GCSE Computer Science Topic 2.3 Robust Programs

Why defensive design?
Helps to ensure programs function properly.
✓ Not breaking
✓ Not producing errors

3 elements of Defensive design:
- Anticipate how users might '<u>**misuse**</u>' their program to prevent it from happening.
- Ensure their code is <u>**well maintained**</u>.
- Reduce the numbers of errors in the code through <u>**testing**</u>.

Planning for contingencies / anticipating misuse
- Computer programs should be designed to COPE with unexpected or erroneous input from users.
- Coders should PLAN for all contingencies that might occur. (accidental and deliberate inputs)

Input validation: Validation checks that data input is sensible, reasonable and appropriate to be processed by the program.

Presence check: Checks that data has actually been entered and the field has not been left blank..

Length check: Checks that a specified number of characters has been entered.

Range check : Checks that the input falls within a certain range. e.g. 1-100

Type check : This checks that the data inputted is a certain data-type e.g. number or letters.

Format check :
Checks that the input is in the correct format e.g.
National insurance number XX999999X

Input sanitisation : Removes any unwanted characters BEFORE passing the data to the program.

Authentication is determining the identity of the user before they can access the program or parts of the program.
This is usually based upon a username and associated password.
TOO MUCH AUTHENTICATION CAN:
- *Affect the functionality of the system.*
- *Can put people off using it.*

Maintainability:
Keeping the code well maintained aids defensive design as it means when editing, improving or testing the code – it is clear and easy to understand what the code should be doing.

Commenting:
#Usually written with **//** or **#**
#Comments are useful for explaining what key features of a program do.
#Well written/clear comments are essential in allowing other programmers to understand your program.

Tab

Indentation :
This is used to separate different statements in a program. This allows other programmers to see the flow of a program more clearly and pick out the different features.
Indentation is usually used to show which statements are part of a previous line of code.
E.g. with ***selection*** and ***iteration***.

Test Plan	A test plan will outline exactly what you're going to test and how you are going to test it. It should cover all the possible paths through a program.
Normal data	Data that the user is LIKELY to input into the program. Data that the program should be able to process.
Extreme / Boundary data	Values at the limit of what the program should be able to handle. This data should still be able to be processed by the program.
Erroneous data	Data that the program should not accept; usually the wrong data type.

Naming Variables:
Variables should be named so that they reflect their purpose.
This helps other programmers keep track and recognise what the variables are when reading /using the program.

- **Testing** ensures that the software produces the expected results and meets the needs of the user.
- Testing makes sure the program is robust.
- Testing should be destructive and should try to find errors rather than just proving the program works.

ITERATIVE TESTING: Tests carried out whilst the program is being developed. The test results are then used to guide further improvements.

FINAL TESTING: This is carried out once the software has been developed.
Alpha testing is done by the developers.
Beta testing is carried out by the potential users of the software.

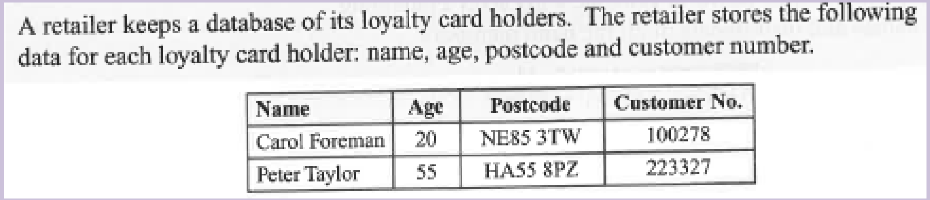
A **syntax error** occurs when the compiler or interpreter doesn't understand something the user has typed because it doesn't follow the rules or grammar of the programming language.
Syntax errors produce a error message which details what is wrong and which line of code contains the error.

Logical errors: The interpreter / compiler will be table to run the code, but the program will do something unexpected.
E.g. using the wrong Boolean operator.
Logical errors are difficult to diagnose / track down.
Logical errors can only be found through testing, using a test plan.

