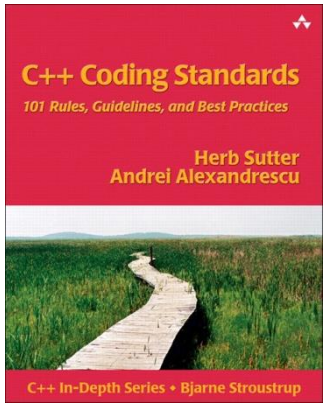
Coding Guidelines

GAM150 – Project



DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

Summmary

1. Why clean code matters
2. Unix Philosophy
3. DigiPen Guidelines
4. Few more tips

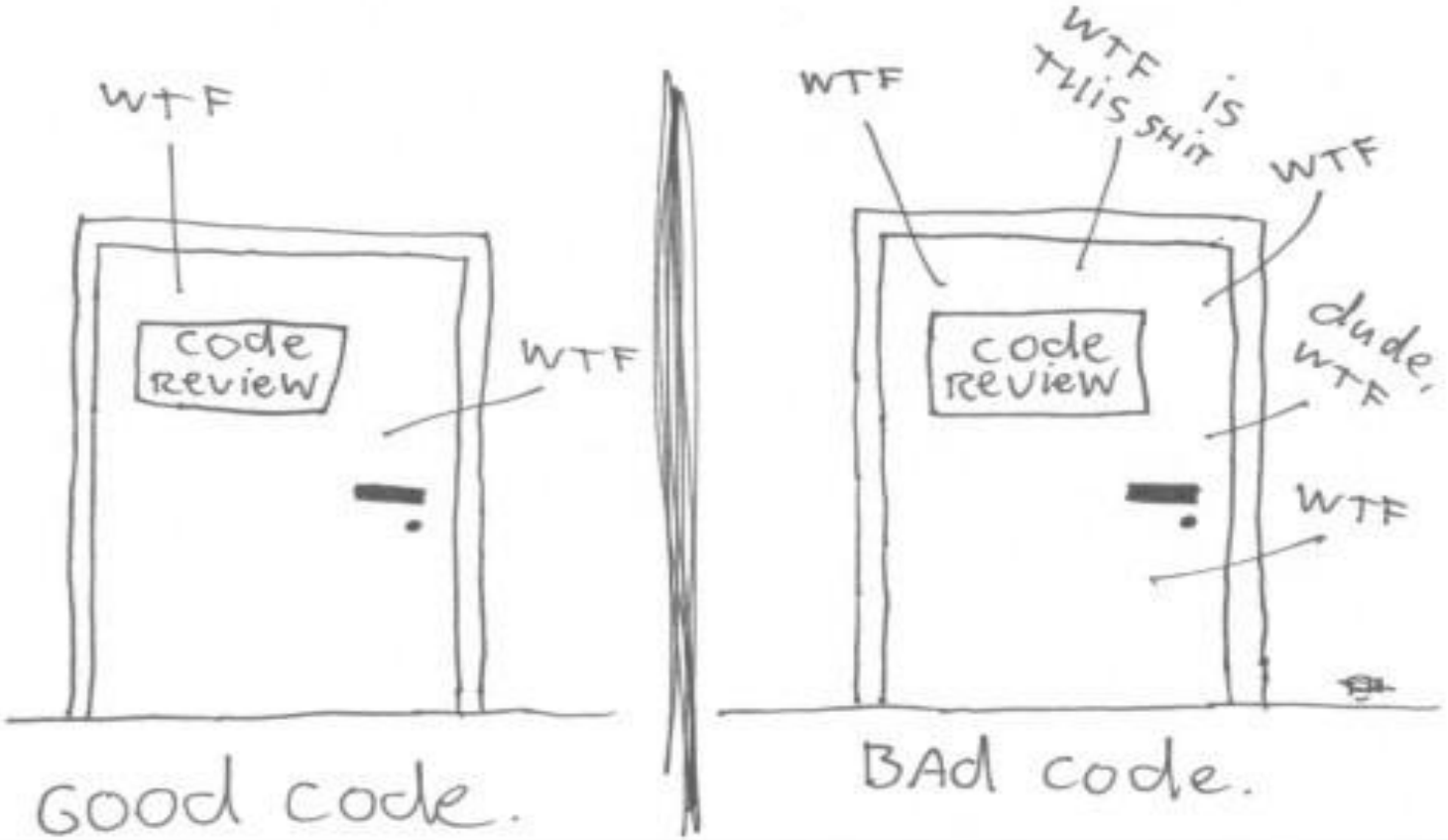
1.

Why clean code Matters

“WTF?”

