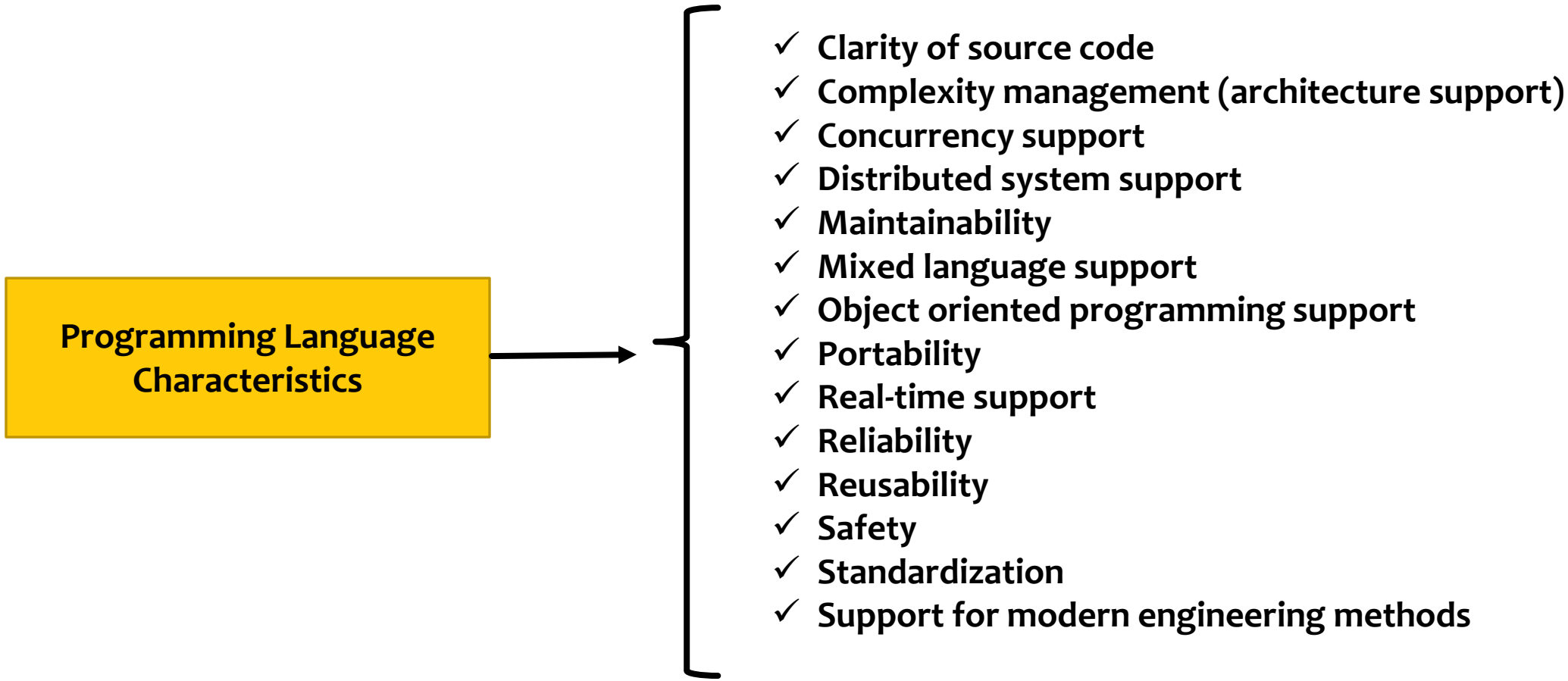
Coding

Yesoda Bhargava

Introduction

- A well-structured code will help in:
  - Reducing the testing and integration time.
  - Maintaining a good documentation.
  - Improving the maintainability of the software product after implementation.
- In order to have a very smooth operation during software maintenance phase, one should not compromise the cost, time, and size of software code.
- The coding phase of a **software development cycle** represents only about 10% to 15% of total software cost.
- The criteria for deciding the quality of a software program is the size (KLOC), memory utilization, readability, execution time, complexity.

Programs written in a particular language will vary widely with respect to these characteristics, depending on the support provided by language, skill, and discipline of the programmers.



Programming language characteristics

- **CLARTIY OF SOURCE CODE: the extent to which inherent language features support source code which is readable and understandable and clearly reflects the underlying logical structure of the program.**
- Most of the life cycle cost of a software system (usually between 60% and 80%) will come during the time after its initial development has been completed.
- This includes efforts to change the software, whether it is to fix the problems or to add new capabilities.
- The most important thing is to write code that others can understand in a clear way. The others are not only other people but also you, who will rewrite the code in several months.


LET US GET SERIOUS
AND CLEAN SOME CODE

Examples

- While naming functions/variables etc. in your code, avoid ambiguity and be specific, to the point.

```
const fetch = async () => {
  return await axios.get('/users')
}
const users = await fetch()
...
```

```
import { fetch } from './utils'

fetch() // fetch ... what?
```

In this code, you can expect what fetch will be getting from the server. But what if the fetch function is exported and used in other files?

Instead, you can name it more specifically.

```
export const fetchUsers = async () => {
  return await axios.get('/users')
}
```

If the variables or functions are in larger scopes in your codebase, use concrete names to make it clear what they're for.

const xxx = validateForm()

In this code, you can understand that validateForm is validating form, but what would you expect to return?

const xxx = isFormValid()

It is then super clear that the method will return true or false.

- Watch out for generic verbs like these:
  - make
  - get
  - set
  - start
  - validate
  - check
  - send

Commenting

- Commenting helps people know as much as possible about code functioning, and commenting can make people understand the code more quickly.
- You need to know the line between a worthless comment and a good one.
- No need to comment if people can easily understand what the code is doing.

```
# Calculate tax based on the income and wealth value and ....
const income = document.getElementById('income').value;
const wealth = document.getElementById('wealth').value;
tax.value = (0.15 * income) + (0.25 * wealth);
#...
```

When to comment

- You need to comment when the code:
  - has flaws, like a performance issue
  - might cause behavior that people wouldn't expect
  - needs to be summarized so that people can easily catch the details
  - needs an explanation for why it had to be written this way

Conditionals

- A lot of if/else statement congest your code and make it unreadable and stuck.
- Reduce the number of conditionals as much as possible.
- Tips to simplify and make your conditionals readable:
  - Handle positive cases first, instead of negative ones.
  - Return early.

Which is more readable for you?

```
if (!debug) {
  // do something
} else {
  debugSomething()
}
```

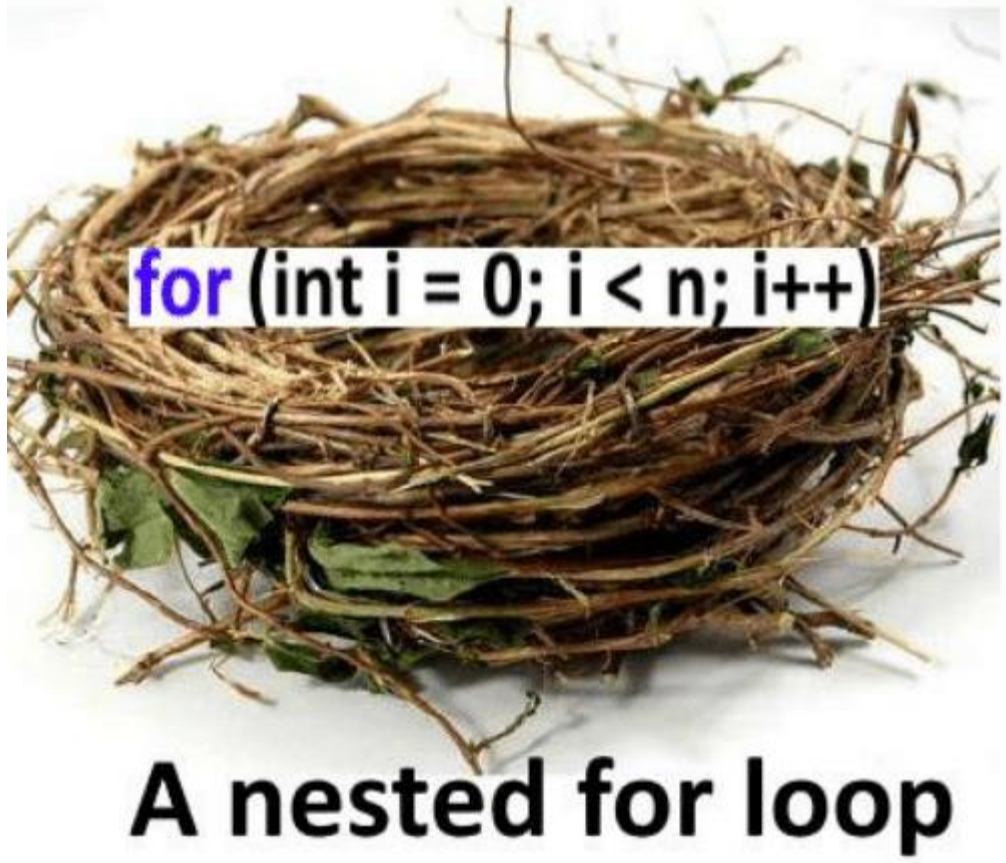
or

```
if (debug) {
  debugSomething()
} else {
  // do something
}
```

But if the negative case is simpler, you can start with the negative conditional check first.

Loops

- Simplifying loops makes your code more readable.
- In real life situations, you will probably run into complex nested loop in objects.
- To avoid complexity for readable code, removing nested loops is essential.
- Nested loops are fine as long as they describe the correct algorithm.



