GCSE Computer Science Knowledge Organiser
SLR 2.3 Producing Robust Programs :
Defensive Designs Considerations

Key Terminology	BCS Definition
Defensive design	"The practice of planning for contingencies in the design stage of a project."
Anticipating misuse	"Ensuring data input by a user meets specific criteria before processing. Range check (e.g., 1 – 31); type check (e.g., a number, not a symbol); presence check (e.g., data has been input); format check (e.g., a postcode is written LLN(N) NLL)."
Authentication	"Techniques and methods that make code easier to debug, update and maintain."
Input validation	"Many programmers use defined naming conventions for variables, contents and procedures. Camel case is a popular one used in the industry where the first word of an identifier uses all lower case and all subsequent words start with a capital letter – e.g., studentsFirstName."

Input validation

Input validation means checking that data input by the user meets specific criteria/rules before processing:

- **Type check:** The input is of the correct data type – e.g., integer, real, string.
- **Range check:** The input is within a predetermined range – e.g., between 1 and 2.
- **Presence check:** All required data has been entered – e.g., reject blank inputs.
- **Format check:** The input is in the correct format – e.g., DD/MM/YYYY.
- **Length check:** The input includes the correct/minimum/maximum number of characters – e.g., password.

By using input validation techniques, a programmer can:

- Make their program more robust and user-friendly
- Prevent further errors occurring later in the algorithm

Division by zero

In mathematics, there is no number that, when multiplied by zero, returns a number that is not zero.

Therefore, an arithmetic logic unit cannot compute a division by zero – for example:

```
num = 0
average = total / num
```

This line of code, while syntactically correct, could potentially cause a program to crash.

A programmer should always check that a variable is not zero before attempting to use it in division – for example:

```
IF num != 0:
average = total / num
ELSE:
print("No scores entered")
```

Printer and other peripheral errors

If a program outputs a document to a printer, it may run out of ink or experience a paper jam. The programmer should not assume that an output to a printer will always be successful and always include options to reprint documents.

Communication error

Online systems require connections to host servers.
If this connection is dropped or cannot be established or the server becomes overloaded, a program may crash or hang when loading or saving data.

A programmer should enable ways for the user to cancel requests or for them to fail gracefully, reporting the connection error. The program may be able to automatically resume when the connection becomes available again.

Disk Errors

Programs that read and write to files need to handle many types of exceptions, including:

- File/folder not found
- Insufficient disk space
- Data corruption
- End of file reached

Robust programs will handle all these situations by checking files and data before attempting to use them for further processing.

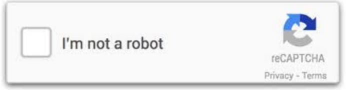
Authentication

Data used by systems should be secure – this can be achieved with:

- Username and password access.
- Password recovery by sending an email with a verification link to the registered email address.
- Data file encryption.

Online bots can automatically submit data via online forms – this can be protected against using software such as reCAPTCHA, which verifies that the user is human.

Programmers should also be aware of SQL injection hacks and other methods used by cybercriminals.



GCSE Computer Science Knowledge Organiser
SLR 2.3 Producing Robust Programs :
Maintainability

Key Terminology	BCS Definition
Maintainability	"The practice of planning for contingencies in the design stage of a project."
Naming conventions	"Many programmers use defined naming conventions for variables, contents and procedures. Camel case is a popular one used in the industry where the first word of an identifier uses all lower case and all subsequent words start with a capital letter – e.g., studentsFirstName."
Indentation	"Makes it easier to see where structures begin and end. Conditions, iterations and code inside procedures and functions should be indented." is known before the loop begins executing."
Commenting	"Used to explains sections of code. Ignored by the compiler."

Greatest common factor program.py

File Edit Format Run Options Window Help

```
#------------------------------------------------------------------------------

def gcf_of(factors1,factors2):
#Finds the greatest common factor (gcf) in two input lists
    index = 0
    #Check all the numbers in the factors1 list until the same number is found i
    #Needs the lists to be in numerical order
    while factors1[index] not in factors2:
        index = index + 1
    #Return the highest number found in both lists
    return factors1[index]

#------------------------------------------------------------------------------

def factors_of(number):
#Returns a list of all the factors for a number
    factors = []
    #Check all numbers from the number input down to 0
    for countdown in range(number,0,-1):
        #If the number divided by the count down has no remainder...
        if number % countdown == 0:
            #...it is a factor and is added to the list
            factors.append(countdown)
    return factors
```

Using indentation.

