Software Architectures

Exercise 2

Felix Baumann
Manuel Gottschlich
Alexey Györi 352678
Vincent Wehrwein
Markus Weller

20. Mai 2017

Aufgabe 2.1

a) Abstract data type

In some cases, more than one instances of very similar objects are required. An entertainment database for example can hold a collection of movies, a collections of books and a collection of games. The similarities of these collections can be implemented in an abstract data type where all three collections are derived. Abstract means that the data type is not fixed to one kind of object but can take different forms like in this example movies, books and games. The motivation behind abstract data types is that we can reuse existing implementations in different situations. If we had implemented movies, books and games as abstract data objects in this example we would have had a lot of code duplication and the program would be difficult to maintain.

Example 1: functional language

```
#define SIZE 5

struct Collection {
    int items[SIZE];
    int numberItems;
};

Collection initCollection ( ) {
        StackType stack;
        CollectionType c;
        for(i=0; i<SIZE; i++) {
                c->items[i] = 0;
    }
        return c;
}

void add(Collection c, int value)
{
    c->items[c->numberItems] = value;
    c->numberItems += 1;
}
// more functions to operate with the collectionï£¡
```

Example 2: OO language

```
public class Collection<T>
{
  private T[] items;
  private int size;

  public Collection ()
  {
    items = new T[10];
    size = 10;
  }

  public void add(T t)
  {
    items[size] = t;
    size++;
  }

  // other methods to work on the collection like isFull, isEmpty, remove, get...
}
```

b)

An Abstract data object is a data structure which holds and manages data. The implementation of the ado is hidden in the body, so other parts of the program can use it without depending of the implementation of the module. The function used to access and process the data are defined in the interface of the ado. Besides them ado often have security functions to check if an operation is legal, for example if a stack is not empty.

Example 1: OO language

The following ado in Java holds data about a movie and can print an information about the movie.

```
public class Movie
{
  private String title;
  private int year;
  private String director;

  public Movie (String title, int year, String director)
  {
    this.title = title;
    this.year = year;
    this.director = director;
  }

  public void printMovieInfo()
  {
    System.out.println("The movie " + title + " was directed in " + year + " by "
  }
}
```

Example 2: functional language

The following Ado is a simple counter in C which starts by 0 and counts up, when the function countUp() is called. The interface consists of void countUp() and int getValue().

```
Counter.h:
int getValue();
void countUp();



Counter.c:
int countValue = 0;

void countUp()
{
    countValue += 1;
}

int getValue()
{
    return countValue;
}
```

Aufgabe 2.2

a) Perl implementation

```
#!/usr/bin/perl

sub allInOne
{
        my ($value) = @_;
        return 2*$value+1;
}

sub mult
{
        my ($value) = @_;
        return 2*$value;
}

sub add
{
        my ($value) = @_;
        return $value+1;
}


sub ex13a
{
        my (@list) = @_;
        my @out;

        foreach my $item (@list)
        {
                $item = allInOne($item);
                push @out, $item;
        }

        return @out;
}


sub ex13b
```

```
{
        my (@list) = @_;
        my @out;

        foreach my $item (@list)
        {
                $item = mult($item);
                $item = add($item);

                push @out, $item;
        }

        return @out;
}

my @list = (1,2,3,4,5,6,8,9,8,7,6,5,4,3,2,1);

my @result = ex13a(@list);
print "@result\n";

my @result = ex13b(@list);
print "@result\n";

#
# Prints:
# 3 5 7 9 11 13 17 19 17 15 13 11 9 7 5 3
# 3 5 7 9 11 13 17 19 17 15 13 11 9 7 5 3
#
```

b) Functional Abstraction

To implement this in a cleaner and simpler way we use the concept of **information hiding**. We put the functions allInOne, add and mult in a seperate Perl Module and import this Module in the Perl Script. This helps us seperating the implementation from the usage.

```
package ex13b;

use Exporter qw(import);

our @EXPORT_OK = qw(allInOne mult add);

sub allInOne
{
        my ($value) = @_;
        return 2*$value+1;
```

```
}

sub mult
{
        my ($value) = @_;
        return 2*$value;
}

sub add
{
        my ($value) = @_;
        return $value+1;
}


1;

#!/usr/bin/perl

use File::Basename qw(dirname);
use Cwd  qw(abs_path);
use lib dirname(dirname abs_path $0);

use ex13b qw(allInOne add mult);

sub ex13a
{
        my (@list) = @_;
        my @out;

        foreach my $item (@list)
        {
                $item = allInOne($item);
                push @out, $item;
        }

        return @out;
}


sub ex13b
{
        my (@list) = @_;
        my @out;
```

```
	foreach my $item (@list)
	{
		$item = mult($item);
		$item = add($item);

		push @out, $item;
	}

	return @out;
}

my @list = (1,2,3,4,5,6,8,9,8,7,6,5,4,3,2,1);

my @result = ex13a(@list);
print "@result\n";

my @result = ex13b(@list);
print "@result\n";

#
# Prints:
# 3 5 7 9 11 13 17 19 17 15 13 11 9 7 5 3
# 3 5 7 9 11 13 17 19 17 15 13 11 9 7 5 3
#
```

Aufgabe 2.3

a) classes are one of the main concepts of Object Orientation (OO). There they are used to define methods and data which can then be derived to subclasses. In functional programming languages, it is used mainly for data type definitions of functions.

b) An **imperative language** uses sequences of statements and definitions to compute something. These statements change the current program status as each one is executed. A **functional language** on the other hand uses function definitions and composition of functions to do computations. These functions are stateless and completely-self contained, i.e. having no interference with other functions. Typically functional programming defines what needs to be done and does one composition to achieve its goal.
For example if you need a program to compute the percentage of students from L2P that have registered for the exam:
imperative
open L2P;
count number of students in L2P;
open Campus;

```
count number of registrations for exam in campus;
divide number of exam registrations by L2P students;
```

functional

```
function getNumberOfCampusOfficeRegs;
function getNumberOfL2PStudents;
function getRelative(number1,number2);
getRelative(getNumberOfCampusOfficeRegs(),getNumberOfL2PStudents());
```

c) A module is a logical unit. Containing a body which is hidden outside of the module and an interface which is to be used by clients and visible to the outside. Modules may be sub-programs, data structures, or just basic functions. The architecture part defines the concrete implementation of the module.

d) A function returns values, whereas a procedure has no return value (return type = void in Java).

e) Functional abstraction modules transform input to output data and hide the operations, they **do not** have a state. Data abstraction modules show the operations and hide the data representation (and also operation realization), they have a state.