Functions

- Some tips to keep in mind when writing functions:
- **Use a summary name to explain what it is doing.**
  - `const filtered = lists.filter(a => !tmp.has(a.code) && tmp.add(a.code))`
  - What happens when you come across this piece of code above? You stop reading the code and try to figure out what it is doing.
  - `const filtered = uniqueByCode(lists)`
  - *For this you probably expect a function that will remove objects with duplicated code by looking at them once.*
- **Create a function for one purpose.**
- **A smaller function is more readable.**

Create a function for one purpose

```
const updateUser = async (user) => {
  try {
    await axios.post('/users', user)

    await axios.post('/user/profile', user.profile)

    const email = new Email()
    await email.send(user.email, 'User has been updated successfully')

    const logger = new Logger()
    logger.notify()
  } catch (e) {
    console.log(e)
    throw new Error(e)
  }
}
```

Now, you might wonder whether the function Should be, *updateUserAndProfile*, *updateUSerAndProfileAndNotify*, or something Else.

When you are stuck, then split your code into smaller parts, because it is harder for people to understand code when it does multiple things at one time.

A smaller function is more readable and understandable, and later, helps in debugging also.

Another example

- You are analysing data for some insights.
- But you are cleaning the variables of different kinds in the same function.

```
clean_Data=function(var){

  data=subset(data,data$var!=NULL)

  if(check_Quantitative(var))
    data=subset(data,data$var<=99)

  else if(check_Qualitative(var))
    data=subset(data,data$var<=5)

  return(data)
}
```

This is a simple example, there are many levels to data processing some time, so, try to modularize the functionality, and keep your functions similar, readable.

- Larger an expression of code, the harder it will be to understand and to maintain.

```
const generateQuery = (params) => {
  const query = {}

  try {
    if (params.email) {
      const isValid = isValidEmail(params.email)
      if (isValid) {
        query.email = params.email
      }
    }

    const defaultMaxAgeLimit = JSON.parse(localStorage.getItem('defaultMaxAgeLimit') || '')
    if (params.maxAge) {
      if (params.maxAge < 25) {
        query.maxAge = params.maxAge
      }
    } else {
      query.maxAge = defaultMaxAgeLimit
    }

    if (params.limit) {
      query.limit = params.limit
    }

    // do a lot of things here
    // ...

  } catch(err) {
    // error handing
  }

  return query
}
```

This function creates some Query for searching some data. If something is wrong with email query, one will have to go through the function inside, find its implementation, and fix it.

After that, you need to check if the changes will affect other code in the function.

```
const email = (query, params) => {
  if (!params.email) return query

  const isValid = isValidEmail(params.email)
  if (!isValid) throw new Error('Invalid email')

  return { ...query, ...{ email: params.email } }
}

const maxAge = (query, params) => {
  const obj = { maxAge: '' }
  if (!params.maxAge) obj.maxAge = JSON.parse(localStorage.getItem('defaultMaxAgeLimit') || '')
  if (params.maxAge && params.maxAge < 25) obj.maxAge = params.maxAge

  return { ...query, ...obj }
}

const limit = (query, params) => {
  if (!params.limit) return query

  return { ...query, ...{ limit: params.limit } }
}


const generateQuery = (params) => {
  let query = {}

  try {
    query = email(query, params)
    query = maxAge(query, params)
    query = limit(query, params)
  } catch(err) {
    // error handing
  }

  return query
}
```

Breaking down the giant code into small pieces makes the code clearer and more readable.

Each concern is separated from the rest of the code, so you can easily debug and test it.

Read this Goodbye, Clean Code

Complexity management

- The extent to which inherent language features support the management of system complexity, in terms of addressing issues of data, algorithm, interface, and architectural complexity.
- The more complex a system gets, the more important its complexity to be managed.
- Complexity management is always difficult, and it is very helpful if the language can facilitate this goal.

Concurrency support

- The extent to which inherent language features support the construction of code with multiple threads of control (*also known as parallel processing*).
- A concurrent programming language is defined as one which uses the concept of simultaneously executing processes or threads of execution as a means of structuring a program.
- Concurrency is rarely directly supported by a language, and, in fact, the philosophy of some languages is that it should be a separate issue for the OS to deal with.
- However, the language support can make concurrent processing more straightforward and understandable.
- It also provides the programmer with more control over how it is implemented.
- For example, Java and the Java-like Scala language natively support threads and synchronization, whereas languages such as C and C++ rely on external libraries for programming concurrency.
- For detail reading on concurrency, follow this and this link.

Distributed System Support

- The extent to which inherent language features support the construction of code to be distributed across multiple platforms on a network.
- Software components of the system are often distributed across multiple platforms on a network.
- In this networked configuration, each platform performs some portion of the system functions.
- But distribution creates many problems. Multiple platforms are generally heterogenous (different hardware and/or operating systems).
- The problem of distribution can be dealt with by tools rather than language.
- More on distributed systems here.

Maintainability

- The extent to which inherent language features support the construction of code that can be readily modified to satisfy the new requirements or to correct the deficiencies.
- Language choice affects code maintainability.
- The ease or difficulty with which a software system can be modified is known as its *maintainability*. The maintainability of a software system is determined by properties of its source code.
- Maintainability is facilitates by language characteristics, those which make it easier to understand and then change the software.
- Languages that let you write the most concise code have the potential to be most maintainable, because brevity is a big advantage.
- The practical difficulty of writing maintainable code in more expressive languages (eg. Scala, Perl) is no doubt the secret to Java and Go's success. Although its easier to write Scala code than Java, but it is also easier to maintain someone else's Java code than someone else's Scala.

Follow this link to understand the important of maintainability in SE.

Mixed Language Support

- The extent to which inherent language features support interfacing to other languages.
- It means the provision of specific capabilities to interface with other languages.
- Mixed language programming is one way to combine the best facilities of various programming languages and to make use of investments in existing code.
- One might wish to do heavy numerical calculations in Fortran, have a graphical interface to that program written in C++, and have other system functions performed in C. This way the individual tasks of your project are executed in the language best suited for each of them.

Mixed Programming Example

cprogram.c

```c
#include <stdio.h>
int main(void) {
  float a=1.0, b=2.0;
  printf("Before running Fortran function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  ffunction_(&a,&b);
  printf("After running Fortran function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  printf("Before running C++ function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  cppfunction(&a,&b);
  printf("After running C++ function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  printf("Before running C function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  cfunction(&a,&b);
  printf("After running C function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  return 0;
}
```

ffunction.f

```
subroutine ffunction(a,b)
  a=3.0
  b=4.0
end
```

cppfunction.C

```c
extern "C" {
  void cppfunction(float *a, float *b);
}
void cppfunction(float *a, float *b) {
  *a=5.0;
  *b=6.0;
}
```

cfunction1.c

```c
void cfunction(float *a, float *b) {
  *a=7.0;
  *b=8.0;
}
```

Execute

```
gcc -c cprogram.c
gfortran -c ffunction.f
g++ -c cppfunction.C
gcc -c cfunction1.c
gcc -o cprogram cprogram.o ffunction.o cppfunction.o cfunction1.o
```

Output

```
Before running Fortran function:
a=1.000000
b=2.000000
After running Fortran function:
a=3.000000
b=4.000000
Before running C++ function:
a=3.000000
b=4.000000
After running C++ function:
a=5.000000
b=6.000000
Before running C function:
a=5.000000
b=6.000000
After running C function:
a=7.000000
b=8.000000
```

Object-oriented programming support

- The extent to which inherent language features support the construction of object-oriented code.
- This form of programming is associated with the software that has good maintainability characteristics because of the encapsulation of classes and objects.
- It also facilitates the creation of reusable software because it encourages well-structured software with well-defined interfaces, and existing abstractions.

Portability

- The extent to which inherent language features support the transfer of a program from one hardware and/or software platform to another.
- To make software readily portable, it must be written using **non-system-dependent** constructs except where system dependencies are encapsulated.
- The system dependent parts, must be re-accomplished for the new platform, but if those parts of the software are encapsulated, a relatively small amount of new code is required to run the software on the new platform.
- Language standardization has a significant impact on the portability because non-standard language constructs can only ported to systems that support the same non-standard constructs.
- BUT ! A language's portability advantage can be severely compromised by poor programming practices.

Real-time support

- The extent to which inherent language features support the construction of real-time system.
- Language can support real-time systems in two ways:
  - A language can provide specific constructs for specifying the time and space constraints of a system.
  - It can support streamlined ways to express the program instructions.
- Example: Real-time systems often have unique requirements in areas such as device control and interrupt handling, and a language can support managing these in a straightforward, predictable manner.
- Since many real-time systems are concurrent systems, real-time support, and concurrency support are closely related.

Reliability

- The extent to which inherent language features support the construction of components that can be expected to perform their intended functions in a satisfactory manner throughout the expected lifetime of the product.
- Reliability is concerned with making a system failure free.
- Language can provide support for this potential reliability problem through consistency checking of data exchanged.
- Language may also provide support for reliability by supporting explicit mechanisms for dealing with problems that are detected when the system is in operation (exception handling)
- Context is always important.

Reusability

- The extent to which inherent language features support the adaptation of code for use in another application.
- It is very common to reuse common data structures such as stacks, queues, and trees.
- When reusing larger portion of code, the biggest issue for reusability is whether the interfaces defined for the code to be reused are compatible with the interfaces defined for the system being created.
- Reusing at any level can be facilitated by language features that make it easy to write independent modules with well-defined interfaces.
- Object-oriented languages provide class (or class-like) constructs for encapsulating sets of definitions that are easily adapted for new program, promoting reusability.

- For more, read this article.


