Reliable Programming

Fault avoidance

- IDEs show syntax errors

What else do we need to look out for other than syntax errors that our IDE takes care of?

- Logic errors

Quality attributes of reliable code:
Availability, security, resilience and reliability

Specialized reliability systems and approaches:

Used in **critical systems** where failure is unacceptable (like airplanes, medical devices, or nuclear plants):

- **"System malfunction can lead to death of the system"** = the program totally fails or even causes real harm.
- **Waterfall model** → A very structured software process used when safety/reliability is key (no room for sloppy updates).
- **Completely redundant systems** → Having backup systems that take over if one fails (e.g., two flight control computers running in parallel).

These may be too time consuming or expensive for most products

Programming for reliability
Writing code that keeps working correctly even when things go wrong.

- **Fault avoidance:** Prevent errors before they happen (careful design, testing).
- **Input validation:** Check user input so bad data doesn't break the program.
- **Failure management:** Handle errors gracefully (try-catch, backups, retries).

Causes for programming errors:

1. **Problem complexity-** we can make mistakes because the problem is challenging and difficult.
    a. The problem may not even seem that complex to be challenging Ex. recording two times: what if they are in different time zones, what if there is daylight savings, what if the country doesn't use time zones
2. **Program complexity**- if the program is too complex, developers slip up and make simple mistakes or fail to understand component connections
3. **Technology**- the tech you use itself can introduce faults,
    a. Ex. because there is a mismatch between the nature of the problem and tech you are trying to use as the solution
    b. The tech inherently has a lot of faults

Program complexity

- **Reading complexity**- how hard this is to read and understand
- **Structural complexity:** number and types of the relationships between components, classes and functions
- **Data complexity**- representation of data relationships between data
    - As relationships increase, things get more complicated
- **Decision complexity-** How many branches or conditions there are in your code
        - Each decision path adds possible outcomes; harder to test and maintain.

Single responsibility principle (SRP)

- Do one thing and one thing only
- Only has one reason to change

Bob martin- “gather together the things that change for one reason; separate those things that change for different reasons”
Example:
Class
DeviceInventory
Laptops
Tablets

Phones
device_assigments

Methods:
Adddevice
Movedevice
Assigndevice
Unassgndevice
Getdeviceassignent

This class manages assignments of devices, list of things its managing and the assignments

Let’s say you added a printIventory method above…
Bob Martin says you shouldn't, rather instead create an entirely new class

inventory Report
reportData
reportFormat

Methods:
updateData
updateFormat
print

This may feel like its increasing complexity, but making everything more bite sized it will help make the system less complex and easier to worth with in the long run

Avoid deeply nested conditional statements:

ageCheck - determine the rate of drivers insurance
YoungDriverAgeLimit = 25
OlderDriverAgeLimit= 70
ElderlyDriverAge= 80

YoungDriverPremiumMultiplier =

multipler : NO_MULTIPLIER
If age < YoungDriverAge:

```
        If experience < YoungDriverExperince
                Multiplier = PremiumYoung
        Else:
                Multiplier= PremiumYoung *IndexFactor
        Else:

                Return multiplier
```

*Instead … bob martin says to use **guards**:*

Use **guard clauses** to check simple conditions first and return early, instead of nesting lots of if statements inside each other.

This would look like:

Guards = switch statements

```
If age < YoungDriverLimit and experience < YoungDriverExperience:
        Return premiumYoung
If
        Return
If
        Return
```

SO instead of working deeper and deeper… you have a single return statement

- Code using guards is pretty reasonable

Avoid deeply nested inheritance:

- Instead of child classes, use field and guards in related code

Deeply Nested Inheritance (hard to manage)

```
class Nurse:
   pass

class TypeA(Nurse):
   pass
```

```
class TypeB(TypeA):
    pass

class TypeC(TypeB):
    pass
```

If you keep adding nurse types this way, it becomes confusing and fragile — changing one class might break others.

Better: Use Fields + Guards

```
class Nurse:
    def __init__(self, nurse_type):
        self.nurse_type = nurse_type

    def assign_task(self):
        if self.nurse_type == "A":
            return "Handles surgery prep"
        elif self.nurse_type == "B":
            return "Handles patient checkups"
        elif self.nurse_type == "C":
            return "Handles medication rounds"
```

Now all nurse types use **one class** with a **field** (nurse_type) and simple **guards** (if checks).
- Easier to read, change, and extend.

Cyclomatic Complexity: measures how many possible paths your code can take — basically, how many different ways it can run depending on decisions (if, else, loops, etc.).

- A single number that tries to represent logical complexity

- **E = edges** (connections between steps)
- **N = nodes** (individual steps or decisions)
- **P = components** (separate pieces of code)

2-3+2= complexity will be..1

There is only one path through this, but if we take a detour, then our complexity has doubled… So how many paths through this code are we taking?

If you add an if or loop (a “detour”), the number of paths goes up — **complexity doubles**, meaning more to test and more chances for bugs.

More paths leads to more unit tests

More paths = more possible ways the code can run → you need **more unit tests** to cover each path and make sure every condition works correctly.

So: Higher cyclomatic complexity → more tests → harder to maintain.

Reading Complexity

Example: G= ai*0.1*a2 ect…

What do these variable names mean? Harder to keep track so its important to use meaningful names