Teaching guide: Structured programming

This resource will help with understanding structured programming. It supports Section 3.2.11 of our current specification (8520). The guide is designed to address the following learning aims:

- Interpret a problem as a sequence of sub-problems.
- Organise programs into subroutines that can be combined to solve the complete problem.
- Be able to evaluate the usefulness of adopting a structured approach to problem-solving and development.

When students begin learning to program they often write small programs to become familiar with the syntax of a language (the keywords and structures that it uses) whilst also keeping the logic of the program focused and brief.

The next step in programming is to use these small-step techniques to solve larger problems, the skills needed to find, say, the largest element in an array are very different from the skills needed to take a problem such as those required for the GCSE non-examined assessment (NEA). Problem-solving such as this requires abstraction and decomposition:

- Abstraction is the art of taking a real-world problem and recognising the important from the unimportant, or at least the necessary from the unnecessary.
- Decomposition is breaking a problem down into manageable parts. If these parts are still too big to be solved immediately then they too should be further broken down.

There are many different techniques that can be used develop solutions to complex problems, we will look at a few that can be used to make a challenging problem simpler to solve.

Analysing the problem

Take the following example:

> *"A gardener (specialising in laying turf) is used to having to calculate quotes by hand and wants a bespoke solution for her company. The solution will need to hold client details including their first and last names and their address along with the quote that has been offered to them (clients only get one quote). The*

gardener should be able to search for clients by entering their surname.

Quotes are created based on the length and width of the area to be turfed and also the quality of turf that the customer wants. The type and the cost of the turf that can be used can change so the gardener wants an easy way to update this information. The gardener works on the assumption that the area to be turfed is rectangular and so the total cost is given by the total area x cost of the turf. The quote is the total cost multiplied by 1.5 to account for the labour of laying the turf and to give the gardener a profit."

In order to solve this entire problem and give the gardener a program that will do what she wants we first need to analyse the problem and highlight the important details.

- Client details need to be persistently stored.
- Quotes are associated with clients.
- Clients need to be searched for by surname.
- It should be possible to update turf costs.
- Quotes are generated by width * length * turf cost * 1.5.

Next, we'll take a look at the data involved in this problem.

- Client details are made up of:
    - Last name (a string).
    - First name (a string).
    - Address, which we can simplify to:
        - First line (a string)
        - Town (a string)
        - County (a string)
        - Postcode (a string).
    - Quotes are in £s (a real) and are associated with clients.
- Turf costs have two parts:
    - type of turf (a string)
    - £ per square metre (a real).

Now we can start to think about the data structures we could use:

- All of the client details need to be stored and searched for by last name.
- Turf types need to be stored and selected by type of turf.

And lastly, what data needs to be stored permanently:

- Client details need persistent storage.
- Turf types and costs need persistent storage.

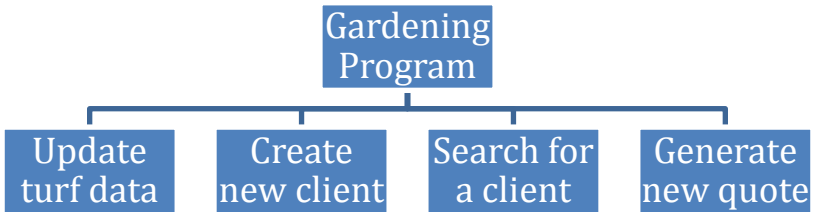
This is a very brief analysis but we've managed to:

1. Summarise the key elements of the problem.
2. Outline the data involved.
3. Identify how related data needs to be ordered (data structures).
4. Identify persistent data.

We've not specified how the problem is going to be solved at this point, but we have said what it should do and the data it needs to do it.

