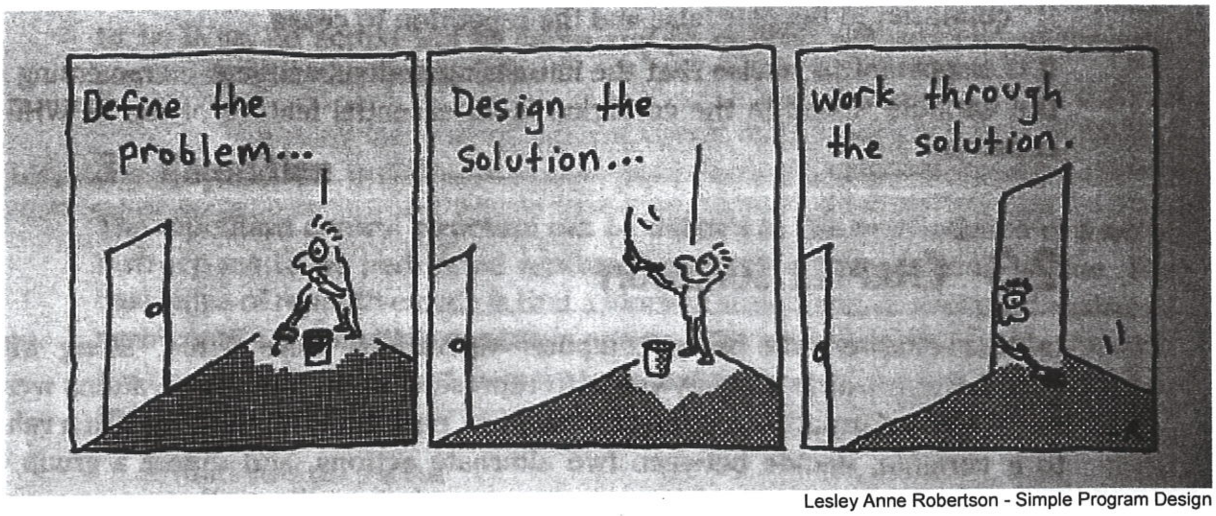
TEMASEK
JUNIOR COLLEGE

Temasek Junior College
JC H2 Computing
Problem Solving & Algorithm Design 1 – Introduction

1 System Development Life Cycle



Lesley Anne Robertson - Simple Program Design

Through applying the knowledge of relevant computing concepts and computational thinking skills in the **system development life cycle** (SDLC), learners will be able to create solutions to **authentic problems**.

The diagram below outlines the key stages of the SDLC (**see Fig 1**):

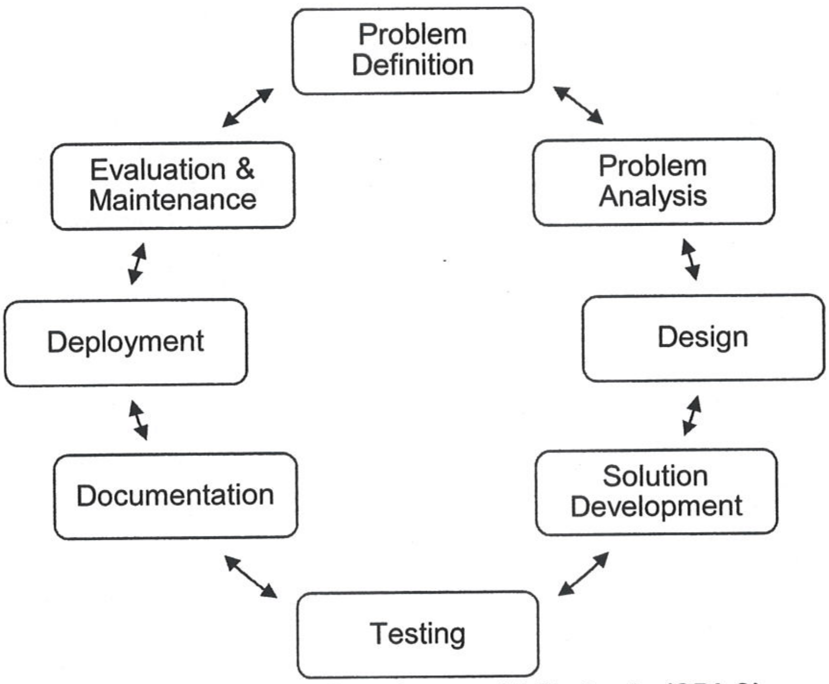


Fig 1: System Development Life Cycle (SDLC)

From the diagram, we can see that besides being cyclical in nature, each stage of the system development life cycle may also be iterative in nature to reflect the fact that more work might be needed in a preceding stage to complete the current stage.

