3 PROBLEM-SOLVING AND DECOMPOSITION

OBJECTIVES

- Explain how to apply a systematic approach to problem-solving.
- Discuss how to create a problem definition.
- Introduce strategies and considerations for the devising of solutions.
- Explain decomposition as a problem-solving strategy.
- Show the benefits of generalising from patterns in problems as well as techniques for creating them.

WHERE TO START

You have your problem. You're ready to start analysing it and coming up with a solution. But not just any solution; one that is specifically formed so that a computer could carry it out. You begin to look at the problem. A question dawns on you.

Where on earth do you start?

It's all very well saying 'come up with a solution'. Real-world problems tend to be big, complex things. Examining any non-trivial problem reveals all manner of hidden details, complex nuances and various facets to consider.

When faced with a complex task, it can be helpful to follow some kind of guide or process, like the instructions for a piece of self-assembly furniture. A step-by-step procedure for problem-solving would be an obvious benefit. Unfortunately, problem-solving is partly a **creative** process. Like painting a landscape or writing a novel, it cannot be totally systematised, but strategies, heuristics and good practices exist to help you during your creative endeavours. These are things that previous generations of problem solvers found useful for attacking a problem's complexity and this chapter will introduce you to them.

A systematic approach

An early example of a systematic approach to general problem-solving was introduced by George Pólya. He was a Hungarian mathematician who in 1945 wrote the first edition of an influential book called *How to Solve It* (Pólya, 1973), which is, impressively, still in print after more than half a century.

Despite its age, Pólya's book still abounds with relevant and helpful tips. Furthermore, it has guided many problem solvers down the years, including various authors who wrote successor books. I will refer to those and Pólya as we examine the various stages of problem-solving.

How to Solve It takes an approach to problem-solving inspired by the best traditions of mathematical and natural sciences.

Whereas the scientific method follows the steps:

- form hypothesis;
- plan experiment;
- execute experiment;
- evaluate results.

Pólya (1973) advocates:

- understand the problem;
- devise a plan;
- execute the plan;
- review and extend.

Computational thinking is compatible with this view of problem-solving. However, this chapter focuses on the first two stages, namely understanding the problem and devising a plan.

Don't panic

Getting started is hard. We all know that. Size, complexity and the sheer number of unknowns can frustrate our attempts to start, but a few rational hints can help us out.

First, **don't be put off by perceived size and complexity**. Big problems are rarely impenetrable monoliths. In fact, you'll usually find that a big problem is really a whole group of smaller, interrelated problems. As we'll see later in the chapter, the trick is to pick them apart and reduce a big problem into smaller, simpler, more manageable ones.

Second, **resist the urge to jump straight into writing a solution**. If you do, the best probable outcome is that you'll solve only a very specific version of the problem. More likely, you'll get partway before finding you're solving the wrong problem altogether.

Now, if you're ready, we can begin by defining the problem.

DEFINING THE PROBLEM

> The hardest part of problem solving is characterising the problem.
>
> (Michaelson, 2015)

Problem-solving involves transforming an undesirable state of affairs (the start point) into a desirable state of affairs (the goal). The start point and the goal are intimately linked. By examining the start point, you nail down exactly what is undesirable and why. In doing so, you reveal more about what your goal should be.

You might find things undesirable for any number of reasons.

- Maybe your current process is too slow, in which case your goal will involve making measurable improvements to the process's speed.
- Maybe you regularly need to make certain decisions, but you have too much data to handle. This implies your goal may be to somehow automate your decision-making strategy or filter information before you analyse it.
- Maybe you have missing information about how something behaves. This suggests producing a model or simulation of it.

When trying to understand the problem, Pólya tells us that 'it is foolish to answer a question that you do not understand', and offers the following advice:

- If someone else gave it to you, try **restating the problem in your own words**.
- Try and represent the problem using **pictures and diagrams**. Humans deal better with visual representations.
- There will be knowns and unknowns at the start. You should ensure that enough information is known for you to form a solution. If there isn't, **make the unknowns explicit**.[14]

Whatever the problem, the key thing to remember is that a goal defines what needs to be done and **not** how it should be done. Thinking about details like designs and algorithms at this stage is too early. Focus instead on what your goal looks like.

When considering what the goal looks like, ask yourself: how do you know when the problem is solved? In other words, how will you know when you're done? 'Done' could mean that efficiency has increased 50 per cent. Or that steps 2 to 5 of a process are executable with zero human intervention.