-Yannick Gerber
(reading code)

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Professional Programmer Responsibilities

- Write Code
  - That **Runs**
  - That Can be easily be **Understood**
  - That Can be easily **Modified**

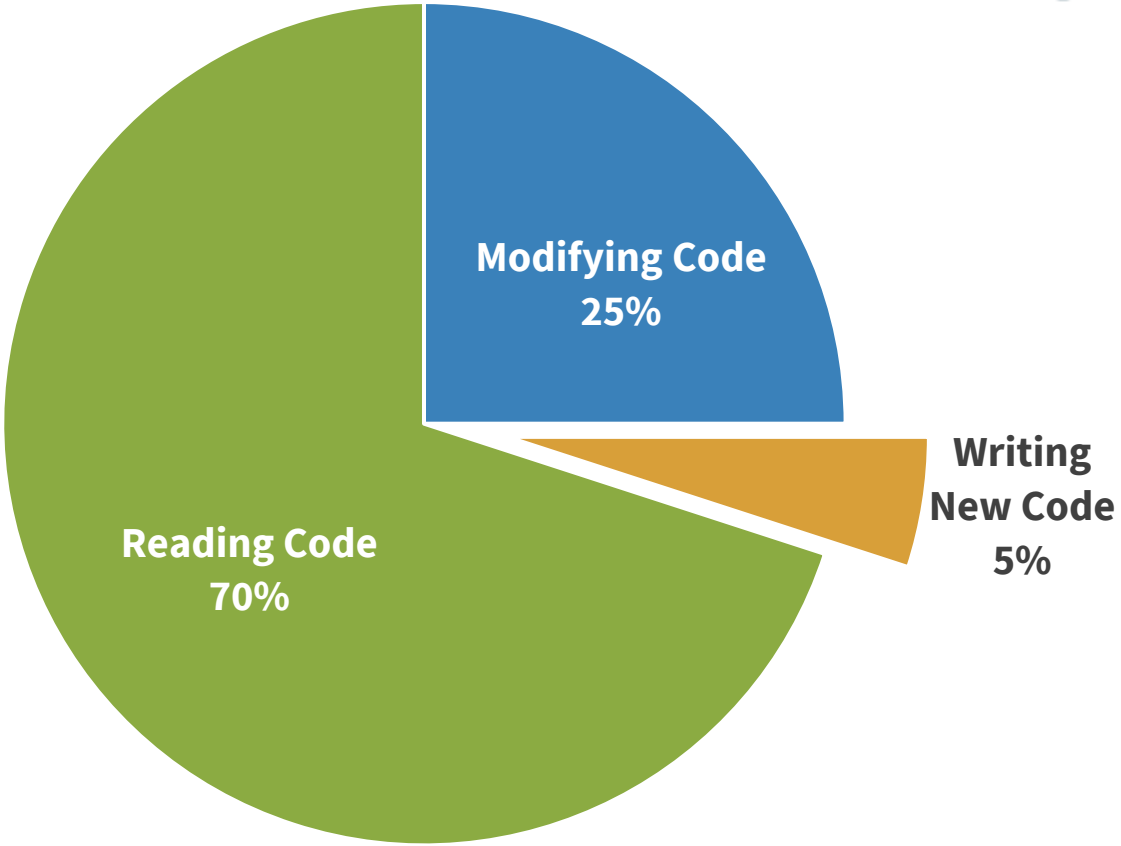
Why ?

