2.1.0 Elements Of Computational Thinking

Revision sheet

1 Thinking Abstractly

Abstraction is the removal of excessive details to arrive at a representation of a problem that consists of only the key features.

1.1 Types Of Abstraction

There are a number of different types of abstraction.

1.1.1 Representational Abstraction

This often involves analysing what is relevant to a given scenario and simplifying a problem based on this information.

1.1.2 Abstraction By Generalisation

This involves grouping together similarities within a problem to identify what kind of a problem it is. It allows certain problems to be categorised as being of a particular type, thus a common solution can be used to solve these problems.

1.1.3 Data Abstraction

In this subcategory, details about how the data is being stored are hidden. This means that programmers can use abstract data structures (eg, stack or queue) without worrying about implementation of them.

1.1.4 Procedural Abstraction

This is where things can happen (eg, pushing data to a stack) without having any knowledge used to implement this functionality. It models what a subroutine does without considering how this is done. Once a procedure has been coded, it can be reused as a black box.

1.1.5 Multiple Levels

Large complex problems make use of multiple levels of abstraction - with the highest levels of abstraction closest to the user, it is these levels that are usually responsible for providing an interface for the user to interact with the hardware whereas the lower levels are responsible for performing tasks through the execution of machine code.

1.2 The Need For Abstraction

Abstraction allows non-experts to make use of a range of systems or models by hiding information that is too complex or irrelevant to the system's purpose. Abstraction enables for more efficient designs during software development as programmer scan focus on elements that need to be built into the software rather than worrying about unnecessary details. This reduces time spent on the project and prevents it from getting unnecessarily large. Abstraction is used throughout networking and programming languages.

1.3 Differences Between Abstraction And Reality

Abstraction is a simplified representation of reality. Real world entities may be represented using computational structures (for example tables and databases) whereas real world values are often stored in variables. Within object oriented programming, objects are abstractions of real world entities.

1.4 Devising Abstractions

There are a number of questions which must be considered when devising an abstract model for a given scenario:

- What is the problem that needs to be solved by the model?
- How will the model be used?
- Who will the model be used by?
- Which parts of the problem are relevant based on the target audience and the purpose of the model?

2 Thinking Ahead

The purpose of thinking ahead is to make programs easy and intuitive for users to use.

2.1 Inputs And Outputs

When designing a system, it is important to plan the inputs and outputs. This allows programmers to consider the most efficient data type which the inputs can be stored in as well as the most efficient order in which to request the inputs from the user. It also allows them to consider the input devices which they may need.

2.2 Preconditions

These are requirements which must be met before the program can be executed, which are put in place to prevent errors or the program crashing. This can include error checking; for example, before removing an item from a stack, you check to see if the stack is empty because if it is, the program would otherwise return an error. Preconditions can also be included in the documentation, in which case it is the users responsibility to ensure inputs meet the requirements specified by these preconditions. By explicitly ensuring these conditions are met, subroutines are made more reusable.

2.3 Caching

This is the process of storing instructions or values in cache memory after they have been used as they may be used again. This is common practice with web pages that a user frequently accesses as it saves time.

2.3.1 Precaching

This is a more advanced method of caching which uses an algorithm to pre-empt what instructions are likely to be needed to be fetched soon, these are then fetched and loaded into the cache before they are needed therefore saving time when they are actually needed.

2.4 Reusable Program Components

Commonly used functions are often packaged into libraries for reuse. Teams working on large projects that will make use of lots of the same components multiple times might choose to put together a library of their own. When designing projects, the problem is decomposed - meaning it is broken into smaller tasks. This allows developers to think about how each task can be solved and identify where program components developed in the past, pr externally-sourced program components can be reused to simplify the development process. Reusable components are more reliable than newly coded components as they have already been tested and perfected which saves time, money and resources.

3 Thinking Procedurally

By thinking procedurally, the tasks of writing a problem is made a lot simpler by breaking down a problem into smaller parts which are easier to understand therefore they are easier to design.

3.1 Problem Decomposition

This is the first stage of procedurally analysing a problem. During this step, the requirements given by the user/ client are broken down into component parts. Ultimately, this step results in a series of sub-problems which are easier to solve. By decomposing the problem, it becomes more feasible to manage and each category of sub-problems can be assigned to different people according to their skill sets. A common decomposition technique is top down design, where a single problem (level 1) is broken into a few sub-problems (level 2) then each sub-problem is further decomposed into many smaller sub-problems (level 3). This multi-level approach is also known as stepwise refinement, this is the preferred method to be used in very large problems. The ultimate result of top-down design is to have a single sub-problem which would be a single subroutine in the code therefore each sub-problem should only be one thing.

3.2 Order Of Steps Needed To Solve A Problem

After decomposing the problem, it is important to think about the order in which operations are performed. For example, some programs require certain inputs to be entered by the user before the processing can be carried out. It might be possible for multiple subroutines to be executed simultaneously within a program - programmers must decided if this is possible based on the interaction between them and the user. A similar principle is needed when considering how a program is used - programs should be built so that

there is one route through the app which is logical (eg, fast food ordering app shouldn't let you pay until you have selected what you want).

4 Thinking Logically

Thinking logically allows you to plan and prepare for different scenarios because it provides a foresight of all the decisions made throughout the whole program. Decision making is one of the most important aspects of problem as good decision making is the fundamental aspect involved in solving problems effectively.

4.1 Decision Making In Problem Solving

A decision is a result reached after some consideration. Decisions have to be continually made when developing software. Commonly one of the first decisions is made when a program is being designed, is which paradigm is going to be used? As decisions are made, the possible solutions we can pick from become limited, which helps us make further decisions.

4.2 Conditions That Affect The Outcome Of A Decision

When making decisions, there are a number of factors which get taken into account. To make an appropriate decision, conditions have to be evaluated and ranked from most to least important; this makes it easier to select the best option for the solution.

4.3 Decisions Affecting The Flow Of A Program

Decisions are made to determine how different parts of the program are completed. Thinking logically also involves identifying where decisions need to be made by users within the program and planning the outcomes of the decisions made - each decision could, in theory, change the entire rest of the program from the users perspective.

5 Thinking Concurrently

Concurrent processing is where more than one task can be completed at any one time. However, this doesn't mean that the tasks have to be completed exactly at the same time (as tasks on a multicore CPU would be); it can mean that multiple tasks are given time slices in the processor, giving the illusion that they are being completed simultaneously. When determining parts of a problem that could be solved concurrently, the first step is often to identify which sections of the problem are related as these can often be solved simultaneously therefore they can be dealt with concurrently.

5.1 Concurrent Processing

The key difference between concurrent thinking and concurrent processing is that concurrent processing uses a computer processor whereas concurrent thinking uses a human brain. Concurrent processing and parallel processing are often confused as being the same thing, however they are slightly different. Parallel processing is when multiple processors are used to complete more than one task simultaneously whereas concurrent processing is where each task is given a time slice of processor time.

5.1.1 Benefits

The number of tasks completed in a given time is increased; less time is wasted waiting for an input or user interaction as other tasks can be completed.

5.1.2 Drawbacks

Concurrent Processing can take longer to complete when a large number of users or tasks are involved as processing cannot be completed at once; there is an overhead in coordinating and switching between processes which reduces program throughput; not all tasks are suited to being broken up and solved concurrently.