# FAULT TOLERANCE - WEBINAR

If reliability engineering isn't making you money, then you are doing it wrong

# Workbook

Presented by

**Dr Chris Jackson**

Reliability engineer

acuitas

reliability through insight


**Chris Jackson, PhD, CRE, CPEng**

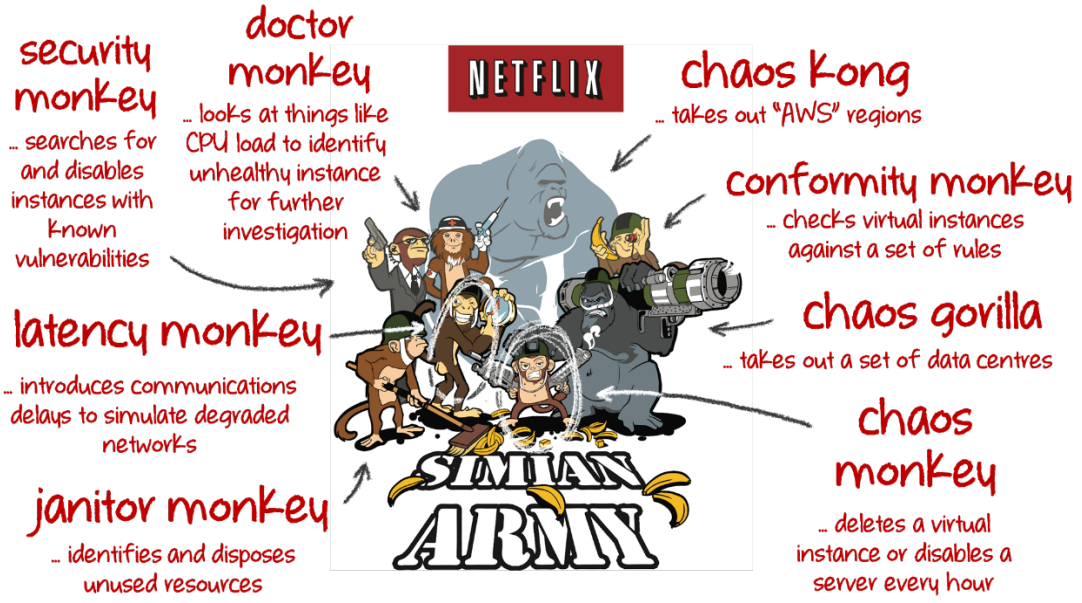
**chris.jackson@acuitas.com**


Reliability Engineer
Former Lieutenant Colonel in the Australian Army
Founder and director of Acuitas Reliability
Co-founder of www.is4.com education
Author of books on how to do 'reliability stuff'
Still learning

Once we have exhausted all practical and feasible **smart design activities**, and we still need to improve reliability, we might need to start **tolerating faults**. This could be entirely acceptable if you have really thought about reliability critically from the start. But like most reliability activities, it needs to be done right!

Netflix has a very reliable system for downloading data at high speed. And how do they do this? Using a process some call 'chaos engineering.' Chaos engineering is used in the development of software systems to make it able to tolerate faults. Experiments make sure the system can deal with unexpected situations. Just because we can't expect something, doesn't mean it won't happen.

Netflix does this by using several programs it refers to as the 'Simian Army.' 'Chaos monkey' was the first program created, and it deletes a virtual instance or disables a server every hour in a controlled way. Engineers can step in to prevent a system failure, but having observe what has happened allows the engineers to change the software code to make it more resilient.




Many organizations focus on improving reliability to the extent that they never measure reliability. This requires a level of cultural courage (that some organizations never get). Because amongst other things, improving reliability means that you need to push your system or product to the edge of its operational ability to work out its VITAL FEW weak points.

The most reliable of all systems are able to deal with things that were not anticipated. It is simply not possible to anticipate every possible scenario the thing you are creating will be subjected to.