I CAN'T REUSE YOUR COMPONENT,
IT HAS A LOT OF DEPENDENCIES
imgflip.com

Safety

- The extent to which inherent language features support the construction of safety-critical system, yielding systems that are fault tolerant, fail-safe, or robust in the face of system failure.
- Safety is related to reliability.
- A system must always do what is expected from it and be able to recover from any situation that might lead to a mishap or actual system hazard.
- Thus, safety tries to ensure that any failures that occur are minor consequences, and even potentially dangerous failures are handled in a fail-safe fashion.
- Language can facilitate this through such features as a rigorous computational model, built-in consistency checking, and exception handling.

Standardization

- The extent to which inherent language definition has been formally standardized by recognized bodies such as ANSI and ISO, and the extent to which it can be reasonably expected that this standard will be followed in a language translator.
- Most popular languages are standardized through ANSI and ISO.

Support for modern engineering methods

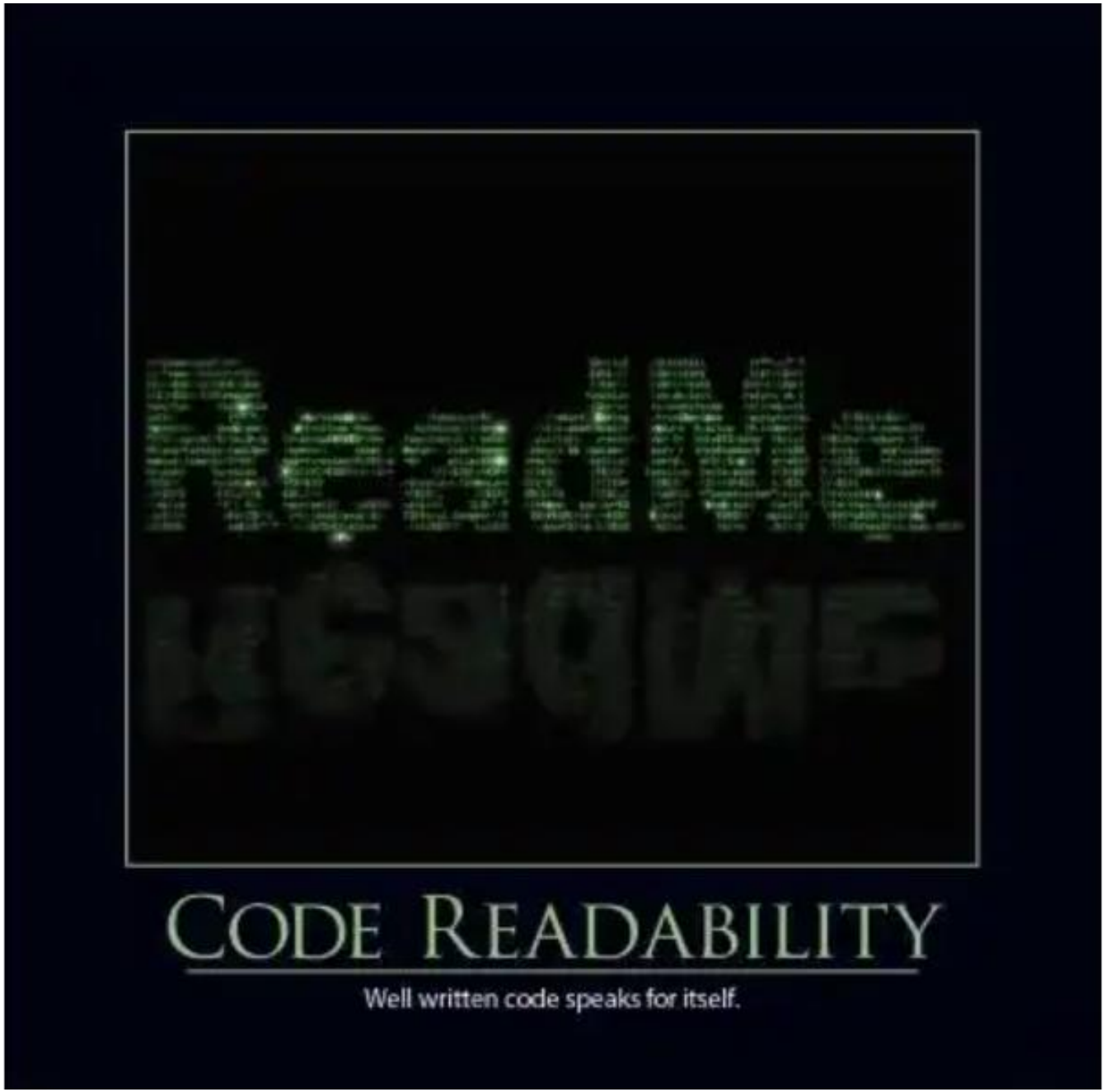
- The extent to which inherent language features support the expression of source code that enforces good software engineering principles.
- Support for modern software engineering methods is to encourage the use of good engineering practices and discourages poor practices.
- Hence, support for code clarity, encapsulation, and all forms of consistency checking are language features that provide this support.

CODING QUALITY

- A poorly written program can have very bad effect at the time of testing and maintenance.
- A good program has certain measurable attributes.
- These attributes can be specified in the **Software Requirements Specification** (SRS) and tested during **test** and **integration** phases.
- These attributes can become quality assurance parameters for the program. These include:
  - Readability, Understandability, and Comprehensibility
  - Logical Structure
  - Physical layout
  - Robustness
  - Memory and execution efficiency
  - Complexity
  - Human factors
  - Reusable code

Readability, Understandability, and Comprehensibility (RUC)

- The most important characteristics of a program are smooth readability, following the logic and its structure.
- RUC helps in debugging and maintenance of a program.
- Coding is a social activity. Your code does not exist in a vacuum, just implementing a lone task.
- The code you write will be re-read by many other developers, who want to either understand or modify how your code works.
- So, writing a readable code is a responsibility.
- Tips to maximize code readability:
  - Indentations
  - Comments and Documentation, but remove unnecessary comments.
  - Follow proper naming schemes , i.e. good classes, good functions, and good objects.
  - Avoid deep nesting
  - Code grouping
  - Limit line length
- The real test of readable code is others reading it. So get feedback from others, via code reviews. May be organize a COMPETITION?


REGEX
MESSAGE
CODE READABILITY
Well written code speaks for itself.

Certain tasks require a few lines of code. It is a good idea to keep these tasks within separate blocks of code, with some spaces between them.

Code Grouping
Note the comment at the beginning

```php
// get list of forums
$forums = array();
$r = mysql_query("SELECT id, name, description FROM forums");

while ($d = mysql_fetch_assoc($r)){
$forums[] = $d;
}

// load the templates
load_template('header');
load_template('forum_list', $forums);
load_template('footer');
```

DRY Principle

- **Don’t Repeat Yourself.**
- ***"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.“***
- The same piece of code should not be repeated over and over again.
- For example, most web applications consist of many pages.
- It's highly likely that these pages will contain common elements. Headers and footers are usually best candidates for this.
- It's not a good idea to keep copy-pasting these headers and footers into every page.

from PAGE C1

Both said the fitness program still hasn't caught on with the entire school.

"Some people are just lazy," Powers said.

The red and blue outdoor equipment installed by Intel volunteers this August was specially designed to help children work on the areas in which they are most lacking. The Sierra Vista playground now includes a pole climb, a vault bar, horizontal ladder (monkey bars), sit-up station, step test station, parallel bars and pull-up bars.

Parents volunteer before school hours and during recess to monitor the equipment and to help chart the students' improvement.

The equipment also is available to the community after school hours, Argano said.

The curriculum is a supplement to the school's PE

strength, flexibility and cardiovascular areas.

Sierra Vista fourth- and fifth-graders also are competing as a group over the next few months for $1,000 for their school as part of the "Cardinal Health Challenge."

"Our function is to reverse the lack of fitness in youth and teach children to take personal responsibility for their health," according to the Web site for Project Fit America, a nonprofit charity that has helped get equipment donated to more than 400 schools in 39 states.

In a study of 6,000 children over the past eight years, 22 percent of fourth- and fifth-graders had excessive cholesterol levels, 32 percent were obese, 37 percent were in poor physical condition, and 40 percent of American children ages 8-12 show one or more risk factors of heart disease, including high blood

The charity blames a lack of funding from state and federal governments for physical education over the last four decades, and with schools averaging only one PE teacher for every 700 students.

Other contributing factors to poor health include a lack of unsupervised play because of fear of abductions and unsafe neighborhoods; increased availability of electronic entertainment, such as television, computers and electronic games; lack of funding for PE programs; and aggressive advertising of junk food.