Try describing how a working solution should look. Or perhaps describe how it should work, but only at a **very** broad level (remember, details are no concern yet). If your problem is to plan a conference (allocating slots to all the speakers, scheduling breaks and meals, arranging enough auditoriums, etc.), then the end product would be a time plan in which every event has a time, location and people assigned.

However you specify the goal, make sure that your language is **clear and specific**. For example, if you aim to improve the speed of the current system, don't specify your end goal simply as 'it should be faster'. Give it measurable accuracy.

If the end goal is more complicated, you could describe every desired 'feature' of the solution. In this case, you'd be writing a specification. As you develop your solution, keep comparing it to the specification to make sure you're solving the problem correctly.

Always write your problem definition and goal as though someone else will eventually take what you've produced and verify for themselves that you've solved the problem. This will force you to take a more objective, considered view.

DEVISING A SOLUTION: SOMETHING TO KEEP IN MIND

Once you have a finished problem definition complete with goal, you can consider the strategy for finding a solution.

This chapter will focus on one strategy in particular – decomposition – since it is the problem-solving strategy at the core of computational thinking, but will also look briefly at several others that can be applied. Before looking at strategies however, there are a few things about the problem-solving process you should keep in mind.

Quality

First, notice that I said **a** solution and not **the** solution. For any problem, there are usually multiple solutions: some good, some terrible and others somewhere in-between. You should focus on finding the best solution you can.

For the **overall** problem, there is likely no perfect solution. Trade-offs between competing parts are almost inevitable. On the other hand, individual parts of a problem may be 'perfected', in as much as their role in the overall solution might be optimisable.[15]

Collaboration

Making problem-solving a collaborative effort is often helpful. Something as simple as explaining your current work out loud often helps you to spot mistakes or potential improvements. Seek out the views of others. People's minds work in different ways.

We may not kill an idea stone dead, as you might do when writing fiction, but a fresh perspective might show where you're going wrong and help you improve it. While you have the attention of other people, try brainstorming with them. Brainstorming sessions thrive on spontaneity. All ideas, however radical they seem, should be recorded, and you should reject nothing out of hand. In fact, wild ideas are to be encouraged. In among those crazy ideas may lie the seeds of a creative new approach that you might ordinarily have missed or self-censored.

Iteration

You should accept that your first attempt at a solution will rarely be the best one. Instead of trying to solve everything in one fell swoop, take an iterative approach. Go back and repeat some of the previous steps in an attempt to improve your current solution.

How many steps you repeat depends on the solution's shortcomings. If your model of the problem is mainly correct but contains an inaccuracy, you may need to go back just a couple of steps to correct the model. If you find that your whole understanding of the problem was wrong, you'll have to go back and reappraise the problem from nearer the beginning.

DECOMPOSITION

As well as providing a guide to general problem-solving, George Pólya's book *How to Solve It* also catalogues different problem-solving techniques called heuristics. A **heuristic** is a specific strategy in problem-solving that usually yields a good enough answer, though not necessarily an optimal one. Examples include trial and error, using a rule of thumb or drawing an analogy.

Computational thinking promotes one of these heuristics to a core practice: **decomposition**, which is an approach that seeks to break a complex problem down into simpler parts that are easier to deal with. Its particular importance to CT comes from the experiences of computer science. Programmers and computer scientists usually deal with large, complex problems that feature multiple interrelated parts. While some other heuristics prove useful some of the time, decomposition almost invariably helps in managing a complex problem where a computerised solution is the goal.

Decomposition is a divide-and-conquer strategy, something seen in numerous places outside computing:

- Generals employ it on the battlefield when outnumbered by the enemy. By engaging only part of the enemy forces, they neutralise their opponent's advantage of numbers and defeat them one group at a time.
- Politicians use it to break opposition up into weaker parties who might otherwise unite into a stronger whole.
- When faced with a large, diverse audience, marketers segment their potential customers into different stereotypes and target each one differently.