Time spent reading code

>

Time spent Writing code

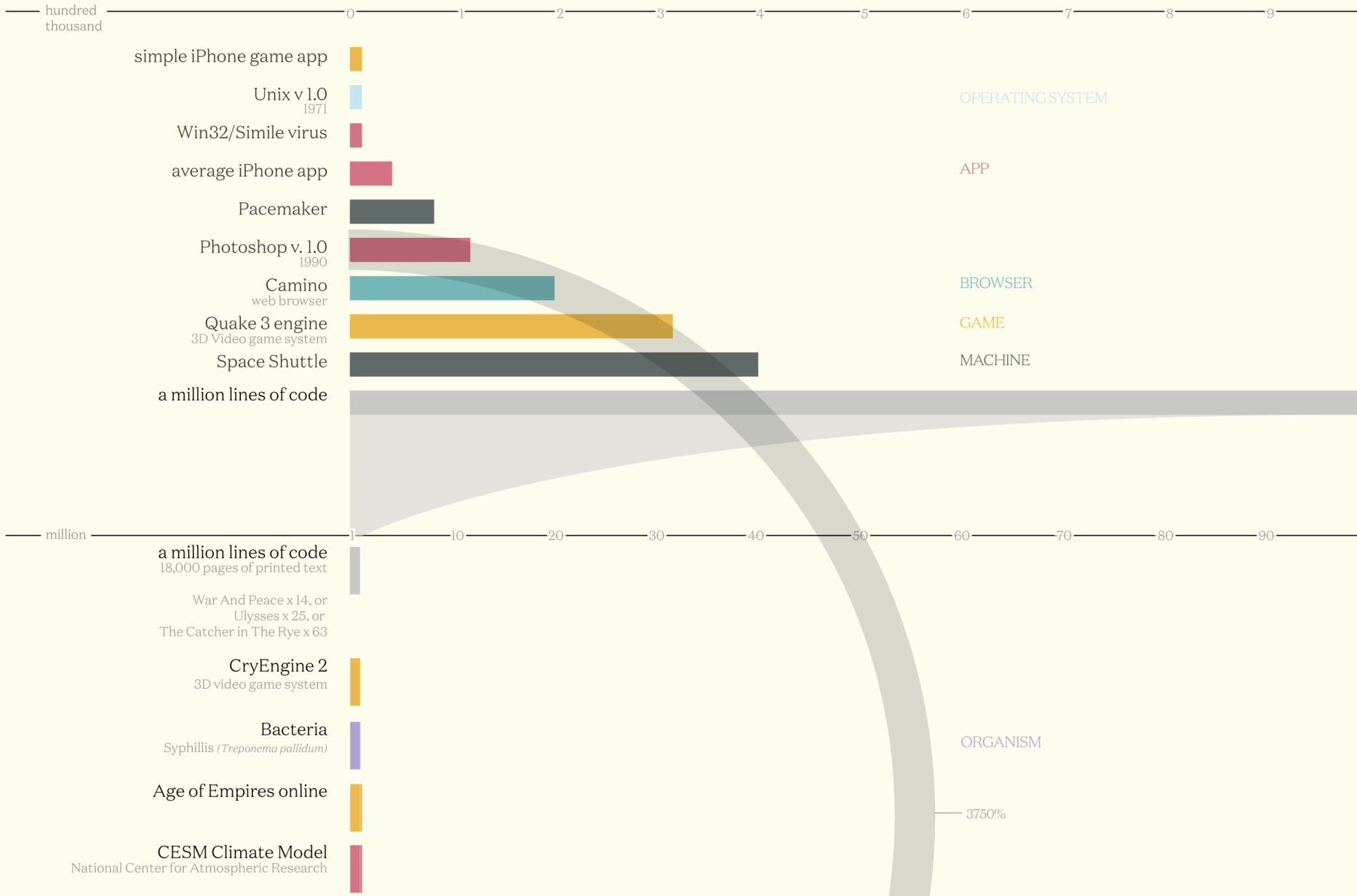
Programmer Activity

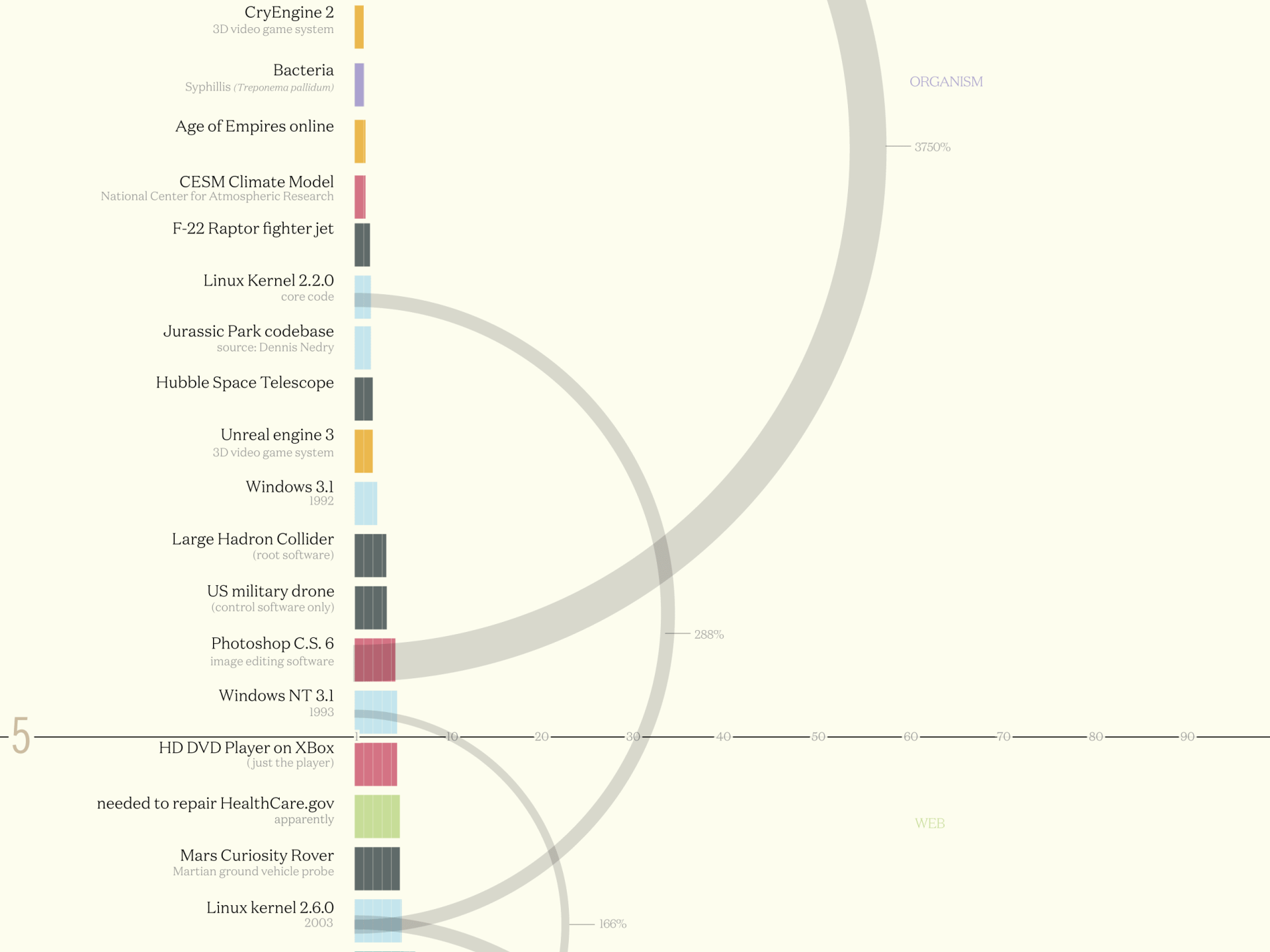