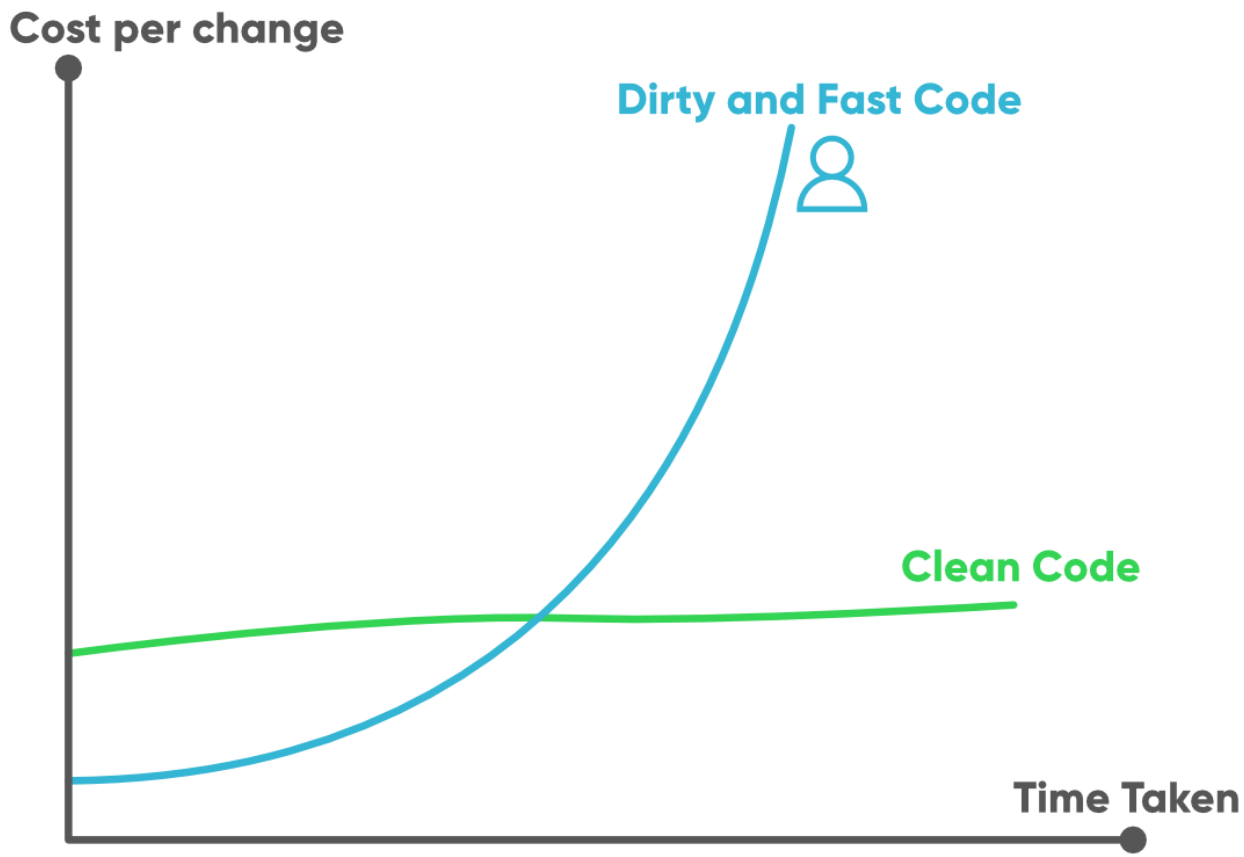
At Sierra Vista, PE comes twice a week for fewer than 30 minutes. The class itself has increasingly become more fitness-oriented and less focused on competition as teachers try to help students identify physical activities they will enjoy

Ever wondered why newspapers are like this?

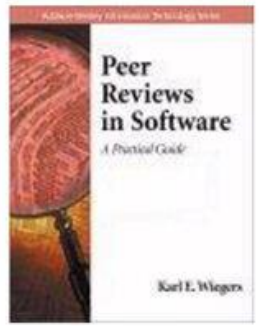
Our eyes are more comfortable when reading tall and narrow columns of text.

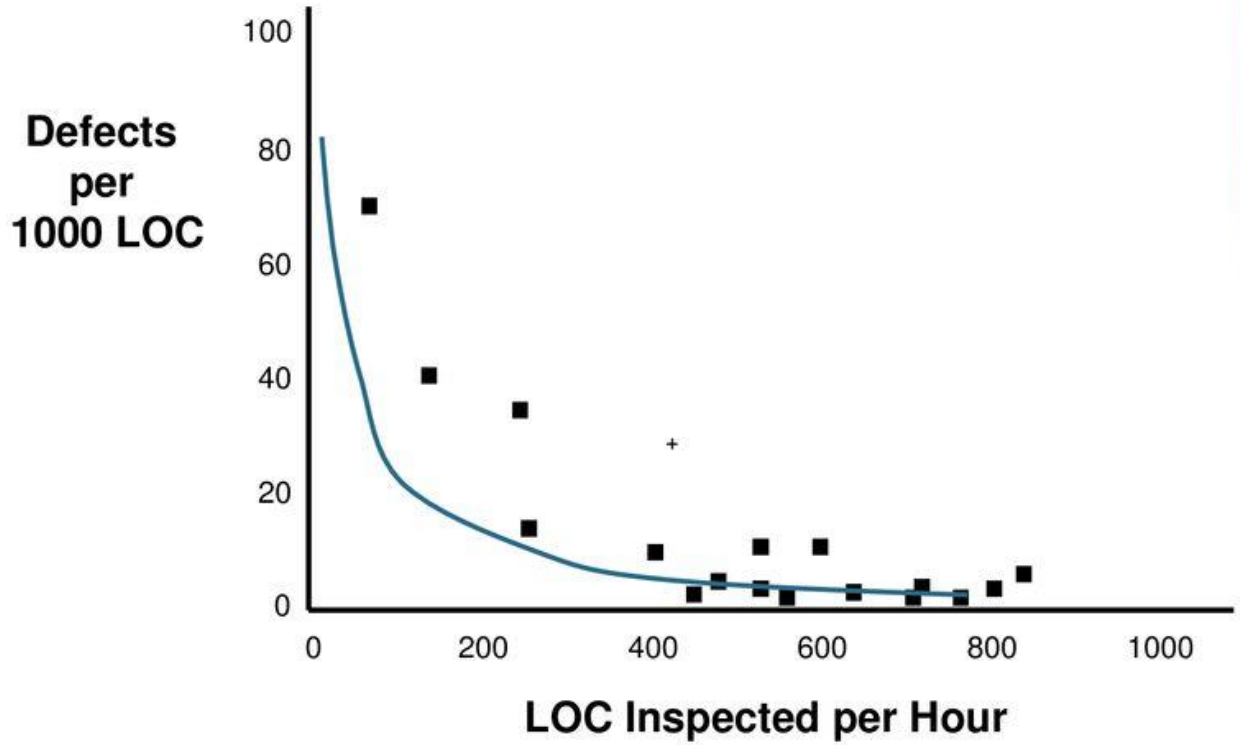
Avoid writing horizontally lines of code

```php
#bad
$my_email->set_from('test@email.com')->add_to('programming@gmail.com')->set_subject('Methods Chained')->set_body('Some long message')->send();
# good
$my_email
->set_from('test@email.com')
->add_to('programming@gmail.com')
->set_subject('Methods Chained')
->set_body('Some long message')
->send();
#bad
$query= "SELECT id, username, first_name, last_name, status FROM users LEFT JOIN user_posts USING(users.id, user_posts.user_id) WHERE post_id = '123'";
# good
$query= "SELECT id, username, first_name, last_name, status
  FROM users
  LEFT JOIN user_posts
  USING(users.id, user_posts.user_id)
  WHERE post_id = '123'";
```


Cost per change
Dirty and Fast Code
Clean Code
Time Taken

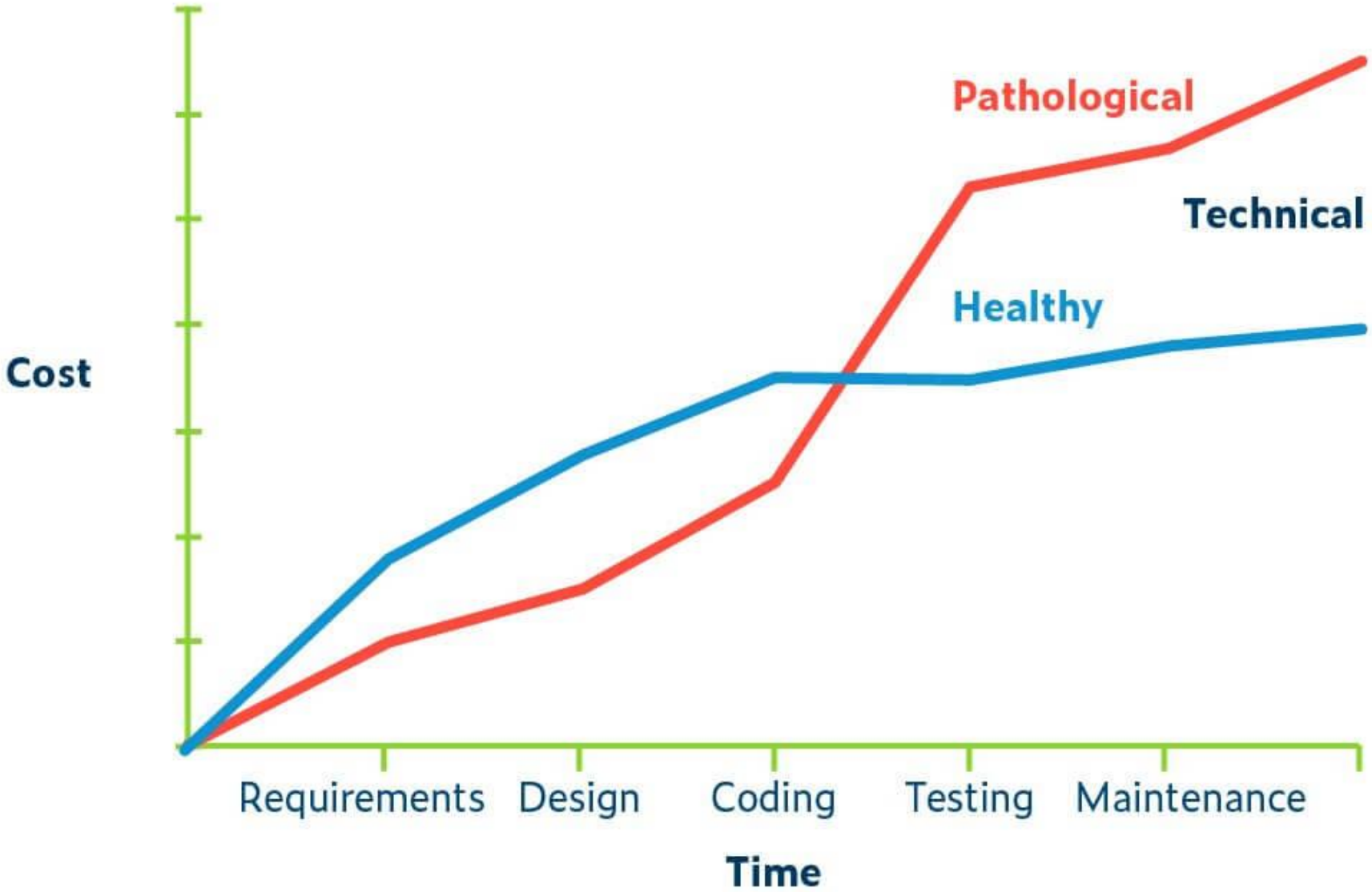
Relationship between code inspection rate and defect density- Effectiveness



