Programming Style or
How to become a highly paid software engineer

John Morris

School of Industrial Education and Technology, KMITL

previously

Engineering, Mahasarakham University

Electrical and Computer Engineering, The University of Auckland

Iolanthe II **leaves the Hauraki Gulf under full sail – Auckland-Tauranga Race, 2007**

Any High Level Programming Language

GOOD STYLE

Writing computer programs

- **After a few years of experience**
  - **Programming will be easy**
  - **You will get it running correctly**
  - ☺ **2nd or 3rd try**
  - ? **1st try??**
    - **You are not human!!**
    - **I sometimes achieve this … but not too often**
    - **Good engineers**
      - **Check their work**
      - **Generally find improvements**
      - **Especially in documentation**
      - **Just like writing these slides**
        - ☺**You will see many previous, *but now abandoned*, versions**

but

Writing computer programs

but

- **Programming well is more important**
- **Your audience**
  A. **Boss?**
  B. **Workmates?**
  C. **Colleagues?**
  D. **Hackers that you want to impress?**
  E. **Readers of your web page?**

Writing computer programs

but

- **Programming well is more important**
- **Your audience**
  A. **Boss?**
  B. **Workmates?**
  C. **Colleagues?**
  D. **Hackers that you want to impress?**
  E. **Readers of your web page?**
- **Correct answer**
  - **All of them, except maybe D ☹**

Writing computer programs

but

- **Programming well is more important**
  - **A. Boss?**
  - **A good boss or team leader will check your code for**
    - **Readability**
    - **Logic**
    - **Efficiency**
      - **Use of correct algorithms**
  - **B. Workmates?**
  - **C. Colleagues?**
    - **Ones that will need to maintain your code**
      - **☺ When you got promoted or moved for better pay**
    - **Your reputation will follow you**

Writing computer programs

- **Your audience – side comments**

D. **Hackers that you want to impress?**

☺ **Professional software engineers don't need them!**

E. **Readers of your web page?**

😐 **Hmmm**

- **Reliability of web sites is sometimes extremely low**
- **Quality is a problem also**
- **Be careful**
- **You can add lots of valuable information to the web**

but

- **please don't join those who clutter the web with false or misleading information**

☺ **Leave that to politicians!!**

Writing computer programs - Summary

- **You are NOT writing code for yourself**

but

- **For your boss, colleagues, team mates, etc**
- **Who have to understand it**
  - **So they can use it in their parts of the project**
  - **And maintain it**
    - **Estimates vary widely**
    - **Up to 70% of effort in a software project is maintenance**
- **Add yourself after 6 months**
  - **If your software is valuable and works well**
  - **Expect to be asked to extend it**
    - **New functions + extra capabilities**
    - **Ported to a new environment**
      - ☺ **Wow .. it would be nice if I could use that on my phone**

Writing computer programs

- **Documentation**
  - **Program statements in high level languages are inherently**
    - **Cryptic**
    - **Focus on minutiae – individual steps in a program**
  - **Usual strategy**
    - **Add comments to your code, particularly**
      - **Descriptions of**
        - **Structures – purpose**
        - **Functions – purpose and prototype**
  - **Documentation tools**
    - **….. →**

Documentation tools

- Many languages have tools which
  - (semi-automatically) generate documentation
  - C++ – doxygen
  - Python – Sphinx, pdoc, pydoctor, doxygen
  - Rust – rustdoc
- Generally, these will
  - Parse your program
  - Detect functions or C++ classes and methods
  - Parse documentation attached to each element
  - Generate some ‘pretty-printed’ HTML or PDF files

rustdoc

- `cargo` has an interface to `rustdoc`
- In your working directory
  (same one you use for cargo build)

  `cargo doc`

- Look in the

  `target/doc/src/<`*name of your directory*`>`

- You will find an `.html` file
  - Click on it to open your browser

  or

  - Search for it with your browser
- If you don't add anything more,
  you will get a pretty-printed copy of your `main.rs`
  - ☺ Perhaps you can submit the `.html` file for your assignment

rustdoc

Advance notice

- **Documentation is important**
  - **So, when I have studied the idiosyncrasies of rustdoc**
  - **There will be a (hopefully short) lecture on using rustdoc**
  - **Optimistically, rustdoc is simple as doxygen**

  So

  - **you can learn how to add some very pretty documentation quickly**

  AND be able to use it for a following assignment

  - **For now ..**
  - **Try `rustdoc` on your current assignment**

    `cargo doc`

  and

  - **Find your pretty-printed source file**

Back to writing good code

- **Techniques to improve your code**
  - **Most of these apply to ANY high level language**

1. Intelligible names for variables, constants, functions

- ☒ **Short and meaningless names**
  - **Functions name f1, g1, ….**
  - **Names preceded by `my_<xyz>`**
  - **Name unrelated to the purpose, *e.g.***

```
let Fluffy: f64 (name of your cat), etc
```