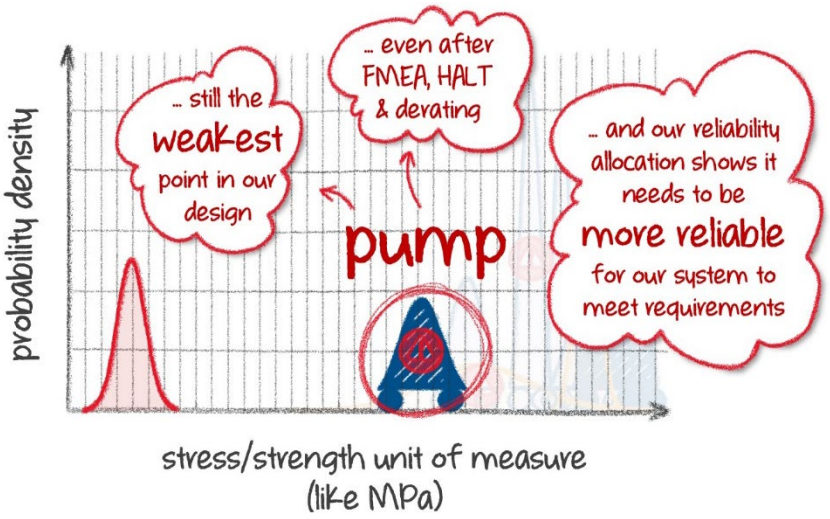
There are many things we need to do before we start thinking about tolerating faults. Starting with a (properly done) **FMEA** is rarely a bad decision. You might use your **FMEA** to help allocate reliability goals to your subsystems and components. And when this is done well, you have well thought out and flexible goals that help keep your team on track, on time, and on budget.

Individual design teams can then use a host of **Design for Reliability (DfR)** activities at their disposal. Perhaps they will conduct **Highly Accelerated Life Testing (HALT)** on initial prototypes. Perhaps they will derate. Once they have identified the **VITAL FEW failure mechanisms**, they can be targeted for design improvement.

But there comes a point where we cannot eliminate any more faults from our design – at least not efficiently or cheaply. This means that we now need to **tolerate faults**.

## How do we tolerate faults?

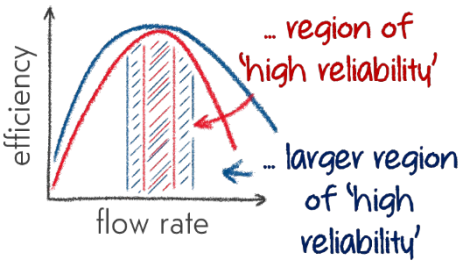
Take for example a pump.




There are **six** key methods for tolerating faults.

## 1. Variability control

**Variability control** involves using a **component that can tolerate larger variation in inputs caused by failures or faults in other components**. This is sometimes called **robustness**. It essentially means that the component keeps working well when confronted with 'out of spec' conditions.



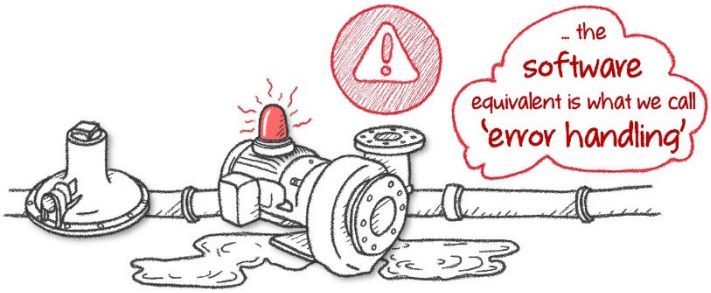
## 2. Fault Containment

### 2. Fault containment


**Fault containment** is the **prevention of the propagation of a failure mechanism across a system due to a fault in a single component**. For our pump, this could involve an upstream regulator.

There are lots of examples of **fault containment** that range from voltage regulators through to filters. Each component has a different range of options based on the dominant **failure mechanisms** (**VITAL FEW**).

## 3. Fault Isolation

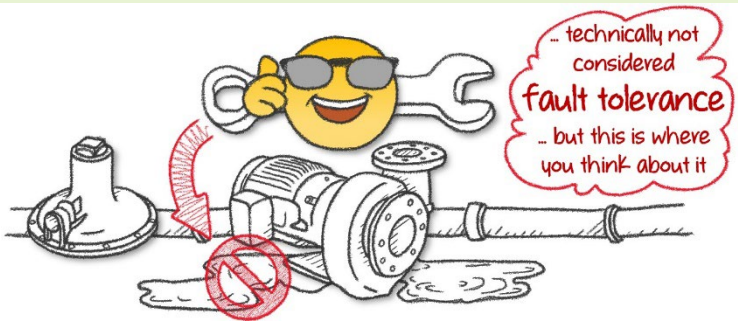



**Fault isolation involves diagnosing a fault when it occurs**. This allows immediate action to address the fault to either prevent it from progressing to a failure, or to minimise the time spent in a 'failed' state.

