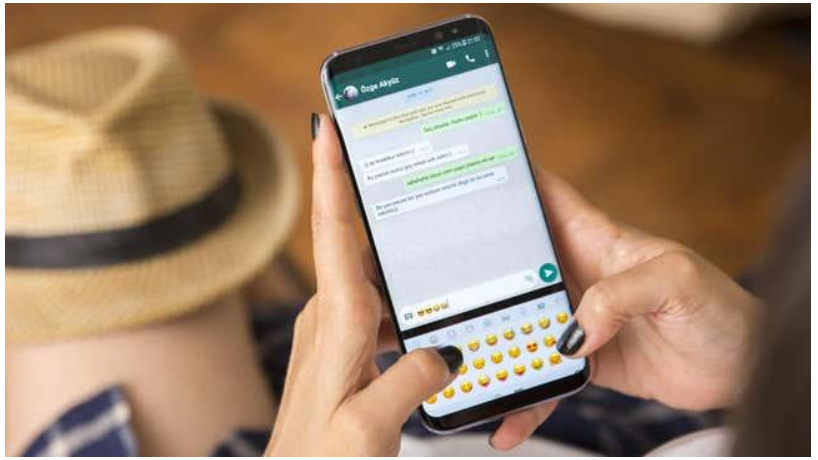
‘Reliable’ Programming

Examples from Sommerville’s 2018 book: “Engineering Software Products, An Introduction to Modern Software Engineering”

Software Quality: Reliability



- Reliability is similar to trust
- Users want to know that software will perform consistently each time they use it.
  - If WhatsApp only sent 50% of our texts, that wouldn't be very useful…
  - Likewise, we want Amazon to store keep our data safe…
  - We certainly want aeroplanes to take off and land safely 100% of the time!

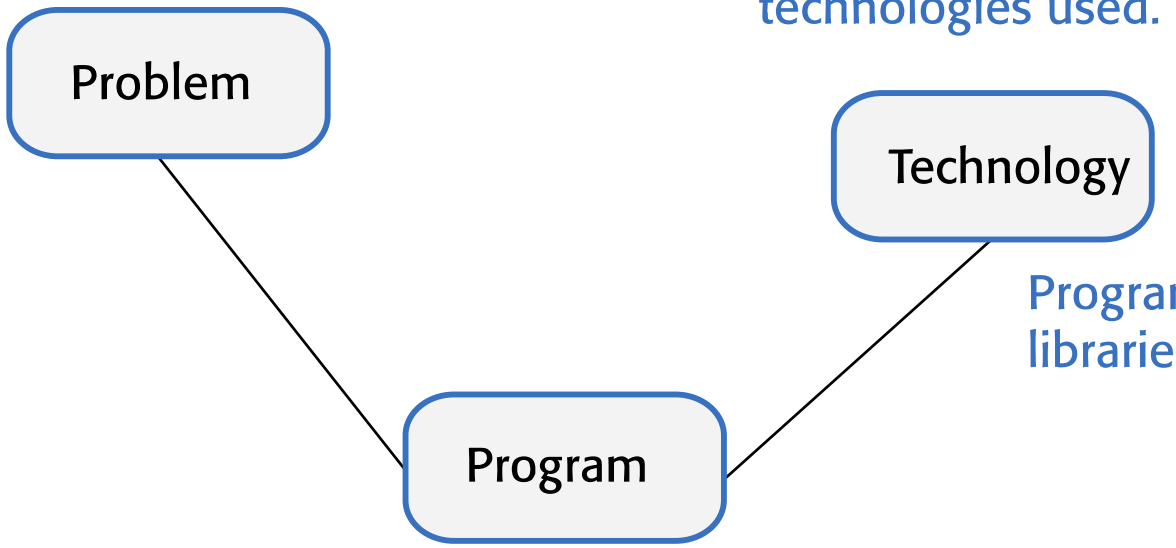
Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Three ways to improve reliability

- **Fault avoidance** - You should program in such a way that you avoid introducing faults into your program.
- **Input validation** - You should define the expected format for user inputs and validate that all inputs conform to that format.
- **Failure management** - You should implement your software so that program failures have minimal impact on product users.

Programmers make mistakes because they don't properly understand the problem or the application domain.

Programmers make mistakes because they use unsuitable technology or they don't properly understand the technologies used.



Programming language, libraries, database, IDE, etc.

Programmers make mistakes because they make simple slips or they do not completely understand how multiple program components work together and change the program's state.

Single Responsibility

- Classes should model **one** entity:
  - Student
  - Player
  - NOT Student_Player_and_Course
- Attributes of a class obviously store **one** value at a time
- Methods to perform **one** action:
  - print()
  - remove()
  - insert()
  - NOT print_and_remove_and_insert_new()

- Single responsibility encourages cohesion and reuse within programs

Types of Complexity: Structural

- **Structural complexity**
  - Functions should do one thing and one thing only
  - Functions should never have side-effects
  - Every class should have a single responsibility
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary

Types of Complexity: Conditional

- **Conditional complexity**
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions
- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- Consider the example code on the next slide

```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

Example of Condition Complexity

- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- For example, the function ‘age_check’ is a short Python function that is used to calculate an age multiplier for insurance premiums.
- The insurance company’s data suggests that the age and experience of drivers affects the chances of them having an accident, so premiums are adjusted to take this into account.
- It is good practice to name constants rather than using absolute numbers, so the program names all constants that are used.

```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

```python
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

```
# Return immediately for fewer 'else' statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            return YOUNG_DRIVER_PREMIUM_MULTIPLIER
     else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            return YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

```
# Return immediately for fewer 'else' statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        else:
            return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# Return immediately for fewer 'else' statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        else:
            return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# No 'else' statements!
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# No 'else' statements!
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

```
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE
        and experience <= OLDER_DRIVER_EXPERIENCE):
        return OLDER_DRIVER_PREMIUM_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

Original with deeply nested if statements

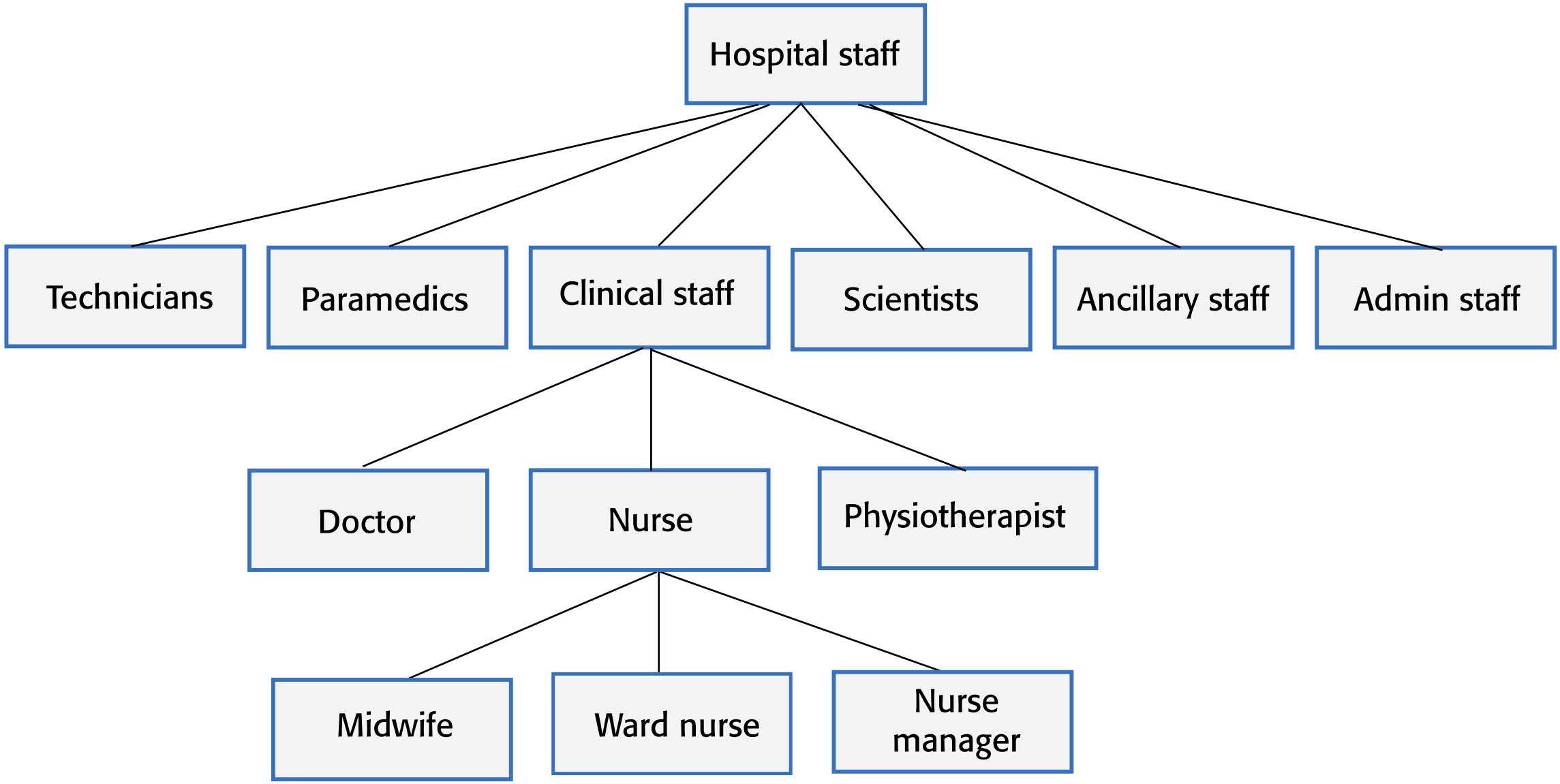
```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

Refactored to use guard clauses

```
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE
        and experience <= OLDER_DRIVER_EXPERIENCE):
        return OLDER_DRIVER_PREMIUM_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

Structural complexity: avoid deep inheritance

- Inheritance appears to be an effective and efficient way of reusing code and of making changes that affect all subclasses.
- However, inheritance increases the structural complexity of code as it increases the coupling of subclasses. The diagram shows part of a 4-level inheritance hierarchy that could be defined for staff in a hospital.


