DESIGN PHASE, II: Detailed Design

This part of the lecture is based on chapter 7 of Schach's "Practical Software Engineering". Its objectives are listed below:

- Realize the importance of detailed design
- Appreciate the advantages and disadvantages of using pseudo-code and detailed design
- Recall the necessity for continual testing during product development and maintenance

The first step in the design process, OOD, has been performed. The second step is to refine the components into functions, assign each function to the appropriate compilation unit, and design each function in detail. This step is called **detailed design** and uses the technique of **stepwise refinement.**

Detailed Design

The detailed design document is given to the programming team for implementation, and must therefore include **adequate information for fault-free implementation.**

Thus, not only must the designers specify exactly what each function is to do and how it is to do it, but **they must also unambiguously define the interface between the function and the rest of the product.**

In order for each function to specify how it is to carry out its task, the designers must specify the **data structures** involved and the **algorithms** executed by the function.

Because the detailed design document is usually presented in pseudo-code, and because pseudo-code is generally similar in style to the programming language in which the implementation is to be performed, the detailed design document usually looks somewhat similar to the implementation language.

So we expect, in case of the C-CC corporation, that it looks in appearance C-like, Fig 7.1 below presents the first refinement of function *main* in C-like pseudo-code.

```
main()
{
  do
  {
    display menu;
    get valid_choice from user;
    switch (valid_choice) of
    {
      case 1: input_new_region_record();
      case 2: input_new_shop_record();
      case 3: update_region_record();
      case 4: update_shop_record();
      case 5: delete_region_record();
      case 6: delete_shop_record();
      case 7: produce_vp_report();
      case 8: produce_region_report();
      case 9: produce_shop_report();
      case 0: quit;
    }
  } while user selects valid_choice
}
```

Figure 7.1 First refinement of detailed design of function main.

In no case should the design team *produce* C-code; the life-cycle model would revert to the extremely expensive build-and-fix model. The SQA group can step in if the detailed design document begins look like a complete implementation and suggest to the designers that each function be sketched in pseudo-code, not implemented in detail.

Compilation Unit shop.c

The figure above mentions nine functions as to be defined. The next step is to decide in which compilation unit they will be put.

This leads to the first refinement of compilation units **region.c** and **shop.c** as in fig. 7.2 below.

Compilation unit	region.c	shop.c
Data Structure Name	region_record	shop_record
Data Structure Fields	region_name (2 chars) region_mgr_name (25 chars) region_mgr_address (25 chars) region_mgr_city (25 chars) region_mgr_zip (10 chars) region_mgr_phone (13 chars)	shop_id (3 chars) mgr_name (25 chars) address (25 chars) city (25 chars) zip (10 chars) phone (13 chars) region_name (2 chars)
Functions	input_new_region_record update_region_record delete_region_record produce_region_report produce_vp_report	input_new_shop_record update_shop_record delete_shop_record produce_shop_report

Figure 7.2 First refinement of compilation units **region.c** and **shop.c**.

Next the functions must be defined.

Lets start with *delete_shop_record*; a refinement is given below:

```
void
delete_shop_record(void)
{
  user specifies shop_id of shop to be deleted;
  perform shop deletion;
  print message to user that shop with given shop_id has been deleted;
}
```

Figure 7.3 First refinement of detailed design of function **delete_shop_record**.

This begs the question how a user specifies a shop. leading to the design in fig. 7.4.

```
void
user_specifies_shop(struct shop shop_record)
{
  do
  {
    do
    {
      ask user to specify shop_id;
      see if the shop with that shop_id is in SHOP_FILE;
      if shop does not exist
      {
        tell user that shop is not in file;
        ask user to specify another shop;
      }
    } while user does not specify shop in SHOP_FILE;
    display the shop;
    ask user if this is correct shop to be deleted;
  } while user replies no;
}
```

Figure 7.4 First refinement of detailed design of function **user_specifies_shop**.