There are things like **Built-in Testing (BIT)**, **Built-in Self Testing (BIST)** and **Built-in Test Equipment (BITE)** which can help automate **fault isolation**. These systems might investigate the entire system during start up to make sure that performance characteristics are 'in specification,' the output voltage is within limits et cetera. In software, this process is largely analogous to the **error handling** process where software can detect when an **error** has occurred, what piece of code was responsible for the **error**, and otherwise advise the user on what he or she might be able to do.

## 4. Proactive Maintenance

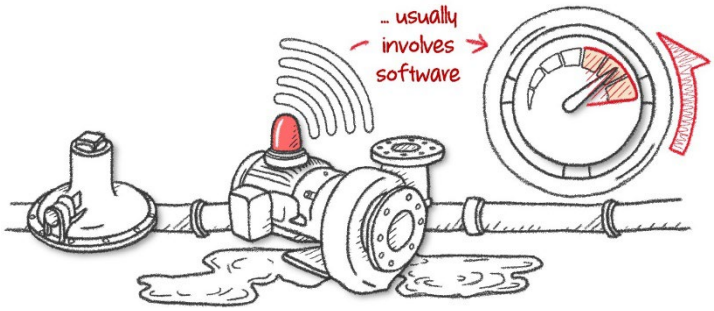
**Proactive maintenance** or **Condition Based Maintenance (CBM)** is all about **addressing faults before they become failures**. So it has a natural 'relationship' with **fault isolation**.




Many reliability engineers don't consider **proactive maintenance** or **CBM** a true '**fault tolerance**' activity. But in practice ... you need to thing about **proactive maintenance** and **CBM** at the point of design where you start thinking about **tolerating faults**. Instead of **BIT**, **BIST** or **BITE**, systems might have sensors that detect the accumulation of damage (lubricant contaminants, fatigue crack length, temperature et cetera) to allow maintenance to occur before **failure**.

## 5. Reversion

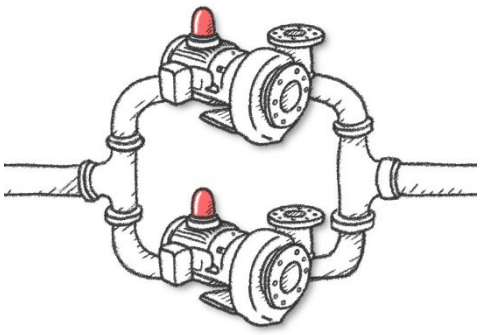
**Reversion** (which is sometimes referred to as **derating**) is the **transition to a sub-optimal operation state that prevents imminent failure caused by an isolated fault**. This too is related to **fault isolation**.

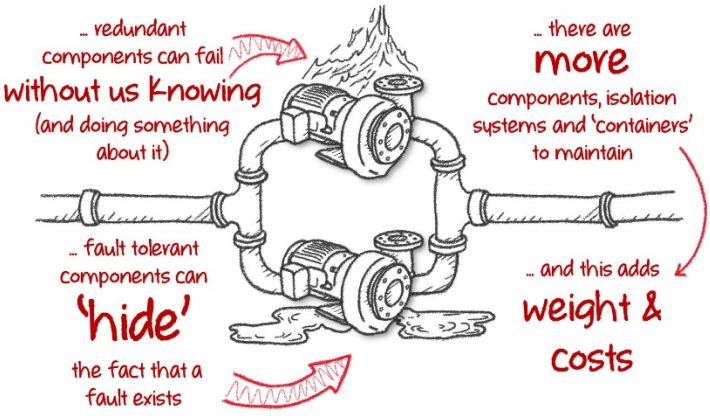



An example of reversion is an engine RPM limiter that enacts when a leak in the coolant system is identified. This will reduce the top speed of a vehicle but will allow the vehicle to continue operating. While **reversion** can be manual (as in, an operator manually detects a fault and adapts the use conditions accordingly), it is typically automated using sensors and software.

