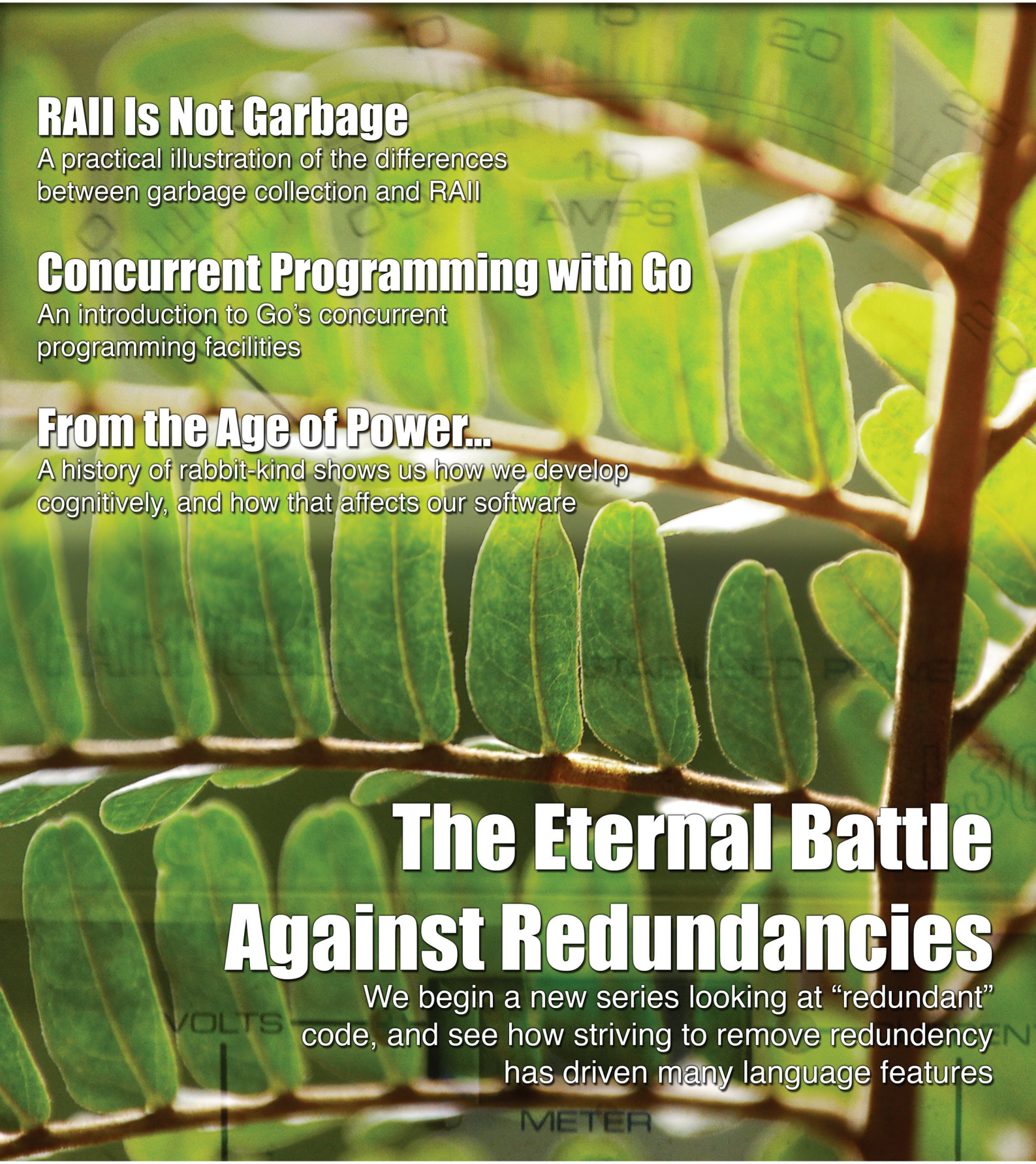
overload 106

DECEMBER 2011 £3



RAII Is Not Garbage

A practical illustration of the differences between garbage collection and RAII

Concurrent Programming with Go

An introduction to Go's concurrent programming facilities

From the Age of Power...

A history of rabbit-kind shows us how we develop cognitively, and how that affects our software

The Eternal Battle Against Redundancies

We begin a new series looking at "redundant" code, and see how striving to remove redundency has driven many language features

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 106

December 2011

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 107 should be submitted by 1st January 2012 and for Overload 108 by 1st March 2012.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

The Eternal Battle Against Redundancies, Part I

The drive to remove redundancies is widely seen as a good thing. Christoph Knabe sees how it has influenced programming languages.

Since the beginning of programming, redundancies in source code have prevented maintenance and reuse. By 'redundancy' we mean that the same concept is expressed in several locations in the source code. Over the last 50 years the efforts to avoid redundancies [Wikipedia] have inspired a large number of programming constructs. This relationship is often not obvious to the normal programmer. Examples include relative addressing, symbolic addressing, formula translation, parameterizable subroutines, control structures, middle-testing loops, symbolic constants, preprocessor features, array initialization, user defined data types, information hiding, genericity, exception handling, inheritance, dynamic dispatch, aspect oriented programming, functional programming, and even program generators and relational databases. These constructs are discussed using examples from 14 widely used programming languages. Whosoever understands the common concept is well equipped for the future.

What can the Zuse computer say today?

