Basic Concepts

CMPT 115/117 lecture slides

Notes written by Mark Eramian, Ian McQuillan, Michael Horsch, Lingling Jin, and Dmytro Dyachuk

Obectives

By the end of this lecture topic, you are expected to be able to:

1. Describe the process of abstraction
2. Give examples of abstraction
3. Describe the process of top-down design (stepwise refinement)
4. Give examples of top-down design (stepwise refinement)
5. Define the terms C. E. R. A. R.
6. Describe the benefits of data organization.

Part I

Abstraction

Outline

Abstraction

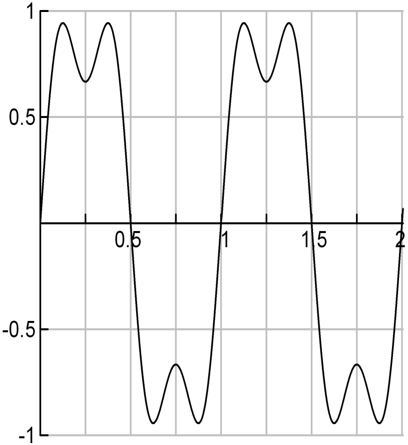
Definition

Abstraction is the process of extracting or distilling the underlying essence or important properties of a concept, removing some or all dependence on real world objects with which it might originally have been connected.

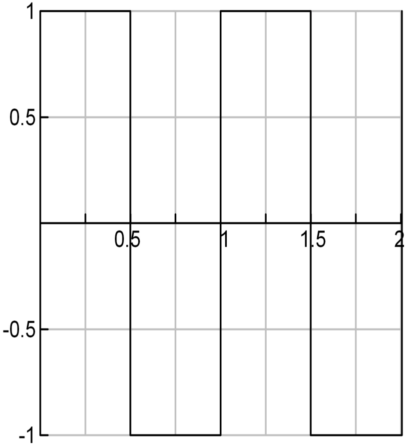
Example: bits

- Information is represented electronically in a computer by voltages, at different levels.
- If the voltage is "high", a computer interprets this as a "1". If the voltage is "low", then a computer interprets this as a "0".
- However, voltage is measured on a continuous scale, not just in two distinct states.

Abstraction: voltages $\Rightarrow$ states



$\Rightarrow$



Levels of abstraction
Exercise

- Give a one-word abstract description of each of the given code.

Levels of abstraction

High(est) level

Purpose or main point only

Sort student numbers ← Purpose

Input: an int array
Output: sorted array ← Specification

More abstract
Less detail

```
void sort(int a[], int size)
{
 // code omitted for brevity
}
```

← Implementation

Purpose described
with some details included

```
load 01246
store 01242
load 01242
compare 0
jump 88762
...
```

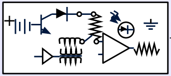
← Assembly code

```
0100101011
1010011011
0110100110
1010110000
1101101001
0101101101
...
```

← Machine code

Less abstract
More detail

Purpose fully realized
with all details included



← Voltages

Low(est) level

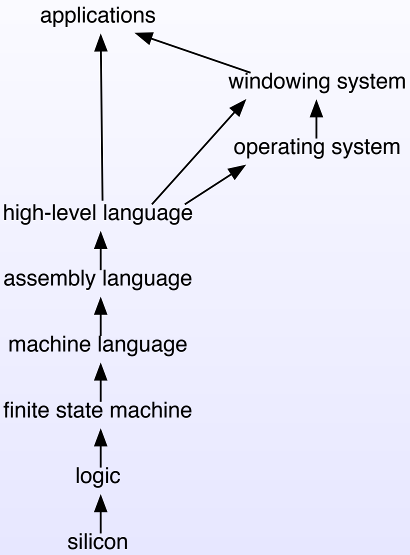
Building abstraction hierarchies

- When we have some abstraction, we can build another abstraction on it.
- The new abstraction is at a *higher* level of abstraction.
- Example: natural sciences

A Hierarchy of scientific abstraction ??

physics, chemistry, biology

Computer system - levels of abstraction



Top-down design (stepwise refinement)

- Software programs and applications may be the most complex artifacts ever created by human beings.
- To manage the complexity, we need strategies!
- When designing large applications, software engineers have found it often easier to design starting at a higher level of abstraction first.

Primary design strategy

- Design top-down.
- Build bottom-up.

Top-down design example

Let's say that we would like to cook spaghetti.

Example

We could break that down into two main tasks:

1. boil water
2. cook spaghetti

Top-down design example

Example

Now *boil water* can be further broken down into more subtasks:

1. pour water into pot
2. put pot on stove
3. turn on stove

Then *pour water in pot* can be further broken down into

1. get pot from cupboard
2. put pot under faucet
3. turn on faucet until full

and so on.

At each step, we refine our tasks into lower-level subtasks.

Top-down design is a proven strategy

- Another example: when designing a web browser, it can be easier to determine, at a high level, what it could do.
- You could describe functions as *goBack, goForward, reload, stop, goToAddress*, without coding them.
  - For each function, determine the information it needs, and what it will do.
  - Design how the functions should interact.
- When you have a pretty good idea that you know how the functions should work, then you can write the code for them.

Discussion: Top-down Design

Use top-down design to plan a vacation. Work in groups of 3-5.

Part II

Software Design Goals

Outline

Application design

- Building a program/application can be an extremely difficult and expensive task.
- It is difficult enough to write one hundred lines of code that works *correctly* and *efficiently*.
- Imagine a standard kind of application, e.g., a web browser, or an operating system, with millions of lines of code.
- Imagine writing code with a team of one hundred people.

Application design

- If we are going to achieve this goal, then we must be extremely organized.
- There is too much room for error otherwise.
- We must develop a set of principles by which we design our software, and live by them.
- If we don't, we may as well give up now.

Design goals

Two important goals in software design:

1. Code is correct
   - Works as intended; given any input, the software produces the desired output.
2. Code is efficient
   - Uses resources, including time, effectively.

It is possible to make choices that drastically affect correctness and efficiency before you've even written a single line of code.

Implementation goals

There are three additional goals which software implementations should achieve:

1. robustness (pg. 23),
2. adaptability (pg. 24),
3. reusability (pg. 25).

Robustness

- A program should produce the correct output for all inputs (correctness).
- It can also handle unexpected inputs (robustness).
- Examples:
  1. A program stores a class database, and it declares an array of size 100. What if someone enters more than 100 students?
  2. A program asks for a date as "DAY MONTH YEAR," but gets "2015 31 2". What should it do?

Adaptability

- It is a big problem if an application needs to be completely rewritten to add some desired functionality.
- Good software is able to adapt to new or changing purposes.
- When changes are needed, good software is able to *evolve* in response to changing conditions or purposes.
- Small changes here and there should not require larger changes everywhere.
- In a large scale application, most of the code stays unchanged for most of the time.

Reusability

- We should be able to use some of the code written for Project A as a component of Project B.
- This is a **massive** time saver for software developers.

Data structures

- When dealing with complex problems, it is important to store and manipulate data in an *organized* fashion.
- Roughly speaking, a *data structure* is a systematic way of both organizing and accessing data.
- We've learnt a few different data structures in CMPT 111, such as
    - integers, characters, floats (atomic data!)
    - arrays,
    - new record types using `struct`.
- We're going to see several important data structures.
- We will emphasize all the design principles we have mentioned.

Data organization - implementation principles

- Our data should be stored *correctly*, should be efficiently retrieved and manipulated.
- We should be able to change the way we store the data, and not have to rewrite everything.
- We should be able to reuse our data structures for use in different systems.