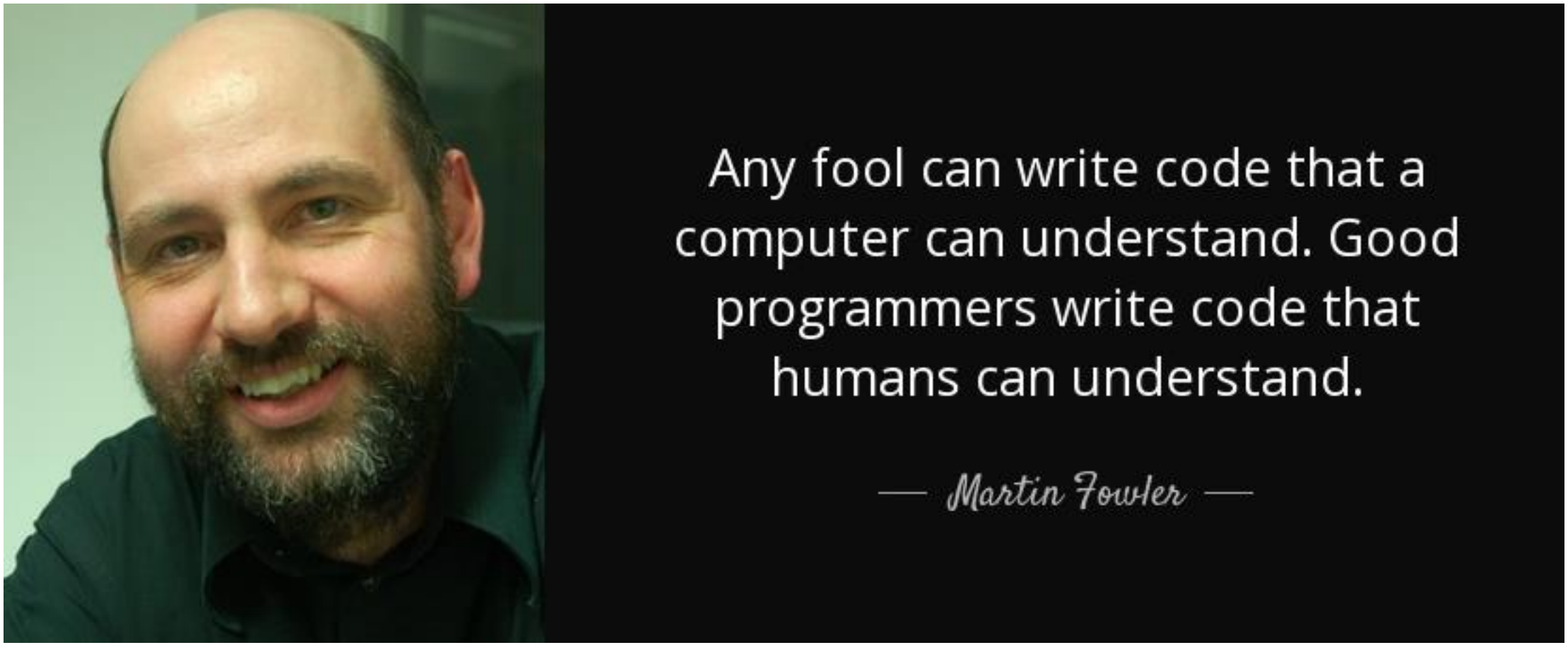


LOC= Lines of Code

Source: Wiegers, Karl, 'Software Technical Reviews: A Practical Guide', p 76, 2001.



Poor quality is cheaper until the end of the coding phase. After that, high quality is cheaper.



Article recommendation

Logical Structure and Physical Layout

- Application of structured programming rules help to create a logically and structurally sound coding.
- A logically designed program will become stronger and long lasting.
- Actual listing of course code demands good use of indentation, separation of key words from identifiers, use of meaningful identifiers, use of comments, proper beginning and end of each block.
- Unstructured code is generally hard to understand, test, and review.
- But even structured code can be problematic if it is too complex.

Robustness

- How well a program can withstand at the time of handling incorrect input data.
- A software product must be protected against the misuse and be designed to deal with bad input data.
- On encountering a bad input data, the execution should not stop and on the other hand sufficient warning should be given on validation or self-correcting capability should be provided.
- This attribute should be specified in user manual, operation manual, and SRS.
- Example : LASCAD failure in less than perfect location details.

There is a good example for this

Very non-robust:

```
(status, value) = get_from_url(...)
do_something_with_value(value)
```

Slightly better:

```
(status, value) = get_from_url(...)
if (is_ok_status(status))
  do_something_with_value(value)
else
  show_error(...)
  exit()
```

Even better:

```
(status, value) = get_from_url(...)
if (is_ok_status(status) and is_valid_value(value))
  do_something_with_value(value)
else
  show_error(...)
  exit()
```

Better still:

```
(status, value) = get_from_url(...)
if (not (is_ok_status(status) and is_valid_value(value)))
  (status, value) = get_from_local_file(...)
if (not (is_ok_status(status) and is_valid_value(value)))
  value = SAFE_DEFAULT_VALUE

do_something_with_value(value)
exit()
```

Robustness Principle

- The general idea is that if you build your software to be very strictly correct in what it does and what it sends to other software, but very flexible and generous in what it will try to accept as input, then it is robust software and contributes to a **robust system: a system that is unlikely to fail, even when errors and faults arise.**

Robustness Testing

Most people do not worry about break failing when they get behind the wheels.

Or about injuries from a toy malfunction.

Why?

- Because the software developer or the quality control engineer has addressed the quality issues.
- If the goal is to deliver high-quality, reliable software systems, then tests must be conducted.
- Quality issues won't go away without testing because the more reliable the product is, the safer it is.
- One of the processes we use for testing quality and reliability is called robustness testing, the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions.
- Think of airline testing, LASCAD.

Memory and Execution efficiency

- Refers to how fast the program works and how much computer memory is used.
- A program must use fast and efficient algorithms.
- Appropriate file, data structures, and access methods should be used.
- Since the memory of a computer is very expensive, care must be taken to minimize the use of memory without sacrificing the execution efficiency.
- Think of LASCAD fiasco.