Descriptions to some of the key stages in the SDLC <u>in relation to computing</u> are provided in the table below.

Stage	Description
Problem Definition	Also known as the **research phase**. The starting stage of the SDLC. Personnel involved need to establish clearly what the problem is by determining the scope of the requirements and data flows.
Problem Analysis	Personnel involved need to think logically about how the problem can be decomposed into smaller and more manageable parts.
Design	Personnel involved often apply **abstraction** techniques e.g. **object oriented programming** (OOP) techniques to focus on important parts of the problem while hiding unnecessary details as they think about possible solutions. Personnel involved **creating an algorithm** that solves the problem. (Note: OOP will be covered in great detail in H2 Computing.)
Solution Development	The algorithm is then translated into a **computer-based program** using a programming language that will work for the functionalities as planned. This process is also known as the **phase of computer-based solution**.
Testing	The computer-based program is then **tested** to ensure that it works as designed. **Normal**, **abnormal** and **boundary** data are often selected to work with a suitable **test plan** designed to test the program rigorously. Depending on the results obtained from the testing process(es), the computer-based program may or may not require further development before deployment.
Documentation	**Critical information** on the computer-based program such as program architecture, compatible operating environments, business rules, databases, files, troubleshooting processes, application installation and code deployment are **documented** to facilitate implementation, future development, maintenance and knowledge transfer.
Deployment	The computer-based program is implemented for the relevant stakeholders. Deployment may be automated, semi-automated or manual.
Evaluation & Maintenance	**Regular maintenance and evaluation** to **update** the computer-based program is desired. The developers of the computer-based program may further evaluate the it and also take into account user-feedback. Maintenance can be • **corrective** e.g. bug fixes to rectify logic or run-time errors. • **adaptive** e.g. enhancement to functionalities. • **perfective** e.g. changing file handling processes from sequential to direct access to boost performance.

Overtime, several scenarios may occur that call for a major overhaul or redesign of the computer-based solution. These include:

- Drastic changes to the parameters and/or nature of the problem.
- Demand for or occurrence of more efficient and better solution.
- Changes to associated operating conditions, system environments, dependencies etc.

The computer-based program will then be deemed to have reached its **end-of-life** and a new SDLC begins.

In the modern world of system development, a new SDLC tend to overlap with an existing one, before the end-of-life of the current computer-based program. This ensures seamless continuity and validity of the computer-based program to meet modern day demands of software capabilities.

2 Algorithms – A Brief Introduction

In the previous section, we note that **algorithms** are created to solve the problem during the design phase of the SDLC **(see Fig 2)**.

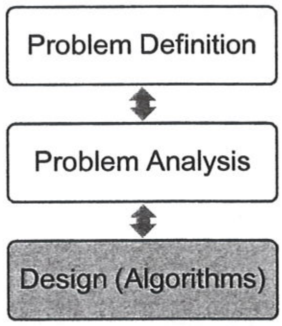


Fig 2: Algorithms are used in the Design stage of the SDLC

An algorithm is a **well-ordered** collection of **unambiguous** and **effectively computable operations**, which when executed, **produces a result** and halts in a **finite** amount of time.

- **Well-ordered** refers to accurate knowledge of the correct sequence of operations to be executed in the algorithm.

- An **unambiguous** operation is one that can be understood and carried out directly by the computing agent without needing to be further simplified or explained. **Ambiguous** statements in an algorithm can leave the computing agent confused and unsure about how to correctly execute the operation the statement is describing.

- An **effectively computable** operation means there exists a computational process that allows the computing agent to complete that operation successfully.

- Algorithms solve problems. **An algorithm must produce a result that is observable**, such as a numerical answer, a new object, or a change to its operating environment. When a desired result is obtained, it can be concluded that the algorithm works correctly for the scenario is being implemented in. It is important to note that a single desired result is insufficient to conclude that the algorithm works for all scenarios. Nevertheless, without some observable result, it cannot be concluded whether the algorithm is correctly designed.

- Another important characteristic of algorithms is that the result must be produced after the execution of **a finite number of operations in a finite amount of time**. The algorithm must eventually reach a statement that says, "Stop, you are done!" or something equivalent.

In brief, algorithms is a **sequence of steps** that can be carried out to perform a task.

