9

Model Structuring: The Enigma Cipher

9.1 Introduction

The elements of object-oriented modelling in VDM++ have now been introduced and the focus can move from theory to practice. Many questions about the use of this technology will already have occurred to the reader. What do "real" models look like? How does modelling relate to testing? How does one use the formal models of data and function (in VDM++) alongside class diagrams in UML? And what are the effects of applying this on projects?

The next three chapters aim to address these questions by means of three case studies. The key theme is that of structuring as a way of dealing with complexity and scale. The study of the Enigma cipher machine in this chapter shows how the object-oriented structure of a model reflects that of the problem it is used to analyse and that this pays off in the ability to take a structured approach to testing. Chapter 10 discusses in greater depth the way UML class diagram models work with VDM++ descriptions of data and functionality. Chapter 11 brings these themes together by telling the story of a commercial application in which UML models derived from an enterprise architecture combine with detailed VDM++ models of data and functionality, and with systematic testing, to yield benefits in terms of product quality and development cost.

This chapter shows the development of a VDM++ model of the famous Enigma cipher machine used by the Germans in the Second World War to encrypt and decrypt messages that were exchanged between military units. The purpose of the model is to get a basic understanding of the cipher mechanism as implemented in Enigma. It is often difficult to apply a new technique for the first time, so the model will be presented in some detail, with a careful explanation of each modelling step and the rationale of many of the design decisions taken during its construction. First (Section 9.4) a UML model will be suggested, revised and restructured before the data and function models in VDM++ are presented in greater detail. After building the model, its validity will be analysed using structured testing techniques in Section 9.5.

Simon Singh's popular scientific work *The Code Book* ([Singh99]), Robert Harris's novel *Enigma* ([Harris95]) and the related major motion picture have certainly raised interest in Enigma recently. In spite of Enigma's fame, there are relatively few

attempts at modelling the machine in the literature, making it an intriguing example for this chapter.

9.2 The Historic Significance of Enigma

The Enigma cipher machine, Enigma for short, played a significant role in modern history. The outcome of the Second World War was certainly influenced by the capabilities of the Allies to crack Enigma (and later Lorentz) ciphers at Bletchley Park, where special machines were designed to automate the code-breaking process. The first generation of those machines were christened *Bombe* and they were built using electromagnetic relays. Anecdotal evidence says that the name was inspired by the ticking noise the relays made during its operation. The first Bombe (called *Victory*) was installed in March 1940 at Bletchley Park, and it was used to crack Enigma-encoded messages. A few months later, some 50 machines were working in parallel to process all intercepted messages.

With the advent of the much more complex Lorentz ciphers later in the war, the capabilities of the Bombe were deemed insufficient and the mathematicians and engineers at Bletchley Park designed a new system based on electronic valves, which was cutting-edge technology at the time. The project was headed by Max Newman, and the chief consultant was mathematician Alan Turing, who also played a major role in designing the Bombe. No fewer than 1500 vacuum tubes were needed to build it, giving rise to its name: *Colossus*. The Colossus was installed at Bletchley Park in December 1943, and it could crack on average 300 messages per month. The number of intercepted Lorentz messages nevertheless increased substantially and the performance of Colossus needed to be improved. A new project was set up to deliver the Colossus Mark-II on June 1, 1944 – only five days before D-Day. The Colossus Mark-II was five times faster than its predecessor. It consisted of 2500 valves and could be programmed; it was one of the first programmable and electronic computers.

After the war, the British government in particular did not wish to disclose its code-breaking capabilities. This is the main reason why ENIAC is generally considered to be the world's first electronic computer, but in fact Colossus was operational about a year earlier. Alan Turing developed most of his theoretical insights in computing at Cambridge in the years before the war. His work on undecidability and the universal Turing machine are now famous. At Bletchley Park he succeeded in applying these theoretical results to break the Enigma code. When security restrictions were lifted in the early 1970s and the significance of the results from Bletchley Park was realised, credit was finally given to those who had earned it. Turing is now widely acclaimed as the founding father of computing science, but he never personally enjoyed that status. Historians believe that he committed suicide in 1954 because he was tried, convicted and openly humiliated for being homosexual. Andrew Hodges's biography *Alan Turing: The Enigma* ([Hodges92]; see also `http://www.turing.org.uk`) provides more insight into the life and work of this brilliant scientist.

9.3 The Enigma Cipher Algorithm

The Enigma cipher machine was designed and developed by Arthur Scherbius and Richard Ritter in 1918. These German inventors created an electronic equivalent of the cipher disk invented by Leon Alberti around 1400. The cipher disk is a simple device that substitutes letters. The earliest substitution cipher is known as the Caesar shift cipher (or Caesar cipher), where each letter is simply replaced by a letter that is a certain distance further along in the alphabet. For example (with an encoding distance of 4) A will be represented by E, B by F, C by G and so on.

The cipher disk of Alberti can be used for the encoding and decoding of messages. The cipher disk actually contains two disks, an inner and an outer disk, both containing 26 positions, each representing a letter in the alphabet. By rotating the inner disk against the outer disk the encoding distance can be set. If A on the inner ring is set adjacent to E on the outer ring, then all other letters will be automatically aligned properly. The encoding (and decoding) of a message is now a simple task that can be performed, for example, directly on the battlefield.

Exercise 9.1 Consider a monoalphabetic cipher such as the Caesar cipher. What is the easiest way to prevent the opponent from guessing the encoding distance? What is the principle weakness of monoalphabetic ciphers? □

The encryption is obviously very weak; you simply have to guess the encoding distance, and there are only 25 possibilities. This is typical for so-called monoalphabetic ciphers. However, Alberti used the same device to make code breaking much more difficult. He used a code word to implement the encoding distance. Take, for example, his own first name, "LEON." Instead of taking a fixed distance, the encoding distance is now determined by the order of the letters in the code word. To encode the first letter, the inner disk is set such that the letter L on the inner ring faces A on the outer ring. Encoding the letter N will now yield C. To encode the second letter, the inner disk is reset such that E (the second letter of the code word) faces A. Encoding the letter H will now yield D. This process is repeated for each letter; the fifth letter is again encoded with the first letter of the code word and so on. This so-called polyalphabetic cipher is much harder to break because for every letter another encoding distance is selected. This technique is also known as the Vigenère cipher. An elegant and abstract specification of a polyalphabetic cypher in VDM–SL can be found in [Jones90] (pages 179–81).

These basic techniques were actually used to design the Enigma. Scherbius and Ritter combined several strategies to improve the strength of the cipher, creating a device that was robust, reliable and very easy to use. The cryptographic power of the Enigma is in sharp contrast with the simple means needed to build such an ingenious device. Scherbius was awarded a patent on Enigma in 1918, but still it took him several years to sell it to the German military. Eventually some 30,000 devices were ordered by the German armed forces. Most notable was the use of Enigma in the German navy where Admiral Dönitz, commander of the U-boat fleet, ordered a special version with even greater cipher strength than the standard Wehrmacht Enigma.

The Enigma cipher machine is basically a typewriter composed of three parts: the keyboard to enter the plain text, the encryption device and a display to show the cipher text. Both the keyboard and the display consist of 26 elements, one for each letter in the alphabet. White space in the plain text was simply ignored, which incidentally makes code breaking even more difficult because word boundaries are removed. The operator, typically a signals person, such as a Marconist aboard a ship, configured the Enigma system once per day and encoded the message by typing each letter and writing down the letter highlighted on the display. The encoded message was then sent using normal Morse code by radio. Of course, the enemy could also receive the Morse signal, but they could not understand its contents without cracking the Enigma code. The intended receiver could simply decode the message by repeating the encoding process on the cipher text. Enigma was configured in the same way as the sender had and the message was decoded by typing in each encoded letter and reading the decoded letter from the display. Having the same procedure for encoding and decoding made the machine very easy to use; errors were seldom made.

The encryption device in the Enigma consists again of three parts: a plugboard, a set of rotors and the reflector. The plugboard is used to create a fixed mapping in which each letter can be *replaced* by any other letter. The board is configured by manually inserting patch cables. The most important part of the Enigma is the so-called rotor, also called the scrambler disk or scrambler. The standard Enigma had three rotor slots (places where rotors could be inserted in the system) and five scrambler disks; the naval version had four rotor slots and eight scrambler disks. The scrambler disks were universal and could be inserted in any Enigma rotor slot, creating a vast number of possible Enigma configurations. A scrambler disk actually consists of two disks with 26 electrical contacts each. Every contact point on disk 1 is electrically wired to a specific contact point on disk 2, creating a fixed mapping. The rotor slots were organised such that the contact points on disk 2 touched the contact points of disk 1 on the next scrambler disk. In this way, electrical current could flow from the plugboard, through all rotor disks toward the reflector. The cryptographic strength of Enigma was increased further by turning the scrambler disks, so that each time a different mapping is used when a letter is encoded (or decoded).

The ingenuity in the design is particularly evident in the way the rotors operate. The right-most rotor, which is closest to the plugboard, is turned whenever a key is pressed on the keyboard. Each rotor has a latch at one of the 26 positions, for each rotor a different position. If the latch is adjacent to the latch of the disk on the left, that rotor will also proceed to the next position (just like the old analog odometer in a car). Whether a rotor moves to the next position at each keystroke is dependent on which rotor is in which slot, the position of the latch on each rotor and the start positions of the rotors.