In 1971 my (high) school inherited a 12-year-old Zuse 22 and I learned programming on it. In the rapidly moving computer domain we would usually consider using such obsolete technology to be a waste of time. But this has provided me with the background for a survey of the programming techniques developed over the last 50 years. The Zuse 22 of the German computer pioneer Konrad Zuse was one of the first mass produced computers in the world (55 machines was a lot in those days!). It had a highly economical construction and was programmed in a machine level language: the Freiburgian Code. The example program in table 1 adds the natural numbers from *n* decrementing to 1, and prints the result (tested by the Z22 simulator of Wolfgang Pavel [Pavel]). In practice only the contents of the Instruction column were punched onto paper tape and read by the computer as a program. The Address column indicates into which word the instruction was stored, and the Comment column corresponds to comments in contemporary languages.

The instructions can have symbolic, combinable operation letters : **B**=Bring, **A**=Add, **S**=Subtract, **T**=Transport, **U**=Umspeichern (store to), **C**=Const-Value, **PP**=if Positive, **E**=Execute from (go to), **D**=Drucken (print), **Z**=Stop. But the addressing was purely numeric with absolute storage addresses. Here the variables *i* and *sum* are stored at the addresses 2048 and 2049 respectively. The algorithm itself is stored from address 2050, where we jump back using the instruction `PPE2050` , if the value of *i* is still positive.

Redundancies appear here in the addresses: 2048 for *i* appears 4 times, 2049 for *sum* 3 times, 2050 for the beginning of the program and the loop twice explicitly and once implicitly (two cells after where the tape content is stored). As a consequence the program is neither relocatable in the working storage nor simply extendable. So there are big difficulties in its maintenance.

Table 1

Address	Instruction	Comment
	T2048T	**T**ransport the following to words 2048 ff.
2048	10'	**i**: Initial value for i is n, here the natural number 10.
2049	0'	**sum**: Initial value is the natural number 0.
2050	B2049	**B**ring the sum into the accu(mulator).
2051	A2048	**A**dd i to the accu.
2052	U2049	Store (**U**mspeichern) accu to sum.
2053	B2048	**B**ring i into the accu.
2054	SC1	**S**ubtract the Constant value 1 from the accu.
2055	U2048	Store (**U**mspeichern) accu to i.
2056	PPE2050	If accu **P**ositive **E**xecute from (go to) 2050
2057	B2049	**B**ring sum into the accu.
2058	D	Print (**D**rucke) accu.
2059	Z0	Stopp
	E2050E	**E**xecute now from 2050

Relative and symbolic addressing

Progress came later for the transistorized Zuse 23 by the development of 'Relative Addressing'. This enabled a programmer to write a subroutine as if it was located at address 0. A certain prefix instruction told the loading program to store the actual start address in a load-time base register, which was usually register 26. Appending `A26` to an address caused the loading program to add the content of register 26 to the value to form an absolute address before storing the instruction to be executed later. So when using relative addressing the conditional jump instruction to the beginning of the program in table 1 would be `PPE2A26` instead of `PPE2050`. By this means the program has become relocatable. Relative addressing was still very economic: it did not need more resources than the register 26 at load time.

True, relative addressing facilitates relocating a subroutine in working storage, but inside the subroutine it is as inflexible as absolute addressing. If we wanted to extend the example by inserting a prefix action before the calculation loop, we would have to shift the relative jump goal `2A26`, too. Thus 'symbolic addressing' was introduced with the Zuse 23 (sold from 1961). See the German programming manual for the Z23 [Zuse23] p. 65ff. The Z23 loading program substituted each bracketed identifier of up to 5 characters by its address. The program from table 1 could be rewritten with the symbolic addresses `(I)`, `(SUM)`, and `(BEGIN)` as in Listing 1.

Now it is possible to insert further instructions at any place without destroying the program. This improvement was such a big one that the assembler for the Siemens 2002 was named after this technique, PROSA (Programming with Symbolic Addresses). Necessary resources for symbolic addressing were an addressing program and a symbol table.

Christoph Knabe learned programming at high school on a discarded Zuse 22, studied computer science from 1972, worked as a software developer at www.psi.de, and since 1990 is professor of software engineering at the Beuth University of Applied Sciences Berlin www.bht-berlin.de. Scala is the 14th language in which he has programmed intensively.

relative addressing facilitates relocating a subroutine in working storage, but inside the subroutine it is as inflexible as absolute addressing

Architecture of the Zuse 22

The design of the Z22 was finished by about 1955, and 55 machines of this type were produced. It formed a whole generation of computer specialists in central Europe. The Z22 was characterized by:

- hardware logic implemented by 600 tubes
- working storage: a magnetic drum of 8192 words @ 38-bit
- registers: a core memory of 14 words @ 38-bit
- peripheral storage: 5 hole punched paper tape
- console I/O: push buttons, glow-lamps, teletype with paper tape reader
- operating frequency: 3 kHz

An instruction consisted of 38 bits: 2 with the value 10, then 5 for conditions, 13 for operations, 5 for a register address, and 13 for a working storage address. Each of the condition and operation bits was programmed by a specific letter and switched a specific gate.

The registers could be accessed by their address, but some were used for special purposes by some operations. Access time for registers was always one CPU cycle, for drum words only if they were accessed in sequence.

Some Registers of the Z22

Number	Special Usage
2	Testable by P or Q if positive or negative
3	Overflow area for accumulator, last bit testable by Y
4	Accumulator, filled by B, added by A, etc., testable by PP etc.
5	Stores return address for subroutines, filled by F

The Zuse 23 of 1961 was logically equivalent, but was implemented using transistors. It had 40-bit words and up to 255 registers.

