TDDE45 - Lecture 4: (Cross-Platform) Software Construction

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-09-12

Part I

Introduction

LINKÖPING UNIVERSITY

Software Engineering

1. Software Requirements (TDDE41 Software Architectures)
2. Software Design (TDDE41, **TDDE45**)
3. Software Construction (**TDDE45**)
4. Software Testing (TDDD04)
5. Software Maintenance

10. Software Quality (TDDE46)

The areas are chapter numbers in SWEBOK (IEEE Computer Society, Bourque, and Fairley 2014). `https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3`

Software Design vs. Construction

Software Design

- Software design principles
- Software structure and architecture
- Design patterns
- User interface design
- **Localization and internationalization**
- Concurrency, security
- Software design notations (incl. UML)

Software Construction

- Minimizing complexity
- Anticipating change
- **API Design and Use**
- Coding, construction for reuse
- Development environments
- Unit testing tools
- Profiling, performance, analysis

(Some of the) definitions from SWEBOK (IEEE Computer Society, Bourque, and Fairley 2014). `https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3`

Part II

API Design

Direct System Calls

```
_start:
  movq $1, %rax    ; use the write syscall
  movq $1, %rdi    ; write to stdout
  movq $msg, %rsi  ; use string "Hello World"
  movq $12, %rdx   ; write 12 characters
  syscall          ; make syscall

  movq $60, %rax   ; use the _exit syscall
  movq $0, %rdi    ; error code 0
  syscall          ; make syscall
```

Making direct system calls to the operating system.

Operating System Layer

```c
int main(int argc, char **argv) {
  int dest_fd, src_fd;
  src_fd = open("a", O_RDONLY);
  dest_fd = open("b", O_WRONLY | O_CREAT);
  if (ioctl(dest_fd, FICLONE, src_fd) < 0) {
    fprintf(stderr, "error: %s\n", strerror(errno));
    return 1;
  }
  return 0;
}
```

Using the OS-specific ioctl function.

This code will reflink the two files if the file system supports it (IDA computers use networked file systems that do not support copy-on-write).

Standardized APIs or libraries

```
#include <stdio.h>
int main(int argc, char **argv)
{
  FILE *f = fopen(argv[1], "r");
  fprintf(f, "Hello World!\n");
}
```

POSIX System Interfaces (and/or the C standard depending on your point of view). Standardizing many common OS operations. Note that POSIX standardizes more than only APIs – also common concepts like locales and shell utilities.

Standardized APIs?

```
/* In POSIX since issue 5; works in Linux */
#include <time.h>
int clock_gettime(clockid_t clock_id, struct timespec *tp);
/* OSX 10.5 was certified Unix 03 (2007), which includes POSIX issue 6.
   But clock_gettime was not supported until MacOS 10.12 (2016).
   So need to use Apple-specific API, which is not
   documented in man-pages.
 */
#include <mach/mach_time.h>
uint64_t mach_absolute_time();
/* Windows does not pretend to support POSIX.
   (Although WSL allows you to program as if on Linux) */
BOOL QueryPerformanceCounter(
  LARGE_INTEGER *lpPerformanceCount
);
```

Sadly, POSIX is not always followed.

Writing portable programs requires a lot of reading and careful coding and a lot of testing. https://blog.kowalczyk.info/article/j/

Cross-platform libraries

```
#include <QElapsedTimer>
/* Since Qt 4.7 (2010) */

void f() {
  QElapsedTimer timer = QElapsedTimer();
  timer.start();
  // ...
  qint64 elapsed = timer.elapsed();
}
```

Cross-platform libraries such as Qt aim to supply the user with the functionality they need without too much platform-specific code.
Most modern programming languages also have libraries that aim to be cross-platform (even C#, but you have to use the API to construct paths; most code tends to use strings and hard-codes Windows \ as directory separator).

So you just use Qt, right?

Qt is huge (if you use everything).

> *I am trying to install Qt 5.9.4 for windows. Install says it needs 35.54Gbytes for downloading. That seems like a lot making me think that I did something wrong. Is this normal? I deselect everything but 5.4.9 and thats how i got it down to 35G.*
>
> `https://forum.qt.io/topic/87608/windows-install-download-size`

It is of course smaller if you only install the libraries for Linux or only for MINGW Windows. But you still need to maintain an installer for the used Qt libraries, which might cause you to not update your Qt version for a long time (especially since newer versions take a long time before they are supported on MacOS and do not support older MacOS versions).