The final piece of the encryption device is the reflector. It is actually responsible for the symmetric (reciprocal) nature of Enigma; encoding and decoding are identical processes. The reflector is placed at the end of the rotor chain. The principal difference between the rotor and the reflector is that the reflector consists of only one disk with 26 electrical contacts, which are all interconnected in pairs. This means that the electric current coming from the last scrambler disk is fed back to that same disk but

on a different position. So the electronic current flows twice through the set of scrambler disks before it reaches the display (seven substitutions for standard Enigma, nine for the naval version). The standard Enigma has a fixed-position reflector, the naval version has an adjustable reflector; it can be set in 26 positions, again to increase the cipher strength.

Enigma was deemed impenetrable. The code was eventually broken due to a combination of luck, weaknesses in the Enigma design (in particular the reflector), wrong use of the code words by the Germans and the endurance and brilliant analysis of the mathematicians and linguists at Bletchley Park. The historic relevance of the Enigma and the people trying to break it can be investigated further, for example, at `http://www.bletchleypark.co.uk`, which also has an on line Enigma simulator. Other Enigma resources are `http://www.bletchleypark.net` and the Web sites of the Deutsches Museum at Bonn (Germany) and the National Cryptologic Museum near Washington, D.C. (USA). Last, but not least, the Web site of Tony Sale: `http://www.codesandciphers.org.uk`. He is the curator of the Bletchley Park Museum and his Web site provides an excellent overview of the Enigma and the machines that were developed to break its code.

9.4 Building the Enigma Model

The information provided in the previous section is sufficient to start modeling Enigma using VDM++. The approach proposed in Chapter 2 will be used by applying the guidelines to the problem. The first guideline is to determine the *purpose of the model*. It is important to gain a *conceptual understanding* of the Enigma device and in particular to investigate the *encryption algorithm* used. The model should reflect the structure of the Enigma device, so that it is possible to investigate the behaviour of each component in isolation as well as the behaviour of a (sub)system that is composed of several components. VDM++ is well suited for describing these kind of architectures.

Interaction with the model is essential to reach these goals. Syntax and type checking can be used to increase confidence in the internal consistency of the model, while prototyping and testing are techniques to validate that the model actually reflects reality. Therefore, as a secondary aim, *structured testing* techniques are presented to show how confidence in the model can be increased. Bearing in mind the purpose of the model, it is not appropriate to present a minutely detailed replica of Enigma; the aim is really to better understand how the Enigma cipher works and to see how this can be modelled effectively in VDM++.

The second step is to create a dictionary of candidate classes, data types and operations that can be extracted from the information provided in the previous section. This step actually covers guidelines 2, 3 and 4 presented in Chapter 2. Such a dictionary could appear as follows:

Potential Classes and Types (Nouns)

- *Reflector: configuration (which input on the disk is connected to which output on the same disk), current position (which input is at absolute position 1 when a character is substituted)*
- *Rotor: configuration (which input on disk 1 is connected to which output on disk 2), current position (which input is at absolute position 1 when a character is substituted), latch position (at what input is the latch located on the rotor, in order to propagate the rotation to the adjacent rotor when the latches are at the same absolute position), next_rotor (the next rotor in the chain), reflector (the reflector connected to the last rotor in the chain)*
- *Plugboard: configuration (which character is replaced by which other character and vice versa; substitutions are always in pairs), first_rotor (which rotor is connected to the plugboard)*
- *Enigma: plugboard (top-level component of the system containing a link to the plugboard)*

Potential Operations (Actions)

- *Keystroke: characters are encoded (or decoded) one by one*
- *Substitute: transposition of a character to another character in the alphabet*
- *Encode & Decode: the transposition of a character is dependent on the direction of the information flow (to or from the reflector, respectively)*
- *Rotate: the rotor turns one position, e.g., after each keystroke or when the latch on the rotor is adjacent to the previous rotor in the chain*

An initial model based on the dictionary is constructed following steps 5 and 6 of the guidelines. The result is the UML class diagram shown in Figure 9.1. The corresponding VDM++ specification is not provided here (it can be found on the book's Web site) but the internal consistency of the VDM++ model has been checked using VDMTools. Note that in this initial design, the Keystroke operation accepts and returns characters whereas all Encode, Decode and Substitute operations use natural numbers in their signatures. These numbers represent the position of the character in the alphabet according to some arbitrary ordering scheme (for example "A".."Z"). The transposition relation can be expressed as a mathematical operation on these indices, which is easier than dealing with the individual characters themselves.

All the elements identified in the dictionary seem to be covered in the initial design, but there is some overlap in functionality and data within the proposed classes. This initial design is far from optimal; it needs modification such that a better balance between structure (organisation of the model, also called *architecture*) and functionality is achieved. This is probably the most difficult part of any design process; sometimes it can take several attempts to "get it right." Design patterns (see [Gamma&95]) are often used to structure the model, but deciding which pattern to use remains an experience-based task. It is nevertheless a very important step because wrongly organized models can cause tremendous maintenance problems later in the system life cycle. Also, testability is greatly influenced by the proper organisation of the model. Analysis of the initial model shows that:
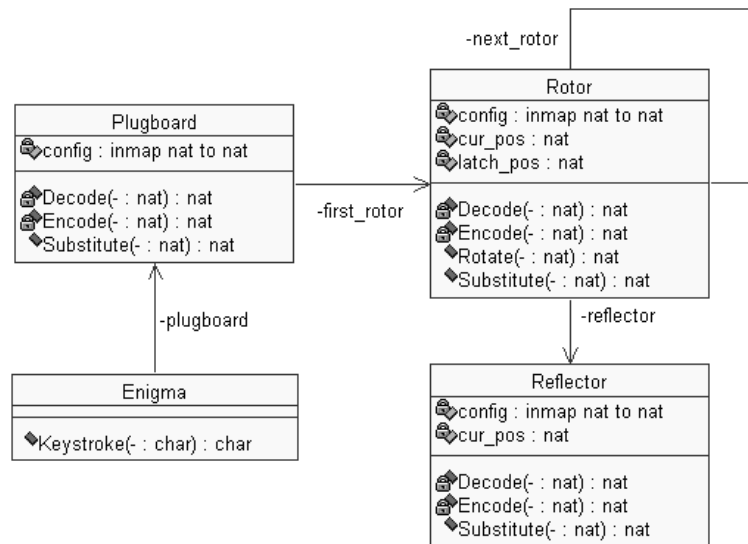
Fig. 9.1: The UML class diagram of the Enigma device – initial design

1. A typical pipe-and-filter architecture (see [Shaw&96]) occurs in the design. Several components are linked in a chain, where the output of a component is used as the input of the next component in the chain. Instead of giving each component a specific relation to the next component in the chain (in the initial design these are `first_rotor`, `next_rotor` and `reflector`), a new class, called `Component`, is provided that captures this notion of chaining through a single instance variable `next`.

2. The `Plugboard`, `Rotor` and `Reflector` classes all contain the operations `Encode`, `Decode` and `Substitute`. Some of these functions are identical for each component, while some have slightly adapted behaviour. The instance variable `config` is also needed in all three classes. It is used to store the way the mapping of indices should occur in each component, but again this is done slightly differently in each case. The instance variable `config` and the operations `Encode`, `Decode` and `Substitute` are ideal candidates for abstraction by inheritance. Therefore, a new class called `Configuration` is introduced that captures this abstraction. This functionality is kept separate from `Component` for two reasons: (1) The top-level class `Enigma` is potentially a subclass of `Component` (to point to the plugboard) but it does not need the configuration information and associated functionality contained in the class `Configuration` and (2) in general, it is wise to keep the structure and functionality separate from each other, for example, to allow reuse of functionality in a different model structure.

The initial design has been modified to achieve a better balance between functionality and structure. Functionality has been grouped so that it is easier to maintain the model because changes do not propagate outside the class boundaries. An overview of the improved model is shown in Figure 9.2. The revised design will be presented one class at a time using the VDM++ notation.
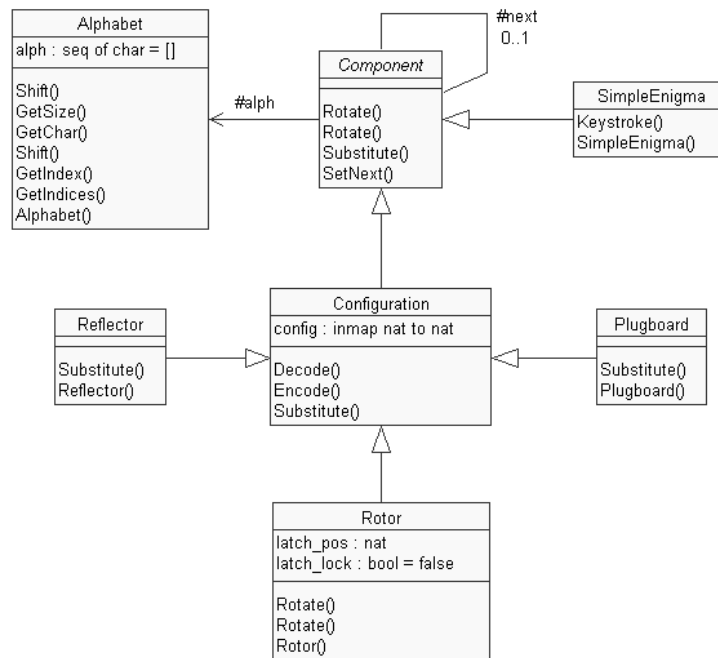
Fig. 9.2: The revised Enigma UML model

9.4.1 Alphabet

The auxiliary class `Alphabet` gathers all the properties and functionality to describe and manipulate the Enigma alphabet. This is important because otherwise this knowledge would be spread around in the rest of the specification, with bits and pieces everywhere. The chances are high that, for example, functionality would be duplicated and the model would be very difficult to maintain.