We are now acquainted with the most common procedure for redundancy elimination: The redundant code part gets a name and we use that name instead of the former, redundant code parts.

```
T2048T
(I) 10'
(SUM) 0'
(BEGIN) B(SUM)
A(I)
U(SUM)
B(I)
SC1
U(I)
PPE(BEGIN)
B(SUM)
D
Z0
E(BEGIN)E
```

Listing 1

Formula translation

In the beginning, technical and scientific calculations dominated computer applications. But as we can see in the words 2050…2053 of the Z22 example, a simple addition needed three instructions (bring, add, store). For the formula $(a+b)*(a-b)$ we would need about 7 instructions. So the need quickly arose for simplifying formula calculations. This was enabled by formulae with variable identifiers, literals, operator signs, operator priorities, and parentheses. The programming language FORTRAN got its name by this feature (Formula Translator). Fortran I was defined in 1956. At that time there was no possibility of defining your own operators.

Subroutines

If you needed an instruction sequence several times, on the Zuse 22 you could jump there by using the call instruction **F** from different program locations. Besides doing the actual jump, this instruction loaded a 'jump back' instruction into register 5. That is why you had to begin each subroutine by copying the contents of register 5 to the end of the subroutine using a U-instruction. This assured a jump back to the calling location when reaching the end of the subroutine.

But often you don't need identical, but only similar processing. In this case Freiburgian Code had the convention of reserving cells before the subroutine for arguments and results. These had to be filled by the caller before the call instruction and retrieved afterwards, respectively. Then it had to jump to the first instruction of the subroutine, which always had to be **B5** followed by a **U** with the address of the last instruction cell of the subroutine. So the program from listing 1, converted into a Zuse23 subroutine with entry address **SUMUP** for summing up the integer numbers from 1 to *n*, is shown in listing 2.

In order to print the sum of the numbers from 1 to 20, you could call **SUMUP** as follows:

```
BC20    U(N)    F(SUMUP)    B(SUM)    D
```

While this had to be obeyed as a convention on the Zuse 23, nowadays it is automated in all higher programming languages by the concept of a subroutine call with an argument list and return value. FORTRAN II and

```
T2048T
(N) 10'
(SUM) 0'
(SUMUP) B5
U(BACK)
B(SUM)
A(N)
U(SUM)
B(N)
SC1
U(N)
PPE(SUMUP)
(BACK)Z0
```

Listing 2

By combining these facilities you could construct arbitrarily complex algorithms

```
C     COMPUTES THE SUM OF THE NUMBERS FROM 1 TO N
      FUNCTION ISUMUP(N)
      ISUM = 0
      DO 99 I = 1, N
   99 ISUM = ISUM + I
      ISUMUP = ISUM
      RETURN
      END
```

Listing 3

Algol introduced the concept of named, parameterized subroutines around 1958. The possibilites for redundancy elimination were enormous and gave rise to the style of procedural programming. A subroutine in FORTRAN II for summing up the numbers from 1 to *n* by a counting loop is shown in listing 3. Identifiers starting with **I** to **N** are considered integers.

Control structures

The building blocks by which programs got their high flexibility and reusability were conditional jumps such as the **PPE** instruction of the Zuse 22. Conditional forward jumps could be used to implement conditional branches. A conditonal backward jump could be used to implement repetition, which would terminate when the condition no longer held. By combining these facilities you could construct arbitrarily complex algorithms. As conditional branches and limited repetitions with or without a control variable were needed frequently, the popular programming languages introduced such constructs. In FORTRAN I (1957) you could sum up the integer numbers from 1 to *n* by the following counting loop:

```
      ISUM = 0
      DO 99 I = 1, N
   99 ISUM = ISUM + I
C  HERE ISUM CONTAINS THE SUM OF THE NUMBERS
C  FROM 1 TO N
```

FORTRAN had half-symbolic addressing. A statement could be identified by a freely electable number, a so-called 'label'. The statement **DO 99 I = 1, N** incremented the control variable **I** from 1 to **N**, and each time all statements up to and including the statement labeled by 99 are repeatedly executed. For branching FORTRAN I offered the arithmetic **IF**:

IF *(expression) negativeLabel,zeroLabel,positiveLabel*

This statement jumps to one of the three enumerated labels depending on the sign of the expression result.

Algol 60 had already introduced nowadays common control structures:

1. A multi-branches cascadable **IF**: **if** *cond* **then** *expr* **else** *expr*
2. A universal loop with control variable, for example:
   - **for i := 1 step 1 until 100 do print(i)** prints the numbers from 1 to 100
   - **for i := 1, i*2 while i<2000 do print(i)** prints the powers of 2 from 1 to 1024.

```
VAR
    x: integer;
    sum: integer := 0;
BEGIN
    readNumber(x);
    WHILE x>0 DO BEGIN
        sum := sum + x;
        readNumber(x);
    END;
    writeln;
    writeln('The sum is ', sum, '.');
END
```

Listing 4

Modern control structures with the purpose of being able to completely avoid jump instructions were introduced by Pascal (1970). Pascal distinguished the pre-testing **WHILE-DO**-loop, the post-testing **REPEAT-UNTIL**-loop, and the counting **FOR-DO**-loop. Nevertheless Pascal still contained the **GOTO** statement, as non-local termination could not be done otherwise

Unfortunately the **WHILE**-loop, planned for repetitions where the number of iterations is not known in advance, implies redundancies in the following use case, which occurs extremely often in practice: We want to process an unknown number of elements, maybe even none. The code in listing 4 reads positive numbers and prints their sum. The procedure **readNumber** is understood to deliver -1 if it can't find any further number. The keywords are typed in upper case, although this is not important in Pascal.

