inappropriate in some regard, they can be adapted and developed into a better form, without losing the core functionality.

> Tracer bullets work because they operate in the same environment and under the same constraints as the real bullets. They get to the target fast, so the gunner gets immediate feedback. And from a practical standpoint they are a relatively cheap solution. To get the same effect in code, we're looking for something that gets us from a requirement to some aspect of the final system quickly, visibly and repeatably.
>
> Tracer code is not disposable: you write it for keeps. It contains all the error-checking, structuring, documentation and self-checking that a piece of production code has. It simply is not fully functional. However, once you have made an end-to-end connection among the components of your system, you can check how close to the target you are, adjusting as necessary.

Distinguishing from prototyping

> Prototyping generates disposable code. Tracer code is lean but complete, and forms part of the skeleton of the final system. Think of prototyping as the reconnaissance and intelligence gathering that takes place before a single tracer bullet is fired.

Design by contract

To understand DBC we have to think of a computational process as involving two stages: the call and the execution of the routine that happens in response to the call (henceforth **caller** and **routine**).

- the caller could be a function expression that invokes a function and passes arguments to it expecting a given output. The function that executes is the routine
- the caller could be an object instantiation that calls a method belonging to its parent class
- the caller could be a parent React component that passes props to a child component

Design by contract means specifying clear and inviolable rules detailing what must obtain at both the call stage and the routine stage if the process is to execute.

Every function and method in a software system does something. Before it starts that something, the routine may have some expectation of the state of the world and it may be able to make a statement about the state of the world

when it concludes. These expectations are defined in terms of preconditions, postconditions, and invariants. They form that basis of a **contract** between the caller and the routine. Hence *design by contract**.***

Preconditions

Preconditions specify what must be true in order for the routine to be called. In other words, the requirements of the routine. What it needs and what should be the case before it even considers executing the task. A **routine should never get called when its preconditions would be violated**.

Postconditions

Providing the preconditions are met, this is what the routine is guaranteed to do. In other words: the state of affairs that must obtain after the routine has ran.

Invariants

Once established, the preconditions and postconditions should not change. If they need to change, that is a separate process and contract. In the processing of a routine, the data may be variant relative to the contract, but by the end the overall conditions establish the equilibrium of the contract.

There is an analogue here with functional programming philosophy: the function should always return the same sort of output, without ancillary processes happening, i.e side-effects.

One way to achieve this is to be miserly when setting up the contract, which overlaps with orthogonality. Only specify the minimum return on a contract rather than multiple postconditions. This only increases the likelihood that the contract will be breached at some point. If you need multiple postconditions, spread them out an achieve them in a compositional way, with multiple separate and modular processes.

> Be strict in what you will accept before you begin, and promise as little as possible in return. If your contract indicates that you'll accept anything and promise the world in return, then you've got a lot of code to write!

Division of responsibilities

> If all the routine's preconditions are met by the caller, the routine shall guarantee that all postconditions and invariants will be true when it completes.

Note that the emphasis of responsibilities is on the caller.

Imagine that we have a function that returns the count of an array of integers. It is not the job of the count routine to verify that it has been passed integers

and then to execute the count. Or, in the event that it is not passed integers, to mutate the data to integers and then execute.

This should be resolved by the caller: it is the responsibility of the caller to pass integers. If it doesn't, the routine simply crashes or raises an exception. It doesn't try to accommodate the input because that does not come down on its side of the contract. The caller has failed to meet the preconditions. If, due to some bug, the routine receives integers and fails to output the sum, then it has failed on its side

Example: type checking

An obvious example of this philosophy is when you perform checks or validation within your code (although validation is more of an issue when you are dealing with user data, not your own internal code). For instance using type checking with dynamically-typed languages.

When we use the `prop-types` library with React we are specifying preconditions: so long at the prop (effectively the caller) passed to the component (effectively the routine) is of type X, the component will render invariantly as R. If the prop is of type Y, an exception will be raised highlighting a breach in the contract.

