Computational Thinking (2.1)

Abstraction: The process of hiding unnecessary detail and highlighting details that are important to improve the clarity of what we are trying to convey.

Only the satellite view of a map is a true representation, however, it is the least useful (it would be pointless to give it to a tourist for navigation). All other map types are abstractions. It would be difficult and confusing to model and display certain features without abstraction. Abstracted maps are therefore much more useful than realistic maps and will be much more capable of meeting their intended purpose.

Thinking Ahead: To prepare for a future event or situation by thinking about what might happen.

Thinking ahead will help you to:

- Maximise code efficiency
- Minimise error

Preconditions: Conditions that affect how you design the solution, imposing sensible, essential restrictions.

Reusing code:

- Shortens development time
- Saves systems resources
- Lowers development costs

Advantages of using libraries:

- Routines are already compiled, so they are tested and error-free
- Saves work for programmers, shortens overall development time, and reduces costs
- Routines may be used multiple times
- Allows the programmer to use other programmers' expertise

Caching: Where instructions or data retrieved from secondary storage and placed into main memory often remain there in case they are needed again before the program using them ends. This results in faster retrieval times, as instructions or data don't need to be fetched from secondary storage, which is much slower.

Prefetching: Where data is retrieved from main memory and placed into the cache before it is required. It is quicker still to access items from the cache than from main memory. Prefetching needs clever algorithms that must predict — with a high degree of certainty—that an instruction or piece of data will be required soon.

Caching improves speed and efficiency. However, caching algorithms can be very complicated to implement. The wrong data is often fetched and cached; subsequently, it must be removed or flushed.

Thinking Procedurally: A disciplined method of thinking in sequence, in order and logically.

Problem Decomposition: This involves breaking down a complex problem into smaller parts that are more manageable and easier to understand.

Break the problem down into a top-down structure diagram using stepwise refinement, where the task the program needs to perform is split into smaller sub-tasks. Each of these sub-tasks can be split up into even smaller sub-tasks until the lowest sub-tasks in the diagram perform a single function and can be tackled as small, independent modules. These sub-tasks can be assigned to individual programmers or small teams and can be written and tested in isolation before being integrated into the overall solution.

Pattern Recognition: This involves finding the similarities or patterns among small, decomposed problems that can help us solve more complex problems more efficiently.

Identifying all the sub-procedures makes it easier to spot if a module that performs any of these tasks already exists. If there is, its code should be reused.

Thinking Logically

Boolean Expression: A statement that, once evaluated, results in either True or False. A Boolean expression may be evaluated as part of a selection statement, causing your code to branch (IF ... THEN). Alternatively, it might be evaluated as part of an iteration statement, which could cause your code to enter or exit a loop (FOR..., WHILE...).

Decision Points: Points at which the program must choose between different actions, based on the values of its current inputs and variables

Thinking Concurrently: involves finding which parts of a program can be processed at the same time, and which parts are dependent on others.

Concurrency: where an application is making progress on more than one task at the same time by giving each task a slice of processor time in turn. This is handled by scheduling algorithms. Concurrency is about dealing with lots of things at once.

With single-threaded programs, you could put your finger on the code and trace it line by line, jumping off to procedures and functions and branching when required. With multiple-threaded programs, it is as if you have multiple fingers tracing through your code at the same time. Each finger is following its own separate thread and is executing its code at the same time.

Parallelism: where multiple parts of a task or multiple tasks are executing at the same time by making use of multi-core and/or multi-processor infrastructure. Parallelism is about doing lots of things at once.

In a single-core CPU you might get concurrency but never parallel processing.

Specialisation: where more specialised co-processors such as a GPU take on the load of certain tasks.

Benefits of Concurrency

- Reactive programming: where a user can interact with applications while other tasks are running in the background.
- Availability of services: where we shouldn't have a long-running task delaying short-running ones.
- Parallelism: A complex program could make better use of multiple resources in a multicore and/or multi-processor system.

These 3 benefits lead to:

- An increase in the number of tasks completed in a given time
- A reduction in the time that would be wasted by the processor waiting on a user or another process

Disadvantages of Concurrency

Safety: Concurrent tasks should not be allowed to corrupt the consistent state of a program.

Liveness: Tasks should not suspend and then indefinitely wait for each other (deadlock).

Resource consumption: Threads can consume resources through scheduling and context-switching, causing overheads. Poorly designed concurrent programs can run slower than their sequential counterparts