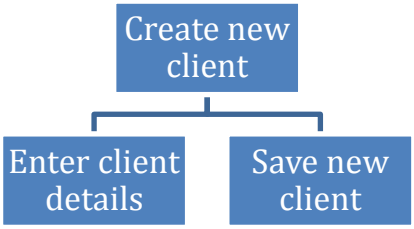
Designing the solution (high-level)

Now we have analysed the requirements of the problem and the data we can start to think about the different 'operations' that are needed to solve this problem. This is the part where we decompose the problem into manageable modules, at this stage it helps to differentiate the 'logic' of the program from the parts that deal with input and output. A first, high-level attempt at this could be:

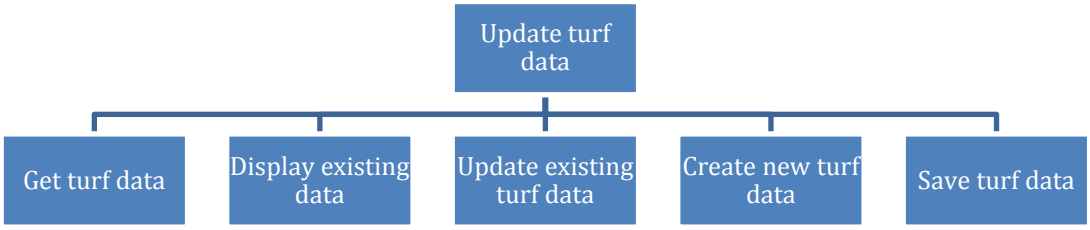


This is a very high-level view of the program, it has abstracted away much of the detail and broken it down into four main modules. When the program is run, the gardener will need to select which of these four main modules they wish to do, so an amended design should also include an introductory menu that, although not specifically stated, should presumably be displayed again once any of the four modules has been completed.

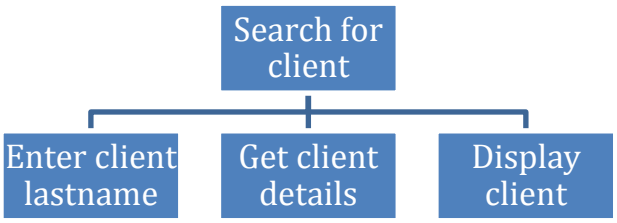
Creating a new client has two sub-modules: entering the details and then saving the new client.

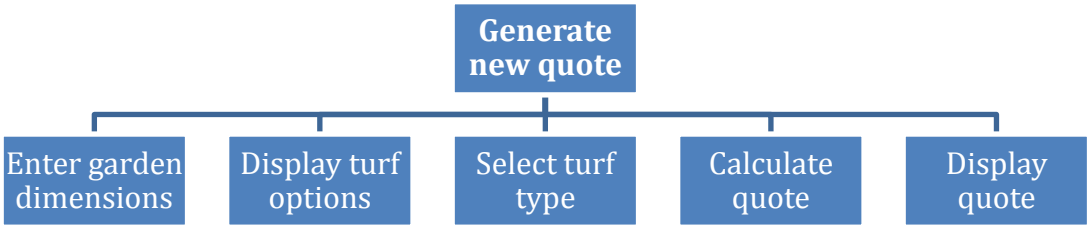


Updating turf data, searching for a client and generating a new quote can each be broken down into more stages:



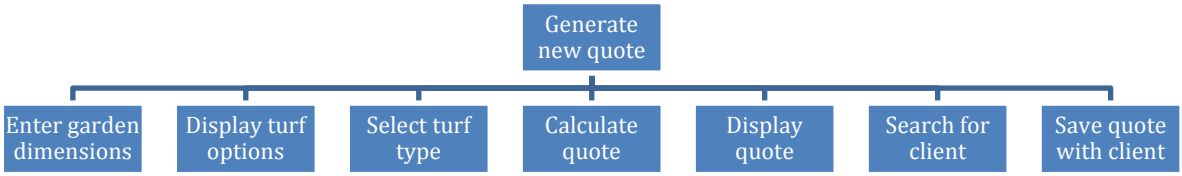
For the next module on searching for a client we will make the assumption that every client has a different last name, this is obviously incorrect for a real-world application but the simplification helps in this instance. An extension to this design would be to associate a unique reference with every client (called a primary key in relational databases) that enables us to select a specific client even if there were duplicate last names.