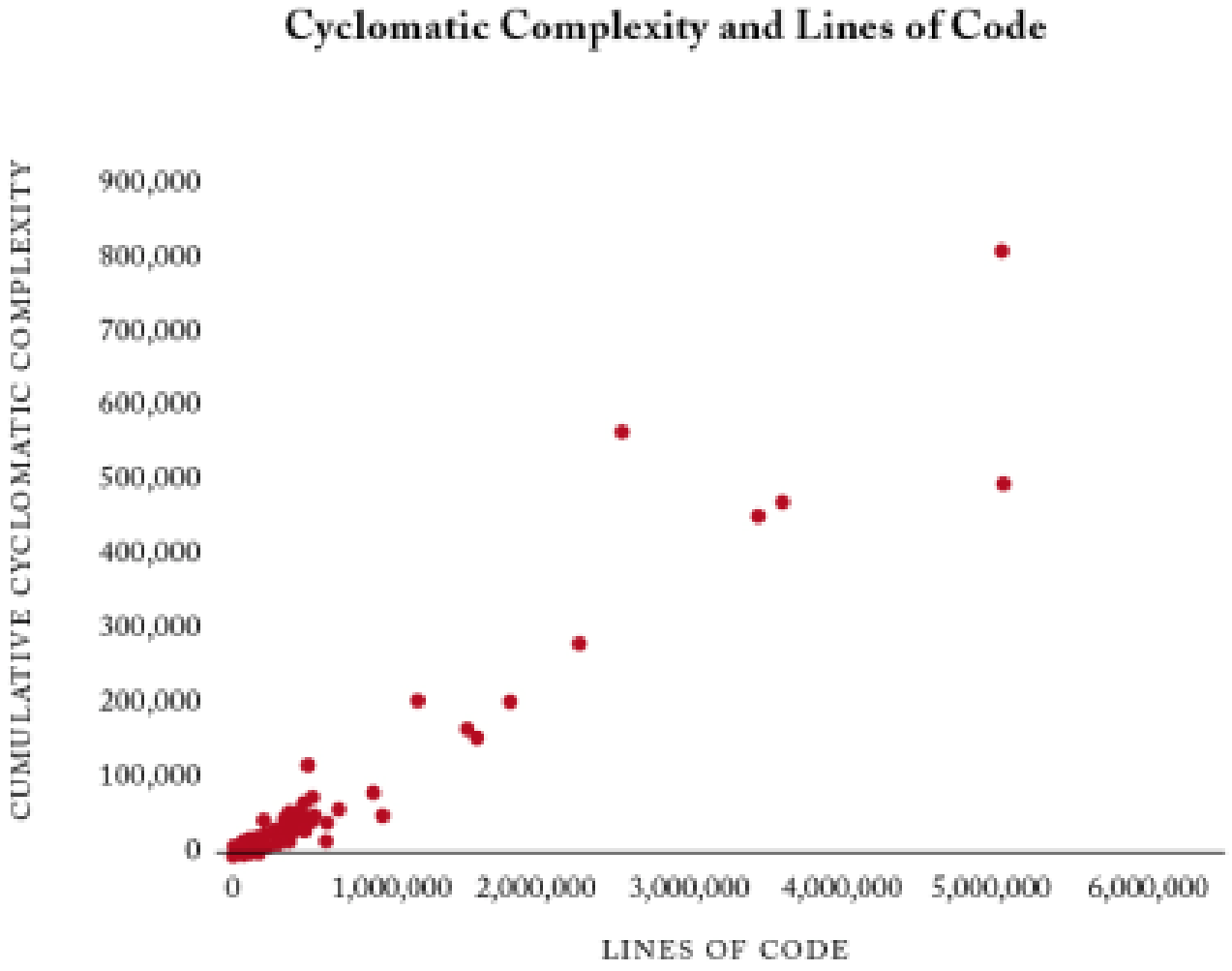
Complexity

- Both algorithmic complexity or structural complexity will make the program difficult to understand and implement.
- Less complexity with less branching of program needs to be used for a good coding.

"Complexity has and will maintain a strong fascination for many people. It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. **Simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time which we too often believe to be unaffordable** ." -Niklaus Wirth

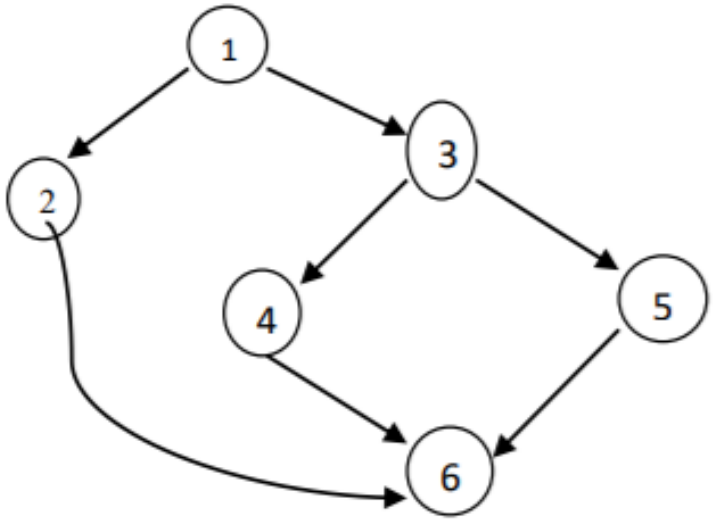
Cyclomatic Complexity V(G)

- A software metric used to measure the complexity of the software.
- Informally described as : Number of decision making points + 1
- Or number of linearly independent paths through the code.
- Eg. If the code has one *if* statement, number of linearly independent paths are… ??



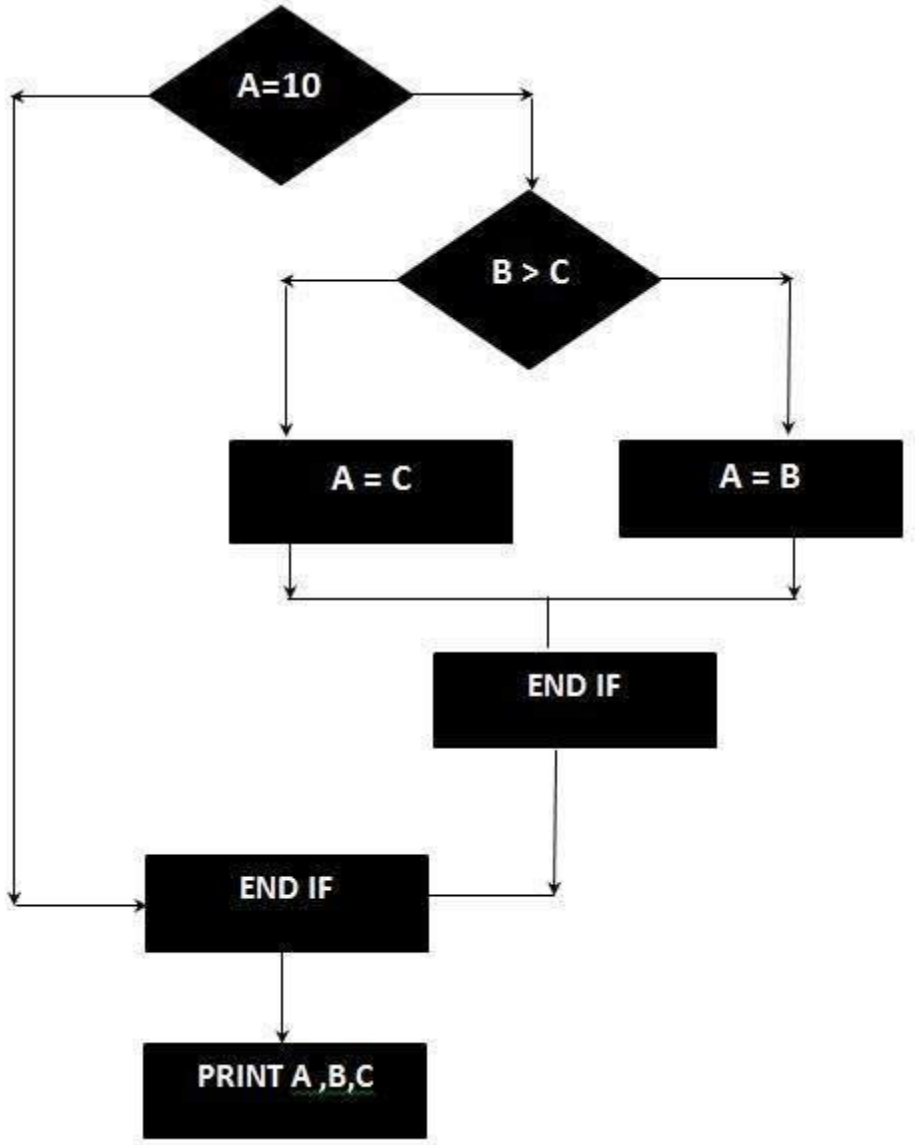
Steps to compute complexity

- Draw the control flow graph of the code
- Count the number of edges = E
- Count the number of nodes = N
- Count the number of connected components = P
- Complexity = E – N + 2P
- Normally in SE , P=1, so formula is often reduced to
- E-N+2
- What is the complexity for the graph?



E=7, N=6, P=1,
V(G)=7-6+2(1)=3

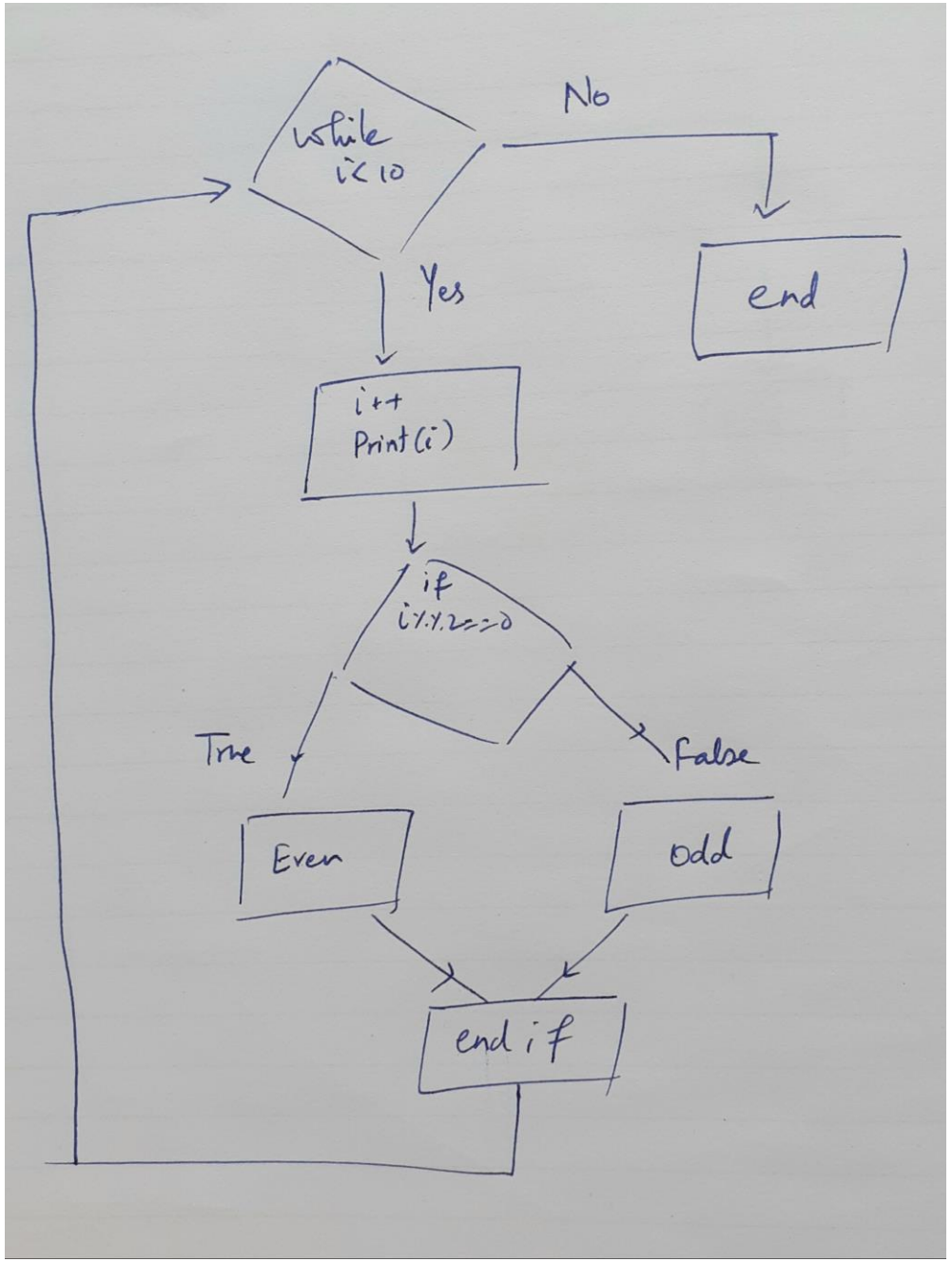
```
IF A = 10 THEN
    IF B > C THEN
    A = B
    ELSE
    A = C
    END IF
END IF
Print A Print B Print C
```