In order to define a shop, a *struct* shop data type is introduced, of which only fields mentioned in fig 7.2 are known already. Any other decision is postponed - that's the guiding principle of stepwise refinement: **postpone any decision which isn't immediately needed.**

Fig. 7.4 defines *user_specifies_shop* using 2 loops, one inner and one outer loop. Such loops are always difficult to design properly; absence of desk-testing usually leads to faults, and also does so in this case – in case there are no shops in SHOP_FILE, the user is stuck. This fault is discovered during maintenance.

Since the C-CC corporation uses C, it has no random access files, only sequential files. Consequently, one needs a function reading the shop records one after another, in *shop_id* order, until the wanted one is located.

When one finds the shop sought for, this function returns the value *true*, if a record with *shop_id* $>$ the sought one is found, this function returns *false*.

Such a function is defined in Fig. 7.5 below.

```
int
find_shop (SHOP_ID_TYPE shop_id; struct shop *shop_record_ptr)
{
  read shop records one-by-one;
  if shop_id matches that of shop record being read
    set shop_record_ptr to point to this shop and return TRUE;
  if shop_id is greater than that of  shop record being read
    return FALSE;
}
```

Figure 7.5 First refinement of detailed design of function **find_shop**.

Next functions read shop records one-by-one and write_shop_record are defined in figures 7.6 and 7.7 below.

```
int
read_shop_record (FILE *file_ptr, struct shop *shop_record_ptr)
{
  read from file with pointer file_ptr;
  set shop_record_ptr to point to the resulting record;
  if end-of-file
    return EOF;
  else
    return 0;
}
```

Figure 7.6 First refinement of detailed design of function **read_shop_record**.

```
void
write_shop_record(FILE *file_ptr, struct shop *shop_record_ptr)
{
  write record pointed to by shop_record_ptr to file with pointer
file_ptr;
}
```

Figure 7.7 First refinement of detailed design of function **write_shop_record**.

Calling *read_shop_record* is done by the following calling sequence:

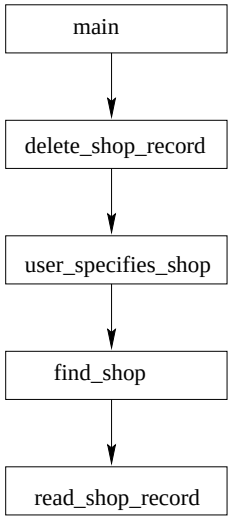

Figure 7.8 Calling sequence of functions designed so far.

Now first the functions called in figure 7.8 are designed. leading to figures 7.9, 7.10 and 7.11.

```
void
delete_shop_record (void)
{
  struct shop_record;

  user_specifies_shop(&shop_record);
  perform_shop_deletion(&shop_record);
  print message to user that shop with given shop_id has been deleted;
}
```

Figure 7.9 Second refinement of detailed design of function **delete_shop_record**.

```
void
user_specifies_shop(struct shop *shop_record_ptr)
{
  do
  {
    do
    {
      ask user to specify shop_id;
      see if the shop with that shop_id is in SHOP_FILE;
      if not find_shop(shop_id, shop_record_ptr)
      {
        tell user that shop is not in file;
        ask user to specify another shop;
      }
    } while user does not specify shop in SHOP_FILE;
    display_shop_record(*shop_record_ptr);
    ask user if this is correct shop to be deleted;
  } while user replies no;
}
```

Figure 7.10 Second refinement of detailed design of function **user_specifies_shop**.


```
int
find_shop(SHOP_ID_TYPE shop_id, struct shop *shop_record_ptr)
{
  FILE *shop_fp;

  open SHOP_FILE with pointer shop_fp;
  if SHOP_FILE does not exist
    print error message and exit;
  set shop_located to FALSE;
  while (read_shop_record(shop_fp,shop_record_ptr) is not equal to EOF)
         and (not shop_located)
  {
    if shop_id matches
      set shop_located to TRUE;
  }
  close SHOP_FILE;
  return shop_located;
}
```

Figure 7.11 Second refinement of detailed design of function **find_shop**. (Design contains a serious fault.)

Actually, a serious fault is made above – in fig 7.11 – because the functions above aren't properly tested.