Hospital staff
Technicians
Paramedics
Clinical staff
Scientists
Ancillary staff
Admin staff
Doctor
Nurse
Physiotherapist
Midwife
Ward nurse
Nurse manager

Structural complexity: avoid deep inheritance

- The problem with deep inheritance is that if you want to make changes to a class, you have to look at all of its superclasses to see where it is best to make the change.
- You also have to look at all of the related subclasses to check that the change does not have unwanted consequences. It’s easy to make mistakes when you are doing this analysis and introduce faults into your program.

Measuring Quality: Code Metrics in VS22

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability ...	Cyclomatic Compl...	Class Coupling	Lines of Source ...	Lines of Executa...	Depth of In
ConsoleAppProject (Debug)	82	51	10	669	195	
ConsoleAppProject	74	1	3	24	5	
ConsoleAppProject.App01	91	6	2	74	14	
DistanceConverter	83	5	2	57	14	
DistanceUnits	100	1	0	11	0	
ConsoleAppProject.App02	100	1	0	12	0	
ConsoleAppProject.App03	91	1	1	23	4	
ConsoleAppProject.App04	80	36	6	363	66	
ConsoleAppProject.App05	56	6	1	173	106	

Maintainability

- Green 20 -1 00
- Yellow 10 – 19
- Red 0 – 9
- (Higher the better)

Complexity

- Lower the better

Lines of Code

- Lower the better

Coupling

- Lower the better

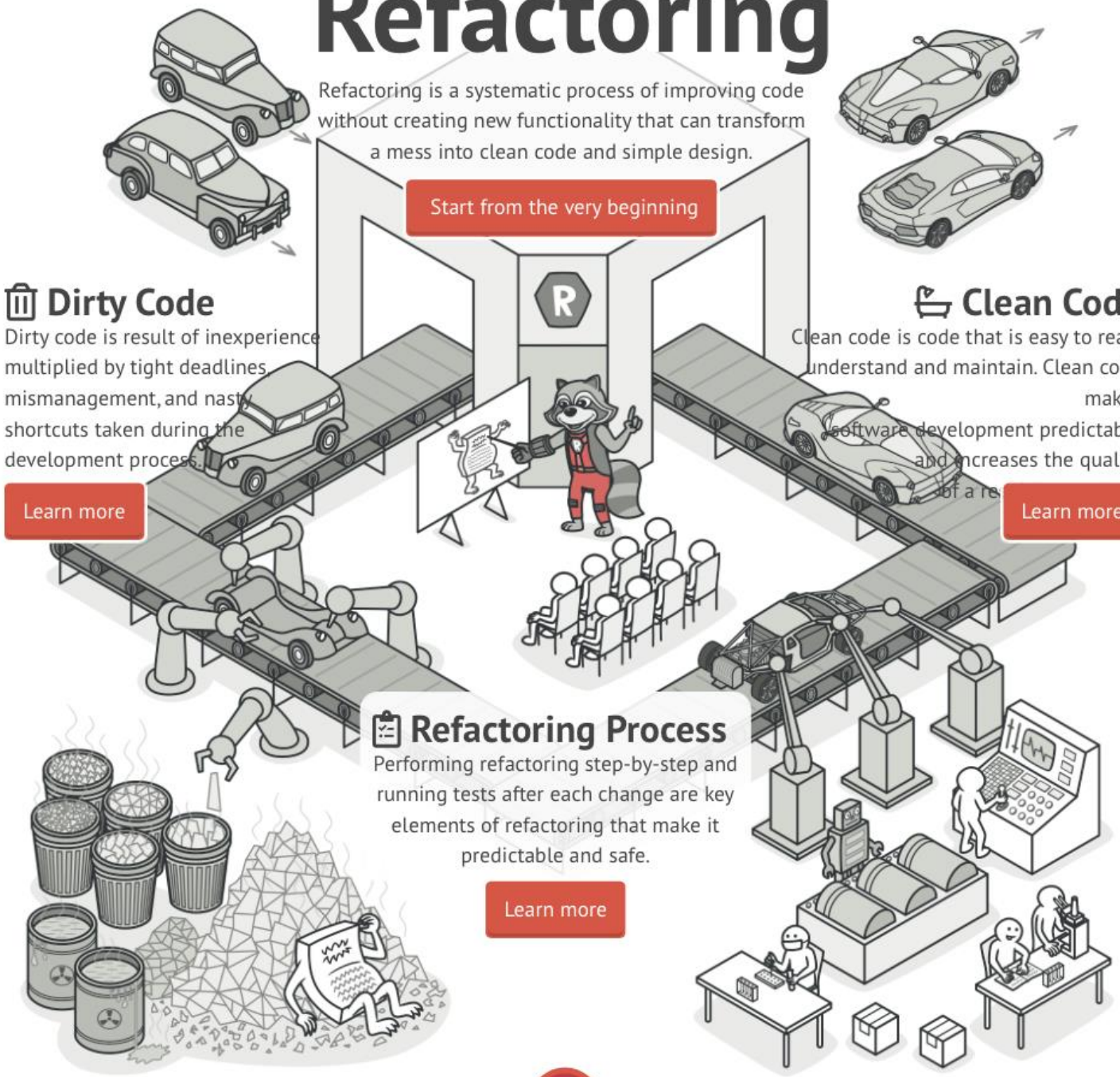
Inheritance Depth

- Lower the better

Refactoring

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Start from the very beginning



Dirty Code

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.

Learn more

Clean Code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a re

Learn more

Refactoring Process

Performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

Learn more

Code Smells

Code smells are indicators of problems tha addressed during refactoring. Code smells easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Refactoring Techniques

Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

Code ‘smells’

- **‘Code smells’ are indicators in the code that there might be a deeper problem.**
- Martin Fowler, a refactoring pioneer, suggests that the starting point for refactoring should be to identify code smells.
- For example, very large classes may indicate that the class is trying to do too much. This probably means that its structural complexity is high.

Martin Fowler

Examples of Code ‘smells’

Large classes

- Large classes may mean that the single responsibility principle is being violated.
- Break down large classes into easier-to-understand, smaller classes.

Long methods/functions

- Long methods or functions may indicate that the function is doing more than one thing.
- Split into smaller, more specific functions or methods.

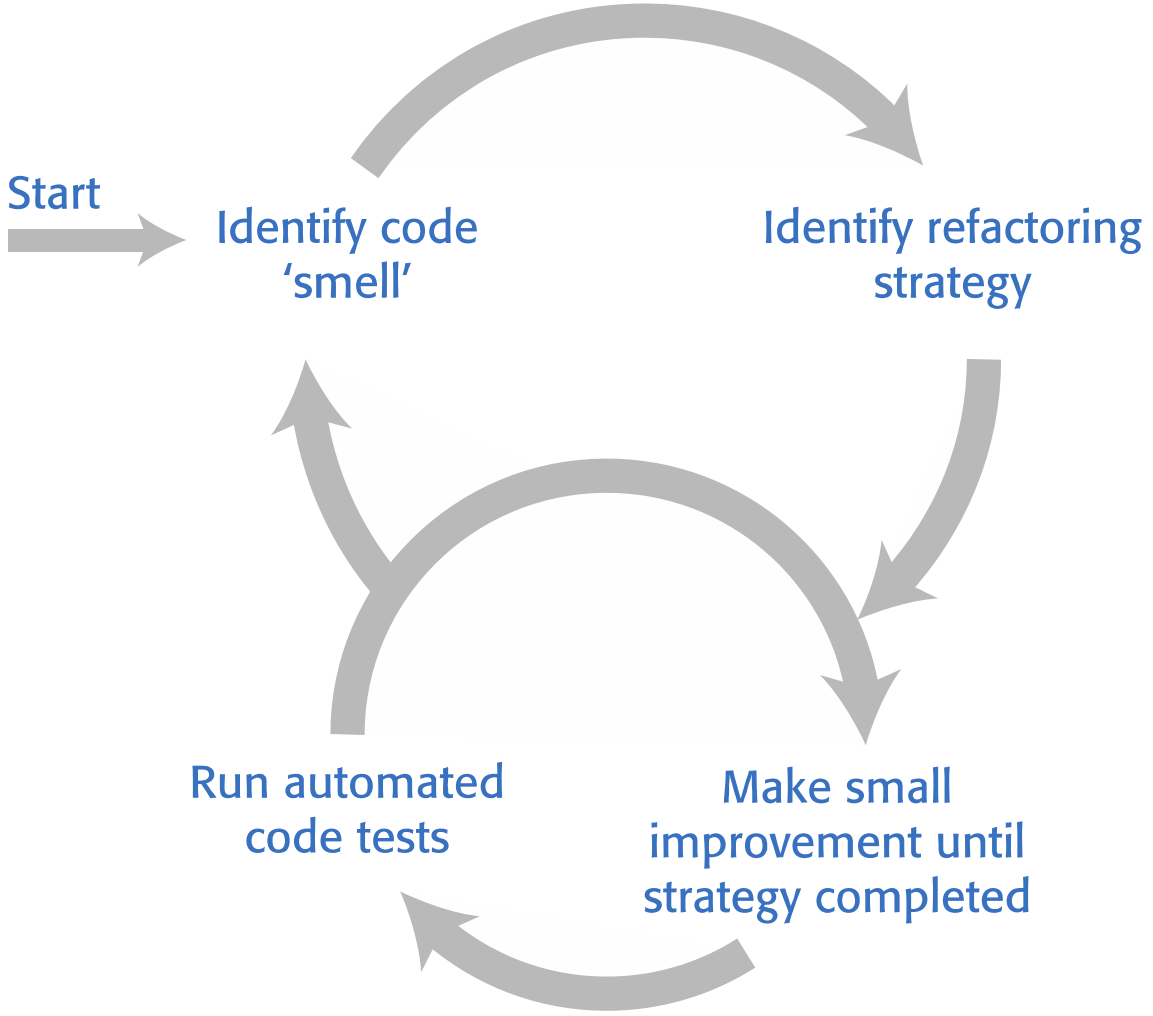
Duplicated code

- Duplicated code may mean that when changes are needed, these have to be made everywhere the code is duplicated.
- Rewrite to create a single instance of the duplicated code that is used as required

Meaningless names

- Meaningless names are a sign of programmer haste. They make the code harder to understand.
- Replace with meaningful names and check for other shortcuts that the programmer may have taken.