The alphabet consists of a sequence of characters. The invariant (as required by step 7 from the guidelines) states that the alphabet must have even length and all characters in the sequence must be unique. The even length property is needed by the `Reflector`. Each input on the disk should be connected to exactly one output and

vice versa. This requirement can be met only if an even number of contact points, and
thus an even length alphabet, is used.

```
class Alphabet

instance variables
  alph : seq of char := [];

inv AlphabetInv(alph)

functions
  AlphabetInv: seq of char -> bool
  AlphabetInv (palph) ==
    len palph mod 2 = 0 and
    card elems palph = len palph
```

The Alphabet class provides a constructor for creating a new Alphabet in-
stance and operations GetChar to retrieve the *n*th character from the alphabet,
GetIndex to retrieve the index of the character in the sequence, GetIndices to
return the set of all possible indices, and GetSize to return the size of the alphabet.
These are shown here.

```
operations
  public Alphabet: seq of char ==> Alphabet
  Alphabet (pa) == alph := pa
  pre AlphabetInv(pa);

  public GetChar: nat ==> char
  GetChar (pidx) == return alph(pidx)
  pre pidx in set inds alph;

  public GetIndex: char ==> nat
  GetIndex (pch) ==
    let pidx in set {i | i in set inds alph
                        & alph(i) = pch} in
      return pidx
  pre pch in set elems alph;

  public GetIndices: () ==> set of nat
  GetIndices () == return inds alph;

  public GetSize: () ==> nat
  GetSize () == return len alph;
```

Note that the set comprehension in `GetIndex` is always a singleton set (a set containing just one element) due to the uniqueness property stated in the invariant. Therefore, `pidx` will always be bound to the proper value by the `in set` operator in the *let* statement.

The substitution algorithm mentioned earlier is the core of Enigma encryption and decryption. It relies on the ability to transpose characters in the alphabet. These transpositions are expressed by the overloaded `Shift` operations, which calculate a new index based on the current index and a transposition distance or offset. The second `Shift` operator is defined for convenience, to transpose a character by just one position. It is used to implement the movement of the Rotor, which will be presented later:

```
  public Shift: nat * nat ==> nat
  Shift (pidx, poffset) ==
    if pidx + poffset > len alph
    then return pidx + poffset - len alph
    else return pidx + poffset
  pre pidx in set inds alph and
      poffset <= len alph;

  public Shift: nat ==> nat
  Shift (pidx) == Shift(pidx, 1)
end Alphabet
```

The strength of the `Alphabet` class is that it is possible to express an alphabet of arbitrary length and arbitrary ordering, which provides a lot of freedom to experiment with the Enigma model using different alphabets.

9.4.2 Component

The class `Component` allows construction of a linked list using an instance variable `next` to point to the next component in the list. An operation `SetNext` is used to instantiate the link. The instance variable `next` should not be assigned more than once. Furthermore, loops in the linked list have to be prevented. This can be done in a precondition on the `SetNext` operation (again step 7 from the guidelines), as shown here. Typically for models of such data structures, recursion is used to gather components already linked into the structure:

```
class Component

instance variables
  protected next : [Component] := nil;
  protected alph : Alphabet
```

```
operations
  public Successors: () ==> set of Component
  Successors () ==
    if next = nil
    then return {self}
    else return {self} union next.Successors();

  public SetNext: Component ==> ()
  SetNext (pcom) == next := pcom
    pre next = nil and
        self not in set pcom.Successors();
```

Exercise 9.2 Write an alternative specification for `Successors` that does not use recursion. □

The precondition on `SetNext` prevents loops by testing the `self` object reference against the set of object references retrieved by `Successors`, which uses recursion to compute the set of object references of all components that are already in the linked list. Note that the instance variable `next` is declared `protected`, which means that derived classes can access it without an extra operation. This modelling style is used here because the model is only instantiated once (at startup) and is not modified during the remainder of its lifetime. Furthermore, each `Component` has a reference to the `Alphabet`. The instance variable `alph` needs to be initialized by the derived classes of `Component`.

Exercise 9.3 The operation `SetNext` is defined such that it is not possible to create circular structures, but is the proposed solution here really sufficient? How can it be broken, and what is needed to fix it? □

Each `Component` should be able to perform two basic tasks: `Substitute` and `Rotate`. The `Substitute` operation takes an index, performs the substitution and finally returns the new index. The implementation of this operation is explicitly delegated to the derived classes of `Component` by means of the `is subclass responsibility` construct. The class `Component` is a so-called *abstract base class* because it is not possible to create an instance of `Component` directly, due to the delegated operation `Substitute`.

In contrast, the `Rotate` operations are executable; their default behaviour is to do nothing. Operation overloading will be used to redefine the behaviour for rotating components (in our case the `Rotor`) in the derived classes. The first `Rotate` operation performs immediate rotation, while the second operation takes into account the current position of the latch, which is passed as a parameter. The latter operation is used to implement the latching mechanism of the `Rotor`, which is presented later.

The reason for using `is subclass responsibility` in the case of the operation `Substitute` and `skip` in the case of the operation `Rotate` is that knowl-

edge is lacking at this level in the model to say anything useful about the substitution. In contrast, it is possible to be explicit about the components that do not need to rotate. Nevertheless, having both operations defined here allows us to use the statements `next.Substitute()` and `next.Rotate()` anywhere in the model without the need to know the actual subtype of the object reference stored in the instance variable `next`:

```
  public Substitute: nat ==> nat
  Substitute (-) == is subclass responsibility;

  public Rotate: () ==> ()
  Rotate () == skip;

  public Rotate: nat ==> ()
  Rotate (-) == skip

end Component
```

9.4.3 Configuration

The class `Configuration` is generic. It contains an instance variable `config`, an injective mapping between natural numbers, used to represent the transposition relation in Enigma components. In the rotor, this instance variable will describe which input on disk 1 (a number in the domain of the mapping) is connected to which output on disk 2 (a number in the range of the mapping). In the plugboard, the numbers in the domain and range represent the indices of the characters that are swapped. In the reflector, the numbers describe which input (domain) is connected to which output (range) on the same disk. The differences in the way the mapping is used make it very hard to write a useful invariant at the generic class level. In cases like these, it is better to postpone the definition of an invariant to the derived classes where specific invariants can be given. For that reason, the variable is declared `protected` so that derived classes can refer to it directly:

```
class Configuration
  is subclass of Component

instance variables
  protected config: inmap nat to nat;
```

Now that the transposition relation is defined, it is possible to give a generic definition for the `Substitute` operation. Two auxiliary operations, `Encode` and `Decode`, are defined for this purpose. The operation `Encode` implements the encod-

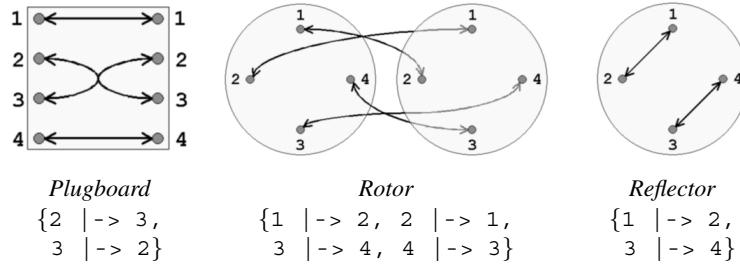| Plugboard	Rotor	Reflector
{2 |-> 3,	{1 |-> 2, 2 |-> 1,	{1 |-> 2,
3 |-> 2}	3 |-> 4, 4 |-> 3}	3 |-> 4}

Fig. 9.3: Example configurations for a four-character Enigma

ing of the index when the data flows from the plugboard toward the reflector, Decode represents the encoding of an index when the data returns from the reflector toward the plugboard. Hence Encode looks at the values in the mapping and Decode uses values from the inverse of the mapping to determine the new index. The operation Substitute first encodes the index, calls Substitute on the next component in the linked list and finally decodes the result of that call and returns the answer to the caller. This nested calling of Substitute goes on until the end of the linked list is reached or a component is encountered that has different behaviour defined for this operation:

```
operations
  protected Encode: nat ==> nat
  Encode (penc) ==
    if penc in set dom config
    then return config(penc)
    else return penc;

  protected Decode: nat ==> nat
  Decode (pdec) ==
    let invcfg = inverse config in
      if pdec in set dom invcfg
      then return invcfg(pdec)
      else return pdec;

  public Substitute: nat ==> nat
  Substitute(pidx) ==
    return Decode(next.Substitute(Encode(pidx)))
  pre next <> nil

end Configuration
```

With the configurations shown in Figure 9.3, `Encode` and `Decode` behaviour is defined in Table 9.1.