# 6. Redundancy

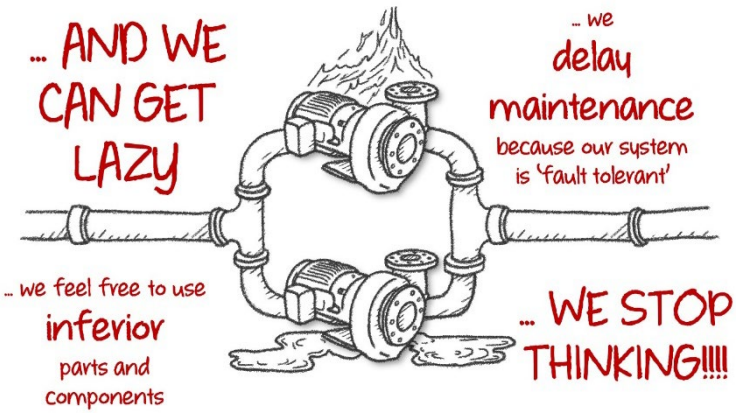
**Redundancy** is the **duplication of components or functions**. For a pump, this could involve the installation of a second **active redundant**, **standby redundant** or **load sharing pump**. The idea is that if one fails, the other can allow the system to continue operating.



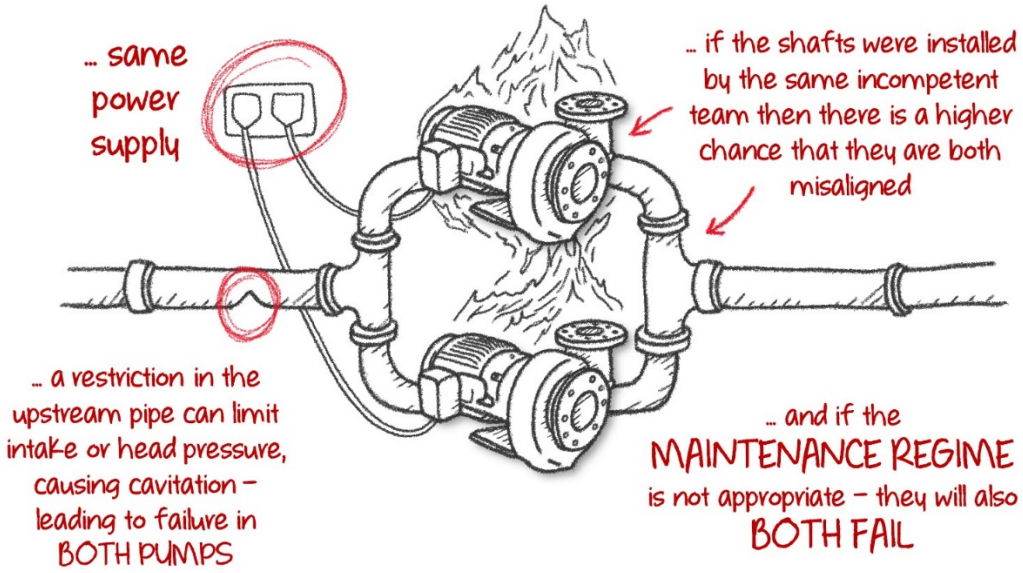



**Redundancy** comes at a cost – so it can't be implemented lightly. There is a hidden issue with redundancy – **Common Cause Failure (CCF)**. It is not uncommon for a single fault to cause the failure in **primary** and **standby components**, meaning we don't get the reliability improvements we hope for.

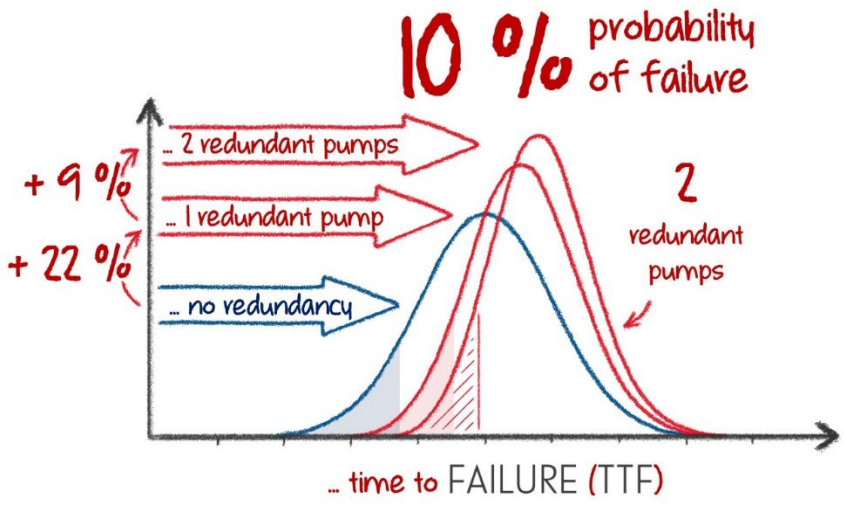
But perhaps one of the more egregious issues with **redundancy** can be the effect it can have on **culture**. **Redundancy** (and **fault tolerance**) often introduces a false sense of security. So we stop thinking and start assuming that everything is going to be OK. Our systems degrade to a point where we have no **fault tolerance**.