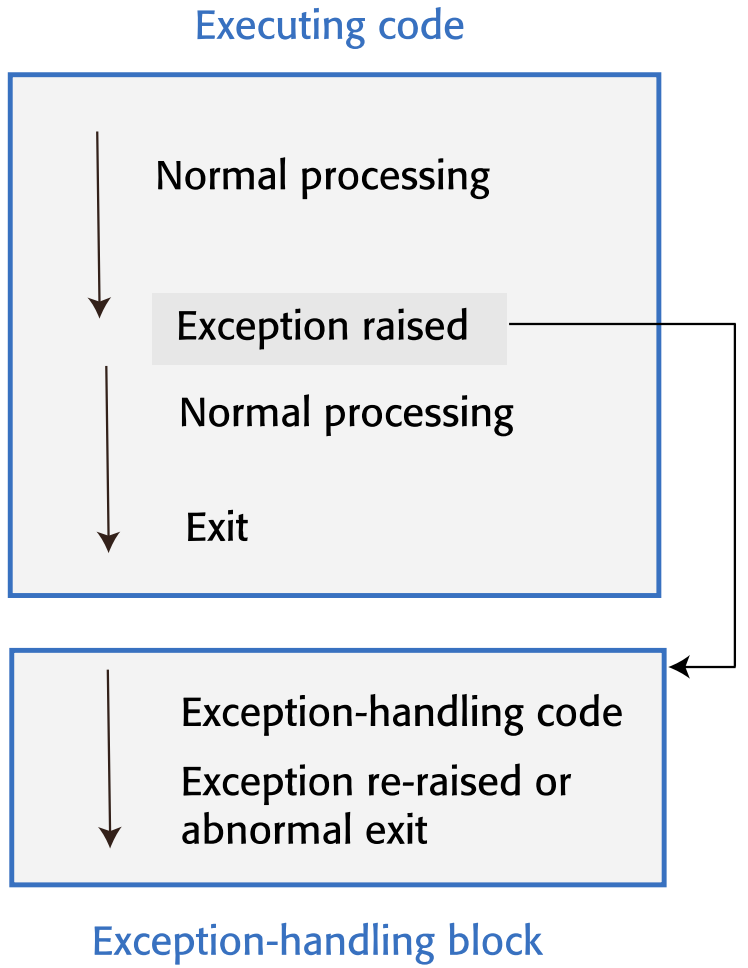
Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering



- Refactoring means changing a program to reduce its complexity without changing the external behaviour of that program.

Exception Handling

- Exceptions are events that disrupt the normal flow of processing in a program.
- In Python, you use **`try-except`** keywords to indicate exception handling code; in Java, the equivalent keywords are **`try-catch`**.



Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering

Exception Handling

- Code functions to raise (throw) exceptions
- Then these objects can be ‘caught’ elsewhere

```
1  def withdraw(amount, balance):
2      if amount > balance:
3          raise ValueError("Insufficient funds.")
4      balance -= amount
5      return balance
```

```
1  withdraw(100, 50)
```

```
---------------------------------------------------------------
ValueError                                Traceback (most rece
Cell In [60], line 1
----> 1 withdraw(100, 50)

Cell In [59], line 3
      1 def withdraw(amount, balance):
      2     if amount > balance:
----> 3         raise ValueError("Insufficient funds.")
      4     balance -= amount
      5     return balance

ValueError: Insufficient funds.
```

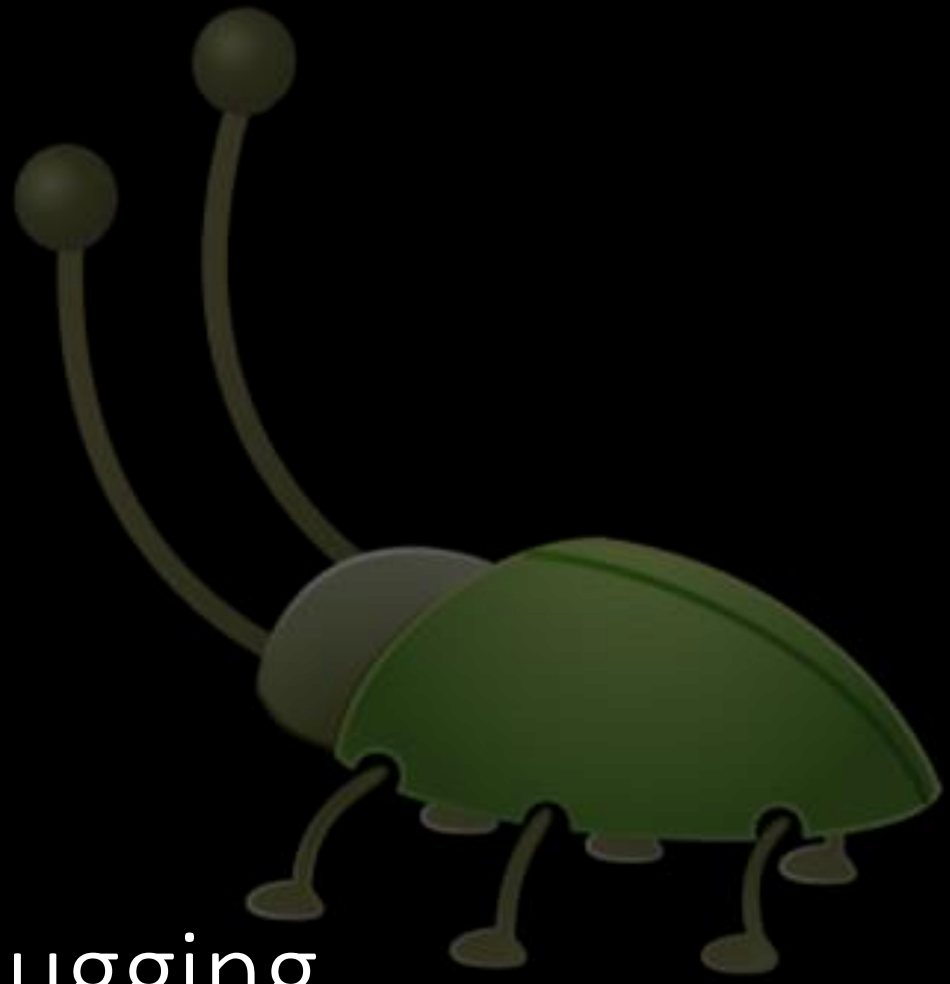
Assertions

- Assertions can be used to check parameters of methods, or values of variables.

```
1  x = 5
2  assert x > 10, "x has to be greater than 10"
```

⊗ 0.2s

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [1], line 2
      1 x = 5
----> 2 assert x > 10, "x has to be greater than 10"