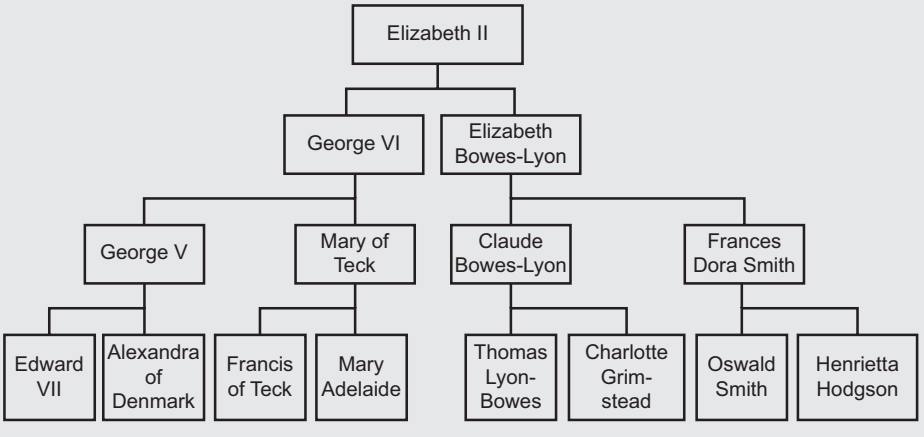
Within the realm of CT, you use divide and conquer when the problem facing you is too large or complex to deal with all at once. For example, a problem may contain several interrelated parts, or a particular process might be made up of numerous steps that need spelling out. Applying decomposition to a problem requires you to pick all these apart.

Recursion: a technique used to simplify a problem. It defines the solution to a large, complex problem in terms of smaller, simpler problems of the same form as the original problem. This is a very powerful idea and one that is heavily used in CS.

By applying decomposition, you aim to end up with a number of sub-problems that can be understood and solved individually. This may require you to apply the process recursively (see above). That is to say, the problem is re-formed as a series of smaller problems that, while simpler, might be still too complex, in which case they too need breaking down, and so on. Visually this gives the problem definition a **tree structure**.

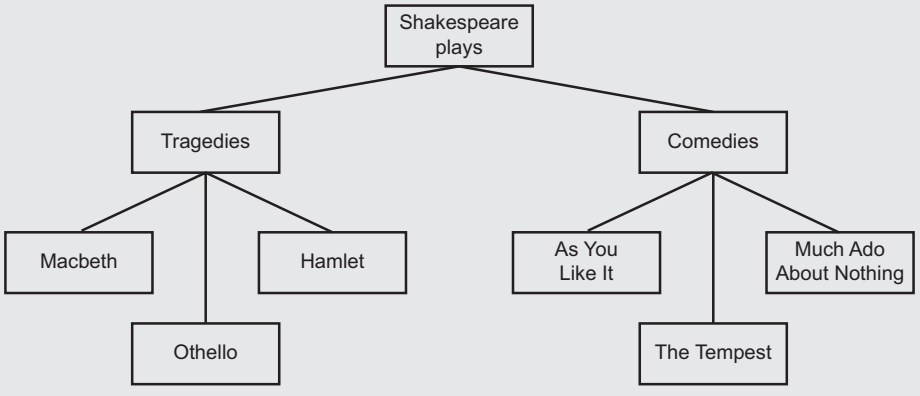
The conceptual idea of a tree structure is very important to CS and often goes hand in hand with the idea of recursion. A tree represents a collection of entities[16] organised into hierarchical relationships. Each 'parent' entity may have any number of 'child' entities (including none). A tree has a single 'root' entity and all childless entities are called 'leaves'. Visually, it resembles a tree – admittedly an upside-down one, with the 'root' at the top and the leaves along the bottom. Trees can represent a huge range of different structures, including something as rigidly defined as a family tree (see Figure 3.1).

Figure 3.1 Royal family tree



Or something put together arbitrarily like arranging Shakespeare's plays by genre (Figure 3.2).

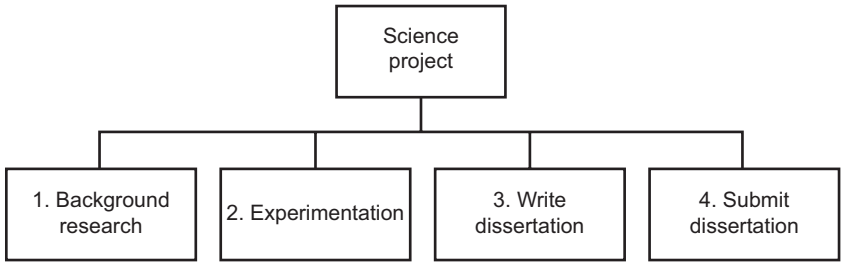
Figure 3.2 Shakespeare plays arranged by genre into a tree structure



Let's look at an example. One task that many of us encounter during our education is the production of a dissertation. Writing an academic work some tens of thousands of words in length can inspire fear, panic or even despair. By itself, one monolithic task entitled 'write dissertation' is unmanageable, but by decomposing the problem you reduce it into smaller parts, each of which possesses far fewer details. You can then focus your attention on each part and deal with it much more effectively. By solving each sub-problem, you end up solving the overall problem.