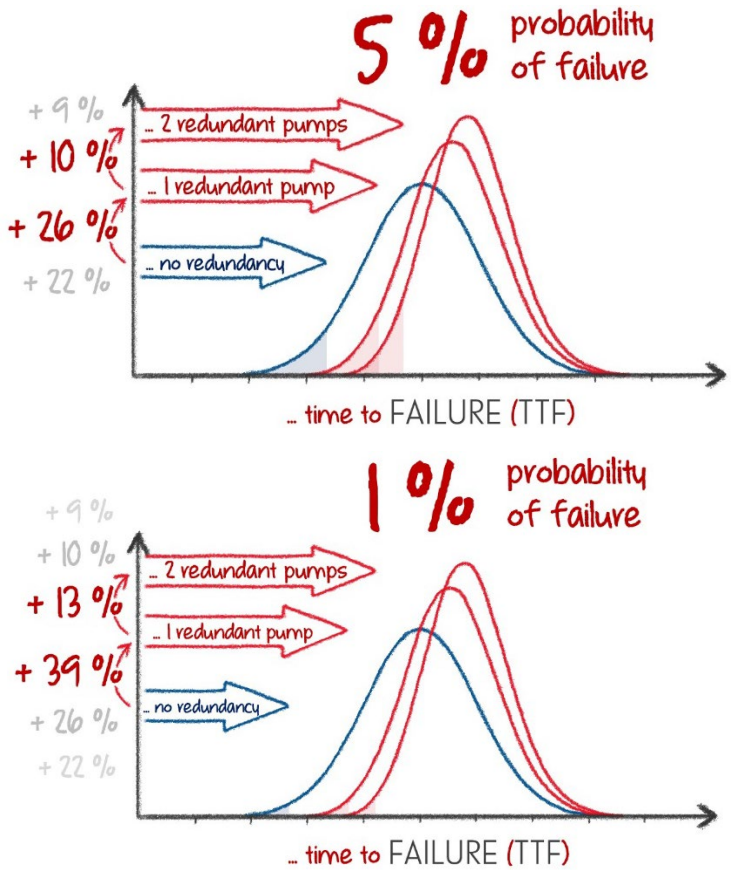
Let's return to **CCF** – with some examples illustrated below.



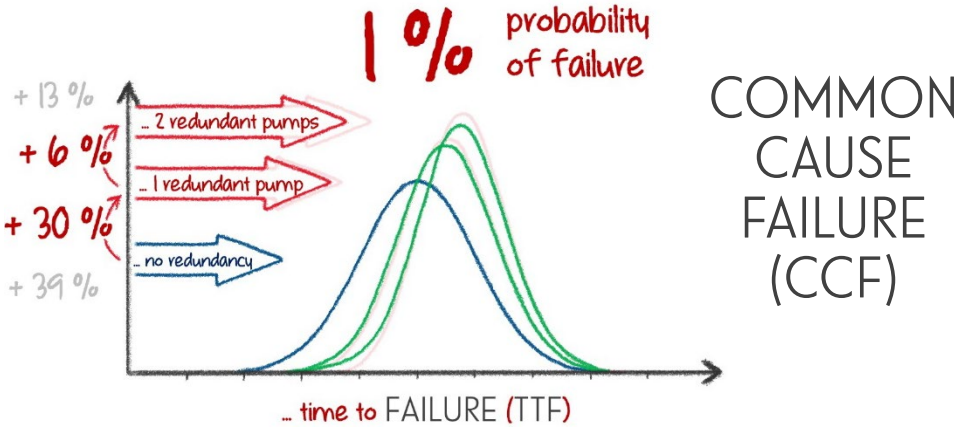

Let's now look at the effect **CCF** can have on redundant systems. Let's say that our **warranty reliability** must be at least 90 %. That means we can tolerate up to 10 % of our systems failing during the **warranty period** for our organization to remain profitable. For our particular pump with its **Time to Failure (TTF)** described by the blue density curve below, adding a single redundant pump extends our **warranty period by 22 %**. Adding another redundant pump extends our **warranty period by a further 9 %**. So you can see how redundancy has a decreasing benefit for additional redundant components.