At this stage we have a design choice to make: as quotes are going to be attached to clients, should we select a client first and then create their quote, or

should the quote be generated and then associated with a client? If this was a real-world problem then the gardener should be involved in this decision but we'll assume that the client's details have already been entered into our program and so once a quote is displayed we can search for a client and finally append the quote to the client details. Generate new quote now looks like this:



Designing the solution (interface)

Now we have the high-level plan broken down into manageable modules we can concentrate on the data that must be passed between those modules in order for them to do their job. As much as possible, we want all of the modules of our program to exist in isolation. That is, they all accept inputs and they may or may not return a value once the processing has finished. This isolation not only keeps our program neat, it also makes these individual modules very easy to test and, if necessary, change (as long as the input and return value are kept the same). These modules can then be turned into subroutines.

Firstly, let's analyse the inputs that all of the modules in the Search for Client section take:

Module	Input(s)
Enter client lastname	(None)
Get all relevant details	Lastname of client, all clients
Display clients	Collection of clients
Get user choice of client	Number of different choice
Display selected client	User choice

The inputs in this table are those needed for the modules to be able to complete their purpose. In order for a module to have an input, some modules may have to return certain values. This table shows the return values needed (additionally the number of different choices is reworded as the calculated length of the number of clients):

Module	Input(s)	Return Value
Enter client lastname		Lastname of client
Get all relevant details	Lastname of client, all clients	Collection of clients
Display clients	Collection of clients	
Get user choice of client	Length of collection of clients	User choice
Display selected client	User choice	

These tables might make more sense if we include another column with the processing that will take place within each one:

Module	Input(s)	Processing	Return Value
Enter client lastname		Prompt the user to enter the last name of a client	Lastname of client
Get all relevant details	Lastname of client, all clients	Filter all of the clients that have the same last name as the input	Collection of clients
Display clients	Collection of clients	Display the clients (all data) from the collection given	
Get user choice of client	Length of collection of clients	Get a numerical value from the user between 1 and number input	User choice
Display selected client	User choice	Display the client given as input	

We can view this data in a graphical way that combines all of the module information used so far with a diagram called a structure (hierarchy) chart where the data that is passed to, and returned from, the modules are represented as arrows (arrows going in to a module are inputs and an arrow coming out is a return value).

We have created a series of contracts here, every module promises to take a certain number of inputs of a certain type and either return nothing or return a value. This, combined with the name of the module itself, constitutes the interface of that module.

Designing solutions using interfaces means that modules can be worked on and developed in isolation from each other or even simultaneously by different developers. These modules can also be tested individually to check that they work (it is far easier to spot errors in your code when they are tested this way). Additionally, the way the modules process their data may also be changed at a later date without affecting the functionality of the system as a whole as long as the interface is preserved.

At this stage it is probably useful to pay some thought to the data and data structures in use in our program: namely clients and turf. Both of these are heterogeneous data structures and so both would lend themselves (using the techniques we have looked at in other teacher resources) to records:

```
RECORD Client
   last_name
   first_name
   address_first_line
   address_town
   address_county
   address_postcode
   quote
ENDRECORD

RECORD Turf
   turf_type
   cost_per_sq_m
ENDRECORD
```

We also require collections of these records which, in the case of the collection of clients, needs to be accessed using the `last_name` field. There are more specialist data structures such as associative arrays and hash tables that would be suitable for this purpose but we will use arrays as we have looked at these in other teacher resource modules.

