Abstract. This seminar work is about the paper "Program Design by Informal English Descriptions" written by Russell J. Abbott and published in 1982. It summarizes the ideas of this paper by applying the introduced method on an own example. In addition it gives an insight in further developments in this particular field of Requirements Engineering and related topics like natural language processing.

Keywords: Requirements engineering, informal descriptions, program design, automation, natural language processing

1 Introduction

Software developers often face the problem, that they need to create a program structure from a given informal description of a problem. It is their goal to extract different program components from this given description. These components might for example be datatypes, objects or operations. With his paper Abbott now offers a procedure to fulfill this task [1]. The final aim of this procedure is the implementation of an Ada Program, which solves the initially given informal task. One of the key ideas of Abbott’s method is the use of nouns and noun phrases for datatypes and objects. Abbott’s attempt is quite intuitive and straightforward (it is divided in 3 main steps). This fact makes his method applicable for software development in practice. In the following section I want to describe his method in detail. Afterwards I will show a further attempt for program design which offers a higher degree of formalization. Moreover I will describe approaches for the automation of processing informal descriptions.

2 Abbott’s method

Russell J. Abbott proposes the following approach for the design of a program given an informal english description:
1. Development of an informal strategy which solves the given problem on the same informal level as the stated description.

2. Formalization of the created strategy in step 1. This is done by defining data types, objects, operators and control structures.

3. Segregation of the solution from step 2 in an Ada specific manner: Building a package and one or more subprograms.

I will now introduce an own example problem and use Abbott’s method to design a program for solving the problem:

*Write a program that can schedule an arbitrary number of jobs on an arbitrary number of machines.*

The aim is now to design a program which can fulfill this task (note that it is not an optimal solution wanted, in general this is also not possible for the given problem). I will start now with the first step of the method in the next section.

### 2.1 Informal Strategy

This is the informal strategy for solving the given example description:

**I.** Do the following steps for all given jobs:

1. Determine on which machine the job must be processed.
2. Get the information of the machine at which timepoint it is free for a new job (we call this the machines workload timepoint) and save this timepoint.
3. Plan the job on the machine at this timepoint.
4. Change the workload timepoint of the machine to the saved timepoint plus the time length of the job.

**II.** Create a schedule from the information of the machines and return this schedule.

This is the informal solution strategy which solves the given informal problem description. In the following section we will go on with Abbott’s method and formalize this strategy.

### 2.2 Formalization of the strategy

This part of the method can be considered as the one which needs the most effort. It is divided into the following sub parts:

1. Identify the data types.
2. Identify the objects for the program.
3. Identify the operations on these objects.
4. Integrate these found structures in a more formal solution strategy.
Data types can be found by looking for common nouns in the informal strategy. A common noun is defined as a name of a class of beings or things. Examples for common nouns are "book", "animal" or "city". So what we need to do now is to look for common nouns in our informal solution strategy. By doing this, we find the following:

- step
- job
- machine
- information
- timepoint
- workload
- length
- time
- schedule

These are all common nouns found in the informal solution strategy. It is important to mention that those common nouns are not all necessary data types in our final program, they are only candidate data types.

In the next step we will identify candidates for objects of the program. These can be found by looking at all proper nouns and direct references. A proper noun is a name of a specific being or thing. Proper nouns are for example: "bible", "tiger" or "Konstanz". Direct references also specify a being or thing like for example "the book" or "this city". Applying these patterns on our example we retrieve the following:

- the following steps
- all given jobs
- the job
- the machine
- the information
- the machines workload timepoint
- this timepoint
- the time length
- this schedule

The found terms are again only candidates for objects in the final program. In the next step we are going to determine the operation candidates for our program. Candidates for operations can be found by looking at verbs, attributes, predicates and descriptive expressions. Doing this we find these candidate operations:

- Do the following steps
- Determine on which machine
- Get the information of the machine
- safe this timepoint
- Plan the job
- Change the workload timepoint
- Create a schedule
Now we are able to create a more formalized solution strategy for our problem. Therefore we can use the found structures. Doing this we end up with the following formalized strategy:

```java
for each Job j in jobs
    do
        Machine m := j.getMachine()
        Timepoint t := m.getWorkloadTimepoint()
        planJob(m, t)
        m.workloadTimepoint := timepoint + j.length
    end for

Schedule s := new Schedule
for each Machine m in machines
    do
        if(m.getWorkloadTimepoint() not 0)
            s.add(m)
    end for
return s
```

We have finished the second step of the method by stating a formalized strategy for solving our problem. In the Next section we will go on with step 3 and build a program structure for our example problem.