Another example would be more advanced type checking with Javascript written using Typescript.

The Law of Demeter

Demeter's Law has applicability chiefly when programming with classes.

It's a fancy name for a simple principle summarised by 'don't talk to strangers'. Demeter's law is violated when code has more than one step between classes. You should avoid invoking methods of an object returned by another method. You should only use your own methods when dealing with it.

Formal

A method m of object O may only invoke the methods of the following kinds of objects:

- O itself
- m's parameters
- any objects created or instantiated within m
- O's direct component objects (in other words nested objects)
- a global variable (over and above O) accessible by O, within the scope of m

Model, View, Controller design pattern

The key concept behind the MVC idiom is separating the model from both the GUI that represents it and the controls that manage the view.

- **Model**
  - The abstract data model representing the target object
  - The model has no direct knowledge of any views or controllers
- **View**
  - A way to interpret the model. It subscribes to changes in the model and logical events from the controller
- **Controller**
  - A way to control the view and provide the model with new data. It publishes events to both the model and the view

For comparison, distinguish React from MVC. In React data is unidirectional: the JSX component as controller cannot change the state. The state is passed down to the controller. Also MVC lends itself to separation of technologies: code used to create the View is different from Code that manages Controller and data Model. In React it's all one integrated system.

Refactoring

> Rewriting, reworking, and re-architecting code is collectively known as refactoring

When to refactor

- **Duplication**: you've discovered a violation of the DRY principle
- **Non-orthogonal design**: you've discovered some code or design that could be made more orthogonal
- **Outdated knowledge**: your knowledge about the problem and you skills at implementing a solution have changed since the code was first written. Update and improve the code to reflect these changes
- **Performance: y**ou need to move functionality from one area of the system to another to improve performance

Tips when refactoring

- Don't try to refactor and add new functionality at the same time!
- Make sure you have good tests before you begin refactoring. Run the tests as you refactor. That way you will know quickly if your changes have broken anything
- Take short, deliberative steps. Refactoring often involves making many localised changes that result in a larger-scale change.

Testing

> Most developers hate testing. They tend to test-gently, subconsciously knowing where the code will break and avoiding the weak spots. Pragmatic Programmers are different. We are *driven* to find our bugs *now*, so we don't have to endure the shame of others finding our bugs later.

Unit testing

A unit test is code that exercises a module. It consists in testing each module in isolation to verify its behaviour. Unit testing is the foundation of all other forms of testing. If the parts don't work by themselves, they probably won't work well together. All the modules you are using must pass their own unit tests before you can proceed.

We can think of unit testing as **testing against contract** (detailed above). We want to test that the module delivers the functionality it promises over a wide range of test cases and boundary conditions.

Scope for unit testing should cover:

- Obviously, returning the expected value/outcome
- Ensuring that faulty arguments/ types are rejected and initiate error handling (deliberately breaking your code to ensure it is handled appropriately)
- Pass in the boundary and maximum value
- Pass in values between the zero and the maximum expressible argument to cover a range of cases

Benefits of unit testing include:

- It creates an example to other developers how to use all of the functionality of a given module
- It is a means to build **regression tests** which can be used to validate any future changes to the code. In other words, the future changes should pass the older tests to prove they are consistent with the code base

Integration testing

Integration testing shows that the major subsystems that make up the project work and play well with each other.

Integration testing is really just an extension of the unit testing described, only know you're testing how entire subsystems honour their contracts.

Commenting your code

In general, comments should detail **why** something is done, its purpose and its goal. The code already shows *how* it's done, so commenting on this is redundant, and violates the DRY principle.

We like to see a simple module-level comment, comments for significant data and type declarations, and a brief class and per-method header describing how the function is used and anything it does that is not obvious

```
/*
Find the highest value within a specified data range of samples

Parameter: aRange = range of dates to search for data
Parameter: aThreshold = minimum value to consider

Return: the value, or null if no value found that is greater than or equal to the threshold
*/
```