The first refinement of find_shop, fig. 7.5, is still correct, but the second one isn't.

Situation overlooked in fig. 7.11 is as follows:

1. Suppose the SHOP_FILE contains 2 records, with *shop_id* 111 and *shop_id* 222.

2. Suppose we try to find shop 111 using fig 7.11

3. The first call of *read_shop_record* leads to reading the shop record with *shop_id* 111, so *shop_located* is set to TRUE.

4. So now the while-statement is executed again! Hence the *read_shop_record* is read a 2nd time and the loop terminates because *shop_located* is true.

5. As a consequence the *shop_record_ptr* points now to the record with identifier 222!!!

6. This fault is corrected in fig. 7.12; check how this happens!


```
int
find_shop(SHOP_ID_TYPE shop_id, struct shop * shop_record_ptr)
{
  FILE *shop_fp;
  struct shop temp_record;                                    [added]

  open SHOP_FILE with pointer shop_fp;
  if SHOP_FILE does not exist
    print error message and exit;
  set shop_located to FALSE;
  while (read_shop_record(shop_fp,&temp_record) is not equal to EOF)
         and (not shop_located)                               [modified]
  {
    if shop_id matches
      set shop_located to TRUE;
      set *shop_record_ptr to temp_record;                    [added]
  }
  close SHOP_FILE;
  return shop_located;
}
```

Figure 7.12 Corrected second refinement of detailed design of function **find_shop**.

Detailed design case study: two more functions

Consider again a calling sequence of *delete_shop_record*, see fig. 7.13.

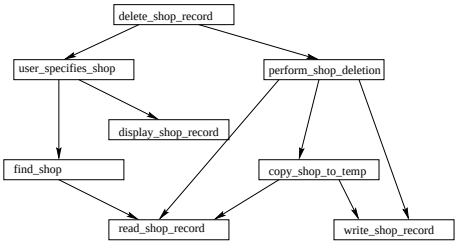

Figure 7.13 Structure chart of functions called from function **delete_shop_record**.

To finish designing this function we design functions *perform_shop_deletion* and *display_shop_record*.

Deleting a record from a sequential file s.a. SHOP_FILE uses the ff. strategy:

- The file is copied to, say, a temporary file *temp_shop_file*, this is done by function *copy_shop_to_temp* in fig. 7.14.

```
void
copy_shop_to_temp(void)
{
  copy the contents of SHOP_FILE, unchanged, to TEMP_SHOP_FILE;
}
```

Figure 7.14 First refinement of detailed design of function **copy_shop_to_temp**.

- Then this temporary file is sought through, copying files **different** from the *file_to_be_deleted* back to the original file SHOP_FILE, as follows

```
void perform_shop_deletion(struct shop *shop_record_ptr)
{
  copy_shop_to_temp();
  while not end-of-file(TEMP_SHOP_FILE)
  {
    read shop record from TEMP_SHOP_FILE;
    if shop_id of this record is not shop_id of
       record pointed to by shop_record_ptr
      write shop record to SHOP_FILE;
  }
}
```

Figure 7.15 First refinement of detailed design of function **perform_shop_deletion**.

- Display shop record is designed

```
void
display_shop_record(struct shop shop_record)
{
  for each field in shop_record
  {
    print name of field;
    print value of that field;
  }
}
```

Figure 7.16 First refinement of detailed design of function **display_shop_record**.

Detailed Design case study: a faulty function

Returning to the first refinement of *main* in fig 7.1, now function *input_new_shop_record* is defined, see fig 7.17 for its first refinement:

```
void
input_new_shop_record(void)
{
  struct shop shop_record;

  get_new_shop_data(&shop_record);
  put_shop_record(&shop_record);
  print(''The following shop record was inserted:'');
  display_shop_record(shop_record);
}
```

Figure 7.17 First refinement of detailed design of function **input_new_shop_record**.

input_new_shop_record calls *get_new_shop_data* and *put_shop_record* and *display_shop_record*, so subsequently the of these are defined, see fig 7.16, 7.18 and fig 7.19.

```
void
display_shop_record(struct shop shop_record)
{
  for each field in shop_record
  {
    print name of field;
    print value of that field;
  }
}
```