Table 9.1: Example `Encode` and `Decode` behaviour

Configuration	Encode	Decode
Plugboard	`Encode(1)  =  1`	`Decode(1)  =  1`
	`Encode(2)  =  3`	`Decode(2)  =  3`
	`Encode(3)  =  2`	`Decode(3)  =  2`
	`Encode(4)  =  4`	`Decode(4)  =  4`
Rotor	`Encode(1)  =  2`	`Decode(1)  =  2`
	`Encode(2)  =  1`	`Decode(2)  =  1`
	`Encode(3)  =  4`	`Decode(3)  =  4`
	`Encode(4)  =  3`	`Decode(4)  =  3`
Reflector	`Encode(1)  =  2`	`Decode(1)  =  2`
	`Encode(2)  =  1`	`Decode(2)  =  1`
	`Encode(3)  =  4`	`Decode(3)  =  4`
	`Encode(4)  =  3`	`Decode(4)  =  3`

Exercise 9.4 Consider the following configurations:

- *Plugboard* = $\{$`1  |-> 3,  3  |-> 1`$\}$
- *Rotor* = $\{$`1  |-> 3,  2  |-> 1,  3  |-> 2,  4  |-> 4`$\}$
- *Reflector* = $\{$`1  |-> 3,  2  |-> 4`$\}$

Calculate the results for `Encode(x)` and `Decode(x)` where
`x in set` $\{$`1,...,4`$\}$. $\square$

9.4.4 Reflector

The class `Configuration` is sufficient for constructing the models for all Enigma components. The `Reflector` is an interesting case because it is at the end of the linked list, so the `next` member variable should be `nil`. Writing the invariant for the `config` member variable is more challenging. Because inputs and outputs of the reflector are on the same physical disk, each position must be an element of either the domain or the range of the mapping (the intersection of domain and range must be empty). Furthermore, the union of the domain and range must be identical to the indices of the alphabet, to ensure that all positions are accounted for. The injective property of the mapping in combination with the previous invariants now guarantees that all positions are connected in pairs. Note that no member variables have been added – only the invariant on the instance variables has been defined:

```
class Reflector
  is subclass of Configuration

instance variables
  inv ReflectorInv(next, config, alph)

functions
  ReflectorInv:
    [Component] * inmap nat to nat * Alphabet -> bool
  ReflectorInv (pnext, pconfig, palph) ==
    pnext = nil and
    dom pconfig inter rng pconfig = {} and
    dom pconfig union rng pconfig = palph.GetIndices()
```

The `Reflector` class contains three state components: the instance variables `next` (from `Component`), `config` and `alph` (from `Configuration`). Note that the last line of the invariant refers to two member variables in a single boolean expression. Care will have to be taken when updating these variables to ensure that the invariant is not invalidated. The effect of this can be seen in the `Reflector` constructor. Suppose the member variable `config` is initialised first and `alph` second. Evaluating the invariant after the first assignment will cause a run-time error because `alph` is not yet initialised. Reversing the assignment order will not solve this problem. In this sort of situation, the `atomic` statement that was introduced in Section 7.3 is used around a block statement. All statements inside the block statement are evaluated as if they are a single (atomic) statement and the invariants are evaluated only at the end of the block.

The starting position of the reflector is set only once (when Enigma is initialised) so there is no need to store that value. Instead, the mapping `pcfg` is changed such that it reflects the starting position when `config` is initialised. A map comprehension is used to modify the mapping. The variable `i` iterates over all values in the domain of the original mapping `pcfg`. This value is used to construct a new mapping where both the domain value (`i`) and the range value (`pcfg(i)`) are updated using the `Shift` operator of alphabet `pa`. Consider the `Reflector` configuration {1 |-> 2, 3 |-> 4} from Figure 9.3. This configuration is changed depending on the start position; the result is shown in Table 9.2.

```
operations
  public Reflector:
    nat * Alphabet * inmap nat to nat ==> Reflector
  Reflector (psp, pa, pcfg) ==
    atomic (alph := pa;
      config := {pa.Shift(i, psp-1) |->
        pa.Shift(pcfg(i), psp-1) |
```

```
        i in set dom pcfg})
 pre psp in set pa.GetIndices() and
      ReflectorInv(next, pcfg, pa);
```

Table 9.2: The start position determines the configuration

Start position	Transposition distance	Configuration
1	0	{1 |-> 2, 3 |-> 4}
2	1	{2 |-> 3, 4 |-> 1}
3	2	{3 |-> 4, 1 |-> 2}
4	3	{4 |-> 1, 2 |-> 3}

Exercise 9.5 Consider the following `Reflector` configuration for a six-character alphabet: {1 |-> 4, 2 |-> 6, 3 |-> 3, 4 |-> 2, 5 |-> 5, 6 |-> 1}. Calculate the new configuration for the transposition distance 2. □

Finally, the operation `Substitute` needs to be redefined because the reflector is at the end of the linked list. The default behaviour (defined in `Component`) would cause a run-time error because `next.Substitute()` does not exist:

```
  public Substitute: nat ==> nat
  Substitute (pidx) ==
    if pidx in set dom config
    then Encode(pidx)
    else Decode(pidx)

end Reflector
```

9.4.5 Rotor

The cryptographic strength of Enigma is mainly due to the rotor. This device ensures that each character is encoded (information flows from plugboard to reflector) using a different transposition relation. This is achieved by turning the rotor one step before the character is substituted. Note that the rotor is not turned when the character is decoded (information flows from the reflector to the plugboard), the identical configuration is used for that. Whether the rotor turns is dependent on the position of the rotor in the linked list and the position of the latch on the rotor. The rotor nearest to the plugboard is turned for each character, independent of its latch position. The second rotor only turns if its latch is at the same position as that of the first rotor. If this is the

case, the second rotor turns to its next position and then waits until the first disk has made a full rotation before turning again one step. This algorithm applies to each pair of adjacent rotors. It resembles the way that old analog odometers in cars work. Each disk has ten positions, numbered 0 to 9. If the rightmost disk moves from 9 to 0, it will also move the adjacent disk on the left to the next position and so on.

The VDM++ class `Rotor` models this behavior. Two new instance variables, `latch_pos` and `latch_lock`, are introduced to model the latch. The instance variable `latch_pos` is used to store the position of the latch on the disk. Both disks are turned one step when the latch positions of two adjacent rotors match. Note that the latches are then again in the identical position. The status is stored in the boolean variable `latch_lock` to prevent the rotor from turning instead of waiting for the adjacent disk to have made a full rotation. The rotor is idle if `latch_lock` is `true` and the latches are adjacent, otherwise it turns:

```
class Rotor
  is subclass of Configuration

instance variables
  latch_pos : nat;
  latch_lock : bool := false;

inv RotorInv(latch_pos, config, alph)

functions
  RotorInv: nat * inmap nat to nat * Alphabet -> bool
  RotorInv (platch_pos, pconfig, palph) ==
    let ainds = palph.GetIndices() in
      platch_pos in set ainds and
      dom pconfig = ainds and
      rng pconfig = ainds and
      exists x in set dom pconfig & x <> pconfig(x)
```

The invariant states that all the possible positions (represented by the indices of all characters in the alphabet) occur in both the domain and the range of `config`. Also note that "straight through" connections (e.g., input 1 is connected to output 1) are allowed, but the existential quantification is added to disallow rotors that have no effect (when all connections are "straight through" connections). At least one connection must have different input and output indices.

The constructor again must ensure that the instance variables are instantiated in a manner consistent with the invariant. It is defined as follows:

```
operations
  public Rotor:
    nat * nat * Alphabet * inmap nat to nat ==> Rotor
  Rotor (psp, plp, pa, pcfg) ==
    atomic (latch_pos := pa.Shift(plp,psp-1);
      alph := pa;
      config := {pa.Shift(i,psp-1) |->
                 pa.Shift(pcfg(i),psp-1) |
                 i in set dom pcfg})
  pre psp in set pa.GetIndices() and
      RotorInv(plp, pcfg, pa);
```

As with the `Reflector`, the initial position of the `Rotor` is only required now; there is no need to store its value. Both the latch position `latch_pos` and the transposition relation `config` are updated with the start position. Note that the `latch_lock` member variable is already initialised (to `false`) in the instance variable block.