We see, that the procedure **readNumber** has to be called redundantly: Once before the **WHILE**-loop, the second time at the end of the loop body, in order to prepare the variable **x** for the next test of the **WHILE**-condition.

That is why C (1973), Modula-2 (1978), and Ada (1980) introduced the possibility of leaving an arbitrary loop by a special jump instruction, in particular an endless loop. Using this we could solve the above task without redundancies (C syntax, listing 5).

This corresponds to the general pattern for processing an unknown-in-advance number of records in a middle-testing loop (listing 6).

Recursion: Recursion is when a function calls itself either directly or indirectly. The ability to do so was introduced by LISP and Algol at around the same time, but the then predominant languages FORTRAN and COBOL did not allow recursion. Some tasks, e.g. traversing trees, are most

```
int sum=0, x;
for(;;){
    readNumber(&x);
    if (x<=0) break;
    sum += x;
}
```

Listing 5

The ability for programmers to define their own control structures was born in LISP

```
initialize
for(;;){
  retrieve
  if (notSuccessful) break;
  process
}
```

Listing 6

elegantly expressed by recursion. Recursion does not directly contribute to the elimination of redundancies. If the recursive call is the last statement of a function, the compiler can replace it by a simple, storage-efficient loop. Compilers of functional programming languages regularly do this optimization.

User-defined control structures

The ability for programmers to define their own control structures was born in LISP (1958), as in this language instructions and data were notated in the same form, the so-called S-expressions. They also had the same internal representation. For example, `(TIMES N 7)` on the one hand means the multiplication of `N` by 7. On the other hand it is simply a list with the elements `TIMES`, `N`, and `7`. A function in LISP I, which was defined as `FEXPR` instead of the usual `EXPR`, had a special property. It evaluated its passed arguments not before executing its function body, but left this until the explicit usage of the function `EVAL` in the function body. So a function body could arbitrarily control the frequency and order of evaluation of its argments.

Later on when the power of this concept was found out, LISP dialects introduced comfortable notations to define `FEXPR`s. For example in MacLISP you could define an `if-then-else` construct, which was not contained in early LISP, by the traditional `COND` as follows:

```
(defun IF fexpr (args)
  (let ((predicate (car args))
        (then (cadr args))
        (else (caddr args)))
   (cond
    ((eval predicate) (eval then))
    (t (eval else)))))
```

The `IF` function gets all arguments unevaluated as a list with the name `ARGS`. `LET` assigns the individual arguments to the values `PREDICATE`, `THEN`, and `ELSE`. `COND` evaluates `THEN` or `ELSE` depending on the evaluation of `PREDICATE`. Example from Steele and Gabriel [LISP].

As a lover of redundancy-free solutions I missed middle-testing loops in the modern object-functional language Scala (2001). But I could define one myself (listing 7).

The function `loopGetTestProcess` has 3 parameter lists. In the first it expects an expression `getItem` for obtaining a next item, in the second a boolean function `shouldContinue` for judging the success of the obtainment, and in the third a function `processItem` for processing the obtained item. By using the generic parameter `[Item]` it is assured that

```
def loopGetTestProcess[Item]
    (getItem: => Item)
    (shouldContinue: Item=>Boolean)
    (processItem: Item=>Unit)
{
    var item = getItem
    while(shouldContinue(item)){
        processItem(item)
        item = getItem
    }
}
```

Listing 7

the types fit together. The redundant call of `getItem` is well encapsulated in the function body and is not visible in the using source code.

As syntactic sugar in Scala you can write an argument list of length 1 with curly brackets thus enabling the following usage:

```
def printLines(in: java.io.BufferedReader){
  loopGetTestProcess(in.readLine())(_!=null){
    println
  }
}
```

In Java this could be done by `break;` and would look like in listing 8.

The big achievement in Scala is that the programmer has defined this control structure himself and can define others too.

Constants

In working storage every cell is modifiable. That is why there were no constants in Freiburgian code. By around 1960, the predominant higher programming languages FORTRAN, Algol, and COBOL had only variables and literals, usable in expressions. e.g. the literal `2` in the FORTRAN expression `A**2 + 2*A*B + B**2` with `**` meaning 'to the power of'. A redundancy problem arises, if you must use the same literal several times. In COBOL 66 you could also use literals to dimension a variable, for example a west-German ZIP code, which consisted of four digits, as `ZIP PICTURE 9(4)`. But if you had done it this way at several locations in your program code, the reorganisation of the German postal systems in 1993 obliged you to change all these locations to five digits: `ZIP PICTURE 9(5)`. In the early higher programming languages there was no possibility to declare variable or array sizes free of redundancy.

```
void printLines(final java.io.BufferedReader
in){
    for(;;){
      final String line = in.readLine();
      if(line==null)break;
      println(line);
    }
}
```

Listing 8

Pascal (1970) solved this problem very cleanly by declarable, symbolic constants, which could be used for dimensioning arrays, as well. E.g.:

```
CONST zipLength: integer := 4;
TYPE ZipCode = PACKED ARRAY[zipLength] OF
 character;
VAR clientZip: ZipCode;
BEGIN
  FOR i := 1 TO zipLength DO write(clientZip[i]);
```

Pascal also introduced the concept of user-defined data types (here the type **ZipCode**). Along with the symbolic dimensioning constants this enabled you to define sizes for a whole software system free of redundancies.