And you may not ship executables statically linked against Qt (LGPL).

There are other issues as well...

Qt is a framework

Yes, QtCore is a C++ library. But Qt is not C++!

Qt programs use special tools to preprocess source code, create resources, prepare translations, etc.

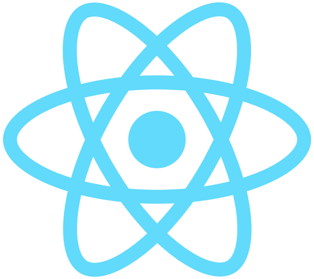
There are different definitions of what a framework is:

- A collection of tools (not just a library) form a framework.
- A library that uses inversion of control.

The Qt GUI functionality uses inversion of control (as does most GUI frameworks).

React Native is another framework

- Based on React (JavaScript framework for building GUIs)
  - Does not quite feel like programming basic JavaScript
  - Similar to Qt vs. C++
- React Native does not use HTML or DOM; instead the native widgets



Some cross-platform libraries

Good design encapsulates different cross-platform behaviour in a single library (cf. adapter pattern). Better design re-uses a library someone else wrote. Usually the standard library for your programming language or perhaps one of these C/C++ libraries:

- C++ STL. Standard library that has been growing. Still lacks much functionality even in C++17.
- Boost. A playground for adding new functionality to STL. Tends to not be backwards compatible with itself.
- QtCore (C++, LGPL). Data structures and common OS functionality.
- GLib (C, LGPL). The non-GUI part of GTK+; similar to QtCore/Qt. Adds its own object orientation to C.
- Apache portable runtime. OS functionality written in C.
- POSIX threads for Win32. If you like pthreads and only need rudimentary multi-threading.
- Intel Threading Building Blocks. A C++ library that promises a scalable, future-proof threading abstraction.
- React (JavaScript)

Note that each of these have advantages or disadvantages. There is no silver bullet.

Part III

Cross-platform issues

LINKÖPING
UNIVERSITY

Cross-platform development

If your code only runs on a single platform, you reach a limited number of customers and you are dependent on the vendor of this particular system.
It is thus **desirable** to make sure your code runs on as many of the following platforms (it is probably not possible to run it on all platforms if you do advanced things):

- Windows (largest user base)
- Linux (this is usually the easiest to program for)
- MacOS
- *BSD
- Android
- iOS

Porting your code on different platforms also helps with testing as bugs tend to reveal themselves more easily.

Some things are easier than others for cross-platform

There are applications that tend to work on both mobile and desktop platforms:

- Game engines such as Unity and Unreal (games simply draw on an OpenGL surface or similar).
- Libraries such as FFMPEG are used to build mobile applications on top of them.
- Then there are pretty much no more examples ☹
  - At least for desktop applications; there are web-based applications that work in both...

MacOS

- Is Unix-based and as such many things works the same as Linux.
- Does not support the same rpath/runpath as Linux.
- Mac executables (apps) are supposed to be able to move freely, but all the tools by default assumes absolute paths for everything.
- Deprecation of common standards such as OpenGL (followed by quickly breaking).
- No freely available cross-compilers.
  - Basically requires you to use Apple hardware, which is expensive and does not rack mount.
- No container support (think docker).
  - Requires you to use buy one machine per OS version you want to test and support.
- Requires extensive changes whenever new XCode versions are released (not maintaining backwards source-code compatibility; only for binaries).
- Works better if you only use XCode and no cross-platform development.

The filesystem

The path separator tends to be either / (Unix) or \ (Windows).
Some programming languages or libraries try to abstract this away; others force you to change your code.
Try to use functions like:

```
joinpath(foo, bar, baz)
```

Rather than:

```
"$foo/$bar/$baz"
```

Also, use functions that check if a path is absolute since on Unix these start with

```
"/"
```

and on Windows with

```
"[A-Z]:\"
```

History of character encodings: Braille (1824-1837)

Braille uses a cell with 2x3 places for dots to represent each letter. Note that the alphabet is slightly different in different countries; this uses French and Swedish letters.

a ⠁ b ⠃ c ⠉

k ⠅ l ⠇ m ⠍ (the second row of letters; only the last row changes)