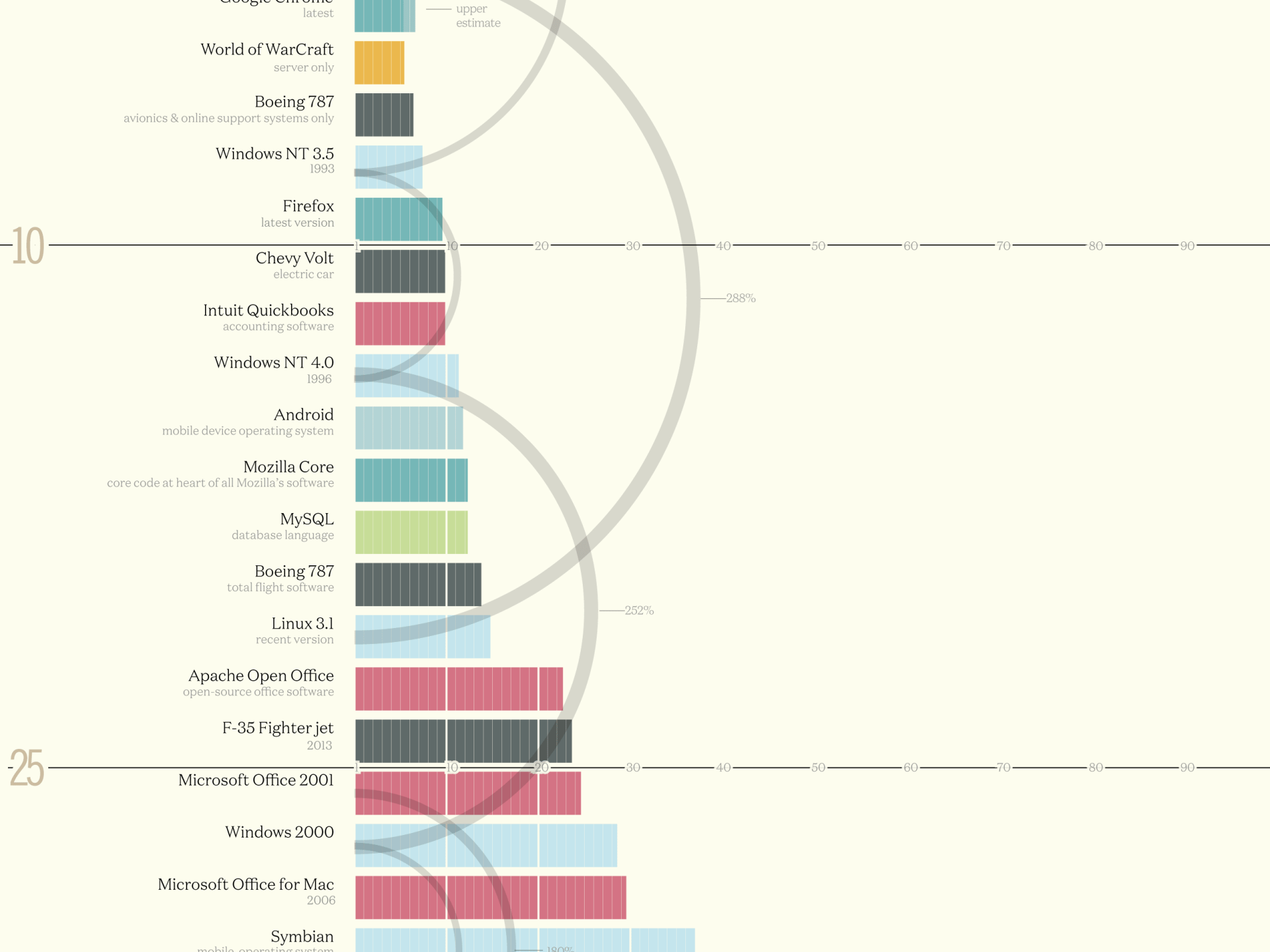
Source : Peter Hallam, C# Compiler Software Design Engineer at Microsoft