The corresponding benefits for redundant pumps when our **warranty reliability** needs to be 95 % and 99 % are illustrated below.

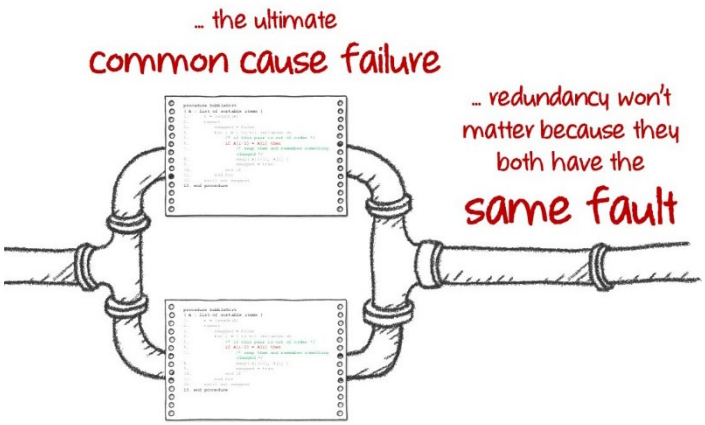


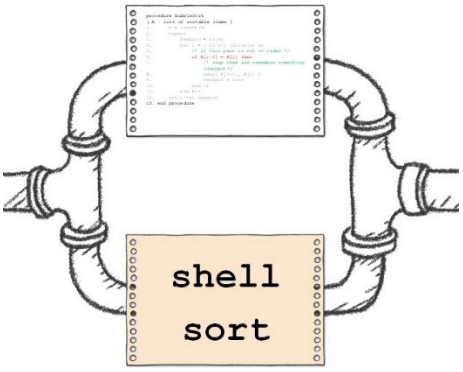

But if just one in ten failures have a **common cause**, then **we don't have redundancy one in ten 'times.'** So instead of getting an increase of 39 % on our **warranty period** (for a 99 % warranty reliability), **CCF** reduces this increase to 30 %. In short, **CCF is the enemy of redundancy**.



## 7. Redundancy – in software

**Software failure** is random – but in a different way to **hardware failure**. **Software failures** are caused by **coding errors** – which are obviously somewhat random. But once code is complete, software will behave in repeatable, non-random ways. So having defective code in '**redundancy**' will not work.

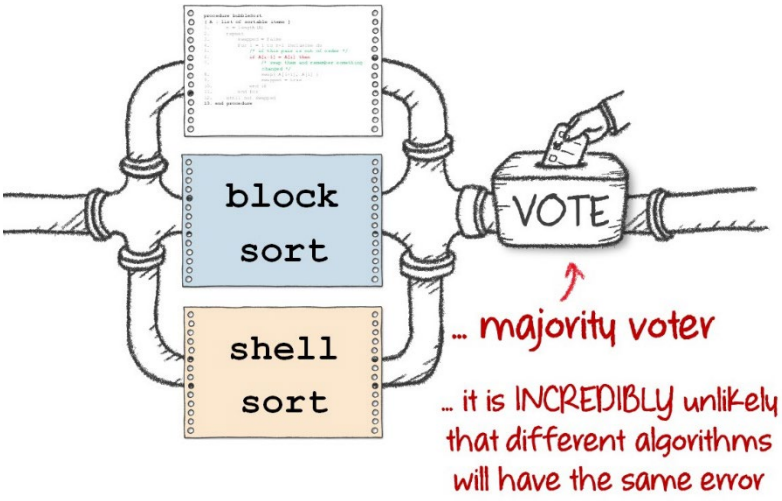



One approach is to have different code in a **redundant configuration**. For example, let's say we are coding software to simply sort numbers. A well-known algorithm is the '**bubble sort algorithm**.' We might get a team of software engineers to code a program based on the '**bubble sort algorithm**.' We then might ask another team to code another program based on the '**shell sort algorithm**.' Both algorithms achieve the same thing, but used different methods.

So we can put the **shell sort algorithm** in redundancy to support the **bubble sort algorithm**. Any **coding errors** in one program are unlikely to manifest themselves in exactly the same way in the other. But there is still a problem. If one program has an **error**, then this means the two programs will have different outputs. How do we know which one is correct?