The process of writing a typical scientific dissertation can be broken down as in Figure 3.3.

Figure 3.3 Science project task breakdown

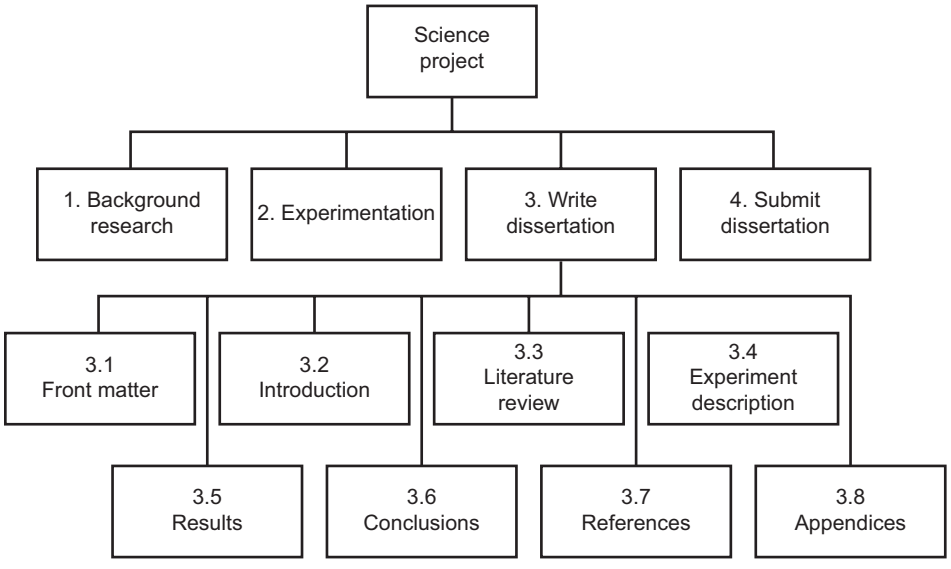


Each of these tasks conceals several sub-tasks. Let's focus on the task 'Write dissertation', which we can break down as in Figure 3.4.

What has this done? First, it has made explicit what constitutes the parent task, making it much clearer what 'write dissertation' actually means. In addition to revealing more

of the task's detail, decomposition has also revealed more unknowns. For example, a dissertation requires a references section, but you may not know which citation style your institution demands.

Figure 3.4 Science project task breakdown revised



This has decomposed the task somewhat, but some sub-tasks may be still too big. Those tasks should be broken down further by applying the same decomposition process. For example, the front matter of a dissertation is made up of several parts, such as:

- 3.1.1 write title page;
- 3.1.2 write copyright section;
- 3.1.3 write abstract;
- 3.1.4 write contents section;
- 3.1.5 write list of figures section.

This time, all the resulting tasks are now manageable. None of them will contain more than a few hundred words, and some can even be done automatically by the word processor.

At this point, you can go back up the tree and find a different task that requires further decomposition. This process needs to be repeated until all the leaves of the resulting tree are problems you can solve individually.

Avoid getting bogged down in the details of **how** to solve the sub-problems. At this stage, we're still focusing on what to do, not how to do it.

Decomposition won't result in a complete plan of action, but it can give you a starting point for formulating one. For example, a fully decomposed problem definition can show all the individual tasks, but doesn't necessarily show the order in which you should tackle them. However, relationships between individual sub-problems should become apparent. In the dissertation example, writing up the results and conclusions (sections 3.5 and 3.6) can't be done before the experimentation (section 2) is completed. That doesn't mean everything in section 3 has to wait; sections 3.1–3.4 could be written before experimentation begins.

Decomposition aids collaboration. If you decompose a problem well (so that the sub-problems can be solved independently), then different people can work on different tasks, possibly in parallel.