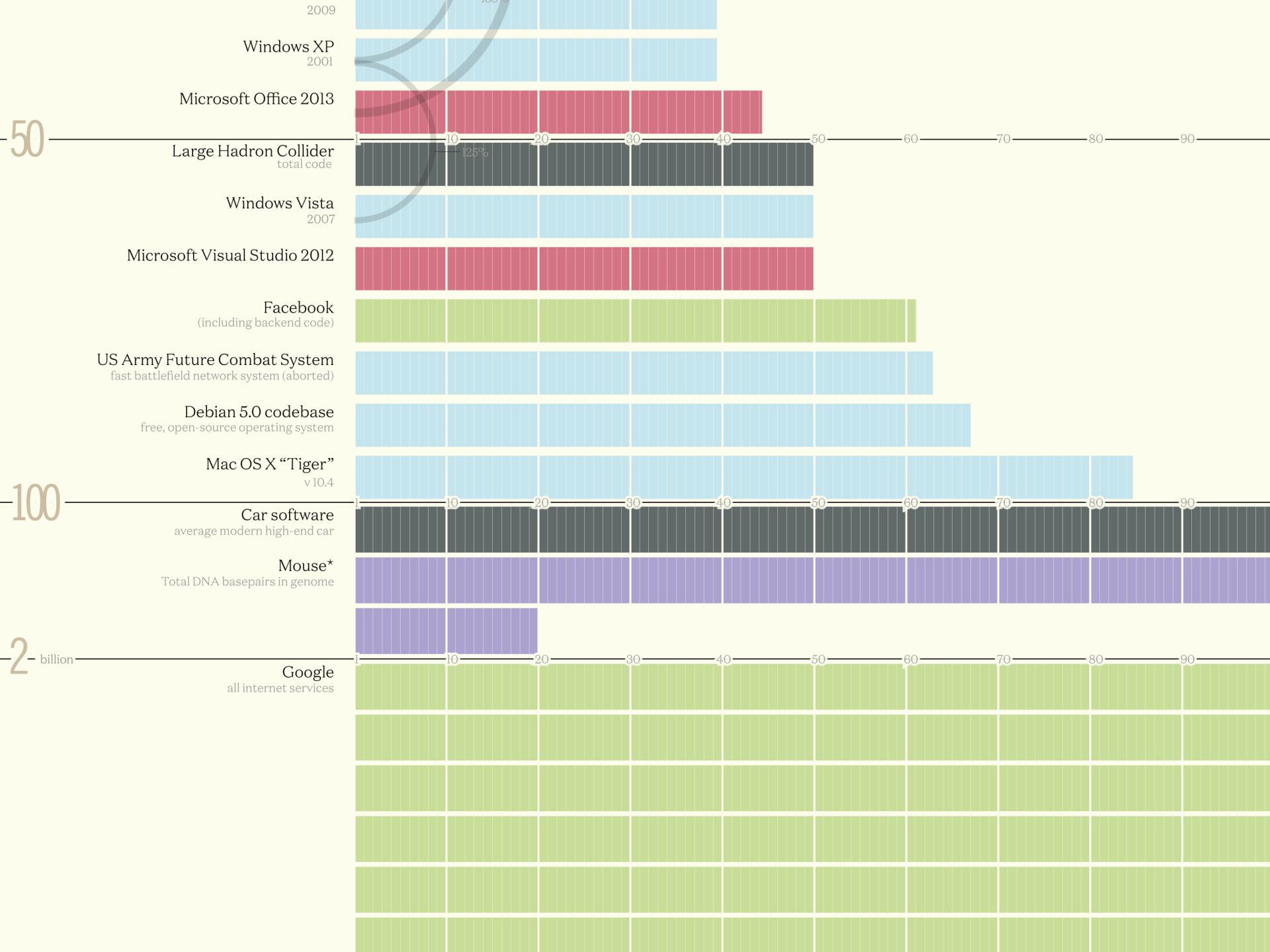
Codebases

Millions of lines of code




CryEngine 2
3D video game system
Bacteria
Syphillis (Treponema pallidum)
Age of Empires online
CESM Climate Model
National Center for Atmospheric Research
F-22 Raptor fighter jet
Linux Kernel 2.2.0
core code
Jurassic Park codebase
source: Dennis Nedry
Hubble Space Telescope
Unreal engine 3
3D video game system
Windows 3.1
1992
Large Hadron Collider
(root software)
US military drone
(control software only)
Photoshop C.S. 6
image editing software
Windows NT 3.1
1993
5
1
10
20
30
40
50
60
70
80
90
HD DVD Player on XBox
(just the player)
needed to repair HealthCare.gov
apparently
Mars Curiosity Rover
Martian ground vehicle probe
Linux kernel 2.6.0
2003
ORGANISM
3750%
288%
WEB
166%


latest
upper estimate
World of WarCraft
server only
Boeing 787
avionics & online support systems only
Windows NT 3.5
1993
Firefox
latest version
10
1
10
20
30
40
50
60
70
80
90
Chevy Volt
electric car
288%
Intuit Quickbooks
accounting software
Windows NT 4.0
1996
Android
mobile device operating system
Mozilla Core
core code at heart of all Mozilla's software
MySQL
database language
Boeing 787
total flight software
252%
Linux 3.1
recent version
Apache Open Office
open-source office software
F-35 Fighter jet
2013
25
1
10
20
30
40
50
60
70
80
90
Microsoft Office 2001
Windows 2000
Microsoft Office for Mac
2006
Symbian


2009
Windows XP
2001
Microsoft Office 2013
50
1
10
20
30
40
50
60
70
80
90
125%
Large Hadron Collider
total code
Windows Vista
2007
Microsoft Visual Studio 2012
Facebook
(including backend code)
US Army Future Combat System
fast battlefield network system (aborted)
Debian 5.0 codebase
free, open-source operating system
Mac OS X “Tiger”
v 10.4
100
Car software
average modern high-end car
Mouse*
Total DNA basepairs in genome
2
billion
Google
all internet services