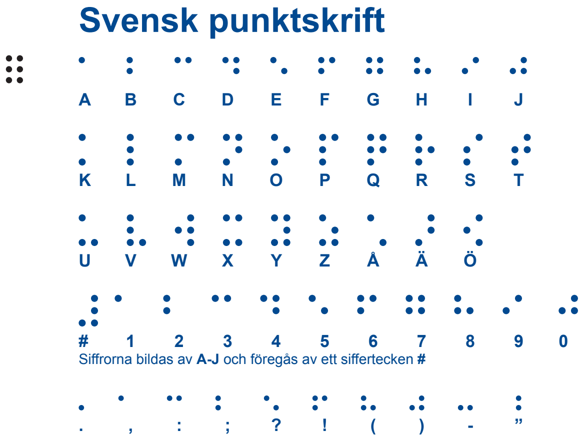
u ⠥ v ⠧ x ⠭ (the third row of letters; the original French did not have w)

A ⠠⠁ (English Braille: 1 cell prefix says the next cell is an upper case letter)

1 ⠁ (same as a)

1 ⠼⠁ (1 cell prefix explicitly says the rest is a number; Braille is ambiguous)

Swedish Braille



Source: Synskadades Riksförbund

History of character encodings: Morse code (1840, 1848, 1865)

	American (Morse)	**Continental (Gerke)**	**International (ITU)**
A	· −	· −	· −
Ä		· − · −	
B	− · · ·	− · · ·	− · · ·
C	· · ·	− · − ·	− · − ·
CH		− − − −	
D	− · ·	− · ·	− · ·
E	·	·	·
F	· − ·	· · − ·	· · − ·
G	− − ·	− − ·	− − ·
H	· · · ·	· · · ·	· · · ·
I	· ·	· ·	· ·
J	− · − ·	· ·	· − − −
K	− · −	− · −	− · −
L	⸺	· − · ·	· − · ·
M	− −	− −	− −
N	− ·	− ·	− ·
O	· ·	· − · · ·	− − −
Ö		− − − ·	

LINKÖPING UNIVERSITY

History of character encodings (pre-computer era)

- Braille (1824-1837). 6-bit binary encoding (suitable for computers), but variable length. 8-bit extensions exist for scientific texts.
- Morse code (1840-1865). Variable length (time of short dot is the time unit; time between two characters is silence the duration of a short dot; the long signal is the duration of three dots).
  - Variable length encodings are not efficient for programs (but are used in data compression schemes and like Morse code more common code points are stored using shorter sequences).
  - Basically uses trinary values (silent, short, long). Computers typically use binary values.
- Baudot code or International Telegraph Alphabet No. 1 (1874; Baud comes from this name). It was a 5-bit binary encoding (encoding only A-Z). ITA2 (1924) used some of the remaining code points to switch character sets (to add digits and punctuation).

History of character encodings (computer era)

- FIELDATA (1956-1962). A 6-bit fixed length binary encoding used in UNIVAC computers (which had a 36-bit word size).
- US-ASCII (1963). A 7-bit fixed length binary encoding still common today.
- Extended ASCII. 8-, 16- and 32-bit computers became more popular and extended ASCII characters sets used the extra available bit for national characters such as åäö in ISO-8859-1 (1985).
- UTF-8 (invented 1992; popular in 2008 or so). Variable length (in units of 8 bits). If the 8th bit is not used in the text, it is identical to US-ASCII.
- There are many more encodings as well. A big mess.

More filesystem: encoding

Why could the following code work on some platforms and fail on others?

```
FILE *fout = fopen("Linköping.txt", "w");
```

Simply put, because different OSes and/or filesystems use different file encodings.
Linux tends to be setup to use UTF-8 whereas Qt uses UTF-16 strings (and translates these strings in its internal file operations).
Windows mixes encodings; you should probably read the documentation to understand the details which is not limited to the following:

- By default, ANSI is used
- If the software chooses, the OEM codepage is used instead
- If sending additional flags, Unicode, UTF-8 or UTF-16LE can be chosen
- If using the wide character functions, UTF-16LE is used
- Also, note that long paths (more than 256 bytes) need to use UNC long paths: `"\\?C:\\..."`

Encodings

```
"Linköping\n"
```

ISO 8859-1 / alias Latin 1 – 1 byte per character (f6 = ö)

```
00000000  4c 69 6e 6b f6 70 69 6e  67 0a                    |Link.ping.|
```

