Learning Aims

- Identify the components of a problem
- Identify the components of a solution to the problem
- Determine the order of steps needed to solve a problem
- Identify sub-procedures necessary to solve a problem

Procedural abstraction

Procedural abstraction means using a **procedure** to carry out a sequence of steps for achieving some task such as calculating a student's grade from her marks in three exam papers, buying groceries online or drawing a house on a computer screen.

Consider, for example, how you could code a program to create the plan for an estate of 100 new houses.

You could use a procedure which will draw a triangle of certain dimensions and colour. The colour and dimensions are passed as arguments to the procedure, for example:

```
procedure drawTriangle(colour, base, height)
```

This procedure may be called using the statement:

```
drawTriangle("red", 4.5,2.0)
```

The programmer does not need to know the details of how this procedure works. She simply needs to know how the procedure is called and what arguments are required, what data type each one is and what order they must be written in. This is called the **procedure interface.**

Problem decomposition

- Thinking procedurally makes the task of writing a program a lot simpler by breaking a problem down into smaller parts which are easier to understand and consequently, easier to design.

- The first stage of thinking procedurally in software development involves taking the problem defined by the user and breaking it down into its component parts, in a process called problem decomposition.

Advantages of problem decomposition

- As well as making the task of **writing the program easier**, breaking a large problem down in this way makes it very much simpler to **test and maintain.**
- A large, complex problem is continually broken down into smaller subproblems which can be solved more easily.
- When a change has to be made, if each module is self-contained and well documented with inputs, outputs and preconditions specified, it should be relatively easy to find the modules which need to be changed, knowing that this will not affect the rest of the program.
- By separating the problem into sections, it becomes more feasible to manage and can be divided between a group of people according to the skill sets of different individuals.

Advantages of using modular design

- Each module focuses on a **small sub-task** and so it is easy to solve and test/debug
- Makes it **easy to maintain** and **update** part of the system without affecting the rest
- The program will be well-structured
- Development can be **shared** between a **team of programmers** so developed **faster**, allocated according to expertise, improving the quality of the final product
- Reduces the amount of code as it can be **reused**, standard library modules can be used and reduces development time.