*Human Genome = 3,300 billion “lines” of code

concept & design: David McCandless
informationisbeautiful.net
research: Pearl Doughty-White, Miriam Quick

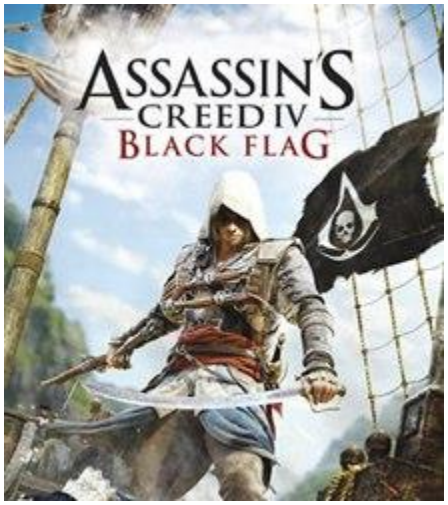
this graphic is a part of
knowledge is beautiful bit.ly/KIB_Books

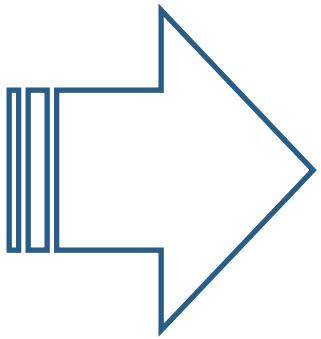
sources NASA, Quora, Ohloh, Wired & press reports
note some guess work, rumours & estimates
data bit.ly/KIB_linescode

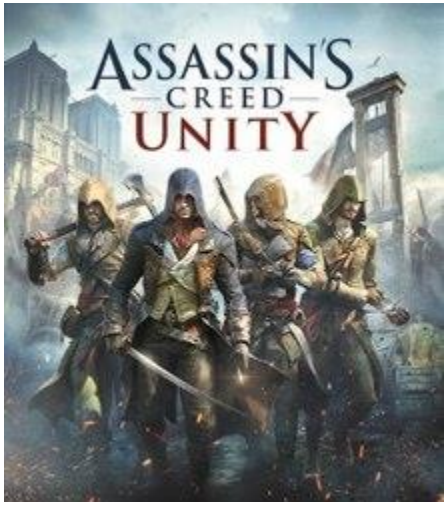
v 0.9// Sep 24th 2015

Code reading time Scales up

- @ Google:
  - Big Code Base
    - 2 Billion lines of code
  - Lots of Programmers
    - 25000 Programmers







Someone: "*Hey, let's remove the contextual crouch and let the player crouch when he wants*"

Source: https://www.engadget.com/2014/10/06/the-crouch-that-changed-assassins-creed/?guccounter=1

“We rewrote like 6 million lines of code for this. It’s not a joke.

Our navigation seems very simple and very accessible, but to do this, it means that the system is calculating, before you’re actually moving, where the possibilities of where you’re actually going to go.”

-Alex Amancio
(Ubisoft)

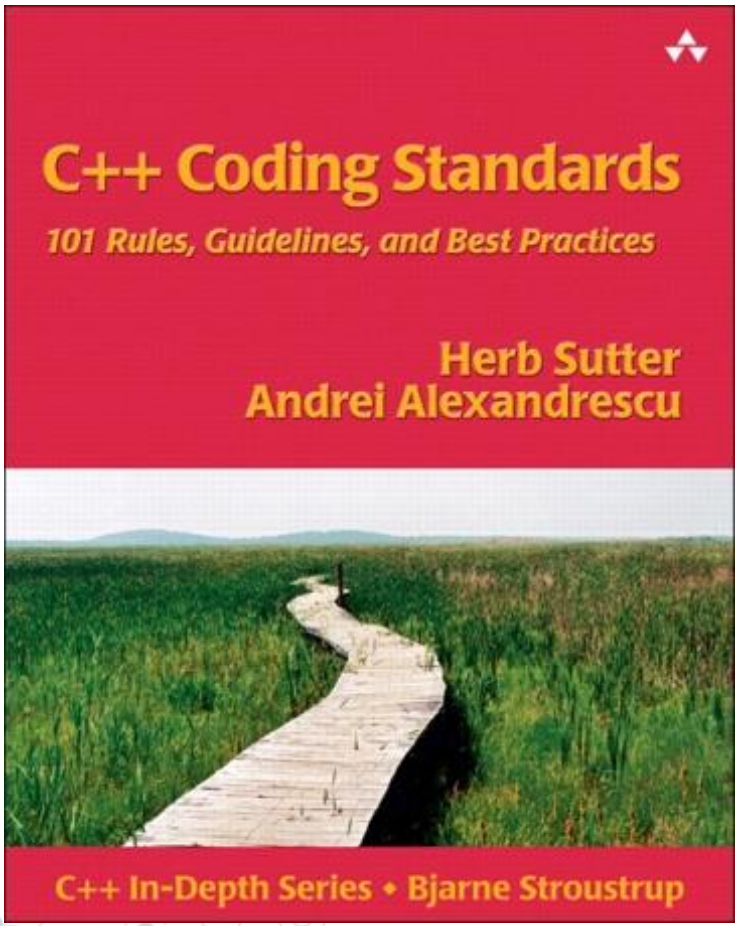
We write good code, so it can be easily read

Reading badly formatted code, or with bad practices, makes us lose a lot of time.