3 The Learning of Fundamental Algorithms

Learning fundamental algorithms and understanding that the creation of algorithms is a core skill is an important component of H2 Computing (and any other Computing education). We will often need to apply standard algorithms and basic mathematics in the creation of programming solutions for a range of problem scenarios.

In designing a solution to a problem, we often express the algorithmic representation of the solution using sequences of steps written in **structured English or pseudocode.** We may also illustrate the sequence of steps using a **flowchart**. A brief description of these methods are provided in the table below.

Method	Description
Structure English	A subset of the English language. It consists of command statements used to describe an algorithm.
Pseudocode	Resembles programming language in that it uses keywords and identifiers to describe an algorithm. However it does not follow the syntax of any programming language.
Flowchart	A graphic organiser that consists of specific shapes linked together to represent the sequential steps of an algorithm

4 Pre-cursor Concepts

In this section, we take a look at some key concepts that undergird the learning of the use of **structured English**, **pseudocode** and **flowcharts**.

4.1 Keywords for Data Types

The table below gives the keywords used to designate atomic data types.

Keyword	Data Type
`INTEGER`	A whole number
`REAL`	A real number (capable of containing a fractional part)
`CHAR`	A single character
`STRING`	A sequence of zero or more characters
`BOOLEAN`	The logical values TRUE and FALSE
`DATE`	A valid calendar date.

Example 1
An estate agent stores details about the properties for sale on a computer system. For each of the following items name the most suitable data type:
(a) the number of bedrooms
(b) the postcode of the property
(c) whether the property is still for sale (eg TRUE)
(d) the tax code (eg A, B, C)

[Solution]
(a) `INTEGER`
(b) `STRING`
(c) `BOOLEAN`
(d) `CHAR`

4.2 Identifiers

Identifiers (the names given to variables, constants, procedures and functions) are written in **mixed case**.

The following rules apply when writing identifiers:
- Can only contain letters (`A-Z`, `a-z`),
- Digits (`0-9`)
- Underscore "_".
- Must start with a letter
- **Cannot** start with a digit.
- Accented letters (e.g. `á`, `ü`) and other characters (e.g. `$`, `&`) should not be used.

Examples of valid identifiers are `CTGroup`, `CT_Group`, `Age`...

4.3 Variables and Variable Assignment

A **variable** is a storage location for a data value that has an identifier.

When we input data for a process, individual values need to be stored in the memory. We will hence need to be able to refer to a specific memory location so that we can write statements of what to do with the value stored there. We refer to these named memory locations as variables.

Assignment is the process of giving a value an identifier (a name). It can refer to the process the occurs when the value associated with a given identifier is changed.

The assignment operator is the ← symbol.

The following table gives some examples and what they signify.

Example	Significance
`CTGroup ← "03/21"`	String `"03/21"` is given the identifier `CTGroup`. Hence `CTGroup` will store the string `"03/21"`.
`Age ← 17`	Value `17` is given the identifier `Age`. Hence `Age` will store the value `17`.
`Height ← 1.67`	Value `1.67` is given the identifier `Height`. Hence `Height` will store the value `1.67`.

When **updating** a value, we can write the assignment as

```
Age ← Age + 1
```

Hence `Age` will now store the value evaluated by `Age + 1`.

When we **copy a value,** we can write the assignment as

```
AgeOfJC2 ← Age
```

Hence `AgeOfJC2` will stored the value same as that stored by `Age`.

The following diagram illustrates an example of the assignment process (**see Fig 3**).

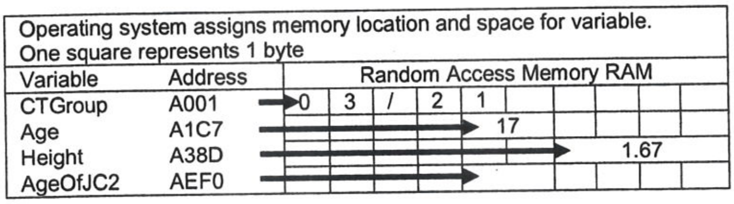


Fig 3: Diagrammatic representation of assignment process

4.4 Operators

(A) Arithmetic Operations

Standard arithmetic operator symbols are used to represent arithmetic operations.

+ Addition

– Subtraction

* Multiplication

/ Division

(B) Comparison Operations

The following comparison operators are used to write statements involving comparison of values.

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

= Equal to

<> Not equal to

(C) Logic Operations

The following logic operators are used to write statements involving logic operations.

AND

OR

NOT

Annex A – Built in Functions

Programming environments provide many built- in functions. Some of them are always available to use; some need to be imported from specialist module libraries.