E=8, N=7, P=1,
V(G)=8-7+2(1)=3

```
public void howComplex() {
        int i=20;

        while (i<10) {
            i++;
            System.out.printf("i is %d", i);
            if (i%2 == 0) {
                System.out.println("even");
            } else {
                System.out.println("odd");
            }
        }
    }
```



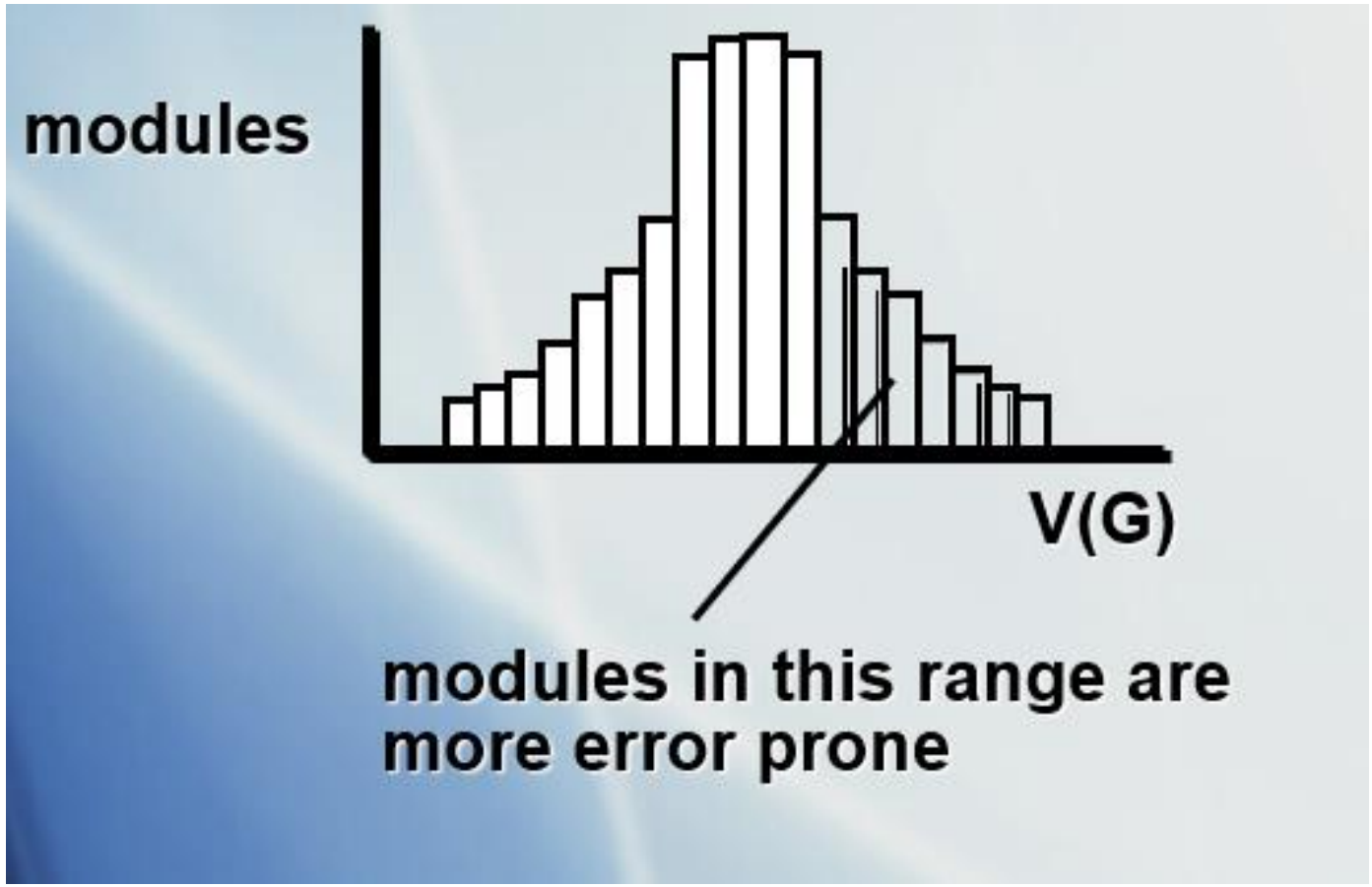
Edges = 8
Nodes = 7
P=1

V(G)=8-7+2(1)=3

OR

Number of decision making points = 2
V(G)=2+1=3

A number of industrial studies have indicated that higher the V(G), higher the probability of errors.



Nesting revisited..

- Generally two or three layers of nesting is considered *ok*.
- Any more than that means adding complexity.
- Nested loops are hard to debug.

```php
<?php
function login ($email, $password, $new_password, $confirm_new_password)
{
   if($email && $password && $new_password && $confirm_new_password) {
      if($new_password == $confirm_new_password) {
         if(login($email, $password)) {
            if(set_password($email, $new_password)) {
               return TRUE;
            }
         }
      }
   }
}
```

Example is not deeply messy, may be because you are familiar with what is happening in this case, but it conveys the idea of badly nested loop.

Careless coding always has a threat of assuming a shaped of tangled code.

To avoid issues with nested looping, you can refactor, restructure and reform Your code.

```php
<?php

function login ($email, $password, $new_password, $confirm_new_password)
{
    if (! $email || ! $password || ! $new_password || ! $confirm_new_password)
        return false;

    if($new_password != $confirm_new_password) return false;

    if(! set_password($email, $new_password)) return false;

    return true;
}
```

Previous code untangled.
Each logic has its own linear thread.
The code breaks out if the condition is not satisfied, else moves on to the next thread.
The above code *break the deeply nested conditionals to a set of single conditions*.
This code is a lot easier to read are maintain then it's ugly counterpart

Code complexity best practices

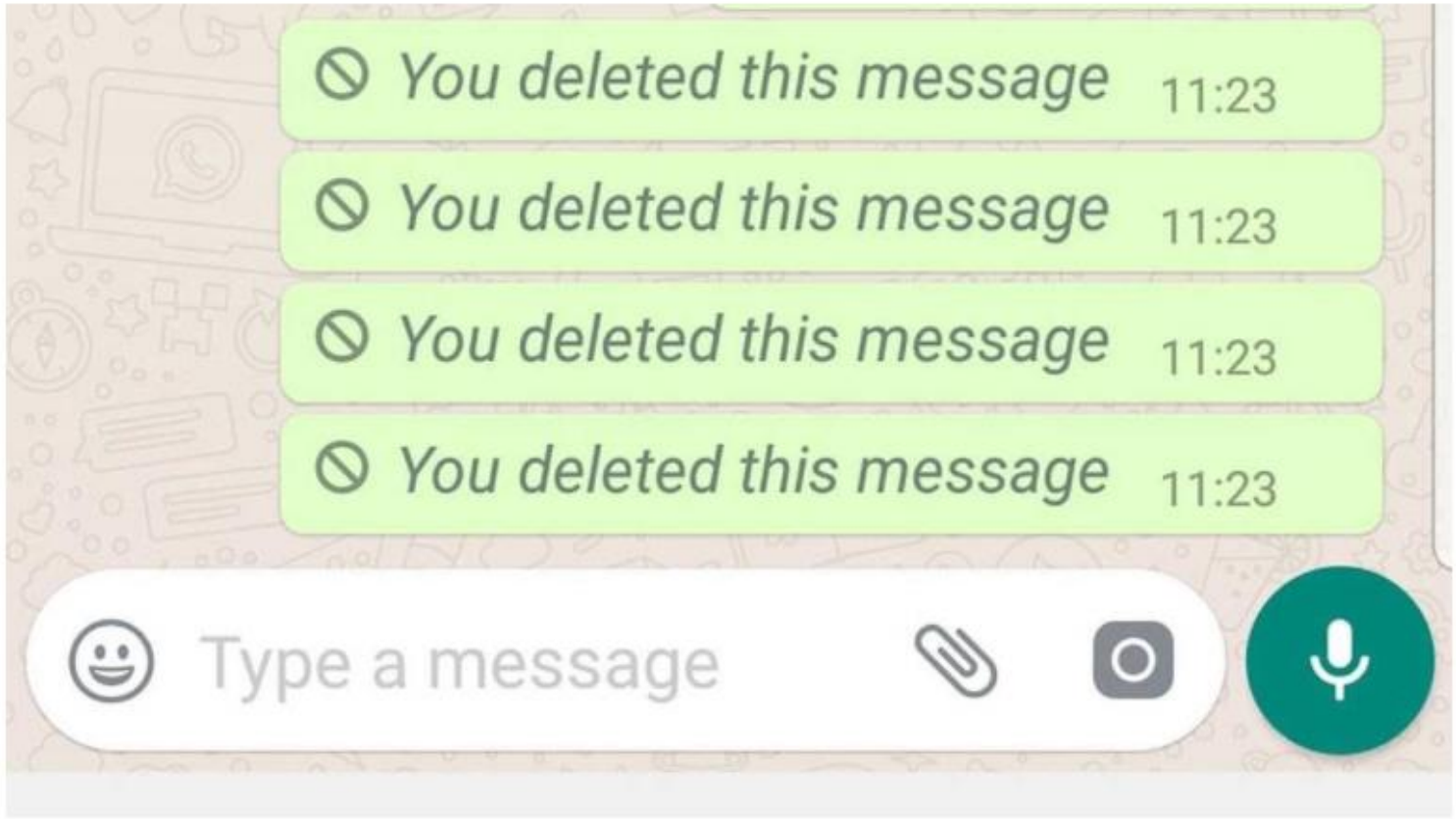
- If you module is too complex, it is time to break it up.
- Focus on worst offenders & break pieces of logic out into helper functions.
- The point of this is to enable good peer review and good unit test.
- Watch out for “if” statements nested 2 or 3 levels deep.
- Avoid excessive “break” , “continue”, nested “if” and “switch” statements.