Code guidelines are **actually** used

- Google C++ Coding Guidelines
  - https://google.github.io/styleguide/cppguide.html
- ISO compiled by Bjarne Stroustrup
  - https://github.com/isocpp/CppCoreGuidelines

For DigiPen Students



- Available in the Library
- Read it for the assignment

Best Practices

- What is meant by “Best Practices”?
  - “A method or technique that has consistently shown results superior to those achieved with other means…”

    -- Wikipedia.org
- What is meant by “Best Coding Practices”?
  - “A set of informal rules that the software development community has learned over time to improve the quality of applications and simplify their maintenance.”

    -- Wikipedia.org

Best Practices
=
principles to produce better code

- Easier to understand
- With less bugs

2.

Unix Philosophy

Ken Thompson and Dennis Ritchie

Unix Philosophy

Design rules for modular software development

https://en.wikipedia.org/wiki/Unix_philosophy

The rules

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of Least Surprise: In interface design, always do the least surprising thing.
11. Rule of Silence: When a program has nothing surprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for “one true way”.
17. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

TLDR

Value **simplicity and maintainability**

over **performance and complexity**

KISS

keep.it.simple.stupid.

Rule of Clarity

Clarity is better than cleverness.

Example: ???

```
float  ?????  (float number)
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = *(long *)&y;    // evil floating point bit level hacking
    i = 0x5f3759df - (i >> 1);  // what the fuck?
    y = *(float *)&i;
    y = y * (threehalfs - (x2 * y * y));   // 1st iteration

    // 2nd iteration, this can be removed
    //y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Example: Quake 3 Arena

```
float Q_rsqrt(float number)
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = *(long *)&y;    // evil floating point bit level hacking
    i = 0x5f3759df - (i >> 1);  // what the fuck?
    y = *(float *)&i;
    y = y * (threehalfs - (x2 * y * y));   // 1st iteration

    // 2nd iteration, this can be removed
    //y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Example: Quake 3 Arena

- Fast inverse square root
  - Used to Normalize Vectors

- hardware advancements, (especially the x86 SSE instruction) made it generally obsolete

Rule of Clarity

- Comment your code properly
- Do not use “Magic Formulas”
  - They might not work as you think, with marginal benefits
- Think about your coworker, how they will use and maintain your code

Rule of Simplicity

Design for simplicity.
Add complexity only where you must

Rule of simplicity

- Develop what you need now
  - Avoid Feature creep

- More features

  == more complexity

  == more bugs

  == more code to maintain

  == more code to test

Rule of Transparency

Design for visibility to make inspection and debugging easier.

Rule of Transparency

- Understand what a class does by reading the Declaration
- Understand the execution while you debug
- Requires :
  - Proper naming
  - Proper comments
  - Proper access (private, protected, public)
  - Proper Function breakdown

Rule of Representation

Fold knowledge into data so program logic can be stupid and robust.

Rule of Representation

- Humans are better at visualizing data than reading code flow
- Move complex information into structures
  - Process structures with algorithm
  - Avoid long winded functions with parameters

```
void EnemyAttack(int currentLife, int attackDamage, int attackChance, Player* target) { ... }
```

If the Attack logic change, Update ALL the function calls

VS

```
struct Enemy
{
    int Life;
    int AttackDamage;
    int AttackChance;
    Player* Target;
};


void EnemyAttack(Enemy* enemy) { ... }
```

Can change all the AI locally, without touching the main calls

Rule of Repair

When you must fail, fail noisily and as soon as possible.

Rule of Repair

- Avoid Defensive programming
  - “Ensure the continuing function of a piece of Software under unforeseen circumstances”
  - Code becomes increasingly complex
- Use Assert !
  - Crash during development, fix the flow issues

```cpp
enum DamageType
{
    Physical,
    Poison,
    Fire,
    Cold,

    Unknown
};

int Player::ResistanceToDamage(DamageType damageType, int damage)
{
    switch (damageType)
    {
    case Physical:
    case Poison:
    case Fire:
    case Cold:
        return damage / this.m_Resistances[int(damageType)];

    default:
        return 0; // the player has taken no damage, needs to be handled as neutral in the UI systems ?

    }
}
```

You have to handle the return of 0 everywhere

```cpp
enum DamageType
{
    Physical,
    Poison,
    Fire,
    Cold,

    Unknown
};

int Player::ResistanceToDamage(DamageType damageType, int damage)
{
    Assert(DamageType != Unknown, "DamageType cannot be Unknown");

    switch (damageType)
    {
    case Physical:
    case Poison:
    case Fire:
    case Cold:
        return damage / this.m_Resistances[int(damageType)];
    }

    Assert(0, "Resistance to Damage not Calculated");
}
```

It will Crash in case of wrong DamageType...