The two overloaded `Rotate` operations that were defined in the base class called `Component` can now be refined. Note that the member variable `next` must be of the proper type – rotors only refer to other rotors or to the reflector. This is tested using the `isofclass` operator. Note that this operator checks the *type* of a variable rather than its value. The operator `isofclass` takes two arguments. The first argument is a class *name*, the second argument is an object reference *expression*:

```
isofclass( name, expression )
```

The operator yields the boolean value `true` if and only if the object reference expression refers to an instance of the same type as class `name` or any of the subclasses of `name`, and `false` otherwise. So, the examples

```
let a = new Alphabet("AB") in
  isofclass(Alphabet, a)

let a = new Alphabet("AB") in
let r = new Reflector(1, a, {1 |-> 2}) in
  isofclass(Component, r)
```

will yield `true`, whereas the examples

```
  let a = new Alphabet("AB") in
    isofclass(Component, a)

  isofclass(Rotor, nil)
```

will yield `false`.

```
  public Rotate: () ==> ()
  Rotate () ==
    (-- propagate the rotation to the next component
     -- and tell it where our latch position is
     next.Rotate(latch_pos);
     -- update our own latch position and take the
     -- alphabet size into account
     if latch_pos = alph.GetSize()
     then latch_pos := 1
     else latch_pos := latch_pos+1;
     -- update the transpositioning relation by
     -- shifting all indices one position
     config := {alph.Shift(i) |->
                 alph.Shift(config(i)) |
                 i in set dom config};
     -- remember the rotation
     latch_lock := true)
  pre isofclass(Rotor,next) or
      isofclass(Reflector,next);

  public Rotate: nat ==> ()
  Rotate (ppos) ==
    -- compare the latch position and the lock
    if ppos = latch_pos and not latch_lock
    -- perform the actual rotation
    then Rotate()
    -- otherwise reset the lock
    else latch_lock := false
  pre ppos in set alph.GetIndices();

end Rotor
```

Notice that the substitution functionality that was modelled in `Configuration` has not changed at all. The substitution and rotation algorithms are kept separate, sharing only the `config` member variable. It makes model maintenance a lot easier

because the functionality is clearly separated. Table 9.3 presents a three-rotor system. Rotor 1 is connected to the plugboard and rotor 3 is connected to the reflector. The member variable `latch_pos` of rotor 1 is changed every time a character is encoded. Rotors 2 and 3 are turned only when they have identical latch positions and the member variable `latch_lock` is false. These situations are illustrated by the "⇒" symbol in Table 9.3.

Table 9.3: A three-rotor system

Rotor 1		Rotor 2		Rotor 3	
pos	lock	pos	lock	pos	lock
2	-	3	false	3	false
⇒ **3**	-	⇒ **3**	false	⇒ **3**	false
4	-	4	true	4	true
1	-	4	false	4	true
2	-	4	false	4	true
3	-	4	false	4	true
⇒ **4**	-	⇒ **4**	false	4	true
1	-	1	true	4	false
2	-	1	false	4	false
3	-	1	false	4	false
4	-	1	false	4	false
⇒ **1**	-	⇒ **1**	false	4	false
2	-	2	true	4	false
3	-	2	false	4	false

9.4.6 Plugboard

The plugboard is used to replace pairs of characters. As with the other components, the injective mapping of indices, `config`, is used to represent this. The `plugboard` is initialised with another injective mapping that contains solely disjoint sets of domain and range elements (note the precondition of the constructor). Each maplet represents a replacement. For example, the mapping {1 |-> 2} implies that the character with index 1 is replaced by the character with index 2. To express the notion of replacement *in pairs*, the mapping is merged with its own inverse. In the example, `config` is {1 |-> 2, 2 |-> 1}. It is sufficient to claim that the domain of `config` shall be a subset of the set of indices of the alphabet, because the domain and range are identical due to the mapping union that was just performed. What happens with indices of the alphabet that do not occur in `dom config`? Recall that the `Encode` and `Decode` operations, which are defined in class `Configuration`, simply return the index if that value does not occur in either the domain or range of the mapping `config`. This behaviour reflects the fact that the character was not swapped by the plugboard:

```
class Plugboard
  is subclass of Configuration

instance variables
  inv PlugboardInv(config, alph)

functions
  PlugboardInv: inmap nat to nat * Alphabet -> bool
  PlugboardInv (pconfig, palph) ==
    dom pconfig subset palph.GetIndices()

operations
  public Plugboard:
    Alphabet * inmap nat to nat ==> Plugboard
  Plugboard (pa, pcfg) ==
    atomic (alph := pa;
      config := pcfg munion inverse pcfg)
  pre dom pcfg inter rng pcfg = {} and
      PlugboardInv(pcfg, pa);
```

The `Substitute` operation of the plugboard is modified because the first rotor
in the linked list needs to be rotated before the transposition actually takes place.
Also note that the member variable `next` needs to be of the correct type (`Rotor` or
`Reflector`):

```
  public Substitute: nat ==> nat
  Substitute (pidx) ==
    (next.Rotate();
     Configuration`Substitute(pidx))
  pre pidx in set alph.GetIndices() and
      (isofclass(Rotor,next) or
       isofclass(Reflector,next))

end Plugboard
```

The operation `Substitute` is redefined for the third time! First it was defined in
the abstract base class `Component`, then it was redefined in the `Configuration`
class and now again in the class `Plugboard`. Although this is allowed in VDM++
it is not always easy to grasp the consequences of the redefinition. For example, not
only the behaviour specifications but also pre- and postconditions are *implicitly* re-
defined. In Figure 9.4, the inheritance relations for the `Substitute` operation are
shown. Note that the operation in class `Reflector` relaxes the precondition that

was specified in the base class. In contrast, the operation in class `Plugboard` makes the precondition much stronger because it also enforces properties on the parameter `pidx`. The original precondition is again implicitly enforced because the operation `Configuration`Substitute` is called inside the body of the operation. Note that the class `Rotor`, which is not shown in the figure, has equivalent behaviour to the class `Configuration`.
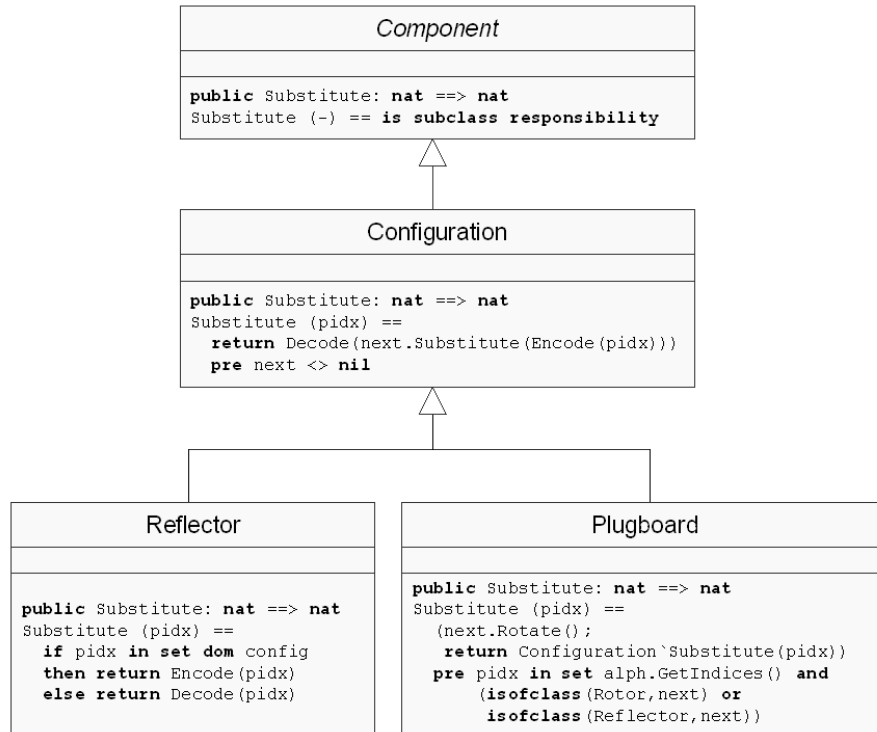
```
                            Component

          public Substitute: nat ==> nat
          Substitute (-) == is subclass responsibility
```
```
                          Configuration

          public Substitute: nat ==> nat
          Substitute (pidx) ==
            return Decode(next.Substitute(Encode(pidx)))
            pre next <> nil
```
```
            Reflector                          Plugboard

                                    public Substitute: nat ==> nat
  public Substitute: nat ==> nat    Substitute (pidx) ==
  Substitute (pidx) ==                (next.Rotate();
    if pidx in set dom config         return Configuration`Substitute(pidx))
    then return Encode(pidx)         pre pidx in set alph.GetIndices() and
    else return Decode(pidx)             (isofclass(Rotor,next) or
                                          isofclass(Reflector,next))
```

Fig. 9.4: Redefinition of the `Substitute` operation

9.4.7 Enigma

All the basic Enigma components have been modeled in the previous subsections and a simple model of the Enigma device is presented here that works on an alphabet of four characters configured as illustrated in Figure 9.3. First some values are defined that are used to initialise these components. The value `refcfg` is used for the configuration of the reflector, `rotcfg` is the configuration of the rotors and finally `pbcfg` is the configuration of the plugboard:

```
class SimpleEnigma
  is subclass of Component

values
   refcfg : inmap nat to nat =
     {1 |-> 3, 2 |-> 4};
   rotcfg : inmap nat to nat =
     {1 |-> 2, 2 |-> 4, 3 |-> 3, 4 |-> 1};
   pbcfg : inmap nat to nat =
     {2 |-> 3}
```

The linked list of components is constructed in reverse order, last element first. The Reflector is created with the new operator and the initial position; the alphabet and configuration are passed as parameters to the constructor. Note that three rotors are constructed that have identical configurations but the initial position and the position of the latch are different for each rotor:

```
operations
  public SimpleEnigma: () ==> SimpleEnigma
  SimpleEnigma () ==
    (dcl cp : Component ;
     alph := new Alphabet("ABCD");
     next := new Reflector(4,alph,refcfg);
     cp := new Rotor(3,3,alph,rotcfg);
     cp.SetNext(next);
     next := cp;
     cp := new Rotor(2,2,alph,rotcfg);
     cp.SetNext(next);
     next := cp;
     cp := new Rotor(1,1,alph,rotcfg);
     cp.SetNext(next);
     next := cp;
     cp := new Plugboard(alph,pbcfg);
     cp.SetNext(next);
     next := cp);
```

The operation Keystroke is the only visible functionality of the SimpleEnigma device. A character is inserted and the encrypted character is returned:

```
  public Keystroke : char ==> char
  Keystroke (pch) ==
    let pidx = alph.GetIndex(pch) in
      return alph.GetChar(next.Substitute(pidx))
  pre isofclass(Plugboard,next)

end SimpleEnigma
```

The interpreter of VDMTools can now be used to observe the behaviour of the completed model. A typical interactive session could look like this:

```
Initializing specification ... done
>> create a := new SimpleEnigma()
>> print a.Keystroke('A')
'D'
>> print a.Keystroke('C')
'A'
>> print a.Keystroke('C')
'A'
>> print a.Keystroke('C')
'D'
```

The next step is to analyse the model and determine its validity. Some confidence in internal consistency has been gained using the VDMToolssyntax checker and type checker. It is now time to test the model systematically.