OTHER EFFECTIVE STRATEGIES

Just because decomposition enjoys status as a core idea, that doesn't mean other problem-solving strategies have no a place in CT. While it's out of this book's scope to discuss all strategies, there's space to briefly explain several of the most useful ones. These strategies can complement decomposition by being applied to one or more of the resulting sub-problems. Each one may not useful in every scenario, but it's good to keep them in mind.

Think critically

The purpose of critical thinking is to question ideas and see that they stand up to scrutiny. You should examine the available information and the decisions you've taken **sceptically**, as if justifying to yourself that they are valid. All key decisions should have a good reason behind them.

If, for example, you were to design the layout of a bookshop, you might ask:

> Have I chosen the right structure for laying out the books on the shelf?

That's a good starting question, but it's not clear what 'right structure' actually means. A quick look at your goal will help you clarify that. If you want it to be quick and easy for the bookshop owner to add new books to the shelves, then it's probably fine for them to be relatively unordered. The customers will have to browse the books, but that's normally expected in a bookshop. But if the venue is more like a library, where the customer typically looks for a specific book, then that suggests a specific structure. Your question then becomes:

> Have I chosen a structure that makes it easy[17] to locate a specific book?

For a bookshop, this usually entails ordering books alphabetically by the author's name. Of course, this also means more work for the bookshop owner, who has to spend some of their time sorting the books.

In addition to validating ideas, you will also expose assumptions in the process. This is very important because assumptions made implicitly could well turn out to be false. Deciding that books need to be sorted assumes that the bookshop owner can afford to put in the time required for it. This raises the question: is that justified? It may turn out that you've exposed the need for further employees or a revised solution.

Perhaps the most common questionable assumption we're all guilty of is that our solutions will work flawlessly. For any system you put in place, **always** ask the question:

> What if it goes wrong?

For a bookshop, you might ask: What if a book is not in the correct place on the shelf? If a gap appears on the bookshelf, how do you distinguish between a misplaced book and one that you simply don't have? Your response to such questions can vary. You could either work to improve your solution and eliminate the possibility of the problem occurring, or you could accept the possibility of it going wrong and put a contingency plan in place to be executed when the failure occurs.[18]

Solve a concrete instance

When you're trying to solve a problem, the terms you're dealing with can sometimes feel quite abstract and devoid of detail. While abstraction is certainly useful (see Chapter 4), it can be challenging to solve a problem when things are not defined in detail.

R. G. Dromey, who authored a problem-solving book inspired by Pólya's book (Dromey, 1982), cites this as a common cause of getting stuck early on. Dromey points out that it's easier to deal with problems in concrete terms because the details are more readily manipulated and understood.

As an example, consider automating the drawing of complex shapes and images. The idea of a shape is rather abstract. Just think how many varied types of shapes and patterns exist. To gain a foothold on the problem, you could take one complex shape (let's say a smiley face, see Figure 3.5) and think about how to solve that single example.

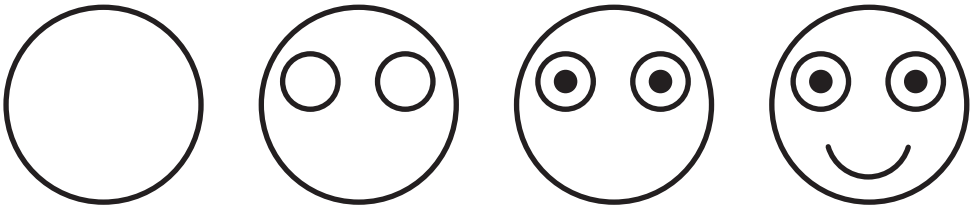
Figure 3.5 Smiley face

By combining this strategy with decomposition, we might find we could draw a smiley face by breaking it down into several simpler shapes, which are each easy to draw:

- A circle, to serve as the head.
- Two concentric circles (the smaller one of which is solid) which make eyes.
- A curve, which makes the mouth.

The solution for this one example points the way to a general solution: any complex, unfamiliar shape can be considered as being made up of several simpler, familiar shapes (see Figure 3.6).

Figure 3.6 Breakdown of drawing smiley face



Find a related problem

Another piece of advice from Dromey, useful for gaining a foothold on a problem, advises you to examine the solution to a closely related problem. This helps for reasons that are similar to the previous strategy. A solution to another similar problem (one that's analogous to or a simplified version of your original problem) can point the way to a solution for your own problem.