C (1973) solved the dimensioning problem less elegantly, as you had to use preprocessor macros instead of symbolic constants. But on the other hand you could even do it without explicit size constants, if you used the **sizeof** operator. So the same thing expressed in C:

```
typedef char ZipCode[4];
ZipCode clientZip;
int i=0;
for(; i<sizeof(clientZip); i++){
  putchar(clientZip[i]);
}
```

C++ (1983) introduced the ability to declare frozen variables at any location in the program code by the keyword **const** and by that to build calculated 'constants'. Beginning with Java (1995) it became normal to dimension arrays only at runtime, or to work with the growable collections.

Preprocessor features

Often a preprocessor was used in order to avoid redundancies. So it is possible to include declarations, which are needed in the same wording in several compilation units, from one single file. This technique was used in COBOL (**COPY**), FORTRAN (**INCLUDE**), and C (**#include**). In this way you could build a data abstraction module in C. You had to write the function declarations of the module in its header file, and the function definitions along with the managed module variables in its implementation file. As an example see the header file `stack.h` for a stack of characters. The lines starting with **#** contain preprocessor directives.

```
#ifndef stack_h
#define stack_h

#define MAX_STACK_SIZE 10
void stack_push(char item);
char stack_pop(void);

#endif
```

Every client of this stack has to include the header file `stack.h` and then you can call the functions declared therein:

```
#include "stack.h"
...
stack_push('X');
```

About 1985–1995 this was the predominant modularization technique in industry, before it was superseded by object orientation.

Stringize Operator: The C/C++ preprocessor contains the **#** operator, which delivers the name of a given macro argument as a string. With this you can program without redundancies simple dialogs, e.g. for test purposes. So you can use in C++ the following macro **PROMPT_READ** in order to print the name of the passed variable, and to read a value into it afterwards:

```
#define PROMPT_READ(var) \
{ cout << #var << "? ";    cin >> var;}
```

Using this macro you can program a console dialog as in listing 9.

A typical dialog usage could look as follows:

```
name? Ulf
age? 22
Ulf is 22 years old.
```

```
string name;
int age;
PROMPT_READ(name);
PROMPT_READ(age);
cout << name << " is " << age <<
   " years old." << endl;
```

Listing 9

Generally preprocessors are considered an inelegant solution. That is why the majority of cases for which you used the preprocessor in C or C++, are solvable in Java by its own language constructs without a preprocessor. However in Java the following is not possible:

- determination of the name of a variable as by the **stringize** operator in C
- delegation of the abortion of a method execution and some accompanying action, e.g. **errno=FAILURE; return;** to another method
- delegation of a certain pattern of exception handling to another method.

  If we had C-like macros in Java we could write

  ```
  TRANSACTION(myAction1(); myAction2())
  ```

  which would be useful to transform a statement sequence into a database transaction by

  ```
  #define TRANSACTION(stmts) \
  {try{stmts; commit();}catch(Exception e) \
  {rollback(e);}}
  ```

Array initialization

Pascal introduced redundancy-free array dimensioning, but initialization still had a problem. As an example we want to declare and initialize an array in Pascal, which contains codes for three allowed actions. Even in modern GNU Pascal you have to indicate the array size 3:

```
VAR actions: ARRAY[1..3] OF char = ('G','P','D');
```

But the size is redundant in relation to the number of initialization elements. The more actions you need, the more difficult becomes the manual numbering.

C introduced to derive an array's size from the length of its initialization list: `static char actions[] = {'G','P','D'};`. Java inherited this useful feature.

Summary and prospects

The early higher programming languages introduced the possibility of avoiding redundancies by extraction, parameterization, and naming of a repeated code snippet. Such code snippets could be: addresses, values, size indications, declarations, and statement sequences. Some code patterns were invoked by syntactical units of the programming language, e.g. loops. In the better case the programmer could give a freely electable name to a code pattern. If a programming language helped to eliminate redundancies better than a competing language, this was a relevant advantage in the battle for dissemination.

In the 2nd part of this article we will deal with the more modern techniques 'information hiding', genericity, exception handling, object-oriented, aspect-oriented, and functional programming as well as domain specific languages and relational data bases, and how they all contribute to redundancy elimination. ■

References

[LISP] Steele/Gabriel: *The Evolution of LISP*, p. 9: http://www.dreamsongs.com/Files/HOPL2-Uncut.pdf

[Pavel] http://www.wpavel.de/zuse/simu/

[Wikipedia] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[Zuse23] http://www.weblearn.hs-bremen.de/risse/RST/WS04/Zuse23/Z23Programmierungsanleitung.pdf

overload 107

FEBRUARY 2012 £3

A Practical Introduction to Erlang

We see some of the main concepts behind this language, illustrated with a practical example

Memory Leaks and Memory Leaks

We look at the common sources of this software fault

The Eternal Battle Against Redundancies

We continue our journey into language design and redundant code

Why Computer Algebra Won't Cure Your Calculus Blues

We see why symbolic algebra isn't a solution for accurate calculus

Many Slices of Pi

We investigate some of the issues involved in writing scalable parallel algorithms, using a Monte Carlo calculation of the value of pi

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 107

February 2012

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 108 should be submitted by 1st March 2012 and for Overload 109 by 1st May 2012.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

The Eternal Battle Against Redundancies, Part 2

Repeated information leads to poor quality software. Christoph Knabe continues to see how removing them has influenced language design.