We can now develop the skeleton code for our program. Skeleton code, like the name suggests, contains just the scaffolding and some core programming constructs – this takes the form of subroutine definitions with input variables and 'dummy' return values but without any of the processing.

```
# main subroutine
SUBROUTINE gardening()
   WHILE True
      display_main_menu()
      choice ← get_user_choice(5)
      IF choice = 1 THEN
```

```
            update_turf_data()
        ELSE IF choice = 2 THEN
            create_new_client()
        ELSE IF choice = 3 THEN
            search_for_client()
        ELSE IF choice = 4 THEN
            generate_new_quote()
        ELSE IF choice = 5 THEN
            QUIT() # subroutine that will exit the program
    ENDWHILE
ENDSUBROUTINE

SUBROUTINE update_turf_data()
    turf_data ← get_turf_data()
    display_turf_data(turf_data)
    new_turf_data ← update_existing_data(turf_data)
    new_turf_data ← create_new_turf_data(turf_data)
    save_turf_data(new_turf_data)
ENDSUBROUTINE

SUBROUTINE create_new_client()
    new_client ← enter_client_details()
    save_new_client(new_client)
ENDSUBROUTINE

SUBROUTINE search_for_client()
    client_data ← get_client_data()
    lastname ← enter_client_lastname()
    client ← get_client_details(client_data, lastname)
    display_client(client)
    # needs to return client for use in generate_new_quote
    RETURN client
ENDSUBROUTINE

SUBROUTINE generate_new_quote()
    dimensions ← enter_dimensions()
    turf_data ← get_turf_data()
    display_turf_data(turf_data)
    choice ← get_user_choice(LEN(turf_data))
    quote ← calculate_quote(dimensions, turf_data[choice])
    display_quote(quote)
    client ←  search_for_client()
    update_client_quote(client, quote)
ENDSUBROUTINE
```

To complete the skeleton code we would need to write out the outlines of a further 15 subroutines referenced in the code above.

```
SUBROUTINE display_main_menu()
ENDSUBROUTINE

SUBROUTINE get_turf_data()
   RETURN []
ENDSUBROUTINE

SUBROUTINE display_turf_data(turf_data)
ENDSUBROUTINE

SUBROUTINE update_existing_data(turf_data)
   RETURN turf_data
ENDSUBROUTINE

SUBROUTINE create_new_data(turf_data)
   RETURN turf_data
ENDSUBROUTINE

SUBROUTINE save_turf_data(turf_data)
ENDSUBROUTINE

SUBROUTINE enter_client_details()
   RETURN Client('', '', '', '', '', '', 0)
ENDSUBROUTINE

SUBROUTINE save_new_client(new_client)
ENDSUBROUTINE

SUBROUTINE get_client_data()
   RETURN []
ENDSUBROUTINE

SUBROUTINE enter_client_lastname()
   RETURN ''
ENDSUBROUTINE

SUBROUTINE get_client_details(client_data, lastname)
   RETURN client_data[0]
ENDSUBROUTINE

SUBROUTINE display_client(client)
ENDSUBROUTINE

SUBROUTINE enter_dimensions()
   RETURN [0.0,0.0]
ENDSUBROUTINE

SUBROUTINE calculate_quote(dimensions, turf)
   RETURN 0.0
ENDSUBROUTINE
```

```
SUBROUTINE update_client_quote(client, quote)
ENDSUBROUTINE
```

This concludes all of the skeleton subroutines defined in the structure charts earlier in this section. Each of these subroutines should be small and simple enough to be solved in these isolated modules. The complete solution would run to many pages, but the following three subroutines are completed as an example of how to go from skeleton code to full code (albeit still in pseudo-code form).

```
SUBROUTINE display_turf_data(turf_data)
   OUTPUT 'Turf Data…'
   FOR i ← 0 TO LEN(turf_data)-1
      OUTPUT turf_data[i].turf_type
      OUTPUT turf_data[i].cost_per_sq_m
   ENDFOR
ENDSUBROUTINE

SUBROUTINE enter_dimensions()
   OUTPUT 'enter the garden width'
   # no validation used here for simplicity
   width ← STRING_TO_REAL(USERINPUT)
   OUTPUT 'enter the garden length'
   length ← STRING_TO_REAL(USERINPUT)
   RETURN [width, length]
ENDSUBROUTINE

SUBROUTINE calculate_quote(dimensions, turf)
   quote ← dimensions[0] * dimensions[1] * turf.cost_per_sq_m * 1.5
   RETURN quote
ENDSUBROUTINE
```

Each of these subroutines has the benefit of being easily understood and focused on just one aspect of the overall problem.

Using interfaces has taken away the need for global variables, these are variables that are accessible throughout the program that, although sounding quite convenient, can confuse code and make it much harder to spot errors. Having said that, there could be a role for some global constants in this program: the factor in the quote for labour (currently hard-coded as `1.5`), and also the names of the external files for the clients and the turf.