9.5 The VDMUnit Framework

The VDMTools interpreter, accessible through the graphical user interface, is well-suited for prototyping. For example, it can be used to explore alternative modeling strategies and to try out new ideas interactively while the model is being constructed. The syntax and type checkers provide feedback on the internal consistency of the model, while the interpreter is used to validate parts of the model, as was shown in the small example session at the end of the previous section. This way of working is very powerful because it gives the modeller immediate feedback on design decisions and forces the modeller to consider the "big picture" – the interactions between functions and operations across the model as a whole. Weaknesses are often spotted when executing the model; they can be corrected at this stage of the development process at relatively low cost.

The purpose of model interaction is to answer the informal question: Does the model work? Unfortunately, it is not possible to answer "yes" to this question in general. First, it is impossible to prove that the set of requirements that describe the proper

operation of the system is complete. There is always a risk of underspecification. A famous example is a military radar system developed in the 1970s to detect incoming ballistic missiles. When the system was first turned on after an elaborate test process, alarms went off that indicated that an enemy missile was fired when nothing had happened. Missile-tracking radars had become so powerful that the moon actually reflected the radar waves. The system alarm was triggered by the moon rising above the horizon. The requirement to filter out the radar reflection from the moon had been completely overlooked by the designers. Second, even if a complete set of requirements could exist, it is impossible to prove (due to Turing's undecidability theory and the so-called halting problem) in general that a computer program will terminate for any sequence of input events. In summary, providing a proof to the consistency and completeness of a model is a *provable unsolvable* problem! Beizer [Beizer90] summarizes this as follows:

- We can never be sure that the specifications are correct.
- No verification system can verify every correct program.
- We can never be certain that a verification system is correct.

The last case proposed by Beizer hints in particular at mechanical verification (a computer program that implements a certain verification algorithm). It is impossible to prove that the tool is bug-free, even though the verification algorithm used is proven to be sound and complete. How much trust do we have in these tools? How accurate are the results they provide?

Because it is not possible to break the theoretical limit, attention moves from absolute proof to a weaker, but suitably convincing, demonstration. The overall aim is to raise confidence in the model by applying several available analysis techniques, trying to approach the theoretical limit as closely as possible. Model checking and formal proof are two such techniques but they are outside the scope of this book. Extended static analysis and external validation combining a VDM++ model with a GUI will be presented in Chapter 13. In the remainder of this section, *structured testing* will be examined.

Beizer states that the purpose of *testing* is to show that a program has bugs. In contrast, the purpose of *debugging* is to find the error or misconception in the model that led to the failure and to design and implement changes that correct the model. It is important to distinguish these terms because they are often confused. The example at the end of Section 9.4.7 used the interpreter for *debugging* rather than *testing*. Testing is a predefined procedure applied under known conditions that has a predictable outcome; debugging starts from a possibly unknown initial condition and often has an unpredictable outcome. The purpose of structured testing is to assess the quality of the model at different levels of abstraction. Beizer defines four significant levels:

Unit Testing: A *unit* is the smallest testable piece of software that can be put under the control of a so-called *test harness* or *driver*. This test harness implements the predefined testing procedure which controls the initial conditions, executes the test and verifies the outcome. Each VDM++ class can be considered such a *unit*.

Component Testing: A *component* is an aggregate of two or more units or compo-
nents. Component testing aims at analysing functional and structural consistency
of the aggregation of those classes and components.

Integration Testing: Several components are aggregated to create even larger com-
ponents. The difference with component testing is that now consistency among
components is analyzed, for example call and return sequences, data validation
criteria and proper handling of data objects passed between components.

System Testing: Several components are aggregated to form the system. System test-
ing concerns issues that can only be exposed by testing the entire system or a
major part of it, such as startup and error recovery.

The general approach is to test software bottom-up, starting with unit testing, and
to continue upward on the abstraction ladder until the system-level tests are reached
and successfully completed. Testing at a higher level should be started if and only
if all tests on the lower levels have been performed. The model should be modified
if a test fails and then *all* tests should be repeated, starting again at the bottom, to
guarantee that the change did not lead to inconsistencies elsewhere in the model. This
process is called *regression testing*. This shows that testing is a process that should be
easy to perform and repeat. The interpreter included in the graphical user interface of
VDMTools is not well suited for this activity. Although it is possibile to build simple
test scripts (basically a list of interpreter commands that will be executed sequentially),
it is much easier to use the command-line version of VDMTools (see Section 3.4.10)
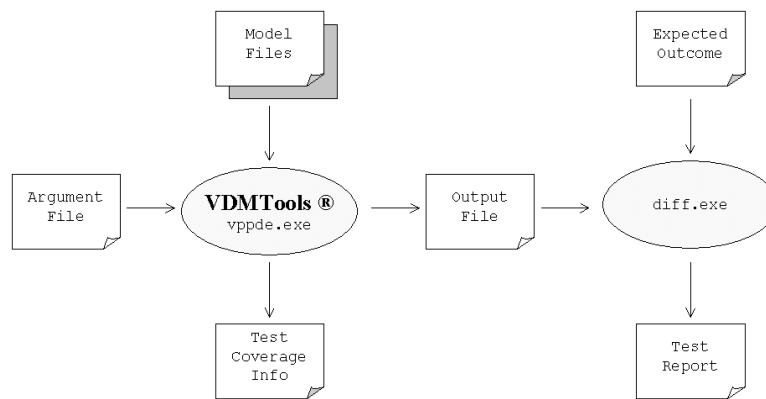in a batch-oriented style. Consider the process flow shown in Figure 9.5.

Fig. 9.5: Testing a model using the command-line version of VDMTools

The argument file contains the test case to be executed; for example, consider the
file `test.arg`, which could read:

```
let str = "ACCC" in
  let en = new SimpleEnigma() in
    [en.Keystroke(str(i)) | i in set inds str]
```

The model files and the argument file are passed to the VDMTools command-line interpreter. Note that the interpreter is invoked with dynamic type checking (-D), invariant (-I), pre- (-P) and postcondition (-Q) checking turned on:

```
vppde -iDIPQ -O test.res test.arg Alphabet.vpp
   Component.vpp Configuration.vpp Plugboard.vpp
   Reflector.vpp Rotor.vpp SimpleEnigma.vpp
```

The file test.res will now yield the following result:

```
"DAAD"
```

The resulting output file can now be compared against a file that contains the expected outcome of the test case, for example, using the standard Unix tool "diff." The expected outcome file is normally written by hand by the tester, often together with the test case. This approach can be easily extended using other standard Unix tools such as "make" and "perl" to construct a *database* of test cases. For example, to maintain overview, a simple directory structure can be used to group test cases for a specific purpose or a specific abstraction level. "make" is used to iterate over all test cases contained in these directories and "perl" is used to create a single test report composed from all individual test results. The pragmatic approach to structured testing presented here seems very favourable, but there are some drawbacks to take into account:

1. VDMTools is restarted for each test case to ensure a clean initial state at the beginning of each test. This may lead to long waiting times when a complete regression test is performed over a large test database.
2. The comparison between the output file and the expected outcome file is only performed on the basis of textual (or binary) equality while the comparison may require some looseness. For example, the set $\{1,2,3\}$ is equivalent to $\{3,2,1\}$ because the order in the set is not relevant but it will be rejected by "diff."
3. Maintenance of the database is error prone and laborious because a test case is spread over at least two files, and it needs to be included in the management code of the database (e.g., a makefile).
4. There is no guidance for the structure of test case itself; basically it can be any arbitrary VDM++ expression. Ideally a simple and generic structure should be used for all test cases to ensure maintainability and ease of use.