Since the beginning of programming, redundancies in source code have prevented maintenance and reuse. By *redundancy* we mean that the same concept is expressed in several locations in the source code. Over the last 50 years the efforts to avoid redundancies [DRY] have inspired a large number of programming constructs. This relationship is often not obvious to programmers in their daily work. In part I [Part1] we talked about relative addressing, symbolic addressing, formula translation, parameterizable subroutines, control structures, middle-testing loops, symbolic constants, preprocessor features, and array initialization. In this part we will investigate higher concepts like object-oriented, aspect-oriented, and functional programming, as well as exception handling and even program generators and relational databases, and how these concepts contribute to redundancy avoidance. These concepts are discussed on the basis of prevalent programming languages. Whosoever understands the common concept is well equipped for the future.

Information hiding

The principle of information hiding was formulated by Parnas [Parnas]. It postulates not to allow direct manipulation of a data structure by clients. Such manipulations are to be done only through operations which are grouped in an interface. Information hiding was the prevalent design criterion in modular programming and it still plays an important role in object-oriented programming.

Enforcing the information hiding principle guarantees that the intended administration operations cannot be bypassed by a module's users. This contributes to redundancy avoidance by the fact that the logic behind the administrative functions cannot migrate into the user's code with the risk of duplication therein. This danger was always present in languages without support for information hiding.

Secure information hiding was enabled in C (1973) by the declaration of file-scope `static` variables. Such variables stayed alive beyond a function call, but were not accessible from outside the source file. Later languages which introduced special constructs for module interfaces and implementations were Modula-2 and Ada.

In C++ (1983) the information hiding principle was extended to user-defined data types (classes) by giving class members private visibility by default, which could be explicitly changed to `public`.

Genericity

COBOL (1960) had composite variables, but only Pascal (1970) introduced user-defined, composite data types as `RECORD`s. C (1973) followed with `struct`s. These constructs increased the robustness of programs, as confusions of e.g. persons with windows, calendar dates, or jobs were detected by the compiler. But the new strictness led to problems in the creation of universal services. Although Pascal had elegant operations for dynamic data structures, it was impossible to program a linked list so that it would be usable for an arbitrary element data type. The link data and the type of the payload data had to be firmly combined in the type for a list node. E.g.:

```
TYPE
  PersonList = ^PersonNode;
  PersonNode = RECORD
    info: Person;
    nextPtr: ^PersonNode;
  END;
```

If you wanted to use the same list management module in Pascal for different payload data types, you had to copy the source text and globally substitute the payload data type name.

The somehow less strict C could bypass such problems by using an untyped pointer, the `void*`. So in C it was possible, although insecure, to implement list management for arbitrary payload data. This list node could be formulated as follows:

```
struct List {
   void* infoPtr;
   struct List* nextPtr;
};
```

Only Ada (1980) achieved a synthesis of user-defined, composite data types (records) with flexible type safety. This concept was named genericity and was accepted by all modern, statically typed languages such as C++ (templates), Java, C#, and Scala. Using genericity you can avoid redundancies if you have to define same-behaviour services for different payload data types. The generic collection classes implemented by this technique are used quite frequently in all contemporary programming languages.

Dynamically typed languages such as Smalltalk or Ruby circumvent the problem described here by postponing the type checks to run-time.

Exception handling

In older programming languages (Lisp, Fortran, Algol, Cobol, Pascal, C) there was no automatic handling of exceptions. After every subroutine call the caller had to check manually whether the subroutine terminated successfully or erroneously. To complicate matters further there was no universal convention for how a subroutine should communicate its failure to the caller. The Unix services written in C used a special value for the function result as well as error codes in the global variable `errno`. The latter way was more suitable for standardization, as it did not have to cope with different function result types, but it was not suitable for the upcoming multi-threading.

How was the `errno` convention applied? After invoking `fopen(filename, "r")` in order to open a file you had to check whether `errno` had a nonzero value. As there were neither destructors nor garbage collection mechanisms in C, errors found could not be easily

Christoph Knabe learned programming at high school on a discarded Zuse 22, studied computer science from 1972, worked as a software developer at www.psi.de, and since 1990 has been professor of software engineering at the Beuth University of Applied Sciences Berlin (www.bht-berlin.de). Scala is the 14th language in which he has programmed intensively.

Inheritance alone enables a **minor avoidance of redundancy** by extracting common state and behavior of several data types into a base class

Mentioned Programming Languages

Note: The concepts are all talked about on the basis of prevalent programming languages. But often they were before tried out in research languages as Simula-67, CLU, MESA, or LISP dialects.

Name	Year	Innovations
Freiburgian Code	> 1958	Programming of the Zuse 22
Freiburgian Code Z23	1961	Relative and symbolic addressing
FORTRAN	1957	Formula translation, FORTRAN II: subroutines, linker
ALGOL	1958	Subroutines, block principle, BNF, control structures, recursion
LISP	1958	Garbage collection, recursion, functional programming (FP)
COBOL	1959	Record variables, long identifiers
Pascal	1970	Record types, pointer types, structured programming
Smalltalk	1972	Dissemination of object-oriented programming (OOP)
C	1973	Preprocessor, **sizeof**, operating system API, information hiding, **break**
Modula-2	1978	Separation of interface/implementation, **if...end**
Ada	1980	Genericity, automatic exception handling
C++	1983	Static typesafe OOP, freezing variable values, late declaration
Java	1995	Static typesafe OOP with garbage collection, stack trace API
AspectJ	2001	Centralized solution of cross-cutting concerns
Scala	2003	Static typesafe synthesis of OOP and FP

collected in working storage, so tended to be immediately reported. But this limited the universal usability of a subroutine, as then the destination of error reporting was not easily chosen by the caller.