UTF-8 – 1 or more bytes per character (c3 b6 = ö)

```
00000000  4c 69 6e 6b c3 b6 70 69  6e 67 0a                 |Link..ping.|
```

UTF-16 (LE) – 2 or more bytes per character (00 f6 (BE) = ö; note: positions 0 through 255 of UCS-2 and Unicode are the same as in ISO-8859-1)

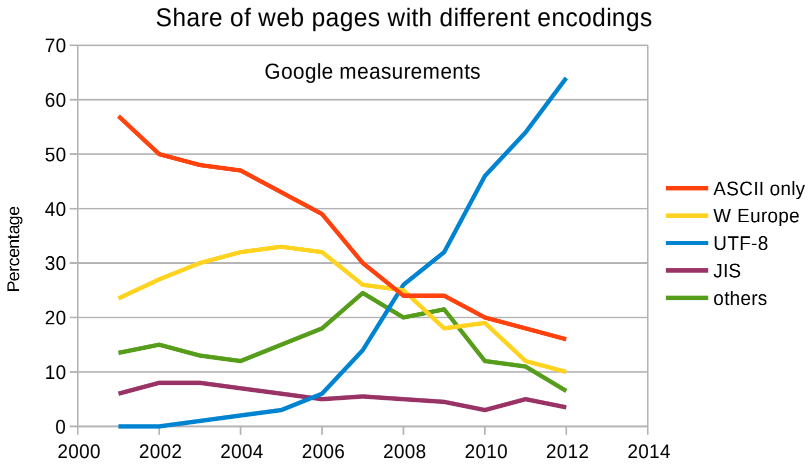
```
00000000  4c 00 69 00 6e 00 6b 00  f6 00 70 00 69 00 6e 00  |L.i.n.k...p.i
00000010  67 00 0a 00                                       |g...|
```

For more basic information on Unicode, see Spolsky 2003.

UTF-8

Code point is often 1 character (such as å, ä, ö or 马), but not necessarily.
For example, å can be stored as c3 a5 (Latin small letter a with ring above) or a (latin small letter a) followed by cc 8a (combining ring above).
Code points are defined in unicode and UTF-8 simply maps byte sequences to unicode code points.
1-byte sequence (7 bits of code points; 87.5%): U+0000...U+007F
`0xxx xxxx`
2-byte sequence (11 bits of code points; 68.8%): U+0080...U+07FF
`110x xxxx 10xx xxxx`
3-byte sequence (16 bits of code points; 66.7%): U+0800...U+FFFF
`1110 xxxx 10xx xxxx 10xx xxxx`
4-byte sequence (21 bits of code points; 65.6%): U+10000...U+10FFFF
`1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx`
Important property: You can synchronize a UTF-8 stream (if highest two bits are 10, it is not the start of a code point).

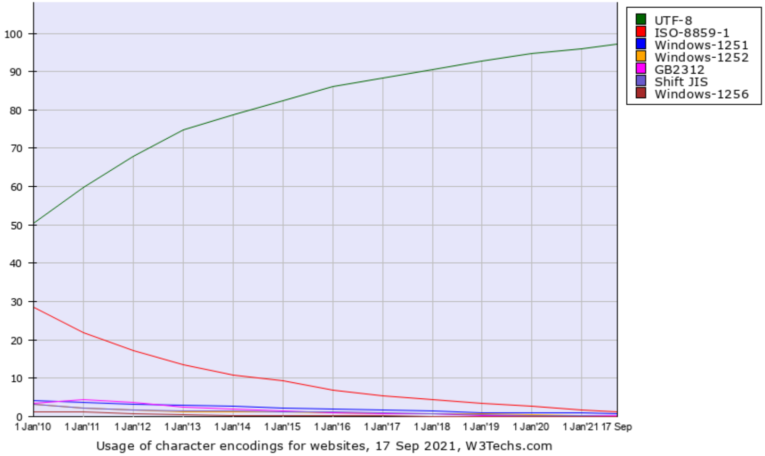
UTF-8 is the encoding of the internet



Source: Chris55, `https://commons.wikimedia.org/wiki/File:Utf8webgrowth.svg`

Highlighted is when Spolsky 2003 was written.

