Software Development 1: Introduction

1. **Introduction**
   - **software development and software engineering**
   - **software requirements and software design**
   - **abstraction**
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. Testing
6. Debugging

Software Development

Software development refers to the process of creating a *correctly functioning* **software system**.

A *software system* consists of one or more **computer programs** along with needed configuration/data files, documentation, etc.

Development obviously involves **implementing** the program(s) in suitable **programming language(s)**.

However, the development process also generally involves other important elements as well:

- **requirements specification**
- **design**
- **testing**
- **documentation**

Software Engineering

Software engineering is the area of computer science that deals with formal *methodologies* for *software systems development*.

A fairly large number of *SE methodologies* are in use, including:

- waterfall model
- test-driven development
- agile development
- extreme programming
- rapid application development (RAD)
- scrum development
- lean development
- dynamic systems development method (DSDM)

Most of these methodologies were designed for the development of *large-scale* software systems, by **teams** of developers.

Incremental Software Development

An important characteristic that many SE methodologies share is that the software system is developed **incrementally**–i.e., in **stages/phases**.

In some, development proceeds from highly *abstract* versions of the entire system to increasingly more *concrete* versions.

In others, development proceeds by incrementally adding working components.

Incremental development may be part of an **iterative process**, where the implementation and even design are successively refined.

Incremental development will be a key topic in these lectures!

Software Requirements Specification

For most applications, it makes little sense to start building a software system until one understands precisely how it is supposed to function.

Important elements of a **software requirements specification** are the **user interface** design and the **input-output behavior** (functional requirements).

Other important elements can include *performance* requirements (e.g., time required to process amount of data), *error handling* behavior, and the *operating system(s)* and computer/networking *hardware* the software must function on.

The SRS may also specify a particular *programming language* if the new system must interface with an existing system.

Software Design

Program **implementation** (writing of code) does *not* commence the instant a software requirements specification is delivered.

Instead, there is a **design** process that determines:

- the **logic** necessary to achieve the spec's goals
- the decomposition into **programs** and **compilation units**
- the decomposition into **modules** (**functions/methods**)
- the **libraries** or calls to be used for particular functionality

Failing to spend adequate time developing a good design typically results in a time-consuming development process due to numerous bugs that are difficult to eliminate.

Even if poorly-designed software is eventually made to work correctly, it will likely be hard to understand and costly to maintain.

Abstraction (in Design and Implementation)

Abstraction is a key mechanism for dealing with the *complexity* of programming.

Abstraction means: *limiting the amount of detail that we must consider at any one time.*

The primary abstraction mechanism for *control/logic* is the use of **functions/methods** (which give a name to a block of code).

Functions/methods also allow us to avoid **code duplication**.

The primary abstraction mechanisms for *data* are **composite data types** (which allow a set of primitive data elements to be treated as a unit).

Abstraction is what enables **incremental software development**.

Abstraction (contd.)

To better understand the role of abstraction in simplifying the development process, consider that we could implement any (non-recursive) program as a single `main`.

Of course the `main` may be huge and difficult to understand, and may also contain significant amounts of (largely) duplicate code.

Breaking the code down into a set of functions/methods permits us to limit the length and complexity of any one module, as well as to avoid code duplication.

Abstraction (contd.)

Consider a `main` containing multiple **code fragments**:
(a code fragment is a sequence of several lines of code)

```
int main()
{
  ...variable declarations...
  ...code fragment A1...
  ...code fragment B...
  ...code fragment C1...
  ...code fragment A2...
  ...code fragment B...
  ...code fragment C2...
}
```

(Notation like `A1` and `A2` indicate the code is largely the same, differing only in the values of some variable(s).)

Abstraction (contd.)

We can simplify `main` by defining functions/methods for each code fragment, containing the fragment's code, along with some appropriate parameters:

```
int main()
{
  ...variable declarations...
  func_A(1,...);
  func_B(...);
  func_C(1,...);
  func_A(2,...);
  func_B(...);
  func_C(2,...);
}

function func_A(x,...)
{
  ...code fragment A(x)...
}
...
```

Abstraction (contd.)

This could nearly literally be our entire new `main`—much shorter and thus much easier to understand, than the original.

It would be an *abstraction* of the original `main`, because we are ignoring the code necessary to implement each of the functions.

(Of course this does depend on the programming language allowing any necessary data to be exchanged among the functions/methods, since their code will no longer be in a shared **scope** as it was in the original `main`.)

Another advantage of this sort of abstraction is that the functions can be implemented (and even tested) *independently* from each other and from `main`.