- **Allowed**
  - **Names derived from mathematical equations**
    - **`x, y, z, x0, x1, ….`**
  - **Indices**
    - **`j, k, m,`**
    - **Note that I exclude `I, l` from my list –**
    - **Easily confused with 1**

Back to writing good code

- **Techniques to improve your code**
  - **Most of these apply to ANY high level language**

1. Intelligible names for variables, constants, functions

☒ **Short and meaningless names .. continued**

- **Even if you are a slow typist**
- **Longer variable names save time eventually**
  - ☺**You need to read your own code**
  - ☺**Cut and paste saves typing mistakes too**

Back to writing good code

2. **Simple (rememberable) naming conventions**
   - **Make your own, but then use them consistently**
   - **Once your program grows, you will find that remembering trivia becomes harder ..**
   - **Rust is extremely annoying in this respect**
     - **`rustc` tries to enforce rules that are NOT part of the formal syntax!**
     - **If the language designers thought such rules useful, they should be part of the formal syntax!!**
       - **Now `rustc` pushes you to use an extra set of rules!!**
     - **Annoyance #1 - warning to re-name a not yet used variable, `k`, as `_k`**
       - ☹ **But I was just checking my syntax before continuing**
       - ☹ **I will use it later!!!**

Back to writing good code

2. **Simple (rememberable) naming conventions**
   - **Make your own, but then use them consistently**
   - **Once your program grows, you will find that remembering**

By the way ..

- **Many of my comments are extremely critical of design strategies**
- **IF you think my comments are not valid,**
- **DO NOT be afraid to put your hand up and raise your voice**
- **My comments are based on personal experience and some biases,**
  ***e.g.* from experiences with other languages, may be evident!**
- **Discussing your reasons (and standing up for them!) will benefit the whole class!!**

not yet used variable, `k`, as `_k`

- ☹ **But I was just checking my syntax before continuing**
- ☹ **I will use it later!!!**

Back to writing good code

2. **Simple (rememberable) naming conventions**
   - **Make your own, but then use them consistently**
   - **Once your program grows, you will find that remembering**

One example of a personal preference

- **Acquired from Java designers may years ago**
- **They starting naming 'getter' and 'setter' methods `set<xxxx>` and `set<xxxx>`**
- **So for my shapes examples, I use names like `setOrigin(Point)` *and* `getOrigin()->Point` consistently, for every new struct**
- **The consistency in my rule is definitely a good thing *but***
- **my choice of '`getXxxx`' and '`setXxxx`' might be less attractive ☺**
- **So …**

he formal

ful,

rules!!

tinuing

Back to writing good code

2. **Naming conventions … more**
   - **Formal syntax for names allows**
     - **Upper case – A, B, C, …**
     - **Lower case – a, b, c, ..**
     - **Arabic numerals**
     - **_**
   - **Mixed in any combination (except no leading numbers)**
   - **Rust pushes you to follow some rules about**
     - **Camel case and**
     - **Snake case**
   - **You can follow them *or***
   - **Make your own**
   - **Your code will be considered 'good' if you are consistent and simple**
   - **Maintainers of your code will (hopefully) follow them easily and**

Back to writing good code

2. **Naming conventions … more**
   - **Rules about camels or snakes or ….**
   - **You can follow them *or***
   - **Make your own**
   - **Your code will be considered ‘good 🏆’**

     if you are consistent and simple
   - **Maintainers of your code will (hopefully) follow them easily and**
   - **Be able to work efficiently with them**

Consistency is the golden rule!!

Back to writing good code

3. **No magic numbers**
   - Literals of any type
     - Numbers, e.g. 0.1, 2.87, …
     - Strings, e.g. “Position”, “Name”, ….
     - Even structures, e.g. Point(0.0,0.0), …
   - Remember software is “Soft”
     - It can be (and will be) easily changed
     - Make sure all these literals are
       - Named (reasonably clear, intelligible name)
         - ☑ `viscosity_cal_factor: f64 = 0.876`
         - ☑ `interest_base_rate: f32 = 5.04`
       - Not
         - ☒ `factor, rate`

In a complex system, there may be several similar constants

Back to writing good code

3. No magic numbers

- In your string decoding exercise,
  I expected to see clearly named constants
  for the key characters, e.g.

```
const UC_zero : u32 = 0x40;
const UC_nine : u32 = 0x49;
```

Many similar names OK ..
Depending on your style or the application

- These constants are placed in a place
  where a maintainer can find them!
- Commonly, put them in a block at the head of your file
- Now code decoding numbers is trivially changed
  - Code itself requires NO changes
  - Just replace the Unicode constants
    - → Now you can decode official Thai documents *or*
      Chinese numbers *or* ….