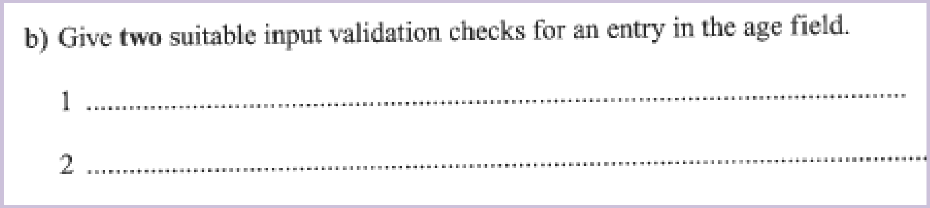
GCSE Computer Science - Topic 2.3 Robust Programs

What I need to know:

Explain the programmers defensively design programs.
State the 3 elements of defensive design.
Explain what planning for contingencies involves.
Describe input validation.
State the function of a presence check.
State the function of a length check.
State the function of a range check.
State the function of a type check.
State the function of a format check.
Describe input sanitisation.
Define authentication.
Explain what is meant by maintainability.
Describe how commenting helps improve maintainability.
Describe how indentation helps improve maintainability.
Describe how variable names help improve maintainability.
Explain why programs are tested.
Describe iterative testing.
Describe final testing.
State what is meant by a syntax error. Give an example.
State what is meant by a logical error. Give an example.
Describe what is meant by a test plan.
What are the three types of data a program should be tested with?
Define normal, extreme and erroneous data.

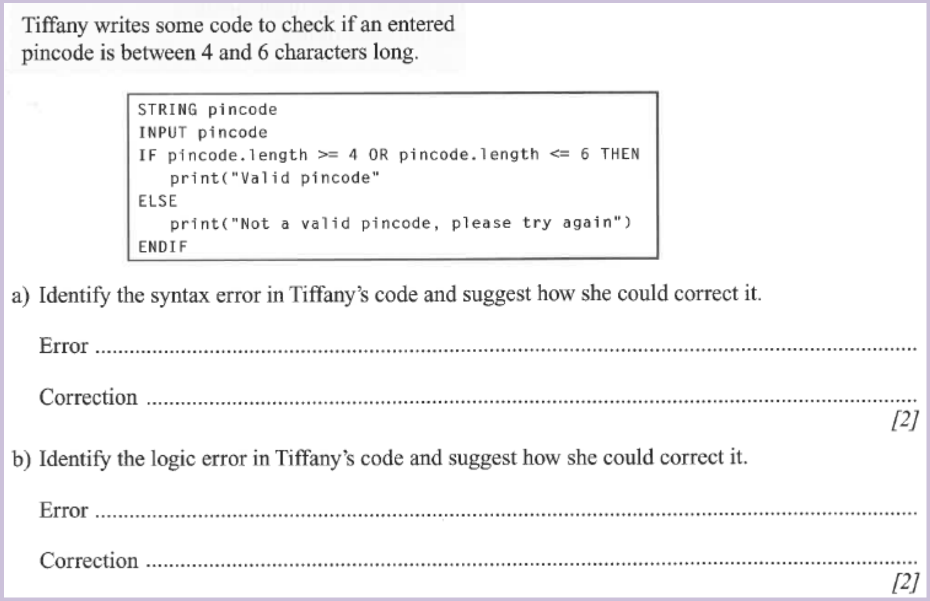

A retailer keeps a database of its loyalty card holders. The retailer stores the following data for each loyalty card holder: name, age, postcode and customer number.

Name	Age	Postcode	Customer No.
Carol Foreman	20	NE85 3TW	100278
Peter Taylor	55	HA55 8PZ	223327


b) Give **two** suitable input validation checks for an entry in the age field.

1 ....

2 ....


Tiffany writes some code to check if an entered pincode is between 4 and 6 characters long.

```
STRING pincode
INPUT pincode
IF pincode.length >= 4 OR pincode.length <= 6 THEN
   print("Valid pincode")
ELSE
   print("Not a valid pincode, please try again")
ENDIF
```

a) Identify the syntax error in Tiffany's code and suggest how she could correct it.

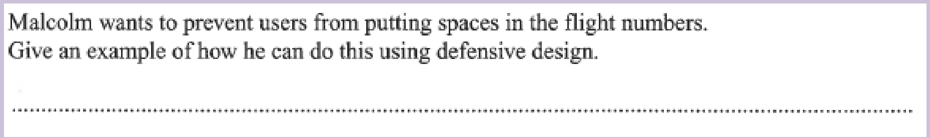
Error ....

Correction .... *[2]*

b) Identify the logic error in Tiffany's code and suggest how she could correct it.

Error ....

Correction .... *[2]*


Malcolm wants to prevent users from putting spaces in the flight numbers.
Give an example of how he can do this using defensive design.

....