Greatest common factor program.py

File Edit Format Run Options Window Help

```
#------------------------------------------------------------------------------

def gcf_of(factors1,factors2):
#Finds the greatest common factor (gcf) in two input lists
    index = 0
    #Check all the numbers in the factors1 list until the same number is found i
    #Needs the lists to be in numerical order
    while factors1[index] not in factors2:
        index = index + 1
    #Return the highest number found in both lists
    return factors1[index]

#------------------------------------------------------------------------------

def factors_of(number):
#Returns a list of all the factors for a number
    factors = []
    #Check all numbers from the number input down to 0
    for countdown in range(number,0,-1):
        #If the number divided by the count down has no remainder...
        if number % countdown == 0:
            #...it is a factor and is added to the list
            factors.append(countdown)
    return factors
```

Using sensible, descriptive identifier names.

Greatest common factor program.py

File Edit Format Run Options Window Help

```
#------------------------------------------------------------------------------

def gcf_of(factors1,factors2):
#Finds the greatest common factor (gcf) in two input lists
    index = 0
    #Check all the numbers in the factors1 list until the same number is found i
    #Needs the lists to be in numerical order
    while factors1[index] not in factors2:
        index = index + 1
    #Return the highest number found in both lists
    return factors1[index]

#------------------------------------------------------------------------------

def factors_of(number):
#Returns a list of all the factors for a number
    factors = []
    #Check all numbers from the number input down to 0
    for countdown in range(number,0,-1):
        #If the number divided by the count down has no remainder...
        if number % countdown == 0:
            #...it is a factor and is added to the list
            factors.append(countdown)
    return factors
```

Using comments to divide the program into distinct sections.

Greatest common factor program.py

File Edit Format Run Options Window Help

```
#------------------------------------------------------------------------------

def gcf_of(factors1,factors2):
#Finds the greatest common factor (gcf) in two input lists
    index = 0
    #Check all the numbers in the factors1 list until the same number is found i
    #Needs the lists to be in numerical order
    while factors1[index] not in factors2:
        index = index + 1
    #Return the highest number found in both lists
    return factors1[index]

#------------------------------------------------------------------------------

def factors_of(number):
#Returns a list of all the factors for a number
    factors = []
    #Check all numbers from the number input down to 0
    for countdown in range(number,0,-1):
        #If the number divided by the count down has no remainder...
        if number % countdown == 0:
            #...it is a factor and is added to the list
            factors.append(countdown)
    return factors
```

Using comments to explain what various parts of the program are designed to do.

Writing maintainable code

To make your code as easy to read and maintain as possible, make sure you use:

- Comments to divide the program into sections and explain:
  - The program's purpose.
  - Sections of code – typically, selections, iterations and procedures.
  - Any unusual approaches taken.
- White space to make program sections easier to see.
- Indentation for every selection and iteration branch.
- Descriptive variable names, explaining each variable's purpose with a comment when it is declared.
- Procedures and/or functions to:
  - Structure code.
  - Eliminate duplicate code.
- Constants – declared at the top of the program.

GCSE Computer Science Knowledge Organiser
SLR 2.3 Producing Robust Programs:
The Purpose and Types of Testing & Suitable Test Data

Key Terminology	BCS Definition
Testing	“Assessing the performance and functionality of a program under various conditions to make sure it works. Programmers need to consider all the devices the program could be used on and what might cause it to crash.”
Iterative testing	“Each module of a program is tested as it is developed.”
Final/terminal testing	“Checking that all the modules of a program work together as expected and the program meets the expectations of users with real data.”
Test data	“Values used to test a program – normal, boundary and erroneous.”
Test data: Normal	“Data supplied to a program that is expected. Using a program written to average student test scores as an example, if allowed scores are 0 – 100, normal test data would include all the numbers within that range.”
Test data: Boundary	“Data supplied to a program designed to test the boundaries of a problem. Using a program written to average student test scores as an example, if allowed scores are 0 – 100, boundary test data could be -1, 0, 1, 99, 100 and 101.”
Test data: Invalid	“Data of the correct type but outside accepted validation limits. Using a program written to average student test scores as an example, if allowed scores are 0 – 100, invalid test data could be -5, 150, etc.”
Test data: Erroneous	“Data of the incorrect type that should be rejected. Using a program written to average student test scores as an example, if allowed scores are 0 – 100, erroneous data might be the string “hello”, the real number 3.725, etc.”