The solution involves having three different programs, coded by different teams, all acting in '**redundancy**.' We then have a '**voter program**' which can identify when at least two outputs are identical. Because it is highly unlikely for two identical outputs to result from two different errors, we likely have a 'correct' output.



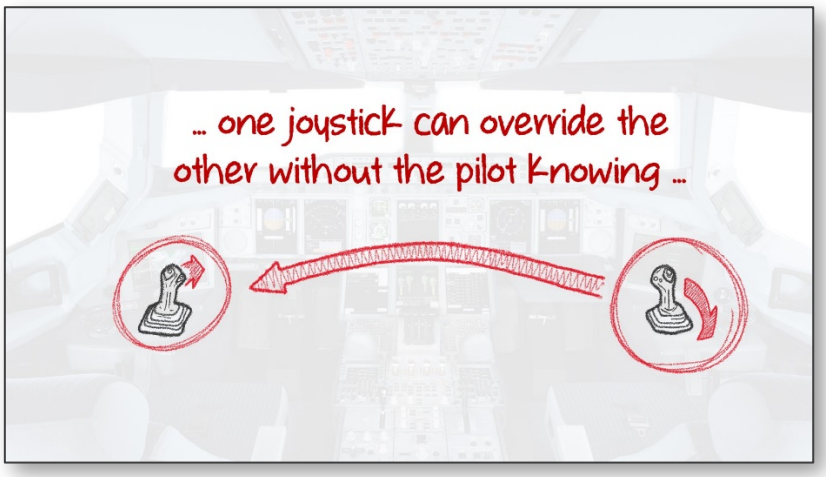

This is called **N-version programming** – where '**N**' refers to the number of 'redundant' programs.

There are lots of different approaches to software that make it **fault tolerant**. **N-version programming** is only one approach. Remember that **software error handling** is a form of **fault isolation**. This comes down to the coding approach and strategy, where 'little' things like coding in a coherent, commented manner that allows other software engineers to modify your code in the future is very valuable.

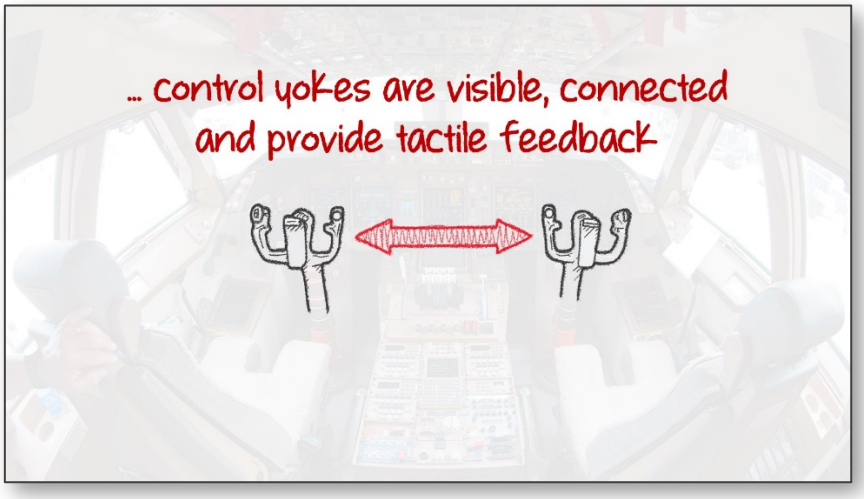
## 8. Human error

**Human reliability** and **human factors** are separate areas of study. Nuclear power plants are examples of complex systems where humans are just as important as the valves and pumps of the emergency cooling system. Any scenario where humans need to be trained and skilled to operate something automatically make them an integral part of a system.

Airplanes are also examples of systems where human (pilots) are an integral part of the system – and by extension reliability. Air France 447 was a flight where an Airbus A330 plunged into the Atlantic Ocean, killing all 228 occupants. One of the issues identified as a contributing factor was the sidestick or joystick controls in the cockpit (as illustrated below).



These controls are fundamentally different to the highly visible and connected control yokes of other airplanes.




The Air France 447 copilot for whatever reason pulled 'back' on the side stick, pushing the nose of the aircraft up. This place it in a stall condition and it started plummeting toward the ocean. The pilot was trying to push the nose down with his side stick – and he couldn't understand why the plane wasn't responding. By the time the crew realized what was happening – it was too late.