Figure 7.16 First refinement of detailed design of function **display_shop_record**.


```
void
get_new_shop_data(struct shop *shop_record_ptr)
{
  for each field in shop_record
  {
    print name of field;
    solicit value of that field from user;
  }
}
```

Figure 7.18 First refinement of detailed design of function **get_new_shop_data**.


```
void
put_shop_record(struct shop *shop_record_ptr)
{
  FILE *temp_shop_fp;
       *shop_fp;
  struct shop temp_record;

  copy_shop_to_temp();
  open TEMP_SHOP_FILE with pointer temp_shop_fp in read mode;
  open SHOP_FILE with pointer shop_fp in write mode;
  set SHOP_INSERTED to FALSE;
  while read_shop_record(temp_shop_fp, &temp_record) is not equal to EOF
  {
    if(temp_record.shop_id matches shop_record_ptr->shop_id)
      and (not shop_inserted)
    {
      overwrite existing record with same shop_id, that is,
        write_shop_record(sop_fp, shop_record_ptr);
      set shop_inserted to TRUE;
    }
    else if(temp_record.shop_id comes after shop_record_ptr->shop_id)
      and (not shop_inserted)
    {
      write_shop_record(shop_fp, shop_record_ptr);
      write_shop_record(shop_fp,&temp_record);
      set shop_inserted to TRUE;
    }
    else write_shop_record(shop_fp, &temp_record);
  }
  [what if the record to be inserted actually belongs at the end of the
  file?]
  if shop_inserted is FALSE
    write_shop_record(shop_fp, shop_record_ptr);
  close SHOP_FILE;
  close TEMP_SHOP_FILE;
}
```

Figure 7.19 First refinement of detailed design of function **put_shop_record**.

Next turn your attention to fig 7.19; it contains a serious error. Can you find it?

- First *copy_shop_to_temp* copies SHOP_FILE to *temp_shop_file*
- Secondly, *temp_shop_file's* records are read one-by-one, comparing the *shop_id* of each *temp_record* to the *shop_id* of the record to be inserted
- Thirdly, if they match the old record is discarded and the new one written to SHOP_FILE. Else, if the *temp_record* has a *shop_id* coming after the *shop_id* of the new record, we've reached the point of insertion. Otherwise, we reach the end of SHOP_FILE and insert it there.

The calling sequence of *input_new_shop_record* is given in fig 7.20.

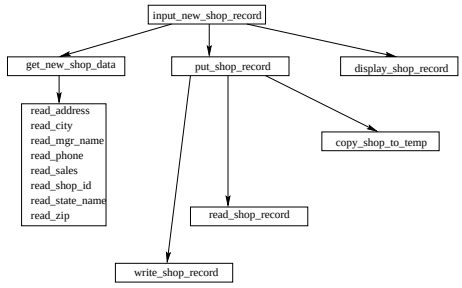

Figure 7.20 Structure chart of functions called from function **input_new_shop_record**

SO WHERE IS THE MISTAKE?

- If there is a *shop_id* match, the old record is overwritten, but this shouldn't always happen. E.g., if shop no. 345 is already in the file, and a new shop with 354 should be inserted, AND A TYPING MISTAKE is made, inadvertedly leading to id 345 having been typed, AN UNINTENDED DELETION OCCURS.
- Such kind of faults tend to be discovered during maintenance!

Detailed Design Case Study: Reusable functions

The third function modifying SHOP_FILE is *update_shop_record*, see figures 7.21, 7.22 and 7.23

```
void
update_shop_record(void)
{
  struct shop shop_record;
  int field_number;

  user_specifies_shop(&shop_record);
  do
  {
    do
    {
      ask user to specify number of field to be updated;
    } while user does not specify valid field number;
    change_shop_field(&shop_record,field_number);
    display_shop_record(shop_record);
    ask user if another field is to be changed;
  } while user wants more fields to be changed;
  put_shop_record(&shop_record);
}
```

Figure 7.21 First refinement of detailed design of function **update_shop_record**.