AssertionError: x has to be greater than 10
```



Debugging

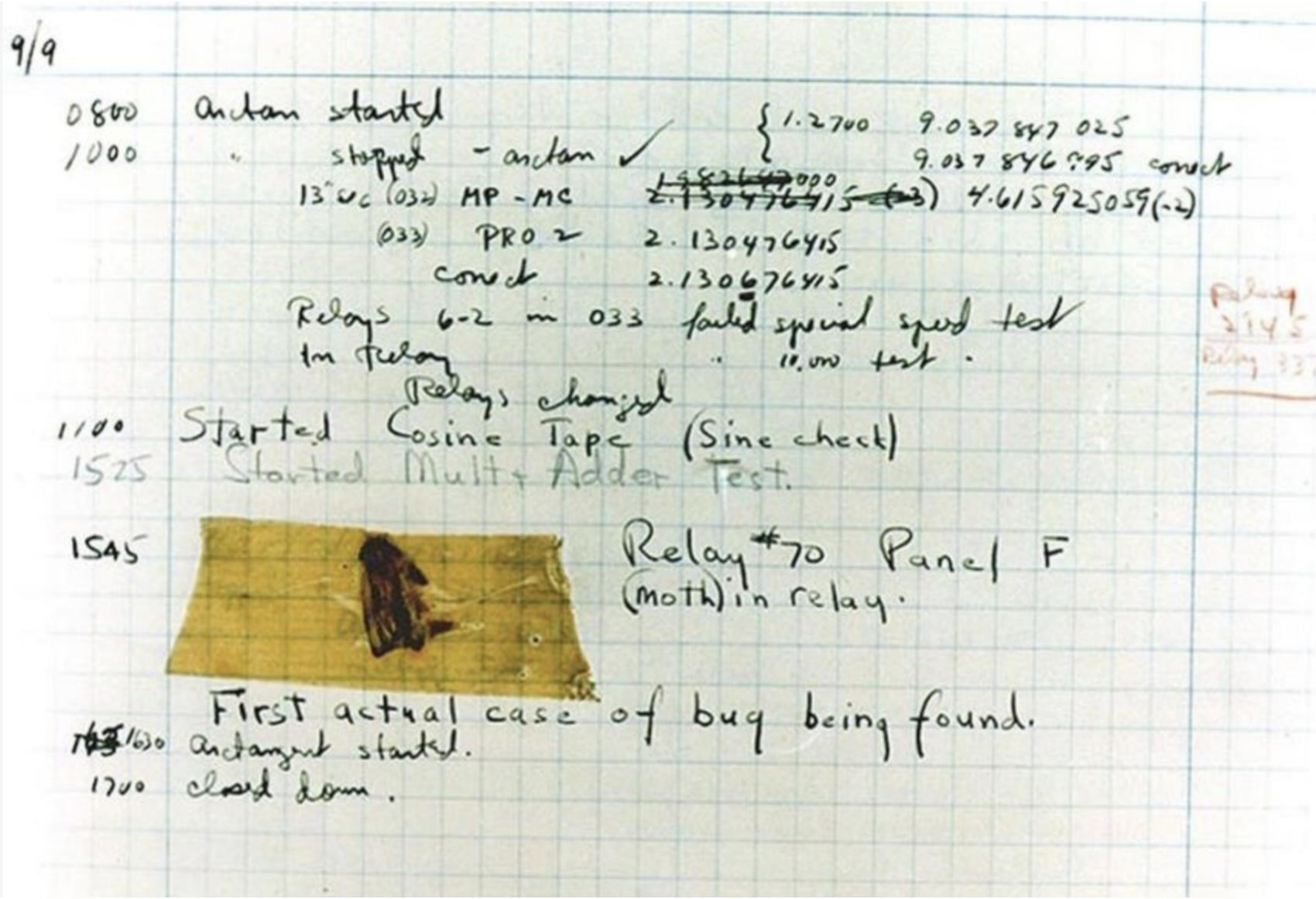
First bug!

PHOTOGRAPH

Computer Bug

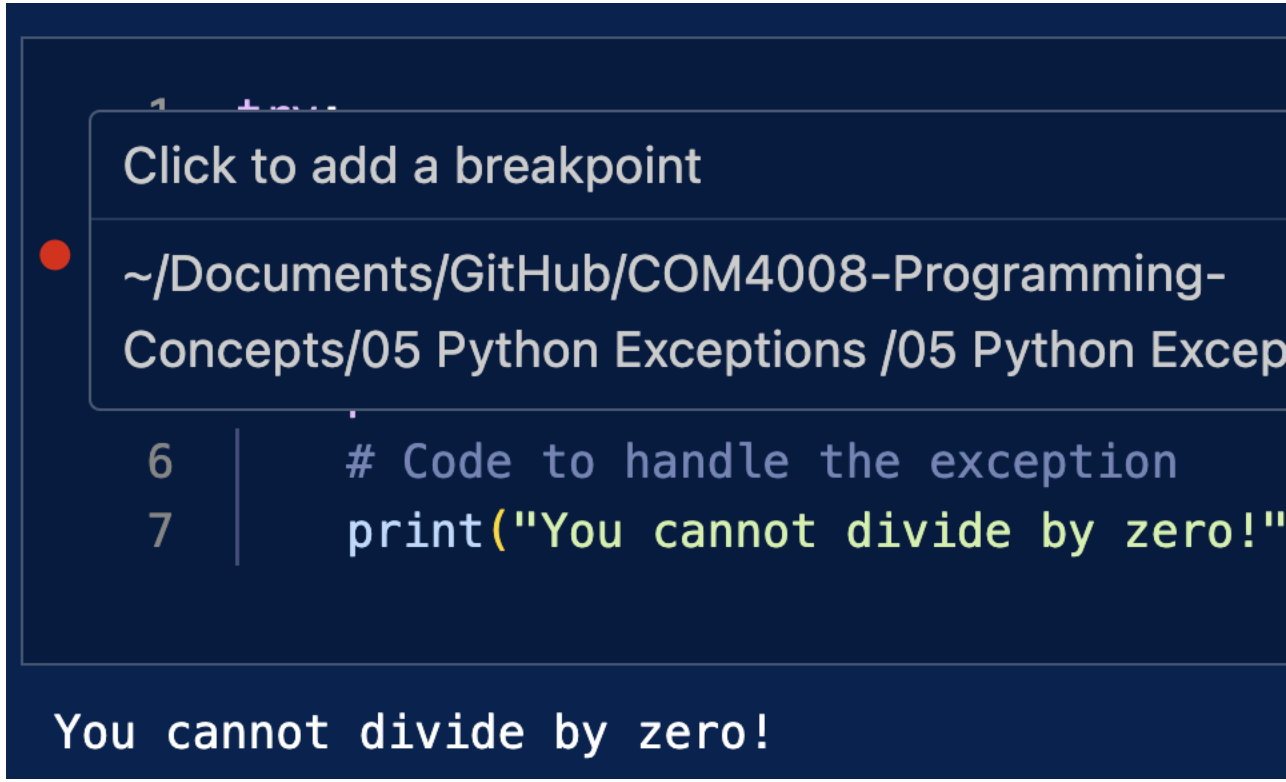
"First actual case of bug being found," according to the brainiacs at Harvard, 1945. The engineers who found the moth were the first to literally "debug" a machine.

> PHOTOGRAPH COURTESY NAVAL SURFACE WARFARE CENTER, DAHLGREN, VIRGINIA



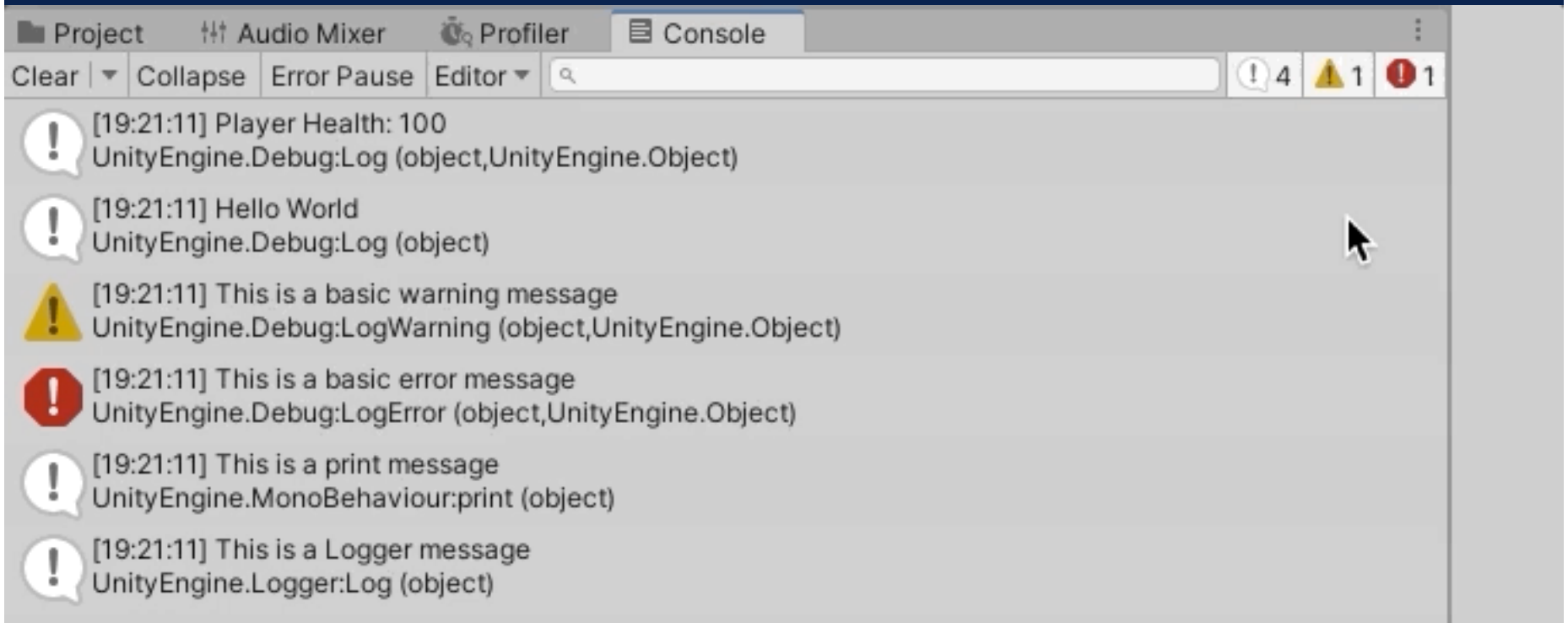
Run ‘with’ debugging

- Click just left of a line number in an IDE to add a breakpoint.
- Break points pause execution at specific lines so you can inspect variables and program flow.
- Step by step execution
- Debug Console (or log in Unity)
- Call Stack
- Exception Tracking



Unity Debug Console

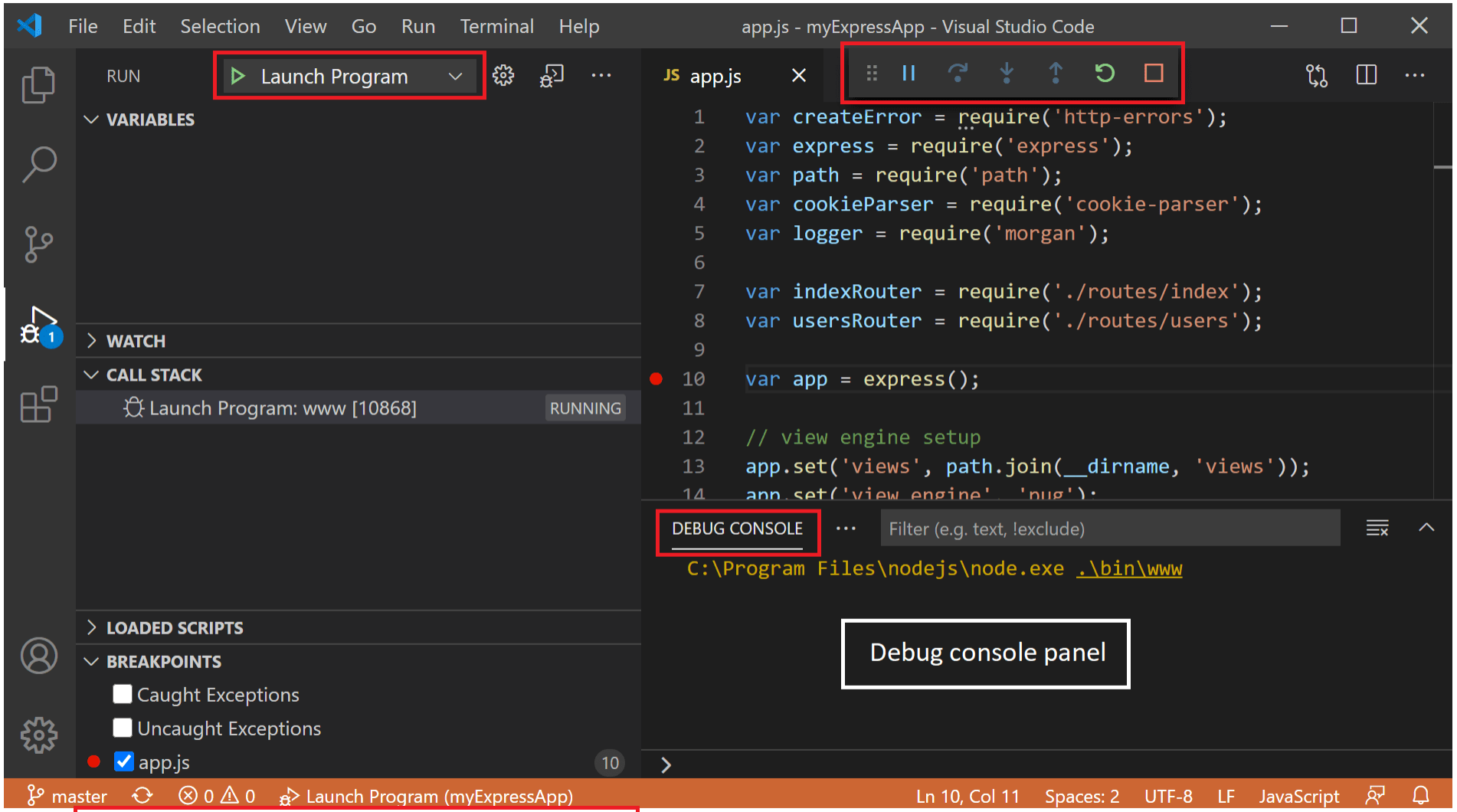
```
0 references
void OnCollisionEnter2D(Collision2D collision)
{
    //collisionText.text = "Collided with: " + collision.gameObject.name;
    // Print a message to the Console when a 2D collision occurs
    Debug.Log("Collided with: " + collision.gameObject.name);
}
```



Visual Studio Code Debug

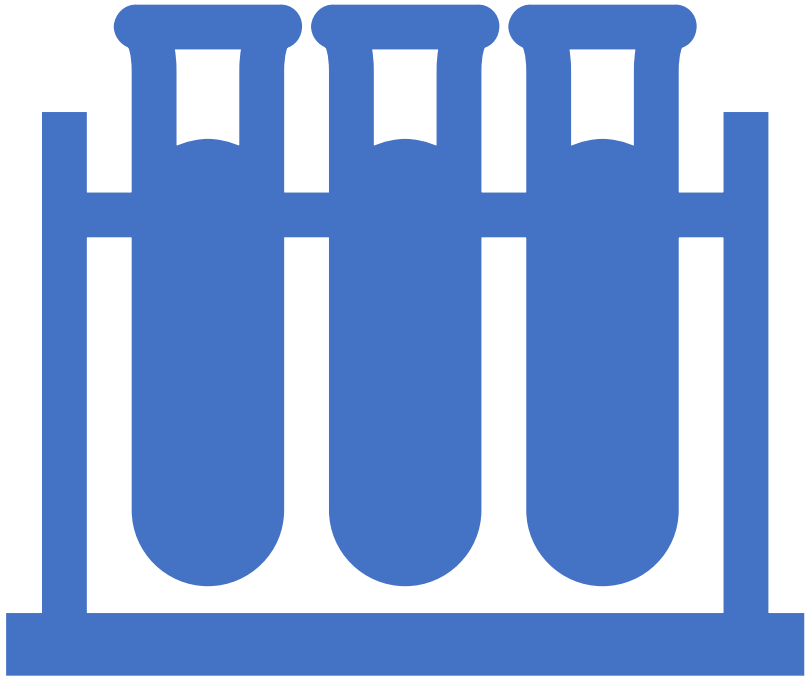
Start debugging

Pause, step over, step in/out, restart, stop



Debug side bar

https://code.visualstudio.com/docs/editor/debugging



Software Testing

Software Testing

- Tests are typically binary with software: either it performs as you expect or it doesn't perform as you expect…
- Tests will take a variety of formats, from manual observation to automation, but usually adhere to the following ruleset:
  - Tests pass if the behaviour is as you expect
  - Tests fail if the behaviour differs from that expected

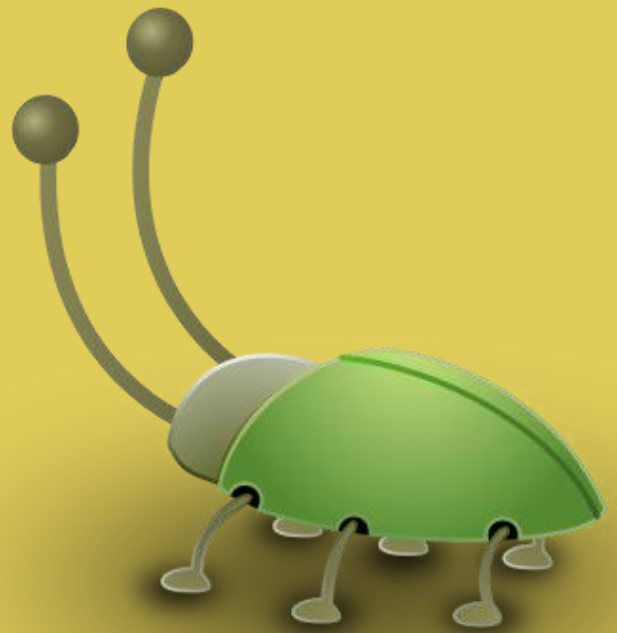
Program bugs

If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.

There are two causes of program bugs:

- ***Programming errors*** - You have accidentally included faults in your program code. For example, a common programming error is an 'off-by-1' error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.