Think back to the earlier example of writing a dissertation. The approach taken was to break the dissertation down into small tasks that could each be dealt with individually. The solution was tailored towards a science student, but such an approach could be reused by a student of law, economics, politics or whatever.

Just as with the 'solve a concrete instance' strategy, you should use this approach with caution. It is used to give you insight into a problem, not to solve it completely. Any solution you find will need some changes. Reusing it incautiously risks unwittingly moving the goalposts: you will end up solving a different problem without realising it. A law student could reuse the approach, but would need to adapt it – for example, they would have no need for an experimentation stage and so could ditch it.

Work backwards

Working backwards involves starting at the goal state and going backwards stage by stage. At each point, you look at the current stage and deduce what is required to arrive at it. This suggests things about the nature of the preceding stage.

This strategy is especially effective with a well-defined goal state. For example, if your problem is to plan a route somewhere and arrive by a certain time, you begin with a very clear goal state. From there, you can work backwards to the starting point, calculate a route and add up the intervening times.

Example: starting from your house on Main Street, arrive at the Town Hall by 3.00 p.m. What is the departure time?

- 3.00 p.m. Arrive at Town Hall
- 2.55–3.00 p.m. Walk from Town Hall underground station to Town Hall
- 2.45–2.55 p.m. Take underground from City bus station to Town Hall underground station
- 2.30–2.45 p.m. Take bus from Main Street bus station to City bus station
- 2.25–2.30 p.m. Walk from house on Main Street to bus station
- 2.25 p.m. Departure time.

PATTERNS AND GENERALISATION

Effective problem-solving actually involves more than just finding a solution. Once you've found a solution, you should put effort into its improvement by finding ways to make it more powerful. Recognising and exploiting patterns is a crucial early step in this process.

Why look for patterns?

As you analyse a problem and begin work on its solution, you'll probably notice that some elements repeat or are at least very similar to one another. If you don't automatically notice the patterns in a solution, you should make an effort to seek them out because they are the first step to making your solution more manageable and powerful.

In an earlier example, we saw an approach to drawing a simple image of a face. This approach broke the image down into simpler, elemental shapes that could be positioned and drawn easily. An algorithm that draws such an image might therefore look like this:

1. Draw circle with radius 30 at position 50,50 with line thickness 2.
2. Draw circle with radius 6 at position 40,40 with line thickness 1.
3. Draw circle with radius 3 at position 40,40 filled black.
4. Draw circle with radius 6 at position 60,40 with line thickness 1.
5. Draw circle with radius 3 at position 60,40 filled black.
6. Draw red line from position 30,70 to position 70,70 with line thickness 1.

In this example, the patterns across instructions should be fairly simple to spot. The emergence of a pattern provides an opportunity for improvement. Instead of them being disjointed and separated, those parts making up the pattern can usually be brought

together and solved using a common approach. In other words, you can take some separate, but similar concepts, and **generalise** them into a single concept. As a result, your solution becomes simpler because it contains fewer distinct concepts, and it becomes more powerful because you can reuse it in other situations and solutions.

Recognising simple patterns

Here's one approach to spotting simple patterns:

- Look for nouns that appear repeatedly. These could correspond to objects that your solution deals with.
- Look for verbs that appear repeatedly. These could be operations that the solution carries out.
- Look for concrete descriptions. These could probably be substituted by placeholders that vary in different situations. For example:
  - adjectives ('red', 'long', 'smooth') which indicate properties of things and could be replaced by the property name (colour, size, texture);
  - actual numbers, which could be replaced with variables.

Let's illustrate all this by examining the algorithm above. What patterns can be seen?

- nouns like circle, radius, position, line, thickness;
- verbs like draw;
- adjectives like red, black, filled;
- numbers in several places.

Looking at the **nouns**, we see that some of them translate into objects (circle and line), while the rest are **properties** of those objects (radius, position, line thickness). Line and circle are specific instances of shapes. Hence, 'shape' is a generalisation of line and circle.

The **verbs** are straightforward – there's only one. Drawing appears to be a fundamental operation in this solution. Operations often need supplying with things to work with. Data given to an operation when it is triggered is called a **parameter**. In this case, drawing doesn't make any sense without having a shape to draw, so the draw operation should be given a shape parameter.