Update *shop_record* uses *user_specifies_shop* from fig 7.10

- Then the field which must be updated is found out
- Since *user_specifies_shop* calls *display_shop_record*, the details of shop in question are now on screen, and the user can tell the number of the filed-to-be-updated.
- Now the function *change_shop_filed* is called; see fig. 7.22

```
void
change_shop_field(struct shop *shop_record_ptr, int shop_field_number)
{
  ask user to specify new value of field with number shop_field_number;
}
```

Figure 7.22 First refinement of detailed design of function **change_shop_field**.

- the changed shop record is now displayed
- Finally this changed record is written to SHOP_FILE by *put_shop_record* from fig 7.19.

Next the routines reading and displaying shop records are depicted. For this to make sense first the fields of the *struct* shop record are defined. A second refinement is *struct* shop's design is given in fig. 7.24.

Data Structure	*struct* shop
Fields	1. shop_id (3 chars) 2. mgr_name (25 chars) 3. address (25 chars) 4. city (25chars) 5. zip (10 chars) 6. state_name (2 chars) 7. phone (13 chars) 8. target_sales (13 x integer) 9. actual_sales (13 x integer)

Figure 7.24 Second refinement of detailed design of **shop** data structure.

Question: why do *target_sales* and *actual_sales* require 13 matches? To prevent errors. C-arrays start with index 0, so the target sales of May should be stored in *target_sales[4]*. This leads to faults. So better have December's target sales referred to by *target_sales[12]*!

Next *get_new_shop_data* is further refined, see fig. 7.25.

```
void
get_new_shop_data(struct shop *shop_record_ptr)
{
  struct shop shop_record;

  read_shop_id(shop_record.shop_id);
  read_mgr_name(shop_record.mgr_name);
  read_address(shop_record.address);
  read_city(shop_record.city);
  read_state_name(shop_record.state_name);
  read_zip(shop_record.zip);
  read_phone(shop_record.phone);
  set shop_record.target_sales[1 thru 12] to -1; [data not available]
  set shop_record.actual_sales[1 thru 12] to -1; [data not available]
  ask if users wishes to initialize target_sales for year;
  if yes
    read_sales(month,shop_record.target_sales) for month 1 thru 12;
  *shop_record_ptr = shop_record;
}
```

Figure 7.25 Second refinement of detailed design of **get_new_shop_data**.

And the functions that actually read the values for the field, s.a. *read_shop_id* are defined.

In the header file (*definitions.h*) the following definitions are needed:

```
#define shop_id_length 3
Typedef char shop_id_type[shop_id_length +1];
```

and *shop.c* contains the definition

```
struct shop
{
  shop_id_type shop_id;
  ...
}
```

Why do we do this? **For reasons of modularity!** If the numbers of characters in *shop_id* changes, only at one place change is needed!

Why is the identifier *shop_id_type* an array of 4 characters? To terminate every string by the null character '\0', and have storage for that – see fig. 7.26 for some of the required definitions.

Now we turn to **reusability**. Recall from transparencies no. 182, 185, 187 and the discussion in last chapter, the need for reusability – the only way known to reduce maintenance costs effectively!

- Essaress Software Engineering has a reusable function *read_digits(result, length)* reading a line entered at a terminal, examining its first *length* characters and checking whether these are digits. If so, it returns three characters followed by $\backslash 0$ in **result**. If not, the user is informed, an invalid character is entered, and reentering is possible.

- Similar reusable functions are *read_alpha* (for strings of alphabetic characters) and *read_string* (for arbitrary strings).

- Now *read_shop_id*, *read_intg_name*, *read_address* and *read_city* are straightforward to program – see figures 7.27 to 7.30.