So the correct handling of a function call in C on Unix, here of the function **fopen**, appeared as follows:

```
FILE* pFile = fopen(filename, "r");
if(errno!=0){
    perror(filename); //prints errno and filename
    fprintf(stderr, "at file %s in line %d\n",
       __FILE__, __LINE__);
    errno = FAILURE;
    return NULL;
}
```

You can easily imagine that correct error handling was highly redundant and made program texts harder to read and understand, and so harder to maintain. Furthermore, you had to write so much to implement this handling that programmers rarely practised it. Fortunately C's preprocessor macros offered a means to partially eliminate this redundancy. You could extract the portion of the example from **if** up to **return NULL;}** into a macro, which should get a context and the function result in case of failure as arguments.

```
#define ERRCHECK(context, failResult) ...
```

The invocation of **fopen** could then be much shorter:

```
FILE* pFile = fopen(filename, "r");
ERRCHECK(filename, NULL)
```

This approach cannot yet solve the problem of functions failing when they were combined in expressions, e.g. **f(x)*g(x). ERRCHECK** could only be applied between two statements, not inside an expression.

Such error handling, which was implemented here manually, is done by contemporary languages automatically, when a function throws an exception. Standardized handling (usually a message with stack trace and program abortion) is guaranteed, although custom handling is possible. Automatic exception handling was popularized by Ada 80. C++ adopted it around 1990, while Java contained it from the beginning (including an API access to the stack trace of a caught exception).

Object-oriented programming

The technique of object-orientation, introduced by Simula 67 and popularized by Smalltalk-80, adopts 'information hiding' for object attributes and contains as innovations 'inheritance', 'reference polymorphism', and 'dynamic method dispatch'. Inheritance alone enables a minor avoidance of redundancy by extracting the common state and behaviour of several data types into a base class. Compared to composition this saves only a (relatively) small amount of writing when accessing an inherited attribute or method. Polymorphism of references enables a flexibility similar to the untyped pointers of C, but considerably more secure, as it constrains the referenced elements to subclasses of the base class. With dynamic dispatch for calls of virtual functions (C++, 1983) came the big, redundancy-avoiding progress, which is nowadays commonly known as the 'Template Method Pattern' [TemplMeth].

Template Method Pattern: As an example let us have a look at the problem of transaction management. In enterprise applications each operation of the business logic must be executed as a transaction. If the logic operation succeeds, the database modifications must be committed, otherwise errors must be reported and the database modifications must be rolled back. Instead of redundantly programming this behaviour in each logic method, you can extract it into an **execute** on a base class **Transaction**, which will call an abstract **action** method, which has to be overridden with the concrete logic operation. In Java, the solution looks like Listing 1.

The *template method* **execute** follows a fixed procedure in order to guarantee the **commit** or rollback. Only the business logic part of the action is conferred in the template method upon the abstract method **doAction**. The programmer of the subclasses has then to implement this method. Usage would follow the pattern shown below and would appear

Aspect-oriented programming enables you to handle concerns that cut across a software system

```
abstract class Transaction {
  public void execute(){
    final Connection con =
       DatabaseUtil.getConnection();
      try{
        doAction();
        con.commit();
      }catch(Exception ex){
        report(ex);
        con.rollback();
      }
  }
  abstract void doAction() throws Exception;
}
```

Listing 1

in a real system hundreds of times, which leads to an enormous reduction of redundancy, although the amount of code is still problematic.

```
new Transaction(){
  public void doAction() throws Exception {
    //Here the actual logic operation is placed.
  }
}.execute();
```

An alternative solution in Java would make use of reflection [Refl], as done by EJB 3.0 application servers internally. Each method of a class annotated as `@Session` is executed as a transaction.

Mixin Programming: In contrast to Java inheritance, Scala (2003) allows the mixing in of several *traits* (partially implemented interfaces), each of which can offer such template methods. The 'diamond problem' usually occurring with multiple inheritance is avoided by an explicitly definable resolution order. By this means you can freely combine different services in a class. In fact the Scala collections framework stands out due to an extremely high internal re-use of a few template methods. This is a big contribution to redundancy avoidance.

Aspect-oriented programming

Aspect-oriented programming enables you to handle concerns that cut across a software system centrally in an `aspect`. The above-mentioned problem of transaction management is exactly such a cross-cutting concern. Let us consider the case where each method of a logic façade should be executed as a transaction. Although the above solution, implementing the method `doAction` in an anonymous subclass of `Transaction`, is technically free of redundancy, it needs a lot of code. In contrast to this, in the solution with AspectJ (2001) in Listing 2, the aspect needs to be noted only once for the whole system. The 'pointcut' `executeAnyFacadeMethod` captures each execution of a method of objects of the type `LgFacade`. The `around` advice surrounds the captured method executions at the location, marked by `proceed`, thus causing the unified transaction management. This solution is not only technically, but also textually, free of redundancies. Usage of AspectJ in Java projects can deliver enormous redundancy savings straightaway.

```
aspect TransactionAspect {
  pointcut executeAnyFacadeMethod
      (LgFacade lgFacade):
      execution(public * *(..)) && this(lgFacade);

  Object around(LgFacade lgFacade):
      executeAnyFacadeMethod(lgFacade) {
    final Connection con =
       DatabaseUtil.getConnection();
    try{
      final Object result = proceed(lgFacade);
      con.commit();
      return result;
    }catch(Exception ex){
      report(ex);
      con.rollback();
    }
  }
}
```

Listing 2

Functional programming

Of the many and powerful constructs of Functional Programming I want to demonstrate only one, which facilitates the extraction of control structures. We take the every-day example that a list of persons should be displayed in a special format obtainable by method `getName` of class `Person`. In Java 5 we would need the function in Listing 3 to transform a list of persons into such a format.