UTF-8 is the encoding of the internet (97.3% of all webpages in September 2021)

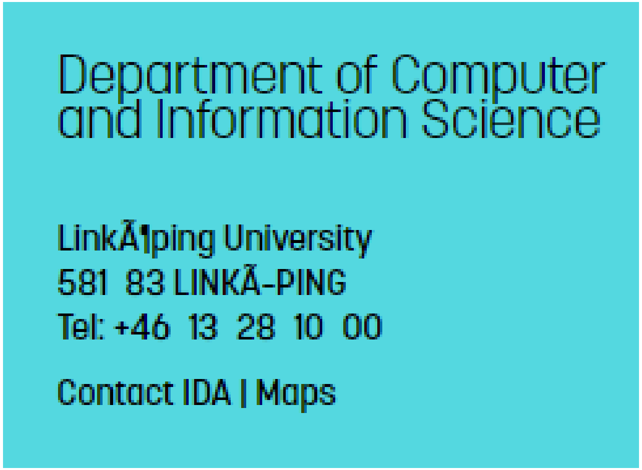


Source: `https://w3techs.com/technologies/history_overview/character_encoding/ms/y`

CJK (Chinese, Japanese, Korean) encodings were a mess before unicode

- Big5: Taiwan, Hong Kong, Macau. Traditional Chinese. A range of values in a 2-byte sequence form the character. If it is outside this range, the character corresponds to some other character set (7-bit or extended ASCII most frequently). The original had around 11000 letters, which did not include names or places. Was still popular with many extensions and variations.
- GB 18030: People's Republic of China. Simplified and Traditional Chinese. Similar to UTF-8 except the mapping to unicode is different and harder to implement. The special mapping makes it backwards compatible with some earlier encodings (GB2312, CP936, GBK 1.0).
- Shift JIS (Shift Japanese Industrial Standards). Single-byte characters are like 7-bit ASCII except $\backslash \rightarrow$ ¥ and ˜ $\rightarrow$ ¯. The 64 half-width kana characters map to single-byte characters as well. The rest is mapped into a two-byte sequence. It is hard to detect when a character starts or ends or if the file is encoded in shift-JIS. (The encoding is still used in Japan; Shift-JIS and EUC-JP account for 10% or so of websites.)
- EUC-KR (used on 17% of Korean websites). Single-byte sequences are 7-bit ASCII. If the first and second bytes are 0xA1-0xFE, it is a Korean character. So there are 93x93 = 8649 Korean characters + 128 US-ASCII characters and the rest (over 7000 characters) are open for extensions (of which there are several).

What happens when you guess the encoding wrong?



Department of Computer and Information Science

LinkÃ¶ping University
581 83 LINKÃ–PING
Tel: +46 13 28 10 00

Contact IDA | Maps

Source: `https://www.ida.liu.se`

Part IV

Localization (L10n) and Internationalization (i18n)

LINKÖPING UNIVERSITY

Internationalization and Localization

Internationalization

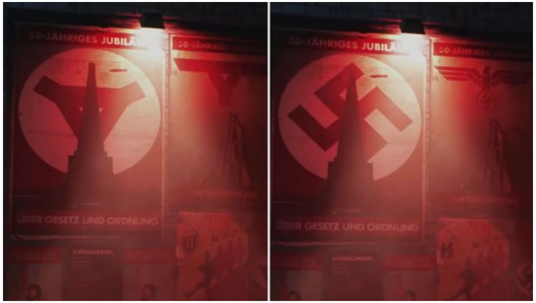
- The process of preparing a program for future localization.
- This is ideally an input at design- or architecture-time.
- Requires guidelines for the programmers.

Localization

- The process of product translation and cultural adaptation for specific countries, regions, dialects, or similar.
- Requires hiring translators and QA staff familiar with the locale.
  - For Open Source projects using gettext, there is `translationproject.org`
- Often requires updated translations also for new versions, adding time to the development cycle.
  - Note that feedback from the localization team may require code changes.

What do we need to localize?

- Numbers, dates, times.
- Text (translations, orientation affects the user interface).
- Cultural differences, conventions, and more.
  - Compare video games released in Germany to other countries (the rules are not as strict since 2018, but still censored).