The extreme programming community has recently put a lot of emphasis back on structured testing. One of the main research themes has been to improve the efficiency of the testing activity in the software life cycle. In their opinion, testing should become as easy as writing code. The problems listed here are typical examples that they want to address and solve. In this section, a solution to these problems is proposed, which is called VDMUnit. It is inspired on the JUnit framework, which was originally designed by Kent Beck and Erich Gamma. JUnit is an open-source framework for testing Java programs (see `http://www.junit.org`). The concepts proposed by the framework are generic and are easily transferred into other modeling and programming languages, such as VDM++. An overview of the VDMUnit framework is provided in Figure 9.6.
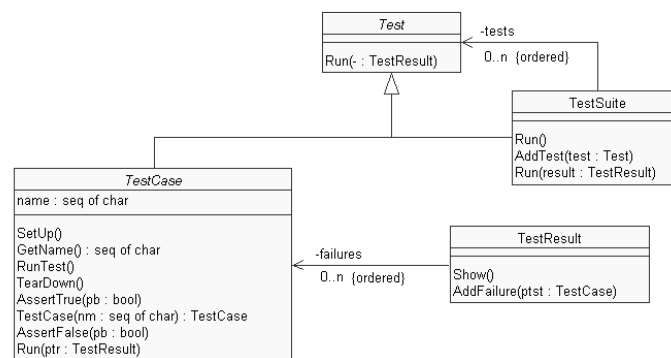

Fig. 9.6: An overview of the VDMUnit framework

9.5.1 Test

The abstract base class `Test` provides a single operation called `Run`, which is used to execute the test. The parameter of type `TestResult` is used to store the outcome of the test. `TestResult` will be presented later in this section:

```
class Test

operations
  public Run: TestResult ==> ()
  Run (-) == is subclass responsibility

end Test
```

Test is a superclass to both TestCase and TestSuite. These classes are needed to build a hierarchy of test cases, which will replace the directory structure in the pragmatic approach shown earlier. A test suite is a collection of either test cases or other test suites. Test suites can be arbitrarily nested and can consist of any number of test cases. This provides maximum flexibility as to how the test database is organised. An example hierarchy is shown in Figure 9.7.
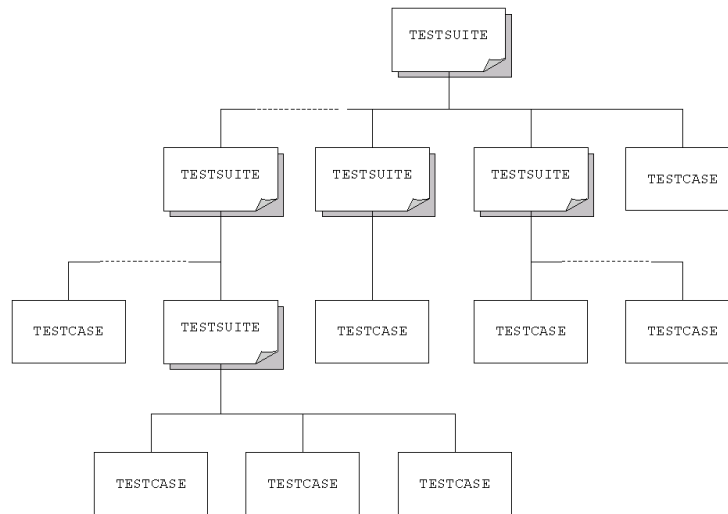

Fig. 9.7: An example hierarchy of test cases

9.5.2 TestSuite

In the TestSuite class the instance variable tests is used to store object references to all test suites and test cases contained in this test suite:

```
class TestSuite
  is subclass of Test

instance variables
  tests : seq of Test := [];
```

The first Run operation is provided for convenience; it is used only by the top-level test suite in the hierarchy. A TestResult instance is created and passed as a parameter to the second Run operation. On return, the results of the test are reported by means of a call to the operation ntr.Show:

```
operations
  public Run: () ==> ()
  Run () ==
    (dcl ntr : TestResult := new TestResult();
     Run(ntr);
     ntr.Show());
```

The second overloaded `Run` operation redefines the operation from the abstract base class `Test`. The operation iterates over all elements in the sequence `tests` and calls the `Run` operation of each element, while passing `result` as a parameter. The parameter `result` is used to "collect" the outcome of each individual test. The operation `AddTest` is used to add a test (which can be a test case or a test suite) to the test suite:

```
  public Run: TestResult ==> ()
  Run (result) ==
    for test in tests do
      test.Run(result);

  public AddTest: Test ==> ()
  AddTest(test) ==
    tests := tests ^ [test];

end TestSuite
```

Exercise 9.6 Redesign the VDMUnit framework so that a strict hierarchy of test suites and test cases is enforced. Currently, it is possible to create loops by adding a test suite to itself, for example. □

9.5.3 TestCase

Each test case is given a symbolic `name`, which makes it possible to generate a human-readable and descriptive error message if a failure is detected during testing:

```
class TestCase
  is subclass of Test

instance variables
  name : seq of char

operations
```

```
public TestCase: seq of char ==> TestCase
TestCase(nm) == name := nm;


public GetName: () ==> seq of char
GetName () == return name;
```

Two operations, `AssertTrue` and `AssertFalse`, are provided to assert test
conditions. Consider the factorial function `fac` from Chapter 5, which is defined as:

```
fac: nat1 -> nat1
fac (n) ==
   if n > 1
   then n * fac(n-1)
   else 1
```

It is now possible to write the following assertions:

```
AssertTrue(fac(3) = 6)
AssertFalse(fac(3) = fac(4))
for all i in set {1,...,10} do
  AssertTrue(fac(i+1) = (i+1)*fac(i))
```

Note that the operation `assertEqual` from the JUnit framework is not sup-
ported because it is unnecessary. The equality operator "=" is defined for all types in
VDM++, so it is sufficient to write `AssertTrue(` $expr_1 = expr_2$ `)`, where *expr* can
be an arbitrary complex expression:

```
protected AssertTrue: bool ==> ()
AssertTrue (pb) == if not pb then exit <FAILURE>;


protected AssertFalse: bool ==> ()
AssertFalse (pb) == if pb then exit <FAILURE>;
```

The assertion operations *raise an exception* if the assert condition is not met. This
is done using the *exit statement*, which has the following syntax:

```
exit
```

or

```
exit expression
```

In the latter case, the expression is used to indicate what kind of exception is raised. Here, the expression <FAILURE> is used. The consequence of the exit statement is that the normal thread of control of the operation is *aborted* and execution is resumed in the innermost exception handler. The innermost exception handler is defined as the last exception handler block that was passed before the exception occurred. Note that the exception handler does not need to be located inside the same class; it can be defined practically anywhere! If the exception handler takes care of the exception, execution is resumed from the exception handler block (note: *not* the location where the exception occurred); otherwise the exception is propagated one level up and so on. A run-time error will occur if an exception is not handled at all. An exception handler block can be defined using the *trap statement*, which has the following syntax:

```
trap pattern with statement-1 in statement-2
```

First, *statement-2* is evaluated. If an exception was raised, the value of *statement-2* is matched against the *pattern*. If there is no matching, the exception is returned as the result of the complete *trap statement*, otherwise *statement-1* is evaluated and the result of this evaluation is also the result of the complete *trap statement*. Note that ptr.AddFailure is called only if an exception with the quote value <FAILURE> occurs in SetUp, RunTest or TearDown. This call will add the current test case to a list of failed test cases as will be shown later:

```
public Run: TestResult ==> ()
Run (ptr) ==
  trap <FAILURE>
    with
      ptr.AddFailure(self)
    in
      (SetUp();
       RunTest();
       TearDown());
```

The operation Run, which is redefined from the base class Test, proposes a standard approach that each test case should implement. First, the operation SetUp is called to create a suitable initial condition for the test case. Second, the operation RunTest will actually perform the test, using the AssertTrue and AssertFalse conditions. Finally, the operation TearDown will be called to clean up in such a way that the system under test can be reinitialized to perform another test case:

```
  protected SetUp: () ==> ()
  SetUp () == is subclass responsibility;

  protected RunTest: () ==> ()
  RunTest () == is subclass responsibility;

  protected TearDown: () ==> ()
  TearDown () == is subclass responsibility

end TestCase
```

Specific test cases can be constructed by inheritance from the class `TestCase` by redefining the operations `SetUp`, `RunTest` and `TearDown`. This approach solves several problems at the same time. The test case and the comparison to the expected outcome are kept together, which makes maintenance a lot easier and transparent. Furthermore, arbitrarily complex assertion statements can be written using the expressiveness of VDM++ which resolves the restrictive textual comparison of the result and expected outcome in the pragmatic test approach. Finally, a "boilerplate" solution, which is very easy to use, is provided for each test case.

9.5.4 TestResult

The class `TestResult` maintains a collection of references to test cases that have failed. The exception handler defined in the operation `Run` of class `TestCase` calls the operation `AddResult`, which will append the object reference of the test case to the tail of the sequence `failures`. The operation `Show` is used to print a list of test cases that have failed or provide a message to indicate that no failures were found. Note that the standard I/O library, which is supplied with VDMTools, is used here. `IO.echo` prints a string on the standard output, just like `System.out.println` in Java. The *def statement* is used to suppress the boolean value returned by `IO.echo`:

```
class TestResult

instance variables
  failures : seq of TestCase := []

operations
  public AddFailure: TestCase ==> ()
  AddFailure (ptst) == failures := failures ^ [ptst];

  public Print: seq of char ==> ()
  Print (pstr) ==
    def - = new IO().echo(pstr ^ "\n") in skip;
```

```
  public Show: () ==> ()
  Show () ==
    if failures = [] then
      Print ("No failures detected")
    else
      for failure in failures do
        Print (failure.GetName() ^ " failed")

end TestResult
```

9.6 Testing the Enigma Model

With the VDMUnit framework defined in the previous section it is possible to apply
structured testing on the Enigma model. All the test cases that were specified can
be found on the book's Web site. Only the test cases for the `Plugboard` and the
`SimpleEnigma` will be shown here, as well as the top-level test suite for the model,
which is called `EnigmaTest`.