- To complete the list, *read_state_name* (which is a little but more complex), *read_zip*, *read_phone* are designed. See figures 7.31, 7.32, 7.33.


```
void
read_shop_id(SHOP_ID_TYPE shop_id)
{
  ask user to specify shop ID;
  read_digits(shop_id,SHOP_ID_LENGTH);
}
```

Figure 7.27 First refinement of detailed design of function **read_shop_id**.

```
void
read_mgr_name(PERSON_NAME_TYPE mgr_name)
{
  ask user to specify shop manager's name;
  read_string(mgr_name,PERSON_NAME_LENGTH);
}
```

Figure 7.28 First refinement of detailed design of function **read_mgr_name**.

```
void
read_address(ADDRESS_TYPE address)
{
  ask user to specify shop street address;
  read_string(address,ADDRESS_LENGTH);
}
```

Figure 7.29 First refinement of detailed design of function **read_address**.

```
void
read_city(CITY_TYPE city)
{
  ask user to specify city name;
  read_string(city,CITY_LENGTH);
}
```

Figure 7.30 First refinement of detailed design of function **read_city**.


```
void
read_state_name(STATE_TYPE state_name)
{
  do
  {
    ask user to specify state (region) name;
    read_alpha(state_name,STATE_LENGTH);
    lowercase_to_uppercase(state_name,STATE_LENGTH);
    if region is not in REGION_FILE
      ask user to specify a state (region) that is on file;
  } while user specifies state (region) that is not in REGION_FILE;
}
```

Figure 7.31 First refinement of detailed design of function **read_state_name**.

```
void
read_zip(ZIP_TYPE zip)
{
  char buffer[BUFFER_SIZE];

  ask user to specify first five digits of shop ZIP code;
  read_digits(zip,5);
  set zip[5] to '-';
  ask user to specify last four digits of shop ZIP code (or 0000 if
  not known);
  read_digits(buffer,4);
  copy first four digits in buffer into zip[6 thru 9];
  set zip[10] to '\0';
}
```

Figure 7.32 First refinement of detailed design of function **read_zip**.

```
void
read_phone(PHONE_TYPE phone)
{
  char buffer[BUFFER_SIZE];

  set phone[0] to '(';
  ask user to specify area code;
  read_digits(buffer,3);
  copy first three digits from buffer into phone[1 thru 3];
  set phone[4] to ')';
  ask user to specify the seven digits of shop phone number;
  read_digits(buffer,7);
  copy first three digits from buffer into phone[5 thru 7];
  set phone[8] to '-';
  copy next four digits from buffer into phone[9 thru 12];
  set phone[13] to '\0';
}
```

Figure 7.33 First refinement of detailed design of function **read_phone**.

The design for *read_sales* (fig 7.34) calls reusable function *month_number_to_string* which converts *month*, an integer between 1 and 12, to string format for display purpose.

Similarly library function *gets*, for getting the line input by the user, and *atoi* (alphanumeric-to-integer) converting a string to an integer, are used.

```
void
read_sales(int month, int sales_array[13])
{
  char line[BUFFER_SIZE];
  char month_string[MONTH_LENGTH];

  do
  {
    month_number_to_string(month, month_string);
    ask user to specify sales for month_string;
    gets(line);
    set sales to atoi(line);
    if sales to atoi(line);
      set sales_array[month] to sales;
    else
      inform user that sales (in thousands of dollars) must be an
      integer in the range 1 thru 9999;
  } while sales specified by user are out of range;
}
```

Figure 7.34 First refinement of detailed design of function **read_sales**.

Detailed Design Case Study: Compilation unit *region.c*

Once compilation unit *shop.c* is finished, compilation unit *region.c* is easy to design. The fields of the *region* data structure are shown in figure 7.35 below.

Data Structure	*struct* region
Fields	1. state_name (2 chars) 2. region_mgr_name (25 chars) 3. region_mgr_address (25 chars) 4. region_mgr_city (25 chars) 5. region_mgr_zip (10 chars) 6. region_mgr_phone (13 chars)

Figure 7.35 Second refinement of detailed design of **region** data structure.

What remains to be done in the design phase?

- The detailed design of the functions producing the three reports is given in the next chapter as an application of *information hiding*.
- The SQA team has to check the design carefully using a review or inspection (see transparencies 53-54).