- ***Understanding errors*** - You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn't include code to handle this.



BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Types of testing

Functional testing

Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.

User testing

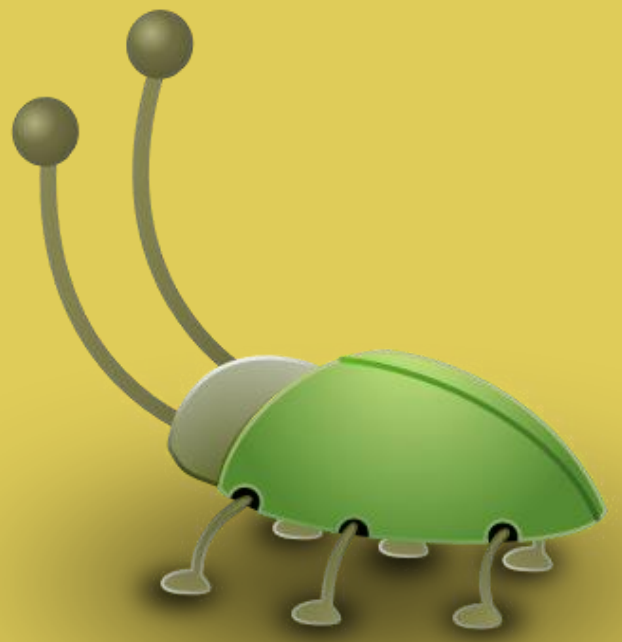
Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.

Performance and load testing

Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.

Security testing

Test that the software maintains its integrity and can protect user information from theft and damage.



Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

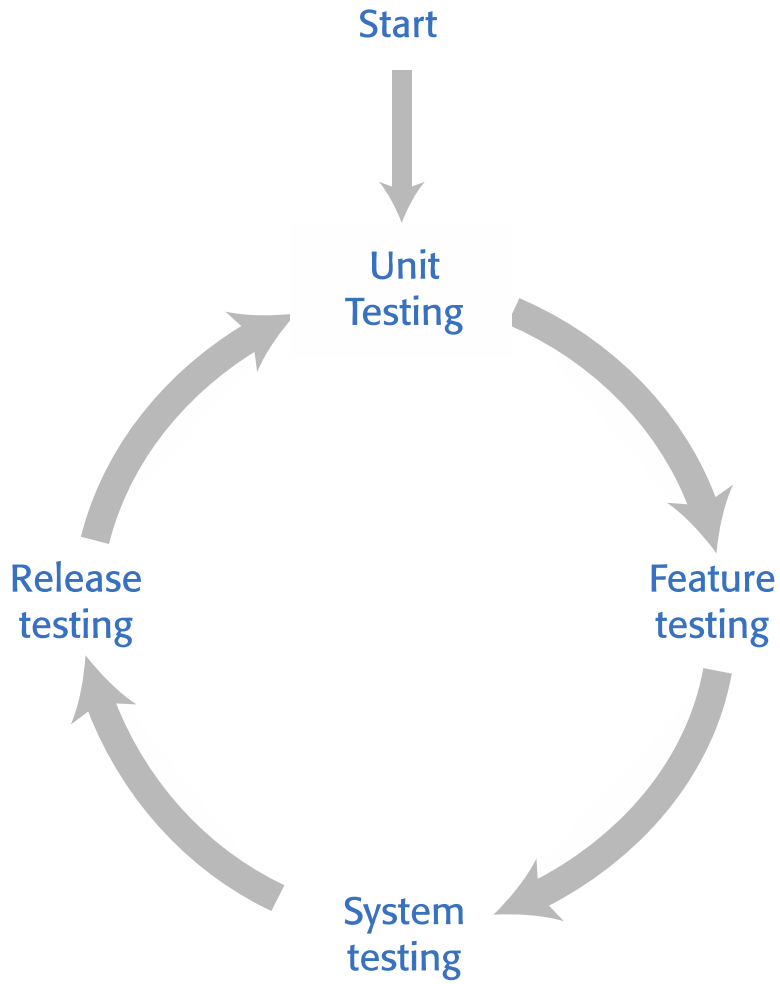
Functional testing processes (1)

Unit testing

- The aim of unit testing is to test program units in isolation.
- Tests should be designed to execute all of the code in a unit at least once.
- Individual code units are tested by the programmer as they are developed.

Feature testing

- Code units are integrated to create features.
- Feature tests should test all aspects of a feature.
- All of the programmers who contribute code units to a feature should be involved in its testing.



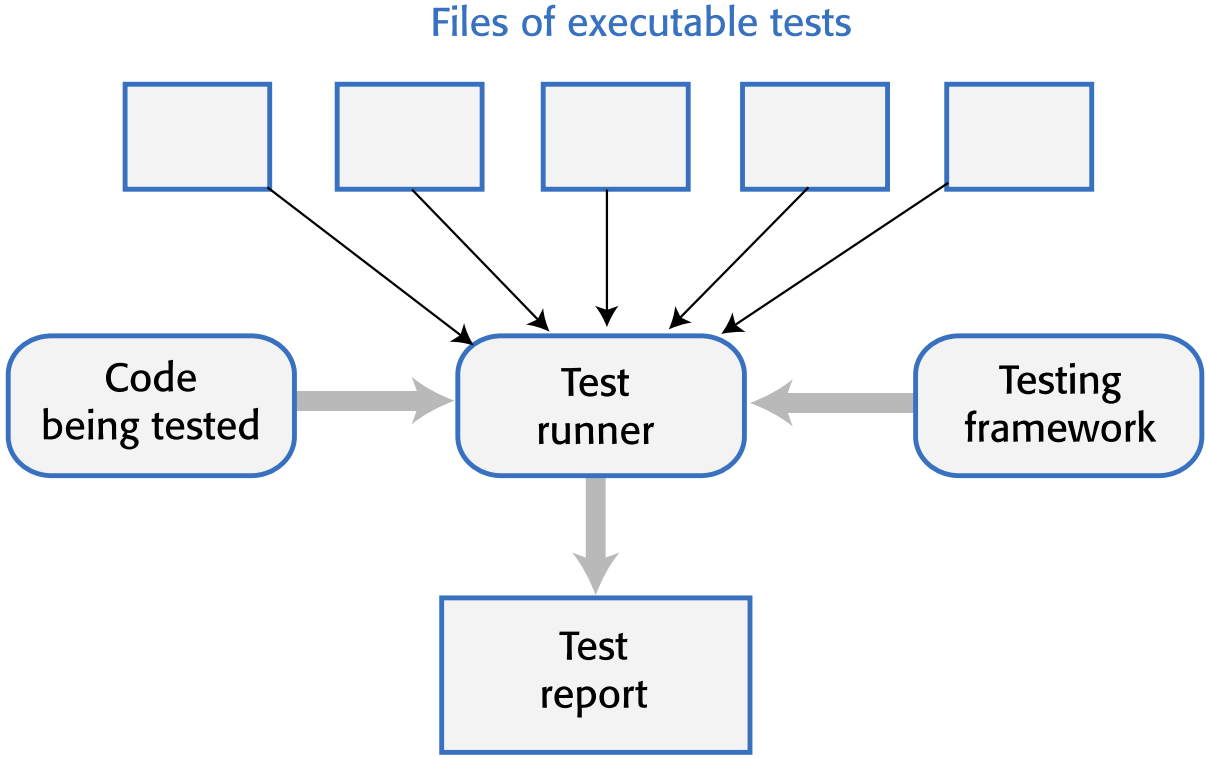
Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering

Test Automation

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Test automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product.



Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Automated tests

It is good practice to structure automated tests into three parts:

Arrange - You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

Action - You call the unit that is being tested with the test parameters.

Assert - You make an assertion about what should hold if the unit being tested has executed successfully.

If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