A usage would look like:

```
personsToNames(persons)
```

The corresponding transformation in Scala would be so compact that no one would write a special function for this purpose:

```
persons.map(_.getName)
```

This is possible since the function `map` from the Scala collections library contains the above algorithm in a general solution and calls the argument function for each element of the `List`. Using the underscore sign `_` we define a mapping from an anonymous argument to the expression

```
public List<String> personsToNames
    (final List<Person> persons){
  final List<String> names =
     new LinkedList<String>();
  for(final Person p: persons){
    names.add(p.getName());
  }
  return names;
}
```

Listing 3

Sometimes an application needs highly redundant code patterns, but the programming language used does not offer a means to extract them

containing the underscore. The type of the argument is inferred from the element type of `persons` and thus needs not to be indicated explicitly.

In a similar way, in Scala you could guarantee the above-mentioned transaction management. What should be executed as transaction would have to be packed into `transaction{...}`, if the method `transaction` is suitably defined. This solution is technically free of redundancies, but it needs slightly more code than with AspectJ. In contrast, Scala needs only a minimum of keywords in comparison to AspectJ.

Program generators / domain specific languages

Sometimes an application needs highly redundant code patterns, but the programming language used does not offer a means to extract them. In such circumstances, as last resort, you could use a brute-force means: code generation. You define a special language, tailored to the problem, in which you can express yourself without redundancies. From that language you generate program code. Classical examples are decision table code generators like DETAB/65 or parser generators like *yacc*. As an example we give a rule of the contemporary parser generator ANTLR for multiplicative operations. This rule means: A product is a sequence of factors, which are separated by '*' or '/'.

```
product
    :    factor
         ( '*' factor
         | '/' factor
         )*
    ;
```

From this ANTLR can generate a parser which recognizes expressions like `a*b/c*d`. You can expand this parser to an interpreter or translator by inserting actions at the end of each line.

Data storage

Redundancies also cause problems in data storage. An example for this is a table of employees with the columns Id, Name, Date of Birth and Department.

Id	Name	Date of Birth	Department
1	Seyfried, Janina	17.01.1974	Human Resources
2	Stahl, Georg	06.06.1985	Sales
3	Schmidt, Sebastian	26.09.1979	Development
4	Müller, Friederike	19.11.1987	Sales

If the department is indicated as a string for each employee, this constitutes a data redundancy causing the following problems: If there is a typo in a department name, the affiliation of the employee to the department can not be recognized automatically. A renaming of a department necessitates modification of many employee rows.

The redundancy-free solution comprises the management of an additional table for departments, whose rows are referred to by a `departmentId` from each employee. Exactly this is achieved by normalization according to the concept of relational databases.

Other concepts of programming languages

This section lists relevant milestones in evolution of programming languages, which are not useful for redundancy avoidance, but are nevertheless worthy of mention.

- Robustness of programs was boosted by the declaration principle (Algol 58), by the locality helped by the block principle (Algol 58) and the late compilation in conjunction with a linker (FORTRAN, COBOL).
- Coding convenience was boosted by dynamically typed languages (LISP) or by the concedeclaration of variables only at their first usage (C++, 1983), by the freezing of computed values (C++), by 'Garbage Collection' instead of explicit deallocation (LISP, 1958).
- Labour division in development was boosted by the technique of separate pt of static type inference (Scala, 2003).
- Understandability was boosted by comments beginning with full line comments in FORTRAN with C, block comments in Algol with `comment` up to `;`, end of line comments in Ada with `--`, documentation comments in Java with `/**` up to nested block comments in Scala. COBOL pioneered long identifiers significantly helping understandability.

Summary

When you see how painfully the steps of progress in programming were achieved over the last 50 years, you really learn to appreciate the state of the art. Even more interesting is recognizing the driving force behind this progress. High redundancies in source code regularly required new programming constructs. In the majority of cases the ability was introduced to give a freely electable name to the redundant code pattern, and to invoke it with parameters from several locations. This happened to addresses, constants, subroutines, classes and generic units. Sometimes the evolution did not go as far and the redundant code patterns only received new keywords. This happened to formulas, loops, branches, and exception handling. When a programming language helped to eliminate redundancies better than a competing language, this was an advantage in the battle for dissemination. We can assume that this will still be true in future. ■

References and further reading

[DRY] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[Parnas] http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf

[Part1] Christoph Knabe: 'The Eternal Battle against Redundancies, Part I', *Overload* 106, December 2011, accu.org, pp. 6-10

[Refl] http://en.wikipedia.org/wiki/Reflection_%28computer_programming%29

[TemplMeth] http://en.wikipedia.org/wiki/Template_method_pattern