Reasons for testing

Four main reasons why a program should be tested include:

- To ensure there are no errors (bugs) in the code.
- To check that the program has an acceptable performance and usability.
- To ensure that unauthorised access is prevented.
- To check the program meets the requirements.

errors.py

File Edit Format Run Options Window Help

```python
print("1. New game")
print("2. Save game")
print("3. Play game")

choice = 0

while choice < 1 or choice > 3:
    choice = int(input("Enter choice: "))
```

Ln: 9 Col: 0

No.	Type of test	Input	Expected output
1	No data		Reject input
2	Erroneous data	j	Reject input
3	Erroneous data	#	Reject input
4	Invalid data	-6	Reject input
5	Invalid data	8	Reject input
6	Invalid data	2.5	Reject input
7	Normal data	2	Accept input
8	Boundary data	1	Accept input
9	Boundary data	3	Accept input

Types of testing

Iterative testing:

- Each new module is tested as it is written.
- Program branches are checked for functionality.
- Checking new modules do not introduce new errors in existing code.
- Tests to ensure the program handles erroneous data and exceptional situations.

Performed whilst the software is being developed

Final / Terminal testing:

- Testing that all modules work together (integration testing)
- Testing the program produces the required results with normal, boundary, invalid and erroneous data.
- Checking the program meets the requirements with real data.
- A beta test may find more errors.

Performed when the program is finished

Test data needs to include a range of:

- **Normal inputs:** Data that should be accepted without causing errors.
- **Erroneous inputs:** Data that should be rejected by the program – includes no input when one is expected.
- **Boundary inputs:** Data of the correct type that is on either edge of the accepted validation limits.
- **Invalid inputs:** Data of the correct type but outside accepted validation limits.

GCSE Computer Science Knowledge Organiser
SLR 2.3 Producing Robust Programs :
How to identify syntax and logic errors & Refining algorithms to make them more robust

Key Terminology	BCS Definition
Syntax error	"Rules of the language have been broken, so the program will not run. Variables not being declared before use. Incompatible variable types (e.g., sum = A); using assignments incorrectly (e.g., 2 + 2 = x); keywords misspelt (e.g., PRNT("Hello"))."
Logical error	"The program runs but does not give the expected output. Division by zero. Infinite loop. Memory full. File not found."

Syntax and logic errors

Syntax error:

The rules of the language have been broken, and the program will not run (compiled languages). Syntax errors can occur for the following reasons:

- Variables not declared or initialised before use.
- Incompatibility of variable types – e.g., total = "A" (total declared as an integer)
- Using assignments incorrectly – e.g., 2 + 2 = x
- Misspelt keywords – e.g., prnt("Enter choice: ")

Logic error:

The program runs but does not produce the expected output. Logic errors can occur for the following reasons:

- Conditions and arithmetic operations are wrong.
- Sequence of commands is wrong.
- Division by zero.
- Exceptions – e.g., file not found.

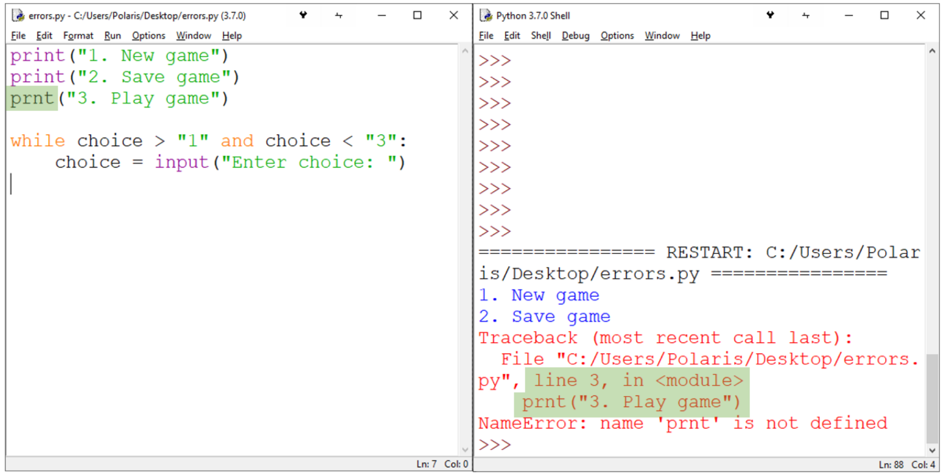
Refining algorithms to make them more robust means:

- Writing code that anticipates a range of possible inputs, which may include invalid or erroneous data.
- Making sure invalid data inputs don't crash the program.
- Ensuring prompts to the user are descriptive and helpful.
- Checking for errors and handling instances of no input.

A common solution is to use the simple exception-handling commands available in most programming languages.

Example of Syntax error

Misspelt keyword (prnt instead of print)



Example of Logic Error

Condition is wrong - > and < are the wrong way round