```
# TestInterestCalculator inherits attributes and
# methods from the class
# TestCase in the testing framework unittest

class TestInterestCalculator (unittest.TestCase):
    # Define a set of unit tests where each test tests
    # one thing only
    # Tests should start with test_ and the name should
    # explain what is being tested

    def test_zeroprincipal (self):
          #Arrange - set up the test parameters
          p = 0; r = 3; n = 31
          result_should_be = 0
          #Action - Call the method to be tested
          interest = interest_calculator (p, r, n)
          #Assert - test what should be true
          self.assertEqual (result_should_be, interest)

    def test_yearly_interest (self):
          #Arrange - set up the test parameters
          p = 17000; r = 3; n = 365
          #Action - Call the method to be tested
          result_should_be = 270.36
          interest = interest_calculator (p, r, n)
          #Assert - test what should be true
          self.assertEqual (result_should_be, interest)
```

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Test name_check function

```
import unittest
from RE_checker import namecheck
class TestNameCheck(unittest.TestCase):
   def test_alphaname(self):
           self.assertTrue(namecheck('Sommerville'))
   def test_doublequote(self):
           self.assertFalse(namecheck("Thisis'maliciouscode'"))
   def test_namestartswithhyphen(self):
           self.assertFalse(namecheck('-Sommerville'))
   def test_namestartswithquote(self):
           self.assertFalse(namecheck("'Reilly"))
   def test_nametoolong(self):
           self.assertFalse(namecheck
   ('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))
   def test_nametooshort(self):
           self.assertFalse(namecheck('S'))
   def test_namewithdigit(self):
           self.assertFalse(namecheck('C-3PO'))
   def test_namewithdoublehyphen (self):
           self.assertFalse (namecheck ('--badcode'))
   def test_namewithhyphen(self):
           self.assertTrue(namecheck('Washington-Wilson'))
```

```
import unittest

loader = unittest.TestLoader()

# Find the test files in the current directory

tests = loader.discover('.')

# Specify the level of information provided
# by the test runner

testRunner =
unittest.runner.TextTestRunner(verbosity=2)
testRunner.run(tests)
```

Unit testing guidelines

Test edge cases
If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.

Force errors
Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.

Fill buffers
Choose test inputs that cause all input buffers to overflow.

Repeat yourself
Repeat the same test input or series of inputs several times.

Overflow and underflow
If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.

Don't forget null and zero
If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.

Keep count
When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.

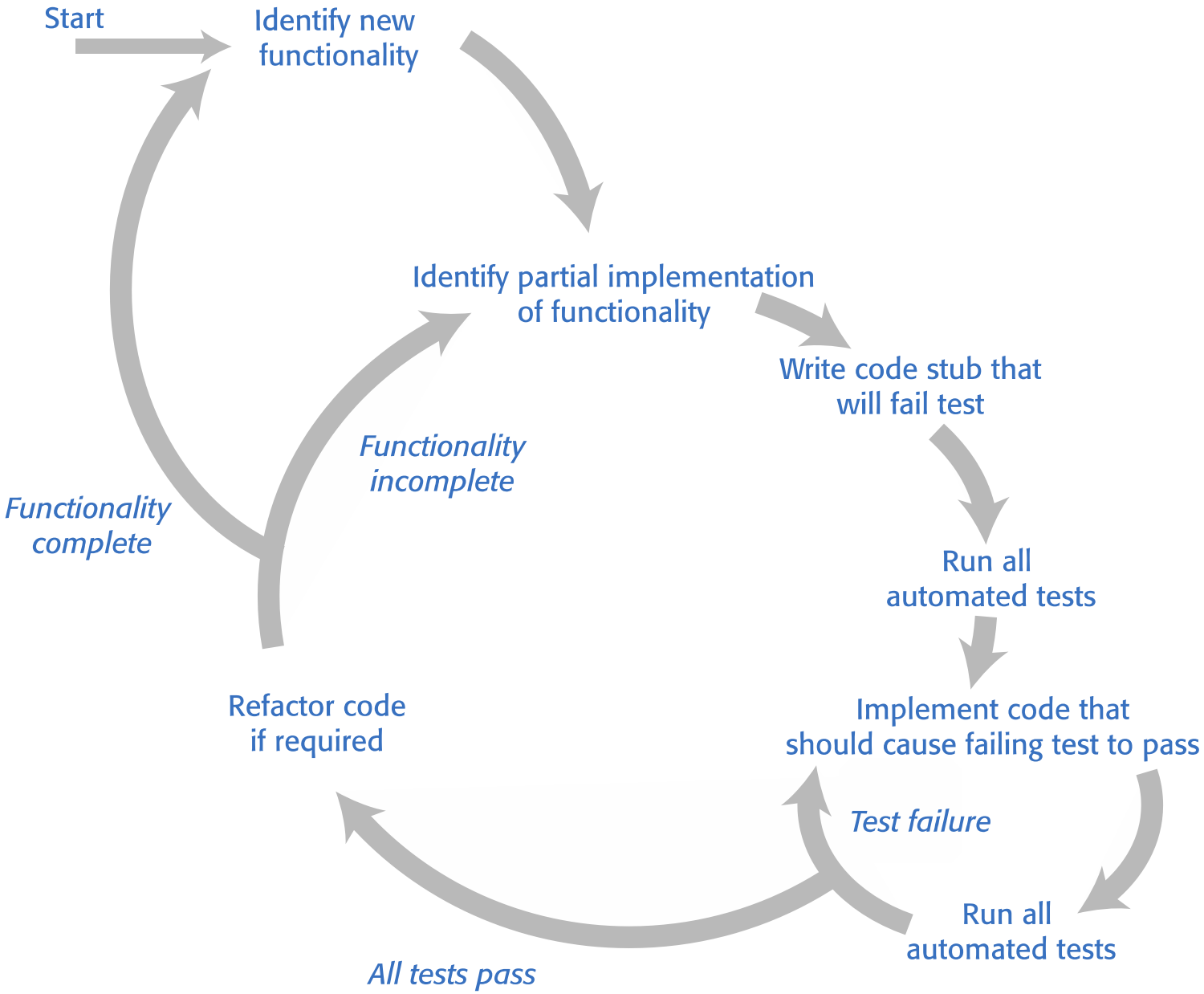
One is different
If your program deals with sequences, always test with sequences that have a single value.