Human Factors

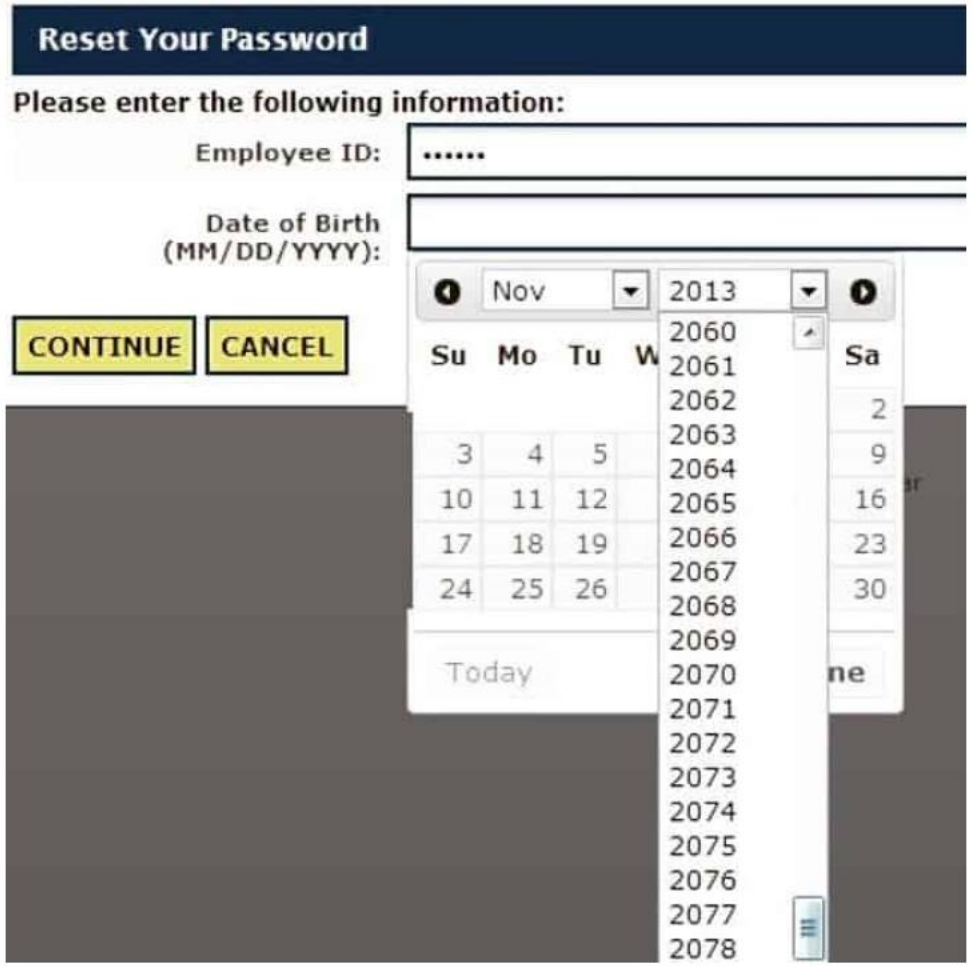
- Proper care must be given in developing the human-to-computer interface.
- Example: an input screen that is hard to read, has more fields than a person can easily comprehend, is not laid out properly, and is not robust with regard to human error will be rejected by the user.
- This effect will be an economic failure of the software product.

WhatsApp delete message feature

Informing the recipient that the sender has deleted a message somewhat defeats the purpose of deleting it in the first place.



What is funny here?



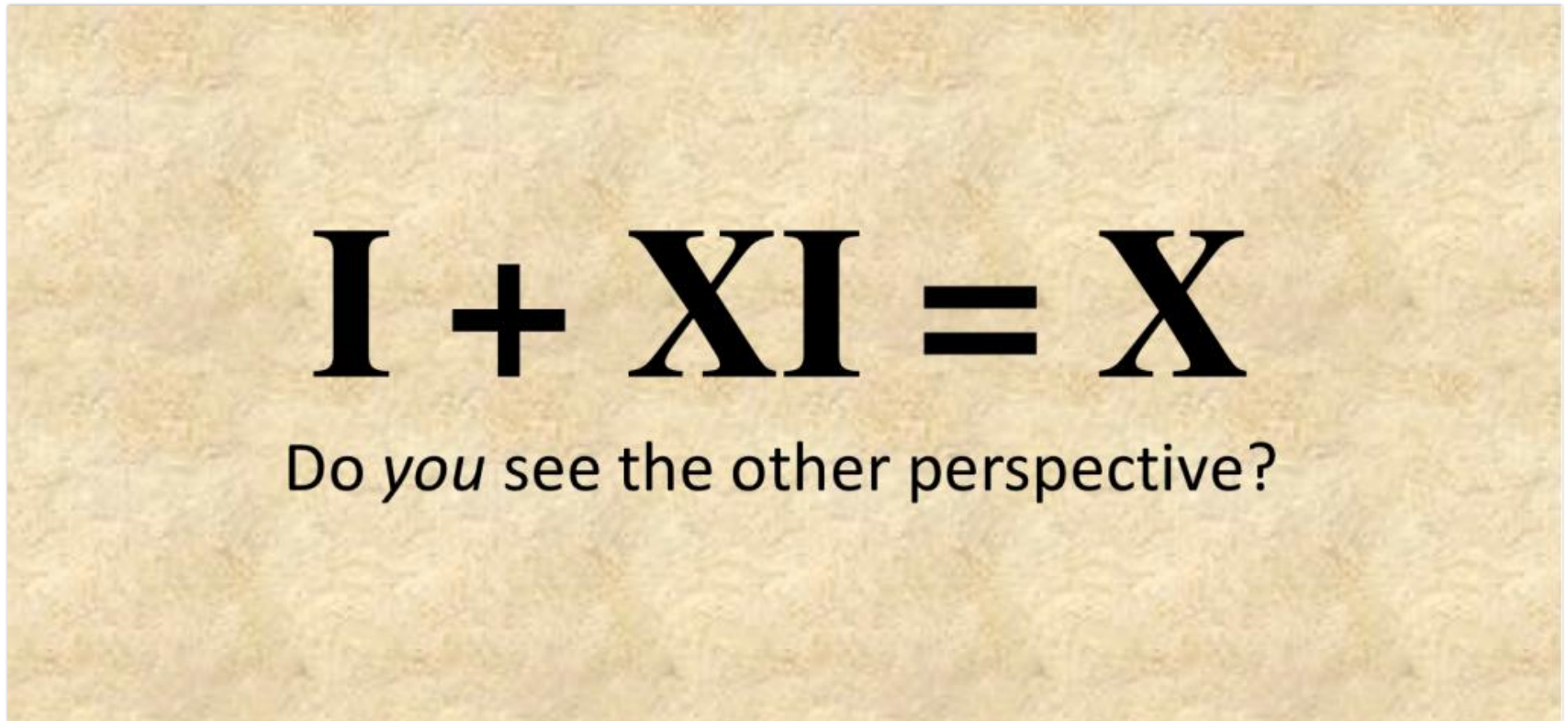
Here?

Video link

Reusable Code

- Segments of code are available in the libraries or market.
- In order to reduce the cost of the software product and make it available in the shortest possible time these segments are used in different parts of the program.
- Hence, the reusability attribute of a program segment could be a measure of segment quality.

Let’s look at the other side….



The other side of reusability

- Reusing a bad code.
- If you repeat yourself a few times evaluate if your code would be made simpler or more complicated by trying to make them reusable.
- Instead of making something simple complicated for the sake of reusability, keep it simple for the sake of sanity. Trade-off concerns.
- The maintenance and integration cost is high

Strongly Recommended article on reusability

Lastly,

- Remember that code isn't telling a computer what to do: it's telling another developer what it's instructing the computer to do.
- To know what simple code reads like and feels like (yes, feels like), you need to go read a lot of really simple code.
- If you're not embarrassed by your old code then you aren't progressing as a programmer.


THIS IS THE WORST SPAGHETTI CODE I'VE EVER SEEN.
THE COMMENTS ARE USELESS, THERE'S VARIABLES THAT DO NOTHING...
WHAT DEPRAVED MADMAN WROTE THIS?
WAIT, I DID THIS TWO WEEKS AGO.

END