Fix the wrong calls, not the consequences of a wrong call

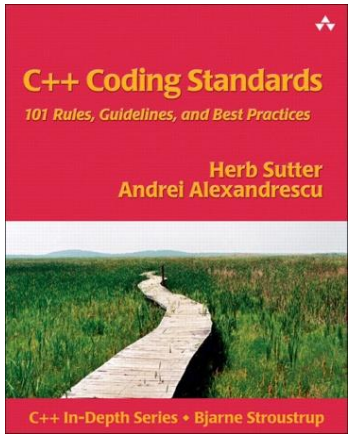
15

Rule of Optimization

Prototype before polishing. Get it working before you optimize it.

Rule of Optimization

- Make your code work fist, optimize later
- Measure Performance. Don’t optimize until you've measured



3.

DigiPen Guidelines

Don’t sweat the small stuff.

Don’t sweat the small stuff

- Know what not to standardize.
- Don’t enforce personal tastes or obsolete practices.
  - “Don’t over legislate naming, but do use a consistent naming convention.”
  - “Don’t prescribe commenting styles, but do write useful comments.”

Compile cleanly at high warning levels

Compile cleanly at high warning levels.

- “Your compiler is your friend.”
  - Warnings are potential problems in your code.
  - Ignoring warnings will allow serious bugs to creep into your code.
  - Consider enabling “Treat warnings as errors” option in VS

Invest in Code reviews

Invest in code reviews

- “More eyes will help make more quality. Show your code and read others’. You’ll all learn and benefit.”
- When done right, the benefits far outweigh the costs
  - Teach / Learn about code blocks
  - Enforce proper usage of systems

Give one entity one cohesive responsibility

Give one entity one cohesive responsibility.

- Entities can be files, classes, structs, variables…
- The smaller your entity, the easier to use.
- One responsibility = easier refinement, update and readiness.

Give one entity one cohesive responsibility.

- Consider simple and short functions.
- How difficult is it to maintain a module (.cpp/.h)?
  - The bigger your files the worst in:
    - Maintenance.
    - Compilation time.
    - Splitting.
    - Reading and navigation.

6 Correctness, simplicity and clarity

Correctness, simplicity, and clarity come first

- Improve readability and maintainability
- Often Clarity > Optimization
  - Modern compilers perform many optimizations to your code
    https://en.wikipedia.org/wiki/Optimizing_compiler
  - Optimize after measuring performance

Minimize Global & Shared data

Minimize Global & Shared data

- Decrease coupling (interdependency)
- Example: animations
  - Should the screen display code know the internals of the animation code?
  - Should the animation code know the internals of the screen display code?
  - Define functions that allow the two systems to operate together.

Hide Information

Hide information

- Don’t give up control of your information.
- Provide an abstract interface that controls access to information.
- Prevents control of data within module
- Potentially improves flexibility

Example: health

- “health” is defined as public
  - (or extern’ed in header)
- Anyone can, and probably will, write:

```
enemy->health -= weapon->damage;
```

- What can happen ?
  - What if health is already zero?
  - What if health becomes negative?

Example: health

- Provide an abstract interface to your data, e.g.:

```
// Set an NPC's health to a specific value.
void NpcSetHealth(int health)
{ ... }

// Modify an NPC's health up/down by a specific amount.
void NpcModifyHealth(int health)
{ ... }
```

- These abstract functions can
  - Prevent out-of-range values
  - Perform error checking
  - Trigger events, if necessary

Example: health

- Consider the following function:

```
bool NpcModifyHealth(int health);
```

- Return value: has the health actually changed ?
  - Useful for triggering visual and/or aural effects

Use const proactively

Example: Gravity

- Can the following value be changed?

```
float Gravity = 9.81f;
```

- What about this one?

```
const float Gravity = 9.81f;
```

Example: Functions parameters

```
void DrawText(char * text)
{ ... }
```

- Does this function modify the “text” data?
- Will the compiler complain if you pass a const value?

```
void DrawText(const char * text)
{ ... }
```

- The reader knows that the contents are not changed.

Avoid Macros

Example: “constants”

```
#define GRAVITY_VALUE 9.81f
```

- Occupies memory anywhere it is referenced

```
const float GravityValue = 9.81f;
```

- Data type is always well defined
- Occupies a single memory location

Avoid Macros

- Difficult to debug
- Hard to read

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

- Dangerous to Write

```
#define square(X) X*X
```

4.

Few more Tips

Dangling Pointers

- Pointer points to invalid memory
  - Deallocated memory
  - Uninitialized pointer (aka wild pointer)
- Crash or silent memory corruption
  - Bugs may appear in unrelated systems
- Solutions
  - Always initialize pointers
  - Always set “freed” pointers to “nullptr”

Memory Leak

- Allocate Memory without releasing it
  - Pointer to a new memory bloc without releasing previous one
- Out of memory crash
- Solutions
  - Minimize allocations / deallocation
  - Bind allocation to the lifetime of an Object (RAII)
    - https://en.cppreference.com/w/cpp/language/raii

Buffer Overflow

- Writing data outside the boundaries of a buffer
  - Data can be overwritten (corrupted)
  - Malicious code can be injected
- Solutions
  - Bounds checking
  - Use “safe” libraries
  - Use buffer overflow protection

Questions?