The **adjectives** red and black are specific cases of the more general concept of colour. The word 'filled', being derived from a verb, actually suggests another operation which we could formalise: filling a shape with a colour. Such an operation would then have two parameters.

All the **numbers** match up with object properties. In other words, these properties (position, radius, thickness) have values that can vary.

These are the generalisations extracted (if an operation expects parameters, they are listed in parentheses after the operation's name):

- **Draw (Shape)**: an operation that renders a shape.
- **Fill (Shape, Colour)**: an operation that makes the interior of a shape solid with a colour.
- **Shape**: a form with an external boundary. Specific shapes include:
  - **Circle**: a type of shape with a radius, position and a line thickness.
  - **Line**: a type of shape with two positions (aka coordinates) and a line thickness.
- **Colour**: red, blue, green and so on.

To summarise what has been done so far: we took a concrete solution, looked for patterns in it and simplified them. The original solution was a specialisation, capable only of drawing a specific shape. The generalisations taken from it form the core concepts of not only the original solution but any solution that involves breaking down a complex image into simple parts.

The solution is not perfect; it could benefit from improvements. For example, line thickness is not, mathematically speaking, a property of a shape. What's more, in our example, line thickness doesn't vary much between individual shapes. We could improve this by introducing another concept called 'Pen', which is responsible for the appearance of the shapes' boundaries. Line thickness would instead belong to the Pen, and it could be changed in-between drawing each shape, if need be.

More complex patterns

Looking for larger and more complex patterns requires you to expand your scope, because you'll need to consider whole groups of instructions at once. It can be harder and will take some experience getting used to, but it's worth it because you can put those patterns to work in powerful ways.

In brief, the guidelines for some of the more complex patterns are:

- Patterns among a sequence of instructions can be generalised into **loops**.
- Patterns among separate groups of instructions can be generalised into **subroutines**.
- Patterns among conditionals or equations can be generalised into **rules**.

Loops

Chapter 2 introduced us to the concept of looping, that is, executing the same series of steps repeatedly. This is a type of generalisation. In order to roll a sequence of steps up into a loop, you need to look at their specific forms and then derive one general form. A block of instructions that appear very repetitive usually gives the game away.

For example, consider this block:

- Draw circle with radius 6 at position 40,40
- Draw circle with radius 3 at position 40,40 filled black
- Draw circle with radius 6 at position 60,40
- Draw circle with radius 3 at position 60,40 filled black

We see a pattern here. Together, the last two steps essentially repeat the first two, only with a slight difference in positioning. We could therefore put these steps into a loop.

- Let coordinates = (40,40), (60,40)
- For each coordinate x,y do the following:
  - Draw circle with radius 6 at position x,y
  - Draw circle with radius 3 at position x,y filled black

You should remember from Chapter 2 how the execution of loops is often controlled by a variable, which encapsulates the thing that varies from iteration to iteration. In this example, the steps in the loop should be executed for each pair of coordinates. So, we set up a list of coordinates in the first line[19] and use that as the controlling variable of the loop. Put into words, the loop essentially says, 'for each coordinate carry out the following steps, where the coordinate values in the current iteration are represented by x and y'. In the first iteration, x = 40 and y = 40. In the second iteration, x = 60 and y = 40.

This generalisation makes the solution more flexible. If you want to draw more eyes in different places, you simply add more coordinates to the list and the loop takes care of the rest. Alternatively, you could draw a cyclops by lopping off a coordinate.

Subroutines

The next type of generalisation concerns patterns among separate blocks of instructions. The two steps inside the example loop encapsulate our approach to drawing an eye. They were put into a loop because the solution just happens to draw a pair of eyes, one after the other.

Imagine that we're drawing several faces. In this case, those two steps would be repeated in several different places. But then imagine you wished to alter the way you draw eyes, maybe by giving the iris a larger radius. You would have to update the steps in all those different places.

Instead of that, we could apply a generalisation that not only keeps the definition of an eye in one place, but also saves time when first writing the solution.