Back to writing good code

3. **No magic numbers**
   - **Similarly – in decoding your named values**
   - **Separators are likely to change**

```
const separatorA: = ':';
const separatorB: = ',';
```

   - **Here, choosing 'good' names may be trickier**
     - **No obvious purpose for the ':'**

     but

```
const s1: = ':';
const s2: = ',';
```

     - **will NOT help**
     - **Fundamentally, design with the next reader/maintainer/user in mind**

Class exercise

- Now your program needs to decode hexadecimal numbers too
- Usual representations add A, B, … , F

  but their codes are NOT contiguous with codes for Arabic numbers
- Can you devise a simple, easily maintained scheme for handling them?
  - e.g. it should be trivially changed to allow a, b, .., f instead
- 30 minute in-class exercise
- Just sketch out your solution
  - Obviously it needs to fit in a few lines of code
- IF you can define a good solution,
  +3 marks bonus on your final exam mark
  - Not much, but might change a 'B' to an 'A' ☺
- You will be asked to explain your solution to the class
- Hint: arrays or small functions are allowed

  ☹ `match` statements will probably not score much

ไม่เป็นไร
ไม่เป็นไร

Class exercise

- **Note:**
  This is an exercise in thinking HOW to make maintainable code
- **The idea that you propose should be transferable to other similar problems**
- **IT IS NOT an exercise in searching Rust crates for canned solutions**

- **Submit your solution on a (single) piece of paper**
- **Hopefully readable**
  - **team work allows you to nominate a team member with reasonable hand writing**
- **Add the names of all contributors**

- **Bonus exercise only .. If you can't do it** ไม่เป็นไร

  but learn from it, in case a similar question appears in the exam

Back to writing good code

4. **Layout your code logically**
   - **Indentations are the key contributor here**
     - **Loops and blocks help readability**
   - **Blank lines help to group logical blocks together**
   - **Eg a struct and its traits should be adjacent and**
   - **Separated from the next struct**
   - **However,**
   - **DON'T waste lots of space**
     - ☒ **E.g. multiple blanks lines .. One is sufficient**
   - **Usually I want to view blocks of code on a single screen**
   - **Rust's rustdoc generates a pretty-printed output**
     - **I assume it does this by parsing the whole program**
     - **Easy to do with a modern compiler-compiler**
     - **So it probably will indent blocks (loops, if, match, etc) ??**

Writing good code

5. Logical structuring

- Basically raise the abstraction level of your code
  - Build small functions to solve low level problems
  - Use those functions to solve higher level problems
  - At each level, your code becomes more abstract
  - It solves larger problems
  - Moves further away from low level, trivial problems
- Fundamentally ..
  - No 10,000 lines of code programs!!
- Possible guide line ..
  - Each function should fit on a page
    OR
  - A screen full of code

  → Easier handling of code

Again view this as a reasonable target
Vary this advice as appropriate

Writing good code

6. **Testability**
   - Design for testing
   - First strategy
   - Add assertions
     - Briefly mentioned in lectures about formatting
     - Extensively added to Rust documentation
     - Assertions provide formal mathematical statements described required states at any point, e.g.
       - Sums should be non-zero
         - *e.g.* in a triangle

- **Self checking code is so important**
- **Another lecture will cover asserts**
- **A really simple idea, but**
- **Rust management of it is not so simple**
- ***C was much simpler, as you will learn later***

d

) ;

cs with a

Writing good code

6. Design for testing

- Second strategy
- Build functions into your code that test it
- Simple example
- Sort the elements in an array
  - But you need to provide the compare function
  - In Rust, there is are traits:
    `Eq, Ord, PartialEq, Partial Ord`
  - A safe programmer will check that he or she used the trait

```
#[derive(Debug, Eq, Ord, PartialEq, PartialOrd)]
struct Person { name: String, age: u32 }
let mut people = vec![ ……];
…
people.sort()
```

  - Because you need to implement the trait functions!!!

Writing good code

6. Design for testing

- Second strategy
  - Adding test functions

```
#[derive(Debug, Eq, Ord, PartialEq, PartialOrd)]
struct Person { name: String, age: u32 }
…
fn inOrder( vec<Person> ) -> bool { … }

let mut people = vec![ ……];
…
people.sort()
assert!( inOrder( people ), “Sorting error” );
```

- `inOrder` function is only there to check your code!
- If it returns true, everything is OK ☺
- Otherwise, your code needs more work ☹

Final word

- So far,
- Set of key points for writing good maintainable code
  - These are probably the most important ones
  - Also they are relevant for ANY high level language
  - For example:
    - assert's are found in C, C++, Python, Rust, …
    - A page from a Python web-site has the title:

      Python's assert:
      Debug and Test Your Code Like a Pro

- Getting that max$ job
- If you want to impress potential employers
- *Explaining that you know how to write maintainable code*
- *Is a key factor for many employers*
- ☹ Hackers are less welcome