9.6.1 PlugboardTest

Class `PlugboardTest` contains the test cases related to class `Plugboard`. A plug-
board cannot be tested without either a reflector or a rotor and a reflector; therefore two
operations are defined: `SimpleTest` and `ComplexTest`, respectively. The values
`refcfg`, `rotcfg` and `pbcfg` are used to define the configurations of the reflector,
rotor and plugboard:

```
class PlugboardTest
  is subclass of TestCase

values
  refcfg : inmap nat to nat =
    {1 |-> 2, 3 |-> 4};

  rotcfg : inmap nat to nat =
    {1 |-> 2, 2 |-> 1, 3 |-> 4, 4 |-> 3};

  pbcfg : inmap nat to nat =
    {1 |-> 3}
```

An alphabet is required to instantiate the reflector, the rotor and the plugboard. The instance variable `alph` is defined for that purpose, which is initialised by the operation `SetUp`. Note that this operation redefines the operation `SetUp` from the base class `TestCase` of the VDMUnit framework:

```
instance variables
  alph : Alphabet


operations
  protected SetUp: () ==> ()
  SetUp () == alph := new Alphabet("ABCD");
```

The operation `SimpleTest` constructs the smallest possible device that contains a plugboard – a plugboard directly connected to a reflector. Observe how the `SetNext` operation is used to connect the two components. The assert conditions were manually calculated and inserted here by the maintainer of the test case:

```
  protected SimpleTest: () ==> ()
  SimpleTest () ==
    (dcl tc : Plugboard := new Plugboard(alph,pbcfg);
     tc.SetNext(new Reflector(1,alph,refcfg));
     AssertTrue(tc.Substitute(1) = 4);
     AssertTrue(tc.Substitute(2) = 3);
     AssertTrue(tc.Substitute(3) = 2);
     AssertTrue(tc.Substitute(4) = 1));
```

The operation `ComplexTest` performs a similar task, but now with an extra component between the plugboard and the reflector, a rotor. Again, the expected outcome is calculated by hand:

```
  protected ComplexTest: () ==> ()
  ComplexTest () ==
    (dcl tc : Plugboard := new Plugboard(alph,pbcfg),
         rot : Rotor := new Rotor(1,1,alph,rotcfg);
     rot.SetNext(new Reflector(1,alph,refcfg));
     tc.SetNext(rot);
     AssertTrue(tc.Substitute(1) = 4);
     AssertTrue(tc.Substitute(2) = 3);
     AssertTrue(tc.Substitute(3) = 2);
     AssertTrue(tc.Substitute(4) = 1));
```

The operations `RunTest` and `TearDown` redefine the operations from the base class `TestCase`. `RunTest` calls the operations `SimpleTest` and `ComplexTest` and `TearDown` is the null operation `skip` because `SimpleTest` and `ComplexTest` only modify their own local state, due to the `dcl` declarations, which are automatically cleaned up after the operation call is completed. Note that the test case could have been split into two separate test cases; it is advisable to do this, especially if the size of the test case is larger than that shown here:

```
  protected RunTest: () ==> ()
  RunTest () == (SimpleTest(); ComplexTest());

  protected TearDown: () ==> ()
  TearDown () == skip;

end PlugboardTest
```

9.6.2 SimpleEnigmaTest

`SimpleEnigmaTest` defines a simple yet very effective test for the Enigma model. Recall that encoding and decoding messages are reversible processes if and only if identical settings are used. Two instances of the class `SimpleEnigma` are created, and for an message of arbitrary length it is now claimed that, for all characters in the message, the decoding of a character is identical to the character itself:

```
class SimpleEnigmaTest is subclass of TestCase

operations
  protected SetUp: () ==> ()
  SetUp () == skip;

  protected RunTest: () ==> ()
  RunTest () ==
    (dcl se1 : SimpleEnigma := new SimpleEnigma(),
         se2 : SimpleEnigma := new SimpleEnigma();
     for ch in "ABCDDCBAABCDDCBAAABBCCDD" do
       AssertTrue(
         se1.Keystroke(se2.Keystroke(ch)) = ch));

  protected TearDown: () ==> ()
  TearDown () == skip

end SimpleEnigmaTest
```

9.6.3 EnigmaTest

Class `EnigmaTest` is the top-level entry point of the test database. It consists of a single operation `Execute`, which constructs a test suite. The test suite is filled with all available test cases, and the tests are performed by calling `ts.Run()`. The result of the test run is eventually shown on the standard output:

```
class EnigmaTest
operations
  public Execute: () ==> ()
  Execute () ==
    (dcl ts : TestSuite := new TestSuite();
     ts.AddTest(new AlphabetTest("Alphabet"));
     ts.AddTest(new ConfigurationTest("Configuration"));
     ts.AddTest(new ReflectorTest("Reflector"));
     ts.AddTest(new RotorTest("Rotor"));
     ts.AddTest(new PlugboardTest("Plugboard"));
     ts.AddTest(new SimpleEnigmaTest("SimpleEnigma"));
     ts.Run())

end EnigmaTest
```

The test cases for the Enigma model have now been defined and the test can be performed. First, a file called `all.arg` is created, which contains the command to start the test run. An instance of `EnigmaTest` is created and the operation `Execute` is called:

```
new EnigmaTest().Execute()
```

The test coverage information file `enigma.tc` is reset before the command-line interpreter is invoked by passing all applicable specification files to the parser of VDMTools:

```
vppde -p -R enigma.tc Alphabet.vpp AlphabetTest.vpp
   Component.vpp Configuration.vpp ConfigurationTest.vpp
   EnigmaTest.vpp Plugboard.vpp PlugboardTest.vpp
   Reflector.vpp ReflectorTest.vpp Rotor.vpp IO.vpp
   RotorTest.vpp  SimpleEnigma.vpp SimpleEnigmaTest.vpp
   Test.vpp TestCase.vpp TestResult.vpp TestSuite.vpp
```

Now the command-line interpreter is invoked to start the test run. Note that the interpreter is started only *once* to execute *all* test cases. This is a major performance improvement over the pragmatic approach presented earlier:

```
vppde -iDIPQ -R enigma.tc -O all.res all.arg
   Alphabet.vpp AlphabetTest.vpp Component.vpp
   Configuration.vpp ConfigurationTest.vpp
   EnigmaTest.vpp Plugboard.vpp PlugboardTest.vpp
   Reflector.vpp ReflectorTest.vpp Rotor.vpp IO.vpp
   RotorTest.vpp SimpleEnigma.vpp SimpleEnigmaTest.vpp
   Test.vpp TestCase.vpp TestResult.vpp TestSuite.vpp
```

As expected, the following result is obtained:

```
No failures detected
(no return value)
```

Despite the fact that no failures were detected and the test suite was composed very carefully, the basic question remains: To what extent is the model really tested? A partial answer is provided by the test coverage information generated during the test run by the interpreter. Test coverage can be analysed in two ways, by generation of test coverage tables and by colouring the source text of the model. In the latter case, statements that have not been exercised by the test suite are highlighted. This makes it relatively easy to design a new test case, which is added to the test suite, that will ensure that the untouched statement is tested in the next test run. By performing a regression test and analysing the test coverage information again it is possible to verify whether the model is fully exposed to test cases. Note that it does not guarantee that the model will always work, it is just a guarantee that each statement of the model is tested by at least one or more test cases. This quality measure is an important criterion in the quest to raise confidence in the model. As an example, the test coverage information of the `Alphabet` class is provided in Table 9.4.

Table 9.4: Test coverage overview for the `Alphabet` class

Name	#Calls	Coverage
Alphabet'AlphabetInv	18	√
Alphabet'Alphabet	6	√
Alphabet'GetChar	52	√
Alphabet'GetIndex	52	√
Alphabet'GetIndices	520	√
Alphabet'GetSize	77	√
Alphabet'Shift	1366	√
Total Coverage		**100%**

9.7 Summary

This chapter has provided a detailed illustration of the construction of a model for a relatively intricate system and has introduced a systematic approach to testing as a means of gaining confidence in the model. Apart from the technical details of the Enigma device and its model, several more general points have been made, with a bearing on the use of formal object-oriented modelling techniques in practice:

- Opportunities for generalisation and reuse were exploited in making model design decisions. Patterns were identified and applied. It is worth investing time in the careful separation of generic aspects from the domain-specific aspects of the model.
- When constructing a model, it is important to bear its future maintenance in mind and let this influence structuring and abstraction decisions.
- There is a clear distinction to be made between testing and debugging. For the purposes of systematic testing, it is worth defining a test framework supporting unit tests to system-level tests and regression testing.
- Investing time in gaining maximum confidence in a model is worthwhile because of the potential gain in debugging and rework costs if major defects get through to implementation.

Validated Designs for Object-oriented Systems
Fitzgerald, J.; Larsen, P.G.; Mukherjee, P.; Plat, N.;
Verhoef, M.
2005, XII, 404 p. 65 illus., Hardcover
ISBN: 978-1-85233-881-7