The table below presents a list of **built-in functions** for reference. In each function below, if the function call is **not properly formed**, the function **returns** an **error**.

Note that the function name may vary for different algorithms and/or programming languages.

`ONECHAR(ThisString : STRING, Position : INTEGER) RETURNS CHAR`
 	
returns the single character at position `Position` (counting from the start of the string with value 1) from the string `ThisString`
Example
`ONECHAR("Barcelona", 3)`
returns `'r'`

`MID(ThisString : STRING, x : INTEGER, y : INTEGER) RETURNS STRING`
 	
returns the string of length `y` starting at position `x` from `ThisString`
Example
`MID("ABCDEFGH", 2, 3)`
returns `"BCD"`

`SUBSTR(ThisString : STRING, Value1 : INTEGER, Value2 : INTEGER) RETURNS STRING`
 	
returns a sub-string from within `ThisString`
`Value1` is the start index position (counting from the left, starting with 1)
`Value2` is the final index position
Example
`SUBSTR("art nouveau", 5, 11)`
returns `"nouveau"`

`LEFT(ThisString : STRING, x : INTEGER) RETURNS STRING`
 	
returns the leftmost `x` characters from `ThisString`
Example
`LEFT("ABCDEFGH", 3)`
returns `"ABC"`

`RIGHT(ThisString: STRING, x : INTEGER) RETURNS STRING`
 	
returns the rightmost `x` characters from `ThisString`
Example
`RIGHT("ABCDEFGH", 3)`
returns `"FGH"`

`LCASE(ThisChar : CHAR) RETURNS CHAR`
 	
returns the character value representing the lower case equivalent of `ThisChar`
If `ThisChar` is not an upper-case alphabetic character then it is returned unchanged
Example
`LCASE('W')`
returns `'w'`

`UCASE(ThisChar : CHAR) RETURNS CHAR`
 	
returns the character value representing the upper case equivalent of `ThisChar`
If `ThisChar` is not a lower case alphabetic character then it is returned unchanged
Example
`UCASE('h')`
returns `'H'`

`CHR(Value : INTEGER) RETURNS CHAR`
 	
returns the character that ASCII code number `Value` represents
Example
`CHR(65)`
returns `'A'`

`ASC(ThisChar : CHAR) RETURNS INTEGER`

returns an integer which is the ASCII value of character `ThisChar`

Example

`ASC('W')`

returns `87`

`LENGTH(ThisString : STRING) RETURNS INTEGER`

returns the integer value representing the length of string `ThisString`

Example

`LENGTH("Happy Days")`

will return `10`

`CONCAT(String1 : STRING, String2 : STRING [, String3 : STRING] ) RETURNS STRING`

Example

`CONCAT("San", "Francisco")`

returns `"SanFrancisco"`

`CONCAT("New", "York", "City")`

returns `"NewYorkCity"`

`TOSTR(ThisNumber : INTEGER) RETURNS STRING`

returns the string value of integer `ThisNumber`

Example

`TOSTR(27)`

returns `"27"`

`TONUM(ThisString : STRING) RETURNS INTEGER or REAL`

returns the integer or real equivalent of the string `ThisString`

Example

`TONUM("502")`

returns `502`

`TONUM("56.36")`

returns the real number `56.36`

- `MOD(ThisNum : INTEGER, ThisDiv : INTEGER) RETURNS INTEGER`
- `INTMOD(ThisNum : INTEGER, ThisDiv : INTEGER) RETURNS INTEGER`

returns the integer value representing the remainder when `ThisNum` is divided by `ThisDiv`

Example

`MOD(10,3)`

returns `1`

`INTMOD(10,3)`

returns `1`

`DIV(ThisNum : INTEGER, ThisDiv : INTEGER) RETURNS INTEGER`

`INTDIV(ThisNum : INTEGER, ThisDiv : INTEGER) RETURNS INTEGER`

returns the integer value representing the whole number part of the result when `ThisNum` is divided by `ThisDiv`

Example

`DIV(10,3)`

returns `3`

`DIV(10,3)`

returns `3`

`INT(ThisNumber : REAL) RETURNS INTEGER`

returns the integer part of `ThisNumber`

Example

`INT(12.79)`

returns `12`

`RND() RETURNS REAL`

returns a random number in the range 0 to 0.99999

Example

`RND()`

returns `0.67351`

Concatenation operator `&` operator – Concatenates two expressions of `STRING` or `CHAR` data type.

Example

`"Temasek" & "█" & " Junior College"`

produces `"Temasek█Junior College"`

`'T' & "0132875Z"`

produces `"T0132875Z"`