```
def namecheck(s):
  # checks that a name only includes
  # alphabetic characters, -, or single quote
  # names must be between 2 and 40
  #   characters long
  # quoted strings and -- are disallowed

  namex = r"^[a-zA-Z][a-zA-Z-']{1,39}$"
  if re.match (namex, s):
    if (re.search ("'.*'", s) or
        re.search ("--", s)):
      return False
    else:
      return True
  return False
```

Test Driven Development (TDD)

Test Driven Development

- Software Requirements are converted into test cases, before the software is fully developed.
- Test cases are designed to be run repeatedly as development progresses. It can be a useful measure of how well the requirements are fulfilled.
- Kent Beck is credited with having ‘rediscovered’ this process after promoting a ‘test-first’ concept in Extreme programming (1999) – where pair programming comes from.
- In 2003, Kent stated that “TDD encourages simple designs and inspires confidence”



Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering

Test Driven Development (TDD) Cycle

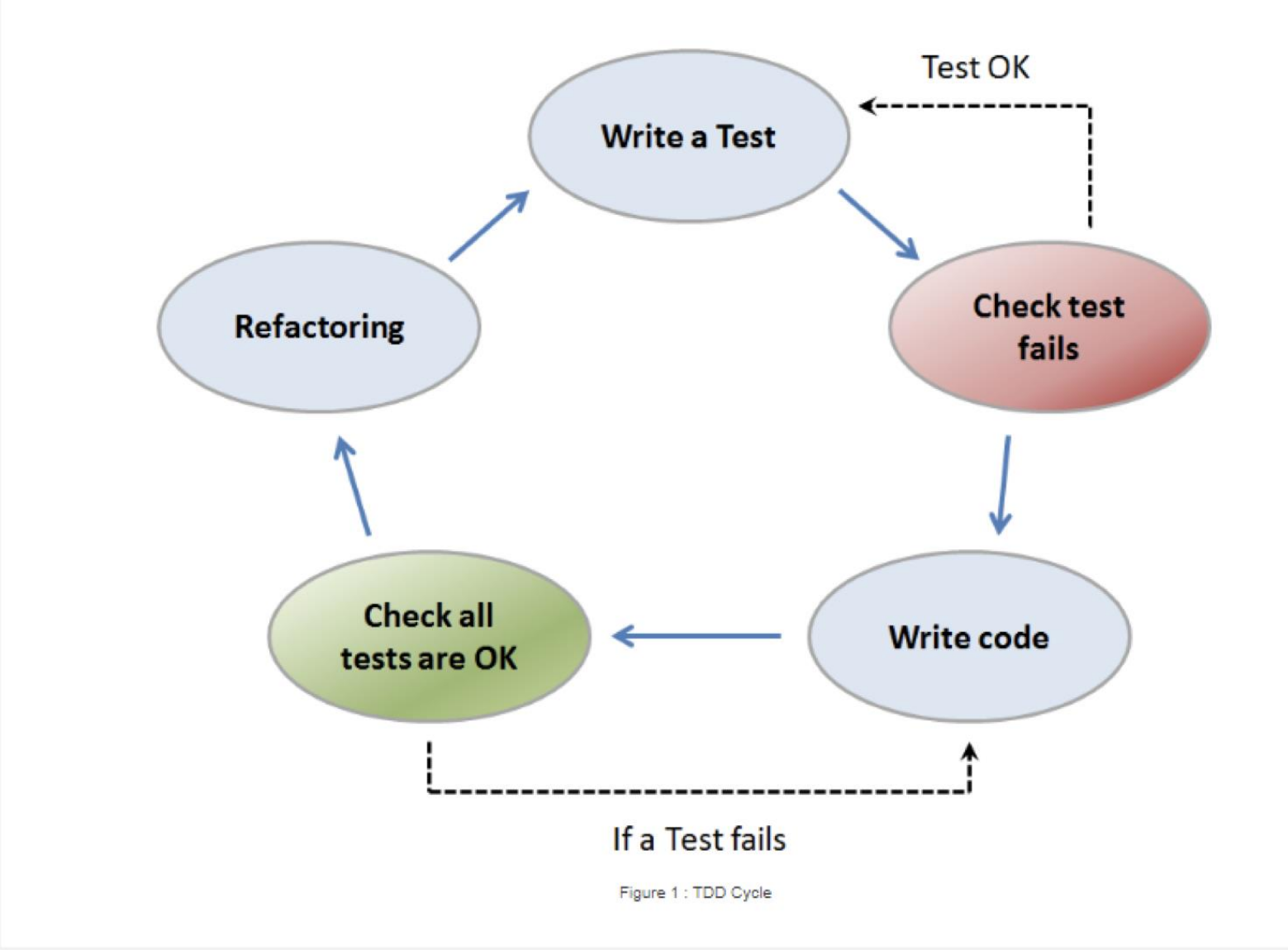


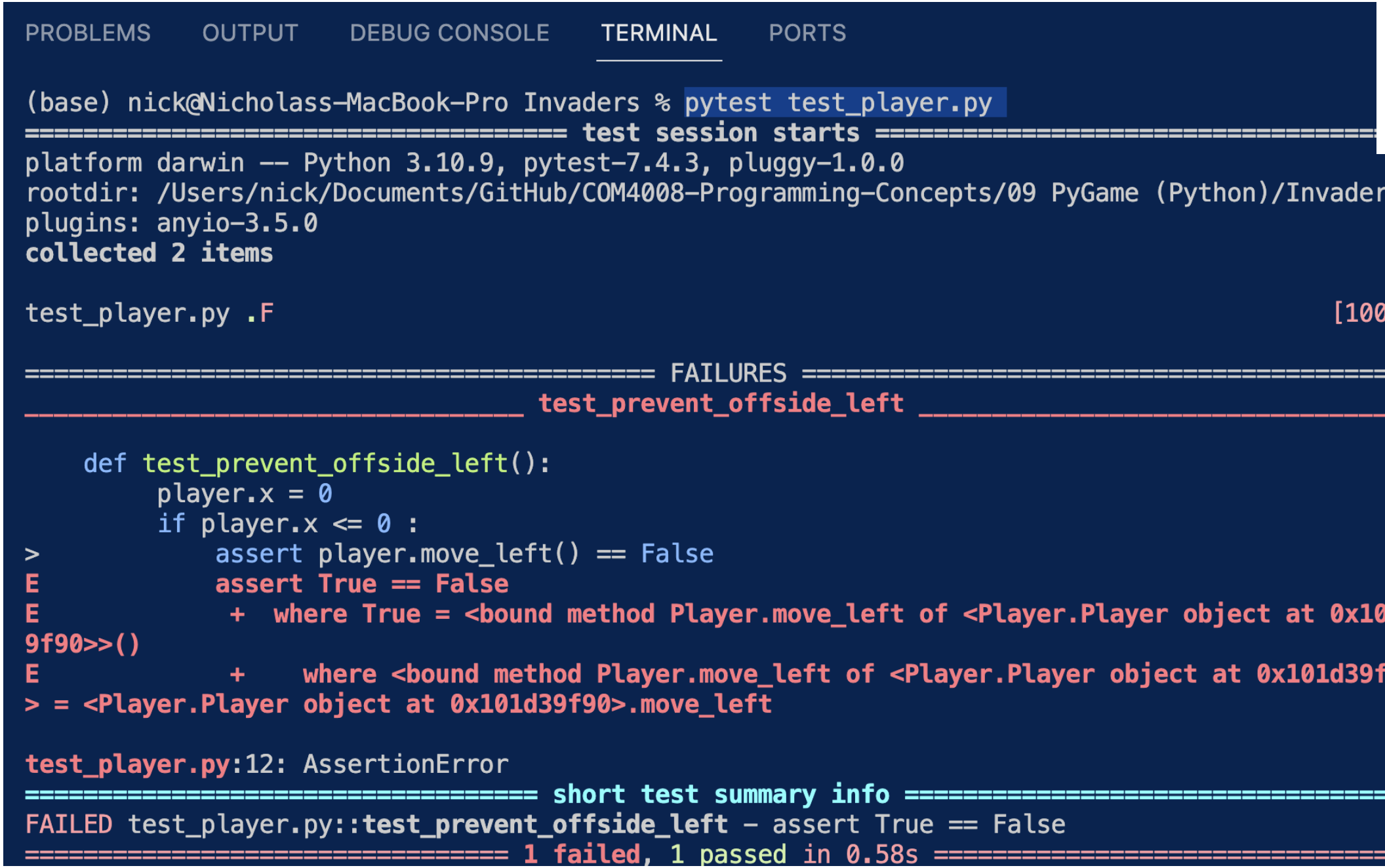
Figure 1 : TDD Cycle

Unit Test Frameworks

Pytest

- Pytest is an open-source package available at: https://docs.pytest.org/en/7.4.x/index.html
- It works on functions that are set up to ‘test’ a behaviour by the outcome of an ‘assertion’.
- This assertion tests to see if a condition is True. If the assertion is True, then the test has passed, if not, then the test fails.