Wolfenstein Youngblood (Bethesda)

GNU Gettext

Extract strings for translation:

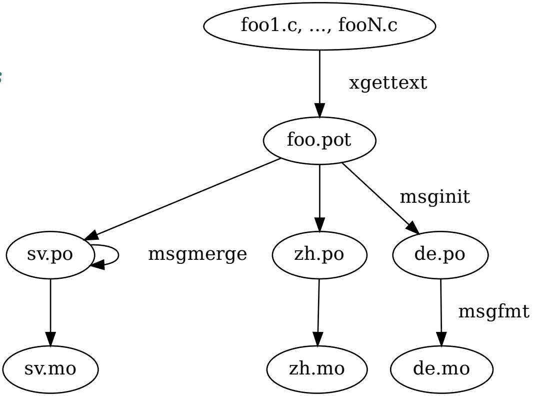
```
gettext("Hello, World!\n");
_("Hello, World!\n"); // Typical alias
```

The pot template file:

```
#: main.c:1
msgid "Hello, world!\n"
msgstr ""
```

The po file with translation filled in:

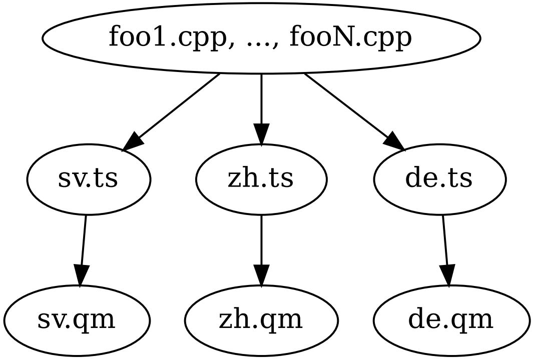
```
#: main.c:1
msgid "Hello, world!\n"
msgstr "Hejsan, världen!\n"
```



Qt Linguist

Similar to gettext, but no analog to pot file.

```
tr("Hello, world!\n");
```



Plural forms

Bad code, bad!

```
tr("%1 item%2 replaced").arg(count).arg(count == 1 ? "" : "s");
```

Better code:

```
QString message;
if (count == 1) {
    message = tr("%1 item replaced").arg(count);
} else {
    message = tr("%1 items replaced").arg(count);
}
```

But some languages have more forms than singular and plural (and others may consider count $==$ 0 to be singular). Polish uses:

- Singular: $n = 1$
- Paucal: $n \bmod 10 = 2 \ldots 4$
- Plural: $n > 0, n \bmod 10 = 0, 1, 5 \ldots 9$

Source: Qt documentation

Plural forms solution

In general, you want multiple strings for translation where languages treat different arguments to the string differently. For counting, there are better solutions:

```
tr("%n item(s) replaced", "", count);
```

French 2 item**s** remplacé**s**

Note: You need an English “translation” to get as good behaviour for English once you start using this pattern in Qt.

The analog function in gettext is called ngettext.

Source: Qt documentation

Where to find some useful data?

`http://cldr.unicode.org` – Unicode Common Locale Data Repository

- What alphabet is used. Do you print text/lines left to right? Top to bottom?
- The names of countries and languages (in each language).
- Date, number format. What calendar is used.
- How sorting works in the locale.
- Names of units. For example Polish:

```
{"temperature-celsius": {
  "displayName": "stopnie Celsjusza",
  "unitPattern-count-one": "{0} stopień Celsjusza",
  "unitPattern-count-few": "{0} stopnie Celsjusza",
  "unitPattern-count-many": "{0} stopni Celsjusza",
  "unitPattern-count-other": "{0} stopnia Celsjusza"
}}
```

- More.

Very useful data, and some of it is provided in standard OS functions.
Quite extensive database (10s of MB of data).

Part V

Conclusions

More things you need to know

For the labs, there are some hints in:

Joel Spolsky. *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)* 2003. URL: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/

For the seminar, also read:

Patrick McKenzie. *Falsehoods Programmers Believe About Names*. 2010. URL: https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/

Andreas Biørn-Hansen et al. "An Empirical Study of Cross-Platform Mobile Development in Industry". In: *Wireless Communications and Mobile Computing* (2019). DOI: https://doi.org/10.1155/2019/5743892

References

Andreas Biørn-Hansen et al. "An Empirical Study of Cross-Platform Mobile Development in Industry". In: *Wireless Communications and Mobile Computing* (2019). DOI: https://doi.org/10.1155/2019/5743892.

IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. ISBN: 9780769551661.

Patrick McKenzie. *Falsehoods Programmers Believe About Names*. 2010. URL: https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/.

References

Joel Spolsky. *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)* 2003. URL: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/.