As a reminder, a concrete example of drawing an eye looks like this:

- Draw circle with radius 6 at position 40,40
- Draw circle with radius 3 at position 40,40 filled black

A generalised version might therefore look like this:

- Draw circle with radius r1 at position x,y
- Draw circle with radius r2 at position x,y filled black

We could then declare that these steps constitute a **subroutine**, that is to say, a sequence of instructions that perform a distinct, often-invoked task and which can therefore be 'packaged' as a unit. It would look something like this:

- 'Draw eye' is a subroutine (r1, r2, x,y):
  - Draw circle with radius r1 at position x,y
  - Draw circle with radius r2 at position x,y filled black

As per convention, any parameters supplied to a subroutine are declared in parentheses after its name. In this case, drawing an eye requires a pair of coordinates (x and y) and two radii (one each for the outer and inner circle of the eye).

As with the loop, the instructions contained inside the subroutine are indented for clarity. Now, whenever you want to draw an eye, you simply call on your subroutine to do the work for you (remembering to fill in the information it needs!):

- Call 'draw eye' with parameters r1 = 6, r2 = 3, x = 40, y = 40
- Call 'draw eye' with parameters r1 = 6, r2 = 3, x = 60, y = 40
- Call 'draw eye' with parameters r1 = 4, r2 = 2, x = 240, y = 40
- Call 'draw eye' with parameters r1 = 4, r2 = 2, x = 250, y = 40

Rules

The drawing routines defined above contain a pattern I've not yet picked out, although you might have spotted it already. When drawing an eye, the radius of the inner circle is always half the radius of the outer circle. If that is indeed the rule, then we can encode it explicitly into the subroutine so it becomes:

- 'Draw eye' is a subroutine (r, x, y):
  - Draw circle with radius r at position x,y
  - Draw circle with radius ½r at position x,y filled black

In addition to equations, conditionals are a good identifier of rules. Rules are normally indicated by words like 'when' and 'unless', as well as patterns like 'if–then'. For example, 'if a circle is not filled with a colour, then it takes on the background colour' – this is a rule that has been implicit in drawing every circle so far.

SUMMARY

Decomposition and generalisation are related. Whereas decomposition involves breaking the problem down into smaller parts, generalisation involves combining those

smaller parts. These actions are not the inverse of one another; when you generalise the individual parts, you're not putting the problem together again in the same way as before.[20]

The point of generalisation is to look at the decomposed parts and find ways to make the solution easier to handle and more widely applicable to similar problems. In the example above, instead of having numerous parts which can each draw only one type of eye, there is one generic part that can draw many different types of eye.

EXERCISES

EXERCISE 1

Mark the following statements as true or false:

A. Your goal defines how the problem should be solved, not what needs to be done.
B. It is inadvisable to begin writing a solution before the goal is defined.
C. For any non-trivial problem, there is likely only one solution.
D. Decomposition guarantees an optimal solution.
E. A tree structure is hierarchical in nature.
F. All nodes in a tree structure have one or more child nodes.
G. Patterns among separate groups of instructions can be generalised into subroutines.

EXERCISE 2

You're planning to hold a birthday picnic for a child and her friends. Break down the preparation into a tree structure of tasks. The facts are:

A. You need to send out the invitations to the parents of the other children.
B. Food you'll provide: sandwiches (ham, chicken, and cheese), homemade cake.
C. Fresh ingredients (meat and dairy) need to purchased on the day.
D. Other things you need to take: disposable cutlery, blankets, games.
E. The park where the picnic is held requires you to reserve a spot.
F. All guests will get a goody bag with sweets when they leave.

EXERCISE 3

Update the drawing of the smiley face discussed earlier in the chapter so that the positioning of the features (eyes and mouth) are calculated automatically based on the positioning of the face.

EXERCISE 4

Further update the drawing of the smiley face so that:

A. All features have colour (for example, skin colour, red lips, brown eyes, etc.).
B. The cheek has a scar.
C. A round, red nose that partially obscures the eyes is included.

EXERCISE 5

You can group animals by their shared characteristics into a hierarchical tree structure (like the example with Shakespeare's plays). Consider these animals: bat, crocodile, fox, human, octopus, ostrich, penguin, shark, snake, swan, turtle. Group them into three different tree structures by:

A. number of legs;
B. whether they can fly;
C. their class (mammal, fish, etc.).

Imagine these animals were used in a 'Twenty Questions' style game where your opponent thinks of an animal and you may ask them a limited number of yes/no questions to work out which animal they're thinking of. Use your tree structures to guide your questioning strategy. Which one of the three structures would minimise the number of questions you would have to ask? Try dividing groupings into sub-groupings to see if that helps to reduce the number of questions.