Problem Definition

5th Lecture

1. Introduction

The ***first step in the problem solving and decision making process is to identify and define the problem.*** Therefore, before a program is written for solving a problem, it is important to define the *problem clearly*. For most software projects, systems analysts approach the user requirements and define the problem that a system aims to solve.

They typically look at the following issues:

- What is the problem?
- What is the input required for supporting the solution process?
- What is the expected output of the solution?
- How do people solve the problem currently?
- Can the problem or part of the problem be more effectively solved by a software solution?

For most programmers, they work in accordance with the design specification produced by the *system analysts/designers*. A design *specification describes the program input, output and functionality of the programs of the system*. Note that the program input may be entered by the user through any input devices or retrieved from data stores (in terms files or database) whereas program output may be stored, printed or displayed on screen, etc. In addition to verifying the correctness of program input, output and functionality, it is important to ensure that any planned user-computer interactions be approved by the users.

2. Computational Problem

In *theoretical computer science*, a **computational problem** ***is a mathematical object representing a collection of questions that computers might be able to solve***.

For example, the problem of factoring "**Given a positive integer n, find a nontrivial prime factor of n**." is a computational problem.

Computational problems are one of the main objects of study in theoretical computer science. The field of ***algorithms*** *studies methods of solving computational problems efficiently.*

A computational problem *can be viewed as an infinite collection of instances together with a solution for every instance.*

For example, in the factoring problem, the instances are the integers ***n***, and solutions are prime numbers ***p*** that describes nontrivial prime factors of ***n***.

Types of computational problems

- A **Decision Problem** is a computational problem where the answer for every instance is either yes or no.

  An example of a decision problem is primarily testing:

  "**Given a positive integer n, determine if n is prime**." A decision problem is typically represented as the set of all instances for which the answer is yes. For example, primarily testing can be represented as the infinite set L = {2, 3, 5, 7, 11, ...}

- In a **Search Problem**, the answers can be arbitrary strings.

  For example, factoring is a search problem where the instances are (string representations of) positive integers and the solutions are (string representations of) collections of primes.

  A search problem is represented as a relation consisting of all the instance-solution pairs, called a search relation.

  For example, factoring can be represented as the relation R = {(4, 2), (6, 2), (6, 3), (8, 2), (9, 3), (10, 2), (10, 5)...} which consist of all pairs of numbers (n, p), where p is a nontrivial prime factor of n.

- A **Counting Problem** asks for the number of solutions to a given search problem.

  For example, a counting problem associated with factoring is

  "**Given a positive integer n, count the number of nontrivial prime factors of n**."

  A counting problem can be represented by a function ***f*** from {0, 1}* to the nonnegative integers. For a search relation ***R***, the counting problem associated to ***R*** is the function $f_R(x) = |\{y: R(x, y)\}|$.

- An **Optimization Problem** asks for finding a "**best possible**" solution among the set of all possible solutions to a search problem.

  One example is the maximum independent set problem: "**Given a graph G, find an independent set of G of maximum size**." Optimization problems can be represented by their search relations.

- In a **Function Problem** a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just "yes" or "no".

  One of the most *famous examples* is the travelling salesman problem: "**Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city**."

  When Can You Use a Problem Definition?

  A problem definition is usually set up at the end of the problem analysis phase.

3. How to Use a Problem Definition

Starting Point

The starting point of a problem definition is the information gathered in the problem analysis stage. The different aspects surrounding the design problem have been analyzed and should be taken into account in the problem definition.

Expected Outcome

A structured description of the design problem, with the goal of creating an explicit statement on the problem and possibly the direction of idea generation. Also, a problem definition clearly written down provides a shared understanding of the problem and its relevant aspects.

Possible Procedure

Answering the following questions will help to create a problem definition:

1. What is the problem?
2. Who has the problem?
3. What are the goals?
4. What are the side-effects to be avoided?
5. Which actions are admissible?

Tips and Concerns

- When analyzing problems there is always a tension between the 'current situation' and the 'desired situation'. By explicitly mentioning these different situations you are able to discuss the relevance of it with other people involved in your project.
- ***Make a hierarchy of problems***; start with a big one and by thinking of causes and effects, divide this problem into smaller ones. Use post-its to make a problem tree.
- A problem can also be reformulated in an opportunity or 'driver'. Doing this will help you to become active and inspired.

4. Planning and Design of Solution

4.1. Top-down Approach

A top-down approach starts with the broadest and the *most general level of the problem*, then works down at the next level with more *specific details given*. The step may repeat in the detailed level until the description of the bottom level is *detailed enough* to map to the selected programming language for implementation. Naturally, a hierarchy of system components is formed as a result of the top-down design approach.

In program development, one of the most well-known top-down development approach is known as the *functional approach* or *functional design*. **Functional design** is described as "***assures that each modular part of a computer program has only one responsibility and performs that responsibility with the minimum of side effects on other parts***". The approach focuses on the functional aspect of a system and proposes different components to implement those functions.

The steps for the top-down approach are as follows:

i. Specify the problem clearly
ii. Break the problem into smaller sub-problems
iii. Repeat Step 2 for each sub-problem until it is small enough to be solved by a software implementation.

4.2. Bottom-up Approach

In brief, a bottom-up approach specifies details of ***individual system components***. The components are then ***linked together*** to form larger parts, which are in turn linked until a complete system is formed.

Bottom-up emphasizes coding and early testing, which can begin as soon as the first module has been specified. This approach, however, *runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. Re-usability of code is one of the main benefits of the bottom-up approach*.

Bottom-up approaches did not receive as much attention as the top-down approaches. The problem was due to the fact that constructing reusable software components is very difficult. The situation sustained until object-oriented approaches to software development become popular. Object-oriented languages like *C++ and Java* enable programmers to practice bottom-up design at ease with features like inheritance.

<Best Regards>

Dr. Raaid Alabady