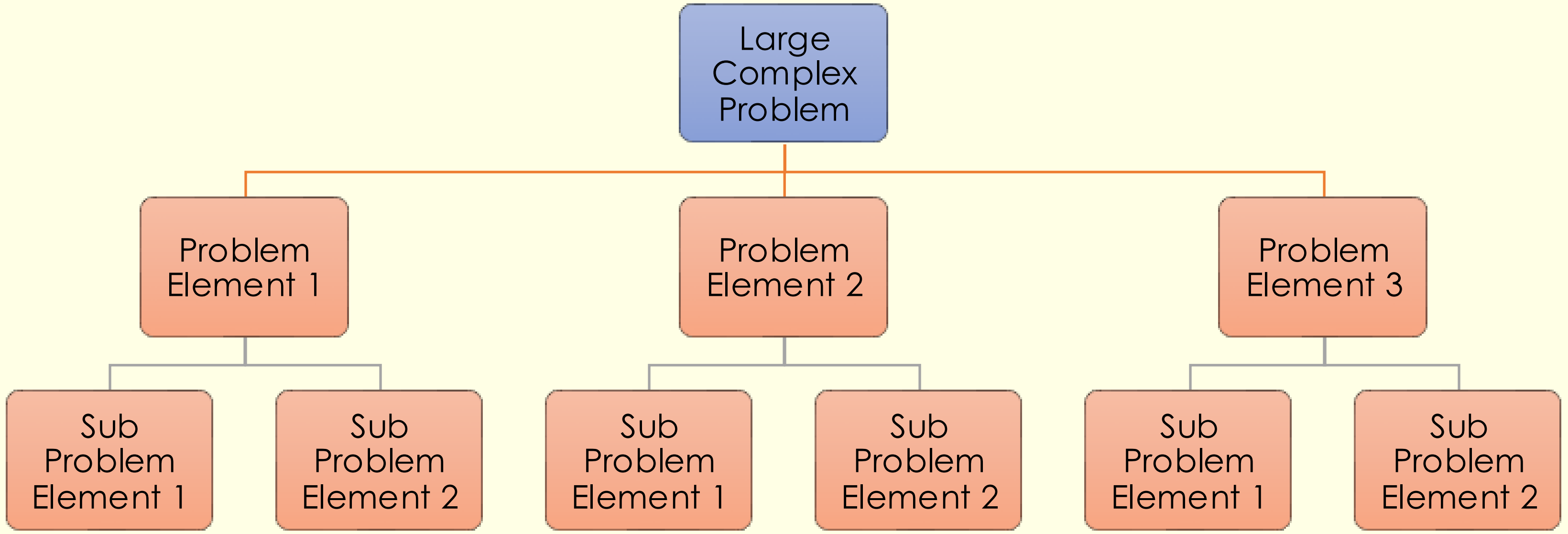
Summary of Decomposition

- Splits problem into sub-problems
- Splits these problems further until each problem can be solved
- Smaller problems are easier to solve
- This approach leads to the **Modular Design** approach for computer programming.
- Team of programmers can work on different elements of the design and speed up production
- Identify if some of the sub-problems can be solved using pre-existing modules and see if new modules can be re-used in different parts of the program.

Top-down design

- Top-down design is the technique of breaking down a problem into the major tasks to be performed
- Each of these tasks is then further broken down into separate subtasks, and so on until each subtask is sufficiently simple to be written as a self-contained module or subroutine.
- Remember that some programs contain tens of thousands, or even millions, of lines of code, and a strategy for design is absolutely essential.
- Even for small programs, top-down design is a very useful method of breaking down the problem into small, manageable tasks.

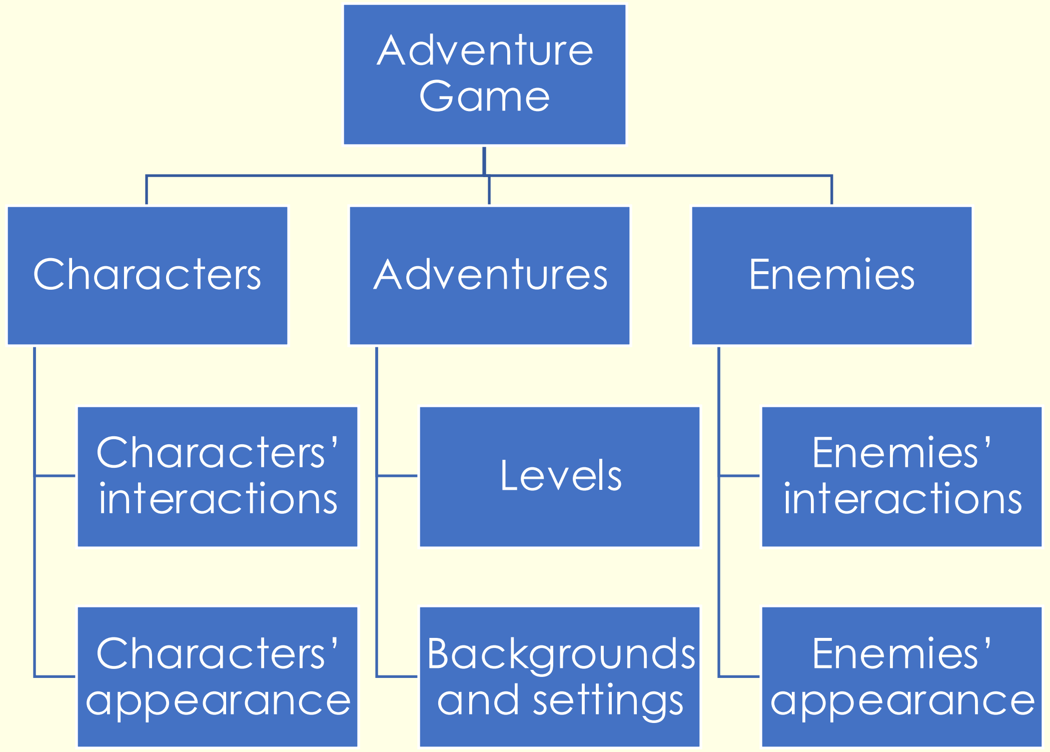
As you can see in the ***'Top-Down'* diagram** below, a problem can be split up into its different elements. These elements can then be further split into sub problems which are more easily to solve.



Example: An adventure game, might be broken down into the following components:

1. Characters
2. Adventures
3. Enemies

These would then be broken down further, as shown:



The aim of using top-down design is to keep splitting problems into subproblems until each subproblem can be represented as a single task and ideally a self-contained module or subroutine.

Each task can then be solved and developed as a subroutine by a different person.

Once programmed, subroutines can also be tested separately, before being brought together and finally integrated.