```
def add(x):
    return x + 1

def test_answer():
    assert add(3) == 5
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
(base) nick@Nicholass-MacBook-Pro Invaders % pytest test_player.py
test session starts
platform darwin -- Python 3.10.9, pytest-7.4.3, pluggy-1.0.0
rootdir: /Users/nick/Documents/GitHub/COM4008-Programming-Concepts/09 PyGame (Python)/Invaders
plugins: anyio-3.5.0
collected 2 items
test_player.py .F [100%]
FAILURES
test_prevent_offside_left
def test_prevent_offside_left():
player.x = 0
if player.x <= 0 :
> assert player.move_left() == False
E assert True == False
E + where True = <bound method Player.move_left of <Player.Player object at 0x101d39f90>>()
E + where <bound method Player.move_left of <Player.Player object at 0x101d39f90> = <Player.Player object at 0x101d39f90>.move_left
test_player.py:12: AssertionError
short test summary info
FAILED test_player.py::test_prevent_offside_left - assert True == False
1 failed, 1 passed in 0.58s
pytest

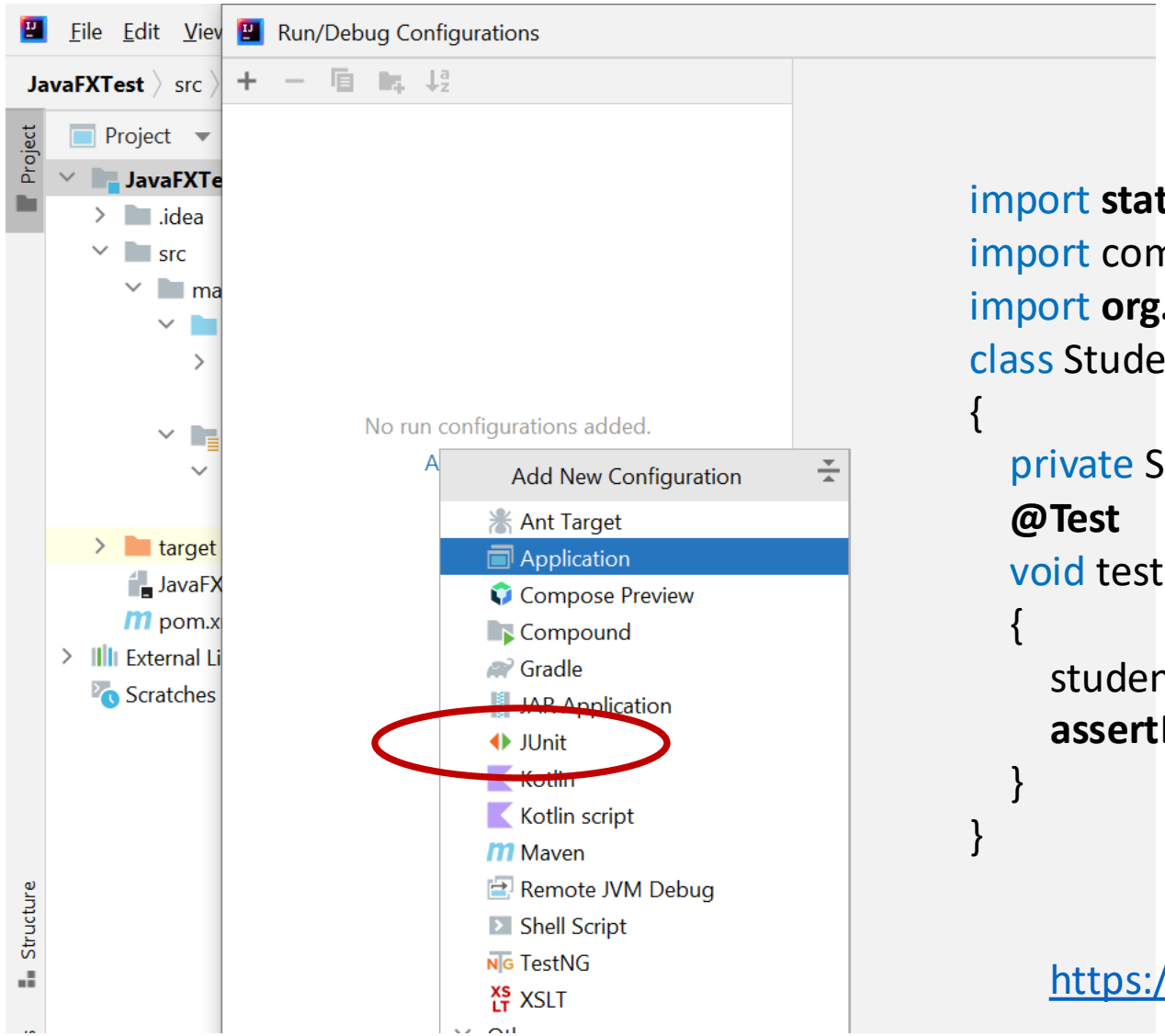
PyUnit

- PyUnit, by contrast is Python's built in Unit Testing framework
- Similar to PyTest, It works on functions that are set up to 'test' a behaviour by the outcome of an 'assertion'.

Toggle line numbers

```
1  import unittest
2  from foobarbaz import Foo # code from module you're testing
3
4
5  class SimpleTestCase(unittest.TestCase):
6
7      def setUp(self):
8          """Call before every test case."""
9          self.foo = Foo()
10         self.file = open( "blah", "r" )
11
12     def tearDown(self):
13         """Call after every test case."""
14         self.file.close()
15
16     def testA(self):
17         """Test case A. note that all test method names must begin with 'test.
18         assert foo.bar() == 543, "bar() not calculating values correctly"
19
20     def testB(self):
21         """test case B"""
22         assert foo+foo == 34, "can't add Foo instances"
23
24     def testC(self):
25         """test case C"""
26         assert foo.baz() == "blah", "baz() not returning blah correctly"
27
```

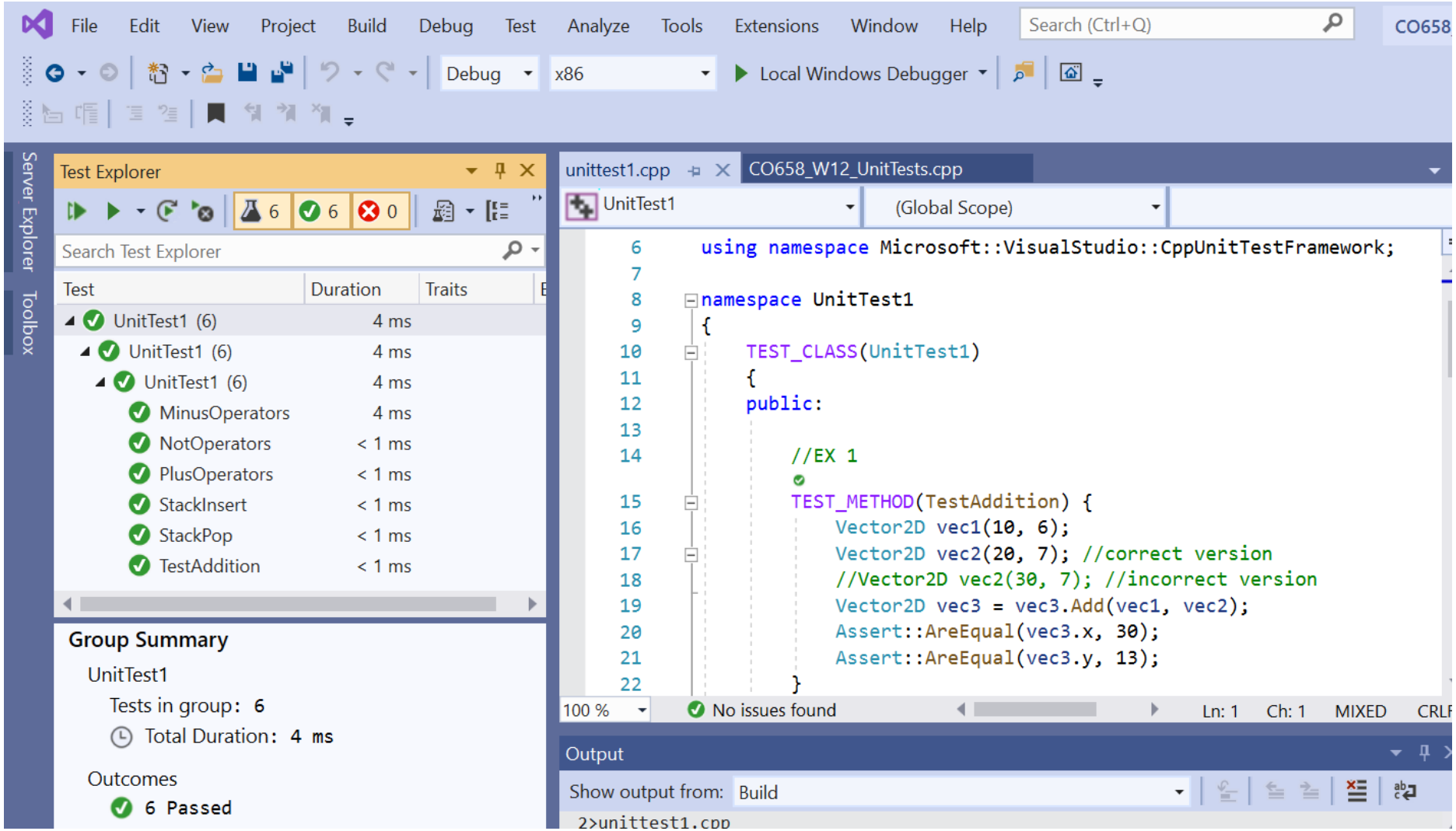
JUnit5 (Java's Unit Testing)




```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import com.company.Student;  //Class location
import org.junit.jupiter.api.Test;
class StudentTests
{
   private Student student = new Student();
   @Test                //annotation
   void testSetID()
   {
      student.setID(1234);    //define a value to test
      assertEquals(1234, student.getID());   // compare expected vs actual
   }
}
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests

Visual Studio 2022's Test Explorer



C++ Unit Testing via GTest

- Supports a wide range of C compilers – GCC and Clang
- Uses assertions (similar to what we've seen in Python)
- Can also support integration testing too.

MySampleApp.h | MySampleApp.cpp

MySampleApp | (Global S

```
70  TEST_F(TestBase, testAddition) {
71      int res = 0;
72      int res_expected = 30;
73      res = pSC->testAdd(10,20);
74      ASSERT_EQ(res, res_expected);
75
76
77      res = pSC->testAdd(100, 200);
78      ASSERT_EQ(res, 300);
79  }
80  TEST_F(TestBase, testMultiplication) {
81      int res = 0;
82      int res_expected = 200;
83      res = pSC->testMul(10, 20);
84      ASSERT_EQ(res, res_expected);
85
86      res = pSC->testMul(100, 200);
87      ASSERT_EQ(res, 20000);
88  }
```

https://google.github.io/googletest/