### 2.3 Segregation and structuring

In this section I will use the programming language Java (instead of Ada) to perform a segregation and structuring of the formal strategy. Therefore I will describe several Interfaces which must be implemented by Java classes to solve the given task. Here are the created Interfaces:

```java
package scheduling.datatypes;

import java.util.List;

public interface Machine {
    /**
     * Returns the actual <code>workloadTimepoint</code> of the machine,
     * this is the timepoint when the machine ends
     * with its last planned job
     */
    public Timepoint getWorkloadTimepoint();
}
```
/**
 * Sets a new `<code>`workloadTimepoint</code>*
 */
public Timepoint setWorkloadTimepoint(Timepoint t);

/**
 * Returns a List of all jobs planned on the machine
 */
public List<Job> getJobs();

/**
 * adds a `<code>`job</code> to the machine
 */
public void addJob(Job job);

}  

public interface Job {

/**
 * Returns the `<code>`machine</code> on which the job should be
 * scheduled
 */
public Machine getMachine();

/**
 * Returns the `<code>`length</code> of the job
 */
public int getLength();

}  

public interface Timepoint {

/**
 * Returns the actual time object
 */
public Date getTime();

}
package scheduling;

import scheduling.datatypes.Machine;

public interface Schedule {
    /**
     * Adds a machine on which at least one job is planned
     */
    public void addMachine(Machine m);
}

public interface Scheduler {
    /**
     * Plans a job on a machine
     */
    public void planJob(Machine m, Job j);

    /**
     * The main method which creates a schedule in the end
     */
    public Schedule createSchedule(List<Job> jobs, List<Machine> machines);
}

The main logic from the formalized strategy can now be implemented in the method createSchedule(List<Machine> machines, List<Job> jobs) which gets a list of machines and a list of jobs as input. So in the end we retrieved the programming structure from our formalized method. But it also can be seen, that on the one hand we did not use all candidates for methods or objects and on the other hand we introduced datatypes which were not needed in the formalized method (like for example the Interface "Scheduler").

2.4 Conclusion

The method which Abbott presents is quite short and therefore very intuitive. Software developers can profit from his method as it is very easy to use. On the other hand it even seems to be to general for allowing approaches of automation. For example relations between different datatypes are not covered at all by Abbott’s method. Another problem might be that an exclusive top down approach seems to be unrealistic. This means that it is probable that there must be performed changes on a lower level (for example in the implementation) which also have impact on higher levels like the formalized strategy. In the following
sections I want to show up further developments which offer a higher degree of formalization and also automation.

3 Further Formalization

There have been further developments in the field of processing informal specifications with the goal of getting program structures. One of these approaches I want to describe a little bit more detailed. This approach was published by A. M. Moreno in 1997 [3]. Moreno’s method consists of the following 9 stages:

1. Extraction of essential informations
2. Identification of synonyms and polysemies
3. Separation of static and dynamic information
4. Static requirements structuring
5. Dynamic requirements structuring
6. Object Model construction
7. Behavior Model construction
8. Integration of both models
9. Verification of both models

One key method Moreno uses for her approach is the break down of the given informal requirements to a so called utility language. Ones the requirements are transformed into this utility language one can use different linguistic patterns to find the corresponding conceptual patterns for the constructed model. I want to give some examples for this utility language transformation:

1. Substitute pronouns with the noun or noun group which they replace
2. Discard adverbs and modal adjectives (probable, necessary, ...)
3. If a modifier refers to several Nominal Syntagmata, that reference will be made explicit in each one

Informal requirements which have been preprocessed by this utility language can easier be mapped into the corresponding conceptual patterns like for example inheritance. It is clear that by this approach the relations between components in the generated model are identified. The strength or weakness of this approach lies in the used mapping from linguistic patterns to conceptual patterns. Abbott’s method is way more general and does not use any relation mappings.

4 Approaches for automation

It comes up the question, whether it is possible to automate the process of building models from informal descriptions. For being able to do this, methods of natural language processing must be used. By the so called Part of Speech (PoS) tagging every term of a sentence is tagged with its word class (for example verb, noun, adjective,...) in the context of the sentence. There are different methods for automatically determine the PoS tag for a certain term. One is to use a
dictionary and assign each term the PoS tag which has the highest probability for this term. Other approaches also consider the tag before the actual term for getting more accurate results. After having performed a PoS tagging on an informal description, it would be possible to look for certain linguistic patterns which themselves again could be mapped to conceptual patterns. Another possible approach might be to use semantic networks like for example WordNet to determine the semantical relation between terms and by this being able to automatically find their conceptual relation in a model [2].

5 Summery

As already mentioned, Abbott’s method is very intuitive and therefore easy to use by developers in practice work. Moreno offers a method which is more formalized and so offers better possibilities for automation. By natural language approaches like the explained Part of Speech tagging, an automation (or at least semi automation) in the process of program design seems to become more and more reachable, although human real world knowlegde is still needed. Technologies like semantical networks might be useful for filling this gap.

References