There is no point designing something that is 'very reliable' if it is difficult to operate in a reliable way.

# F•M•E•A

**4. proactive maintenance**
... addressing faults before they become failures

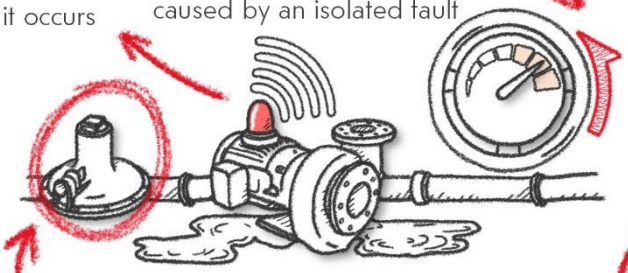
**3. fault isolation**
... diagnosing a fault when it occurs

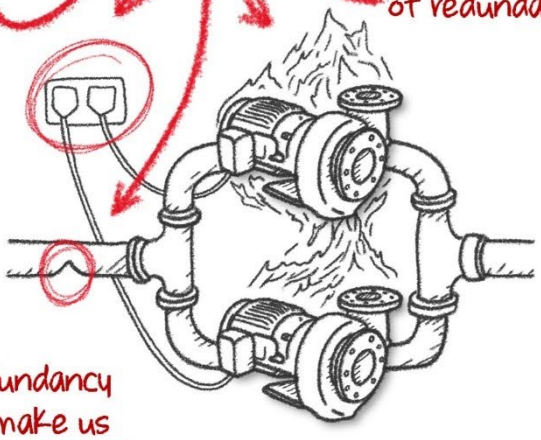
**5. reversion**
... transition to a sub-optimal operation state that prevents imminent failure caused by an isolated fault

... once we have exhausted all feasible and practical **SMART DESIGN** activities, we need to start TOLERATING FAULTS

RELIABILITY ALLOCATION

COMMON CAUSE FAILURE ... diminishes the positive benefit of redundancy



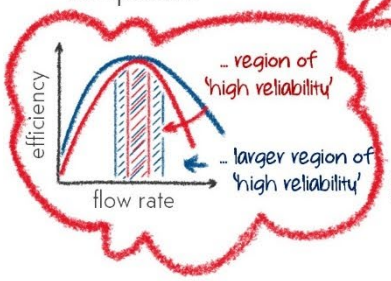


**2. fault containment**
... the prevention of the propagation of a failure mechanism across a system due to a fault in a single component

**1. variability control**
... the component can tolerate larger variation in inputs caused by failures or faults in other components

... redundancy can make us COMPLACENT

**6. redundancy**
... duplication of components or functions




redundant **SOFTWARE** programs must be coded by different programmers

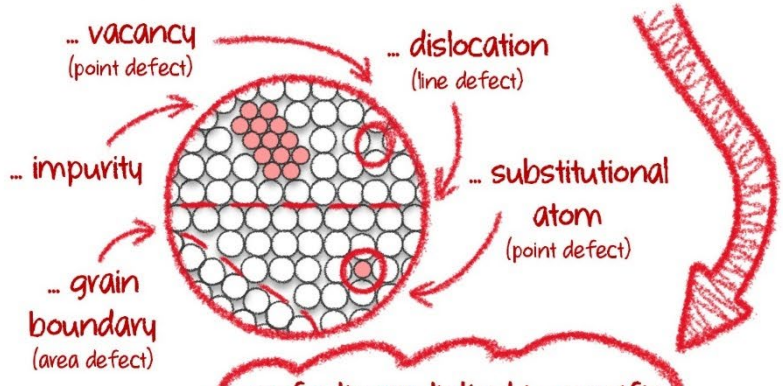
N-version programming

... fault injection

## fault tolerance

a **fault** is often defined as the immediate cause of failure OR an imperfection or deficiency in the software that may, under some conditions, contribute to a failure.

HARDWARE faults include impurities in metals that might start a fatigue crack, a crack in a seal that wasn't that allows moisture to migrate inside a housing which then goes on to cause corrosion and many others.




... so faults are linked to specific FAILURE MECHANISMS

a SOFTWARE fault might be a coding error where instead of two numbers being multiplied, they are added. Or the decimal place being put in the wrong spot of a number.

```
if A[i-1] > A[i] then
if A[i-1] = A[i] then
```

... the correct line of code is on the top ... the line of code with a fault is on the bottom